Project Phase 1: A Detailed Analysis of SUPERCOP on the Dragonboard APQ8060.

Kevin Burns, Robert Lyerly, Reese Moore, Philip Kobezak
Virginia Polytechnic Institute & State University
1185 Perry Street, Blacksburg VA
{kevinpb, rlyerly, ram, pkobezak}@vt.edu

1 Introduction

Computers are a ubiquitous part of our society. As computers become increasingly connected, more of our daily lives are becoming digitized. As such, it is important that we find new ways to ensure security and privacy. The field of Cryptography involves designing algorithms and protocols as a means of ensuring services interact with each other in a secure, private way. Traditionally, cryptographic theory was used to develop algorithms that were highly efficient and very secure on large desktop CPUs. However, with the dawn of mobile computing there is an increased need for power-aware cryptography. Research in this field seeks to strike a balance, searching for algorithms that can be used in low-power settings while still being strong and secure. New cryptographic algorithms must be evaluated on a wide variety of hardware platforms to understand how they perform in practice.

Hash functions play an important part in cryptography. They are heavily used for authentication and digital signing, two components of cryptography that are necessary for information security. Designing a good hash function is a complex task - the algorithm must have several characteristics such as uniformity, efficiency and infeasability of reversing the hash. The National Institute of Standards and Technology (NIST) is responsible for maintaining many cryptographic algorithms, including hash functions. NIST recently held a competition to generate an alternative to SHA-1 and MD5 because of known attacks for these hash functions. Many different researchers submitted implementations to the competitions, and the Keccak algorithm was chosen as the official implementation for SHA-3 on October 2, 2012. However, the software hosted at http://bench.cr.yp.to contains all the SHA-3 implementations submitted to NIST so that anybody may test them. In the first phase of our project, we were tasked with benchmarking these submissions.

1.1 Hardware Overview

Our platform for the first phase of the project was a Snapdragon S3 APQ8060-based Dragonboard, used for prototyping and developing for the Android plat-

form (hereafter referred to as "the Dragonboard"). The Dragonboard implements a complete wireless phone system, including a wireless RF card, a sensor card (with accelerometer and gyroscope) and a touchscreen. The Snapdragon S3 APQ8060 contains several cores for computation and processing:

1. ARM1136J-S 384 MHz embedded microprocessor

2. Qualcomm dual-core Scorpion microprocessor (up to 1.7 GHz), which has the ARM NEON SIMD extensions

3. Qualcomm QDSP6000 and QDSP4000 DSP cores

4. ARM7 resource and power management microprocessor

5. Adreno 220 GPU

Most compute-intensive tasks are run on the larger Scorpion cores (which are designated as "application cores"), the QDSP6000 DSP and the Adreno 220 GPU. The ARMv7 instruction set architecture (ISA) is the 7th-generation of the ARM ISA. It is a RISC ISA and contains a standardized 3-stage pipeline (although implementations may contain longer pipelines) and 16 x 32-bit registers. ARMv7 also defines the NEON SIMD extensions which specify the interface to a 128-bit SIMD core with an independent pipeline and register file. It can perform basic arithmetic and logic operations on varying size data types, including signed/unsigned 8-bit, 16-bit, 32-bit or 64-bit words. The QDSP6000 (or "Hexagon") DSP is a programmable DSP designed for application use. It has several features which make it a flexible platform, including symmetric multi-processing and VLIW/SIMD instructions (in fact, Linux has been ported to run on the Hexagon). The Adreno 220 GPU is used for 2D and 3D rendering on Android. It implements several graphic APIs and can be used concurrently by several of the other cores for interleaving CPU, DSP and graphics operations.

1.2 Problem Statement

The objective of the first phase of the project was to do a detailed perfomance profiling for a set of hash algorithms provided by the SUPERCOP testsuite. The analyses should try to determine why each algorithm implementation excelled, compared to other implementations, on the target platform (Dragonboard APQ8060).

1.3 Partitioning

1.4 Profiling

2 Algorithms

2.1 blake

The BLAKE hash function is based on the ChaCha stream cipher, with an additional XOR each round. The way it works is, the ChaCha algorithm works with a 4 by 4 array of words. BLAKE combines an 8-word hash value with 16 words from the message repeatedly, in turn truncating the ChaCha result

to obtain the next hash value.The BLAKE hash function has two variants, a variant with a word size of 32 bits and a variant with a word size of 64 bits. The first two hash functions we evaluate, blake32 and blake256, are of the 32 bit word variety.
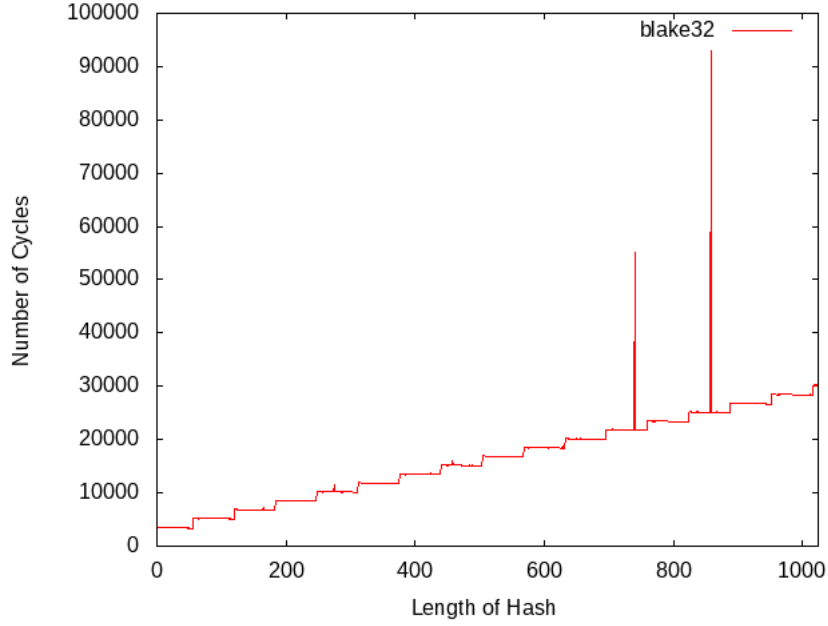
2.1.1   blake32

Figure 1: BLAKE with 32-bit words, 10 rounds, and 256-bit output

From the graph above we see that there is a stepwise characteristic. The steps seem to increment every time the hash is divisible by 64. This characteristic is present due to the fact that during each iteration 2 words are taken from the message at one time. So a total of 64 bits of the actual message is processed. This happens 8 times a round, therefore 512-bits are processed per round. So, as our hash size increases by multiples of 64 bytes we will see more and more iterations having to be processed.
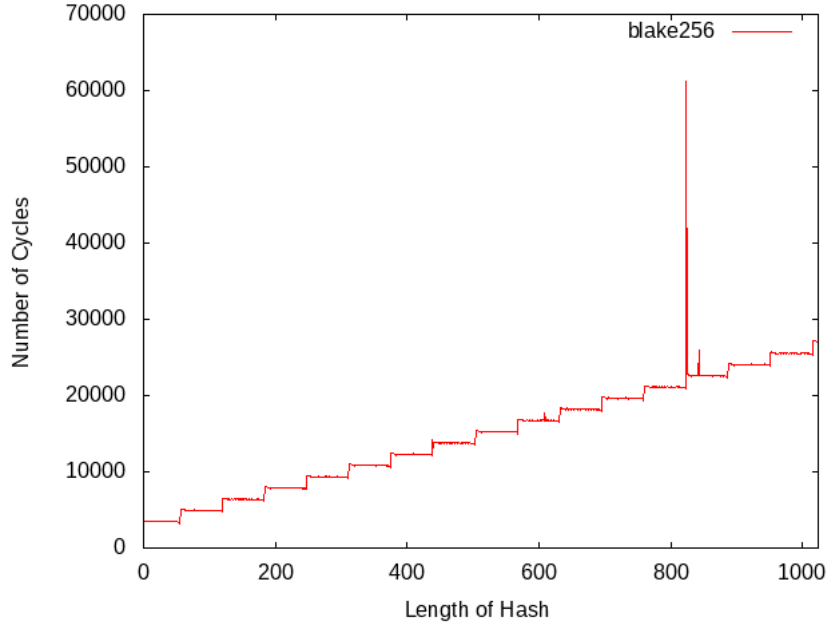
2.1.2 blake256

Figure 2: BLAKE with 32-bit words, 14 rounds, and 256-bit output

The graph above shows a very similar pattern to the blake32 graph. The reason is that blake256 also uses a 32 bit word size. The difference we see between the blake32 and this graph is the number of cycles. When analyzing the data I found that the cycles per byte for blake32 was twice that of blake256. This seems counter intuitive as blake256 has an additional 4 rounds to compute each iteration. I speculate that this is due to the compiler options chosen by the script we ran to find the fastest implementation of each algorithm. The options chosen for blake256 were (compiler gcc -mcpu=arm1136jf-s -O -fomit-frame-pointer -fno-schedule-insns 4.4.5) versus the options for blake32 (gcc -mcpu=arm9tdmi -O2 -fomit-frame-pointer 4.4.5). As we can see the architecture chosen by the script is not the target architecture and could lead to a number of mis-optimizations.
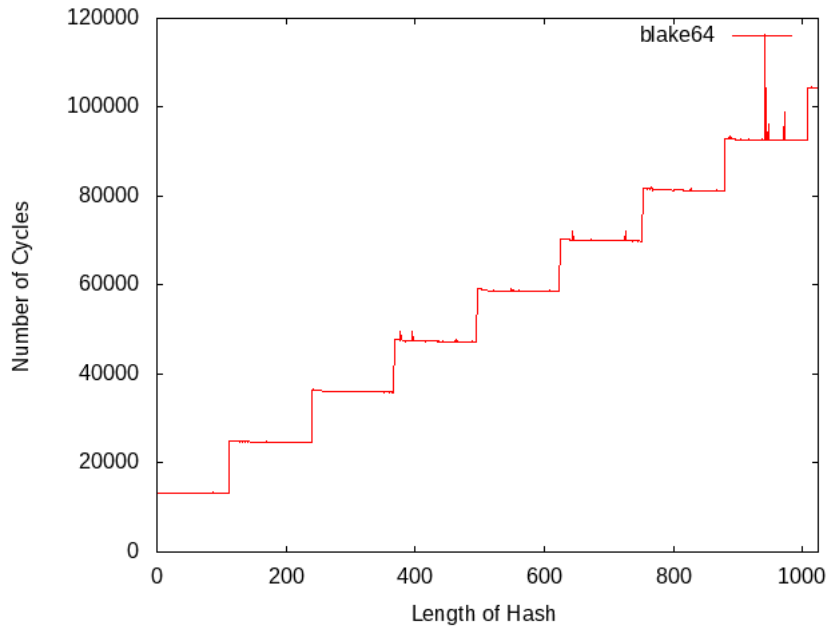
2.1.3 blake64

Figure 3: BLAKE with 64-bit words, 14 rounds, and 512-bit output

The blake64 hashing function increases the word size to 64 bits. Since nothing else changes we see that the widths of each step in our stepwise graph is doubled, so 1024 bits long. The cycles per byte were higher than that of the 32-bit word sizes. This is due to the fact that the target processor is a 32 bit architecture and does therefore doesn't have a chance to utilize 64 bit words.
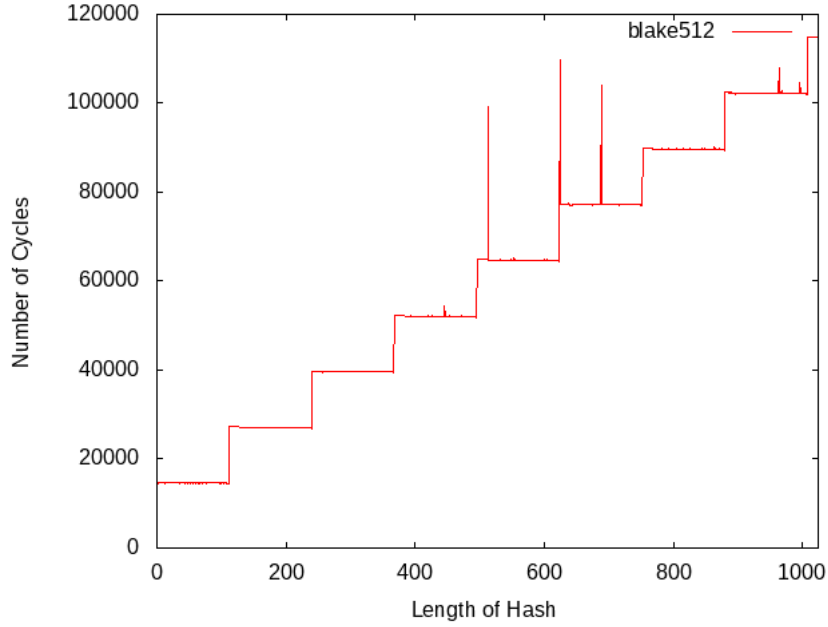
2.1.4 blake512

Figure 4: BLAKE with 64-bit words, 16 rounds, and 512-bit output

The difference between blake512 and blake64 is the number of rounds. This would explain the identical step width, 128 bytes, along with the increase in cycles for blake512.

2.2 cubehash

I won't explain the details of cubehash here. I will instead talk about how the choice of variable sizes in the cube hash equation is reflected on the graph, along with the possible influence from the architecture. If we represent the equation as CubeHashi+r/b+fh(m). Where i is the number of initialization rounds, r is the number of rounds per message block, b is the length of the blocks taken from the message, f is the number finalization rounds, h is the number of output bits, and m is the message. We're given that r is 8 rounds per block, b is 16 bytes in a block, and h is 512 bits of output.
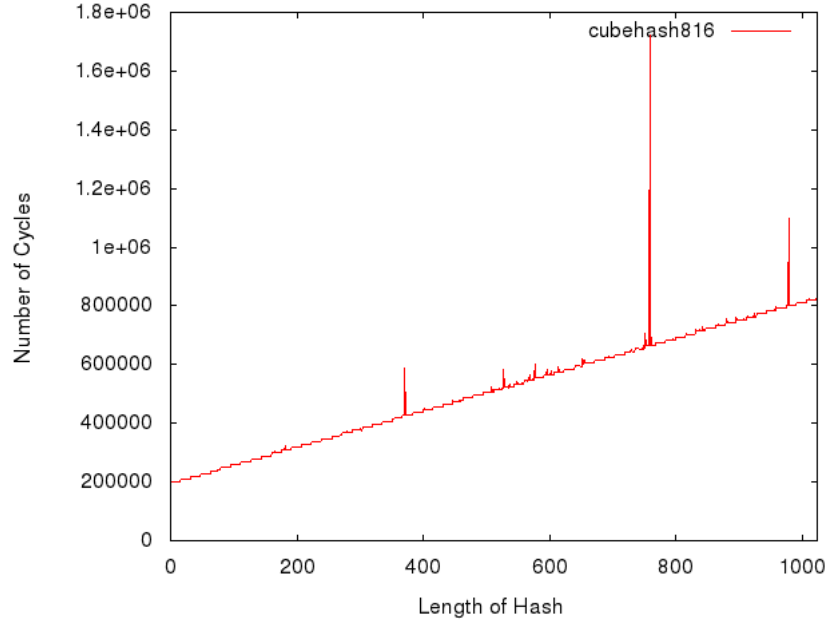
2.2.1 cubehash816

Figure 5: CubeHash8/16 with 512-bit output

It is a little hard to see in the above graph, but it takes on a similar shape to the graphs found in the blake algorithms. That is, it is as step like characteristics. The length of each step is 16 bytes, 128 bits. This appears to be from the size of the blocks processed each round. Each block is 16 bytes as described above. When the message increments by a multiple of 16 we see a rise in the step. As that's one more block that needs to be processed. Our target platform can provide 16, 32bit registers for the processing each block. This may be a responsible for the given speed.

2.3 groestl

The groestl algorithm is another block cipher that is based from AES. This function works with l-bit blocks where l is 2n and n is the number of output bits. In this case n is 256 and 512 as denoted in the names.
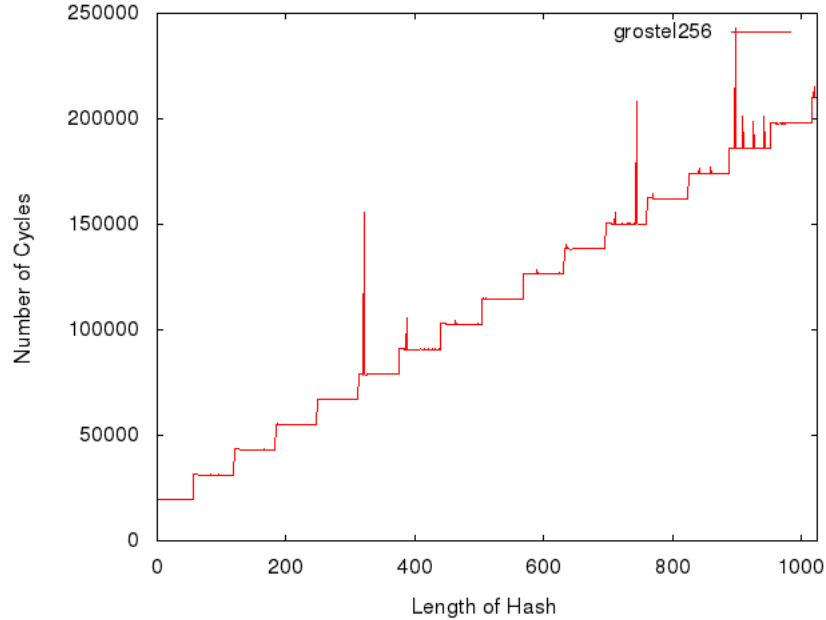
2.3.1 groestl256

Figure 6: Grstl with 256-bit output

The graph above also shares the step like characteristics of the other graphs. Each step for this plot is 64 bytes, 512 bits in length. From what we know about groestl, it works with l-bit blocks, where we derived above l is 512 bits. We can then correlate this to the nature of the step size witnessed in the graph above.

2.4 groestl512

2.5 JH

The JH family of hash functions are a set of functions that were submitted to the NIST SHA-3 competition. The submission defines four specific hash algorithms: JH-224, JH-256, JH-384, and JH-512. All of these separate functions are implemented through a very similar construction, and as such one would expect them to exhibit similar performance properties as well as optimization opportunities.

Wu [?] describes the hash function family in his submission to the NIST competition. JH is constructed through a simple to describe compression function that is shared by all variants. The compression function maintains 1024 bits of internal state and operates on 512 bit blocks at a time. Even before doing performance analysis, this would indicate that we should expect to see a jump in the number of cycles required every 64 bytes, as at those byte boundaries the function has to process another block of data. The differences between the functions are in how the context of the hash is seeded and how much of the data

is taken off the end to form the digest, and so we would expect to see similar performance across all of the JH functions.

2.5.1 JH-224
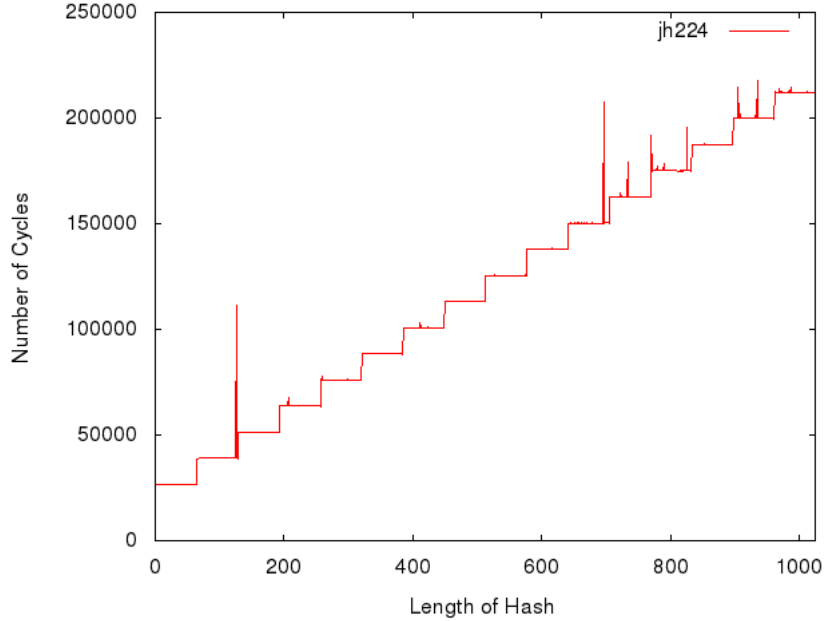

Figure 7: JH-224 with 224-bit output

The algorithm selected for benchmarking JH-224 was bitslice_opt32 compiled with `gcc -mcpu=arm920t -O2 -fomit-frame-pointer` with GCC version 4.4.5. This choice makes sense given the 32 bit word length of this Snapdragon processor, and the fact that JH is designed such that bitslicing is an effective optimization technique [**?**].

We note that the prediction that there would be a jump in the number of cycles used every 64 bytes was confirmed. We further note that each jump appears to be roughly equivalent, implying that they are attributable to similar phenomena, further confirming our suspicions.

Wu [**?**] gives details about the bit-slicing implementation used for JH. The implementation he defines performs bit-slicing optimizations on components of the 512-bit block used in the encryption permutation, and leaves the overall construction largely unaffected. This explains why we still see the 64 byte block boundaries playing an important roll in defining the shape of the performance graph.
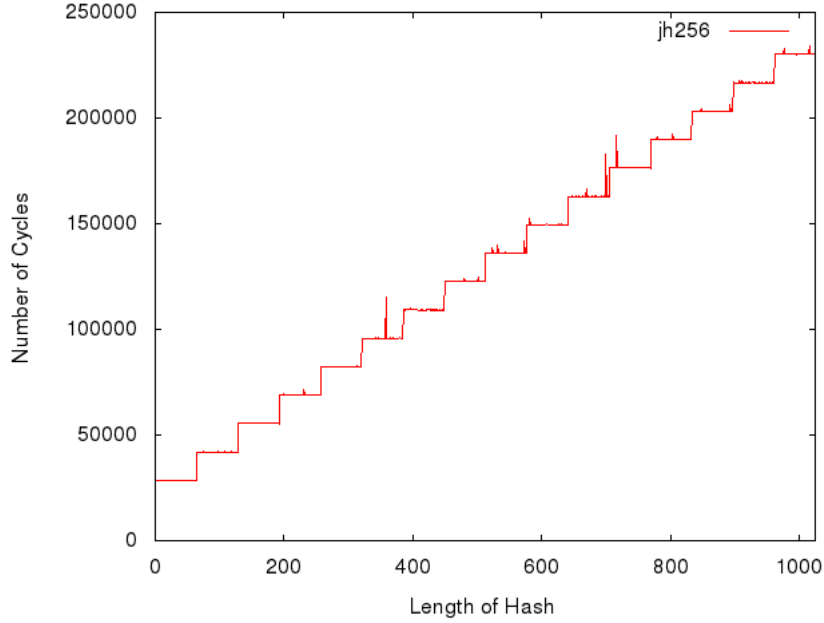
2.5.2 JH-256

Figure 8: JH-256 with 256-bit output

The algorithm selected for benchmarking JH-256 was bitslice_opt32 compiled with `gcc -mcpu=arm8 -O3 -fomit-frame-pointer` with GCC version 4.4.5. Again, this choice of algorithm makes sense given that the Snapdragon processor has a 32 bit word length. Interestingly, different optimization options were selected during the profiling process for JH-256 than for JH-224, however this would lead us to believe that this implementation is designed in such a way that the compiler is not given enough leeway to make meaningful optimizations based on the differences.

We again note that our predicted graph shape held, and that the growth of this function is roughly similar to that of JH-224. This is what we would expect given that all variants of JH share a compression function.
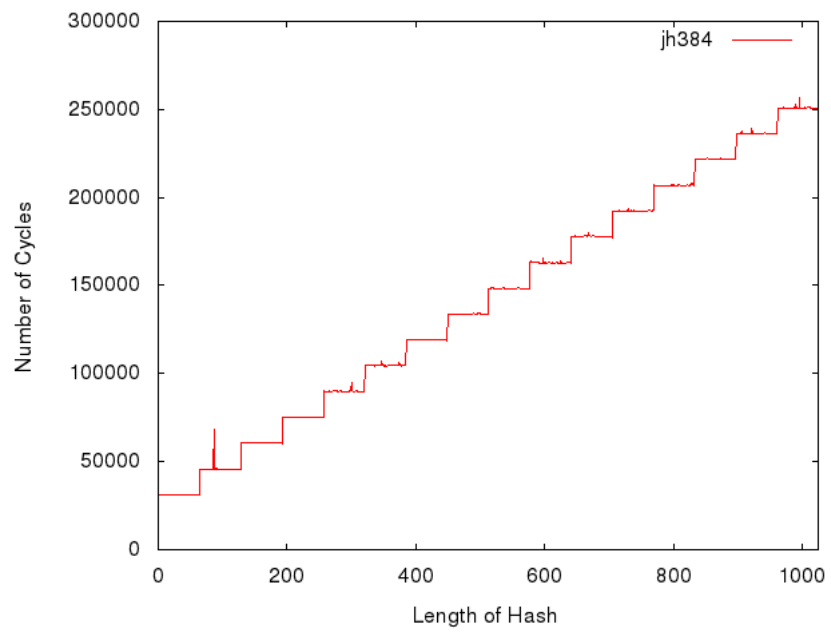
2.5.3 JH-384

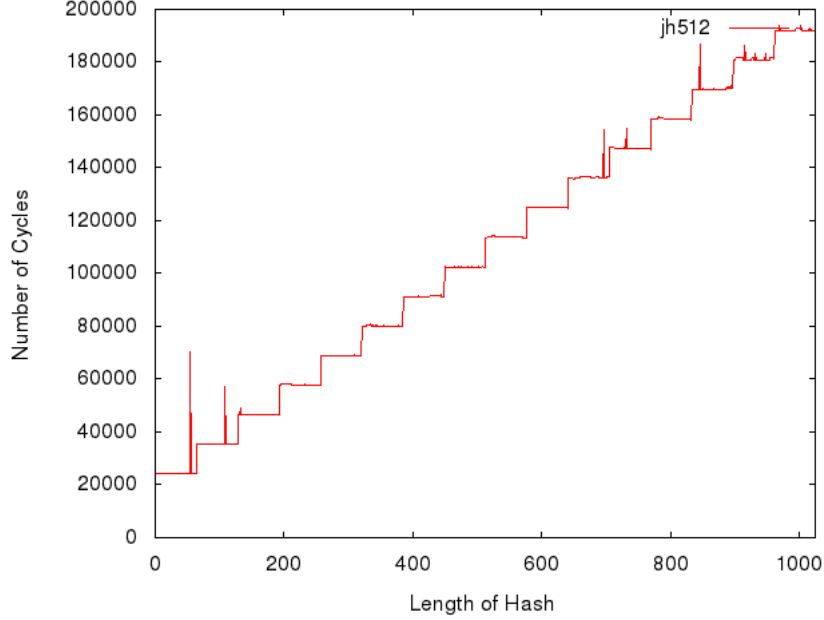
Figure 9: JH-384 with 384-bit output

2.5.4 JH-512

Figure 10: JH-512 with 512-bit output

The algorithm selected for benchmarking JH-512 was bitslice_opt32 compiled with `gcc -mcpu=cortex-a9 -mfloat-abi=softfp -mfpu=neon -Os -fomit-frame-pointer` with GCC version 4.4.5. This optimized algorithm makes sense for the 32 bit Snapdragon processor, and given that JH is easily optimized through bit-slicing. In this instance we have a different set of optimization parameters, we do note a slight drop in the cycles consumed at the high levels so perhaps optimization for the NEON FPU is beneficial or optimization for size allows better management of the cache. Regardless, the performance graph is very similar to the other graphs both in size and growth rate, and this can be attributed to the shared compression function used across all lengths of JH.

2.6 Keccak

Keccak is an entry into the NIST SHA-3 cryptographic hash competition, and ultimately was chosen by NIST to become SHA-3. Keccak describes a family of cryptographic hash functions designed as a sponge construction. Bertoni et al. describe Keccak in [**?**]. They build the specific variants of Keccak on top of a permutation function called Keccak-f[b].

Variants of Keccak are primarily defined by three security parameters: the width of the Keccak-f permutation: b, the capacity (the probability of success for any generic attack against a sponge construction): c, and the number of rounds in Keccak-f: n_r [**?**]. The choice of these parameters has an effect not only on the security level of the Keccak variant, but also the performance. r defines the

bitrate of the sponge function, that is to say how many bits the sponge function processes at a time, and accordingly how many calls to the Keccak-f function are going to be made for a bitstring of a given length. Therefore, the r value is directly proportional to what we should see as the length between upward steps in the number of cycles required to process data.

Interestingly, the designers say that although Keccak-f is standardized on having a width of 1600 bits, this is optimized for a 64 bit processor. They suggest that users of 32 bit and even embedded processors tune the function to have a smaller width, such as 800 to obtain better performance without totally sacrificing security [**?**].
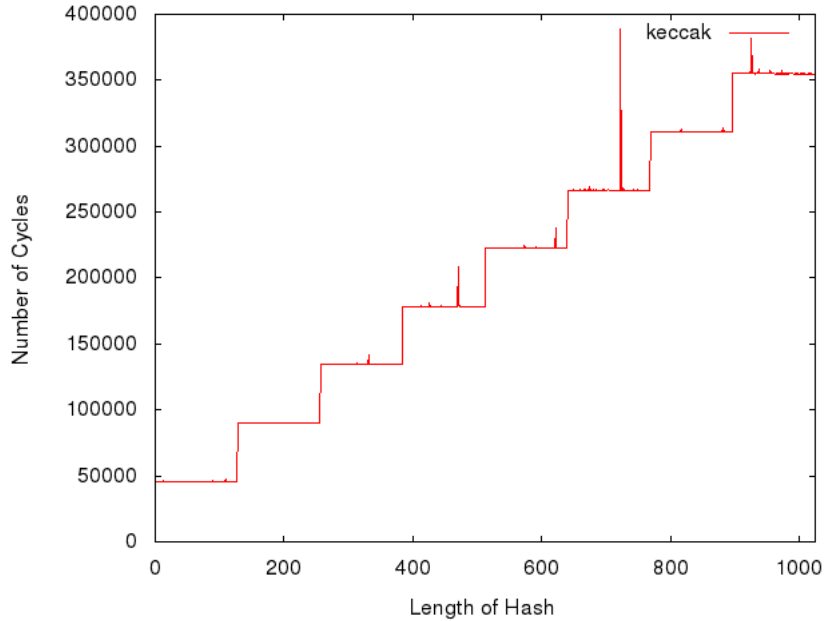
2.6.1 keccak

Figure 11: Keccak[]=Keccak[r=1024,c=576,nr=24] with 1024-bit output

The algorithm selected to benchmark Keccak was compact, which had been compiled with `gcc -funroll-loops -O -fomit-frame-pointer` with GCC version 4.4.5. This version of Keccak has the parameters $r = 1024, c = 576, n_r = 24$.

We note the stair stepped function that we associate with an additional call to the underlying Keccak-f permutation function. We note that the steps occur every 128 bytes, which is in line with our assumption that they would be defined by the bitrate $r = 1024$.
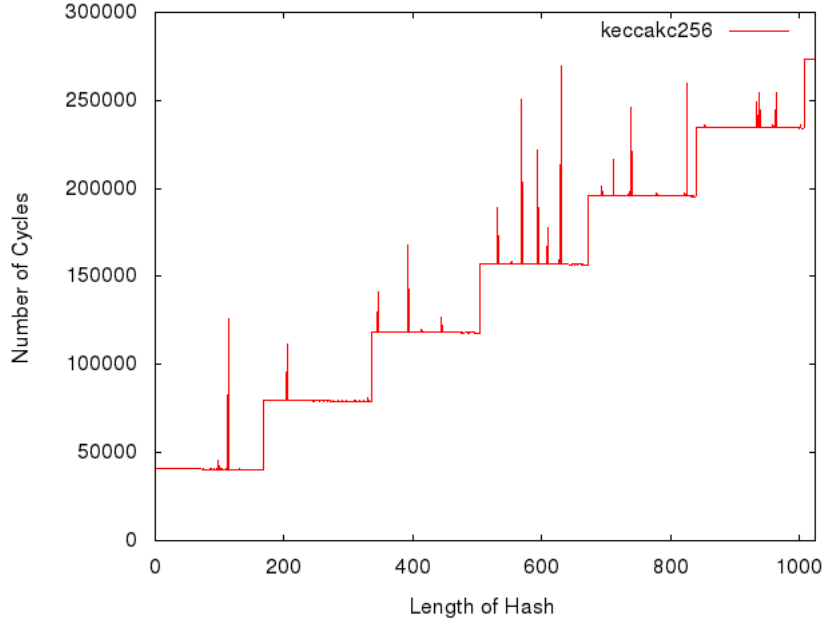
13

2.6.2 keccakc256

Figure 12: Keccak[r=1344,c=256,nr=24] with 1344-bit output

The algorithm selected to benchmark Keccakc256 was compact, which had been compiled with `gcc -mcpu=cortex-r4 -O2 -fomit-frame-pointer` with GCC version 4.4.5. This variant of Keccak has the parameters $r = 1344, c = 256, n_r = 24$.

This is the same implementation style as Keccak, which is unsurprising given that all versions of Keccak are built on a similar construction. Additionally, this version of Keccak has similar parameters to Keccak above. The longer bit rate correctly predicts the longer period between steps on the graph when compared to the above implementation of Keccak
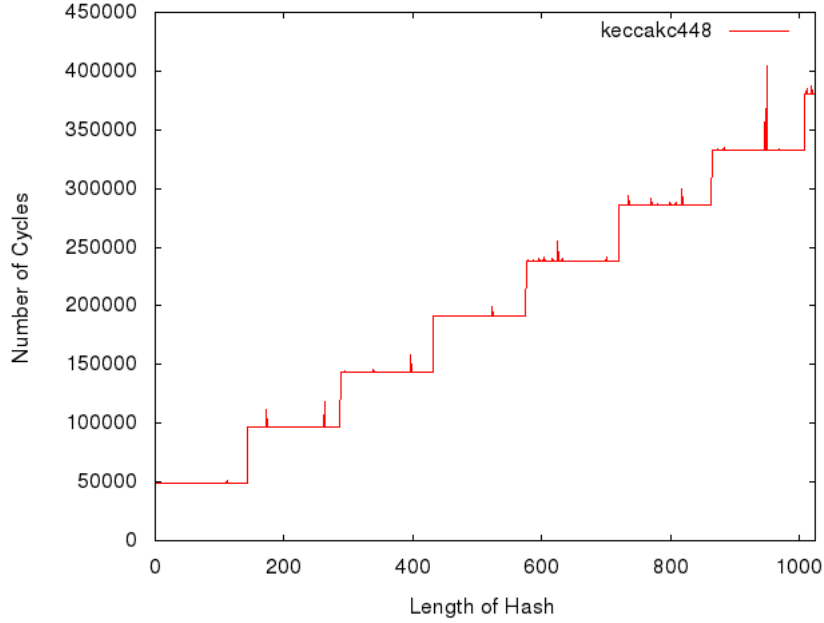
14

2.6.3 keccakc448

Figure 13: Keccak[r=1152,c=448,nr=24] with 224-bit output

The algorithm selected to benchmark Keccakc448 was compact, which had been compiled with `gcc -Os -fomit-frame-pointer` with GCC version 4.4.5. This variant of Keccak has the parameters $r = 1152, c = 448, n_r = 24$.

Unsurprisingly, this variant of Keccak uses the same implementation as the other variants studied so far. This is because the Keccak family uses the same construction internally, so optimizations from one variant for a particular processor will work fairly well on another.

This run shows the stair stepped pattern as well, but it is a shorter period between the steps. This is a result of the lower bit rate of this variant.
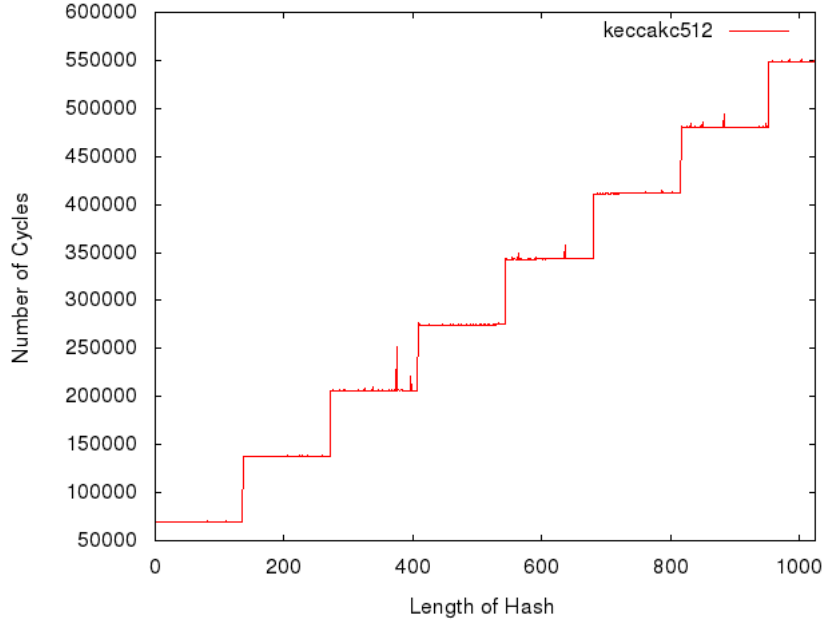
2.6.4   keccakc512

Figure 14: Keccak[r=1088,c=512,nr=24] with 256-bit output

The algorithm selected to benchmark Keccakc512 was compact8 which had been compiled with `gcc -mcpu=strongarm1100 -O3 -fomit-frame-pointer` on GCC version 4.4.5. This variant of Keccak has the parameters $r = 1088, c = 512, n_r = 24$.

Interestingly, this implementation is not what the other implementations had been compiled with, though it is a similar implementation.

The prediction regarding the stair stepped pattern as well as the period between the steps being proportional to the bit rate however was upheld regardless of the differences in version. This is because the amount of work being done is somewhat inherent in the algorithm, and these implementations are very similar.
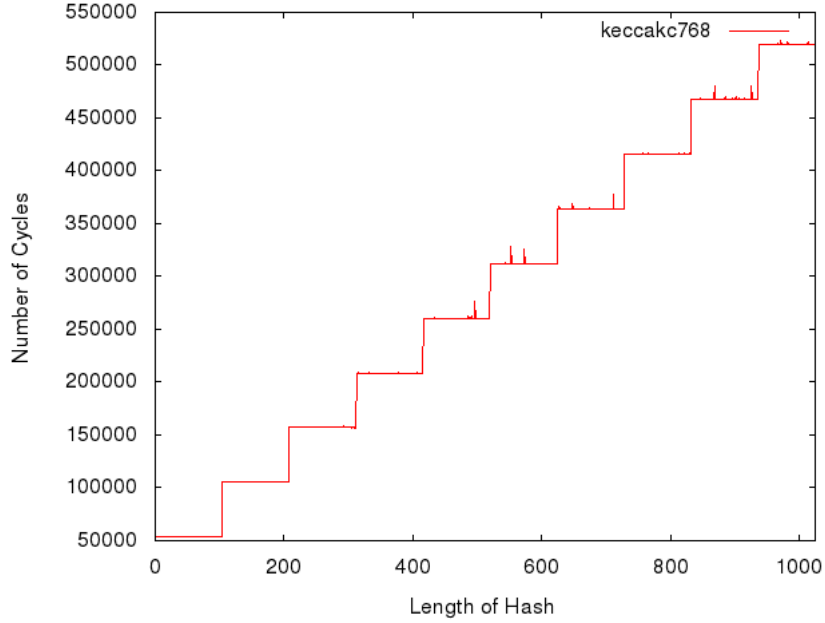
2.6.5 keccakc768

Figure 15: Keccak[r=832,c=768,nr=24] with 384-bit output

The algorithm selected to benchmark Keccakc768 was compact which had been compiled with `gcc -mcpu=strongarm1100 -O -fomit-frame-pointer` on GCC version 4.4.5. This variant of Keccak has the parameters $r = 832, c = 768, n_r = 24$.

A return to implementations of Keccak being compiled with compact on this board, this run exhibited the exact same properties that we have been seeing before. The stepped pattern is now shorter still because of the direct correlation with the bit rate r. This performance property is a property of the algorithm family.
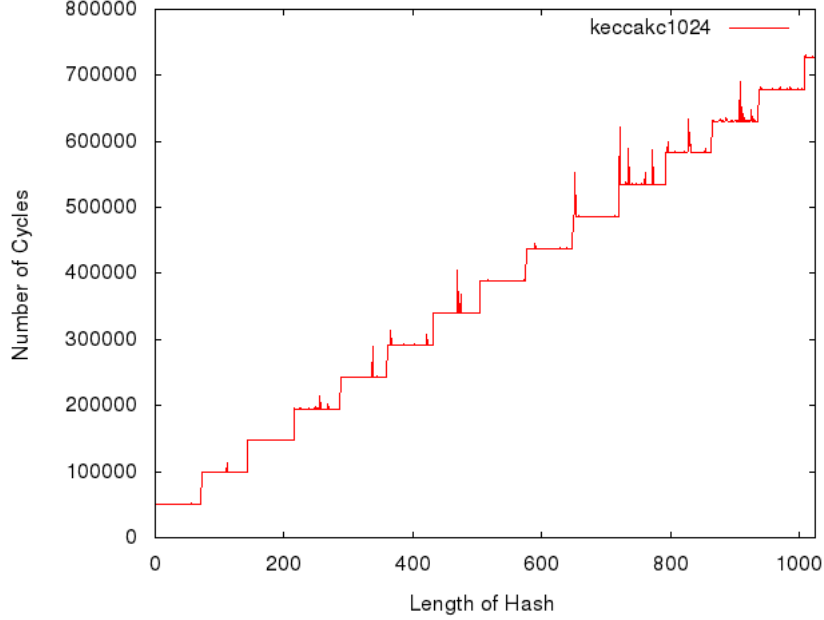
2.6.6   keccakc1024

Figure 16: Keccak[r=576,c=1024,nr=24] with 512-bit output

The algorithm selected to benchmark Keccakc1024 was compact which had been compiled with `gcc -mcpu=cortex-a8 -mfloat-abi=softfp -mfpu=neon -O -fomit-frame-pointer` on GCC version 4.4.5. This variant of Keccak has the parameters $r = 576, c = 1024, nr = 24$.

The version of Keccak with the shortest bit rate that we tested, and it still upholds the prediction that the bit rate is directly correlated with the period between the steps. In this variant, the jumps in cycle count is very rapid, causing the algorithm to consume a much higher number of clock cycles by the time it has finished compared to the lower capacity variants of Keccak.

2.7   md5

2.8   mgrøstl256

The modified-Grøstl (mgrøstl) algorithm is a modification of the Grøstl hash algorithm chosen as one of the five finalists of the SHA-3 competition. Similar to other hash algorithms, an arbitrary length input is padded so that it can be divided into 1024-bit blocks. The algorithm works by performing a compression function on each block; the compression function involves additions and permutations that reduces the 1024-bit input to a 512-bit intermediate hash. The output from each of these compression operations serves as input to the next block with a final reduction that shrinks the message to a digest size of 256 bits.
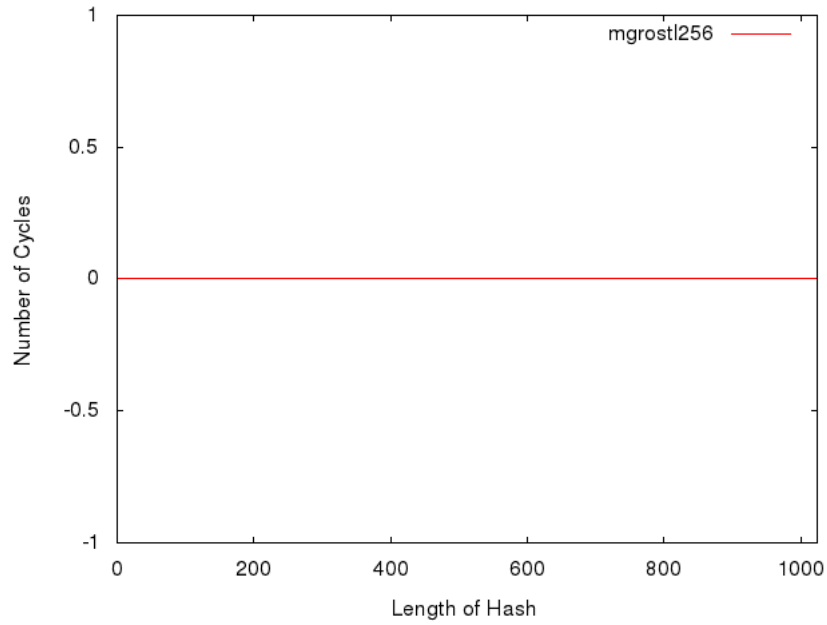
Figure 17: Modifided-Grøstl with 256-bit output

2.9 sha256

The sha256 and sha512 algorithms are specific implementations of the previous generation SHA-2 hash function defined by NIST. After effective attacks were found on SHA-1, NIST moved to a better hash algorithm, one which features several improvements (including fewer collisions and no known effective attacks). Four variants of the algorithms were standardized based on the number of bits produced in the digest (including 256 and 512 bits for sha256 and sha512, respectively).
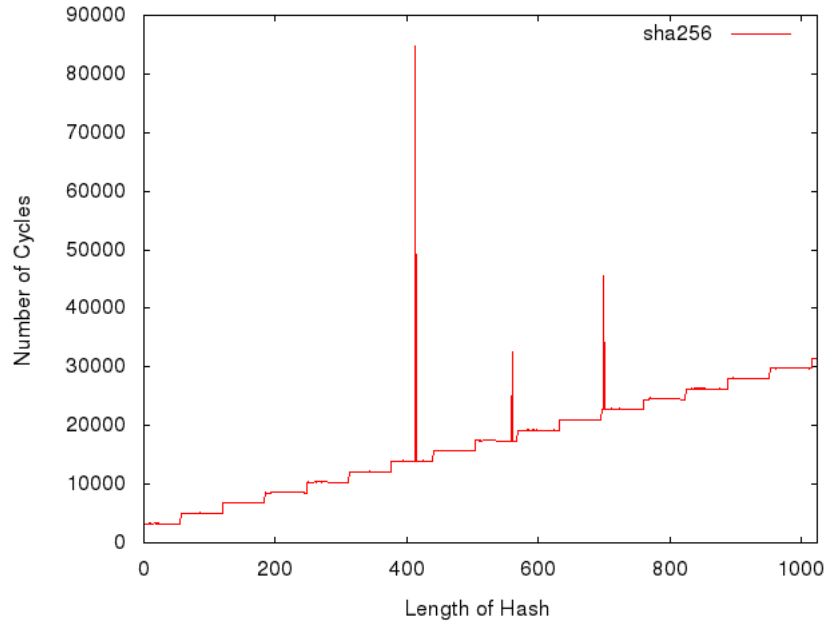
Figure 18: SHA-2 with 256-bit output

The first algorithm, sha256, works by chopping a message into 512-bit blocks and hashing each block down to 256-bits before continuing on to the next. The result of entire hash function is added together with the original hash value, producing the final digest. As is evident from the graph, performance models a step function with a step size of 64 bytes. The number of blocks needed to represent the message is $\lceil$message length$/64\rceil$ (this converts the number of block into an integer multiple of 64). As soon as the message requires a higher multiple of 64 bytes, the algorithm must process another block of data. Therefore, for every additional 64 bytes of data, another additional 64 rounds of SHA-2 must be performed.
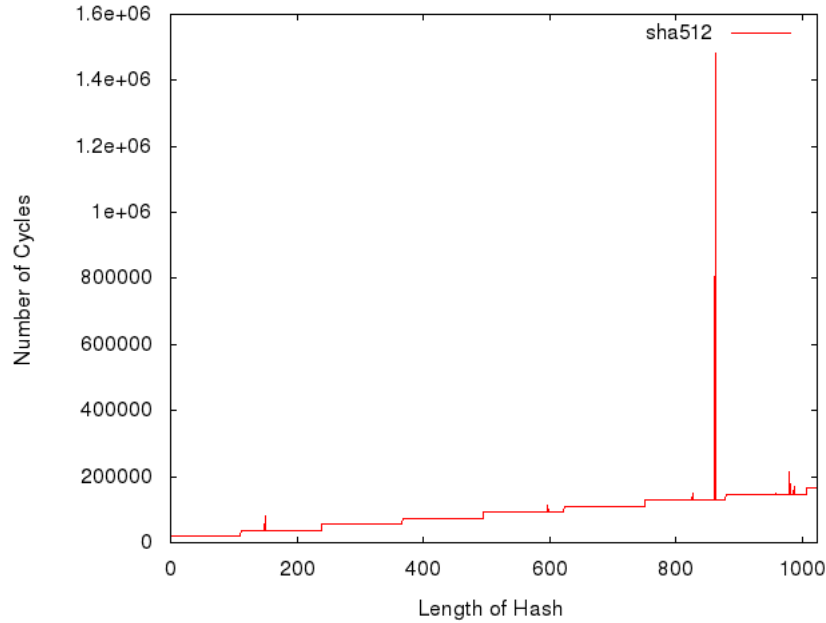
2.10   sha512

Figure 19: SHA-2 with 512-bit output

The second SHA-2 algorithm benchmarked, sha512, again produces a step function; closer inspection reveals this graph has slightly different characteristics than the one produced by running sha256. There are some fundamental changes in the underlying algorithm - first, the digest length is 512-bits and the block size is 1024 bits. This means that for every additional 128 bytes needed to represent the message, the algorithm must process an extra block of data - hence the step size of 128 bytes. Because this block length is longer the algorithm must perform more calculations per round, leading to a higher cycles per byte. Additionally, sha512 performs 80 rounds versus the 64 performed by sha256, leading to another increase in cycle count.

2.11   skein256256

The Skein family of hash functions were submitted to the competition by a varied group of individuals and was one of the five finalists for SHA-3. The Skein hash functions are based on a tweakable block cipher, named the Threefish block cipher. The Threefish block cipher favors more rounds versus higher complexity; it only performs XOR, addition and shift operations. It uses keys (which are the same length as the block) and a 128-bit tweak value to generate the key schedule. These generated keys are then injected every couple of round of the "Mix" operation (XOR, addition, shift); this entire sequence constitutes one round, with the Skein hash performing varying number of such rounds depending on the block size. One final feature of the Skein family of hash functions is the

21

ability to have arbitrary length outputs, implemented using UBI chaining mode. Because these hash functions work on 64-bit words, it is not well-suited to the 32-bit architecture of the Scorpion processors. However, the ARM NEON SIMD instructions can be used to perform many of the round calculations to speed up computation.
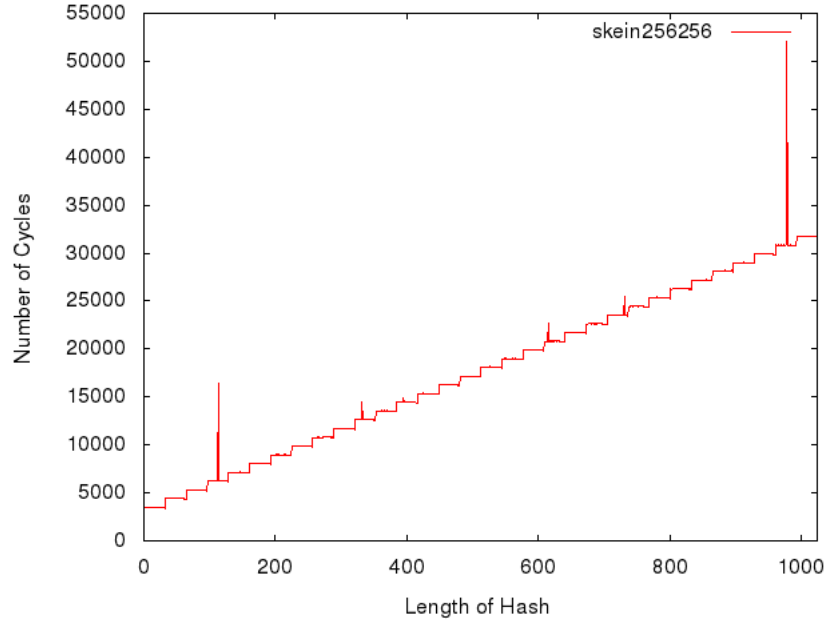
Figure 20: Skein-256 with 256-bit output

This instance of the Skein hash function has a block size of 256-bits (32 bytes) and produces a 256-bit digest. In this instance, the block cipher operates on 32 byte blocks; naturally, with every higher multiple of 32 a new block must be processed, leading the to the step function represented in the graph. Because the block size is smaller, there are fewer calculations for the block cipher, resulting in shorter runtimes in comparison to the other Skein hash functions.
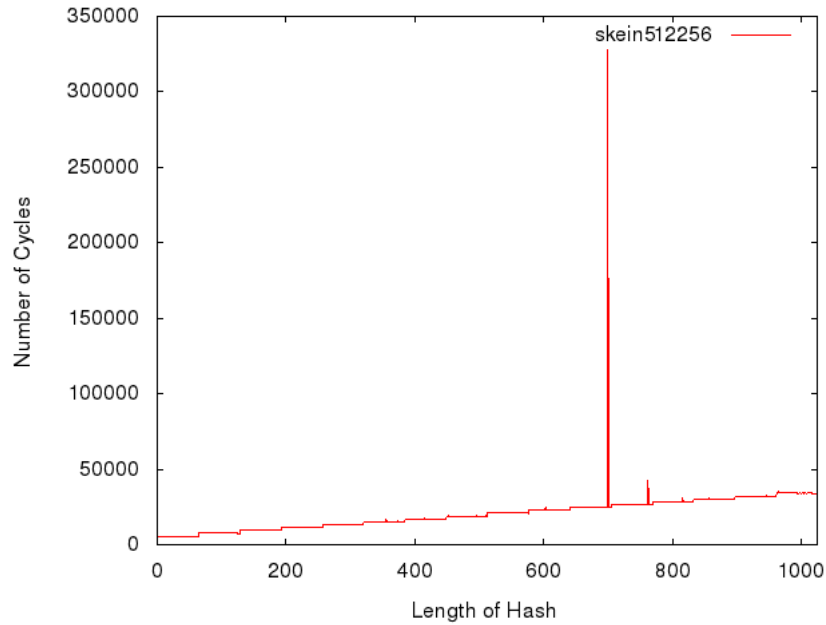
2.12   skein512256

Figure 21: Skein-512 with 256-bit output

This instance of the Skein hash function has a block size of 512-bits (64 bytes) and produces a 256-bit digest. Again, because the block size is 64 bytes, every higher multiple of 64 bytes creates an extra block which must be processed, similar to the previous Skein functions. The performance of skein512256 is very comparable to the performance of skein256256. This is because although there are twice as many operations per block in skein512256 (it must handle 64 bytes instead of 32) but there are half as many blocks to process, hence a comparable number of operations must be performed for each algorithm. The step function is more coarse-grained than skein256256 because of the word size.
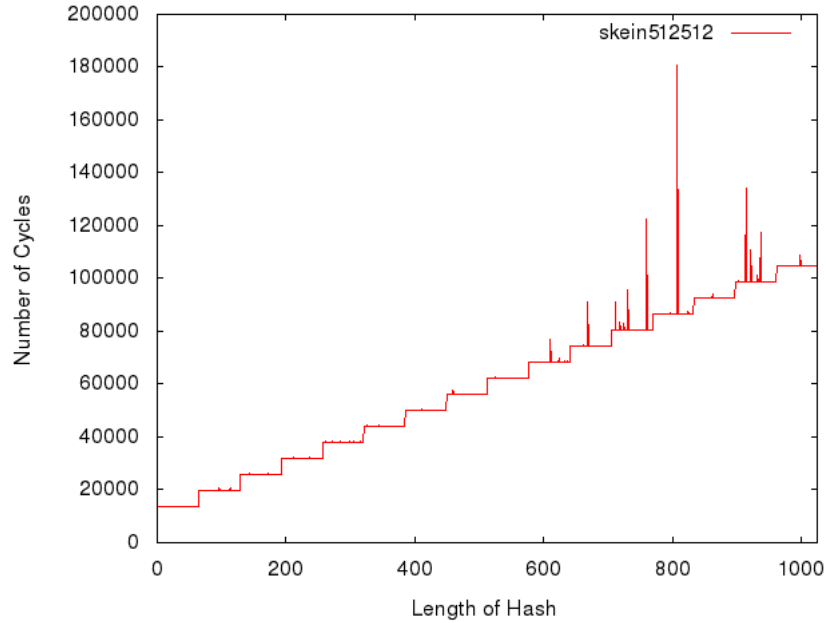
2.13 skein512512

Figure 22: Skein-512 with 512-bit output

This instance of the Skein hash function has a block size of 512-bits (64 bytes) and produces a 512-bit digest. Similar to the other Skein hash funcions, the step size is dictated by the block size; its length is 64 bytes because this instance of the Skein hash function works on 64 byte blocks. This algorithm shows perfomance slower than skein512256, however - this is because of the output length of the hash. Skein hash functions use Unique Block Iteration (UBI) chaining to produce the variable size output. Part of this chaining involves XOR'ing message blocks together with blocks run through the Threefish cipher, and using that output as the input to the next block's chaining. Because a skein512512 must XOR more words together when chaining the blocks (because the block size is larger), UBI chaining causes performance penalties in comparison to skein512256.
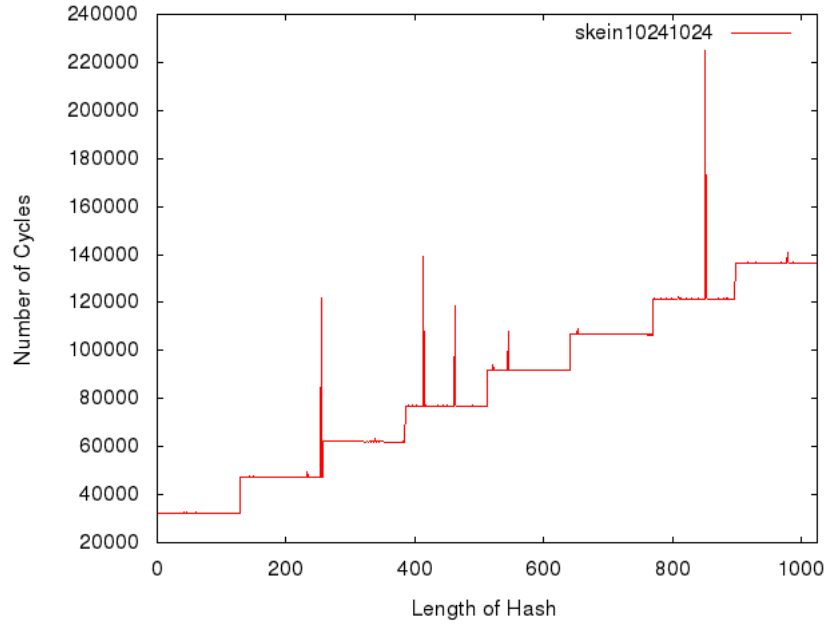
24

2.14 skein10241024

Figure 23: Skein-1024 with 1024-bit output

This instance of the Skein hash function has internal state size of 1024-bits (128 bytes) and produes a 1024-bit digest. Because the Threefish block cipher works on 128 byte blocks, the step function has a step size of 128 bytes; that is, the number of blocks that must be processed increases with every higher multiple of 128. Following the analysis for skein512512, this algorithm must also pay a performance penalty for the UBI chaining; it must XOR more words together for each block. Finally, this implementation has more rounds than other Skein hash functions (80 rounds instead of 72). Therefore, this implementation has the worst performance of any of the Skein has functions on this architecture.

Conclusions