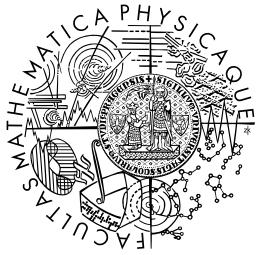


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jakub Břečka

A Decompiler for Objective-C

Department of Software Engineering

Supervisor of the master thesis:	RNDr. Jakub Yaghob, Ph.D.
Study programme:	Computer Science
Study branch:	Software Engineering

Prague 2016

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: A Decompiler for Objective-C

Author: Jakub Břečka

Department: Department of Software Engineering

Supervisor: RNDr. Jakub Yaghob, Ph.D., Department of Software Engineering

Abstract:

Objective-C is a popular programming language primarily used on the OS X and iOS platforms. We present a practical approach to decompilation of programs written in Objective-C and compiled for the x86 and AArch64 architectures using LLVM. Based on already-known generic reverse engineering techniques and compiler theory, this thesis analyzes new challenges and opportunities that occur in Objective-C binaries. We then offer solutions and algorithms that allow a decompiler to better recognize the high-level structures commonly used in Objective-C source codes. The thesis introduces an implementation of a new decompiler called "Cricket", an interactive GUI application for OS X, which uses the described algorithms and pattern matching methods to reconstruct source code in Objective-C. The decompiler tries to maximize readability of the output and allows user interaction to further modify the generated source code. The implemented software is then evaluated on a popular open-source framework and the results are compared to a competing product.

Keywords: Objective-C, decompiler, program analysis

I would like to thank my wife-to-be, Michaela, for her endless support, and my supervisor, Jakub Yaghob, for his patience, advice and help with this thesis.

Contents

Introduction

In recent years, reverse engineering has become a widely popular branch of software engineering. Main uses of reverse engineering include security analysis of closed-source products or the need to debug code without source code. While **disassembling** an unknown binary certainly can be hard and no current disassembler produces a 100% perfect result every time, when we deal with compiler-generated (and not deliberately obfuscated) code, the problem is considered solved, or at least solved for the vast majority of common cases.

Decompilation, on the other hand, is still a problem that resists being solved in a general way, even for relatively simple programs. Some high-level programming languages, for example *Java*, do not compile to machine code and are, therefore, significantly easier to decompile, because the binary preserves a lot of information about functions, variables and types. That is not the case for C and C++, which leave very few traces for a decompiler to use.

Another programming language that is similar to C and C++, is **Objective-C** (Obj-C for short), which has become very popular recently due to the rise of the iOS and OS X platforms. Since Objective-C is a strict superset of C, all the difficulties in decompilation of C are inherited as well, at least in the general case. There are, however, specifics of the Objective-C language that cause more information about the original source code to be preserved. These could either be directly used or more easily reconstructed by a heuristic analysis. This includes the *dynamic nature* of the language, its runtime, and the fact that common Obj-C source codes typically does not resemble pure C, but uses high-level language constructs heavily.

In this thesis, we will be analyzing how the general problem of decompilation changes when we focus on the Objective-C language, the OS X and iOS platforms and the AArch64 and x86 architectures (these are the only supported CPU architectures on OS X and iOS). The explicit goal of this thesis is to create a tool for decompiling binary Objective-C programs compiled for these architectures. The tool will be used primarily as an aid in manual malware analysis and implementation of security software – the ultimate motivation is to allow an independent researcher to analyze and verify an implementation of a closed-source system, for example to search for a hidden backdoor in a security-critical product.

The thesis is divided into several chapters. The first chapter, **Background**, provides the necessary overview of reverse engineering, disassembling and decompilation, currently available tools and techniques and discusses how well they work. The second chapter, **The Objective-C Language and Runtime**, deals with the external and internal features of Objective-C and how they could help decompilation. The next chapter, **Low-level Reverse Engineering Techniques** describes what tasks are needed to build a low-level part of a reverse-engineering

tool. The fourth chapter, **High-level Code Reconstruction**, describes the major decompilation challenges and what individual problems must be solved to implement a decompiler. The next chapter, **Objective-C Specific Decompilation**, focuses on the Objective-C language from the point of view of the decompiler and describes what features can help decompilation as well as what new problems we face. The sixth chapter, **The "Cricket" Objective-C Decompiler**, introduces our actual implementation of an Objective-C decompilation tool and describes the major architecture and design points. The seventh chapter, **Evaluation**, compares our expectations to the actual results of our decompiler. The last chapter, **Conclusion**, evaluates how well the set goals were achieved and discusses possible future work.

Chapter 1

Background

This chapter focuses on providing the reader with necessary background to understand what is the current state of the reverse engineering industry, what goals we are trying to achieve and what is possible and what is not.

In this chapter and in the whole thesis, we will only consider languages, compilers, and tools that work on and compile to physical CPU instructions. This is how most software written in C, C++ and Objective-C is distributed. We will not describe scripting languages that are interpreted or languages that distribute their code in bytecode format (e.g. Java).

1.1 Purposes of software reverse engineering

Many software vendors ship their products in compiled code only. End users usually do not need to have the source code of the software they are using, and binary distributions protect companies' intellectual property, because they do not reveal how the software works.

However, sometimes it would be really useful to have the source code of a closed-source software product, especially for developers and researchers:

- A developer working on an application that uses a closed-source library can run into trouble, because some aspects of the library's API might not be well documented. Having access to the source code can immediately reveal what the library expects.
- All software has bugs. Without access to the source code, debugging is extremely hard.
- When a closed-source library's code executes in the same address space as your program's code, a memory corruption can cause a crash *in the library code*. Even when the bug is not in the library, having source code would help diagnose that the corruption comes from elsewhere.
- Security and privacy related software often makes promises about their behavior, for example that it will not have backdoors, it will store the data in an encrypted way only, or that it will not leak user's data to the vendor's servers. Verifying these properties in closed-source programs is extremely difficult.

- Some platforms impose a limitation on what APIs are allowed to be used, even when this is technically not enforced. The source code would immediately show what APIs are called.
- Malware needs to be analyzed by security researchers to correctly assess its impact and level of threat. This task, which is often manual, would be greatly simplified and sped up when done on source-code level rather than on the level of machine instructions.

Furthermore, there are valid use cases when even with the source code of a program, it is beneficial to analyze the binary form using reverse engineering techniques:

- Compiler engineers often need to analyze the compiled output of a compiler to verify that the optimizations used are correct.
- Designing a software protection scheme (DRM, copy protection, etc.) requires knowing what the attacker can and will do.

The area of *defending against reverse engineering* is out of scope of this thesis, but it is covered in great detail in [1], which describes various techniques from *code obfuscation*, *tamperproofing*, *software watermarking*, *plagiarism detection*, *birthmarking* and *hardware-based protection*.

For our purposes, we will only focus on compiler-generated code that has not been intentionally obfuscated.

1.2 Existing disassemblers

One of the most commonly used tools from the toolset of every reverse engineer is a **disassembler**, which processes the executable binary file containing compiled machine instructions and provides the user with a human-readable form of these instructions. The instruction listing is provided in *opcode mnemonics*, with each instruction typically on a separate line. Some platforms provide a standard disassembler tool, like `objdump` on Linux or `otool` on OS X. An output of such command-line disassembler might look like the following:

```
$ objdump -M intel -d /lib64/libreadline.so.6
...
0000000000019c70 <rl_vi_put>:
   19c70: 41 54                   push   r12
   19c72: 55                      push   rbp
   19c73: 89 f5                   mov    ebp,esi
   19c75: 53                      push   rbx
   19c76: 89 fb                   mov    ebx,edi
   19c78: e8 b3 c8 ff ff          call   16530 <__ctype_b_loc@plt>
   19c7d: 48 8b 00                mov    rax,QWORD PTR [rax]
   19c80: 48 63 d5                movsxd rdx,ebp
   19c83: f6 44 50 01 01          test   BYTE PTR [rax+rdx*2+0x1],0x1
   19c88: 75 16                   jne    19ca0 <rl_vi_put+0x30>
   19c8a: 4c 8b 25 3f 81 22 00    mov    r12,QWORD PTR [rip+0x22813f]
                                  # 241dd0 <_rl_possible_control_prefixes+0x1650>
   19c91: 48 8b 05 48 7e 22 00    mov    rax,QWORD PTR [rip+0x227e48]
                                  # 241ae0 <_rl_possible_control_prefixes+0x1360>
...
```

This shows the list of instructions from the function called `rl_vi_put` in the libreadline shared library, which is compiled for the x86-64 architecture. The complete reference and manual for the instruction set of x86-64 is available on Intel's website as *Intel® 64 and IA-32 Architectures Software Developer Manuals* [2], which also contains a detailed description of the architecture itself. For assembly listings, we will always use the so-called "Intel syntax" (which is also used in Intel reference manuals).

The standard command-line disassembler tools provide a quick way to get a complete disassembly of a binary, but they have many limitations, do not provide any non-trivial analysis and they are unsuitable for a more serious reverse engineering work. *IDA – Interactive Disassembler* (formerly named *IDA Pro*) is a commercial product that is an **interactive disassembler**, but it also offers a full reverse engineering IDE [3].

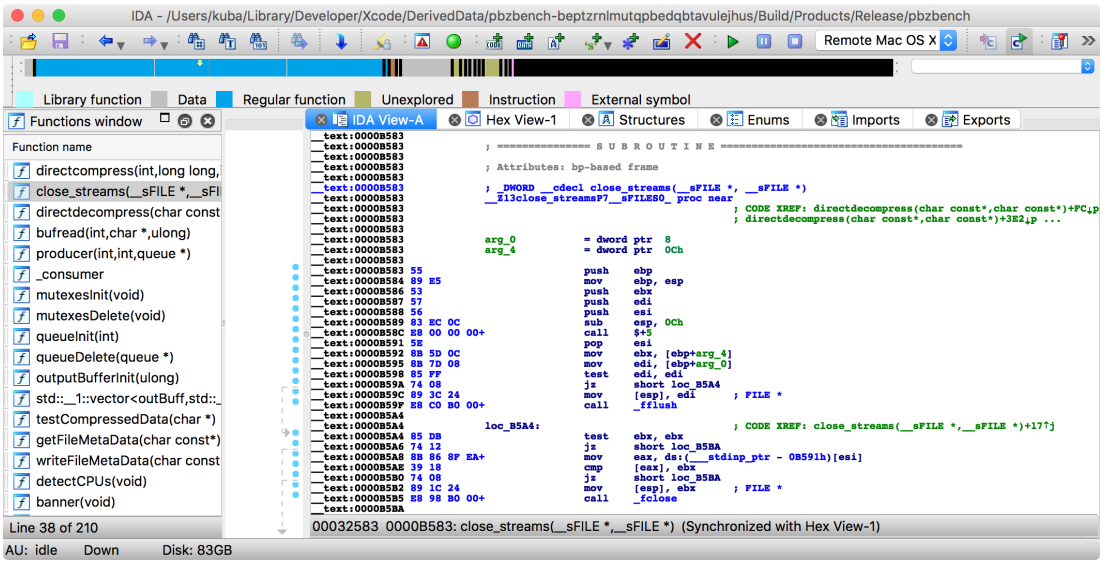


Figure 1.1: IDA by Hex-Rays, an interactive disassembler

It has some very advanced features and it performs various analyses on the disassembled programs to discover individual functions, global variables, stack variables, function parameters and their types. It understands a lot of idioms that are commonly used on the platforms it supports and decodes them. One of the most useful features is the ability to get a list of *cross-references*, which shows all the places (both code and data) where a symbol is referenced from. This allows the engineer, for example, to quickly find all callers of a function, or to find which code uses a particular string constant.

Another powerful feature is the ability to view graphs of basic blocks. This visualizes the control flow of a function gives a good overview of how complex the function is.

IDA supports dozens of processor architectures and platforms, and due to its long history and success, it is well known among reverse engineers.

Hopper is a newer interactive disassembler that focuses on x86 and ARM architectures [4]. It shares many of the same features with IDA, but it comes at a much more available price.

Both IDA and Hopper are end-user products, but another important set of

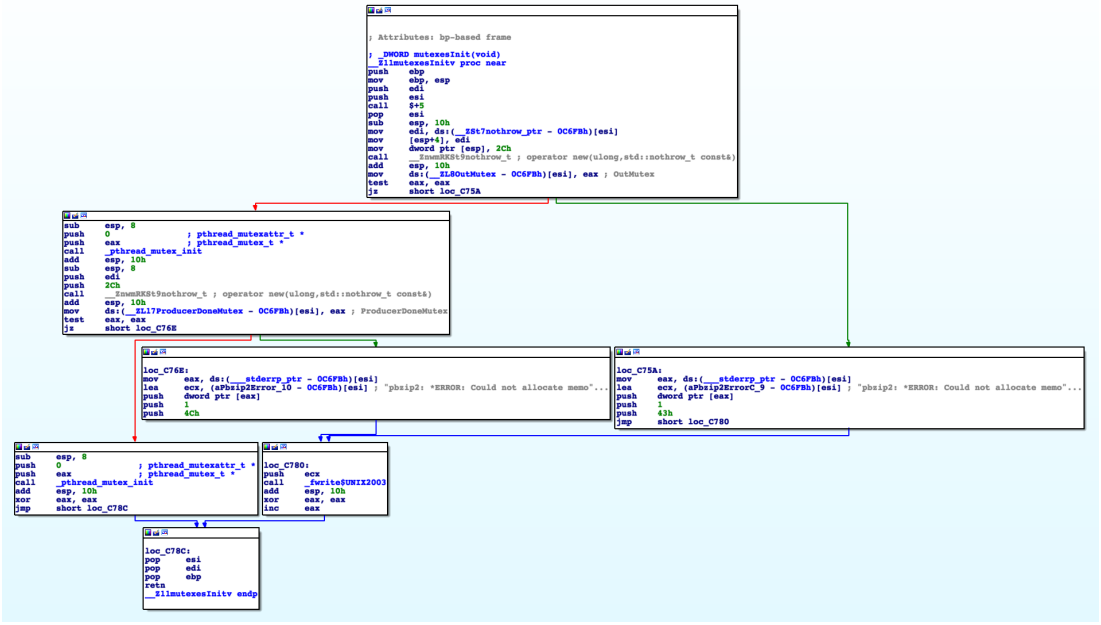


Figure 1.2: A graph of basic blocks in IDA

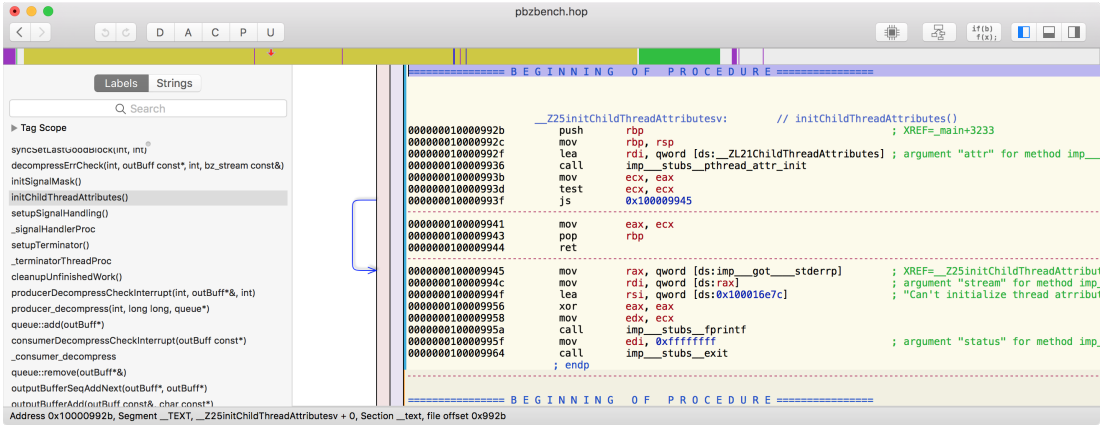


Figure 1.3: Hopper

tools are **disassembler libraries**, which allow an API-level access to assembly instructions decoded from a binary. *Capstone* [5] is a modern disassembly framework with support for multiple processor architectures and it provides very detailed information about individual disassembled instructions, including some of the instruction semantics (e.g. implicitly used register). Various binary analysis tools have been built on top of Capstone, because of its reliability and easy-to-use API. The following code sample shows how to disassemble a byte stream of x86-64 code.

```
from capstone import *

CODE = b"\x55\x48\x8b\x05\xb8\x13\x00\x00"

md = Cs(CS_ARCH_X86, CS_MODE_64)
for i in md.disasm(CODE, 0x1000):
    print("0x%x:\t%s\t%s" %(i.address, i.mnemonic, i.op_str))
```

This list of disassembler products is by no means complete. We are only providing a highlight of some of the available tools that are related to our goals. There are other disassemblers that are more popular on other platforms, like *OllyDbg* [6] on Windows, or open-source tools, like *x64dbg* [7].

1.3 Existing decompilers

Although there are many disassemblers available on the market, useful **decompilers** seem to be very rare. This seems to be caused by the fact that disassembling a binary is a very complex but still a straightforward task and the output can be either correct or wrong. Decompilation, on the other hand, is by its nature a heuristic task, it tries to reconstruct information that needs to be guessed, which is, in some cases, even impossible. The most famous academic publication on decompilation is a work called *Reverse Compilation Techniques* by C. Cifuentes [8], which describes a plethora of algorithms that can reconstruct various pieces of high-level information from the assembly code.

Both the interactive disassemblers mentioned in the previous section, IDA and Hopper, come with a decompiler (the IDA decompiler, *Hex-Rays Decompiler* [9], is actually a separate product by the same company). IDA decompiler does not state any explicit Objective-C support and due to its overall unavailability (pricing) it will not be further considered.

Hopper, on the other hand, explicitly supports decompiling Objective-C code. It features a non-interactive mechanism that produces a *high-level pseudo-code*, which highly resembles Objective-C source code. The main downside of Hopper is that the decompiler only supports the x86 architecture, and not the other major Objective-C architectures (ARM and AArch64).

The quality of the decompiled outputs from Hopper varies greatly. While the decompiler works very well on unoptimized (debug) builds, working on fully optimized binaries often produces pseudo-code that identifies function arguments incorrectly or fails to recognize the control flow of a function.

The following includes an example source code and Hopper's reconstructed pseudo-code:

```
// Original source code:
- (void)appendHTTPBodyPart:(AFHTTPBodyPart *)bodyPart {
```

```
    [self.HTTPBodyParts addObject:bodyPart];
}

// Output from Hopper 3.11.17 on an optimized x86-64 build:
void -[AFMultipartBodyStream appendHTTPBodyPart:](void * self, void * _cmd,
                                                  void * arg2) {
    objc_storeStrong(var_18, arg2);
    rax = [self HTTPBodyParts];
    rax = [rax retain];
    var_20 = rax;
    [rax addObject:0x0];
    [var_20 release];
    objc_storeStrong(0x0, 0x0);
    return;
}
```

The decompiled output detected method calls, but their arguments are wrong (`addObject` is called with `0x0` as a parameter). However, in general, Hopper gives a decent overview of a function in pseudo-code, and it will later be used as a competitor product for a comparison with our implementation of an Objective-C specific decompiler.

Chapter 2

The Objective-C Language and Runtime

Objective-C is a programming language with a long history. However, in recent years, it has gained a lot of popularity [13], because it is used as a primary language when developing applications for the OS X and iOS platforms. This chapter presents an overview of the language with its features, syntax and structure.

The language has several implementations that support different features. Historically, *GCC*'s implementation was officially supported for many years, until *Clang* [10], the front-end for the *LLVM* compiler [11], became production quality. Ultimately, with the release of Xcode 5 in 2013 [12], Clang became the only supported Objective-C compiler for both OS X and iOS.

There is no standard that would be an authoritative source of all language features. Instead, the most recent released version of Clang is used as the *de facto standard*. This also means that the language changes quite often and Apple, being one of the major development companies behind LLVM/Clang, traditionally announces new language features almost every year. Additionally, it caused GCC's support of Objective-C to be delayed, and it seems that some newer language features (e.g. *blocks*) are not going to be implemented in GCC at all.

2.1 Syntax of Objective-C

This section provides a short introduction to the syntax of Objective-C, but it is not meant to completely describe it exhaustively. For a complete overview of the language, see [14] and [15].

Objective-C is a strict superset of C, meaning that any valid C program is also a valid Objective-C program. Similarly to C, source code files consist of header files (`*.h`) and implementation files (`*.m`, a short for *messages*), and they are used together with preprocessor directives that include the headers into source files. The C preprocessor is fully available to the programmer and it has no special behavior for Objective-C.

Assuming that the syntax of C and C++ is well-known, let us take a look at what additional syntax is provided by Objective-C, most of which adds a thin-layer support for object-oriented programming. Obj-C has an unusual syntax (at least for a C programmer) of defining instance and class methods. Here are a few examples of method definitions and their equivalents in C++:

```
// Objective-C:
@interface MyClass
- (void)shutdown { ... }
- (int)randomNumberUpTo:(int)maximalNumber { ... }
- (int)divideNumber:(int)numerator byNumber:(int)denominator { ... }
+ (id)singletonInstance { ... }
@end

// C++:
class MyClass {
public:
  void shutdown() { ... }
  int randomNumberUpTo(int maximalNumber) { ... }
  int divideNumberByNumber(int numerator, int denominator) { ... }
  static MyClass *singletonInstance() { ... }
};
```

An Obj-C method definition starts either with a minus sign indicating an instance method, or a plus sign indicating a class (static) method. Next, the parentheses contain the return type, which can be either any C type or an Obj-C class. This is followed by a method name that *contains named arguments divided by colons and spaces*. The actual name of the third method is `divideNumber:byNumber:`. The parameters have their types in parentheses again, and then the implementation of the function is enclosed in curly braces.

The name of the method, such as `divideNumber:byNumber:` is called a **selector**. The colons in the selector are important as they indicate the number and position of function arguments.

A type representing *any Obj-C object* is marked by the `id` keyword and it can be effectively used as an ultimate superclass of all objects. It is used extensively for example in container implementations: Arrays and dictionaries take their values as `id`, as any Obj-C object can be stored in these containers.

Let us take a look at a few examples of *calling* a method (again with the C++ equivalents):

```
// Objective-C:
[self shutdown];
int x = [self randomNumberUpTo:42];
int result = [self divideNumber:42 byNumber:6];
id instance = [MyClass singletonInstance];
id o = [[NSObject alloc] init];

// C++:
this->shutdown();
int x = this->randomNumberUpTo(42);
int result = this->divideNumberByNumber(42, 6);
MyClass *instance = MyClass::singletonInstance();
void *o = NSObject::alloc().init();
```

A method call in Obj-C is usually called **message sending**, and individual objects can be **receivers** of messages. Each call is wrapped in brackets and contains the *receiver* (object on which the method is called), a space and a *method name* with arguments inserted into the name after each colon. If the method has multiple parameters, spaces are inserted into the name of the method to separate them. In the first three cases, we use `self` as the receiver, which refers to the same object that we are currently in. The fourth example calls a **class method**

(static method), where the receiver is the *class object itself*. The last example shows a **nested call**, where the result of `[NSObject alloc]` is used as a receiver for the call to the `init` method.

Note that all of this can be still mixed with C code, and it is not unusual to do that. For example the probably most common function, `NSLog`, is a C function that can print Obj-C objects into the console output:

```
NSLog(@"The object is: %@", [MyClass singletonInstance]);
```

It is a `printf`-like variadic function and its first parameter is a **format string**, which accepts most `printf` format specifiers. The `%@` specifier represents any valid Obj-C object. Another important thing to notice in this function call is that the format string in quotes is prepended with an extra `@` sign. This denotes that the string constant is not a C string, but instead an instance of the `NSString` class (an Objective-C string). The language has several other basic data types that can be constructed with `@`:

```
@"Hello, world!"    // NSString
@42                 // NSNumber with an integer value
@3.14               // NSNumber with a floating point value
@YES                // NSNumber with a boolean value
@[a, b]             // NSArray, ‘a‘ and ‘b‘ must be other Obj-C objects
@{key: value}       // NSDictionary, ‘key‘ and ‘value‘ are Obj-C objects
@(expr)             // "boxing", NSString or NSValue, based on the type of ‘expr‘
```

There are several other language syntax elements:

- The `BOOL` type is a standard boolean type with values `YES` or `NO`.
- When dealing with objects, `nil` is the keyword equivalent to the `NULL` value in C.
- Inside a method implementation, `self` refers to the current object, `super` refers to the superclass object.
- `id` is the superclass of all objects.

2.1.1 Object-oriented programming

The *objective* part of Objective-C is designed as a thin layer in the compiler front-end, which implements the necessary parsing of declarations and definitions of classes, methods and properties. A typical class definition and implementation in Obj-C is usually split into two files. A header file, for example `Cat.h`, contains the interface of the class:

```
// Cat.h
@interface Cat : Animal                    // class name and superclass

@property NSString *name;                  // a string property "name"
@property int numberOfLives;               // an int property "numberOfLives"

- (id)initWithName:(NSString *)name;      // an initializer method
- (void)jump;                             // a plain instance method

@end
```

The class has a **name** (`Cat`), a **superclass** (`Animal`), **properties** (`name` and `numberOfLives`), and **methods**. Furthermore, a class can have **instance variables** (*ivars*). Defining a property automatically generates a *hidden instance variable*, which is used as a backing store for the value of the variable. The `initWithName:` method is an **initializer**.

An implementation file, for example `Cat.m`, contains the source code of the defined methods:

```
// Cat.m
@implementation Cat

- (id)initWithName:(NSString *)name {
  self = [super init];
  if (self) {
    self.name = name;
    self.numberOfLives = 9;  // a cat has 9 lives, initially
  }
  return self;
}

- (void)jump {
  NSLog(@"%@ has jumped", self.name);
  if (rand() % 6 == 0) {
    // this jump did not work out as planned
    self.numberOfLives = self.numberOfLives - 1;
  }
}

@end
```

This shows how an initializer is usually structured: It calls a superclass' initializer and if it succeeds, it performs its own additional initialization. The example also shows that a function body can mix ordinary C code structures (*if* statements, `rand()` function call, etc.) with Objective-C syntax (assigning a property value via the "`.`" operator).

2.1.2 Control-flow structures

Objective-C inherits classic C statements for most control-flow: **If** and **if-else** statements, **for**, **while** and **do-while** loops, the ternary operator (`condition ? expr_true : expr_false`), **switch** statements and the **break** and **continue** keywords (and even **goto**) work exactly the same as in C programs:

```
for (int i = 0; i < 10; i++) { ... }
if (expression) {
  ...
} else {
  ...
}
while (condition) {
  ...
}
```

In addition to that, Objective-C contains a collection-controlled loop statement, called a **for-each** loop (or **for-in** loop), which iterates over all elements in an Objective-C container:

```
- (void)printAllElementsFromStringArray:(NSArray *)array {
  for (NSString *element in array) {
    NSLog(@"%@", element);
  }
}
```

2.1.3 Blocks

Blocks (or **dispatch blocks**) are an Objective-C language extension, which implements *closures* capturing the lexical context from the place where they are defined, very similar to C++11 *lambda functions*. Blocks are defined by using the caret symbol (^) and arguments in parentheses, for example:

```
// Objective-C:
dispatch_block_t my_block = ^() { NSLog(@"hello from block"); };

// C++:
auto my_lambda = [] { NSLog(@"hello from C++ lambda"); }
```

The `dispatch_block_t` type represents a block with no arguments and a `void` return type, but in fact it is just a *typedef* for the real block type syntax, which is similar to C function pointers:

```
typedef void (^dispatch_block_t)(void);  // no-input void-returning block
typedef void (*func_ptr_t)(void);        // no-input void-returning C function
```

For a full comparison of blocks with C++ lambda functions, see [16].

Blocks can have different arguments and return types. Invoking a block is done by using parentheses on the variable that contains a reference to the block:

```
typedef void (^int_arg_block_t)(int);
typedef int (^int_returning_block_t)(void);

int_arg_block_t b = ^(int a) { NSLog(@"argument = %d", a); };
int_returning_block_t b2 = ^() { return 0x29a; };

// invoking blocks:
b(10);
int output = b2();
```

Perhaps the most interesting feature of a block is that it can **capture variables**. Variables can be captured **by value**, which is the value at the moment of the creation of the block. This means that any subsequent modifications do not affect the value inside the block, and the block cannot modify that variable either:

```
int a = 10;
dispatch_block_t b = ^() {
  NSLog(@"a = %d", a);
  a = 42;  // this does not affect the outer-scope "a" variable
};
b();
```

By using the `__block` modifier on a variable, it will be captured **by reference** and the lifetime of the variable will also be promoted to be tied to the lifetime of the capturing block:

```
__block int a = 10;
dispatch_block_t b = ^() {
  NSLog(@"a = %d", a);
  a = 42;  // this *does* change the value of "a"
};
b();
NSLog(@"a = %d", a);  // prints 42
```

The concept of blocks and variable capturing becomes very convenient in combination with asynchronous programming. **Grand Central Dispatch** (GCD) is a standard library that works with blocks, and for example the `dispatch_async` function will schedule a block to be executed on a different *queue* (e.g. in a background thread):

```
MyClass *object = ...;
dispatch_async(backgroundQueue, ^{
  // this is run in a background thread:
  [object performHeavyCalculations];

  dispatch_async(dispatch_get_main_queue(), ^{
    // this is run in the main thread again:
    [object redraw];
  });
});
```

2.2 Standard libraries

Objective-C as a language contains very few classes. An external standard library called **Foundation** implements basic data types (e.g. NSString, NSNumber) and containers (e.g. NSArray, NSDictionary), but also advanced types and classes (e.g. NSLock, NSOperationQueue, NSFileManager) that combined together provide programmers with a rich library to build their programs on. A large part of Foundation is written in Objective-C and the interfaces are provided as Obj-C classes, but a lot of the features are actually implemented in a lower-level C library called **CoreFoundation**. While the division is not strict, CoreFoundation generally provides C APIs that Foundation builds upon and wraps them in a nicer object-oriented layer.

These two libraries are integrated into a much larger OS X umbrella framework called **Cocoa** (and **Cocoa Touch** for iOS). The description of these libraries and their features is out of scope for this thesis.

2.3 Objective-C runtime

The compiler front-end of Objective-C depends on and works closely with the language runtime, which is open-sourced by Apple at [17]. The runtime provides implementations for several features of the language, such as *dynamic dispatch*, *method caches*, *reference counting*, *weak references*, *monitors*, the `NSObject` class or the *introspection API*.

2.3.1 Dynamic dispatch

One of the main tasks of the runtime is to dispatch method calls to the method implementations. This is due to Objective-C being a **dynamically dispatched language**, where each method call is only resolved at runtime.

In contrast, a C function call is typically compiled into a `CALL` instruction, which simply transfers control flow directly into the code of the callee. The address of the callee is known at compile time and directly embedded into the `CALL` instruction. However, when an Objective-C method is called, the address of the callee is not computed at compile-time. Instead, a general dispatcher called `objc_msgSend` is invoked, which resolves the *selector* into a function implementation pointer and transfers control to it:

```
// Method call in Objective-C:
id object = ...;
[object methodWithNumber:42];

// The compiler actually rewrites the call into:
id object = ...;
SEL selector = @selector(methodWithNumber:);
objc_msgSend(object, selector, 42);
```

A *selector* is a *string* (a `char *` in C) name of the method, but there is one more requirement to make a `char *` a valid selector: It has to be *uniqued* in the whole process space, so that the dynamic message dispatch can construct method caches from the pointer values of selectors. In Obj-C, we can use the `@selector(methodWithNumber:)` syntax for that and the compiler will perform the *unification* during compile time.

2.3.2 Reference counting

Objective-C objects maintain a **reference count**, which is increased by a call to `objc_retain` and decreased by `objc_release`. Most modern Objective-C code uses a technology called **automatic reference counting** (ARC), in which the compiler itself emits calls to these functions whenever a reference to an object is stored or erased. This means that users do not have to manually maintain the reference counts and objects are automatically deallocated when no references to them exist anymore.

A situation, when two (or more) objects reference each other, is called a **retain cycle**. Such objects cannot be deallocated even when no external references to them exist, causing a *memory leak*. To avoid this, **zeroing weak references** exist, which hold a reference to an object without increasing its reference count. When the object is deallocated, all of its weak references are zeroed. This is useful for example in tree structures, where a parent holds a strong reference to its children, but the children contain only a weak reference to the parent:

```
@interface BinaryTreeNode

@property (strong) BinaryTreeNode *leftChild;
@property (strong) BinaryTreeNode *rightChild;
@property (weak) BinaryTreeNode *parent;

@end
```

In this case, erasing a reference to a child automatically deallocates the whole subtree, and there is no need to explicitly unset the `parent` references.

2.3.3 Public Objective-C runtime API

The publicly available API that the runtime provides is documented in Apple's documentation at [18]. It is a C API that is available to a programmer, but the compiler uses it as well and generates calls to these APIs. The following code lists a few examples of the API:

```
// returns the name of a class
const char *class_getName(Class cls);

// creates an instance of the class
id class_createInstance(Class cls, size_t extraBytes);

// sets the value of an ivar in an object.
void object_setIvar(id obj, Ivar ivar, id value);

// registers a new selector
SEL sel_registerName(const char *str);
```

The runtime API provides various introspection of objects, as well as registering new classes at runtime or manipulating with methods (e.g. replacing a method with another implementation, known as **method swizzling**).

Chapter 3

Low-level Reverse Engineering Techniques

Traditional software development process consists of manually writing *source code* and using tools (such as a compiler) to generate *software products*:

- The **source code** is a high-level representation of the program. It is **human-readable**, understandable and it is the primary form of the program. The source code is written by a developer and it is stored on the developer's computer.
- The **product** is what is later delivered to the users and the format of it varies greatly for different programming languages and environments. When dealing with C and other *natively compiled* languages, this is usually one or more **binary executables or libraries**, which contain code in the form of architecture-specific instructions.

When generating the software product, several tools are used to transform the source code into a binary, such as a **compiler**, **assembler** and **linker**. The *code* is being transformed from the source form into several different representations during this process, until it becomes the final binary. Usually, each representation is more low-level than the previous one. This is shown in figure 3.1. **Reverse engineering** tried to reverse this process and the transformations performed to reconstruct higher-level representations of code.

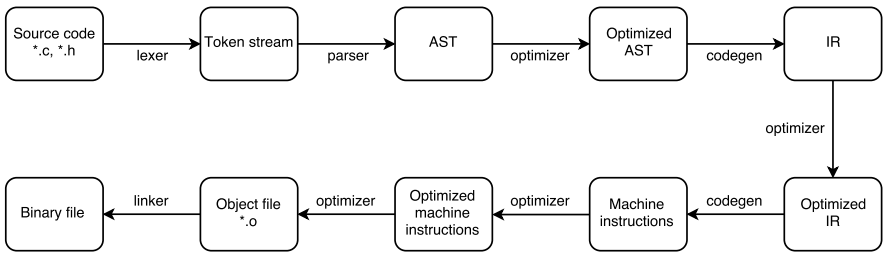


Figure 3.1: General stages of traditional compilation.

This chapter provides an overview of the transformation that a compiler performs, analyzes what information is lost or preserved in each stage, and discusses how can reverse engineering can recover lost information.

3.1 Traditional compilation transformations

Let us demonstrate an example of a very simple program and what representations it might go through when being compiled. Here is the **source code** in C:

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("Hello, %s\n", argv[0]);
  return 0;
}
```

When the compiler starts translating this program, it will convert it into a **token stream**. While doing so, it will also **preprocess** the source code, resolving all preprocessor macros and directives. If we for now ignore the inclusion of the `stdio.h` system header file, the token stream can look like this:

```
int 'int', identifier 'main', l_paren '(', int 'int', identifier 'argc',
comma ',', char 'char', star '*', identifier 'argv', l_square '[', r_square ']',
r_paren ')', l_brace '{'

identifier 'printf', l_paren '(', string_literal '"Hello, %s\n"', comma ',',
identifier 'argv', l_square '[', numeric_constant '0', r_square ']',
r_paren ')', semi ';',

return 'return', numeric_constant '0', semi ';',

r_brace '}'

eof ''
```

In this representation, some information from the original source is already lost. All comments, whitespace, and formatting styles were removed, because they do not affect how the program is compiled. Macros and preprocessor directives are lost as well. Modern compilers actually make extra effort to track this information (by saving source locations for each token or tracking macro expansions), so they can use them for example when printing an error message.

Next, the tokens will be parsed to form an **abstract syntax tree** (AST), which is the tree representation of the source. The following lists the AST in a text form, and a visual form is shown in figure 3.2.

```
`-FunctionDecl ... main 'int (int, char **)'
  |-ParmVarDecl ... argc 'int'
  |-ParmVarDecl ... argv 'char **':'char **'
  `-CompoundStmt
    |-CallExpr ... 'int'
    | |-ImplicitCastExpr ... 'int (*)(const char *, ...)'
    | | `-DeclRefExpr ... Function 'printf' 'int (const char *, ...)'
    | |-ImplicitCastExpr ... 'const char *'
    | | `-ImplicitCastExpr ... 'char *'
    | |   `-StringLiteral ... 'char [11]' "Hello, %s\n"
    | `-ImplicitCastExpr ... 'char *'
    |   `-ArraySubscriptExpr ... 'char *'
    |     |-ImplicitCastExpr ... 'char **'
    |     | `-DeclRefExpr ParmVar 'argv' 'char **'
    |     `-IntegerLiteral ... 0
    `-ReturnStmt
      `-IntegerLiteral ... 'int' 0
```

The AST representation is still very close to the original source code. Some information is lost, for example types are usually turned into canonical forms: Declaring a variable as `unsigned int` would be indistinguishable from just `unsigned`.

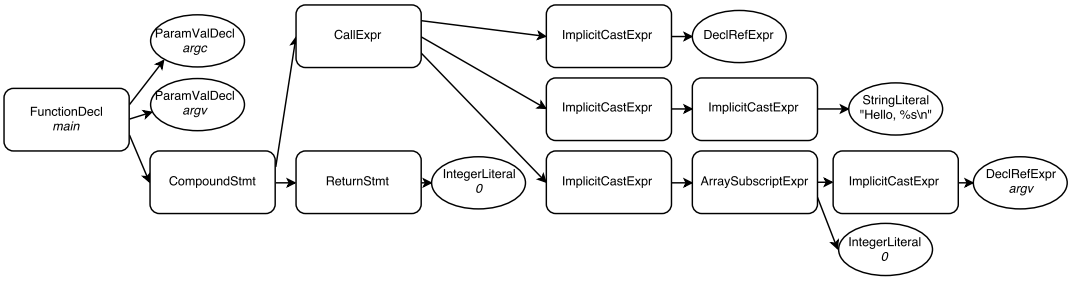


Figure 3.2: AST visualization.

When the code is present in the AST form in the compiler front-end, several optimizations can be done. In Clang, however, a lot of optimizing techniques are not used (e.g. constant folding) on the AST level, both because Clang wants to be able to map the AST very closely to the original source code and also because the LLVM back-end will perform the optimization anyway (and maybe in a better way).

The AST is then transformed into an **intermediate representation** (IR). For Clang this is the **LLVM IR**, which is a well-defined language that consists of code in low-level instructions, but it also knows about high-level concepts, such as functions and modules.

```
@.str = private unnamed_addr constant [11 x i8] c"Hello, %s\0A\00", align 1

define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  store i32 0, i32* %retval
  store i32 %argc, i32* %argc.addr, align 4
  store i8** %argv, i8*** %argv.addr, align 8
  %0 = load i8*** %argv.addr, align 8
  %arrayidx = getelementptr inbounds i8** %0, i64 0
  %1 = load i8** %arrayidx, align 8
  %call = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
          ([11 x i8]* @.str, i32 0, i32 0), i8* %1)
  ret i32 0
}
```

This is a major transformation, because the resulting code does not resemble the original source code anymore. The instructions now represent very simple steps, e.g. a single memory access, a function call or a single addition, etc. Any non-trivial expression in the source code is likely to be expressed with several LLVM instructions. LLVM also depends on **static single assignment** (SSA) form and provides **infinite number of virtual registers**. This means that the code can use any number of local variables (as virtual registers; their names start with the percent symbol), but each can be assigned only once.

LLVM can do many optimizations that operate on the IR level and the output is in the same format. It should be obvious that the shown code is sub-optimal: It contains several unnecessary operations and variables, e.g. the `%retval` variable is allocated in main memory (via the `alloca` instruction), then a zero is stored into it, and then it is never used again. An optimized version of the same function could look like this:

```
@.str = private unnamed_addr constant [11 x i8] c"Hello, %s\0A\00", align 1

define i32 @main(i32 %argc, i8** nocapture readonly %argv) #0 {
entry:
  %0 = load i8** %argv, align 8, !tbaa !1
  %call = tail call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
          ([11 x i8]* @.str, i64 0, i64 0), i8* %0) #1
  ret i32 0
}
```

Back-end optimizing transformations can lose all sorts of information about the code that they work on. They can move instructions, remove code or variables, or even completely change control flow and structure of a function. In our case, the optimization *actually helped us* and the function now looks more like the original code. We can easily tell that the function performs a load from memory, then calls `printf` and returns zero.

Eventually, the back-end needs to abandon the target-independent LLVM IR form and lower the instructions into **machine instructions**. This transformation is complex and involves several phases including instruction selection, scheduling and several more optimizations. The machine instructions typically still have several levels, and before *register allocation* they operate on virtual registers:

```
%vreg1<def> = COPY %RSI
%vreg2<def> = MOV64rm %vreg1, 1, %noreg, 0, %noreg
ADJCALLSTACKDOWN64 0, %RSP<imp-def,dead>, %EFLAGS<imp-def,dead>, %RSP<imp-use>
%vreg3<def> = LEA64r %RIP, 1, %noreg, <ga:@.str>, %noreg
%vreg4<def> = MOV32r0 %EFLAGS<imp-def,dead>
%vreg5<def> = COPY %vreg4:sub_8bit
%RDI<def> = COPY %vreg3
%RSI<def> = COPY %vreg2
%AL<def> = COPY %vreg5
CALL64pcrel32 <ga:@printf>, <regmask>, %RSP<imp-use>, %RDI<imp-use>,
              %RSI<imp-use>, %AL<imp-use>, %RSP<imp-def>, %EAX<imp-def>
ADJCALLSTACKUP64 0, %RSP<imp-def,dead>, %EFLAGS<imp-def,dead>, %RSP<imp-use>
%vreg6<def> = COPY %EAX
%EAX<def> = COPY %vreg4
RETQ %EAX
```

This representation is again fundamentally different from what we saw in the previous steps. We have completely lost almost all type information and the registers that we operate on have their own sizes. A lot of machine-specific ABI and **calling convention** rules are involved; actually, most of the instructions that we see are just to satisfy the platform's ABI. Notice that while *somewhere* we are already using *actual instructions* (`LEA64r`), there is still a lot of meta-instructions, e.g. `COPY` or `ADJCALLSTACKDOWN64`. The next phases of the code generation will get rid of these:

```
PUSH64r %RBP, %RSP, %RSP           ; flags: FrameSetup
%RBP = MOV64rr %RSP                ; flags: FrameSetup
%RSI = MOV64rm %RSI, 1, 0
%RDI = LEA64r %RIP, 1, <ga:@.str>
%EAX = XOR32rr %EAX, %EAX
CALL64pcrel32 <ga:@printf>
%EAX = XOR32rr %EAX, %EAX
%RBP = POP64r %RSP, %RSP
RETQ %EAX
```

These are the final machine-level instructions, but still in the internal LLVM format. We lost some information when the meta-instructions, that were adjusting the top of the stack, are now turned into `PUSH64r` and `POP64r`. But more importantly, the *information about uses of registers* is also lost. While in the previous form, we knew that the function call uses the `%RDI` and `%RSI` registers, in the current form, **we can not tell which registers are actually used by the callee**. It might even look like the assignments to the `%RDI` and `%RSI` registers are not necessary, because they are not used anywhere else in the function. However, they *are used* by the `printf` function.

The next representation is an **assembly file**. Here is the code using *Intel syntax*:

```
_main:                                  ## @main
    PUSH    RBP
    MOV     RBP, RSP
    MOV     RSI, QWORD PTR [RSI]
    LEA     RDI, [RIP + L_.str]
    XOR     EAX, EAX
    CALL    _printf
    XOR     EAX, EAX
    POP     RBP
    RET

.section    __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
    .asciz    "Hello, %s\n"
```

This time, the transformation is pretty straightforward. We again lost some information when e.g. we are using the `MOV` mnemonic to represent both a register-to-register move (`MOV64rr` in the previous code) and a memory load (`MOV64rm`). However, this information is not really lost, but instead hidden in the Intel x86 specifications and manuals. We are also now much more explicit about how we generate and access the string constant: we instruct the linker to put it into a special section in the resulting binary.

We still use some meta-information in the code that needs to be resolved by the assembler and the linker. One of them is the way we refer to `L_.str` and `_printf`. These get only resolved when we know the actual locations of the code and data, and the code can look like this:

```
; Section __text
                _main:
0x100000f50         PUSH        RBP
0x100000f51         MOV         RBP, RSP
0x100000f54         MOV         RSI, QWORD PTR [RSI]
```

```
0x100000f57         LEA         RDI, QWORD PTR [0x100000f8a]
0x100000f5e         XOR         EAX, EAX
0x100000f60         CALL        0x100000f6a
0x100000f65         XOR         EAX, EAX
0x100000f67         POP         RBP
0x100000f68         RET

; Section __stubs
0x100000f6a         JMP         QWORD PTR [0x100001010]

; Section __cstring
0x100000f8a         DB          "Hello, %s\n", 0

; Section __la_symbol_ptr
0x100001010         DQ          ...
```

Notice that there are no more symbolic names at all, and everything is referred by an integer constant or offset. There is one more *final* form of this code and that is the actual **binary representation** inside the executable on the disk (here shown in hexadecimal listing):

```
55 48 89 E5 48 8B 36 48
8D 3D 2C 00 00 00 31 C0
E8 05 00 00 00 31 C0 5D
C3 90
```

The binary representation is actually very interesting, because *everything, including data and meta-data* is stored in the binary as well, and it is definitely not obvious what part of it is code, data or meta-data. One could say that this last step loses the most information so far, because if we open the executable file in a hex editor, we will probably not be able to make any sense of it at all.

However, this is exactly the form that our program is shipped in and how it is delivered to the users. Now let us take a look at what we can tell from such a binary.

3.2 Analyzing binaries

The task of a reverse engineer starts with a file on disk, typically a *binary executable*. The format of it is defined by the platform that this binary is supposed to run on, for example the most common Linux format is **Executable and Linkable Format** (ELF), and it is used not only for executables, but also for shared libraries and object files. We will focus on analyzing binaries for the *Darwin* platform (used on OS X), which uses the **Mach-O** file format and is described in [19].

Besides metadata and headers, a Mach-O file contains **load commands** which describe the content of the binary and its desired runtime environment in terms of **segments** and **sections**. A typical installation of OS X with a freely available developer tools (Xcode) contain several tools that can help us analyze Mach-O files. Having a binary to analyze, the `otool` (*object file displaying tool*) can show us these load commands and sections:

```
Load command 0                          Load command 1
      cmd LC_SEGMENT_64                       cmd LC_SEGMENT_64
```

```
  segname __PAGEZERO                    segname __TEXT
   vmaddr 0x0000000000000000             vmaddr 0x0000000100000000
   vmsize 0x0000000100000000             vmsize 0x0000000000001000
...

Section                               Section
  sectname __text                       sectname __cstring
   segname __TEXT                        segname __TEXT
      addr 0x0000000100000f50               addr 0x0000000100000f8a
      size 0x0000000000000019               size 0x000000000000000b
    offset 3920                           offset 3978
...
```

The load commands are instructions for the operation system's loader and they reveal important properties about the program. It is a 64-bit executable, and the first load command describes a `__PAGEZERO` segment, which should make the first 4 GiB of address space (starting at address `0x0`) to be inaccessible and any access to it should trigger a segmentation fault. The second command sets up a 4 KiB `__TEXT` segment starting at `0x100000000`, where code and data are present. The two sections, `__text` and `__cstring`, instruct the loader to *map* parts of the contents of the file to the respective virtual addresses. The `__text` section (which usually contains executable instructions) will be created from `0x19` bytes starting at offset 3920 in the file and mapped into the process address space starting at address `0x100000f50`; and similarly for the `__cstring` section (which contains string constants used by the program).

The naming and types of sections do not actually need to follow these rules, and *code obfuscation* and other software protection schemes will intentionally try to mislead us. However, binaries produced by standard compilers and linker will follow a lot of *customary* conventions, and we can deduce information from them. For example, the `__cstring` section will contain only NULL-terminated string literals and constants, and we would not expect code instructions to live in this section. We should even be able to dump a list of all strings used by a binary very easily by simply splitting the full contents of this section at each NULL terminator. The section description above conveniently shows the file offset and length of this data:

```
$ xxd -s 3978 -l 0xb a.out
0000f8a: 4865 6c6c 6f2c 2025 730a 00              Hello, %s..
```

The *code section* (named `__text`) can be assumed to contain raw machine instructions, and after we dump it in a similar way, we can use a *disassembler* to try to reproduce the *assembly listing*. We will look into that in one of the following sections, but for now, let us see what other sections and information we can gather from the binary.

3.3 Dynamic linking

After the kernel loads an executable into user space and sets up its segments and virtual memory, it is very common to first give control to the **dynamic linker**, which will load and link external libraries into the process' address space. The binary specifies which libraries it depends on, and what **external symbols** it

uses from them. Only after all references are resolved (which is a *recursive task*, because libraries can depend on other libraries), the actual initialization of the program starts. When talking about Mach-O and modern OS X, a segment called `__LINKEDIT` contains a blob of data that contains the necessary information for the dynamic linker called `dyld`. A third-party tool called `jtool` can show what all is contained in this segment:

```
$ jtool --pages a.out
...
0x2000-0x2108 __LINKEDIT
  0x2000-0x2008 Rebase Info     (opcodes)
  0x2008-0x2020 Binding Info    (opcodes)
  0x2020-0x2030 Lazy Bind Info  (opcodes)
  0x2030-0x2060 Exports
  0x2060-0x2068 Function Starts
  0x2068-0x2080 Code Signature DRS
  0x2068-0x2068 Data In Code
  0x2080-0x20c0 Symbol Table
  0x20c0-0x20d0 Indirect Symbol Table
  0x20d0-0x2108 String Table
```

Most of the contained data is in the form of structures that reference others, but the very last item in the list, "String Table" contains again a list of NULL-terminated strings that the other structures use:

```
$ xxd -s 0x20d0 -l 0xc8 a.out
00020d0: 2000 5f5f 6d68 5f65 7865 6375 7465 5f68   .__mh_execute_h
00020e0: 6561 6465 7200 5f6d 6169 6e00 5f70 7269  eader._main._pri
00020f0: 6e74 6600 6479 6c64 5f73 7475 625f 6269  ntf.dyld_stub_bi
0002100: 6e64 6572 0000 0000
```

We can immediately spot the name of the function in our source code, `_main` (prefixed with an underscore), and also the external function we are calling, `_printf`. Of course, there is a better way to view these **exported and imported symbols**:

```
$ nm -m a.out
100000000 (__TEXT,__text) [referenced dynamically] external __mh_execute_header
100000f30 (__TEXT,__text) external _main
          (undefined) external _printf (from libSystem)
          (undefined) external dyld_stub_binder (from libSystem)
```

It might be somewhat unexpected that we can see the `_main` symbol here, because it is an internal function in our source code and there is no need for this symbol to be externally visible or importable. The reason is that our binary is not **stripped**, which is typical for debug versions of binaries. We probably would not be able to see internal symbol names for release and production binaries, where the `strip` command is usually run on the final binary.

The imported function (`_printf`), however, **has to be visible** and its name has to be contained in the binary, because the dynamic linker uses the name to resolve and link this external symbol. Using `nm` on any dynamically linked program (i.e. almost all programs today) will reveal the symbol names and libraries of all externally linked functions and other symbols.

Based on how a binary is compiled (optimized vs. non-optimized, debug vs. release, stripped vs. non-stripped, etc.), more sections can be present, e.g. `__unwind_info` is used to store **compact unwind** information, `__eh_frame` is needed to support zero-cost **exception handling** in C++. Debugging data (source files and line information, variable and procedure descriptions, etc.) can be stored in another section.

3.4 Disassembly

Notice how much information we were able to retrieve *without ever seeing the actual instructions* in the code sections. To see the instructions, we can run a **disassembler** on the `__text` section:

```
$ otool -t a.out
0000000100000f50 55 48 89 e5 48 8b 36 48 8d 3d 2c 00 00 00 31 c0
0000000100000f60 e8 05 00 00 00 31 c0 5d c3
$ otool -t -v a.out
0000000100000f50  PUSH   RBP
0000000100000f51  MOV    RBP, RSP
0000000100000f54  MOV    RSI, QWORD PTR [RSI]
0000000100000f57  LEA    RDI, QWORD PTR [RIP + 0x2c]
0000000100000f5e  XOR    EAX, EAX
0000000100000f60  CALL   0x100000f6a
0000000100000f65  XOR    EAX, EAX
0000000100000f67  POP    RBP
0000000100000f68  RET
```

In our case, we were able to successfully and accurately disassemble the whole code section, and an important fact to notice is that even at first glance, some instructions appear to be of higher importance than others. The `CALL` and `RET` actually reveal some high-level structure of the function and the program. We should also be able to recognize the **function prologue and epilogue**, where the first two instructions (`PUSH RBP` and `MOV RBP, RSP`) form the standard prologue that sets up the **stack frame**. The last two instructions, `POP RBP` and `RET`, tear the stack frame down and return. We will deal with these later when we will discuss ABI and calling conventions.

We can safely tell that the whole listing is a single function, and another piece of high-level information that can be deduced is the use registers as inputs to the function. The `MOV RSI, QWORD PTR [RSI]` instruction *reads* the value of `RSI`, but the function never sets this value before. From this, we can infer that `RSI` in an input to the function.

Our sample is very simple and contains no branches, and all of the code of the function is in a single **basic block**. We will show more complex examples later in the next chapter.

3.4.1 Recognizing procedures

Even when we know the exact extent of the code section, disassembling all of the instructions correctly is a complex and error-prone task. Some architectures, for example x86, use variable-length instructions and wrongly assessing the beginning

of a function or mistranslating one instruction can lead to more errors during disassembly.

Another situation where the job of a disassembler is much harder is when code is mixed with data inside the same section. The compiler can do this for example for **jump tables** or **large literals**, where for both convenience and performance, the required data can be embedded between functions, or even between basic blocks. **Alignment padding** might also be evaluated as data, but compiler-generated padding usually uses `NOP` instructions (to mitigate damage when such code is accidentally executed).

For further processing, finding all procedures (and their sizes) is crucial. While correctly and precisely disassembling arbitrary x86 code is a theoretically hard problem [20], there are several heuristics and other indicators we can use:

- The function prologue and epilogue are commonly very distinguishable and their instruction sequences are unlikely to appear within function bodies.
- Some procedure beginnings are explicitly listed (e.g. exported functions).
- Data sections can contain pointers to beginnings of procedures, but they are very unlikely to contain pointers to the inside of a procedure.
- Objective-C metadata explicitly list all methods and pointers to the implementations.

A common technique used by disassemblers is to perform an initial scan of the whole code section, trying to disassemble all instructions, and if any byte sequence is not recognized, we just skip those bytes. Then we look for instructions, such as function calls, and for pointers in data sections, which will extend the **set of known function entry points**. Subsequently, we perform the disassembly again, starting from those known entry points, which can reveal more of them, so we iterate as long as we discover more functions. Whenever a new possible entry point is found, we heuristically analyze the first few instructions to see if they look like a known function prologue.

When we have a set of function entry points, we will cut the code section at these points. Each resulting region is one function. A post-processing step is necessary to detect possible mistakes, such as *padding* or *jump tables*, which are commonly present at the end of a function. This is often indicated by unrecognizable instructions. We will heuristically look for known prologues and jump-table formats to trim the function. Handling jump tables is described in the next chapter.

A possible mistake is that two (or more) following functions are recognized as one. The solution to this problem is to detect that in a later stage of decompilation – the basic block detection algorithm (described in the next chapter) will detect that the control-flow graph is not contiguous. We can then split the recognized function into two and re-run the disassembly for the newly discovered procedure.

3.5 Summary

We have described how a compiler transforms source code into a binary representation. Analyzing such a binary with existing standard tools can reveal useful metadata. We have shown a heuristic technique to discover function entry points

and their sizes. This will be used as an input in the next chapter, where we will reconstruct high-level code for an individual function.

Chapter 4

High-level Code Reconstruction

As we saw in the previous chapters, retrieving segments and sections, individual function boundaries, assembly instructions and other low-level information is achievable with a high degree of accuracy. In this chapter, we will try to work towards generating a high-level code assuming that we already know

- the extent of the function (starting and ending addresses),
- individual instructions that are correctly disassembled,
- the ABI, calling convention and general information about the architecture used, and
- addresses and signatures of other functions.

With this information, we will analyze and transform the instructions of a function into a pseudo-code that hides architecture-specific details and improves readability of the code.

4.1 Calling conventions and ABI

4.1.1 Function prologue and epilogue

Most modern architecture ABIs specify that functions should begin with a **prologue**, which has the purpose of setting up the stack frame for the actual execution of the function and that they should end with an **epilogue**, which tears the stack frame down and returns to the caller. For x86-64, one of the most common ABIs is the *System V Application Binary Interface* specification [21]. This document allows several actual implementations, but a typical function prologue consists of these instructions:

```
0000000100000f50  PUSH   RBP
0000000100000f51  MOV    RBP, RSP
```

This saves the `RBP` register by pushing it onto the stack, and then stores the current stack pointer (from the `RSP` register) as a new value into the `RBP` register, which serves are the **frame base pointer** throughout the execution of the whole function. This allows debuggers to always locate the start of the frame just by looking at the `RSP` register. A typical epilogue on x86-64 does the opposite of the prologue:

```
0000000100000f67  POP    RBP
0000000100000f68  RET
```

First, we restore the original value of `RBP`, which will now store the *previous* (caller's) frame base pointer, and as a very last instruction in the function, the `RET` instruction will transfer control to the calling function. If all the functions in the program follow this convention, `RBP` always points to a valid frame, but we can also always access the previous frame by looking into memory *one word below the frame*, which will contain the caller's saved frame pointer. This forms the runtime **call stack** chain and it is easy for a debugger to walk this list and display a **backtrace** of the current thread. The frame pointer serves another important purpose, since it can be also used to access local variables (located on the stack). This will be described later in this chapter.

In some cases, the prologue and epilogue can be different, or the frame pointer might not be set up in the prologue. If the `RBP` register is used to hold the frame pointer, it cannot be used as a general purpose register for the function execution. As an optimization, Clang and GCC have a `-fomit-frame-pointer` flag, which will allow using `RBP` as a regular register for register allocation, so fewer variables need to be spilled onto the stack. However, the function prologue still needs to save and restore the original `RBP` to satisfy the ABI requirements.

The prologue often allocates space on the stack for local variables:

```
0x100000ef0 <+0>:  PUSH   RBP
0x100000ef1 <+1>:  MOV    RBP, RSP
0x100000ef4 <+4>:  SUB    RSP, 0x10
```

The `SUB` instruction moves the stack pointer two words down, thus allocating 16 bytes on the stack. From this information alone, it is impossible to tell whether these are two 8-byte variables or a 16-byte integer (or another combination which totals 16 bytes). But the size of the stack frame is an important information that will be later used by various analyses. If a function allocates space on the stack in the prologue, it will also deallocate it in the epilogue, perhaps with a corresponding `ADD RSP, 0x10` instruction.

For the purposes of decompilation, the function prologue and epilogue are not very interesting. They are target-specific low-level concepts and we can completely skip analyzing the instructions from them. We are only interested in the fact that the function *has a frame* and the *size of the frame*.

4.1.2 Function arguments

The architecture ABI specifies how parameters are passed from the caller to the callee, which forms the **calling convention**. The System V ABI for x86-64 [21] says that in common cases, the caller is supposed to store the integer parameters in registers `RDI`, `RSI`, `RDX`, `RCX`, `R8` and `R9`. Floating-point values are stored in XMM registers (`XMM0`–`XMM7`). If there are more than six integer or more that 8 floating-point arguments, the remaining ones are stored on the stack. There are several exceptions (e.g. when variadic functions are used or when large structs are passed as arguments), and the precise specification is very complex, but the general idea is that arguments are passed through predefined registers and when they do not fit they are present on the stack. The caller is also supposed to clean

up the stack after the call has returned (if any arguments were stored on the stack). The callee accesses the stack parameters indirectly via the `RSP` (or `RBP`) register, by reading memory below its own function frame.

If we accept the fact that most functions have a low number of arguments (there are some statistical measurements of this, e.g. [22] claims 2.8 is the average, and [24] analyzed Windows DLLs with the outcome that most functions have 3 or fewer arguments, but the measurements always depend on what software is being analyzed), this means that most functions will only pass arguments in registers. A typical function call on x86-64 can then look like this:

```
0x100000f08 <+4>:  MOV    RDI, 0x1
0x100000f0d <+9>:  MOV    RSI, 0x2
0x100000f12 <+14>: MOV    RDX, 0x3
0x100000f17 <+19>: CALL   0x100000ee9
```

This stores three constant values into three registers. The `CALL` instruction then transfers control to the function at the specified address, while also saving the **return address** to the stack (which is used by the `RET` instruction to restore control back to the caller). The callee can directly access the three arguments in the `RDI`, `RSI` and `RDX` registers.

Analyzing such a simple situation is easy, but the content of the registers at the time of the call might not be so obvious, as assigning the register value does not necessarily need to happen right before the `CALL` instruction. A proper **data-flow analysis** is necessary for us to be able to deduce the register values, which will be described later.

The situation is simpler with input parameters. If we know the signature of the function, then the ABI directly tells us in which registers (or stack locations) are the arguments stored. If we do not know the signature, we can deduce that a register is used as an input argument when it is being read from without assigning it a value before:

```
0x100000f0c <+0>:  ...                  ; prologue skipped
0x100000f10 <+4>:  MOV    RAX, RDI      ; RDI is stored into RAX
0x100000f13 <+7>:  ADD    RAX, 0x2A
0x100000f17 <+11>: ...                  ; function continues
```

In this sample, the `RDI` register is obviously an input parameter. It is quite safe to say that this function has *at least one integer argument*.

If an integer value is passed through a register, but the integer type is smaller than the native register size (8 bytes on x86-64), then the argument is stored in the register anyway, but only a part of the register is considered valid data. For example, to pass a `char` (1 byte) argument through the `RDI` register, the callee can use the `DL` subregister (which is the lowest byte of `RDI`). This can also give our analyses a hint about the types of the parameter: If the function reads from the `DL` register, we can assume that the first parameter is 1 byte. If it reads from full `RDI`, the data is probably full-width (8 bytes).

By constructing the set of all register and stack locations are read, but not written to first, we can deduce the argument signature of the function.

4.1.3 Return values

If a function has a simple integer return value, it is returned via a register (`RAX` for x86-64). This is achieved by simply leaving a value in that register before returning from the function (via the `RET` instruction). The most basic example of a function that returns a constant value is:

```
0x100000f0c <+0>:  MOV    RAX, 0x29A     ; store the return value in RAX
0x100000f10 <+4>:  RET
```

Besides the `RET` instruction, this function only executes a single instruction to store the return value in `RAX`. It is also an example where the function *does not have a proper prologue and epilogue*, because it does not use any stack space and it does not even use any registers (except the return value register).

If we know the signature of a function, we can immediately tell which register we expect to hold the return value. However, if the function signature is unknown, we might need to make a heuristic guess. It might not be possible to distinguish between a situation, where `RAX` contains a return value and a situation, when it contains a leftover value from a previous computation that should be discarded. A simple heuristic can be used: If the last value written to `RAX` before returning is never used in other computation, it is more likely to be a return value. Otherwise, it would be a **dead store** and the compiler is likely to eliminate such unnecessary operation.

Additional ABI rules apply for more complex situations, e.g. when a larger integer is returned or when returning a floating-point value. One special case is worth mentioning: If the return value is a large structure, it is usually returned by writing it to a memory place. However, the callee does not have such an easily-available storage, because the stack must be left in the same state as it was at the beginning of the call. Therefore, the System V ABI specifies, that the caller is responsible for setting up this space, usually in its own stack frame, and passing an additional **hidden argument** containing the pointer to this memory. This hidden argument is prepended before the others.

```
large_struct function_returning_a_struct(int p) {
  ...
  return x;
}
```

Such a function in C/C++ is actually transformed by the compiler into:

```
void function_returning_a_struct(large_struct *tmp, int p) {
  ...
  *tmp = x;
  return;
}
```

For our purposes, we should note that it is impossible for the analysis to distinguish between these two variants. Unless we know the signature of the function beforehand, we again have to make a heuristic guess.

4.1.4 Callee-saved and scratch registers

Each function must also follow the specification of which registers can be used as **scratch registers** and which are **callee-saved**. Callee-saved registers must

be left in the same state as they were at the beginning of the call, which means that for the caller it looks as if they were not used and the values in them did not change. If the callee wants to use these registers, it must save the original values (usually on the stack), and restore them before transferring control back to the caller. Scratch registers can be used for any computation and they do not need to be restored.

We can use this knowledge when analyzing a `CALL` instruction to make assumptions about what registers are *clobbered* during the call. This will be important later during data-flow analysis.

4.1.5 Stack slots and local variables

The prologue allocates space for stack-based local variables, however not all user-defined local variables need to have a stack slot. The compiler usually tries to fit as many local variables into registers as possible, to avoid using main memory at all. Some variables might need to be **spilled** (promoted into a stack-based variable), but the compiler might also generate temporary and auxiliary local variables (e.g. to allow a called function to return a large structure). Sometimes it can completely eliminate local variables.

There are also cases where a user-defined local variable *must occupy a stack slot*, for example whenever we need to take an address of the variable (and perhaps pass it to some other function).

This means that it is not trivial to reconstruct local variables in a reasonable manner. Treating all stack slots as local variables can be both too excessive and insufficient. It is also not straightforward how can we get the number of stack slots, their sizes and data types. In the previous sections, we saw that the prologue usually shows the total size of the stack frame, but not the individual items, so a more complex analyses and heuristics are necessary.

On x86-64, the individual stack items can be accessed indirectly via the `RBP` or `RSP` registers. `RBP` points to the end of the stack frame, so a pointer to an item is obtained by subtracting a constant offset from `RBP`. Access via `RSP` is similar, but uses a positive offset, and it is also much less common because the stack pointer can sometimes change during a function execution so it is harder for the compiler to track the changing offsets. Such access (a load or a store) is usually done with a single instruction that calculates the correct pointer and performs the memory access in the same step. From the size of the access we can deduce the size of the stack item:

```
0x100000ef0 <+0>:  ...                                ; prologue skipped
0x100000ef4 <+4>:  SUB    RSP, 0x10
0x100000efd <+13>: MOV    BYTE PTR [RBP - 0x1], 0xA
0x100000f01 <+17>: MOV    BYTE PTR [RBP - 0x2], 0x14
0x100000f05 <+21>: MOV    QWORD PTR [RBP - 0x10], 0x1E
0x100000f0d <+29>: ...                                ; function continues
0x100000f12 <+34>: ADD    RSP, 0x10
0x100000f16 <+38>: ...                                ; epilogue skipped
```

This code allocates 16 bytes of stack storage (by moving the stack pointer 16 bytes down). The first stack memory access happens at the address of `RBP-1`, and this pointer is accessed as a *byte pointer*. The same happens for the stack item at `RBP-2`, but the third access happens as an 8-byte store at `RBP-16`. This suggests

that there are two 1-byte wide items and one 8-byte item on the stack. The bytes at `RBP-3` up to `RBP-8` are unused due to alignment.

Note that the compiler is unlikely to generate the code mentioned above, because it only stores constant values to the stack items, *unless we explicitly need the addresses of these stack items*, for example when passed to a `scanf(fmt, &var1)` call.

The x86-64 System V ABI also requires that a 128-byte **red zone** directly below the stack pointer can be used without moving the stack pointer. This is not too important for us, besides the fact that accessing a stack item *beyond* the stack pointer is possible.

A quite uncommon feature of C/C++ is **variably-sized arrays** and **alloca calls**, which allow allocating space on the stack of non-constant size. The compiler achieves this by adjusting the stack pointer by a dynamic offset, and such a variable is allocated below all the regular variables. Accesses to this variable are done via another register that stores a pointer to it. When the function returns, all of this space is deallocated by simply restoring the stack pointer to the saved value.

4.1.6 AArch64

The AArch64 [25] architecture share most of the concepts mentioned in the previous sections, but the implementations and the ABI is slightly different. On AArch64, parameters are passed via `R0`–`R7` registers; the `FP` register is used as a frame pointer.

The function calling mechanism is different from x86. The `BL` instruction calls a subroutine, but instead of pushing the return address onto the stack, it stores the return address into `LR`, the **link register**. Returning from a function is done via the `RET` instruction which reads a program address from the `LR` register and transfers control to it. This means that the `LR` register must be explicitly saved (pushed onto the stack) before a function can make calls, and restored before it can return.

An example function with a prologue, a function call and an epilogue might look like this:

```
0x100006214 <+0>:   STP    FP, LR, [SP, #-16]!
0x100006218 <+4>:   MOV    FP, SP
0x10000621c <+8>:   ...
0x10000622c <+24>:  BL     0x1000005A0              ; function call
0x100006230 <+28>:  ...
0x100006238 <+36>:  LDP    FP, LR, [SP], #16
0x10000623c <+40>:  RET
```

The first instruction of the function prologue (`STP`, *store pair*) saves the values of `FP` and `LR` to the stack, while also adjusting (pre-indexing) the stack pointer by 16 bytes. The second instruction then establishes the new frame pointer. The epilogue then restores `FP`, `LR`, and also `SP` (by adjusting it by 16 bytes up).

4.2 Control flow analysis

Only very simple functions execute all instructions linearly, more interesting functions have a more complex **control flow**. On the assembly level, most control flow is carried out by **branches**, **indirect branches** and **conditional branches**. These primitives partition a function into **basic blocks**, each of which is a (maximal) series of instructions that is always executed completely. This means that the first instruction of a basic block is the beginning of the function or a target of some of the branches. No other instruction of a basic block is a target of a branch. The last instruction of a basic block can be a branch, a return instruction or a non-control-flow instruction (which means the basic block performs a **fall through** at the end). In this definition we do not view function calls as branches, nor do we care about exception handling, signals, interrupts or other non-standard control flow.

4.2.1 Basic block detection

The first step in analyzing control flow is basic block detection, for which a "standard" algorithm exists, described e.g. in [23]. The algorithm constructs a set of **leaders**, which is a set of all branch targets within the function. The set initially contains only the first instruction in the function, but is extended to find all branch targets; then the function is split at each leader. We present an extended algorithm which also finds constructs successors and predecessors between the basic blocks.

1. Set *leaders* to be a set containing one element, which is the address of the first instruction in the function.
2. Add addresses of all conditional and unconditional branch targets into the set of *leaders*.
3. Add addresses of instructions immediately following all conditional branches into the set of *leaders*.
4. Split the instruction sequence at each *leader* in such a way that each *leader* starts a new basic block, which ends at the next *leader* or the end of the function.
5. For each basic block B, construct a set of successors *succ(B)* as follows:

   - If the basic block ends with an unconditional branch, *succ(B)* has a single single element, which is the block having branch target as the leader.
   - If the basic block ends with a conditional branch, *succ(B)* has two elements, one which is the block with the branch target as the leaders, and the block which immediately follows B.
   - If the basic block ends with a return instruction, *succ(B)* is empty.
   - If the basic block does not end with a branch or a return instruction, *succ(B)* contains the block that immediately follows B.

6. For each basic block B, construct a set of predecessors *pred(B)*, such as *succ(B)* contains C if and only if *pred(C)* contains B.

7. Mark the basic block which has the very first instruction of the function as the leaders as the *entry basic block*.

It should be easy to see that this algorithm finds all basic blocks and constructs a **control-flow graph** (directed edges are formed from the *succ(B)* sets). The algorithm however does not handle *indirect branches* (which are a common way of representing *switch statements*), because it assumes that each branch has a known target address and that the target is only one. We will work towards resolving this later in the chapter.

Let us take a look at an example of how this algorithm works on the following function (which implements the Euclidean algorithm to find the greatest common divisor) for x86-64:

```
gcd:
    0x100000e5a <+0>:  PUSH   RBP
    0x100000e5b <+1>:  MOV    RBP, RSP
    0x100000e5e <+4>:  MOV    RDX, RSI
    0x100000e61 <+7>:  MOV    RAX, RDI
    0x100000e64 <+10>: TEST   RDX, RDX
    0x100000e67 <+13>: JE     0x100000E7B               ; <+33>
    0x100000e69 <+15>: MOV    RCX, RDX
    0x100000e6c <+18>: XOR    EDX, EDX
    0x100000e6e <+20>: DIV    RCX
    0x100000e71 <+23>: TEST   RDX, RDX
    0x100000e74 <+26>: MOV    RAX, RCX
    0x100000e77 <+29>: JNE    0x100000E69               ; <+15>
    0x100000e79 <+31>: JMP    0x100000E7E               ; <+36>
    0x100000e7b <+33>: MOV    RCX, RAX
    0x100000e7e <+36>: MOV    RAX, RCX
    0x100000e81 <+39>: POP    RBP
    0x100000e82 <+40>: RET
```

Naturally, the <+0> offset becomes the first *leader*, and we will add all the branch targets (<+33>, <+15> and <+36>) to the set of *leaders* as well. Then we also add all offsets that are immediately after a conditional jump (<+15> and <+31>) to get the final set of leaders. When the function is split into basic blocks, it looks like this:

```
; --- basic block #1 --- successors = {#2, #4} --------------------------
    0x100000e5a <+0>:  PUSH   RBP
    0x100000e5b <+1>:  MOV    RBP, RSP
    0x100000e5e <+4>:  MOV    RDX, RSI
    0x100000e61 <+7>:  MOV    RAX, RDI
    0x100000e64 <+10>: TEST   RDX, RDX
    0x100000e67 <+13>: JE     0x100000E7B               ; <+33>
; --- basic block #2 --- successors = {#2, #3} --------------------------
    0x100000e69 <+15>: MOV    RCX, RDX
    0x100000e6c <+18>: XOR    EDX, EDX
    0x100000e6e <+20>: DIV    RCX
    0x100000e71 <+23>: TEST   RDX, RDX
    0x100000e74 <+26>: MOV    RAX, RCX
    0x100000e77 <+29>: JNE    0x100000E69               ; <+15>
; --- basic block #3 --- successors = {#5} ------------------------------
    0x100000e79 <+31>: JMP    0x100000E7E               ; <+36>
; --- basic block #4 --- successors = {#5} ------------------------------
    0x100000e7b <+33>: MOV    RCX, RAX
```

```
; --- basic block #5 --- successors = {} ------------------------------
    0x100000e7e <+36>: MOV    RAX, RCX
    0x100000e81 <+39>: POP    RBP
    0x100000e82 <+40>: RET
; ---------------------------------------------------------------------
```

Basic block `#2` has an interesting property that it has itself as one of its own successors. That is perfectly valid and indicates that the block forms a loop. The graph of basic blocks can be easily visualized, as shown in figure 4.1.

This function has a single **exit block** (a block with no successors ending with a `RET` instruction), but other functions can have more than one of those; in cases where the function *never exits*, it can even have no such basic blocks. On the other hand, each function must have exactly one *entry basic block*.

The basic block graph can already give us a lot of non-trivial information about the function. Loops in the graph are very likely to be loops in the original source code. While technically the graph can have almost any shape, it is very common that the graph has a linear path from the entry block to the exit block (if it has only one), and branches on this path only form detours that later merge back to the linear path. An interesting metric of a function is **cyclomatic complexity**, which is defined as the number of linearly independent paths through the function [26]. The higher this number, the more "complex" the structure of the function is control flow.

4.2.2 Inlining

One of the most common optimizations the compiler performs is **inlining**, which embeds the body of a called function into the caller, avoiding the overhead of a function call, stack frame setup and register saving and restoring. There are many rules and heuristics about inlining, which behaves differently in different compilers, under different optimization settings and even in different versions of the same compiler. The reason is that inlining can hurt performance or bloat code size. However, we can make some observations:

- Most compilers cannot inline functions defined in another translation unit (unless LTO is used, which is still uncommon).
- A function that is non-trivial and is called from multiple callers is less likely to be inlined.
- Trivial functions (returning a constant, not calling other functions, performing just some simple calculation on its parameters, etc.) are more likely to be inlined.
- Functions that set up a large stack frame are unlikely to be inlined.
- Any function that needs to be dynamically resolved cannot be inlined.

Analytically, it is impossible to perfectly detect that inlining happened. The inlined function body is not preserved as it is, but it is rather included in all subsequent optimizations, together with the caller's instructions.

We can either ignore inlining altogether, but if we want to try to detect inlined functions, such analysis will be much easier to be done at a later stage of decompilation. Once we are able to reconstruct the semantics and AST of several functions, we can try to involve *code similarity techniques*, which can reveal that

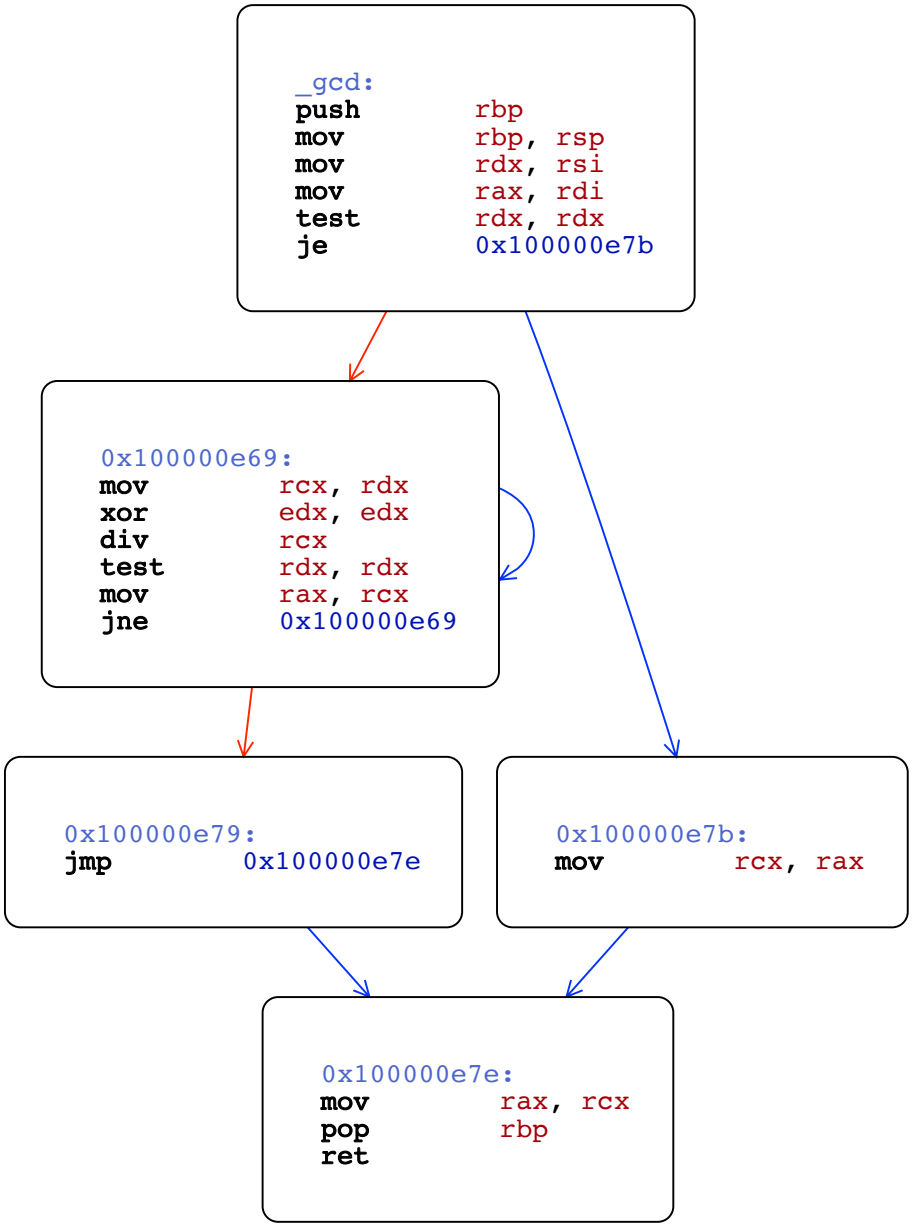


Figure 4.1: Graph of basic blocks (from the *Hopper* application)

multiple functions share a piece of code. Many of these techniques are described and evaluated in *A Comparison of Similarity Techniques for Detecting Source Code Plagiarism* [27].

4.2.3 Recognizing high-level control flow statements

If we have a graph of basic blocks, we could rewrite the complete control flow into a higher-level language directly using **goto statements**. Each basic block would get a **label** at the beginning and branches would become either unconditional or conditional goto statements. However, this is not how code is usually written; programmers write source code using **if statements**, **for loops** and **while loops**, so it is important for a decompiler to recognize these patterns and try to reconstruct the high-level language constructs.

We introduce a basic catalogue of control-flow subgraph patterns in figure 4.2. These patterns are sufficient to describe common structures that compilers generate for *if*, *while* and *for* statements.

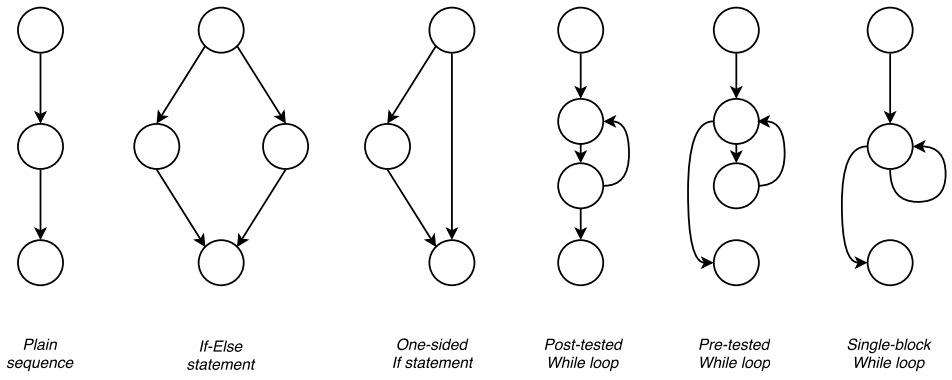


Figure 4.2: Basic control flow patterns

Our catalogue certainly does not describe all possible control-flow statements, but it will suffice to demonstrate our analyses. A much more complete pattern catalogue including many edge cases is presented in *Reverse Compilation Techniques* [8].

Analyzing a more complex control-flow graph then consists of finding these patterns in the graph and then **reducing the matched subgraph** into one entity, which represents the high-level language construct and embeds the matches graph nodes as AST-like subnodes. This step can allow further reductions to happen, and by repeating the pattern matching we can continue until we end up with a single-entity graph. An example of such matching is shown in figure 4.3. At the end of the control-flow analysis, we have a single node describing the function as an AST-like structure.

Simply finding matches from the pattern catalogue can often fail, because there simply will not be any subgraph that can be reduced. The reasons for that include:

- A loop is exited with a **break** or **continue** statement.

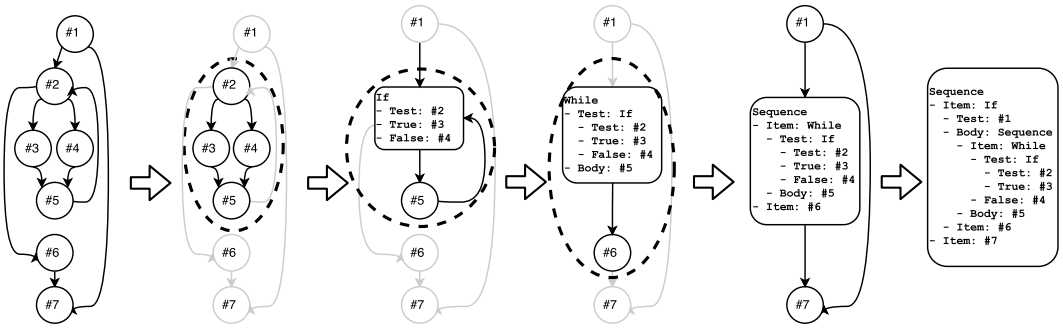


Figure 4.3: Example of pattern matching in a control-flow graph. First step matches an *if-else* statement, second step recognizes a *while* loop. Third step matches a sequence of blocks into a single graph node. Last step shows recognizing an *if* statement and a sequence.

- The catalogue is missing a specific control-flow structure.
- The compiler chose to generate an unusual control-flow edge, such as a jump into the middle of a loop. This can happen if two parts of a function contain the same or similar code and the compiler reuses one basic block for multiple code instances in the source code.
- The source code actually contained an unstructured jump (goto).

In a situation where we cannot match any pattern from the catalogue, we can *sacrifice an edge* from the control-flow graph and replace it with a *goto* statement. A special case of this operation can be done within a loop, to replace an edge with a *break* statement (in case the conditional edge points to the follow-up block of the loop), or *continue* statement (the edge points to the loop condition test node). Selection of which edge to sacrifice is inherently a heuristic decision, but there are various indicators that can help, and we should never remove an edge that will break the connectivity of the graph. **Backedges** (pointing towards numerically lower addresses) are uncommon and usually indicate ends of loops; if such an edge does not seem to belong to a particular loop, it is a good candidate for removal. Pattern matching can also try to detect cases where some simple structure matches, but there is an extra *entry* or *exit* edge. This can correspond to an actual *goto* in the source code (e.g. for bailing out of a function with a cleanup).

When the function has multiple exit basic blocks, we can create an artificial "final" exit block that would become a single point of function's exit, if that helps our pattern matching algorithm. An even more beneficial transformation, however, would be the opposite, because the compiler often prefers to have a single exit basic block, and *early returns* from the function result in multiple unrelated edges from various blocks into the exit node. In this case, **duplicating the exit node** (assuming it is small) will restore the original intent (early return).

Removing an edge from the control-flow graph or duplicating exit nodes creates new opportunities for catalogue-based pattern matching. If removing one edge does not unblock pattern matching, we will remove another edge. Eventually, the pattern matching must succeed, because, in worst case, the remaining graph will be a tree, which can always be matched. Control-flow analysis can be summarized

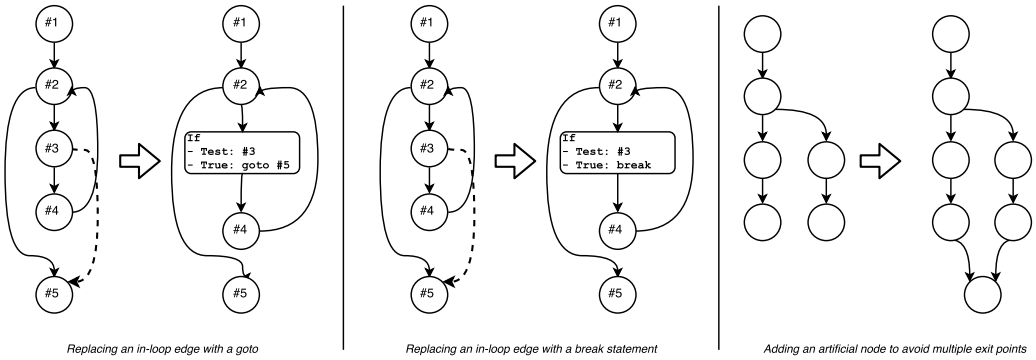


Figure 4.4: Examples of removing an extraneous in-loop edge, and dealing with a function with multiple exit points

in the following algorithm:

1. Initially, let G be a control-flow graph of basic blocks with edges corresponding to the successors of each basic block.
2. While the number of nodes in G is greater than one, do the following:

- If a subgraph that matches a catalogue pattern exists, then:
  - Replace the subgraph in G with a single new node based on the high-level statement that the pattern describes.
- Otherwise:
  - Heuristically choose an edge to sacrifice. Replace the edge and its origin node with a new node that describes a *goto* statement (or a *continue*/*break* statement if possible).

3. G now contains a single node which describes the function's control flow as a tree structure using high-level language constructs.

4.2.4 Indirect branches and switch statements

Switch statements are often compiled into an **indirect branch**, where the target address is a register value rather than a constant. This makes control-flow analysis harder, because we do not know what possible targets does such a branch have until we perform a data-flow analysis on the register value. Data-flow analysis, however, requires we already detected all basic blocks, which creates a circular dependency!

The high-level *switch* statement always has a pre-defined set of possible branch destinations. This list of branch targets is compiled into a **jump table** (also called *switch table*), which is either a list of offsets or addresses. In the following x86-64 example, the table contains *negative offsets* from the beginning of the jump table itself:

```
0x100000de0 <+0>:   ...
0x100000de6 <+6>:   LEA    RAX, QWORD PTR [0x100000E1C]  ; jump table
```

```
    0x100000ded <+13>: MOVSXD RCX, DWORD PTR [RAX + 4*RDI]  ; RDI is the index
    0x100000df1 <+17>: ADD    RCX, RAX
    0x100000df4 <+20>: JMP    RCX                           ; indirect jump
    0x100000df6 <+22>: ...
    0x100000e17 <+55>: RET
; jump table with 6 entries:
    0x100000e1c        DD     0xffffffda
    0x100000e20        DD     0xffffffee
    0x100000e24        DD     0xfffffff3
    0x100000e28        DD     0xfffffff3
    0x100000e2c        DD     0xfffffff4
    0x100000e30        DD     0xfffffff9
```

The jump table is placed immediately after the instructions of the function, *in the same section*. This is an example of *data* stored in the *code section*, which must be analyzed differently, and skipped by the disassembler.

The `LEA` instruction simply loads the jump table address into `RAX`, and the second instruction indexes into this table using the expression `RAX + 4*RDI` and loads the found offset into `RCX`. This offset is *added* to the address of the jump table, but since it is *negative*, the result will point into the function's body.

To be able to distinguish this pattern even before basic block analysis has been performed, we might need to resort to imprecise heuristics again. One such method, which has a surprisingly high success, is to use a *sliding window* to find this pattern, which has several very distinctive properties:

- There is a list of small negative or positive integers at the end of the function, which often does not disassemble properly.
- There is an indirect branch in the function.
- There is a "register + 4*register" type of expression used.
- An address is loaded into a register, which points just beyond the last instruction of the function.

These properties are unlikely to be present in other types of code, and they can also be analyzed by simply looking at individual instructions, without the need for any higher-level analysis.

To add support for jump tables into the basic block detection algorithm, we simply have to add the list of jump table targets (calculated from the offsets) to the set of *leaders* before the function is portioned into basic blocks. The sets of predecessors and successors of the basic block ending with the indirect jump also need to be updated accordingly. The rest of control-flow analysis needs to be prepared to find basic blocks with more than two successors.

In certain cases, the compiler might choose not to generate a jump table, but instead use a different method, for example a comparison-based binary search, a series of comparisons or a combination of a jump table with other methods. In either case, control-flow analysis should be already able to recognize such patterns. If we want to properly fold these methods into a single switch statement, we need to do that later, after data-flow analysis.

4.3 Data flow analysis

Just as control-flow analysis tries to reconstruct what high-level structures were used to transfer control, **data-flow analysis** inspects what happens to the all the data and values that are used throughout a function and tries to deduce what are the high-level semantics of the operations.

Code represented with machine instructions will necessarily have low-level aspects that are either impractical or even impossible to represent in high-level source code. These can include manipulations with the stack pointer, using multiple arithmetic operations to perform one semantic calculation, using registers and stack slots, auxiliary address calculations, using CPU flags, etc. We will perform data-flow based analyses that will try to eliminate or transform low-level instructions into high-level code. At the end, we want all code to be transformed into an AST, describing the whole function using high-level statements and expressions only.

A practical implementation of a decompiler will likely use some sort of **intermediate representation** (IR), a middle-level code that will not contain the low-level and architecture-specific details (like flags and specific set of registers), but will still be in the form of instructions. The decompilation is then split into three parts:

1. *Generating the IR.* Individual machine instructions are semantically analyzed and rewritten into a target-independent intermediate language. The number of instructions is initially larger than before.
2. *IR-level optimizations and simplifications.* Data-flow analysis is performed and its results are used to optimize the code, removing unnecessary instructions.
3. *AST generation.* The remaining IR is transformed into an AST.

Here is an example of a low-level instruction on AArch64, which assigns a constant value to a register while also left-shifting it:

```
MOVZ       X9, #0xC200, LSL #16
```

This could be represented in a generic IR, which hides architecture details and limitations (such as that on AArch64 it is impossible to directly assign 64-bit constants to registers and instead the instructions need to use shifts and other workarounds):

```
$immediate_value := 0xc200
$shifted_value := left_shift $immediate_value, 16
$register_x9 := $shifted_value
```

Since the values all are constants, this can be optimized into a single direct assignment, but if we forget about the optimizations for a while, this could be transformed into an AST, which is also graphically shown in figure 4.5.

```
- Sequence
  - Assignment("imm", Constant(0xc200))
  - Assignment("shifted_value", LeftShift(Variable("imm"), Constant(16)))
  - Assignment("register_x9", Variable("shifted_value"))
```

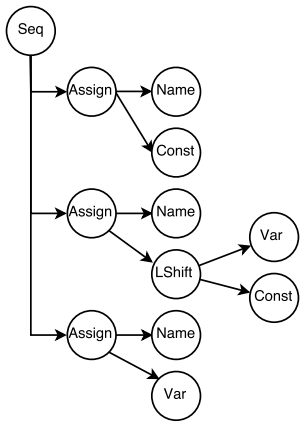


Figure 4.5: Graphical representation of an AST

In this section, we will not describe a specific IR language nor a specific AST representation, because the syntax and structure of these can greatly vary. Most concepts of data-flow analysis can be explained without it, and where necessary we will use a pseudo-code language.

4.3.1 Value definitions and uses

Most assembly instructions operate on registers. An instruction can either read a value from a register (and use it in a computation), write a new value into a register (discarding the previous value) or both (use the current value to compute a new one). When a register is written to, we say that the instruction **defines the value**. After a value is *defined*, it can also have *uses*. Another instruction **uses a value** when it is used as an input in the instruction. Finding the *definitions* and *uses* for all values is an important step in data-flow analysis: If we know that a value is *never used*, we can simply discard its definition. If a value has some uses, we can replace a use with the definition (inline it) without changing the semantics of the function.

Some instructions can have **side effects**, which means that the instruction does something more than just a pure mathematical calculation. Moving or removing instructions with side effects cannot be done easily, because we have to prove the semantics does not change first. Another common anomaly is **implicitly-used registers**, which act as input or output of an instruction, but they are not part of the textual representation of the instruction (the *mnemonic*). Both these *features* might not be obvious when reading an assembly code. Let us take a look at the infamous x86 `IMUL` instruction (which performs a signed multiplication) as an example:

```
...
0x10000aa71         MOV         EAX, 0xabc
0x10000aa76         MOV         EBX, 0xdef
0x10000aa7b         IMUL        EBX
...
```

In this case, the `IMUL` instruction *implicitly uses* the `EAX` register as its second input (besides `EBX`), but it also *implicitly uses* another register for its output,

which is written into the `EDX:EAX` register pair (the output of a multiplication is twice the size as the inputs). Secondarily, it also changes the `CF` and `OF` flags to indicate whether a *carry* and an *overflow* happened.

All of the semantics, side effects and both explicit and implicit register uses must be properly analyzed for the data-flow analysis to produce correct results. This is another place where having an IR, which abstracts these problems away, helps simplify the analysis. For the rest of the section, we are going to assume that we can identify all semantics of all instructions.

An instruction then has:

- a (possibly empty) list of input variables, each of which has
- a set of possible **definitions** (instructions that *define* the value loaded from this variable in a *reachable* way),
- a (possibly empty) list of input constants (immediate values),
- a (possibly empty) list of output variable, each of which has
- a set of possible **uses** (instructions that *use* the value written to the output variable in a *reachable* way).

For each instruction, we can also say which variables are **live** at that point: If the instruction lies on a path between a variable's definition and its use in a *reachable* way, then the variable is *live* at this instruction, otherwise it is not live.

The IR can be structured in a way that if the instruction returns something, it is always a *single value*, which may include multiple inner values, but it is a single entity. In that case we can say that the whole instruction has *uses* (rather than individual output variables). If an output variable does not have any *uses*, we say that the variable is **killed** by the instruction.

Computing the *definitions* and *uses* of all variables within a single basic block is straightforward. The following algorithm will find these, but it will also construct the sets of all input and output variables of a basic block:

1. Initially:
   - Let m be an empty multi-map (or an explicit algorithm input in the later version).
2. For each instruction i in the basic block, do the following:
   (a) If i reads from one of more variables v:
      - Find key-value pairs *(v, j)* in m and set *used-definitions(i)* to be all such values of j.

   (b) If i writes to one or more variables v:
      - If the variable is already one (or more) of the keys in m, remove such entries.
      - Add the *(v, i)* key-value pair into m.

For further enhancements of the algorithm, we will call the initial state the multi-map m as *basic-block-inputs(B)*, and the final state as *basic-block-outputs(B)*. We can see an example of the output of the algorithm on the following basic block:

```
; basic-block-inputs(B) = {}
1   $a := 10       ; used-definitions(1) = {},  m = {$a=>1}
2   $b := $a * 2   ; used-definitions(2) = {1}, m = {$a=>1, $b=>2}
```

```
3   $c := $x       ; used-definitions(3) = {},  m = {$a=>1, $b=>2, $c=>3}
4   $d := $b + 30  ; used-definitions(4) = {2}, m = {$a=>1, $b=>2, $c=>3, $d=>4}
5   $a := 15       ; used-definitions(5) = {},  m = {$a=>5, $b=>2, $c=>3, $d=>4}
; basic-block-outputs(B) = {$a=>5, $b=>2, $c=>3, $d=>4}
```

The remaining task of finding *uses* and *definitions* beyond basic-block boundaries is more complex. The basic idea is to propagate the *basic-block-outputs* of one BB into the *basic-block-inputs* map of another BB when there is an edge in the control-flow graph between these two blocks. This is similar to classic *variable liveness analysis algorithms* [28].

1. Initially, set *basic-block-inputs(B)* and *basic-block-outputs(B)* to empty sets for all basic blocks.
2. Run the single-basic-block algorithm on the entry basic block.
3. While *basic-block-inputs(B)* or *basic-block-outputs(B)* are changing, do the following:
   - For each basic block pair *(A, B)* where an control-flow edge from A to B exists, do:
   (a) Add all *basic-block-outputs(A)* into *basic-block-inputs(B)*.
   (b) Run the single-basic-block algorithm on B.

It should now be easy to see why m is a multi-map – several possible definitions of the same variable can exist at the beginning of a basic block. This happens in the following example:

```
bb_entry:   1   $a := 10
            2   $b := 0
            3   if ($a > 0) goto 5
bb_cond:    4   $b := 42
bb_exit:    5   $c := $b              ; used-definitions(5) = {2, 4}
```

The algorithm will propagate the definition of `$b` into the last block (`bb_exit`) from two sources: Once from `bb_entry`, and once from `bb_cond`. This means that *basic-block-inputs(bb_exit)* will contain two possible definitions of `$b`: {`$b=>2`, `$b=>4`}. If loops are present, a definition of a variable might be propagated from the same from to itself – this is perfectly valid and indicates that the block uses a value from the previous iteration of the loop.

It is easy to see that the algorithm *must finish* at some point, because there is a finite number of key-value pairs in the multi-maps, and the algorithm only grows the multi-maps, never removes from them. Once there is nothing to add, the algorithm finishes. The time complexity of the algorithm certainly depends on how the data structures used are actually implemented, and how sparse the control-flow graph is. For our purposes, it will be sufficient to mention that bit vectors and linked lists allow implementing the calculations as vector operations, which compute the definitions of *all variables at once*; furthermore, compiler-generated control-flow graphs usually have far less edges than complete graphs [29].

4.3.2 SSA form

Static single assignment (SSA) form is a special property of certain IR languages, in which each variable is assigned (*defined*) exactly once, and each variable

is defined before all of its uses. This can be beneficial for data-flow analysis [31], because all names in the function are *unique*, thus there is no need to search for definitions or uses of variables. A good example of such an SSA language is the LLVM IR [30].

Of course, some programs *require* to write a variable more than once. These variables could be promoted to memory accesses, but that would very inefficient. Instead, the SSA form allows the use of a special type of instruction, the **phi node** (also marked as "φ"). The *phi* instruction selects a value from a list of variables or constants based on which basic block was the real predecessor (in the runtime sense of the word). There are additional restrictions on the use of *phi* nodes, for example if they are present they must be before all other instructions in the same basic block. The previous example could be rewritten into SSA form as follows:

```
bb_entry:   1   $a := 10
            2   $b1 := 0
            3   if ($a > 0) goto 5
bb_cond:    4   $b2 := 42
bb_exit:    5   $c := phi bb_entry=>$b1, bb_cond=>$b2
```

Notice that we renamed the two assignments to `$b` into separate variables to satisfy the SSA-form requirements. The value of `$c` is calculated by the *phi* node, which chooses `$b1` in case the execution went directly from the `bb_entry` basic block (the branch at line 3 was taken), or `$b2` in case the execution actually went through the `bb_cond` block.

The *phi* node is not something that physical CPUs implement, so SSA is rather a conceptual intermediate representation that, when lowered to machine instructions, eliminates the phi nodes by converting them to traditional instructions. However, the SSA form can be efficiently generated [32] and allows several compiler transformations to be done more easily or effectively.

4.3.3 Value propagation

Propagating values (variables and constants) from their definitions to their uses is an extremely common compilation optimization. This transformation simply *replaces an input operand of an instruction* with its assigned value. On the other hand, it might not be immediately obvious why would it benefit decompilation. The need for value propagation can include these reasons:

- As a result of the transformation of low-level assembly instructions into an IR language, many expressions can be rewritten into a larger-than-necessary number of IR instructions.
- Other transformations during decompilation can create opportunities for value propagation.
- Value propagation might create opportunities for other transformations.
- The compiler might need to generate several instructions to cover one semantical operation, because the target assembler cannot perform the operation in one instruction.

Once data-flow analysis provides definitions and uses of all variables, it is simple to perform value propagation. All we have to do is identify which values should be propagated, for example variables with a single use, assignments with no arithmetic operations, call arguments. As an example let us take a look at a previously mentioned IR:

```
$immediate_value := 0xc200
$shifted_value := left_shift $immediate_value, 16
$register_x9 := $shifted_value
```

In this piece of code, we can immediately perform propagation of the `$immediate_value` variable and then, in a second step, we can also propagate the `$shifted_value` variable:

```
$immediate_value := 0xc200
$shifted_value := left_shift 0xc200, 16
$register_x9 := left_shift 0xc200, 16
```

There is one important restriction on value propagation: If the value that we are trying to propagate is a variable, it must not be changed on any path between the source and the target of the propagation:

```
$a := $var      ; $var is an input to this basic block
$var := 20
$b := $a
```

In this example, we cannot replace `$a` in the third line with `$var`. Doing so would change the semantics of the function, because `$var` has a different value on line 1 and line 3. This problem can sometimes be solved by renaming some of the uses of a variable to a different name, but there are situations where this is not possible either. Note that in SSA form, this problem does not exist, because all variables are assigned exactly once.

4.3.4 Constant folding

When an arithmetic instruction is performed on constants only, we can directly replace the whole calculation with its result. This operation is called **constant folding**, and it would be rare to find opportunities for it in optimized compiler-generated code. However, there will often be *foldable* expressions during various stages of decompilation. The previously mentioned example can be folded into:

```
$immediate_value := 0xc200
$shifted_value := 0xc2000000
$register_x9 := 0xc2000000
```

4.3.5 Dead code elimination

Another common compiler technique that will be useful for decompilation is **dead code elimination**, which identifies instructions that are safe to remove without changing the semantics of the function. Immediate opportunities for this include variable definitions that do not have any uses. Let us assume the previous example is the full function body which returns the last variable:

```
bb_entry:    $immediate_value := 0xc200
             $shifted_value := 0xc2000000
             $register_x9 := 0xc2000000
             return $register_x9
```

In this case, it is easy to see that the two first instructions can be removed, because they do not have any side-effects, they do not write into memory, and the values they define are not used by any other instructions.

```
bb_entry:    $register_x9 := 0xc2000000
             return $register_x9
```

We could also perform another propagation and elimination to fold the function into a single instruction:

```
bb_entry:    return 0xc2000000
```

The important requirement for safe elimination is that the removed instruction does not have any side effects.

4.4 AST generation

In the previous steps, we already recognized high-level control-flow structures, and we performed data-flow analysis, which optimized the instruction sequences and removed unnecessary variables and code. The next step is to generate an **abstract syntax tree**, a representation where the function is not described by instructions anymore, but rather by statements and expressions. The main difference is that each node in an AST can have an arbitrarily complex subtree, and secondarily, we will start using concrete data types and type safety.

In tree data structures, including ASTs, a common programming technique is the **visitor pattern**, which is often used to generate, print, transform, optimize or search the AST. The idea is to define actions for all possible types of nodes in the AST, and then recursively apply these actions. The whole algorithm is often defined in a single class, separating it from the classes that describe the individual AST nodes.

Transforming the IR into an AST can be done trivially by simply rewriting each instruction into an appropriate AST expression, while turning each IR variable into an AST variable. In figure 4.6 we have a sample IR of the GCD function, where the instructions are already optimized to some degree (but not completely, there are still some opportunities). The `$rsi` and `$rdi` variables are function inputs.

This can be directly transformed into the following AST representation (ignoring variable declarations and function header):

```
Sequence([
  - Assignment("rax", Variable("rsi"))
  - Assignment("rbx", Variable("rdi"))
  - Assignment("zf", Equals(Variable("rbx"), Constant(0)))
  - WhileStatement(
      Test: Negate(Variable("zf"))
      Body: Sequence([
        - Assignment("rcx", Variable("rax"))
```

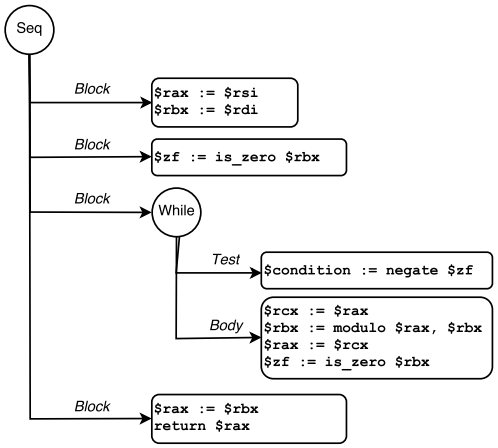


Figure 4.6: IR with recognized control-flow

```
        - Assignment("rbx", Modulo(Variable("rax"), Variable("rbx")))
        - Assignment("rax", Variable("rcx"))
        - Assignment("zf", Equals(Variable("rbx"), Constant(0)))
       ])
     )
  - Assignment("rax", Variable("rbx"))
  - Return(Variable("rax"))
])
```

During the initial AST generation, we probably want to already apply some optimizations, because we can use the data-flow analysis results. We can identify variables that are only used once, but which has not been propagated because the IR can only perform one operation per instruction. This allows us to fold multiple calculations into a single complex expression, which can improve readability, but we have to apply empirical limits so we do not end up generating functions with only one extremely complex statement.

We also have to choose high-level **data types** for the variables used in the AST. This is inherently a heuristic task, because many low-level instructions do not indicate whether we are operating on *signed or unsigned* integers. When the variable type does not match what the high-level language expects in some expression, *casts* have to be generated. This can indicate that the type of the variable is wrong, and we can try to perform an additional pass over the AST that will try to remove as many casts as possible (since programmers usual try to avoid using casts).

4.4.1 AST transformations

Once we have completely switched to operate on an AST, it is time to perform optimizing transformations on it, in a process called **AST rewriting**. We will have a set of transformation rules that identify sub-optimal code patterns and replace them with equivalent but more readable variants, similarly to how IDEs suggest refactoring options.

Let us list just a few examples of useful AST transformations:

- If the body of a *while* loop ends with a statement that is the same as the statement before the loop, the statement can be moved into the loop condition:

```
a = expression(...);
while (a) {
  ...
  a = expression(...);
}
```

Can be transformed into:

```
while (a = expression(...)) {
  ...
}
```

- A multi-statement expression inside an *if* condition can be moved out:

```
if ((a, b)) ...
```

Can be transformed into:

```
a;
if (b) ...
```

- If a variable is used in two disjoint scopes independently, it can be split into two separate variables that can be declared inside the scopes.
- *While* loops where an incrementing variable is detected can be transformed into *for* loops.
- *Goto* statements from inside a loop can be changed into *break* and *continue* statements if possible.

4.4.2 Printing the source code

A standard way of producing the actual source code from an AST is to use the visitor pattern again, and each node prints its own syntax including the serialized outputs of sub-nodes. For example, an `IfStatement` AST node would be reduced to a string of the format `if (%1) { %2 } else { %3 }`, where `%1`, `%2`, `%3` are replaced by the string representations of the three sub-nodes (the test node, the "true" body and the "false" body).

Producing the source code this way can also perform other tasks that can lead to interesting features of a decompiler:

- **Syntax highlighting** is trivially achievable, because the AST describes the full syntax. When printing the source code, we can easily color different node types differently, without any need of text parsing.
- The visitor implementation can keep track of **indentation** so the output is always correctly indented. Whenever we descend into a new scope (e.g. into the body of an *if* statement), we increment the current indent and all statements will print with more spaces/tabs at the beginning of the line.

- Besides text output, we can also produce **ranges for each node**, so we can easily find out which node is under the user's cursor. An advanced editor can then allow features like block collapsing or corresponding-bracket highlighting.
- By having text ranges for nodes, **editing** the AST can be allowed. The editor can let the user directly apply transformations to nodes, like swapping the *then* and *else* bodies of an *if* statement (and negating the condition to preserve the semantics).

Chapter 5

Objective-C Specific Decompilation

In the previous chapters, we discussed what all steps are needed to build a generic decompiler. Now we will take a look at programs written in Objective-C, what extra information we can extract from Obj-C binaries, and how we can decompile specific language constructs.

5.1 Runtime type information

Due to the dynamic nature of Objective-C, all **classes**, **methods**, **properties** and **instance variables** generate metadata in the resulting binary, which is used by the language runtime. This information is stored in separate sections, which include:

- The `__objc_classlist` section contains a list of all **classes** defined by the program, as a list of pointers to class descriptors. A *class descriptor* is a structure containing the superclass, the class name, instance size, implemented protocols, methods, instance variables layout and list of properties.
- Each **instance variable** (*ivar*) description contains its name, byte offset in the object and encoded type. Properties also list their names and type. Methods and class methods include their full selectors, including encoded types of arguments and the return value.
- Sections `__objc_methname` and `__objc_classname` lists the names of classes and methods (selectors) used by the program.
- References to externally-provided classes and methods are listed in `__objc_classrefs` and `__objc_selrefs`.
- In the `__objc_ivar` section, we will find *offsets* for individual instance variables for all classes defined by the program.

There are other important metadata preserved in other sections, and a complete specification of the structures in the Objective-C runtime header files [33]. The reason why so much type information is stored in the binary is to allow several advanced features of Obj-C: Classes can be *created at runtime*, including adding

methods and instance variables, they can also be looked up by name during runtime. Various information about objects and classes can be queried at runtime, and **key-value coding** [34], **key-value observation** [34] and **method swizzling** [36] build on these features. In fact, the whole **dynamic dispatch** approach requires to be able to list methods available to an unknown object.

This means that we are able to reconstruct almost complete class declarations, and there are existing tools that dump these declarations in the form of header files, which can be then used to link against libraries which do not provide header files [37].

For decompilation, certainly the most interesting piece of information are the type information about method arguments and instance variables, plus the fact that instance variable offsets are not "hard-coded" but they are always read from a global variable. Consider the following machine instruction listing:

```
0x100000ef4 <+0>:  PUSH   RBP
0x100000ef5 <+1>:  MOV    RBP, RSP
0x100000ef8 <+4>:  MOV    RBX, QWORD PTR [0x100001130]
0x100000efa <+7>:  MOV    RBX, QWORD PTR [RDI + RBX]
0x100000eff <+11>: ADD    RDX, RBX
0x100000f04 <+16>: MOV    RAX, RDX
0x100000f07 <+19>: POP    RBP
0x100000f08 <+20>: RET
```

While it is not particularly hard to analyze the inputs and outputs of this function, if we apply the Obj-C metadata into our analysis, we will immediately know that this function is a method of a certain class, we will realize that `0x100001130` stores the offset of an ivar and we will know the type signature of the method and it is much easier to analyze the function:

```
; Method signature: (long)my_method:(long)arg;
; (hidden) input argument "self" is in RDI
; (hidden) input argument "cmd" is in RSI
; input argument "arg" is in RDX
; output is in RAX
    0x100000ef4 <+0>:  ...                                  ; function prologue
    0x100000ef8 <+4>:  MOV    RBX, _$_MyClass_$_my_ivar_$_offset
    0x100000efa <+7>:  MOV    RBX, QWORD PTR [RDI + RBX]    ; read "my_ivar" ivar
                                                            ; of "self" object
    0x100000eff <+11>: ADD    RDX, RBX                      ; add "RBX" to "arg"
    0x100000f04 <+16>: MOV    RAX, RDX                      ; return the result
    0x100000f07 <+19>: ...                                  ; function epilogue
```

Now it is very easy to say that this method simply reads the `my_ivar` instance variable of the `self` object, and adds the value to its argument of type `long`. We do not have to guess whether the final value of `RAX` should be discarded or if it is a real return value, because we know the signature of the method.

The type encodings do not describe every single possible type, but they do include class types, common integer and floating-point types, strings, arrays, pointers, selectors and structures. A full specification of the encoding is in the *Objective-C Runtime Programming Guide* [38], but here is a few examples:

```
// variable types:
char c;         // type encoding: c
long l;         // type encoding: l
char *str;      // type encoding: *
```

```
id obj;          // type encoding: @
NSArray *arr;    // type encoding: @"NSArray"
SEL selector;    // type encoding: :

// function signatures:
- (id)init;                                // signature: @16@0:8
- (void)dealloc;                           // signature: v16@0:8  ... "v" is void
- (id)initWithCoder:(NSCoder *)decoder;    // signature: @24@0:8@16
```

Type encodings for simple variables usually use one letter (`c` for `char`). The encodings for full function signatures contain the return type first, and they also include numbers which indicate offset in the stack frame. If we ignore the numbers, decoding the signature is easy: `@16@0:8` without numbers is `@@:`, which means a function returning an object (the first `@`), and taking two arguments, first is an object (the second `@`), second is a selector. All Objective-C methods have these first two arguments, which indicate the receiver of the object (hidden `self` argument) and the invoked selector (hidden `cmd` argument).

5.2 Function calls and message passing

Objective-C code does not use classic direct function call mechanisms nor *virtual tables* for method calls. Instead, all method calls (also called *message sends* or *message passing*) are routed via a **dynamic dispatch** mechanism. Let us take a look at an example of an Obj-C method call and a roughly equivalent C code, which is similar how the compiler translates the call:

```
// Obj-C:
// "dictionary" is an instance of NSMutableDictionary
[dictionary setObject:myObject forKey:myKey];

// C:
char *selector = "setObject:forKey:";
objc_msgSend(dictionary, selector, myObject, myKey);
```

The `objc_msgSend` function is the main dispatcher, which looks up the selector in the object's class method table, and invokes that method (via an optimized *tail call*). This dispatcher is heavily optimized, so a fast path (method is present in the method cache) performs less than 20 instructions on x86-64 [39]. Since `objc_msgSend` handles all selectors, it is a variadic function, and in fact a whole family of these functions exist for various purposes:

```
id objc_msgSend(id self, SEL op, ...);
id objc_msgSendSuper(objc_super *super, SEL op, ...);  // call superclass method
id objc_msgSendSuper2(objc_super *super, SEL op, ...);  // newer ABI
void objc_msgSend_stret(id self, SEL op, ...);  // returns a struct
void objc_msgSendSuper_stret(id self, SEL op, ...);
void objc_msgSend_fpret(id self, SEL op, ...);  // returns floating-point number
void objc_msgSend_fp2ret(id self, SEL op, ...);  // returns two FP values
```

These prototypes are still only "informational", because these functions sometimes do not follow them. For example, `objc_msgSend` is declared to return `id`, but it can also return integer numbers or `void`.

Let us take a look how a method call looks like in compiled code:

```
0x100001a63 <+0>:  MOV    RDI, ...                      ; set the receiver
0x100001a67 <+4>:  MOV    RSI, QWORD PTR [0x10003cdb8] ; load the selector
0x100001a6e <+11>: CALL   _objc_msgSend                 ; call objc_msgSend
```

The data stored at `0x10003cdb8` is a **selector reference**, which contains a pointer to an actual string containing the method's selector. For better readability, we can rewrite the assembly listing to contain the selector name more explicitly:

```
0x100001a63 <+0>:  MOV    RDI, ...
0x100001a67 <+4>:  MOV    RSI, _$_SELECTOR_$_setObject:forKey:
0x100001a6e <+11>: CALL   _objc_msgSend
```

Calling **class methods** works very similarly, because all classes are Obj-C objects as well. The only difference is that the receiver of the message is not an instance, but it is a pointer to the class (which is really a global variable), which is loaded from a **class reference** pointer:

```
0x1000184c1 <+0>:  MOV    RDI, _$_CLASS_$_NSMutableSet ; QWORD [0x100003dce]
0x1000184c7 <+6>:  MOV    RSI, _$_SELECTOR_$_set       ; QWORD [0x10003cdb8]
0x1000184cb <+10>: CALL   _objc_msgSend
```

To be able to analyze function calls, we simply data-flow analysis to find the values of `RDI` and `RSI` at the time of the call to `objc_msgSend` (for x86-64). The selector will be constant in almost all cases so we will be able to find out which method is being called. From this we can deduce what arguments does it expect (and in which registers or stack positions) and what is the returned value.

5.2.1 Function calls as opaque statements

For decompilation, this behavior has one subtle but extremely important property: The compiler has to treat each method call as an opaque, unoptimizable branch. This means that it *cannot inline* any method into another, and it cannot move any memory accesses over a function call. Other optimizations are also limited, for example a method call *cannot* be eliminated, even when the target has an empty body. After any method call, all memory has to be treated as *clobbered*, so any global variables used have to be re-read. This is the price of having a dynamic language where an implementation of a method is allowed to be changed at runtime.

Similar applies to *property accesses*, which are actually directly transformed into function calls, either to the getter or the setter of the property. When the programmer does not supply the accessor functions, they are automatically generated by the compiler. Let us look at the following Obj-C example:

```
@interface MyClass : NSObject
@property long myNumber;
@end

@implementation MyClass
- (long)myMethod {
  long sum = 0;
  for (int i = 0; i < 10; i++) {
    sum += self.myNumber;
  }
  return sum;
```

```
}
@end
```

The property read inside the loop is actually turned into a method call and a getter is generated by the compiler:

```
- (long)myMethod {
  long sum = 0;
  for (int i = 0; i < 10; i++) {
    sum += [self myNumber];
  }
  return sum;
}

- (long)myNumber {
  return self->_myNumber;  // ivar access
}
```

The function call and the fact that it needs to dynamically dispatched prevents the compiler from inlining the ivar access and optimizing the loop into a single multiplication. This is a major difference from C++ code, where method inlining is very common, and even when a method is invoked via the *virtual table mechanism*, there are still optimization opportunities (specialization or the fact that virtual tables of one object cannot be changed after the object is constructed).

The resulting optimized code of this loop can look like the following assembly listing, where it is easy to recognize a loop and a function call inside the loop:

```
    ...
    0x100000e79 <+11>: MOV    R14, RDI
    0x100000e7c <+14>: MOV    R12, 0xA                  ; R12 starts at 10
    0x100000e89 <+27>: MOV    RBX, 0                    ; RBX is the result
-------------------------------------------------------------------------
    0x100000e8b <+29>: MOV    RDI, R14
    0x100000e8e <+32>: MOV    RSI, _$_SELECTOR_$_myNumber
    0x100000e91 <+35>: CALL   _objc_msgSend             ; call -[self myNumber]
    0x100000e97 <+41>: ADD    RBX, RAX
    0x100000e9a <+44>: DEC    R12
    0x100000e9d <+47>: JNE    0x100000E8B               ; <+29>
-------------------------------------------------------------------------
    0x100000e9f <+49>: MOV    RAX, RBX                  ; return result via RAX
    ...
```

In summary, this all is very good news for decompilation, because certain patterns are more likely to be preserved in Obj-C compiled code than in compiled code written in other languages. Based on the fact that method calls and property accesses are very common in most Obj-C code, we will see that automatic decompilation can often reconstruct a very high-quality source code.

5.3 Reference counting, MRR and ARC

Each dynamically allocated object in Objective-C has a **reference count** (also called **retain count**). When we need to keep an object alive, we **retain** it to increase its reference count, and when we do not need the object anymore, we have to **release** it to decrement the reference count. When the retain count drop

to zero, the object is deallocated. On top of that, Objective-C provides a concept of **autoreleasing**, which releases a reference to an object, but only marks the object to be deallocated later and not immediately.

There are two major compiler modes for reference counting:

- **Manual Retain-Release** (MRR) means that the programmer is completely is managing the retain count of all objects used in their code. This is done via cals to `objc_retain` and `objc_release` (from C code) or calling the `retain` and `release` selectors on Obj-C objects.

A strict set of memory-management rules exists and the programmer must follow them in order to avoid memory corruptions, leaks and undefined behavior [40]. The rules influence *naming of functions*, for example when a function contains "Create" in its name, it will always return a "+1" reference count [41].

- **Automatic Reference Counting** (ARC) is a modern compiler technology which implements the strict rules for maintaining reference counts in the compiler itself and it is what most modern Objective-C code uses. Whenever an object's reference is stored to a local variable, global variable or an ivar, the compiler automatically retains the object. Similarly, when a reference is removed (or replaced), the previous object is released. Variables with this behavior are called **strong references**, and the language also provides a concept of non-retaining variables, called **weak references**. These are often used to avoid cyclic object references (*retain cycles*) which cause memory leaks.

Developers are actively discouraged from writing code with manual retain count management, because it is error-prone and makes code less readable, so we will also focus on ARC.

When analyzing an Obj-C binary, we will often see calls to retain count management functions, such as `objc_storeStrong`, `objc_autoreleaseReturnValue`, `objc_retainAutoreleasedReturnValue`, `objc_retain` and `objc_release`, even when the original source code contained no such calls. This is the result of ARC, the exact rules about how the compiler behaves and what function calls it generates is described in Clang's documentation [42]. it is desirable to produce decompiled source code which does not manually call the ARC functions, so let us take a look at an example of an IR code and how it can be transformed:

```
; Function prototype:
; - (void)methodWithObject:(id)object;
; Input "object" is in register RDX.
bb_entry:    $local_object := $register_rdx
             call objc_retain, $local_object
             $result := call objc_msgSend, $local_object, _$_SELECTOR_$_method
             $result2 := call objc_retainAutoreleasedReturnValue, $result
             call objc_release, $local_object
             return $result2
```

In this case, we can recognize that the retain-release pair on `$local_object` and the retaining of the autoreleased return value from the method call are both results of ARC and we can remove them. The retain and releases can be simply stripped off, and the call to `objc_retainAutoreleasedReturnValue` can be changed to return the object that was passed to it:

```
; - (void)methodWithObject:(id)object;
bb_entry:    $local_object := $register_rdx
             $result := call objc_msgSend, $local_object, _$_SELECTOR_$_method
             $result2 := $result
             return $result2
```

5.4 Modern Objective-C syntax

Starting with Objective-C 2.0, several language affordances have been added to improve readability of the source code. This section discusses several such features and how the decompiler should recognize patterns during analyses to generate a more high-level code.

5.4.1 Objective-C literals

A full list of Objective-C literals is available in Clang's documentation [43]. Straightforward rewritings of the generated AST can be used to replace the "old" syntax of the following examples:

```
NSNumber *num = [NSNumber numberWithInt:42];
NSString *str = [NSString stringWithUTF8String:"hello"];
NSArray *arr = [NSArray arrayWithObjects:o1, o2, nil];
NSDictionary *d = [NSDictionary dictionaryWithObjectsAndKeys:o1, k1, o2, k2, nil];

id obj = [array objectAtIndex:idx];
[mutableArray replaceObjectAtIndex:idx withObject:obj];
id obj = [dictionary objectForKey:key];
[mutableDictionary setObject:obj forKey:key];
```

These can be "modernized" into:

```
NSNumber *num = @42;
NSString *str = @"hello";
NSArray *arr = @[o1, o2];
NSDictionary *d = @{ o1: k1, o2: k2 };

id obj = array[idx];  // idx is an integer
mutableArray[idx] = obj;
id obj = dictionary[key];  // key is an object
mutableDictionary[key] = obj;
```

Both the "old" and "new" examples are valid decompilation results, but the latter expressions are more readable, so a decompiler should choose to prefer them.

5.4.2 The initializer pattern

A recommended way of writing an initializer for a class looks like the following:

```
- (instancetype)init {
    self = [super init];
    if (self) {  // self can be NULL e.g. when we are out of memory
        ...  // perform additional setup
    }
    return self;
}
```

The `instancetype` keyword indicates that the method returns an object of the same class as the class that the method is defined in, or one of its subclasses. It helps the compiler in static type checking. The second unusual thing is the assignment to `self`. It might look like it has some special semantics, however, `self` is a simple local argument to the function, and during the execution of the function, it acts as an ordinary local variable.

Detecting this pattern is straightforward again, the following assembly example shows the call to `[super init]` on x86-64, which stores the result in `RAX`. This register then holds the `self` variable for the rest of the function, and is then returned. During AST generation or rewriting, we will simply rename this local variable to `self`.

```
    0x100000e50 <+0>:  ...                                ; function prologue
    0x100000e67 <+23>: MOV    RSI, _$_SELECTOR_$_init
    0x100000e6e <+30>: MOV    RDI, ...                    ; super-call options
    0x100000e72 <+34>: CALL   _objc_msgSendSuper2         ; call [super init]
    0x100000e77 <+39>: TEST   RAX, RAX                    ; RAX is self
    0x100000e7a <+42>: JE     0x100000e8b                 ; <+59>
-------------------------------------------------------------------------
    0x100000e7c <+44>: ... additional instance setup
-------------------------------------------------------------------------
    0x100000e8b <+59>: ...                                ; function epilogue
    0x100000e90 <+64>: ret                                ; returns self in RAX
```

5.4.3 Method call chaining

Another AST-level optimization that the decompiler should perform is embedding several method calls that operate on the previous call's result into a single chain of calls. This is a common pattern in most object-oriented programming languages, and in Objective-C this might look like the following examples:

```
NSMutableArray *array = [[NSMutableArray alloc] init];
NSString *version =
    [[[NSBundle mainBundle] infoDictionary] valueForKey:@"CFBundleVersion"];
NSURLRequest *req = [NSURLRequest requestWithURL:[NSURL URLWithString:str]];
```

Whether a decompiler should embed a call into another needs to be a heuristic decision, most likely based on the complexity of the resulting expression. If the two function calls have a short textual representation (e.g. the name of the method), then we should more likely embed them. This basically emulates the developer's thinking when writing code, where they will also consider the total length of an expression in the source code.

Some patterns, like the alloc-init chained call in the previous example, are almost always embedded together, and detecting those special situations will further improve the generated source code readability.

5.5 Blocks

Blocks are a major feature of Objective-C that is extensively used both by library vendors and end users. Many popular libraries and framework exist purely to provide easy-to-use API via blocks, and the language feature is even available in pure C when using Clang as the compiler. One of the most common uses of

blocks involves **Grand Central Dispatch** (GCD), a standard library used for asynchronous execution and multithreading.

Let us take a look at a complex example of declaring a block, and its later invocation:

```
__block long shared_variable;
long (^my_block)(long) = ^(long input) {
    shared_variable += input;
    return 42;
};
...
long output = my_block(10);
```

This block takes one integer as its input and returns an integer value as well. However, it also captures a special variable marked with `__block`, which means the variable will be promoted to a *heap-allocated variable* so it is accessible even when the declaration goes out of scope. Special rules apply if the block operates with objects or other blocks.

Successfully decompiling blocks is certainly non-trivial, as it involves:

- finding the *block descriptor* in the binary,
- analyzing the type information from the block descriptor,
- deducing which arguments are captured and how (by value, by reference, `__block` variables),
- at the block creation site, analyzing the captured local variables,
- at the call site, tying local variables to the block's invoke routine.

Note that the implementation details of blocks are actually part of a platform's ABI, which means that the structure will not be changed in the future. This is what allows us to pattern match and parse the structure, even when they are not a public API.

5.5.1 Block descriptors

The compiler produces a **block descriptor** for each block that it compiles. This is a global structure which, among other things, contains the block's *signature*, which is a description of the type of the block (inputs and outputs) and types of captured variables. Note that the descriptor does not contain a pointer to the block's compiled code, as one descriptor can be used by multiple blocks of the same type.

A rough structure of the descriptor looks like this:

```
struct Block_descriptor {
    unsigned long reserved;
    unsigned long size;
    void *copy_helper;
    void *dispose_helper;
    const char *signature;
}
```

Finding the block descriptor among all other global structures is not obvious, as there is no list of all block in the binary's metadata. Pattern matching over static data in the binary can be used with the following hints:

- On 64-bit platforms, the `size` member is 40, indicating that the block descriptor has five 8-byte-wide members.
- The `copy_helper` and `dispose_helper` members point to beginning of helper functions, which are in the code section and are not part of any class method. These helper functions are very likely to contain calls to `Block_object_assign` and `Block_object_dispose` APIs.
- The `signature` pointer points to a string containing a type encoding.
- The address of the descriptor is referenced from the code section.

The last item, a reference from the code section, is important to find as we will use it to find the pointer to the actual code of the block. If we know where the block descriptor is referenced from, we can look to nearby instructions, as they are likely to contain the code pointer. This again needs to be heuristic, but a pointer to the beginning of an otherwise-unreferenced function is unusual. If the binary we are analyzing is not stripped (which removes local symbol names), then this job is much easier, because the block's code will be a function with the string `_block_invoke` in its name.

As mentioned before, there can be multiple places that reference the block descriptor. In such a case, all of these instances are creating different blocks of the same signature.

With this we can build a database of all blocks that are defined by the program, and for each block we will know its signature, code pointer and places where these are referenced from.

5.5.2 Blocks on the stack

When a function declares and defines a block, it will create a specific structure on the stack, called a **block literal**. The same structure is used when the block is a global variable. If we take a look at the example above, the compiler will generate the following local variable:

```
struct Block_literal {
    void *isa = &_NSConcreteStackBlock;  // _NSConcreteGlobalBlock for globals
    int flags;
    int reserved;
    void *invoke = ...;  // pointer to the block's code
    Block_descriptor *descriptor = ...;  // pointer to the block descriptor

    void *shared_variable = ...;  // pointer to the byref captured variable
    ... // possibly other captured variables
}
```

Such a large stack item should be easy to recognize as well, mainly because of the `isa` pointer, which always points to `_NSConcreteStackBlock` or `_NSConcreteGlobalBlock` classes. The structure of the block literal explains why we are likely to see references to the block's code and block descriptors close to each other. This large structure has to live on the stack and cannot be optimized into registers.

5.5.3 Captured variables

When the stack block is allocated, it also captures local variables. When a variable is **captured by value**, its current value is simply copied to the end of the block literal. Here is an example assembly listing of a function which defines a block inside:

```
0x100000e38 <+0>:  PUSH   RBP
0x100000e39 <+1>:  MOV    RBP, RSP
0x100000e3c <+4>:  SUB    RSP, 0x30         ; a large structure on the stack
0x100000e40 <+8>:  MOV    RAX, &_NSConcreteStackBlock
0x100000e47 <+15>: MOV    QWORD PTR [RBP - 0x28], RAX   ; set isa
0x100000e4b <+19>: MOV    DWORD PTR [RBP - 0x20], 0xC0000000 ; set flags
0x100000e52 <+26>: MOV    DWORD PTR [RBP - 0x1C], 0x0
0x100000e59 <+33>: MOV    RAX, 0x100000E8B              ; code pointer
0x100000e60 <+40>: MOV    QWORD PTR [RBP - 0x18], RAX   ; set invoke
0x100000e64 <+44>: MOV    RAX, 0x100001040              ; descriptor pointer
0x100000e6b <+51>: MOV    QWORD PTR [RBP - 0x10], RAX   ; set descriptor
0x100000e6f <+55>: MOV    QWORD PTR [RBP - 0x8], RDX    ; capture RDX by value
...
```

Of course, such a code sequence can be optimized in many ways, but once we recognize the large stack item and its structure, we can identify the individual member assignments (isa, invoke pointer, descriptor, captured variables).

This means that every generated block literal has a different structure, based on what variables are captured.

Capturing a value by reference works differently, because it transforms the `__block` variable into another large stack structure:

```
// By-reference declaration:
__block int a;

// Transformed into:
struct block_byref_a {
    void *isa;
    struct block_byref_a *forwarding;
    int flags;
    int size;
    void *copy_helper;
    void *dispose_helper;
    int captured_a;  // actual storage
};
```

A reference to this structure is stored into block literal. Since we know the signature of the block, we can identify that we are capturing a by-ref value and from this we can recognize the by-ref stack structure. All subsequent accesses to the actual storage are done via the `forwarding` pointer, which again is something that we should rewrite into direct assignments:

```
block_byref_a->forwarding->captured_a = 10;

// Should be decompiled into:
a = 10;
```

5.5.4 Block code

The actual body of a block needs to able to access the captured variables, which are available through the block literal. A pointer to the block literal is added as an extra (hidden) argument to the block's signature. Let us take a look at an example of a block definition:

```
long x;
long (^my_block)(long) = ^(long input) {
    long a = x;  // access to a captured variable
    long b = input;  // access to an explicit input
    ...
};
```

This actual generated block body will be translated by the compiler into:

```
long my_block_invoke(Block_literal *lit, long input) {
    long a = lit->x;  // access to a captured variable
    long b = input;  // no change
    ...
}
```

When decompiling a block body, we just need to take the extra parameter into account and recognize accesses via the block literal pointer. We already know what variables and types are captured in the block literal and at what offsets these variables are.

5.5.5 Invoking a block

We have described what the construction of a block looks like. Actually *calling a block* is much simpler, as it only means calling the `invoke` method stored in the block literal. Any variable captures are already resolved in the block literal, we only have to provide explicit parameters.

```
- (void)directlyInvokeBlock:(dispatch_block_t)block {
  block();
}

// Is equivalent to:
- (void)directlyInvokeBlock:(block_literal *)block_literal {
  block_literal->invoke();
}
```

5.5.6 Generating source code for blocks

If we want to generate correct source code for decompiled blocks, we have to analyze them together with the outer function that they are defined in. From the block descriptor, we will learn the input and output parameters, and from the outer function's block literal, we will understand what variables are being captured.

Dealing with the captured variables is fairly simple: All we have to do in the block's code is to rename local variables in such a way that they have the same name as the outer function's captured variables.

Let us look at a complete listing of an optimized assembly for a simple function that creates a block and calls a GCD function with it:

```
; Method signature:
; - (void)myMethod:(long)arg;
; Input "arg" is in register RDX
myMethod:
    0x100000dd1 <+0>:   PUSH   RBP
    0x100000dd2 <+1>:   MOV    RBP, RSP
    0x100000dd5 <+4>:   PUSH   R14
    0x100000dd7 <+6>:   PUSH   RBX
    0x100000dd8 <+7>:   SUB    RSP, 0x30
    0x100000ddc <+11>:  MOV    R14, RDX
    0x100000ddf <+14>:  XOR    EDI, EDI
    0x100000de1 <+16>:  XOR    ESI, ESI
    0x100000de3 <+18>:  CALL   _dispatch_get_global_queue
    0x100000de8 <+23>:  MOV    RDI, RAX
    0x100000deb <+26>:  CALL   _objc_retainAutoreleasedReturnValue
    0x100000df0 <+31>:  MOV    RBX, RAX
    0x100000df3 <+34>:  MOV    RAX, &_NSConcreteStackBlock
    0x100000dfa <+41>:  MOV    QWORD PTR [RBP - 0x38], RAX
    0x100000dfe <+45>:  MOV    DWORD PTR [RBP - 0x30], 0xC0000000
    0x100000e05 <+52>:  MOV    DWORD PTR [RBP - 0x2c], 0x0
    0x100000e0c <+59>:  MOV    RAX, &my_block_body
    0x100000e13 <+66>:  MOV    QWORD PTR [RBP - 0x28], RAX
    0x100000e17 <+70>:  MOV    RAX, &my_block_descriptor
    0x100000e1e <+77>:  MOV    QWORD PTR [RBP - 0x20], RAX
    0x100000e22 <+81>:  MOV    QWORD PTR [RBP - 0x18], R14
    0x100000e26 <+85>:  LEA    RSI, [RBP - 0x38]
    0x100000e2a <+89>:  MOV    RDI, RBX
    0x100000e2d <+92>:  CALL   _dispatch_async
    0x100000e32 <+97>:  MOV    RDI, RBX
    0x100000e35 <+100>: CALL   _objc_release
    0x100000e3b <+106>: ADD    RSP, 0x30
    0x100000e3f <+110>: POP    RBX
    0x100000e40 <+111>: POP    R14
    0x100000e42 <+113>: POP    RBP
    0x100000e43 <+114>: RET

my_block_body:
    0x100000e44 <+0>:  PUSH   RBP
    0x100000e45 <+1>:  MOV    RBP, RSP
    0x100000e48 <+4>:  MOV    RSI, QWORD PTR [RDI + 0x20]
    0x100000e4c <+8>:  MOV    RDI, &format_string
    0x100000e53 <+15>: XOR    EAX, EAX
    0x100000e55 <+17>: POP    RBP
    0x100000e56 <+18>: JMP    _printf

my_block_descriptor:
    0x100001060:       DQ     0x0000000000000000
    0x100001068:       DQ     0x0000000000000028
    0x100001070:       DQ     0x0000000100000f2e "v8@?0"
    0x100001078:       DQ     0x0000000100000f59 ""

format_string:
    0x100000f2a:       DB     "%ld", 0
```

It fairly simply to recognize the block's body (`my_block_body`), its descriptor (`my_block_descriptor`) and the block creation site (starting at `0x100000dfa` in `myMethod`). The block descriptor contains a signature, `v8@?0`, which means "a function returning `void`, total argument size of 8 bytes, and a block literal pointer

at offset 0". The resulting block literal structure looks like this (annotated with offsets):

```
struct my_block_literal {
    void *isa;                      // offset 0x0
    int flags;                      // offset 0x8
    int reserved;                   // offset 0xc
    void *invoke;                   // offset 0x10
    Block_descriptor *descriptor;   // offset 0x18
    long arg1;                      // offset 0x20
}
```

This allows us to decompile the block's body, and resolve the `RDI + 0x20` pointer dereference into a nice member access. We can rewrite the function directly into an IR:

```
; void my_block_body(my_block_literal *lit);
my_block_body:   $register_rsi := extract_member $register_rdi, arg1 ; offset 0x20
                 $register_rdi := &format_string
                 $register_eax := 0x0
                 call printf, $register_rdi, $register_rsi
```

This can be further optimized, and we can generate the AST. Notice the access to the captured variable via the `lit` argument.

```
void my_block_body(my_block_literal *lit) {
    long arg1 = lit->arg1;
    printf("%ld", arg1);
}
```

Now, rewriting the outer function directly into IR results in the following:

```
; - (void)myMethod:(long)arg;
myMethod:   $register_r14 := arg
            $register_rdi := 0x0
            $register_rsi := 0x0
            $register_rax :=
                call dispatch_get_global_queue, $register_rdi, $register_rsi
            $register_rdi := $register_rax
            $register_rax :=
                call objc_retainAutoreleasedReturnValue, $register_rdi
            $register_rbx := $register_rax
            set_member $lit, isa, &_NSConcreteStackBlock        ; offset 0x0
            set_member $lit, flags, 0xc0000000                  ; offset 0x8
            set_member $lit, reserved, 0x0                      ; offset 0xc
            set_member $lit, invoke, &my_block_body             ; offset 0x10
            set_member $lit, descriptor, &my_block_descriptor   ; offset 0x18
            set_member $lit, arg1, $r14                         ; offset 0x20
            $register_rsi := address_of $lit
            $register_rdi := $register_rbx
            call dispatch_async, $register_rdi, $register_rsi
            $register_rdi := $register_rbx
            call objc_release, $register_rdi
```

Since we already recognized the `my_block_literal` variable on the stack, we already know which offsets are accessed by the assignments and we can generate the AST:

```
- (void)myMethod:(long)arg {
    long local1 = arg;
    void *local2 = dispatch_get_global_queue(0x0, 0x0);
    void *local3 = objc_retainAutoreleasedReturnValue(local2);
    my_block_literal lit;
    lit->isa = &_NSConcreteStackBlock;
    lit->flags = 0xc0000000;
    lit->reserved = 0x0;
    lit->invoke = &my_block_body;
    lit->descriptor = &my_block_descriptor;
    lit->arg1 = local1;
    void *local4 = &lit;
    dispatch_async(local3, local4);
    dispatch_release(local3);
}
```

Now we will combine the two decompiled sources, while also applying some more simplification (removing of ARC memory management, propagating and remaining variables). we will embed the code from the block body into the outer function and replace all references to `lit->arg1` with `arg` to resolve the variable capture. The full final result is as simple as the following decompiled code:

```
- (void)myMethod:(long)arg {
    dispatch_queue_t queue1 = dispatch_get_global_queue(0x0, 0x0);
    dispatch_block_t block1 = ^() {
        printf("%ld", arg);
    };
    dispatch_async(queue1, block1);
}
```

5.5.7 Lazy-initialization pattern

Objective-C provides a convenient and extremely efficient way to perform thread-safe lazy initialization using the **dispatch_once** macro. it is usage is almost ubiquitous throughout Obj-C applications and so it is important to recognize it. However, doing so requires additional support from the decompiler. A typical use of *dispatch_once* looks like the following:

```
- (id)singletonInstance {
    static MyClass *instance = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        instance = [[MyClass alloc] init];
        ...  // other initialization of "instance"
    });
    return instance;
}
```

The `dispatch_once` function is defined as a macro, and it expands to the following expansion, which is optimized in such a way that the fast path does not even need to perform a function call:

```
- (id)singletonInstance {
    static MyClass *instance = nil;
    static dispatch_once_t onceToken;
    dispatch_block_t block = ^{
```

```
        instance = [[MyClass alloc] init];
        ...  // other initialization of "instance"
    };
    if (onceToken != 0xffffffffffffffff) {
        dispatch_once(&onceToken, block);  // a function call, not a macro call
    }
    return instance;
}
```

Recognizing this pattern is straightforward, if we assume that we can already handle block definitions. The decompiler then simply needs to wrap the call to `dispatch_once` and the comparison into a single macro call. The referenced global variable `onceToken` should also be moved into the function as a "static" variable, which further optimizes this pattern.

5.6 Objective-C fast enumeration

Iterating over an Objective-C collections, such as NSArray, NSSet or NSDictionary, can be in a traditional C-style way:

```
NSArray *array = ...;
for (int i = 0; i < array.count; i++) {
    id item = [array objectAtIndex:i];
    ...;
}
```

However, this involves at least two function calls per iteration: One to retrieve the item and one to get the length of the array. Furthermore, the access to `array.count` is an opaque statement to the compiler, it cannot optimize the loop (e.g. reading the array length cannot be moved out of the loop, unrolling the loop is not possible). Therefore, Objective-C offers **fast enumeration** (also called **for-in loop** or **for-each loop**), which uses a different syntax for the *for* loop [44]:

```
NSArray *array = ...;
for (id item in array) {
    ...;
}
```

Fast enumeration is based on the `NSFastEnumeration` protocol, which is implemented by NSArray, NSSet, NSDictionary and other collection classes. The protocol consists of a single method, which uses a helper structure `NSFastEnumerationState`:

```
typedef struct {
    unsigned long state;
    id *itemsPtr;
    unsigned long *mutationsPtr;
    unsigned long extra[5];
} NSFastEnumerationState;

@protocol NSFastEnumeration
- (NSUInteger)countByEnumeratingWithState:(NSFastEnumerationState *)state
                                  objects:(id [])buffer
                                    count:(NSUInteger)len;
@end
```

When a collection implements this protocol, the method is supposed to store a C array of objects in `buffer` (of up to `len` elements), and return the number of elements stored. The function will be called repeatedly until it returns 0, indicating that we have reached the end of the collection. The `state` parameter can be used by the implementation for bookkeeping information (e.g. position in the array), and can include mechanisms to detect collection mutation during iteration, which is forbidden and indicates a programmer's error. The `itemsPtr` field of the `NSFastEnumerationState` structure is the actual pointer to the returned data – the implementation can choose whether it will set `itemsPtr` to point to the user-supplied `buffer`, or whether it will point it to some internal data structure.

A *for-in* loop is translated by the compiler into the following:

```
id item;
NSFastEnumerationState state = { 0 };
id objects[16];
NSUInteger limit;

limit = [array countByEnumeratingWithState:&state objects:&objects count:16];
if (limit > 0) {
    unsigned long m = *state.mutationsPtr;
    while (limit > 0) {
        for (unsigned long i = 0; i < limit; i++) {
            if (m != *state.mutationsPtr) objc_enumerationMutation(array);
            item = state.itemsPtr[i];

            ... // body of the for-in statement
        }
        limit = [array
            countByEnumeratingWithState:&state objects:&objects count:16];
    }
}
item = nil;
```

5.6.1 Recognizing fast enumeration in compiled code

First thing to note is that the generated compiled code contains multiple nested control-flow structures including two loops. This is going to make pattern matching very complex because we need to match against multiple (5 or more) basic blocks.

First of all, spotting that a function uses fast enumeration is very simple, we just need to detect references to `objc_enumerationMutation` and the `countByEnumeratingWithState:...` selector. The source code is extremely unlikely to contain calls to these methods directly. However, more issues arise from the fact the compiler is able to optimize the nested loops in many ways, and the actual generated code structure differs under various optimization levels. But we should still note that the compiler is limited by the following:

- The calls to `countByEnumeratingWithState:...` are opaque to the compiler, it cannot be inlined and almost no information can be statically inferred about the returned values.
- Both the `state` and `objects` local variables need to live on the stack and cannot be optimized out, because their addresses are needed as arguments to the opaque function call.

- The code from the body of the *for-in* loop is unlikely to be moved out from the loop or even within the most inner loop of the generated code, because the `item` variable is only calculated as the last compiler-generated statement. it is also calculated in a way which is hard to statically reason about (it is loaded from a pointer that is modified by a previous opaque method call).

Based on this, we can deduce that most of the generated auxiliary basic blocks will not contain any decompilation-interesting code. If we can detect which basic block contains the user-written body of the *for-in* loop, we may discard the two generated loops altogether.

As an example, let us take a look at figure 5.1, which shows the CFG of a function with a *for-in* loop. It should not be hard to spot the two inner loops, and also a short block which calls `objc_enumerationMutation`. we will use this block as an anchor, because its successor block must be the basic block containing the user's *for-in* body.

The approach we can take here is to only analyze the inner-most block and treat the whole CFG subgraph as if the loops were not executed at all (the very first branch based on the `limit` variable returns false). In this case, we would be able to transform the following CFG structure:

```
- Sequence
  - ...
  - If
    - Test: Basic Block #1            ; contains the reference to the container
    - True: Sequence
      - Basic Block #2
      - While
        - Test: Basic Block #3
        - Body: While
          - Test: Basic Block #4
          - Body: Sequence
            - If
              - Test: Basic Block #5
              - True: Basic Block #6  ; contains call to objc_enumerationMutation
            - Basic Block #7          ; contains user's body of the for-in loop
      - Basic Block #8
  - ...
```

This can be simplified into:

```
- Sequence
  - ...
  - Basic Block #1
  - FastEnumeration
    - Array: ... ; extracted from Basic Block #1
    - Loop variable: ... ; extracted from Basic Block #7
    - Body: Basic Block #7
  - ...
```

This just presents the idea how such a change might work. The full implementation of such a transformation would require coordination on both the CFG-recognition level, data-flow analysis and post-AST optimization. In case of

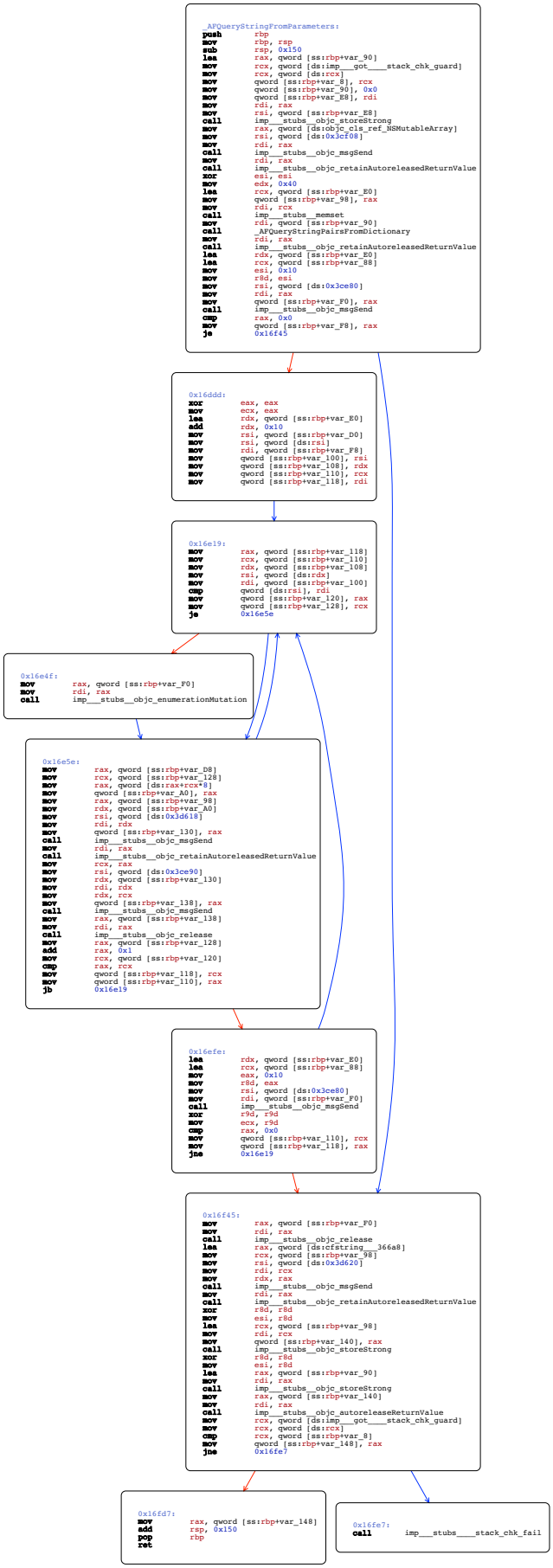


Figure 5.1: An example of an optimized compiled *for-in* loop from the AFNetworking project. Graphical output from the *Hopper* program.

nested *for-in* loops or if the body if the loop contains other interesting control-flow statements, we should analyze the basic blocks from the innermost to the outermost one, as the presented CFG algorithm in the previous chapters did.

Alternatively, the transformation can be done on the AST level. This, however, comes with a downside that a lot of the necessary work is much easier done by the data-flow analysis on an IR level.

Let us take a look at an example. An already optimized output of the decompiler could produce an AST like the following:

```
void *_AFQueryStringFromParameters(void *arg1) {
    NSFastEnumerationState state;
    id objects[16];
    local_mutable_array1 = ...;
    rax = ...;
    void *local1 = &state;
    void *local2 = &objects;
    void *array = rax;
    rax = [array countByEnumeratingWithState:local1 objects:local2 count:0x10];
    if (rax != 0x0) {
        ...
        do {
            do {
                ...
                if (...) {
                    objc_enumerationMutation(...);
                }
                rax = ...;  // "rax" becomes the loop variable
                rax = [rax URLEncodedStringValue];
                rax = [rax retain];
                local3 = rax;
                [local_mutable_array1 addObject:rax];
                [local3 release];
                ...
            } while (...);
            rax = [array
                countByEnumeratingWithState:local1 objects:local2 count:0x10];
            ...
        } while (rax != 0x0);
    }
    rax = ...;
    return rax;
}
```

The transformation can completely ignore the "..." parts of the complex control-flow structure, and replace the outer *if* statement with a *for-in* structure. The receiver of the first `countByEnumeratingWithState...` call is the container that we are iterating. The *loop variable* is retrieved from the inner-most block.

```
void *_AFQueryStringFromParameters(void *arg1) {
    NSFastEnumerationState state;
    id objects[16];
    local_mutable_array1 = ...;
    rax = ...;
    void *local1 = &state;
    void *local2 = &objects;
    void *array = rax;
    rax = [array countByEnumeratingWithState:local1 objects:local2 count:0x10];
```

```
    for (rax in array) {
        rax = [rax URLEncodedStringValue];
        rax = [rax retain];
        local3 = rax;
        [local_mutable_array1 addObject:rax];
        [local3 release];
    }
    rax = ...;
    return rax;
}
```

Further optimizations can eliminate the auxiliary local variables and the method call to `countByEnumeratingWithState...`, and also the ARC memory management calls. The resulting method can look like this:

```
void *_AFQueryStringFromParameters(void *arg1) {
    local_mutable_array1 = ...;
    rax = ...;
    void *array = rax;
    for (rax in array) {
        rax = [rax URLEncodedStringValue];
        [local_mutable_array1 addObject:rax];
    }
    rax = ...;
    return rax;
}
```

5.7 Class type system

Being object-oriented, the Objective-C language provides a type system that allows class hierarchies with inheritance and protocols, which describe interfaces that classes can implement. However, all class type information is only used statically during compilation and during runtime no type checks are performed. Since Objective-C dynamically dispatches method calls resolved at runtime, it does not matter what actual type an instance is, as long as it responds so the selectors that we send to it.

This means that we can treat all object instances as the `id` type, which can describe any object type. This will decompile to correct code, but the readability of such code would suffer, because one of best practices when writing Objective-C code is to declare variables, arguments and return types as the most specific type possible. The two following code sequences are compiled to the identical compiled result:

```
- (id)methodReturningArrayWithMyString {
    id array = [NSMutableArray array];
    id str = @"my string";
    id str2 = [str uppercaseString];
    [array addObject:str2];
    return array;
}

- (NSArray *)methodReturningArrayWithMyString {
    NSMutableArray *array = [NSMutableArray array];
    NSString *str = @"my string";
    NSString *str2 = [str uppercaseString];
```

```
    [array addObject:str2];
    return array;
}
```

Furthermore, upcasting and downcasting of an instance is unrestricted and also unchecked; downcasting to a wrong class type is valid as long as we do not invoke an unavailable method.

System header files provide method prototypes for standard system-provided classes. As we shown in the previous sections, we can reconstruct method prototypes for classes that are defined in the binary that we are analyzing, but the metadata stored in the binary do not indicate what class a parameter or a return type is, they only store an information that the value *is an Objective-C object*. This means that we have to deduce actual object specific types of arguments, return types and local variables.

5.7.1 Type analysis

The type analysis is best done on the AST level, where we can already assign types to all variables that we are working with. The decompiler needs to propagate types from already-known function return types and infer the types onto local variables. Based on what selectors are called on parameters, we can deduce the actual classes of them. There are several heuristics we can also use:

- Objective-C recommends programmers to use a very verbose **naming system** that often includes the method's return type or the types of its parameters, e.g. a method called `componentsJoinedByString:obj` suggests that `obj` is an NSString, `isEqualToNumber:obj` identifies that `obj` is an NSNumber *and* that the method returns a boolean.
- Some method names are available on a too broad set of types, such as the `description` method, which is present in almost all classes. However, we can construct a *set of selectors* that are called on a variable and then find a type which responds to all of them.
- Properties and ivars *contain their class types in the metadata*. When we find an access to a property or an ivar, we can infer the type of the assigned or read variable. However, the actual variable can still be a superclass or subclass of the property or ivar class.

5.8 Summary

The chapter discussed several features of the Objective-C language, runtime and compiler which directly affect the design of a potential Objective-C decompiler. Among other things, we have seen that there are specifics about the language and resulting compiled code, which will make the job of a decompiler both easier and harder:

- Various metadata about classes, methods and blocks can be used to reconstruct proper method signatures and identify property accesses, instance variable accesses, method calls and block instantiation and invocation.

- Blocks, fast enumeration, reference counting and high-level language syntax are extra features that the decompiler must support, but all are recognizable using pattern matching on appropriate decompilation levels.
- Function calls and property accesses are *opaque statements* to the compiler, which means it cannot optimize beyond them.
- Naming of methods can be used to infer types of variables and parameters.

Chapter 6

The "Cricket" Objective-C Decompiler

This chapter introduces a practical implementation of an Objective-C decompiler called "**Cricket**", which uses the techniques and algorithm described in previous chapters. Cricket supports the **Mach-O** binary format and code compiled for **i386**, **x86-64** and **AArch64** (64-bit ARM) architectures using LLVM as the compiler.

Cricket runs on OS X, but it is written mostly in Python with possible porting to other platforms in mind. The decompiler consists of four major components:

- The main **analysis core**, which performs binary analysis, disassembling, basic block detection, intermediate code generation, control-flow analysis, data-flow analysis, AST generation and source-level rewriting optimization.
- A **graphical user interface** (GUI) featuring an *integrated development environment* (IDE), in which users can analyze and decompile programs in a convenient and interactive way. The GUI serves as a controller to the analysis core and allows the user to influence and amend the analysis outputs.
- **Command-line interface** (CLI), which is an alternative to the GUI and which automatically generates a decompilation result in textual output. It allows far less configuration and user interaction, but it can be used from scripts and automated analysis environments.
- A **test suite** containing various automated tests of individual parts of the decompiler. They verify that the supported Objective-C structures decompile correctly.

Figure 6.1 shows how the main components interact. The *analysis core* acts as a back-end component and the other components are possible front-ends to it.

The decompiler's only software requirements are an installation of the OS X operating system, 10.10 (named *Yosemite*) or newer, and an installation of the Xcode development package (version 6 or higher). Both are provided free of charge by Apple and can be installed on supported Mac computers. A packaged distribution of the decompiler (an application bundle named *Cricket.app*) does

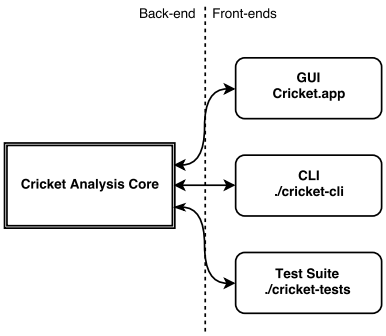


Figure 6.1: Main components of the *Cricket* decompiler.

not require any additional software, as it embeds all of its dependencies within the package.

For local development, however, several libraries need to be present on the developer's machine:

- Python 2.x [45],
- Homebrew [46],
- Qt [47],
- PyQt for Python 2.x [48],
- the `capstone` [49], `distorm3` [50] and `pycparser` [51] Python packages (available via Python's `pip` package manager), and
- Graphviz [52].

The application's `README` file contains instructions how to properly install these software packages, how to run the GUI application, how to use the command-line interface and how to run the provided test suite. The source code also contains a script called `deploy.sh` which creates a stand-alone distribution of Cricket.app as described above. A short user documentation is included as an appendix to this thesis, and developer documentation is provided in the `README` file and also as comments in the applications source code.

Several sample binary executables are provided as well to demonstrate the capabilities of the decompiler. These include both synthetic examples which show how individual language constructs are handled, as well as real-world optimized binaries generated from popular open-source projects.

6.1 Main design decisions

The main motivation to create a **GUI application** is based on the goals and practical use-cases of malware analysis, security software implementation verification and other *manual* work by a researcher analyzing compiler-generated binaries. The **interactivity** of such a GUI IDE is a key convenience factor, and interactive disassemblers are very popular tools among security researchers.

Different use-cases, such as automatic reconstruction of lost source code, automatically searching for certain code structures or recompilation, were only considered as secondary, but valid goals. For these cases, a non-interactive command-line interface was designed.

Naturally, the decompiler was written as an OS X application, because the Objective-C language is extremely often tied to OS X and iOS development, most of which is done on a desktop OS X system. Although Objective-C code can be compiled to other binary formats, **Mach-O** is the only supported executable and library format that can be loaded into Cricket. it is also the platform's standard format for OS X and iOS programs. It may seem that the software is too tied to OS X, but Cricket was written in Python and with portability in mind. Most of the analysis core is completely platform independent, and the GUI framework used (Qt) also works on most major platforms. This means that creating a port for other systems would require only small changes, for example in the binary format parsing.

The choice of Python as the primary language to write a decompiler might seem odd since traditional compilers and compiler tools which need to operate on individual instructions usually use strongly-typed languages. There are several reasons behind this: The Python+Qt+PyQt combination provides a very easy way to design the GUI without writing code and to conveniently handle GUI events, yet still being completely platform independent. Python is a dynamic programming language which does not need the source code to be explicitly compiled before running, which helps speed up development, especially compared to larger projects in C++, which often suffer from long compilation times. Thirdly, there is a huge number of readily available Python packages and bindings for almost any area of software development, including disassemblers and other binary analysis frameworks.

The GUI front-end provides an environment in which user can fulfill a complete analysis workflow, starting by opening a binary file (executable or a library), selecting a class and method to analyze, overseeing the individual levels of decompilation and generating the resulting source code and displaying it in an editor window with syntax highlighting. In all the decompilation steps, the user can observe and influence the intermediary results. This differs from other available products, which usually do not allow the user to see unfinished decompilation outcome, but it proved to be of tremendous help for the development of the tool itself, while also giving expert users more options to get better decompilation results.

6.2 Analysis core

The main analysis core of Cricket has a multi-stage pipeline architecture, shown in figure 6.2.

The following decompilation stages are used during decompilation:

- **Binary loading** is responsible for parsing the binary file, finding individual sections and segments, parsing the Objective-C metadata (class and method lists, type information, finding block descriptor) and finding all possible functions, including unnamed non-Obj-C procedures. It also analyzes external references (dynamically linked libraries). The techniques for this are described in chapter 3.

  For this purpose, Cricket uses several system-provided and 3rd-party tools, because binary format is not the end goal of this thesis. For example,

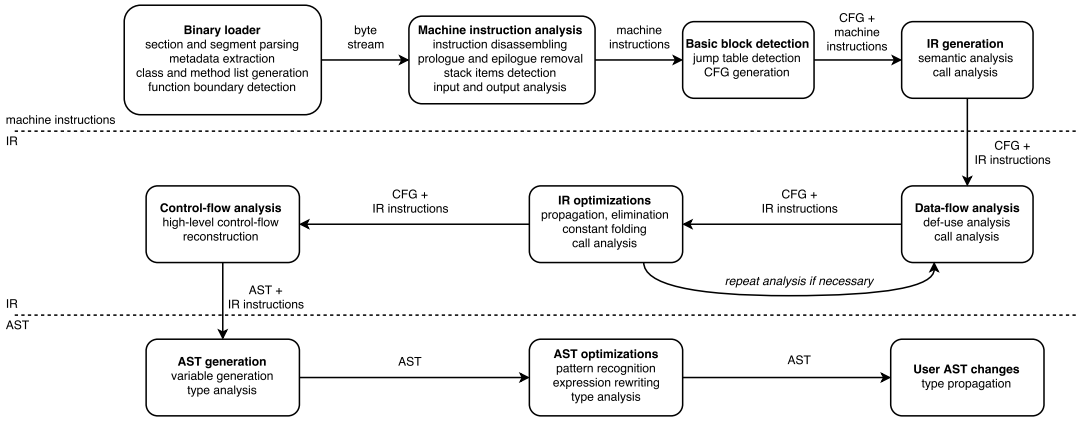


Figure 6.2: Architecture of the analysis core.

the system tools named `otool` and `dyldinfo` are able to list sections, find Objective-C classes and methods and list external symbols.

We also perform two heuristic passes over the code and data sections to detect additional functions that are not described in the metadata, as described in section 3.4.1. One pass looks for `CALL` instructions with a constant target over the code section. The second pass skims the data sections and looks for data that look like pointers into the code section, and for any such pointers we perform a heuristic detection whether the code looks like the beginning of a function (based on the first few instructions, e.g. a `PUSH RBP; MOV RBP, RSP` is an extremely common function prologue on x86-64 but it is also very unlikely to be used elsewhere). This detects functions accessed from virtual tables and other data structures.

Based on the results, we cut the code section on each detected function boundary to get a list of both named and unnamed procedures.

- **Machine instruction analysis** then operates on a single function. On this level, we already know the bounds of the function and we will use the Capstone disassembler framework [49] to transform the stream of bytes into assembly instructions and to get detailed information about each instruction.

  Based on the function prologue and epilogue, we will learn how large the stack frame of the function is (see section 4.1.1), and in case it is an Obj-C method, we will know what arguments and return type it has and in which registers these are expected (see section 4.1.2). We will try to reconstruct individual stack items based on the access sizes of each stack offset in the assembly listing (see section 4.1.5).

  Afterwards, this analysis step tries to remove all assembly instructions that are unnecessary in the following stages. This includes the function prologue and epilogue, stack setup, saving and restoring of callee-saved instructions (as defined by the ABI), and several common compiler-generated patterns, such as stack-overflow detection mechanisms [53]. We also detect tail-calls that use a direct `JMP` instruction and replace them with an explicit `CALL` and `RET` pair.

- **Basic block detection** uses the algorithm described in section 4.2.1 to detect all basic blocks from the single list of assembly instructions. Special analysis is done to detect jump tables (see section 4.2.4). A control-flow graph with successor and predecessor lists is constructed.

- **IR generation** is a stage which performs **semantic analysis** to rewrite assembly instructions into a custom IR language (called **µCode**), which is thoroughly described later in this chapter in sections 6.2.1 and 6.2.2.

  The IR still consists of instructions, but it tries to abstract away from machine-specific behavior. It has an infinite amount of registers. Individual instructions usually do not have any side effects and they explicitly state their inputs and outputs. Special handling is needed to handle `CALL` instructions because the inputs and outputs of a function call differ based on the signature of the function.

  A significant part of the implementation of Cricket is the semantic analysis which contains descriptions how each machine instruction should be translated into the IR, which needs to rewrite all of each instruction's semantics. The Capstone disassembling library provides useful insights into machine instructions, as it is able to decode instructions and extract the opcodes, individual parameters, describe memory accesses involving index pointers, etc.

  Cricket provides semantic support for 3 major CPU architectures: AArch64, i386, and x86-64. The last two are implemented by the same semantic analysis module.

  The output of this stage consists of IR instructions only, so from now, all further analysis is architecture-independent.

- **Data-flow analysis** and **data-flow optimizations** are performed on the IR based on the algorithms described in section 4.3.

  The IR which is used as input to the data-flow analysis may contain some constructs which are not proper IR: Stack item accesses are not distinguished and instead they are performed as memory accessed via the frame pointer and offset. Function calls can be "unresolved", which means that we do not know what the actual parameters are.

  Both these irregularities make data-flow analysis very problematic because its algorithms require that we know what inputs and outputs each instruction has. If a function call is "unresolved", we have to be conservative and assume that *any* currently defined variable can be used as a parameter. To "resolve" the function call, we, however need some results of the data-flow analysis: For example, in Obj-C method calls, the inputs depend on the selector we are calling, but we might not be able to deduce the selector until we propagate and constant-eliminate the value in the variable containing the selector.

  This problem is solved by performing the data-flow analysis in a conservative mode where unresolved calls are treated as if they had all variables as inputs, and all memory pointers are assumed that they might alias. Then we perform IR optimizations that are again conservative, with the intention

that the results will help us resolve function calls and memory accesses. A special optimization tries to promote accesses to a stack item into a local variable, which certainly enhances the IR, but sometimes is not possible (e.g. when a pointer to the stack item is used as an argument to a function call). The data-flow analysis is then repeated and so on.

- **Control-flow analysis** takes an input in the form of individual basic blocks, their relations (the control-flow graph) and their IR contents. Then it performs a catalogue-based pattern matching on the CFG to detect various high-level control-flow structures, as described in section 4.2.3. The result is an AST of those recognized structures, but the leaf nodes of the tree are still basic blocks with IR code.

- **AST generation** enters the last representation of the code as it generates a full abstract syntax tree from the optimized IR instructions, as described in section 4.4. Variables in IR are transformed into local variables in the generated Objective-C code and generic types of variables are chosen. Instructions are rewritten into statements that perform the same calculations.

- **AST optimization** performs various enhancements on the AST to improve readability of the generated source code (see section 4.4.1). This includes type analysis and type inferring, which will try to use the most specific types possible, and get rid of extraneous casts. Expression embedding and call nesting are done.

- **User AST changes** are then applied, if the user chooses to. There are a lot of refactorings available to the user, starting from simple variable renaming, to restructuring control-flow statements. These are used by a user to further enhance the quality of the decompiled result, where the automatic decompilation did not choose the best option. Local variable names are one obvious thing where the decompilation will often provide sub-optimal results. Allowing the user to post-process the source code at the AST level can still offer a more convenient way of analyzing the code than a regular text editor.

This gives a general overview of the Cricket decompiler and the individual steps and representations that are used during the decompilation. The following sections discuss the design and internals of some of the parts of Cricket.

6.2.1 Intermediate code

Cricket introduces its own custom intermediate code language, called **µCode**, which is specifically designed to allow decompilation-specific optimizations and also to serve as a common ground for later stages of decompilation. Architecture-specific machine instructions are translated into *µCode* instructions by a semantic analysis module, but after that all of the following transformations are architecture independent. Supporting multiple architectures is then achieved by simply having multiple semantic analysis modules.

Production compilers often have a well-defined IR as well, which separates both the language-specific front-ends and target-specific back-ends, and allows

the same compiler infrastructure to be reused for several source code languages and several result architectures. An option to use some existing IR language and infrastructure, instead of designing a custom intermediate instruction language, was considered. For example, the **LLVM IR** [30] seems to be an interesting option, because of advantages such as having a set of transforms that already perform optimizations on the IR. However, it was decided that the benefits of introducing a decompilation-specific custom IR would outweigh those provided by the LLVM IR:

- LLVM IR strongly prefers to be in SSA form, and most existing transformations require it. Requiring our IR generation and optimization passes to work in SSA mode would make the analysis more complex.
- LLVM IR does not allow certain constructions, such as "unresolved" function calls.
- The existing infrastructure of LLVM is well suited for forward compilation, but not much for decompilation.
- Lastly, not depending on an existing IR helps speed up the development and changes.

The actual syntax and semantics of *μCode* were designed to be very simple to read and easy to generate. As a result, a single machine instruction often needs to be translated to several *μCode* instructions, and it is the task of subsequent optimizations to simplify the calculations. A function written in *μCode* has an **infinite number of registers** available for its use, and each register has an associated size in bytes. For example, a register "`val.8`" has a size of 8 bytes. The smallest register size is 1 byte, which is also used to represent boolean 1-bit variables. There is no maximum size of a register, but we will rarely use sizes larger than 8 bytes (native register size on 64-bit architectures). Registers are also called *variables* and there is no distinction between a variable and a register.

Each *μCode* instruction has an **opcode** starting with a lowercase letter "u", followed by uppercase mnemonic. For better readability, instruction opcodes also show the size of the resulting value. Instructions always **explicitly state their input and output variables**, with the exception for the `uCALL` instruction which can be "unresolved" (see below). Unless specifically stated they have no side effects and do not produce other outputs.

Let us take a look at a few examples of move instructions and basic integer arithmetic operations:

```
uMOV.8      rbx.8 := 0x1               ; store constant value into "rbx"
uMOV.8      rax.8 := rbx.8             ; store the value of "rbx" into "rax"
uADD.8      var1.8 := rax.8 + rbx.8    ; add "rax"+"rbx", store result in "var1"
uSUB.8      var2.8 := rax.8 - 0x1      ; subtraction
uDIV.8      quotient.8 := rax.8 / 0x2  ; division
uMOD.8      remainder.8 := rax.8 % 0x2 ; modulo
uNOP                                   ; does nothing
```

The registers used do not need to be explicitly declared and the first use automatically declares the register and its size. The register sizes are final, for example, once the program uses `rax.8` as a register, it cannot use `rax.4` later. With the exception of function inputs, each register needs to be defined (by an instruction that writes to the register) before it is used as an input to a subsequent

instruction. Constants do not explicitly state their sizes, they have an implicit size based on the other inputs and outputs of the instruction.

Most instructions require their inputs and outputs to be of the same size. If a smaller-sized variable is to be used in a larger-sized calculation, it needs to be zero- or sign-extended, and similarly for truncation:

```
uEXTEND.8   larger.8 := EXTEND(smaller.1)      ; zero-extending
uTRUNC.1    byte.1 := TRUNC(larger.8)          ; truncating
```

The supported architectures use **flags** to represent boolean properties of register values and calculation outputs. In *μCode*, flags are regular 1-byte variables, and the `uFLAG` instruction needs to be explicitly used to calculate them:

```
uFLAG.1     zf.1 := ZERO(register.8)    ; stores 1 to "zf" if register is zero
uFLAG.1     of.1 := OVERFLOW(a.8 + b.8) ; stores 1 to "of" if "a+b" overflows
```

Memory accesses are done by using the `uSTORE` and `uLOAD` instructions:

```
uMOV.8      ptr.8 := 0x10000896a     ; stores a constant value to "ptr"
uSTORE.1    *(ptr.8) := b.1          ; stores "b" (1 byte) into "*ptr"
uLOAD.8     val.8 := *(otherptr.8)   ; loads an 8-byte word into "val"
```

Notice that the size of the `uSTORE` instruction indicates how many bytes are being written to. The pointer size is always the native register size on the used architecture (8 bytes on 64-bit architectures and 4 bytes on 32-bit architectures), but the size of the memory access can be different.

Function calls are done via the `uCALL` instruction. The syntax of this instruction differs based on the call arguments and return type. If the function call does not return anything, the instruction does not have any output register either. A special case is an "unresolved" function call marked by "`...`" in the parameter list. In this case, a subsequent analysis needs to resolve the call arguments (either by matching the function from a database of function prototypes or by heuristic analysis). Functions can also be called indirectly when the pointer to the function is stored in a register.

```
uCALL       result.8 := my_function(rdi.8, rsi.8)  ; fully resolved call
uCALL       other_function()                       ; void-returning function
uCALL       result.8 := unresolved_function(...)   ; unresolved function call
uCALL       rax.8()                                ; indirect call
```

Both conditional and unconditional branches are expressed by the `uBRANCH` instruction, which specifies a 1-byte wide register as its condition (or `0x1` when the branch in unconditional) and a target label, which must be an address of one of the function's basic block:

```
0x1000000e0:
  uMOV.8      result.8 := 0x0
  uFLAG.1     zf.1 := ZERO(rdi.8)
  uBRANCH     zf.1, 0x1000002f8                  ; branch if "zf" is "1"
0x100000120:
  uMOV.8      result.8 := 0x2a
  uBRANCH     0x1, 0x1000002f8                   ; unconditional branch
0x1000002f8:
  uRET        result.8                           ; return "result"
```

The `uBRANCH` instruction requires that the target label is a constant. To handle *switch* statements, a `uSWITCH` instruction exists. The `uRET` instruction exits the function and optionally returns a register as the result of the function.

As *μCode* is not meant to be manually written or being parsed, it is missing some explicit information, such as the function header (with the inputs and outputs and the appropriate registers). The Cricket decompiler keeps track of those internally, and the editor provides these in generated comments.

Types in *μCode* are only specified in byte sizes and there is no distinction between signed and unsigned types. There is no register aliasing and no subregister accesses are allowed. When several registers need to form a larger structure, we will just create the variable as a total size of the structures as well as `uEXTRACTMEMBER` and `uSETMEMBER` with either integer offsets or field names of known structures. Assuming the following is a known C structure:

```
struct two_ints {
    long first_integer;
    long second_integer;
};
```

We can then extract and set members of this structure either directly via offsets or by names:

```
uCALL             ret.16 := function_returning_two_ints()
uEXTRACTMEMBER.8  first.8 := EXTRACT(ret.16, 0x0)
uEXTRACTMEMBER.8  second.8 := EXTRACT(ret.16, 0x8)
uSETMEMBER.8      ret.16 := SET(ret.16, first_integer, 0x2a)
uSETMEMBER.8      ret.16 := SET(ret.16, second_integer, 0x29a)
```

There is no need to declare that a variable is a particular structure type. Partial accesses to variables in *μCode* always use little-endian byte order.

6.2.2 Semantic analysis

The semantic analysis module in Cricket transforms machine instructions into *μCode* while maintaining almost all semantics of the original instructions. There are two implementations of the semantic analysis, one for AArch64 and one for both 32- and 64-bit Intel architectures (i386 and x86-64), due to the similarities in their instruction sets.

Some instructions have a 1-to-1 mapping into *μCode*, but some can contain complex expressions (e.g. indexed memory accesses), so temporary variables need to be introduced. For example the following AArch64 instruction:

```
MOVZ     X14, #0xc200, LSL #16
```

Can be translated into this *μCode* (ignoring the obvious constant folding we can immediately perform):

```
uMOV.8        temp1.8 := 0xc200
uMOV.8        temp2.8 := 0x10                  ; decimal 16
uSHIFTLEFT.8  x14.8 := temp1.8 << temp2.8      ; perform the left shift
```

Indexed memory addressing on x86-64 is another example of a complex expression in a single instruction:

```
; x86-64:
  MOV       R14, QWORD PTR [RDX + R15 * 8]

; uCode:
  uMOV.8    tempaddr.8 := rdx.8
  uMOV.8    tempidx.8 := r15.8
  uMUL.8    tempidx.8 := tempidx.8 * 0x8
  uADD.8    tempaddr.8 := tempaddr.8 + tempidx.8
  uLOAD.8   r14.8 := *(tempaddr.8)
```

There are several fundamental difficulties in the semantics of the machine instructions that the semantic analysis must handle. All of the supported architectures allow **sub-register accesses**, but *μCode* does not have direct support for this. Such support was intentionally left out because it would make data-flow analysis much more complicated (without sub-register accesses there is no register aliasing and we can easily find *all* accesses to one variable just by looking for its name within the function).

This means that expressing sub-register accesses in *μCode* needs to use more complex patterns. For example, the following sequence of x86-64 instructions:

```
  MOV     RBX, RAX
  MOV     BYTE PTR [RCX], BL            ; stores one byte
                                        ; BL is the lowest byte of RBX
```

Can be expressed in *μCode* with:

```
  uMOV.8          rbx.8 := rax.8
  uTRUNC.1        bl.1 := TRUNC(rbx.8)      ; "bl.1" is not a subregister,
                                            ; it is a regular variable
  uSTORE.1        *(rcx.8) := bl.1
```

Secondly, several machine instructions return more than one value, which is done by writing to multiple registers by one instruction. Sometimes this can be easily solved by splitting the instruction into more *μCode* instructions, for example, the x86-64 integer division instruction (`DIV`) returns the quotient in `RAX` and the remainder in `RDX`. In *μCode* this is represented by:

```
  uDIV.8      rax.8 := src1.8 / src2.8
  uMOD.8      rdx.8 := src1.8 % src2.8
```

If either of these results is unused by the rest of the function, it will be eliminated by later optimizations. When the multiple-register output of an instruction cannot be logically split to perform several calculations, a larger register must be used to store the result. However, the result must be stored in the original registers to maintain the semantics of the following instructions

```
; x86-64:
  MUL         RBX                       ; multiplies RAX * RBX,
                                        ; result is stored in RDX:RAX

; uCode:
  uMUL.16           tempres.16 := rax.8 * rbx.8
  uEXTRACTMEMBER.8  rax.8 := EXTRACTMEMBER(tempres.16, 0x0)      ; low 8 bytes
  uEXTRACTMEMBER.8  rdx.8 := EXTRACTMEMBER(tempres.16, 0x8)      ; high 8 bytes
```

Alternatively, `uTRUNC` can be used to extract the lower 8 bytes of the result.

Arithmetic as well as explicit comparison instructions often set CPU flags. The `uFLAG` instructions are generated to capture all of the semantics:

```
; x86-64:
  TEST        RCX, 0x1                    ; sets SF, ZF and PF
                                          ; based on bitwise (RCX & 0x1)

; uCode:
  uFLAG.1     sf.1 := SIGN(rcx.8 & 0x1)
  uFLAG.1     zf.1 := ZERO(rcx.8 & 0x1)
  uFLAG.1     pf.1 := PARITY(rcx.8 & 0x1)
```

6.2.3 Stack items promotion

When registers are not enough to store all of the function's local and temporary variables, the function often allocates space in its **stack frame**. The function prologue commonly uses one register to be the **frame pointer**, through which all the **stack items** are accessed.

The following example shows an x86-64 assembly listing which uses two 8-byte stack items are accessed via the RBP register:

```
  PUSH    RBP                           ; function prologue
  MOV     RBP, RSP                      ; set up the frame pointer in RBP
  SUB     RSP, 0x10                     ; allocate a 16-byte stack frame
  MOV     QWORD PTR [RBP - 0x8], 0x2A   ; store to the stack item at offset -8
  MOV     QWORD PTR [RBP - 0x10], 0x29A ; store to the stack item at offset -16
  ...
  MOV     RAX, QWORD PTR [RBP - 0x8]    ; load the stack item at offset -8
  ADD     RSP, 0x10                     ; destroy the stack frame
  POP     RBP                           ; restore the caller's frame pointer
  RET
```

In *μCode*, the frame pointer is passed as an implicit input to functions and we can directly translate the stack item accesses as memory loads and stores. The IR initially looks as follows:

```
  uADD.8      tempaddr1.8 := rbp.8 + (-0x8)
  uSTORE.8    *(tempaddr1.8) := 0x2a
  uADD.8      tempaddr2.8 := rbp.8 + (-0x10)
  uSTORE.8    *(tempaddr2.8) := 0x29a
  ...
  uADD.8      tempaddr3.8 := rbp.8 + (-0x8)
  uLOAD.8     rax.8 := *(tempaddr3.8)
  uRET        rax.8
```

In order to generate a reasonable high-level code, the stack item accesses must be promoted to local variables (registers). Cricket performs a special optimization pass over the IR to produce the following:

```
  uMOV.8      stackitem_0x8.8 := 0x2a
  uMOV.8      stackitem_0x10.8 := 0x29a
  ...
  uMOV.8      rax.8 := stackitem_0x8.8
  uRET        rax.8
```

However, there are many restrictions that apply, otherwise, the transformation can be invalid:

- We must know the number and sizes of the stack items.

- For a stack item that we want to promote, we must be able to find *all of its uses*. If we miss an access during the promotion, it will no longer access the same variable, which will break the original semantics.
- Pointers to stack items can be used throughout the function and can enter as arguments to function calls.
- Unresolved function calls can possibly access stack items.
- When a function call uses a stack item pointer, we must be able to distinguish the extent of the accessed data. For example, if multiple stack items form a structure or an array, the pointer to the beginning of the structure looks the same as a pointer to the first item only.

6.2.4 Unsupported instruction handling

The set of supported and recognized instructions under x86 and AArch64 is not complete, as the instruction sets of both these architectures contain hundreds of different opcodes. However, a large portion of them is rarely used and a lot of instructions are obscure instructions that are exclusive to hand-written assembly. Therefore, Cricket supports an empirically useful subset of the instructions, which was constructed by analyzing the compiled outputs (both unoptimized debug builds and optimized release builds) of several large open-source projects produced by a recent LLVM compiler and counting occurrences of opcodes from full disassembly listings. This results in Cricket being able to recognize all instruction from *most* functions (more than 95% of functions in the analyzed projects are completely recognized).

Cricket also has a limited way of handling instructions that cannot be semantically analyzed. In a lot of cases, we can still analyze the disassembled instruction's inputs and outputs and mark them properly in the resulting *μCode*:

```
uASM        __asm { PAUSE }
uASM        rax.8 := __asm { LAHF }  ; the LAHF instruction stores
                                     ; output to "RAX" on x86
```

In such cases, the data-flow analysis will still work properly and even though the resulting source code will contain the `__asm` statement, in most cases it will not break the decompilation progress on other parts of the code.

6.2.5 Handling x86-specific patterns

Generated code on x86-64 is often **position independent**, which means that function calls and data references are done *relative to the current instruction's address* (program counter, PC), available in the `RIP` register. While this is a useful feature for libraries (as they can be mapped to any memory location), decompilation needs to be able to find exact reference addresses. Cricket removes references to `RIP` early during IR generation to resolve the addresses. On 32-bit x86, some instructions do not allow PC-relative addressing and a common practice to achieve it is to store the current PC in a regular register using the following code sequence:

```
; i386:
  CALL $+5              ; Call the next instruction, the "CALL"
```

```
                    ; instruction is 5 bytes long
  POP EAX           ; EAX now contains the current PC
```

Cricket detects this pattern in the function prologue and explicitly stores the PC to `EAX` when generating the IR.

The x86 architectures often make use of sub-registers, which allow accesses to smaller-than-native parts of physical registers. This also includes writes to sub-registers, which leave the rest of the register unaffected. However, there is an exception to this rule on x86-64: Writes to 32-bit registers are zero-extended to overwrite the full 64-bit registers. So the following two instructions are equal in behavior on x86-64:

```
; x86-64:
  XOR   RAX, RAX        ; zero out RAX
  XOR   EAX, EAX        ; zero out EAX, but *also* zero the rest of RAX
```

On x86-64, floating-point values and operations are most commonly handled in the XMM registers, and the ABI specifies that these registers are also used for arguments and return values. However, most compilers prefer them over the classical stack-based x87 FPU instructions, which are unsupported by Cricket.

Another irregularity on x86-64 is **stack red zones**. Functions are allowed to use 128 bytes beyond the stack pointer as their storage, assuming that they do not call other functions (as that would possibly overwrite the red zone). Cricket stack item detection includes support for red zones.

6.2.6 Handling AArch64-specific patterns

AArch64 uses fixed-width 4-byte instructions. This means that a single instruction cannot contain a full 64-bit reference and not even a full 32-bit address. The compiler works around this limitation by producing code sequences that calculate the required reference, for example:

```
; AArch64:
  ADRP    X25, #0x40000
  LDR     X22, [X25, #0x398]          ; load data from 0x40398

  ADRP    X8, #0x56000
  ADD     X8, X8, #0x4a0              ; x8 now contains 0x564a0
```

The problem with such sequences is that we need to be able to find references to code and data for various analysis, even before data-flow analysis is performed, for example for the discovery of all function beginnings in a binary. Cricket heuristically detects several of such patterns by looking for two- or three-instruction sequences when skimming the code section.

Fortunately for decompilation, AArch64 does not contain **predicated instructions**, as the 32-bit ARM architectures do. With only a few exceptions (for explicitly conditional instructions), only control-flow branches can be conditionally executed.

6.2.7 Limitations

There are some limitations in the instructions supported by Cricket at this time:

- Floating-point support is limited to XMM registers on x86-64 and unsupported on AArch64.
- Vector instructions are unsupported.

6.3 Interactive decompilation

Since Cricket is an **interactive decompiler**, users do not just give it an input and receive a decompiled output. In fact, the user can oversee, analyze, influence, correct and learn from all the stages of binary analysis, disassembly, and decompilation.

After a binary file is opened by the user, a list of all classes and methods is shown in the left sidebar of the GUI application, which can also be switched to list all available functions (not just Obj-C methods), all known *blocks* or all recognized symbols (including data). Selecting a method in the list opens its complete disassembly, with comments explaining the signature of the method, its return value and in which register is it expected and recognized *stack items*. This is shown in figure 6.3.

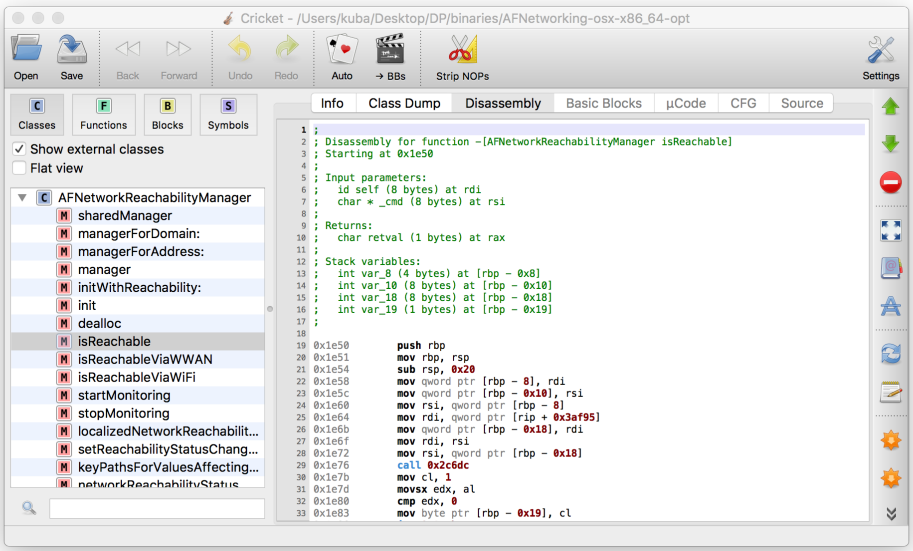


Figure 6.3: Disassembly of an Objective-C method in Cricket.

Various tasks and workflows are allowed because of this. A security researcher can get a nice overview of an unknown binary program just by looking at available classes, their methods, and signatures. It will be obvious at first sight whether the program uses some code or symbol name obfuscation. Clicking a class (instead of a method) produces a **class dump**, which lists all of its instance variables, properties, and methods, further helping the user to understand the purpose and structure of a particular class.

After basic-block detection is performed, a visual representation of the control-flow graph is displayed, which offers an indication of how complex the logic of one function is, as shown in figure 6.4.

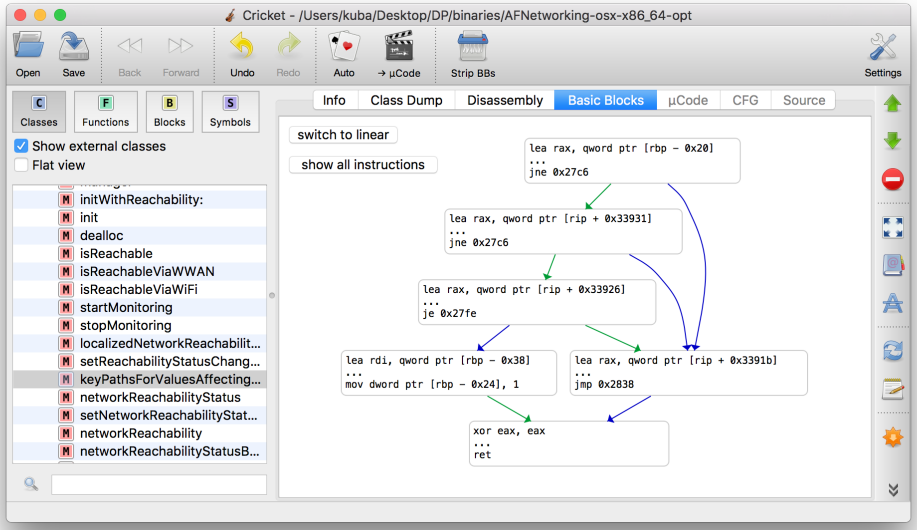


Figure 6.4: Control-flow graph visualization in Cricket.

When assembly instructions are transformed into *µCode*, the result is shown in an editor window which supports additional features, such as showing "explanations" of what were the original instructions or showing *definitions* and *uses* of inputs and outputs, which can be seen in figure 6.5. The editor also allows the user to manually perform various transformations of the IR.

6.4 Test suite

Cricket comes with a set of integration tests, which also serve as regression tests. Each test is in the form of a single Objective-C source file, usually with a single method that contains the source code which is to be compiled and then decompiled back. The test is run for each combination of the supported architectures and compile flags (optimized or not).

A test then contains **comments in a special format** that are ignored during compilation and decompilation, but which instruct the test harness to verify the results of the decompilation. For example, the `MATCH-UCODE` directive says that a regular expression match must in the generated *µCode*:

```
// MATCH-UCODE: uRET {.*}
```

This requires that a `uRET` instruction with an additional parameter must be present. If multiple `MATCH` directives are specified then the *µCode* must contain the matches in the same order as in which the directives are present. A `MATCH-UCODE-NOT` directive can be used to require a certain regular expression *does not match* any part of the generated *µCode*. This is inspired by *LLVM "lit" tests* [54].

6.4.1 Test types

There are three categories of available tests:

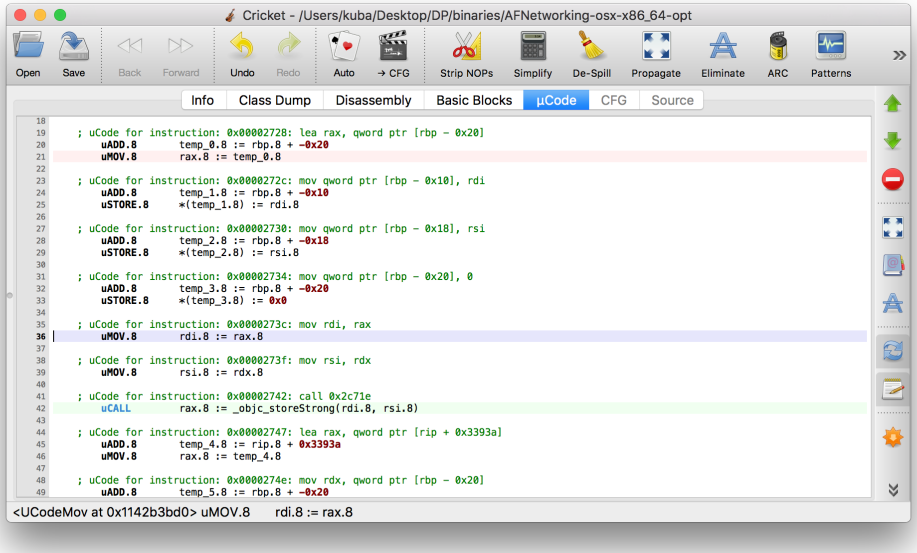


Figure 6.5: Explained generated IR with highlighted *current line* (blue), *definitions of used values* (red lines) and *uses of produced values* (green lines).

- **µCode tests** are used to test that the generated and already optimized IR contains the expected instructions. The following is an abbreviated example of a *µCode* test. Note that instead of actual register names, we use a {.*} regular expression, because we do not care about their names.

```
// MATCH-UCODE: uMUL.{4|8} {.*} := {.*} * 0x6ed
// MATCH-UCODE: uADD.{4|8} {.*} := {.*} + 0x79a1
// MATCH-UCODE: uRET {.*}
...
- (long)math_test:(long)arg {
    long a = arg;
    long b = a * 1773;
    long c = b + 31137;
    return c;
}
```

- **CFG tests** perform verification of high-level control-flow reconstructed IR. For example, a test that checks correct recognition of a *switch* statement can check for the following:

```
// MATCH-CFG: CFGSwitch
// MATCH-CFG: Case Node
// MATCH-CFG: Case Node
// MATCH-CFG: Case Node
```

- **Source-level tests** are executed on the final decompilation results. They check that the decompilation AST contains the expected expressions and statements.

  One extra feature of source-level tests is that they can attempt to *re-compile* the decompiled result again and then even run a test method against the re-compiled source. This can verify both that the generated result is

syntactically correct and compilable and that the behavior and semantics of the original code are preserved.

6.5 Source code structure

The software attachment of this thesis is an archive containing the following items:

```
.
|- Cricket.app/                    ... OS X .app application bundle
|- Cricket-source-2016-07/         ... the source code of Cricket
```

The `.app` bundle is the actual Cricket application. It does not require any specific installation, and can be run directly, or it can be copied to the user's `/Applications` directory. The source code contains the following structure:

```
.
|- analysis/            ... analysis core module
   |- arch/             ... architecture-specific modules
   |- asm/              ... binary analysis module
   |- callprototypes/   ... function prototype database module
   |- source/           ... AST-level analysis module
   |- tools/            ... various tools
   |- ucode/            ... IR analysis module
   |- binary.py         ... the "Binary" class
   |- function.py       ... the "Function" class
   |- transforms.py     ... defines available code transforms
   |- types.py          ... type system module
|- distbuilder/         ... various tools used to generate the distribution
|- externals/           ... 3rd party Python libraries
|- ide/                 ... the GUI front-end module
   |- bb/               ... basic-block graphs UI
   |- editor/           ... text editor and highlighting UI
   |- icons/            ... various graphics used by the application
   |- app.py            ... main Qt Cricket app
   |- graphs.py         ... graph plotting functions
   |- welcome.py        ... welcome window code
   |- welcome.ui        ... welcome window Qt UI
   |- window.py         ... main decompilation window code
   |- window.ui         ... main decompilation Qt UI
|- testcases/           ... test suite
|- README.md            ... usage instructions, programmer's manual
|- cricket-cli*         ... script to run the command-line version Cricket
|- cricket-gui*         ... script to run the GUI for Cricket
|- cricket-tests*       ... script to run the test suite
|- deploy.sh*           ... script to generate a .app distribution
```

6.6 Summary

We presented an implementation of *Cricket*, an Objective-C decompiler, which supports the major Apple-supported architectures, compiler, and binary format. We have shown the main design decisions and what is the architecture of the complete program, the analysis core, and the GUI.

The next chapter will show how good the decompilation results are, compare them to a competing product and discuss the goals of this thesis and how well were they achieved.

Chapter 7

Evaluation

In this chapter, we will be comparing the decompilation results of our implemented Objective-C decompiler against a competitor product on the market. For the comparison, we will use a popular open-source library.

There are several commercial and open-source decompilers available on the market, however, only very few of them are popular among reverse engineers due to their varying quality. Perhaps the most popular tool, *IDA*, has not been evaluated because of its very expensive pricing. In the end, another widely used decompiler, *Hopper*, was selected for the evaluation because it is an actively developed product and because it has an explicit support for Objective-C.

7.1 Methodology

The following software is used in the evaluation:

- The compiler used is **Apple LLVM version 7.3.0 (clang-703.0.31)**, which is included in Xcode 7.3.1 shipped in May 2016.
- **OS X El Capitan 10.11.5** as the operating system.
- **Hopper Disassembler 3.11.17** as a comparison competing product (latest version as of July 2016).
- The tested code base is the most popular Objective-C open-source project on GitHub: **AFNetworking**, with the git revision `2a53b2c3` (top of master development as of July 2016).
- The current development version of **Cricket**.

We will build two versions of the open-source library:

- A debug, unoptimized build of the library for OS X (x86-64 architecture).
- A release, fully optimized build for OS X (x86-64 architecture).

Since Hopper does not support decompilation of ARM/AArch64 code, this architecture is not included in the evaluation.

The compiled build of AFNetworking contains **20 classes** with a total of **441 methods**, which have the following properties:

- The library defines **51 distinct blocks**. Blocks are used extensively throughout the library both for internal (e.g. network callbacks are implemented with blocks) and external purposes (e.g. the API expects the user to supply a completion handler as a block).

- Most of the methods are **property accessors** generated by the compiler.
- Most of the methods consist of a **single basic block** (in the release build). This includes all of the compiler-generated property accessors, but even if we exclude them, most of the remaining methods still only have a single basic block.
- In the source code, most functions have *less that 10 lines of code*.

Due to the prevalence of methods that would be uninteresting to decompile (e.g. single-block property accessors), we will be evaluating a chosen subset of methods. This subset is selected to contain various interesting samples of individual Objective-C features, including blocks, *for-in* loops, method calls, property accesses, various control-flow structures.

Each method will be decompiled using an automatic mode in Hopper and in Cricket. The original source code and the resulting decompilation result are then qualitatively compared and the following properties are evaluated:

- How many lines of code (excluding empty lines) does the original method have? (column "L")
- How many basic blocks does the compiled function have? If it uses blocks, how many basic block do the defined blocks have? (column "BBs")
- Is the high-level control-flow reconstructed? Is it exactly the same as the source code, is it different but valid or is it incorrect? (column "CF")
- Are the blocks recognized and integrated into the decompiled function? (column "BR")
- Are the variables and data, which the function manipulates, used correctly? Is the decompiled data-flow correct or does it have a significant mistake? (column "DF")
- How many lines of code (excluding empty lines) does the decompiled result have? In case the decompilation is missing a significant part of the method, for example when it does not include a block's definition, we leave this metric blank. (column "LO")

7.2 Results

This section presents the results of the evaluation. In the following tables, method names have been shortened and class names omitted for brevity. The full list of tested methods, including their full names, signatures and source codes is available in appendix B, which also includes the full decompilation results from both Hopper and Cricket. Some methods in the table have "0" source code lines, which means that the method is compiler-generated and there is no corresponding source code.

Bold highlight means Cricket performed better in the test, *italic highlight* means we performed worse.

The following table shows the results comparing Hopper and Cricket on a **debug build of AFNetworking**:

			Hopper				Cricket			
Method name	L	BBs	CF	BR	DF	LO	CF	BR	DF	LO
sharedManager	6	3 (+1)	Valid	Fail	Fail	-	**Exact**	**Exact**	**Exact**	**7**

Method name	L	BBs	Hopper CF	Hopper BR	Hopper DF	Hopper LO	Cricket CF	Cricket BR	Cricket DF	Cricket LO
managerForDomain	4	1	Exact	-	Fail	9	Exact	-	**Valid**	*11*
managerForAddress	4	1	Exact	-	Exact	7	Exact	-	Exact	**6**
isReachable	1	3	Valid	-	Valid	7	Valid	-	Valid	*8*
stopMonitoring	4	4	Valid	-	Exact	5	**Exact**	-	*Valid*	*8*
pinnedCertificates	0	1	Exact	-	Exact	2	Exact	-	Exact	2
GET	1	1	Exact	-	Fail	12	Exact	-	**Valid**	*17*
validatesDomainName	0	1	Exact	-	Valid	2	Exact	-	**Exact**	2
setValidatesDomainName	0	1	Exact	-	Exact	2	Exact	-	Exact	2
initWithBaseURL	1	1	Exact	-	Fail	8	Exact	-	**Exact**	**1**
init	10	4	Valid	-	Valid	38	Valid	-	Valid	**24**
invalidateSession	7	1 (+4)	Exact	Fail	Fail	-	Exact	**Exact**	**Valid**	**25**
respondsToSelector	10	13	Valid	-	Valid	37	Valid	-	Valid	*44*
certificatesInBundle	7	9	Valid	-	Fail	45	**Exact**	-	**Valid**	**25**

The following table shows the results comparing Hopper and Cricket on a **release build of AFNetworking**:

Method name	L	BBs	Hopper CF	Hopper BR	Hopper DF	Hopper LO	Cricket CF	Cricket BR	Cricket DF	Cricket LO
sharedManager	6	3 (+1)	Valid	Fail	Fail	-	**Exact**	**Exact**	**Exact**	**6**
managerForDomain	4	1	Exact	-	Valid	10	Exact	-	Valid	**6**
managerForAddress	4	1	Exact	-	Fail	6	Exact	-	**Valid**	**5**
isReachable	1	3	Valid	-	Valid	7	Valid	-	Valid	*10*
stopMonitoring	4	4	Valid	-	Valid	9	**Exact**	-	Valid	9
pinnedCertificates	0	1	Exact	-	Exact	2	Exact	-	Exact	2
GET	1	1	Exact	-	Exact	10	Exact	-	*Valid*	**3**
validatesDomainName	0	1	Exact	-	Valid	2	Exact	-	**Exact**	2
setValidatesDomainName	0	1	Exact	-	Exact	2	Exact	-	Exact	2
initWithBaseURL	1	1	Exact	-	Exact	2	Exact	-	Exact	**1**
init	10	3	Valid	-	Valid	24	**Exact**	-	**Exact**	**19**
invalidateSession	7	1 (+4)	Exact	Fail	Fail	-	Exact	**Exact**	**Valid**	**14**
respondsToSelector	10	11	Fail	-	Valid	-	**Valid**	-	Valid	**31**
certificatesInBundle	7	9	Valid	-	Valid	38	**Exact**	-	Valid	**22**

7.3 Discussion

Let us now analyze the results from the tables above and the outputs in appendix B. The first important thing to notice is that Hopper completely fails to analyze functions that contain blocks within them. Although it supports decompiling them, in this test project, any function containing a block completely confused the decompilation, which even produced invalid statements, such as `dispatch_async(..., __NSConcreteStackBlock);`. This is one area where Cricket performs significantly better, and in all the evaluates method, blocks were always recognized correctly.

Secondly, Hopper sometimes fails to recognize the high-level control-flow structures, and in these cases, it gives up completely and produces a flat function with *goto* statements instead of all control flow. Surprisingly, this happens both for debug and release builds. Cricket always recognizes *at least some* of the control-flow statements, inserting only individual *gotos*. On top of that, Cricket's

heuristic for *early-returns* produces a much more readable code, which in most cases matches the control-flow of the original source code.

The number of cases where Hopper incorrectly assigns a variable or performs a calculation on a wrong variable, is surprisingly high. Even more unexpectedly, this happens more often in debug builds rather than release builds. This reason for that seems to be that release builds often make much more use of registers instead of stack items, and Hopper is very imprecise when dealing with local variables stored on the stack. Explicitly, when an address of a stack variable is used as a parameter into a function call, Hopper tends to produce wrong results. This often leads to further errors in the output.

Cricket, on the other hand, supports this behavior better and in most cases, it recognizes the data-flow of the methods correctly.

In terms of output line count, both Hopper and Cricket produce varying results. Simple and short methods are usually decompiled into just a few lines of code and are very readable. Larger methods tend to be more confusing to read because of the missing local variable names.

Overall, we can conclude that Hopper decompiles significantly more *correct output*, which is very important for manual reading of the decompiled output.

Chapter 8

Conclusion

The goal of the thesis was to create an interactive tool for decompilation of Objective-C applications, with the purpose to support manual work with unknown binaries (e.g. malware analysis). The implementation of our decompiler called *Cricket* meets this goal and provides very interesting results in comparison with current state-of-the-art competitor products. We have shown that we can produce much more readable, concise and correct decompilation outputs for typical Objective-C programs.

The comparison shows that out decompiler can correctly analyze cases where a major competitor fails. We support complex Objective-C structures, such as blocks and *for-in* statements, which is a unique feature of our decompiler and which greatly improves the results.

Because Cricket is an interactive GUI tool which shows the progress and individual steps of the decompilation, it can be also used as a learning and experimental environment. Students can learn the principles of decompiling, but also how code transformations and common compiler theory algorithms work in general.

Secondarily, this thesis provides a generic description of how Objective-C binary programs are structured and how these structures can be recognized into high-level language constructs. This can serve as valuable input for further research, for example, protection scheme design (DRM).

8.1 Future work

While the implementation of the presented decompiler already shows a lot of promise, there are still areas which could be improved or new features that could be added. These range from natural improvements and enhancements to completely new use cases and work-flows:

- **Porting to other platforms**. Cricket currently only works on OS X and allows analyzing Mach-O binary formats. To make the tool available to a larger general public audience, ports to Linux and Windows and other binary formats (ELF, COFF) should be done.
- **Supporting more instructions**. Cricket only has a limited understanding of floating-point instructions and no support for vector and other less frequently used instructions.

- Supporting **more CPU architectures**. Cricket currently only supports i386, x86-64 and AArch64 architectures.
- Adding support for **other programming languages**, for example, C and C++ are natural candidates as a lot of software for OS X and iOS is written in these languages.
- **Obfuscated code and manually written assembly** are currently not supported by Cricket. Extending it to allow decompilation of such programs would be a major change, but it would allow new uses of the tool.
- Reworking the graphical interface to be a **full binary analysis IDE**. Cricket is currently not suitable for other binary analysis tasks other than decompilation.

Chapter 9

References

[1] Collberg, Christian and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Upper Saddle River, NJ: Addison-Wesley, 2010. Print. ISBN 0321549252.

[2] Intel® 64 and IA-32 Architectures Software Developer Manuals. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[3] *IDA – Interactive Disassembler* by Hex-Rays. `https://www.hex-rays.com/products/ida/`.

[4] *Hopper* by Cryptic Apps. `http://www.hopperapp.com/`.

[5] *Capstone – The Ultimate Disassembler*. `http://www.capstone-engine.org/`.

[6] *OllyDbg*. `http://ollydbg.de/`.

[7] *x64dbg*. `http://x64dbg.com/`.

[8] Cifuentes, Cristina. *Reverse Compilation Techniques*. PhD thesis. Queensland University of Technology, Australia, 1994. Print.

[9] *Hex-Rays Decompiler*. `https://www.hex-rays.com/products/decompiler/index.shtml`.

[10] *Clang: a C language family frontend for LLVM*. `http://clang.llvm.org`.

[11] *The LLVM Compiler Infrastructure*. `http://llvm.org`.

[12] *Xcode* by Apple Inc. `https://developer.apple.com/xcode/`.

[13] *TIOBE index for Objective-C*, `http://www.tiobe.com/tiobe_index?page=Objective-C`.

[14] *Programming with Objective-C*, `https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html`.

[15] *Concepts in Objective-C Programming*, https://developer.apple.com/library/mac/documentation/General/Conceptual/CocoaEncyclopedia/Introduction/Introduction.html.

[16] *Objective-C Blocks vs. C++0x Lambdas: Fight!*, https://www.mikeash.com/pyblog/friday-qa-2011-06-03-objective-c-blocks-vs-c0x-lambdas-fight.html.

[17] *objc4* project by at opensource.apple.com, http://opensource.apple.com/source/objc4/.

[18] *Objective-C Runtime Reference*, https://developer.apple.com/library/mac/documentation/Cocoa/Reference/ObjCRuntimeRef/index.html.

[19] *OS X ABI Mach-O File Format Reference*, https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachORuntime/index.html.

[20] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham: *Differentiating Code from Data in x86 Binaries*. http://www.utdallas.edu/~kxh060100/wartell-pkdd11.pdf.

[21] *System V Application Binary Interface*, AMD64 Architecture Processor Supplement. http://www.x86-64.org/documentation/abi.pdf.

[22] Jhala, R., & Bosschere, K. D. *Compiler construction: 22nd International Conference, CC 2013*. Berlin: Springer, 2013.

[23] Kakde, O. G. *Comprehensive compiler design*. Bangalore: Laxmi Publications, 2006.

[24] *Function arguments statistics*, Dennis Yurichev. http://yurichev.com/blog/args_stat/.

[25] *Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf.

[26] McCabe, Thomas J. *Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric*. National Institute of Standards and Technology (NIST), 1982. NIST Publication 500-235.

[27] Beth, Bradley. *A Comparison of Similarity Techniques for Detecting Source Code Plagiarism*. The University of Texas at Austin, 2014. https://www.cs.utexas.edu/~bbeth/files/AComparisonOfSimilarityTechniquesForDetectingSourceCodePlagiarism.pdf.

[28] Appel, Andrew W. *Modern Compiler Implementation in ML*. Cambridge University Press, 2008. ISBN 0-521-60764-7.

[**29**] Boissinot, B. et al. *Fast Liveness Checking for SSA-Form Programs*. CGO 2008. http://www.rw.cdl.uni-saarland.de/~grund/papers/cgo08-liveness.pdf.

[**30**] *LLVM Language Reference Manual*, LLVM Project. http://llvm.org/docs/LangRef.html.

[**31**] Van Emmerik, Michael. *Static Single Assignment for Decompilation*. University of Queensland, 2007. PhD Thesis.

[**32**] Cytron, Ron; Ferrante, Jeanne; Rosen, Barry K.; Wegman, Mark N. & Zadeck, F. Kenneth. *Efficiently computing static single assignment form and the control dependence graph*. ACM Transactions on Programming Languages and Systems 13, 1991.

[**33**] *The "runtime.h" file from Objective-C runtime project*. https://opensource.apple.com/source/objc4/objc4-493.9/runtime/runtime.h

[**34**] *Key-Value Coding Programming Guide*, a documentation page from developer.apple.com. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/KeyValueCoding/Articles/KeyValueCoding.html.

[**35**] *Introduction to Key-Value Observing Programming Guide*, a documentation page from developer.apple.com. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/KeyValueObserving/KeyValueObserving.html.

[**36**] *Method Swizzling*, article in the NSHipster magazine. http://nshipster.com/method-swizzling/.

[**37**] *The "class-dump" project on GitHub*. https://github.com/nygard/class-dump.

[**38**] *Objective-C Runtime Programming Guide, Type Encodings*, a documentation page from developer.apple.com. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtTypeEncodings.html.

[**39**] *An Illustrated History of objc_msgSend*. http://sealiesoftware.com/msg/x86-mavericks.html.

[**40**] *Advanced Memory Management Programming Guide*, a documentation page from developer.apple.com. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html

[**41**] *Memory Management Programming Guide for Core Foundation*, a documentation page from developer.apple.com. https://developer.apple.com/library/mac/documentation/CoreFoundation/Conceptual/CFMemoryMgmt/Concepts/Ownership.html

[42] *Objective-C Automatic Reference Counting (ARC)*, Clang documentation. http://clang.llvm.org/docs/AutomaticReferenceCounting.html.

[43] *Objective-C Literals*, Clang documentation. http://clang.llvm.org/docs/ObjectiveCLiterals.html.

[44] *Cocoa Core Competencies, Enumeration*, a documentation page from developer.apple.com. https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/Enumeration.html.

[45] *Python programming language*, https://www.python.org.

[46] *Homebrew, the missing package manager for OS X*, http://brew.sh.

[47] *Qt*, https://www.qt.io.

[48] *PyQt*, https://www.riverbankcomputing.com/software/pyqt/intro.

[49] *Capstone*, http://www.capstone-engine.org.

[50] *diStorm3*, https://github.com/gdabah/distorm.

[51] *pycparser*, https://github.com/eliben/pycparser.

[52] *Graphviz*, http://www.graphviz.org.

[53] *"Strong" stack protection for GCC*, https://lwn.net/Articles/584225/.

[54] *lit – LLVM Integrated Tester*, http://llvm.org/docs/CommandGuide/lit.html.

List of Abbreviations

- ABI — Application Binary Interface, instruction-level contract how compiled code needs to behave in order to be compatible with other code on the same platform
- API — Application Programming Interface, source-code level contracts that libraries provide and programmers are free to use in their programs
- ARC — Automatic Reference Counting
- AST — Abstract Syntax Tree, a compiler-specific representation of source code during early stages of compilation
- BB — Basic Block
- CFG — Control Flow Graph
- CLI — Command-line Interface
- CPU — Central Processing Unit
- DAG — Directed Acyclic Graph
- DLL — Dynamically Linked Library
- DRM — Digital Rights Management, a way of protection software and media from unauthorized use and reverse engineering
- ELF — Executable and Linkable Format, the most common format of compiled programs on Linux and other Unix systems
- GCC — GNU Compiler Collection, the major compiler for C and C++ on Unix systems
- GCD — Grand Central Dispatch, an Apple-provided system library for effective threading and parallelism
- GUI — Graphical User Interface
- IDE — Integrated Development Environment
- IR — Intermediate Representation, a way of representing the code being compiled that is neither high-level (AST) nor low-level (machine instructions)

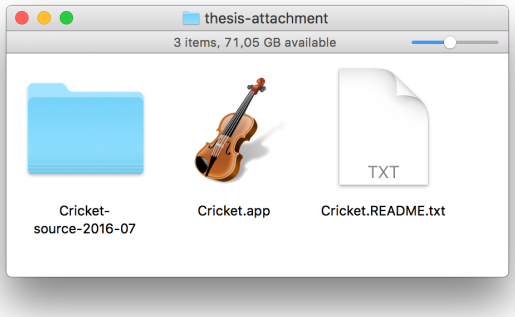
- KVO — Key-value Observation
- LLVM — formerly “Low-level Virtual Machine”, nowadays not an acronym anymore, but the name of the project
- LTO — Link-time Optimizations.
- MRR — Manual Retain-Release
- NOP — No Operation, an instruction which doesn’t do anything
- Obj-C — Objective-C
- PC — Program Counter
- SSA — Static Single Assignment, a form of IR where each variable is assigned exactly once

Appendix A: Cricket User's Manual

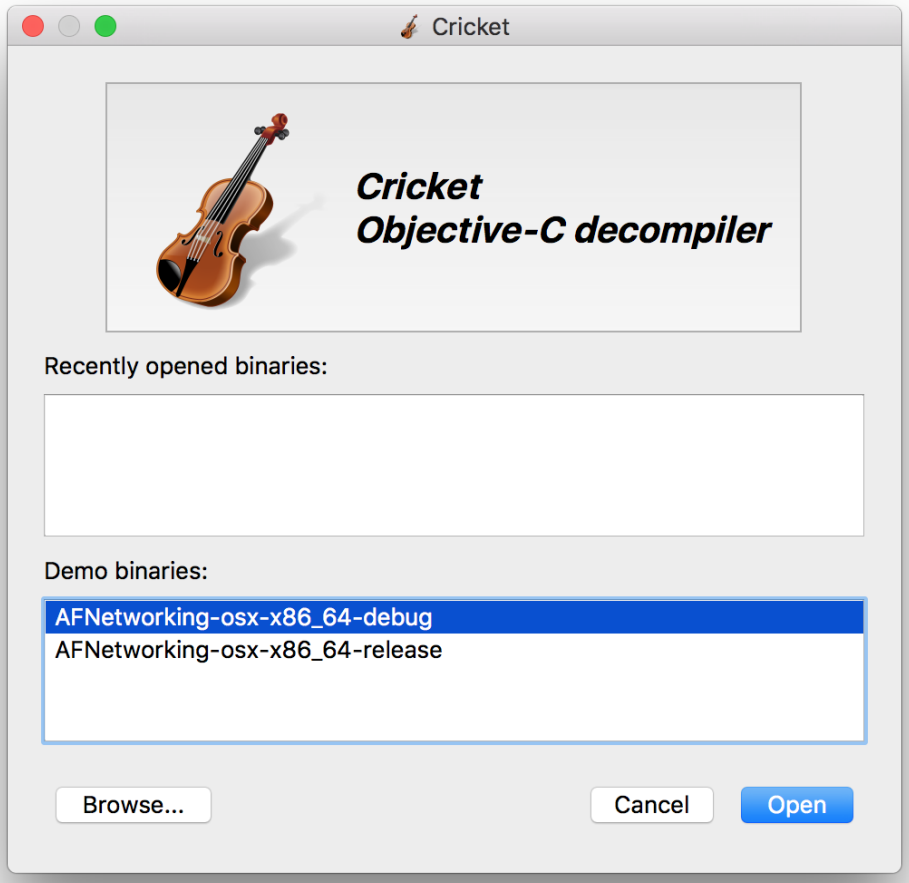
Running Cricket

Cricket requires OS X version 10.10 (Yosemite) or higher and Xcode version 6 or higher installed.

Cricket is provided as an OS X `.app` application bundle. To run it, simply double-click the Cricket's icon in the folder:



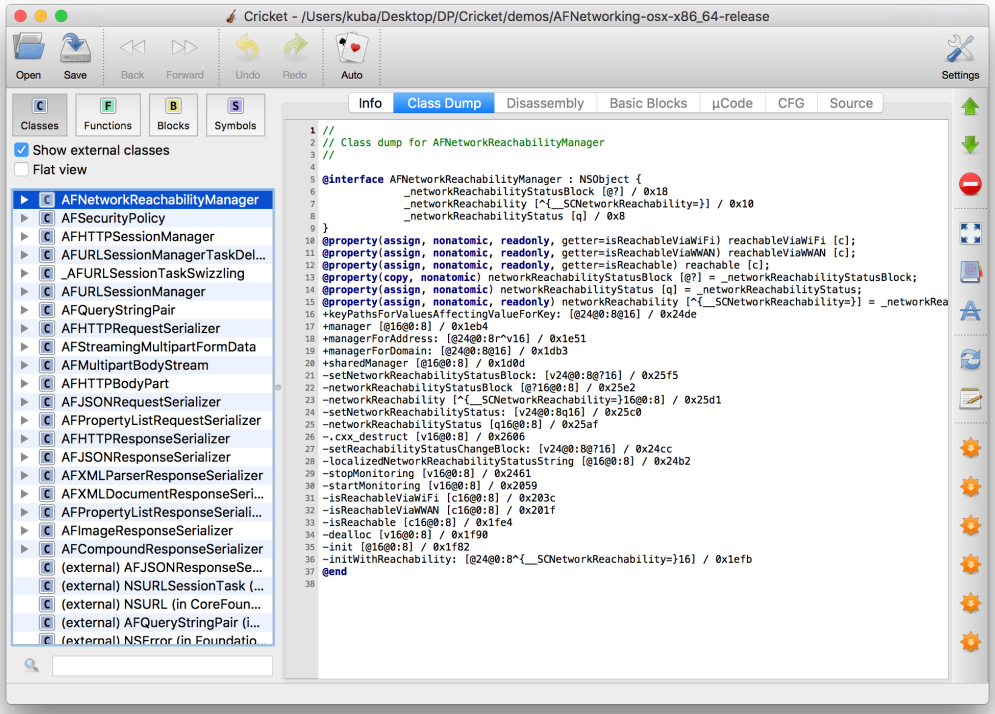
Cricket's welcome screen will be shown:



This window allows you to select which binary file to open in Cricket: You can either select a recently-opened file, one of the demo binaries, or press the "Browse..." button to open a file selection dialog.

Main window

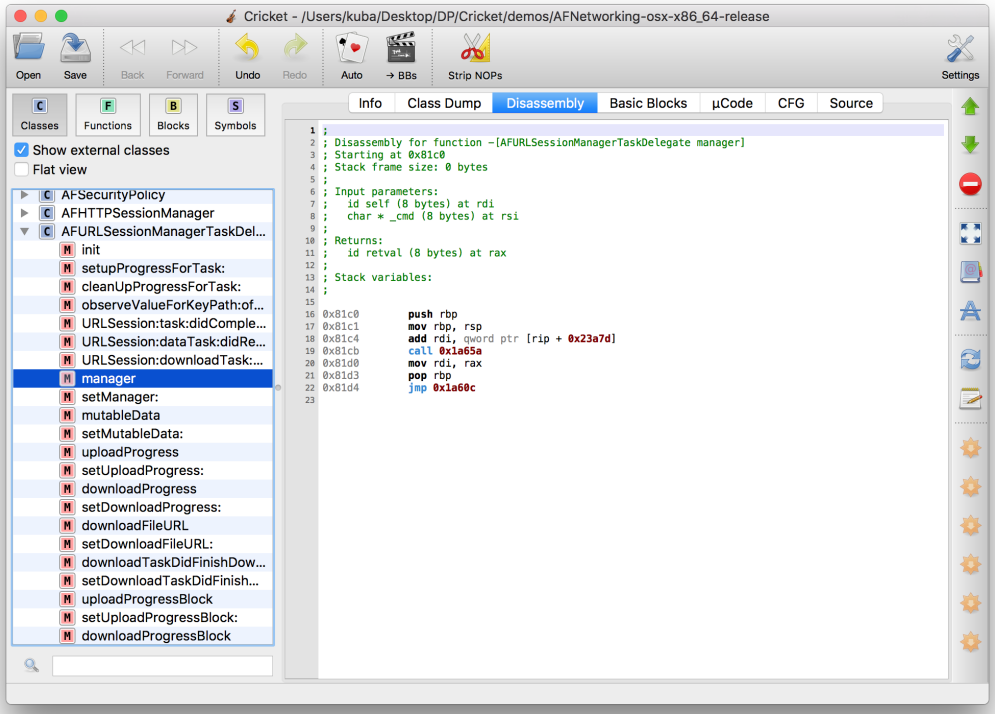
After a binary is selected, the main Cricket's analysis window will be shown:



The left bar shows a list of all classes and methods found in the binary. You can switch to view all functions (including C functions) using the "Functions" button. The "Blocks" button switches the list to show all recognized blocks. "Symbols" lists all symbols available in the binary.

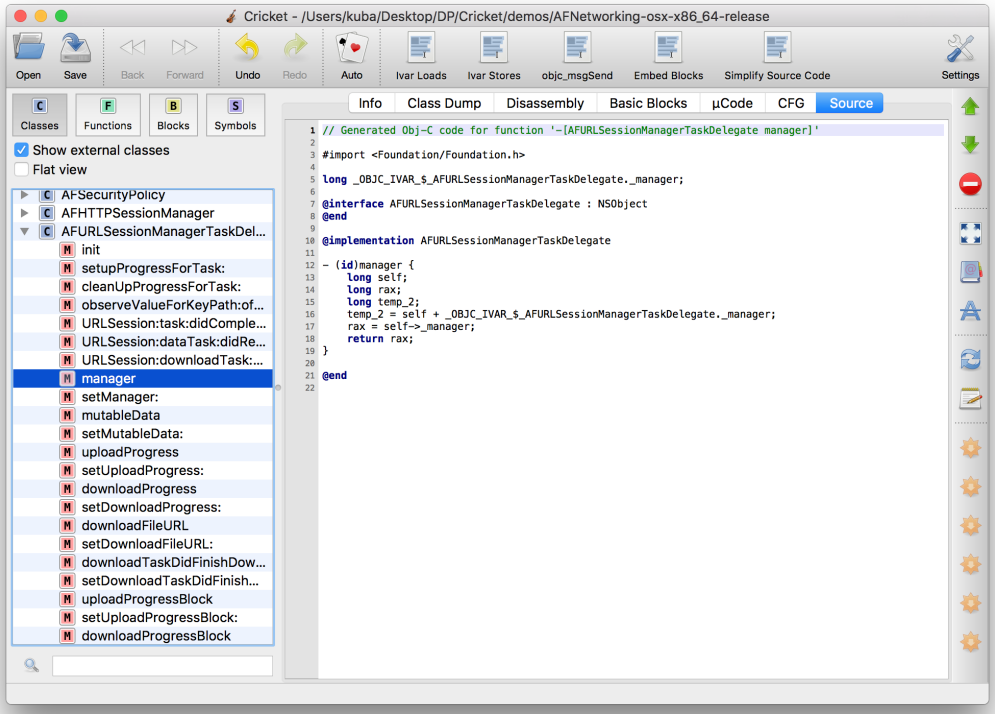
In the class view you can use the "Show external classes" checkbox to also include externally-linked classes in the list. "Flat view" switches the list from hierarchical (where methods are shown as subitems to classes) to a simple flat list.

The next step is to select a method to decompile from the left-side list. After a method is selected, its full disassembly listing will be shown:



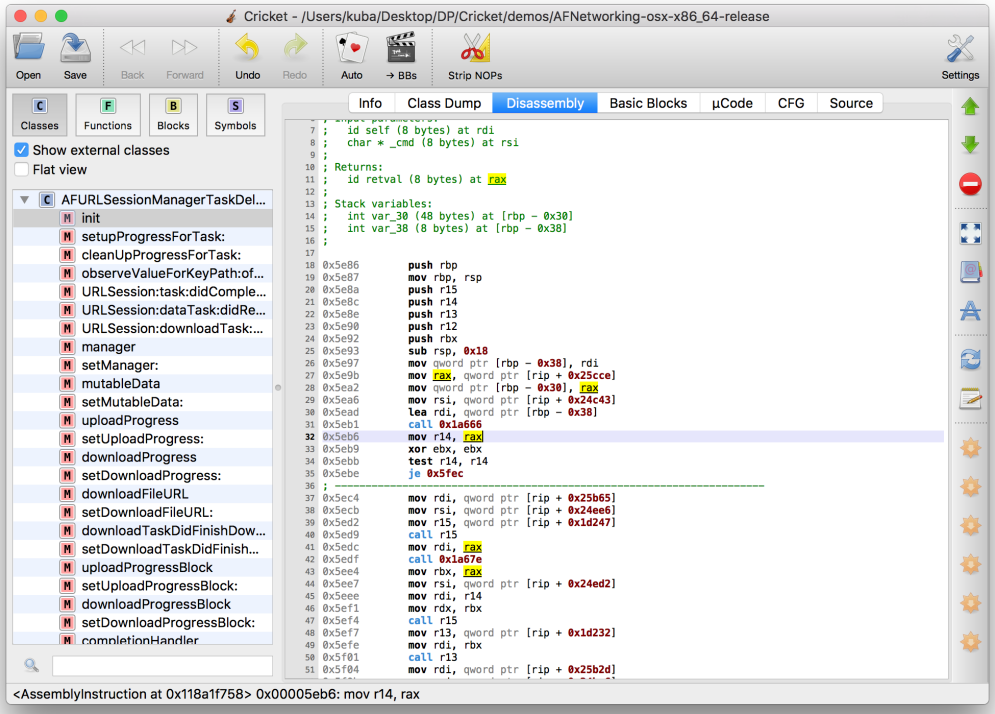
Automatic decompilation

The easiest way to decompile a function is to press the “Auto” button in the toolbar. This will invoke a fully automatic decompilation mode, which analyzes all referenced blocks and the function itself, and performs all the stages of decompilation and display the decompilation result:

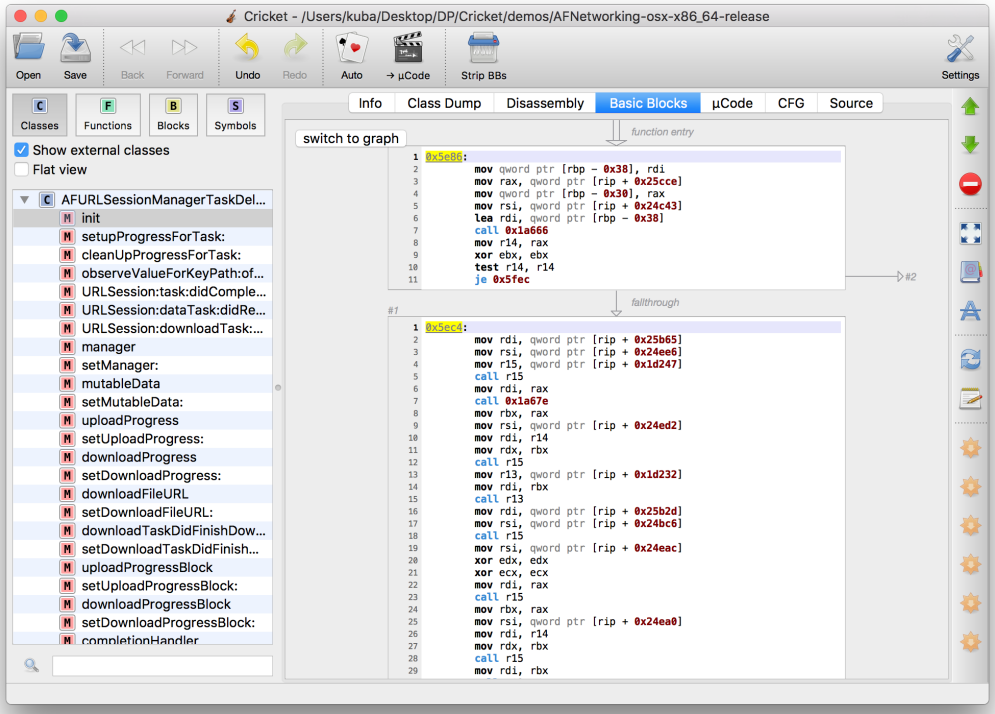


Manual decompilation

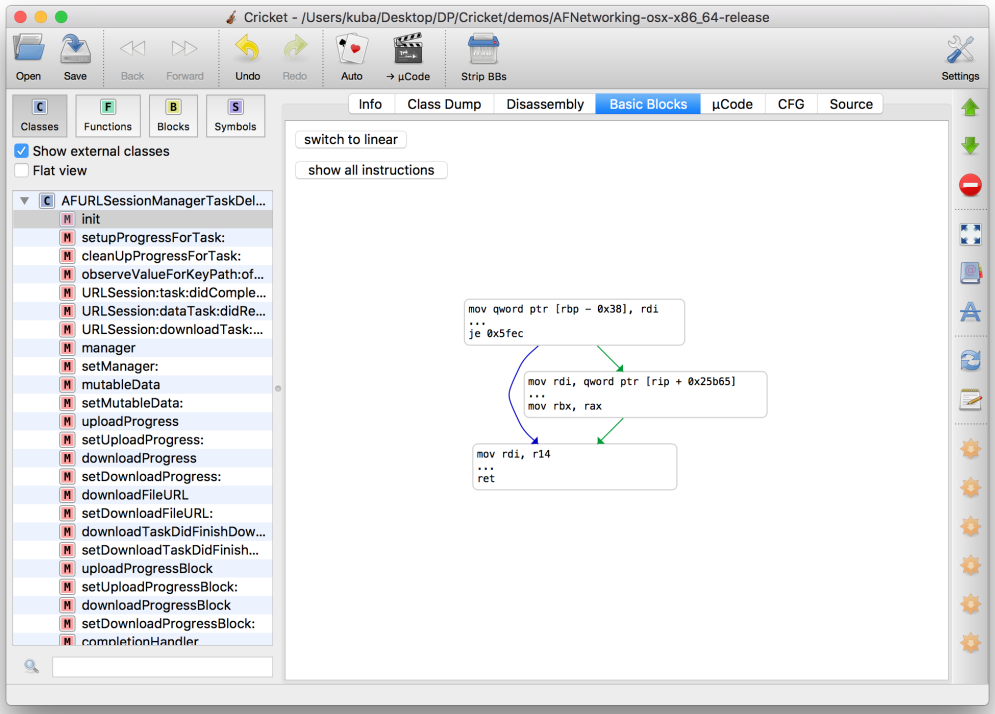
To perform manual decompilation and to inspect the intermediate representations, start by selecting a method from the list, which will show a disassembly listing. The disassembly text window support color highlighting and also when you select a register, instruction name or a constant, all uses of the same item will be highlighted:



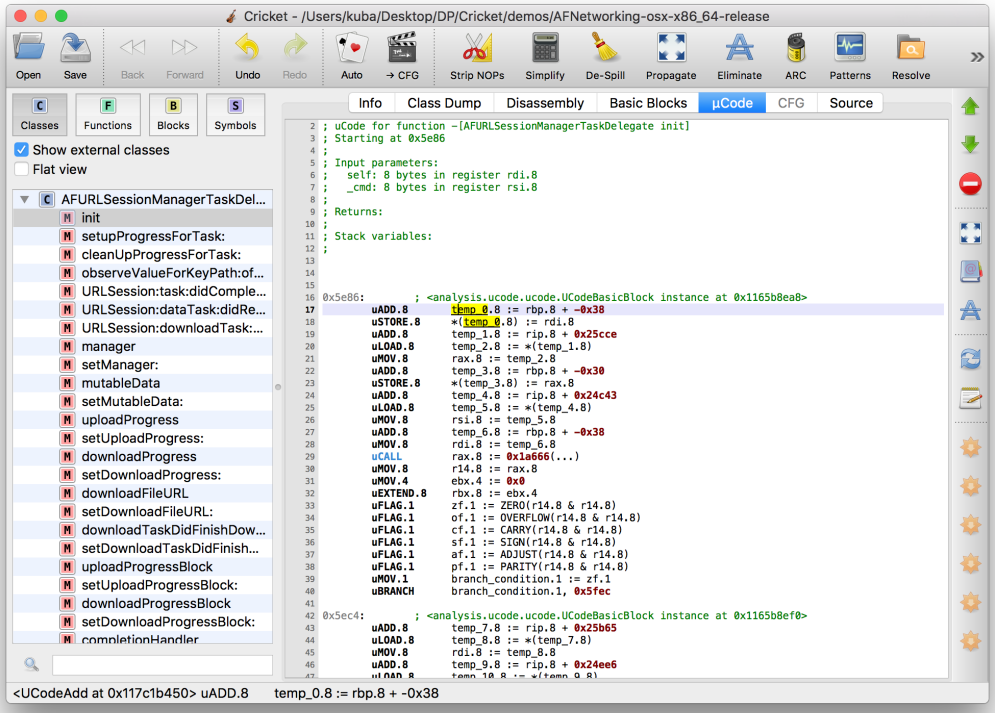
If the function contains any NOP instructions, you can click the “Strip NOPs” button to remove them. Clicking the “-> BBs” button will perform basic block detection and removal of low-level architecture specific idioms (function prologue and epilogue), and show the results on the next page, “Basic Blocks”:



This lists all the basic blocks in the function and also indicates conditional and unconditional jumps with arrows. You can click the "switch to graph" button to show the basic block graph in a visual form:



The “Strip BBs” button will remove basic blocks that are unnecessary (e.g. compiler-generated stack overflow protection). Clicking the “-> uCode” button will transform the basic blocks and machine instructions into uCode:

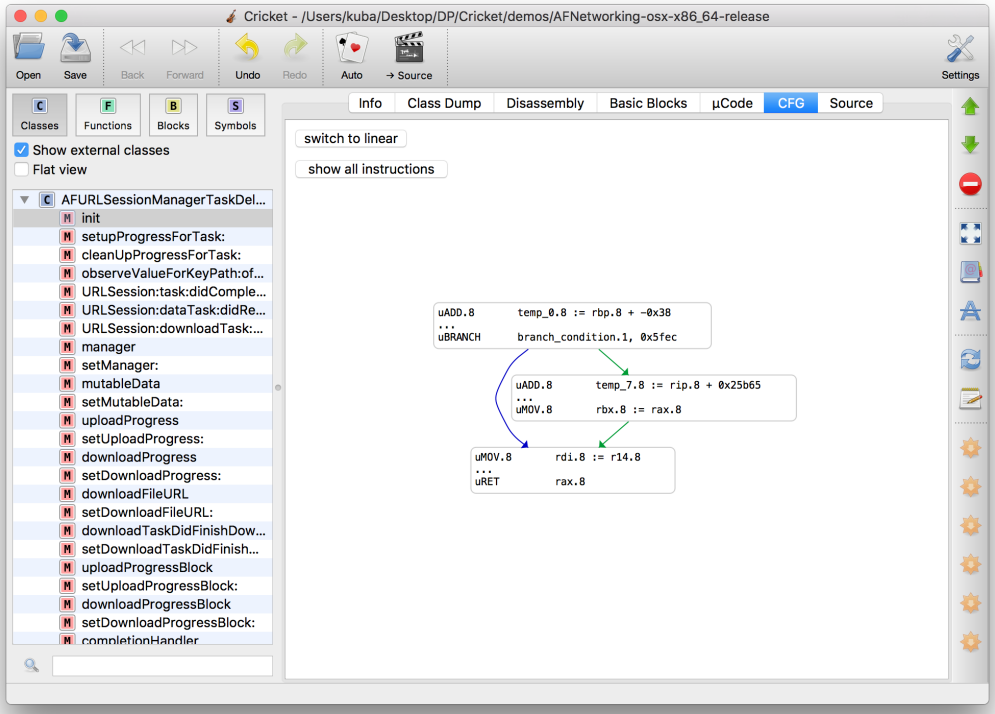


In the uCode tab, there are several possible transformation you can perform:

- “Strip NOPs” removes all NOP instruction (they can be generated by other transformations).
- “Simplify” performs constant folding and other expression simplification.
- “De-Spill” promotes stack variables to local registers.
- “Propagate” tries to propagate value definitions to their uses.
- “Eliminate” tried to remote instructions which are unnecessary.
- “ARC” performs removal of automatic reference counting function calls.
- “Patterns” matches several useful instruction patterns and transforms them into simpler ones.
- “Resolve” tries to resolve function parameters and types.

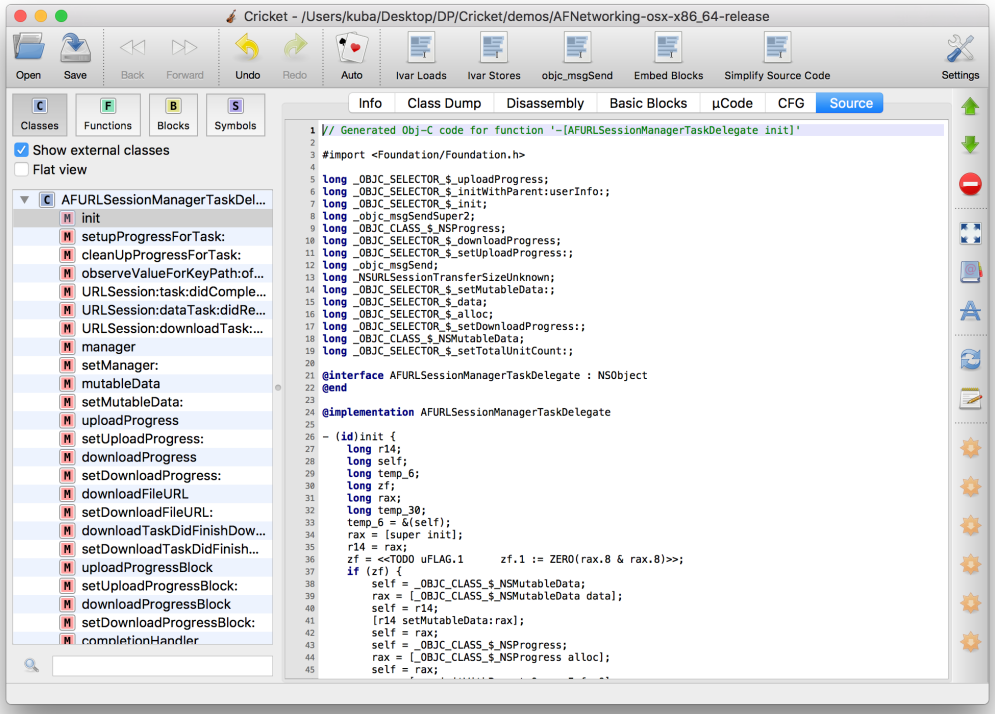
All of these can be performed on the whole uCode by clicking the button in the toolbar. To perform the transformation on a single instruction, select the appropriate item from the “Instruction” menu bar.

Clicking the “-> CFG” button, a control flow reconstruction tab will be shown:



The task of manual decompilation is to collapse multiple nodes into simpler control-flow statements. If you click a basic block, you will see what patterns match this basic block in the "CFG" menu bar or by the orange icons in the right bottom toolbar. Applying this match will simplify the CFG.

Once the CFG is a single node (all control-flow patterns are recognized), you can click the "-> Source" button to generate the AST:



More transformations are available on the AST level:

- "Ivar Loads" will replace loads including ivar offsets with an explicit ivar access.
- "Ivar Stores" will replace stored including ivar offsets with an explicit ivar access.
- "objc_msgSend" matches C calls to objc_msgSend and replaces them with the Objective-C syntax.
- "Embed Blocks" includes block bodies into the function.
- "Simplify Source Code" will perform various simplification of the AST including removal of empty statements.

Appendix B: Evaluation Source Codes and Results

Original source codes

```
1  // +[AFNetworkReachabilityManager sharedManager]
2  + (instancetype)sharedManager {
3      static AFNetworkReachabilityManager *_sharedManager = nil;
4      static dispatch_once_t onceToken;
5      dispatch_once(&onceToken, ^{
6          _sharedManager = [self manager];
7      });
8
9      return _sharedManager;
10 }
```

```
1  // +[AFNetworkReachabilityManager managerForDomain:]
2  + (instancetype)managerForDomain:(NSString *)domain {
3      SCNetworkReachabilityRef reachability =
       SCNetworkReachabilityCreateWithName(kCFAllocatorDefault, [domain
       UTF8String]);
4
5      AFNetworkReachabilityManager *manager = [[self alloc] initWithReachability:
       reachability];
6
7      CFRelease(reachability);
8
9      return manager;
10 }
```

```
1  // +[AFNetworkReachabilityManager managerForAddress:]
2  + (instancetype)managerForAddress:(const void *)address {
3      SCNetworkReachabilityRef reachability =
       SCNetworkReachabilityCreateWithAddress(kCFAllocatorDefault, (const struct
       sockaddr *)address);
4      AFNetworkReachabilityManager *manager = [[self alloc] initWithReachability:
       reachability];
5
6      CFRelease(reachability);
7
8      return manager;
```

```
}
```

```
// -[AFNetworkReachabilityManager isReachable]
- (BOOL)isReachable {
    return [self isReachableViaWWAN] || [self isReachableViaWiFi];
}
```

```
// -[AFNetworkReachabilityManager stopMonitoring]
- (void)stopMonitoring {
    if (!self.networkReachability) {
        return;
    }

    SCNetworkReachabilityUnscheduleFromRunLoop(self.networkReachability,
    CFRunLoopGetMain(), kCFRunLoopCommonModes);
}
```

```
// -[AFSecurityPolicy pinnedCertificates]
// No source-code, autogenerated.
```

```
// -[AFHTTPSessionManager GET:parameters:success:failure:]
- (NSURLSessionDataTask *)GET:(NSString *)URLString
                   parameters:(id)parameters
                      success:(void (^)(NSURLSessionDataTask *task, id
    responseObject))success
                      failure:(void (^)(NSURLSessionDataTask *task, NSError *
    error))failure
{

    return [self GET:URLString parameters:parameters progress:nil success:
    success failure:failure];
}
```

```
// -[AFSecurityPolicy validatesDomainName]
// No source-code, autogenerated.
```

```
// -[AFSecurityPolicy setValidatesDomainName:]
// No source-code, autogenerated.
```

```
// -[AFHTTPSessionManager initWithBaseURL:]
- (instancetype)initWithBaseURL:(NSURL *)url {
    return [self initWithBaseURL:url sessionConfiguration:nil];
}
```

```objc
// -[AFURLSessionManagerTaskDelegate init]
- (instancetype)init {
    self = [super init];
    if (!self) {
        return nil;
    }

    self.mutableData = [NSMutableData data];
    self.uploadProgress = [[NSProgress alloc] initWithParent:nil userInfo:nil
    ];
    self.uploadProgress.totalUnitCount = NSURLSessionTransferSizeUnknown;

    self.downloadProgress = [[NSProgress alloc] initWithParent:nil userInfo:
    nil];
    self.downloadProgress.totalUnitCount = NSURLSessionTransferSizeUnknown;
    return self;
}
```

```objc
// -[AFURLSessionManager invalidateSessionCancelingTasks:]
- (void)invalidateSessionCancelingTasks:(BOOL)cancelPendingTasks {
    dispatch_async(dispatch_get_main_queue(), ^{
        if (cancelPendingTasks) {
            [self.session invalidateAndCancel];
        } else {
            [self.session finishTasksAndInvalidate];
        }
    });
}
```

```objc
// -[AFURLSessionManager respondsToSelector:]
- (BOOL)respondsToSelector:(SEL)selector {
    if (selector == @selector(URLSession:task:willPerformHTTPRedirection:
    newRequest:completionHandler:)) {
        return self.taskWillPerformHTTPRedirection != nil;
    } else if (selector == @selector(URLSession:dataTask:didReceiveResponse:
    completionHandler:)) {
        return self.dataTaskDidReceiveResponse != nil;
    } else if (selector == @selector(URLSession:dataTask:willCacheResponse:
    completionHandler:)) {
        return self.dataTaskWillCacheResponse != nil;
    } else if (selector == @selector(
    URLSessionDidFinishEventsForBackgroundURLSession:)) {
        return self.didFinishEventsForBackgroundURLSession != nil;
    }

    return [[self class] instancesRespondToSelector:selector];
}
```

```objc
// +[AFSecurityPolicy certificatesInBundle:]
+ (NSSet *)certificatesInBundle:(NSBundle *)bundle {
    NSArray *paths = [bundle pathsForResourcesOfType:@"cer" inDirectory:@"."];

    NSMutableSet *certificates = [NSMutableSet setWithCapacity:[paths count]];
```

```
    for (NSString *path in paths) {
        NSData *certificateData = [NSData dataWithContentsOfFile:path];
        [certificates addObject:certificateData];
    }

    return [NSSet setWithSet:certificates];
}
```

Hopper decompilation results for debug build

```
// +[AFNetworkReachabilityManager sharedManager]
void * +[AFNetworkReachabilityManager sharedManager](void * self, void * _cmd)
     {
    objc_storeStrong(0x0, __NSConcreteStackBlock);
    if (*_sharedManager.onceToken != 0xffffffffffffffff) {
            dispatch_once(_sharedManager.onceToken, 0x0);
    }
    objc_storeStrong(0x0, 0x0);
    rax = objc_retainAutoreleaseReturnValue(*_sharedManager._sharedManager);
    return rax;
}
```

```
// +[AFNetworkReachabilityManager managerForDomain:]
void * +[AFNetworkReachabilityManager managerForDomain:](void * self, void *
    _cmd, void * arg2) {
    objc_storeStrong(0x0, arg2);
    var_20 = SCNetworkReachabilityCreateWithName(*_kCFAllocatorDefault, [
    objc_retainAutorelease(0x0) UTF8String]);
    var_28 = [[self alloc] initWithReachability:var_20];
    CFRelease(var_20);
    var_48 = [var_28 retain];
    objc_storeStrong(var_28, 0x0);
    objc_storeStrong(0x0, 0x0);
    rax = [var_48 autorelease];
    return rax;
}
```

```
// +[AFNetworkReachabilityManager managerForAddress:]
int +[AFNetworkReachabilityManager managerForAddress:](int arg0, int arg1, int
     arg2) {
    var_20 = SCNetworkReachabilityCreateWithAddress(*_kCFAllocatorDefault,
    arg2, arg2);
    var_28 = [[arg0 alloc] initWithReachability:var_20];
    CFRelease(var_20);
    var_38 = [var_28 retain];
    objc_storeStrong(var_28, 0x0);
    rax = [var_38 autorelease];
    return rax;
}
```

```
// -[AFNetworkReachabilityManager isReachable]
char -[AFNetworkReachabilityManager isReachable](void * self, void * _cmd) {
    var_8 = self;
    var_19 = 0x1;
    if (sign_extend_64([var_8 isReachableViaWWAN]) == 0x0) {
            var_19 = sign_extend_64([var_8 isReachableViaWiFi]) != 0x0 ? 0x1 :
     0x0;
    }
    rax = sign_extend_64(var_19 & 0x1 & 0xff);
    return rax;
}
```

```
// -[AFNetworkReachabilityManager stopMonitoring]
void -[AFNetworkReachabilityManager stopMonitoring](void * self, void * _cmd)
    {
    var_8 = self;
    if ([var_8 networkReachability] != 0x0) {
            SCNetworkReachabilityUnscheduleFromRunLoop([var_8
    networkReachability], CFRunLoopGetMain(), *_kCFRunLoopCommonModes);
    }
    return;
}
```

```
// -[AFSecurityPolicy pinnedCertificates]
void * -[AFSecurityPolicy pinnedCertificates](void * self, void * _cmd) {
    rax = self->_pinnedCertificates;
    return rax;
}
```

```
// -[AFHTTPSessionManager GET:parameters:success:failure:]
void * -[AFHTTPSessionManager GET:parameters:success:failure:](void * self,
    void * _cmd, void * arg2, void * arg3, void * arg4, void * arg5) {
    objc_storeStrong(0x0, arg2);
    objc_storeStrong(0x0, arg3);
    objc_storeStrong(0x0, arg4);
    objc_storeStrong(0x0, arg5);
    *rsp = 0x0;
    var_78 = [[self GET:0x0 parameters:0x0 progress:0x0 success:0x0 failure:
    stack[2031]] retain];
    objc_storeStrong(0x0, 0x0);
    objc_storeStrong(0x0, 0x0);
    objc_storeStrong(0x0, 0x0);
    objc_storeStrong(0x0, 0x0);
    rax = [var_78 autorelease];
    return rax;
}
```

```
// -[AFSecurityPolicy validatesDomainName]
char -[AFSecurityPolicy validatesDomainName](void * self, void * _cmd) {
    rax = sign_extend_64(self->_validatesDomainName);
    return rax;
```

```
}
```

```
// -[AFSecurityPolicy setValidatesDomainName:]
void -[AFSecurityPolicy setValidatesDomainName:](void * self, void * _cmd,
    char arg2) {
    self->_validatesDomainName = arg2;
    return;
}
```

```
// -[AFHTTPSessionManager initWithBaseURL:]
void * -[AFHTTPSessionManager initWithBaseURL:](void * self, void * _cmd, void
     * arg2) {
    objc_storeStrong(var_18, arg2);
    rax = [self initWithBaseURL:0x0 sessionConfiguration:0x0];
    var_8 = rax;
    var_20 = [rax retain];
    objc_storeStrong(0x0, 0x0);
    objc_storeStrong(var_8, 0x0);
    rax = var_20;
    return rax;
}
```

```
// -[AFURLSessionManagerTaskDelegate init]
void * -[AFURLSessionManagerTaskDelegate init](void * self, void * _cmd) {
    rax = var_28;
    rax = [[rax super] init];
    var_10 = rax;
    objc_storeStrong(0x0, rax);
    if (var_10 == 0x0) {
            var_8 = 0x0;
    }
    else {
            rax = [NSMutableData data];
            rax = [rax retain];
            var_40 = rax;
            [var_10 setMutableData:rax];
            [var_40 release];
            rax = [NSProgress alloc];
            rax = [rax initWithParent:0x0 userInfo:rcx];
            var_50 = rax;
            [var_10 setUploadProgress:rax];
            [var_50 release];
            rax = [var_10 uploadProgress];
            rax = [rax retain];
            var_58 = rax;
            [rax setTotalUnitCount:*_NSURLSessionTransferSizeUnknown];
            [var_58 release];
            rax = [NSProgress alloc];
            rax = [rax initWithParent:0x0 userInfo:rcx];
            var_68 = rax;
            [var_10 setDownloadProgress:rax];
            [var_68 release];
            rax = [var_10 downloadProgress];
```

```
            rax = [rax retain];
            var_70 = rax;
            [rax setTotalUnitCount:*_NSURLSessionTransferSizeUnknown];
            [var_70 release];
            var_8 = [var_10 retain];
    }
    objc_storeStrong(var_10, 0x0);
    rax = var_8;
    return rax;
}
```

```
// -[AFURLSessionManager invalidateSessionCancelingTasks:]
void -[AFURLSessionManager invalidateSessionCancelingTasks:](void * self, void
     * _cmd, char arg2) {
    var_50 = [objc_retainAutoreleaseReturnValue(__dispatch_main_q) retain];
    [self retain];
    dispatch_async(var_50, __NSConcreteStackBlock);
    [var_50 release];
    objc_storeStrong(var_40 + 0x20, 0x0);
    return;
}
```

```
// -[AFURLSessionManager respondsToSelector:]
char -[AFURLSessionManager respondsToSelector:](void * self, void * _cmd, void
     * arg2) {
    var_10 = self;
    var_20 = arg2;
    if (var_20 == @selector(URLSession:task:willPerformHTTPRedirection:
    newRequest:completionHandler:)) {
            rax = [var_10 taskWillPerformHTTPRedirection];
            rax = [rax retain];
            var_1 = (rax != 0x0 ? 0x1 : 0x0) & 0x1 & 0xff;
            [rax release];
    }
    else {
            if (var_20 == @selector(URLSession:dataTask:didReceiveResponse:
    completionHandler:)) {
                    rax = [var_10 dataTaskDidReceiveResponse];
                    rax = [rax retain];
                    var_1 = (rax != 0x0 ? 0x1 : 0x0) & 0x1 & 0xff;
                    [rax release];
            }
            else {
                    if (var_20 == @selector(URLSession:dataTask:
    willCacheResponse:completionHandler:)) {
                            rax = [var_10 dataTaskWillCacheResponse];
                            rax = [rax retain];
                            var_1 = (rax != 0x0 ? 0x1 : 0x0) & 0x1 & 0xff;
                            [rax release];
                    }
                    else {
                            if (var_20 == @selector(
    URLSessionDidFinishEventsForBackgroundURLSession:)) {
                                    rax = [var_10
    didFinishEventsForBackgroundURLSession];
```

```
28                                      rax = [rax retain];
29                                      var_1 = (rax != 0x0 ? 0x1 : 0x0) & 0x1 & 0
      xff;
30                                      [rax release];
31                              }
32                              else {
33                                      var_1 = [[var_10 class]
      instancesRespondToSelector:var_20];
34                              }
35                      }
36              }
37      }
38      rax = sign_extend_64(var_1);
39      return rax;
40  }
```

```
1   // +[AFSecurityPolicy certificatesInBundle:]
2   void * +[AFSecurityPolicy certificatesInBundle:](void * self, void * _cmd,
        void * arg2) {
3       var_8 = *___stack_chk_guard;
4       objc_storeStrong(0x0, arg2);
5       var_A8 = [[0x0 pathsForResourcesOfType:@"cer" inDirectory:@"."] retain];
6       var_B0 = [[NSMutableSet setWithCapacity:[var_A8 count]] retain];
7       memset(var_F8, 0x0, 0x40);
8       rax = [var_A8 retain];
9       var_110 = rax;
10      rax = [rax countByEnumeratingWithState:var_F8 objects:var_88 count:0x10];
11      var_118 = rax;
12      if (rax != 0x0) {
13              var_120 = *var_E8;
14              var_128 = var_F8 + 0x10;
15              var_130 = 0x0;
16              var_138 = var_118;
17              do {
18                      do {
19                              var_140 = var_138;
20                              var_148 = var_130;
21                              if (**var_128 != var_120) {
22                                      objc_enumerationMutation(var_110);
23                              }
24                              var_100 = [[NSData dataWithContentsOfFile:*(var_F0
       + var_148 * 0x8)] retain];
25                              [var_B0 addObject:var_100];
26                              objc_storeStrong(var_100, 0x0);
27                              var_138 = var_140;
28                              var_130 = var_148 + 0x1;
29                      } while (var_148 + 0x1 < var_140);
30                      rax = [var_110 countByEnumeratingWithState:var_F8 objects:
        var_88 count:0x10];
31                      var_130 = 0x0;
32                      var_138 = rax;
33              } while (rax != 0x0);
34      }
35      [var_110 release];
36      var_150 = [[NSSet setWithSet:var_B0] retain];
37      objc_storeStrong(var_B0, 0x0);
38      objc_storeStrong(var_A8, 0x0);
```

```
39      objc_storeStrong(0x0, 0x0);
40      var_158 = [var_150 autorelease];
41      if (*___stack_chk_guard == var_8) {
42              rax = var_158;
43      }
44      else {
45              rax = __stack_chk_fail();
46      }
47      return rax;
48  }
```

Hopper decompilation results for release build

```
1   // +[AFNetworkReachabilityManager sharedManager]
2   void * +[AFNetworkReachabilityManager sharedManager](void * self, void * _cmd)
        {
3       if (*_sharedManager.onceToken != 0xffffffffffffffff) {
4               dispatch_once(_sharedManager.onceToken, __NSConcreteStackBlock);
5       }
6       rax = objc_retainAutoreleaseReturnValue(*_sharedManager._sharedManager);
7       return rax;
8   }
```

```
1   // +[AFNetworkReachabilityManager managerForDomain:]
2   void * +[AFNetworkReachabilityManager managerForDomain:](void * self, void *
        _cmd, void * arg2) {
3       r15 = *_kCFAllocatorDefault;
4       r12 = [arg2 retain];
5       rbx = [objc_retainAutorelease(arg2) UTF8String];
6       [r12 release];
7       rbx = SCNetworkReachabilityCreateWithName(r15, rbx);
8       r14 = [[self alloc] initWithReachability:rbx];
9       CFRelease(rbx);
10      rdi = r14;
11      rax = [rdi autorelease];
12      return rax;
13  }
```

```
1   // +[AFNetworkReachabilityManager managerForAddress:]
2   int +[AFNetworkReachabilityManager managerForAddress:](int arg0) {
3       rbx = SCNetworkReachabilityCreateWithAddress(*_kCFAllocatorDefault, rdx);
4       r14 = [[arg0 alloc] initWithReachability:rbx];
5       CFRelease(rbx);
6       rdi = r14;
7       rax = [rdi autorelease];
8       return rax;
9   }
```

```
1   // -[AFNetworkReachabilityManager isReachable]
2   char -[AFNetworkReachabilityManager isReachable](void * self, void * _cmd) {
```

```
3       rbx = self;
4       rcx = 0x1;
5       if ([self isReachableViaWWAN] == 0x0) {
6               rcx = [rbx isReachableViaWiFi] != 0x0 ? 0x1 : 0x0;
7       }
8       rax = rcx & 0xff;
9       return rax;
10  }
```

```
1   // -[AFNetworkReachabilityManager stopMonitoring]
2   void -[AFNetworkReachabilityManager stopMonitoring](void * self, void * _cmd)
        {
3       rbx = self;
4       r14 = @selector(networkReachability);
5       if (_objc_msgSend(self, r14) != 0x0) {
6               rbx = _objc_msgSend(rbx, r14);
7               rax = CFRunLoopGetMain();
8               rdi = rbx;
9               SCNetworkReachabilityUnscheduleFromRunLoop(rdi, rax, *
        _kCFRunLoopCommonModes, _kCFRunLoopCommonModes);
10      }
11      return;
12  }
```

```
1   // -[AFSecurityPolicy pinnedCertificates]
2   void * -[AFSecurityPolicy pinnedCertificates](void * self, void * _cmd) {
3       rax = self->_pinnedCertificates;
4       return rax;
5   }
```

```
1   // -[AFHTTPSessionManager GET:parameters:success:failure:]
2   void * -[AFHTTPSessionManager GET:parameters:success:failure:](void * self,
        void * _cmd, void * arg2, void * arg3, void * arg4, void * arg5) {
3       r12 = [arg2 retain];
4       r13 = [arg3 retain];
5       rbx = [arg4 retain];
6       r14 = [self GET:r12 parameters:r13 progress:0x0 success:rbx failure:arg5];
7       [rbx release];
8       [r13 release];
9       [r12 release];
10      rax = [r14 retain];
11      rax = [rax autorelease];
12      return rax;
13  }
```

```
1   // -[AFSecurityPolicy validatesDomainName]
2   char -[AFSecurityPolicy validatesDomainName](void * self, void * _cmd) {
3       rax = sign_extend_64(self->_validatesDomainName);
4       return rax;
5   }
```

```
// -[AFSecurityPolicy setValidatesDomainName:]
void -[AFSecurityPolicy setValidatesDomainName:](void * self, void * _cmd,
    char arg2) {
    self->_validatesDomainName = arg2;
    return;
}
```

```
// -[AFHTTPSessionManager initWithBaseURL:]
void * -[AFHTTPSessionManager initWithBaseURL:](void * self, void * _cmd, void
     * arg2) {
    rax = [self initWithBaseURL:arg2 sessionConfiguration:0x0];
    return rax;
}
```

```
// -[AFURLSessionManagerTaskDelegate init]
void * -[AFURLSessionManagerTaskDelegate init](void * self, void * _cmd) {
    r14 = [[self super] init];
    rbx = 0x0;
    if (r14 != 0x0) {
            rbx = [[NSMutableData data] retain];
            [r14 setMutableData:rbx];
            [rbx release];
            rbx = [[NSProgress alloc] initWithParent:0x0 userInfo:0x0];
            [r14 setUploadProgress:rbx];
            [rbx release];
            rbx = [[r14 uploadProgress] retain];
            r12 = *_NSURLSessionTransferSizeUnknown;
            [rbx setTotalUnitCount:r12];
            [rbx release];
            rbx = [[NSProgress alloc] initWithParent:0x0 userInfo:0x0];
            [r14 setDownloadProgress:rbx];
            [rbx release];
            rbx = [[r14 downloadProgress] retain];
            [rbx setTotalUnitCount:r12];
            [rbx release];
            rbx = [r14 retain];
    }
    [r14 release];
    rax = rbx;
    return rax;
}
```

```
// -[AFURLSessionManager invalidateSessionCancelingTasks:]
void -[AFURLSessionManager invalidateSessionCancelingTasks:](void * self, void
     * _cmd, char arg2) {
    var_10 = [self retain];
    dispatch_async(__dispatch_main_q, __NSConcreteStackBlock);
    [var_10 release];
    return;
}
```

```
1  // -[AFURLSessionManager respondsToSelector:]
2  char -[AFURLSessionManager respondsToSelector:](void * self, void * _cmd, void
       * arg2) {
3      rdi = self;
4      rbx = arg2;
5      if (@selector(URLSession:task:willPerformHTTPRedirection:newRequest:
       completionHandler:) == rbx) goto loc_b75b;
6
7  loc_b71b:
8      if (@selector(URLSession:dataTask:didReceiveResponse:completionHandler:)
       == rbx) goto loc_b764;
9
10 loc_b724:
11     if (@selector(URLSession:dataTask:willCacheResponse:completionHandler:) ==
        rbx) goto loc_b76d;
12
13 loc_b72d:
14     if (@selector(URLSessionDidFinishEventsForBackgroundURLSession:) == rbx)
       goto loc_b776;
15
16 loc_b736:
17     rbx = [[rdi class] instancesRespondToSelector:rbx];
18     goto loc_b79a;
19
20 loc_b79a:
21     rax = sign_extend_64(rbx);
22     return rax;
23
24 loc_b776:
25     rsi = @selector(didFinishEventsForBackgroundURLSession);
26     goto loc_b77d;
27
28 loc_b77d:
29     rax = _objc_msgSend(rdi, rsi);
30     rax = [rax retain];
31     rbx = rax != 0x0 ? 0x1 : 0x0;
32     [rax release];
33     goto loc_b79a;
34
35 loc_b76d:
36     rsi = @selector(dataTaskWillCacheResponse);
37     goto loc_b77d;
38
39 loc_b764:
40     rsi = @selector(dataTaskDidReceiveResponse);
41     goto loc_b77d;
42
43 loc_b75b:
44     rsi = @selector(taskWillPerformHTTPRedirection);
45     goto loc_b77d;
46 }
```

```
1  // +[AFSecurityPolicy certificatesInBundle:]
2  void * +[AFSecurityPolicy certificatesInBundle:](void * self, void * _cmd,
       void * arg2) {
3      var_30 = *___stack_chk_guard;
4      r14 = [[arg2 pathsForResourcesOfType:@"cer" inDirectory:@"."] retain];
```

```
    var_F8 = [[NSMutableSet setWithCapacity:[r14 count]] retain];
    intrinsic_movaps(var_C0, 0x0);
    intrinsic_movaps(var_D0, 0x0);
    var_E0 = intrinsic_movaps(var_E0, 0x0);
    var_F0 = intrinsic_movaps(var_F0, 0x0);
    rax = [r14 retain];
    var_100 = rax;
    r15 = [rax countByEnumeratingWithState:var_F0 objects:var_B0 count:0x10];
    if (r15 != 0x0) {
            r13 = *var_E0;
            do {
                    r12 = 0x0;
                    do {
                            if (*var_E0 != r13) {
                                    objc_enumerationMutation(var_100);
                            }
                            r14 = [[NSData dataWithContentsOfFile:*(var_E8 +
    r12 * 0x8)] retain];
                            [var_F8 addObject:r14];
                            [r14 release];
                            r12 = r12 + 0x1;
                    } while (r12 < r15);
                    r15 = [var_100 countByEnumeratingWithState:var_F0 objects:
    var_B0 count:0x10];
            } while (r15 != 0x0);
    }
    [var_100 release];
    r15 = [[NSSet setWithSet:var_F8] retain];
    [var_F8 release];
    [var_100 release];
    if (*___stack_chk_guard == var_30) {
            rdi = r15;
            rax = [rdi autorelease];
    }
    else {
            rax = __stack_chk_fail();
    }
    return rax;
}
```

Cricket decompilation results for debug build

```
// +[AFNetworkReachabilityManager sharedManager]
+ (id)sharedManager {
    long temp_2 = ^() {
        long temp_2 = &(self);
        *(257456) = [self manager];
    };
    static dispatch_once_t onceToken;
    dispatch_once(&(onceToken), temp_2);
    return *(257456);
}
```

```
// +[AFNetworkReachabilityManager managerForDomain:]
+ (id)managerForDomain:(id)arg_10 {
    long var_18 = 0;
    long temp_3 = &(var_18);
    *(temp_3) = arg_10;
    id rax = [var_18 UTF8String];
    rax = _SCNetworkReachabilityCreateWithName(*(_kCFAllocatorDefault), rax);
    long var_20 = rax;
    rax = [self alloc];
    [rax initWithReachability:var_20];
    _CFRelease(var_20);
    *(temp_3) = 0;
    return 0;
}
```

```
// +[AFNetworkReachabilityManager managerForAddress:]
+ (id)managerForAddress:(long)arg_10 {
    long rax = _SCNetworkReachabilityCreateWithAddress(*(_kCFAllocatorDefault),
     arg_10);
    long var_20 = rax;
    rax = [self alloc];
    [rax initWithReachability:var_20];
    _CFRelease(var_20);
    return 0;
}
```

```
// -[AFNetworkReachabilityManager isReachable]
- (char)isReachable {
    id rax = [self isReachableViaWWAN];
    long rcx = rcx && 18446744073709551360 || 1;
    if (!(rax == 0)) {
        [self isReachableViaWiFi];
    }
    rcx = rcx && 1;
    rax = rcx;
    return rax;
}
```

```
// -[AFNetworkReachabilityManager stopMonitoring]
- (void)stopMonitoring {
    long var_8 = self;
    id rax = [self networkReachability];
    if (!(rax == 0)) {
    } else {
        [var_8 networkReachability];
        rax = _CFRunLoopGetMain(var_8, @selector(networkReachability));
        _SCNetworkReachabilityUnscheduleFromRunLoop(rax, rax);
    }
}
```

```
// -[AFSecurityPolicy pinnedCertificates]
- (id)pinnedCertificates {
```

```
    long temp_7 = self + _OBJC_IVAR_$_AFSecurityPolicy._pinnedCertificates;
    return self->_pinnedCertificates;
}
```

```
// -[AFHTTPSessionManager GET:parameters:success:failure:]
- (id)GET:(id)arg_10 parameters:(id)arg_18 success:(void *)arg_20 failure:(
    void *)arg_28 {
    long var_18 = 0;
    long temp_3 = &(var_18);
    *(temp_3) = arg_10;
    long var_20 = 0;
    long temp_9 = &(var_20);
    *(temp_9) = arg_18;
    long var_28 = 0;
    long temp_14 = &(var_28);
    *(temp_14) = arg_20;
    long var_30 = 0;
    long temp_19 = &(var_30);
    *(temp_19) = arg_28;
    *(temp_19) = 0;
    *(temp_14) = 0;
    *(temp_9) = 0;
    *(temp_3) = 0;
    return [self GET:var_18 parameters:var_20 progress:0 success:var_28
    failure:var_30];
}
```

```
// -[AFSecurityPolicy validatesDomainName]
- (char)validatesDomainName {
    long temp_7 = self + _OBJC_IVAR_$_AFSecurityPolicy._validatesDomainName;
    return self->_validatesDomainName;
}
```

```
// -[AFSecurityPolicy setValidatesDomainName:]
- (void)setValidatesDomainName:(char)arg_10 {
    long temp_17 = self + _OBJC_IVAR_$_AFSecurityPolicy._validatesDomainName;
    self->_validatesDomainName = arg_10;
}
```

```
// -[AFHTTPSessionManager initWithBaseURL:]
- (id)initWithBaseURL:(id)arg_10 {
    return [0 initWithBaseURL:arg_10 sessionConfiguration:0];
}
```

```
// -[AFURLSessionManagerTaskDelegate init]
- (id)init {
    long temp_28;
    long var_38;
    long var_8;
    long temp_0 = &(0);
```

```
    id rax = [super init];
    long var_10 = rax;
    if (!(rax == 0)) {
        var_8 = 0;
    } else {
        var_38->off_0 = var_10;
        rax = [NSMutableData data];
        <<TODO uGETMEMBER.8 temp_28.8 := var_38.12[0x0]>>;
        [temp_28 setMutableData:rax];
        rax = [[NSProgress alloc] initWithParent:0 userInfo:0];
        [var_10 setUploadProgress:rax];
        rax = [var_10 uploadProgress];
        [rax setTotalUnitCount:*(_NSURLSessionTransferSizeUnknown)];
        rax = [[NSProgress alloc] initWithParent:0 userInfo:0];
        [var_10 setDownloadProgress:rax];
        rax = [var_10 downloadProgress];
        [rax setTotalUnitCount:*(_NSURLSessionTransferSizeUnknown)];
        var_8 = var_10;
    }
    return var_8;
}
```

```
// -[AFURLSessionManager invalidateSessionCancelingTasks:]
- (void)invalidateSessionCancelingTasks:(char)arg_10 {
    long temp_0 = arg_10;
    _dispatch_async(__dispatch_main_q, ^() {
        long rax;
        long temp_2 = &(temp_0);
        long var_18 = block_literal;
        if (temp_0 == 0) {
            rax = [*(var_18 + 32) session];
            [rax invalidateAndCancel];
        } else {
            rax = [*(var_18 + 32) session];
            [rax finishTasksAndInvalidate];
        }
    });
    *(^() {
        long rax;
        long temp_2 = &(temp_0);
        long var_18 = block_literal;
        if (temp_0 == 0) {
            rax = [*(var_18 + 32) session];
            [rax invalidateAndCancel];
        } else {
            rax = [*(var_18 + 32) session];
            [rax finishTasksAndInvalidate];
        }
    }) = 0;
}
```

```
// -[AFURLSessionManager respondsToSelector:]
- (char)respondsToSelector:(char *)arg_10 {
    long temp_50;
    long var_10;
```

```
    long temp_88;
    long temp_71;
    long temp_29;
    long temp_8;
    var_10->off_0 = self;
    long var_20 = arg_10;
    BOOL zf = arg_10 == @selector(URLSession:task:willPerformHTTPRedirection:
    newRequest:completionHandler:);
    BOOL branch_condition = !(zf);
    if (branch_condition) {
        <<TODO uGETMEMBER.8 temp_8.8 := var_10.15[0x0]>>;
        rax = [temp_8 taskWillPerformHTTPRedirection];
        zf = rax == 0;
        branch_condition = !(zf);
        arg_10 = branch_condition && 1;
    } else {
        zf = var_20 == @selector(URLSession:dataTask:didReceiveResponse:
    completionHandler:);
        branch_condition = !(zf);
        if (branch_condition) {
            zf = var_20 == @selector(URLSession:dataTask:willCacheResponse:
    completionHandler:);
            branch_condition = !(zf);
            if (branch_condition) {
                zf = var_20 == @selector(
    URLSessionDidFinishEventsForBackgroundURLSession:);
                branch_condition = !(zf);
                if (branch_condition) {
                    <<TODO uGETMEMBER.8 temp_71.8 := var_10.15[0x0]>>;
                    [temp_71 didFinishEventsForBackgroundURLSession];
                } else {
                    <<TODO uGETMEMBER.8 temp_88.8 := var_10.15[0x0]>>;
                    rax = [temp_88 class];
                    [rax instancesRespondToSelector:var_20];
                }
            } else {
                <<TODO uGETMEMBER.8 temp_50.8 := var_10.15[0x0]>>;
                [temp_50 dataTaskWillCacheResponse];
            }
        } else {
            <<TODO uGETMEMBER.8 temp_29.8 := var_10.15[0x0]>>;
            [temp_29 dataTaskDidReceiveResponse];
        }
    }
    long rax = arg_10;
    return rax;
}
```

```
// +[AFSecurityPolicy certificatesInBundle:]
+ (id)certificatesInBundle:(id)arg_10 {
    long temp_94;
    long var_88;
    long var_148;
    BOOL cf;
    long temp_75;
    long var_130;
    long var_f8;
```

```
    id rax = [arg_10 pathsForResourcesOfType:@"cer" inDirectory:@"."];
    rax = [rax count];
    rax = [NSMutableSet setWithCapacity:rax];
    long var_b0 = rax;
    _memset(&(var_f8), 0, 64);
    long temp_32 = &(var_f8);
    long temp_33 = &(var_88);
    long var_110 = rax;
    BOOL zf = rax == 0;
    for (id temp_75 in rax) {
        rax = [NSData dataWithContentsOfFile:temp_75];
        [var_b0 addObject:rax];
        temp_94 = var_148 + 1;
        cf = temp_94 > rax;
        var_130 = temp_94;
    }
    rax = [NSSet setWithSet:var_b0];
    return rax;
}
```

Cricket decompilation results for release build

```
// +[AFNetworkReachabilityManager sharedManager]
+ (id)sharedManager {
    static dispatch_once_t onceToken;
    dispatch_once(&(onceToken), ^() {
        long temp_0 = &(self);
        *(182960) = [self manager];
    });
    return *(182960);
}
```

```
// +[AFNetworkReachabilityManager managerForDomain:]
+ (id)managerForDomain:(id)arg_10 {
    id rax = [arg_10 UTF8String];
    rax = _SCNetworkReachabilityCreateWithName(*(_kCFAllocatorDefault), rax);
    long rbx = rax;
    rax = [[self alloc] initWithReachability:rbx];
    _CFRelease(rbx);
    return rax;
}
```

```
// +[AFNetworkReachabilityManager managerForAddress:]
+ (id)managerForAddress:(long)arg_10 {
    long rax = _SCNetworkReachabilityCreateWithAddress(*(_kCFAllocatorDefault),
     arg_10);
    long rbx = rax;
    rax = [[self alloc] initWithReachability:rbx];
    _CFRelease(rbx);
    return rax;
}
```

```
// -[AFNetworkReachabilityManager isReachable]
- (char)isReachable {
    id rax = [self isReachableViaWWAN];
    long zf = rax;
    BOOL branch_condition = !(zf);
    if (branch_condition) {
        rax = [self isReachableViaWiFi];
        zf = rax;
        branch_condition = !(zf);
    }
    rax = branch_condition;
    return rax;
}
```

```
// -[AFNetworkReachabilityManager stopMonitoring]
- (void)stopMonitoring {
    long rbx = self;
    id rax = [self networkReachability];
    if (rax) {
        return;
    }
    rax = [rbx networkReachability];
    rbx = rax;
    rax = _CFRunLoopGetMain();
    _SCNetworkReachabilityUnscheduleFromRunLoop(rbx, rax, *(
    _kCFRunLoopCommonModes), _kCFRunLoopCommonModes);
}
```

```
// -[AFSecurityPolicy pinnedCertificates]
- (id)pinnedCertificates {
    long temp_3 = self + _OBJC_IVAR_$_AFSecurityPolicy._pinnedCertificates;
    return self->_pinnedCertificates;
}
```

```
// -[AFHTTPSessionManager GET:parameters:success:failure:]
- (id)GET:(id)arg_10 parameters:(id)arg_18 success:(void *)arg_20 failure:(
    void *)arg_28 {
    long var_30;
    var_30->off_0 = self;
    return [var_30->off_0 GET:arg_10 parameters:arg_18 progress:0 success:
    arg_20 failure:arg_28];
}
```

```
// -[AFSecurityPolicy validatesDomainName]
- (char)validatesDomainName {
    long temp_3 = self + _OBJC_IVAR_$_AFSecurityPolicy._validatesDomainName;
    return self->_validatesDomainName;
}
```

```
// -[AFSecurityPolicy setValidatesDomainName:]
- (void)setValidatesDomainName:(char)arg_10 {
    long temp_4 = self + _OBJC_IVAR_$_AFSecurityPolicy._validatesDomainName;
    self->_validatesDomainName = arg_10;
}
```

```
// -[AFHTTPSessionManager initWithBaseURL:]
- (id)initWithBaseURL:(id)arg_10 {
    return [self initWithBaseURL:arg_10 sessionConfiguration:0];
}
```

```
// -[AFURLSessionManagerTaskDelegate init]
- (id)init {
    long temp_30;
    long temp_6 = &(self);
    id rax = [super init];
    long r14 = rax;
    if (rax) {
        rax = [NSMutableData data];
        [r14 setMutableData:rax];
        rax = [[NSProgress alloc] initWithParent:0 userInfo:0];
        [r14 setUploadProgress:rax];
        rax = [r14 uploadProgress];
        temp_30 = *(_NSURLSessionTransferSizeUnknown);
        [rax setTotalUnitCount:temp_30];
        rax = [[NSProgress alloc] initWithParent:0 userInfo:0];
        [r14 setDownloadProgress:rax];
        rax = [r14 downloadProgress];
        [rax setTotalUnitCount:temp_30];
        rax = r14;
    }
    return rax;
}
```

```
// -[AFURLSessionManager invalidateSessionCancelingTasks:]
- (void)invalidateSessionCancelingTasks:(char)arg_10 {
    long temp_9 = arg_10;
    _dispatch_async(__dispatch_main_q, ^() {
        long rsi;
        long temp_0 = &(temp_9);
        id rax = [*(&(self)) session];
        id rbx = rax;
        if (temp_9 == 0) {
            rsi = @selector(finishTasksAndInvalidate);
        } else {
            rsi = @selector(invalidateAndCancel);
        }
        block_literal = rbx;
        rax = _objc_msgSend(rbx, rsi);
    });
}
```

```
// -[AFURLSessionManager respondsToSelector:]
- (char)respondsToSelector:(char *)arg_10 {
    long temp_14;
    long rbx = arg_10;
    BOOL zf = @selector(URLSession:task:willPerformHTTPRedirection:newRequest:
    completionHandler:) == arg_10;
    if (zf) {
        _cmd = @selector(taskWillPerformHTTPRedirection);
    } else {
        zf = @selector(URLSession:dataTask:didReceiveResponse:
    completionHandler:) == rbx;
        if (zf) {
            _cmd = @selector(dataTaskDidReceiveResponse);
        } else {
            zf = @selector(URLSession:dataTask:willCacheResponse:
    completionHandler:) == rbx;
            if (zf) {
                zf = @selector(
    URLSessionDidFinishEventsForBackgroundURLSession:) == rbx;
                if (zf) {
                    rax = [self class];
                    rax = [rax instancesRespondToSelector:rbx];
                    temp_14 = rax;
                    temp_32 = temp_14;
                    rax = temp_32;
                    return rax;
                }
                _cmd = @selector(didFinishEventsForBackgroundURLSession);
            } else {
                _cmd = @selector(dataTaskWillCacheResponse);
            }
        }
    }
    long rax = _objc_msgSend(self, _cmd);
    long temp_32 = temp_14;
    rax = temp_32;
    return rax;
}
```

```
// +[AFSecurityPolicy certificatesInBundle:]
+ (id)certificatesInBundle:(id)arg_10 {
    long var_b0;
    long temp_44;
    BOOL cf;
    long temp_55;
    long var_f0;
    id rax = [arg_10 pathsForResourcesOfType:@"cer" inDirectory:@"."];
    rax = [rax count];
    rax = [NSMutableSet setWithCapacity:rax];
    long var_f8 = rax;
    var_f0->off_0 = 0;
    long var_100 = rax;
    long temp_27 = &(var_f0);
    long temp_28 = &(var_b0);
    long zf = rax;
    for (id temp_44 in rax) {
        rax = [NSData dataWithContentsOfFile:temp_44];
```

```
19          [var_f8 addObject:rax];
20          temp_55 = temp_55 + 1;
21          cf = temp_55 > rax;
22      }
23      rax = [NSSet setWithSet:var_f8];
24      return rax;
25  }
```