Real time symbolication

By **Mark Sandstrom**

TestFlight Blog

The SDK

If you've integrated the TestFlight SDK into any of your applications you may have noticed that we recently released symbolication for crash reports. You can now upload a .dSYM via the crashes page (or the upload API) and crashes you receive will be symbolicated on the fly.

Our mission at TestFlight is to help developers create the highest quality apps possible. We believe that real time tools improve the feedback loop by shortening the testing cycle and enable developers to spend more time building higher quality apps.

We've been hard at work on all of our SDK features, and we thought we'd share our experience regarding how we tackled symbolication. For information about the SDK, visit the SDK page at https://testflightapp.com/sdk/.

What is symbolication, and why should I care?

Symbolication is the process of translating addresses in crash reports to function names, method names, file names and line numbers. Raw crash reports received from users via email or downloaded from iTunes Connect look somethings like this:

```
8   libobjc.A.dylib                     0x33d6cc8b 0x33d68000 + 19595
9   CoreFoundation                      0x33893465 0x3388b000 + 33893
10  HelloTestFlight                     0x0002d109 0x2b000 + 8457
11  HelloTestFlight                     0x0002d0df 0x2b000 + 8415
12  CoreFoundation                      0x33899571 0x3388b000 + 58737
13  UIKit                               0x333ceec9 0x333b2000 + 118473

0x2b000 - 0x3d000  HelloTestFlight armv7 <a9603692b66f3a72847c380a50ce2347>
```

After symbolication this same crash report looks something like this:

```
8   libobjc.A.dylib                     0x33d6cc8b objc_exception_throw + 71
9   CoreFoundation                      0x33893465 -[__NSArrayI objectAtIndex:] + 161
10  HelloTestFlight                     0x0002d109 -[HTFViewController indexOutOfBounds] (HTFViewCo
11  HelloTestFlight                     0x0002d0df -[HTFViewController doIndexOutOfBounds] (HTFView
12  CoreFoundation                      0x33899571 -[NSObject(NSObject) performSelector:withObject:
13  UIKit                               0x333ceec9 -[UIApplication sendAction:to:from:forEvent:] +

0x2b000 - 0x3d000  HelloTestFlight armv7 <a9603692b66f3a72847c380a50ce2347>
```

Specifically, symbolication has converted addresses and offsets like 0x2b000 + 8457 into symbols, file names and line numbers like -[HTFViewController indexOutOfBounds] (HTFViewController.m:52). In addition to the method, the file name and line number where the crash occurred are pinpointed:

```
    HTFViewController.m:

    50  - (void)indexOutOfBounds {
    51      NSArray *array = [NSArray arrayWithObject:@"HelloTestFlight"];
->  52      [array objectAtIndex:2];
    53  }
```

This information is similar to the feedback Xcode provides when a crash occurs in development, and it is invaluable for tracking down crashes in the field.

Symbolicating a raw crash report

A hard requirement for symbolication is that a "DWARF with dSYM"; was generated when your app was built. If you use the "Archive"; option in Xcode 4, a .dSYM containing a DWARF (Debugging with Attributed Record Formats) file is automatically generated and saved in the archive for you.

Given a raw crash report it's actually fairly simple to symbolicate it:

1. Launch Xcode
2. Open the Organizer (⌘⇧2)
3. Select the devices tab and drop the crash report on "Device Logs";

After a few moments a fully symbolicated version of the crash will appear. The tool that Apple uses to symbolicate behind the scenes is a perl script named symbolicatecrash. Given that Xcode is installed, you can find symbolicatecrash on your machine as follows:

```
$ find /Developer -name symbolicatecrash
```

With a crash report named testflight.crash in hand, instead of using Xcode you can symbolicate the crash via the command line:

```
$ $(find /Developer -name symbolicatecrash) testflight.crash
```

symbolicatecrash does quite a few things to make the process of symbolication simple. Notably, the tool figures out where to look for debugging information. You may have noticed the following line in the crash reports above:

```
0x2b000 - 0x3d000  HelloTestFlight armv7 <a9603692b66f3a72847c380a50ce2347>
```

This line tells us that the HelloTestFlight binary was loaded in memory at address 0x2b000 on an armv7 device when the crash occured. Additionally the UUID (universally unique identifier) of the specific binary that was loaded is a9603692b66f3a72847c380a50ce2347. The format for binaries on Darwin (the open source core of Mac OS X and iOS) is called Mach-O, and Apple engineers made the fabulous decision to tag every Mach-O binary with a UUID for easy identification.

symbolicatecrash uses the binary UUID to find the corresponding .dSYM and .app bundles via spotlight (i.e. the mdfind command line utility). The DWARF within the .dSYM is tagged with the same UUID as the binary, which is what makes this search possible. In addition to the DWARF, symbolicatecrash locates debugging symbols for libraries in the crash report by scanning various standard locations on the file system and inspecting candidate binaries. These tasks are accomplished by involking several command line utilities:

- mdfind and mdls, a.k.a Spotlight
- otool - object file displaying tool
- size - print the size of the sections in an object file
- lipo - create or operate on universal files

Finally, symbolication of addresses is performed by:

- atos - convert numeric addresses to symbols of binary images or processes

The point is that symbolicatecrash spawns numerous subprocesses to symbolicate a crash report, the primary being atos. This method works well for symbolicating crash reports one at a time, but symbolicatecrash is not suited for batch symbolication. In fact, symbolicatecrash takes about 2 seconds per crash report:

```
$ time for i in {1..10}; symbolicatecrash testflight.crash > /dev/null; done

real    0m19.394s
user    0m7.553s
sys     0m9.175s
```

The challenges of batch symbolication

There are a couple challenges that make real time symbolication as a service difficult. Although 2 seconds per crash report may not seem like a long time, when there are thousands of crash reports pouring in every fraction of a second matters.

The first challenge is speeding up symbolicatecrash, and we were actually able to offload most of this work altogether.

The second challenge is based in the fact that symbolicatecrash relies on Mac OS X/Darwin specific tools. While some of the tools are open source (http://opensource.apple.com/source/cctools/cctools-806/) and portable versions do exist (http://code.google.com/p/iphone-dev/), the key tool, atos, is closed source. We did some research and it turns out that with a little work it's possible to use gdb to convert addresses to symbols, but alas, Apple's version of gdb isn't portable.

Here's what we did to overcome these obstacles.

Speeding up symbolicatecrash

Speeding up symbolicatecrash is really a matter of being smarter about symbolicating library symbols. Scanning standard locations and spawning subprocesses to inspect libraries is very slow. Since developers associate .dSYMs with builds, there is no need to use Spotlight or otherwise search for debugging symbols. To speed things up we first created an index of library binaries based on UUID (and rewrote symbolicatecrash accordingly). That sped up symbolication about 8x.

This isn't the approach we deployed, however. It turns out that libraries on iOS include debug symbols on each device, so what we actually do is symbolicate libraries client-side before sending crash reports to the TestFlight servers. This process of symbolication is different from the one described above since we aren't dealing with DWARFs and there's no file name and line number information to surface.

Writing a portable atos

Running a cluster of Mac OS X machines seems like the obvious, quick solution for providing symbolication as a service. There a few drawbacks to this approach, however:

- We would have to ship data back a forth to a remote cluster
- Maintaining a remote cluster is somewhat complex and operationally time consuming
- A Mac OS X cluster can be expensive to operate

After some research we determined that re-writing atos would involve:

1. Parsing universal files
2. Parsing Mach-O binaries
3. Parsing DWARF
4. Deriving line and file information from DWARF "statement programs";

The file formats and derivation process are well documented, so we were convinced that we could write the tool. This is a better investment than maintaining a cluster, in our opinion. That, and we figured it would be a fun challenge.

I won't go into too many details, but here's what atos basically does:

Given the stack frame:

```
10  HelloTestFlight                     0x0002d109 0x2b000 + 8457
    ^                                   ^          ^
    Binary image name                   runtime    load
                                        address    address
```

1. Determine the relative address of the symbol:

```
runtime address     = 0x0002d109
load address        = 0x2b000
relative address    = runtime address - load address    = 0x2109
```

2. Determine the vm address the binary was built at by looking at the __TEXT Mach-O load command:

```
vmaddr              = 0x1000
address             = vmaddr + relative address         = 0x3109
```

3. Load the DWARF data from the DWARF-related Mach-O sections
4. Find a DWARF compilation unit containing a subprogram with the given address; the subprogram name is the symbol
5. Derive the line information for the address using the compilation unit's statement program

If you're interesting in learning more about Mach-O and DWARF, here are the definitive resources:

http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html
http://dwarfstd.org/doc/dwarf-2.0.0.pdf

The Result

The resulting version of symbolicatecrash is a simple wrapper around atos that parses crash reports and offloads the rest to atos. Here is how it performs:

```
$ time for i in {1..10}; symbolicatecrash testflight.crash > /dev/null; done

real    0m0.997s
user    0m0.685s
sys     0m0.267s
```

...on linux, which means we can symbolicate crashes locally.

We're pretty excited about these results. This is almost a 20x performance increase over the version of symbolicatecrash provided with Xcode. Not only can we now provide real time symbolication for developers, but we succeeded in keeping our infrastructure simple which allows us to focus on creating even more tools.