A few months ago Apple announced a 'new feature,' called **Bitcode**. In this article, I will try to answer the questions like what is Bitcode, what problems it aims to solve, what issues it introduces and so on.

What is Bitcode?

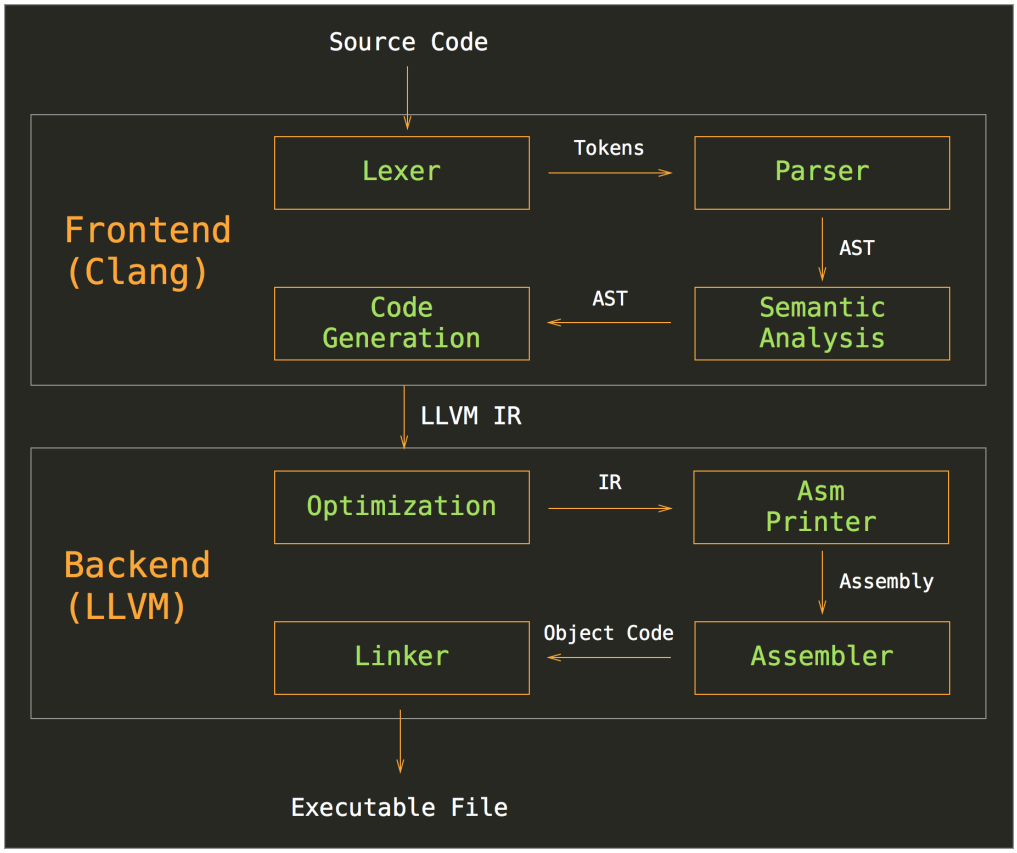
To answer this question let's look at what compilers do for us. Here is a brief overview of compilation process:

- **Lexer**: takes source code as an input and translates it into a stream of tokens;
- **Parser**: takes stream of tokens as an input and translates it into an AST;
- **Semantic Analysis**: takes an AST as an input, checks if a program is correct (method called with correct amount of parameters, method called on object actually exists and non-private, etc.), fills in 'missing types' (e.g.: `let x = y`, x has type of y) and passes AST to the next phase;
- **Code Generation**: takes an AST as an input and emits some high-level IR (intermediate representation);
- **Optimization**: takes IR, makes optimizations and emits IR which is potentially faster and/or smaller;
- **AsmPrinter**: another code generation phase, it takes IR and emits assembly for particular CPU;
- **Assembler**: takes assembly and converts it into an object code (stream of `0`s and `1`s);
- **Linker**: usually programs refer to already compiled routines from other programs (e.g.: `printf`) to avoid recompilation of the same code over and over. Until this phase these links do not have correct addresses, they are just placeholders. Linker's job is to resolve those placeholders so that they point to the correct addresses of their corresponding routines.

You can find more details here: The Compiler.

In the modern world these phases are split into two parts: **compiler frontend** (*lexer*, *parser*, *semantic analysis*, *code generation*) and **compiler backend** (*optimization*, *asm printer*, *assembler*, *linker*). This separation makes much sense for both language designers and hardware manufacturers. If you want to create a new programming language you 'just' need to implement a frontend, and you get all available optimizations and support of different CPUs for free. On the other hand, if you created a new chip, you 'just' need to extend the backend and you get all the available languages (frontends) support for your CPU.

Below you can see a picture that illustrates compilation process using Clang and LLVM:



This picture clearly demonstrates how communication between frontend and backend is done using IR, LLVM has it is own format, that can be encoded using LLVM bitstream file format - Bitcode.

Just to recall it explicitly - **Bitcode is a bitstream representation of LLVM IR**.

What problems Apple's Bitcode aims to solve?

Again, we need to dive a bit deeper and look at how an OS runs programs. This description is not precise and is given just to illustrate the process. For more details I can recommend reading this article: How OS X Executes Applications.

OS X and iOS can run on different CPUs (*i386*, *x86_64*, *arm*, *arm64*, etc.), if you want to run a program on any OS X/iOS setup, then the program should contain object code for each platform. Here is how a binary might look like:

When you run a program, OS reads the 'Table Of Contents' and looks for a slice corresponding to the OS CPU.

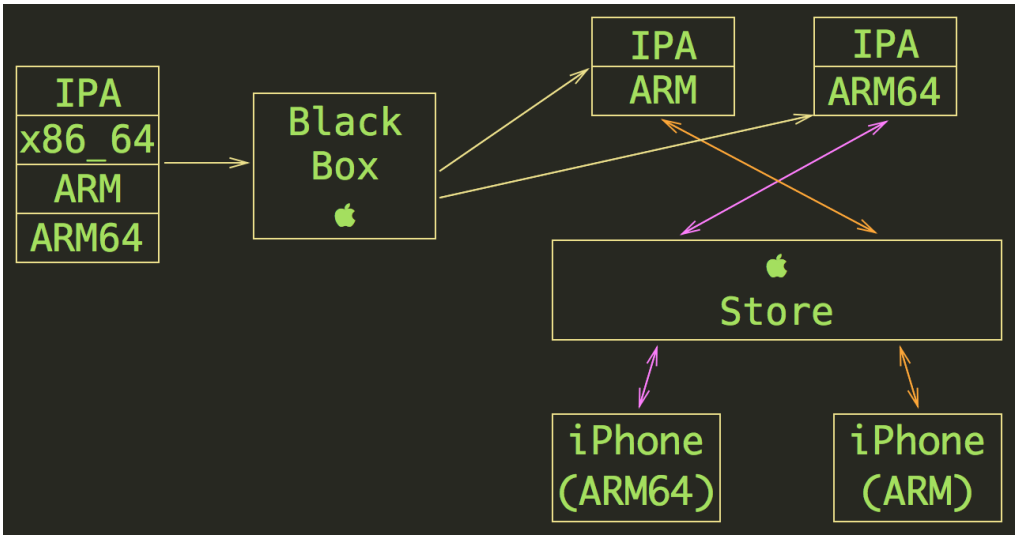
For instance, if you run operating system on *x86_64*, then OS will load object code for *x86_64* into a memory and run the program.

What's happening with other slices? Nothing, they just waste your disk space.

This is the problem Apple wants to solve: currently, all the apps on the AppStore contain object code for *arm* and *arm64* CPUs. Moreover, third-party proprietary libraries or frameworks contain object code for *i386*, *x86_64*, *arm* and *arm64*, so you can use them to test the app on a device or simulator. (Can you imagine how many copies of Google Analytics for *i386* you have in your pocket?)

UPD: I do not know why, but I was sure that final executable contains these slices as well (*i386*, *x86_64*, etc.), but it seems they are stripped during the build phase.

Apple did not give us that many details about how the Bitcode and App Thinning works, so let me assume how it may look:



When you submit an app (including Bitcode) Apple's 'BlackBox' recompiles it for each supported platform and drops any 'useless' object code, so AppStore has a copy of the app for each CPU. When an end user wants to install the app - she installs the only version for the particular processor, without any unused stuff.

Bitcode might save up to 50% of disk space per program.

UPD: Of course, I do not take in count resources, it is just about binary itself. For instance, an app I am working on currently has size ~40 megabytes (including assets, xibs. fonts), a size of a binary itself is ~16 megabytes. I checked sizes of each slice: ~7MB for armv7 and 9MB for arm64, if we crop just one of them, it will decrease the size of the app by ~20%.

What problems does Bitcode introduce?

The idea of Bitcode and recompiling for each platform looks really great, and it is a huge improvement, though it has downsides as well: the biggest one is security.

To get the benefits of Bitcode, you should submit your app including Bitcode (surprisingly). If you use some proprietary third-party library, then it also should contain Bitcode, hence as a maintainer of a proprietary library, you should distribute the library with Bitcode.

To recall: **Bitcode is just another form of LLVM IR.**

LLVM IR

Let's write some code to see LLVM IR in action.

```
// main.c
extern int printf(const char *fmt, ...);

int main() {
  printf("Hello World\n");
  return 0;
}
```

Run the following:

```
clang -S -emit-llvm main.c
```

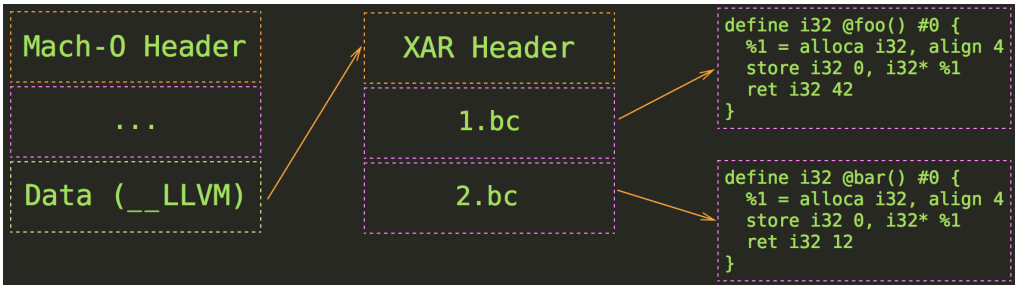
And you'll have `main.ll` containing IR:

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

; Function Attrs: nounwind ssp uwtable
define i32 @main() {
  %1 = alloca i32, align 4
  store i32 0, i32* %1
  %2 = call i32 (i8*, ...)* \
      @printf(i8* getelementptr inbounds \
            ([13 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}

declare i32 @printf(i8*, ...)
```

What can we see here? It is a bit more verbose than original C code, but it is still much more readable than assembler. Malefactors will be much happier to work with this representation, than with disassembled version of a binary (and they do not even have to pay for tools such Hopper or IDA).

How could malefactor get the IR?

iOS and OS X executables have their own format - Mach-O (read Parsing Mach-O files for more details). Mach-O file contains several segments such as Read-Only Data, Code, Symbol Table, etc. One of those sections contain xar archive with Bitcode:



It is really easy to retrieve it automatically, here I wrote a simple C program that does just that: bitcode_retriever. The workflow is pretty straightforward. Let's assume that *some_binary* is a Mach-O file that contains object code for two CPUs (*arm* and *x86_64*), and each object code is built using two source files:

```
$ bitcode_retriever some_binary
arm.xar
x86_64.xar
$ xar -xvf arm.xar
1
2
$ llvm-dis 1 # outputs 1.ll
$ llvm-dis 2 # outputs 2.ll
```

Bitcode does not store any information about original filenames but uses numbers instead (1, 2, 3, etc.).

Also, probably you do not have `llvm-dis` installed/built on your machine, but you can easily obtain it, see this article for more details: Getting Started with Clang/LLVM on OS X.

Another potential issue (can't confirm it) - Bitcode thingie works only for iOS 9, so if you submit your app to the AppStore and it includes Bitcode, then malefactor can get the whole IR from your app using iOS 7/8 and jailbroken device.

I know only one way to secure the IR - obfuscation. This task is not trivial itself, and it requires even much more efforts if you want to introduce this phase into your Xcode-Driven development flow.

Summary

- Bitcode is a bitstream file format for LLVM IR
- one of its goals is to decrease a size of an app by eliminating unused object code
- malefactor can obtain your app or library, retrieve the IR from it and steal your 'secret algorithm.'

Useful links

- LLVM IR- language reference manual
- LLVM Bitcode - Bitcode file format
- The Compiler - Clang/LLVM compilation phases
- How OS X Executes Applications
- Parsing Mach-O files
- bitcode_retriever - tool that retrieves xar-archives with bitcode from mach-o binary
- o-llvm - obfuscator based on LLVM