PBKDF2 Memo (study of code)

Dreycey Albin

Call stack/tree for code base and Crypto Hashing

This Memo works as a written analysis and study of the PBKDF2 code base as it has been initially delivered. A primary goal of current project is to understand the code base, since there is a need to optimize the currently working code base and modify what is currently written. This memo covers the initial code size (i.e. '.text' size), the initial run times per optimization level, and an over view of bottle-neck functions.

Initially passing test cases

The code base has both unit tests and timing tests initially built-in. As a first look into the functionality, here we detail the feedback from the tests in the current state. ***Figure 1*** shows the initial feedback for the test cases before optimization.

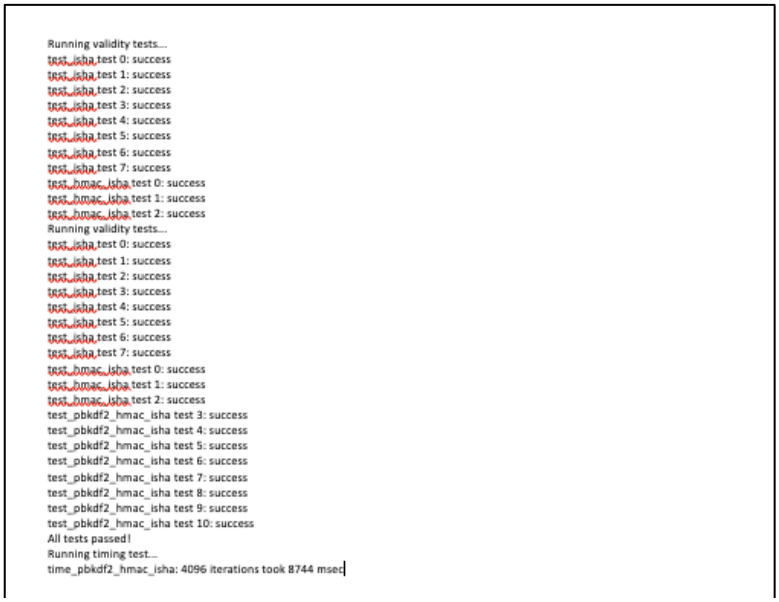

Figure 1 - UART feedback from the unit tests for the initial code base before optimization.

Call stack/tree for code base and Crypto Hashing

The call tree for the Cryptographic hashing is nested several functions deep (***Figure 2***). The top-most function is the PBKDF2 function which calls upon an F function twice. The F function then calls upon HMAC to perform 4097 (iterations) of an XOR operation to generate the key. For each call to HMAC, HMAC subsequently calls a set of ISHA commands twice (an 'outer' and 'inner' hashing). This set of ISHA commands is as follows: (1) Reset ISHA, (2) ISHA input, (3) ISHA input, (4) ISHA Result.

This set of commands defines the basic operations of the PBKDF2 cryptographic hashing, as well illustrate how the most-nested functions may act as timing bottlenecks. The cryptographic hashing could be done using ISHA alone, but the added layer of the outer and inner hashing (HMAC) with and XOR operation (PBKDF2) offers a more secure algorithm.

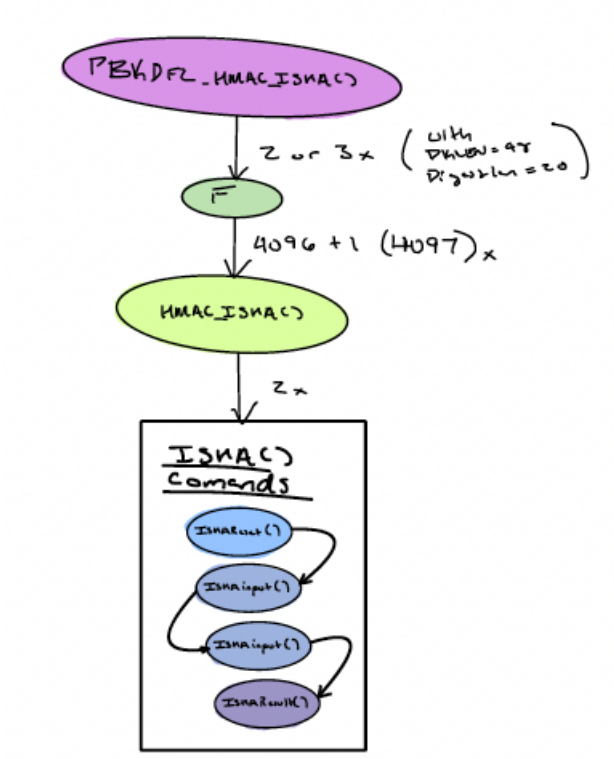

Figure 2 - Illustrations showing the call tree for all of the major functions within the code base pre-optimization.

Initial size of code base

In addition to timing, the initial size of the code base in machine language is also of importance. The goal would be to add optimizations without dramatically adding source code. To ensure we can measure the size of the code base as additions are made, the following command is used:

arm-none-eabi-objdump -x Debug/PBKDF2.axf | grep .text | head -10

Using the 'objdump' command (above) for the compiled binary of the project at different optimization levels resulted in the following table (**Table 1**). It is clear the largest amount of code is in the lowest optimization level, and that the size-optimized code base significantly reduced the size of the code base.

Table 1 - Size of '.text' at different optimization levels in the code base pre-optimization

Optimization level	Hex Code	Bytes
-O0	00005240	21056
-O3	00004594	17812
-Os	0x00003af8	15096

Initial running time of code base

The primary goals for optimizing the code base as it currently stands is to increase the speed for which the code base computes the PBKDf2 cryptographic hashing. As such, initial efforts have been directed to finding 'bottlenecks' in the code base that could be causing delays in runtime execution speeds.

First step was focused on finding which function call causing the time-performance. After searching top-down, it was found that the F functions call to HMAC-ISHA during the XOR operations is where the biggest drop in performance is occurring (8 seconds to 0.270 seconds when blocked).

```
102
103     for (int j=1; j<iter; j++) {
104       //hmac_isha(pass, pass_len, temp, ISHA_DIGESTLEN, temp);
105       for (int i=0; i<ISHA_DIGESTLEN; i++)
106         result[i] ^= temp[i];
107     }
```

Figure 3 - Preventing HMAC-ISHA from being called within the F function XOR iterations scaled the time down dramatically. (8 seconds to 270 mSec)

Digging further into the HMAC-ISHA function, it was quickly found that the calls to the underlying ISHA methods are taking up majority of the time. Without the ISHA calls, the 4096 iterations takes 1.7 seconds (from 8 seconds). Next, looking at the ISHA functions separately, it was quickly noticed that the ISHAInput takes up a significant proportion of the time (**Figure 4**). This was first done sequentially, where each call to the ISHA functions were commented out, then opened from top-to-bottom. Another method consisted of only uncommenting one at a time (Figure 5). Using both methods it is clear that the ISHAInput method takes up majority of the call time.

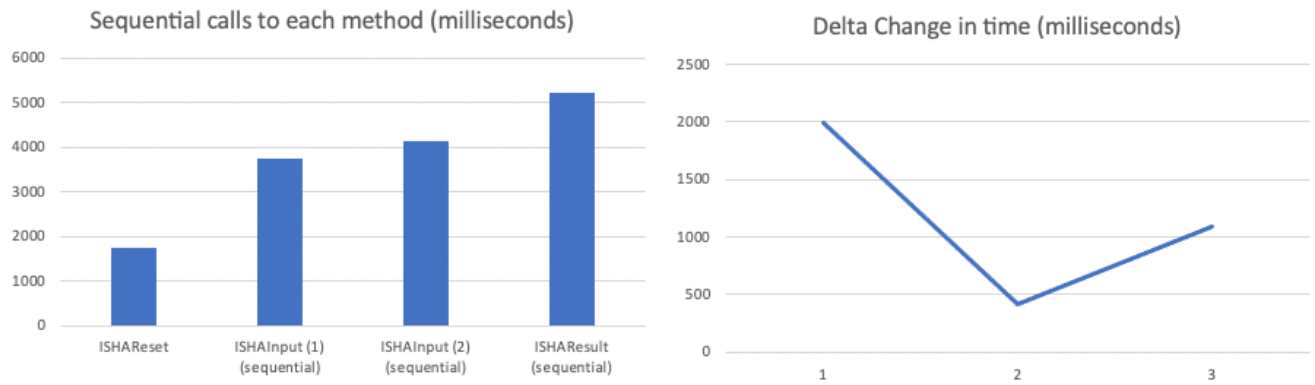

Figure 4 - Amount of time per function. Here the functions were sequentially ran and timed (left) in order to look for the delta (right).

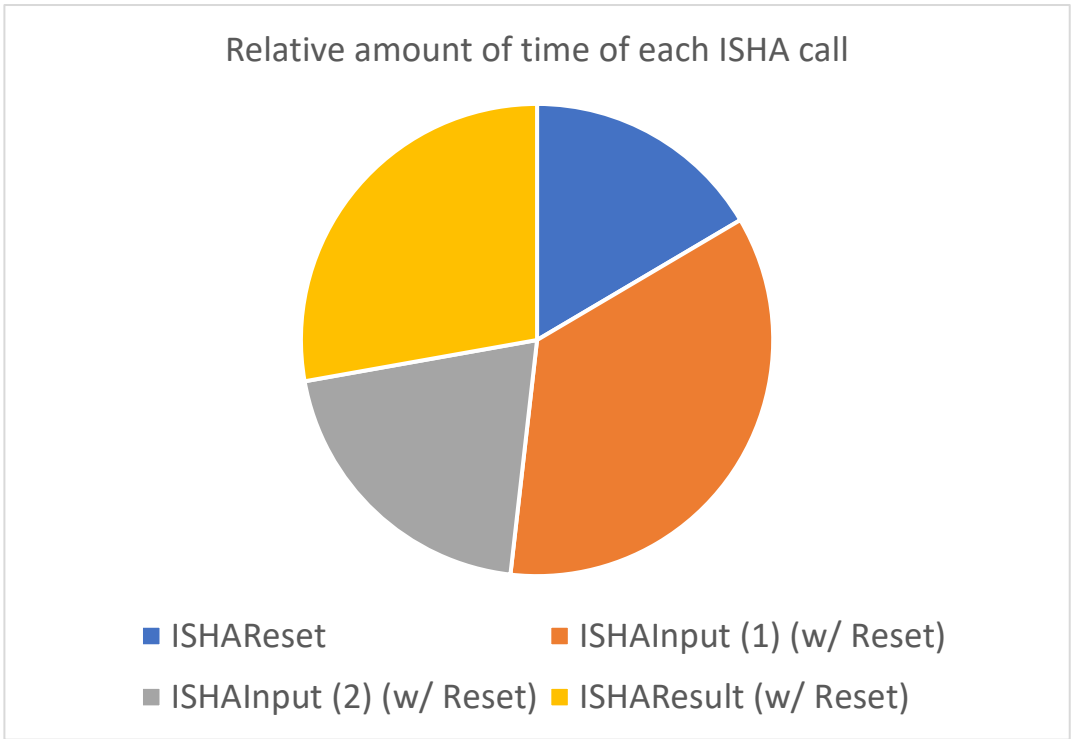

Figure 5 - relative proportion of time needed per function when tested seperately.

Digging further into the ISHAInput function, the following metrics are obtained for each section of the function:

ISHAInput – 3740 msec -> 1990 msec
Function call – 12 msec
While loop – 110 + 248 = 348 msec
ISHAProcess (**looped**) – 700 msec =>
If statements (looped) – ~1500 msec [773+740]

Shrinking each of these subcalls may lead to a dramatic improvement in performance.

ISHAInput (1): [with ISHAReset (1)]
time_pbkdf2_hmac_isha: 4096 iterations took 3740 msec
→ If **ISHAProcessMessageBlock**() commented out.
time_pbkdf2_hmac_isha: 4096 iterations took 3039 msec
→ If top portion of while loop commented out
time_pbkdf2_hmac_isha: 4096 iterations took 2227 msec
➔ If both commented out (i.e. empty while loop)
time_pbkdf2_hmac_isha: 4096 iterations took 2110 msec
→ If empty function
time_pbkdf2_hmac_isha: 4096 iterations took 1762 msec

Limitations to code base with optimizations

hmac_isha()
This functions while loop was completely unraveled, and the 'A' variable was put onto a register for speed improvements. The variable t was removed. Bswap32 is used to turn the array into the correct endian.

F()
The while loop was unraveledthat way the inner loop did not need to be used. This is a limitation because it caused a lot of overhead.