Hickory Hash™: Implementing an Instance of an Algebraic Eraser™ Hash Function on an MSP430 Microcontroller

Iris Anshel, Derek Atkins, Dorian Goldfeld, and Paul E. Gunnells

SecureRF Corporation
100 Beard Sawmill Rd #350, Shelton, CT 06464
ianshel@securerf.com, datkins@securerf.com, dgoldfeld@securerf.com, pgunnells@securerf.com

Abstract. Recently a novel family of braid based cryptographic hash function candidates was published, claiming to be suitable for use in low resource environments. It was shown that the new hash function family performed extremely well on a range of cryptographic test suites. In this paper we instantiate an instance of the hash family, called Hickory Hash, fix a set of parameters, implement it on a Texas Instruments MSP430 16-bit microcontroller, and compare its performance characteristics to SHA2. We show that the Hickory Hash can be a viable tool for low-power, constrained devices like those associated with the Internet of Things.

Keywords: Algebraic eraser, group theoretic cryptography, braid groups, hash functions, IoT

1 Introduction

In 2005 the Algebraic Eraser™ Key Agreement Protocol (AEKAP, also called Algebraic Eraser Diffie–Hellman, AEDH) [1] introduced a novel one-way function, E-Multiplication. AEDH (and as a consequence, E-Multiplication) have been implemented on several low-power, constrained devices and has proven to run extremely quickly, with low power use and minimal computation and storage requirements.

While that original paper on AEDH used E-Multiplication only to create the key agreement protocol, the authors have since created additional cryptographic constructions all based on the same E-Multiplication one-way function. In particular, one interesting construction is the Algebraic Eraser Hash (AEHash) construction [2], a cryptographic hash based on braids, matrices, finite fields, and E-Multiplication.

All currently known attacks on AEDH ([3], [4], [5]) have been refuted ([6], [7], [12]). These refuted attacks focused on the construction of the AEDH shared secret and not on the hard problem that is the foundation of the E-Multiplication one-way function. Therefore, these refuted attacks are not relevant to the AEHash construction, since it does not use the same formulation. The successful refutation of these attacks makes the one-way function an interesting base on which to build various cryptographic constructions.

In April, 2016, the authors of AEHash published a paper where they more succinctly introduced the hash family [8] based on the E-Multiplication one-way function. In that paper they also run various test suites, including the NIST Statistical Test Suite [14], and they show that AEHash passes[1]. However, while that paper introduced the hash algorithm mathematics and analysis, it did not provide any parameters.

Previous Work

The AEKAP has been implemented in a range of constrained and low footprint devices where there is very little space for additional functionality, especially where operations like ECC or RSA cannot fit due to available code, RAM, power, or computational resources. Since the new AEHash runs on the same already implemented E-multiplication engine it can also fit on such devices, furthering their cryptographic capability.

Braid-based hash functions have been studied for over a decade. Patrick Dehornoy published a survey [11] which discusses several potential braid-based hash methods. The AEHash uses E-Multiplication as its irreversible, one-way function to map from a braid word in B_N to a finite set of bits.

[1] We show the results of the various statistical tests, including the NIST tests, in Appendix C

Our Contribution

As presented in [8] the AEHash is presented in theoretical form. Specifically, it is lacking specific hash braids and initialization data. This makes it hard to test and, more importantly, hard to analyze.

In this paper we introduce a fully-defined instance of AEHash that we call Hickory Hash$^{\mathrm{TM}}$, define initialization and hash processing data, then implement that instance on a low-powered, constrained device (a Texas Instruments MSP430FR5969 16-bit microcontroller), and compare Hickory Hash to expected behavior of SHA2 on the same platform.

2 The Braid Group, Colored Burau matrices, and the Algebraic Eraser

Let B_N denote the N-strand braid group, and let $\{b_1, b_2, \ldots, b_{N-1}\}$ denote the Artin generators. An element $\beta \in B_N$ can be viewed as an expression in the Artin generators, $\beta = b_{i_1}^{\epsilon_1}\, b_{i_2}^{\epsilon_2}\, \cdots\, b_{i_k}^{\epsilon_k}$, where $i_j \in \{1, \ldots, N-1\}$, and $\epsilon_j \in \{\pm 1\}$, and where the generators themselves satisfy the following identities: for $i = 1, \ldots, N-1$, we have

$$b_i b_{i+1} b_i = b_{i+1} b_i b_{i+1},$$

and for all i, j with $|i - j| \geq 2$, we have

$$b_i b_j = b_j b_i.$$

By associating each generator b_i with the transposition $\sigma_i = (i \;\; i+1) \in S_N$, the permutation group on N letters, each braid $\beta \in B_N$ determines a permutation in S_N:

$$\beta = b_{i_1}^{\epsilon_1}\, b_{i_2}^{\epsilon_2}\, \cdots\, b_{i_k}^{\epsilon_k} \longmapsto \sigma_\beta = \sigma_{i_1} \cdots \sigma_{i_k}.$$

Each generator b_i, and its inverse b_i^{-1}, also determines an $N \times N$ colored Burau matrix as follows. Let $\{t_1, \ldots, t_N\}$ be a set of N indeterminates. The colored Burau matrices of b_i and b_i^{-1}, denoted $CB(b_i)$ and $CB(b_i^{-1})$ respectively, are defined by

$$CB(b_i) = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & t_i & -t_i & 1 & & \\ & & & & \ddots & \\ & & & & & 1 \end{pmatrix}, \quad CB(b_i^{-1}) = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & 1 & -\frac{1}{t_{i+1}} & \frac{1}{t_{i+1}} & & \\ & & & & \ddots & \\ & & & & & 1 \end{pmatrix} \tag{1}$$

where the indicated variables appear in row i. In the case of $CB(b_1)$ the leftmost t_1 is omitted, and likewise in the case of $CB(b_1^{-1})$ the leftmost 1 is omitted Thus each braid generator $b_i^{\pm 1}$ determines a colored Burau/permutation pair $(CB(b_i^{-\pm 1}), \sigma_i)$. We now wish to define a multiplication of colored Burau pairs so that they form a group and the natural mapping from B_N to said group is a homomorphism. To accomplish this, we require the following observation. Given a Laurent polynomial

$$f(t_1, \ldots, t_N) \;\in\; \mathbb{Z}[t_1^{\pm}, t_2^{\pm}, \ldots, t_N^{\pm}],$$

a permutation in $\sigma \in S_N$ can act (on the left) by permuting the indices of the variables. We denote this action by $f \mapsto {}^{\sigma}f$:

$${}^{\sigma}f(t_1, t_2, \ldots, t_N) = f(t_{\sigma(1)}, t_{\sigma(2)}, \ldots, t_{\sigma(N)}).$$

We extend this action to $N \times N$ matrices over $\mathbb{Z}[t_1^{\pm 1}, t_2^{\pm 1}, \ldots, t_N^{\pm 1}]$ denoted, $\mathcal{M}$, by acting on each entry in the matrix, and denote the action in the same way. The general definition for multiplying two colored Burau pairs is now defined as follows from the definition of $\mathcal{M} \rtimes S_N$: given $b_i^{\pm}, b_j^{\pm 1}$, the colored Burau/permutation pair associated with the product $b_i^{\pm 1} \cdot b_j^{\pm 1}$ is

$$(CB(b_i^{\pm 1}), \sigma_i) \circ (CB(b_j^{\pm 1}),\ \sigma_j) = \Big(CB(b_i^{\pm 1}) \cdot ({}^{\sigma_i}CB(b_j^{\pm 1})),\ \sigma_i \cdot \sigma_j\Big).$$

Given any braid

$$\beta = b_{i_1}^{\epsilon_1} b_{i_2}^{\epsilon_2} \cdots b_{i_k}^{\epsilon_k},$$

the colored Burau pair $(CB(\beta), \sigma_\beta)$ is given by

$$(CB(\beta), \sigma_\beta) = \\ (CB(b_{i_1}^{\epsilon_1}) \cdot {}^{\sigma_{i_1}} CB(b_{i_2}^{\epsilon_2}) \cdot {}^{\sigma_{i_1}\sigma_{i_2}} CB(b_{i_3}^{\epsilon_3})) \ \cdots \ {}^{\sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_{k-1}}} CB(b_{i_k}^{\epsilon_k}), \ \ \sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_k}).$$

As the length of the braid β increases, the entries in the matrix $CB(\beta)$ become very high degree Laurent polynomials. In order to make use of the above colored Burau representation we require an additional component. Let q be a prime power, and let $\mathbb{F}_q$ be the finite field of q elements. A vector of t-values is a collection of non-zero field elements:

$$\{\tau_1, \tau_2, \ldots, \tau_N\} \subset \mathbb{F}_q^\times.$$

Given a vector of t-values, we can evaluate any Laurent polynomial $f(t_1, t_2, \ldots, t_N)$ to obtain an element of $\mathbb{F}_q$:

$$f(t_1, t_2, \ldots, t_N) \downarrow_{t\text{-values}} = f(\tau_1, \tau_2, \ldots, \tau_N).$$

We extend this notation to matrices over Laurent polynomials in the obvious way. Further, if $\sigma \in S_N$ and $\mathfrak{m} \in \mathcal{M}$ we define

$${}^{\sigma}\mathfrak{m} \downarrow_{t\text{-values}} = \ ({}^{\sigma}\mathfrak{m}) \downarrow_{t\text{-values}} .$$

We can now define E-multiplication which is in essence a right action of $\mathcal{M} \rtimes S_N$ on $N_q \times S_N$. Given the ordered pairs,

$$(M, \sigma), \quad (CB(\beta), \sigma_\beta),$$

where $\beta \in B_N$, $\sigma_\beta \in S_N$, and $M \in GL_N(q)$ (the group of $N \times N$ invertible matrices with entries in $\mathbb{F}_q$), and $\sigma \in S_N$, E-multiplication, (denoted by $\star$) is given by

$$(M', \sigma') = (M, \sigma) \star (CB(\beta), \sigma_\beta) \in (M', \sigma') \in N_q \times S_N.$$

We define E-multiplication inductively. When the braid $\beta = b_i^{\pm}$ is a single generator or its inverse, we put

$$(M, \sigma) \star \left(CB(b_i^{\pm}), \ \sigma_{b_i^{\pm}}\right) = \left(M \cdot {}^{\sigma}\left(CB(b_i^{\pm})\right) \downarrow_{t\text{-values}}, \ \ \sigma \cdot \sigma_{b_i^{\pm}}\right).$$

In the general case, when $\beta = b_{i_1}^{\epsilon_1} b_{i_2}^{\epsilon_2} \cdots b_{i_k}^{\epsilon_k}$, we put

$$(M, \sigma) \star (CB(\beta), \sigma_\beta) = (M, \sigma) \star (CB(b_{i_1}^{\epsilon_1}), \sigma_{b_{i_1}}) \star (CB(b_{i_2}^{\epsilon_2}), \sigma_{b_{i_2}}) \star \cdots \star (CB(b_{i_k}^{\epsilon_k}), \sigma_{b_{i_k}}), \tag{2}$$

where the right side of (2) is evaluated left-to-right. One can check that this is independent of the expression of β in the Artin generators.

The above definition lies at the core of Algebraic Eraser based protocols; for details and examples, we refer to [1]. We will require a slight modification in the definition of E-multiplication to construct the AEHash. In the definition of $\star$ above the t-values remain the same at every step of the iterative process (2). Now we wish to define a new operation, denote $\star'$, in which the t-values themselves are permuted along the way. Assuming as above that $\beta = b_{i_1}^{\epsilon_1} b_{i_2}^{\epsilon_2} \cdots b_{i_k}^{\epsilon_k}$, we define

$$T_1 = t\text{-values} = \{\tau_1, \ldots, \tau_N\},$$

and let

$$T_2 = {}^{\sigma_0 \cdot \sigma_{b_{i_1}}} T_1.$$

The second step of the E-multiplication,

$$(M, \sigma_0) \star \left(CB(\beta), \ \sigma_\beta\right) = (M, \sigma_0) \star \left(CB(b_{i_1}^{\epsilon_1}), \ \sigma_{b_{i_1}}\right) \star \left(CB(b_i^{\epsilon_2}), \ \sigma_{b_{i_2}}\right) \star \cdots \star \left(CB(b_k^{\epsilon_k}), \ \sigma_{b_{i_k}}\right),$$

is given by

$$\Big(M \cdot^{\sigma_0} (CB(b_{i_1}^{\epsilon_1})) \downarrow_{T_1},\ \sigma_0 \cdot \sigma_{b_{i_1}}\Big) \star \big(CB(b_i^{\epsilon_2}),\ \sigma_{b_{i_2}}\big). \tag{3}$$

We modify the original definition of E-multiplication by defining a new operation $\star'$ in the following way. Modify the second step of the E-multiplication in (3) by using the set T_2 for t-values to obtain,

$$\Big(M \cdot^{\sigma_0} (CB(b_{i_1}^{\epsilon_1})) \downarrow_{T_1} \ \cdot^{\sigma_0\sigma_{b_{i_1}}} (CB(b_{i_2}^{\epsilon_2})) \downarrow_{T_2},\ \ \sigma_0 \cdot \sigma_{b_{i_1}} \cdot \sigma_{b_{i_2}}\Big).$$

Iterating this process we obtain the operation $\star'$. It is this variation of E-multiplication that we will use to define our hash function. We remark that one can also define an algebraic eraser-based hash function by replacing the operation $\star'$ with the original E-multiplication operation $\star$. Unlike $\star$, the $\star'$-operation does not define an action of $\mathcal{M} \rtimes S_N$ on $N_q \times S_N$.

3 The AEHash Function

Let S denote a string of bits and let λ denote a fixed non-zero positive integer. Upon padding S sufficiently we can assume that the length of S (denoted $\mathrm{Card}(S)$) is divisible by λ, and S can be broken into a union of $D_S = \mathrm{Card}(S)/\lambda$ disjoint blocks, each of which has length λ:

$$S = \bigcup_{i=1}^{D_S} \mathrm{Block}(i).$$

By letting $v(i)$ denote the integer that the binary string $\mathrm{Block}(i)$ represents, we have, that $0 \leq v(i) \leq 2^\lambda - 1$.

The AEHash function, H_{AE} is specified by the following data:

$$\big\{B_N,\ \ q,\ \ \lambda,\ \ t\text{-values} = \{\tau_1, \ldots, \tau_N\},\ \ \{c_0, c_1, \ldots, c_{2^\lambda - 1}\} \subset B_N,\ \ (n_0, \sigma_0) \in N_q \times S_N\big\},$$

where

- B_N is the braid group on N strands;
- q is a power of a 2, the t-values are invertible elements in $\mathbb{F}_q$;
- the collection of braid group elements $\{c_0, c_1, \ldots, c_{2^\lambda - 1}\}$ is fixed and assumed to generate a free submonoid of B_N;
- $(n_0, \sigma_0) \in GL_N(q) \times S_N$ is an ordered pair.

The output of the AEHash is defined to be the sequence of bits that specify the matrix, which is evaluated through a sequence of E-multiplications $\star'$. The length of the AEHash is given by

$$N^2 \cdot \mathrm{ceil}\big(\log_2(q)\big),$$

where for $x > 0$, the function $\mathrm{ceil}(x)$ denotes the ceiling of x (i.e., the smallest integer n such that $x \leq n$).

The lengths of the elements c_i, will impact the efficiency of the hash function. In our initial testing we chose the length to be in the range of $2N$.

Each element c_i is given as a fixed expression in the Artin generators and is thus associated with a fixed sequence of colored Burau matrices/permutations pairs. We evaluate the operation $\star'$ using this explicit sequence of colored Burau pairs and, abusing the notation slightly, we denote the output by $(CB(c_i), \sigma_{c_i})$. It is important to remark that since $\star'$ is not an action, the braids c_i must be used as specified and not rewritten using the braid relations[2].

[2] Implementing $\star'$ implies iterating down each Artin generator of each hash braid element c_i, in order, and permuting the t-values at each step

The string S, having been broken into blocks of length λ, is associated with a sequence of braid words:

$$c_{v(1)},\ c_{v(2)},\ \ \ldots,\ \ c_{v(D_S)}.$$

Thus S is associated with a sequence colored Burau/permutation pairs:

$$\big(CB(c_{v(1)}),\ \sigma_{c_{(v(1))}}\big),\ \big(CB(c_{v(2)}),\ \sigma_{c_{v(2)}}\big),\ \ \ldots,\ \ \big(CB(c_{v(D_S))},\ \sigma_{c_{v(D_S)}}\big).$$

The hash of the string S, denoted $H_{\mathrm{AE}}(S)$, is defined to be the matrix part of the output of the iterative modified E-multiplication

$$(n_0, \sigma_0) \star' (CB(c_{v(1)}),\ \sigma_{c_{v(1)}}) \star' (CB(c_{v(2)}),\ \sigma_{c_{v(2)}}) \star' \cdots \star' (CB(c_{v(D_S)}),\ \sigma_{c_{v(D_S)}}).$$

4 Basic Analysis of the AEHash

The output of AEHash is, definitionally, a string of bits of length $N^2 \cdot \mathrm{ceil}\big(\log_2(q)\big)$. An upper bound for the size of the collection of all possible hashes the AEHash can generate is given by

$$q^{N^2}.$$

Recalling the assumption that the subgroup of B_N generated by $\{c_0, c_1, \ldots, c_{2^\lambda - 1}\}$, is free on the set of fixed braids $\{c_0, c_1, \ldots, c_{2^\lambda - 1}\}$. The number of possible sequences in the fixed braids of length D_S, is given by

$$2^{D_S \cdot \lambda},$$

and thus reversing the first step of the AEHash has $D_S \cdot \lambda$ - bit security. Note that $D_S \cdot \lambda$ coincides with the length of the message, so the security will only be high for long messages.

The only known method for reversing E-multiplication, to date, is a brute force procedure, resulting in $D_S \cdot \lambda$ - bit security. Furthermore, given two distinct strings S_1, S_2 whose block decomposition is given by

$$S_j = \bigcup_{i=1}^{D_{S_j}} \mathrm{Block}(j_i), \quad (j = 1, 2),$$

where $D_{S_j} = \frac{\mathrm{Card}(S_j)}{\lambda}$, the braids defined by the associated sequences of the fixed braids, say

$$\{c_{S_1, v(1)},\ c_{S_1, v(2)}),\ \ \ldots,\ \ c_{S_1, v(D_{S_1})}\},$$

$$\{c_{S_2, v(1)},\ c_{S_2, v(2)}),\ \ \ldots,\ \ c_{S_2, v(D_{S_2})}\},$$

will necessarily be distinct: in a free group two (reduced) words are equal if and only if their expressions are identical. Thus any collisions the AEHash might have cannot emerge when the string is replaced by the sequence of braid elements, and must stem from either an element in the kernel of the function that takes braids to colored Burau/permutations pairs, or colored Burau/permutation pairs to matrices with field entries/permutation pairs.

An upper bound for the running time of the AEHash can be obtained as follows. Let

$$L_C = \mathrm{Max}\big\{\mathrm{ArtinLength}(c_i) \mid i = 0, \ldots, 2^\lambda - 1\big\}.$$

Then AEHash(S) requires at most

$$L_C \cdot \frac{\mathrm{Card}(S)}{\lambda}$$

E-multiplications in order to obtain the $N \times N$ matrix. Once this matrix is in place the entries must be converted to bits. The individual braids, c_i, must have sufficiently long length so that the initial matrix the AEHash produces does not have too many zeros.

We remark that the homomorphic Hash function introduced by Zémor's [17], which is shown to be robust in [13], is somewhat structurally similar to the Hickory Hash function, in the case there are only two hash braids which are chosen to be pure. The additional complexity of non-pure braids, which ensures the one-way nature E-multiplication is in place, serves support to the Hickory hash function.

5 Generating the AEHash Parameters for Hickory Hash

The choices of the parameters N, q, and λ, and the length of the braids c_i, impact of the running time and memory requirements of the AEHash. In a constrained environment keeping $q \leq 5$ and $\lambda \leq 8$ is clearly appropriate. While it is possible to use $N \geq 8$, in order to insure sufficient mixing in the course of evaluating the AEHash, the lengths of the braid c_i would be greater that $2 \cdot N$, and hence the run time begins to become an issue were we to use larger N.

The central question is how to choose the braids used in the Hickory Hash. What emerges from our initial study has been the need for each braid c_i to impact each of the nodes $\{1, 2, \ldots, N\}$ by displacing them (at least) twice to the right. This property insures that every generator will appear in each c_i, and the length of c_i is at least $2n$. In order to prevent cancellations between the c_i's we will choose them to be positive braids. As an example, let $N = 4$, and consider the braids

$$x \;=\; b_2\, b_3\, b_2\, b_3\, b_1\, b_2\, b_1\, b_2,$$

and

$$y \;=\; b_3\, b_2\, b_2\, b_1\, b_3\, b_2\, b_1\, b_1.$$

Focussing on x, note that the node 2 initially moves twice to the right via the initial sub word $b_2\, b_3$, and then move three to the left. Following node 4, it moves three to the left until the final subword $b_1\, b_2$ moves it twice to the right. The braid y is itself the product of two words each of which move each node once to the right, and hence y moves each node twice to the right.

Braids constructed with this property will almost always have nontrivial permutations, and said permutations will impact the entries of the colored Burau matrix of other braids of this nature. The number of braids that move each node twice at a time (to the right) grows very rapidly. While it is outside the scope of this paper, given $N \geq 4$ it can be shown that the number of possible such braids is,

$$\frac{1}{4}\left(-2 - \left(1-\sqrt{2}\right)^N\,(1+\sqrt{2}) + (-1+\sqrt{2})\left(1+\sqrt{2}\right)^N\right).$$

The parameters we chose can be found in Appendix A, and include explicit choices of the braids.

6 Implementing Hickory Hash on an MSP430

The Texas Instruments MSP430 is a class of 16-bit microcontrollers frequently used for low-power embedded devices and numerous other Internet of Things endpoints. Different processors within the family provide different amounts of flash and RAM, different onboard peripherals, and varying clock speeds.

For our implementation we used C, and compiled the code to the MSP using TI's GCC compiler with the -O3 compiler optimization option. For our test cases we decided not to pack the data as compactly as possible, and we hard-coded the parameters. The finite field math was performed by lookup tables (because we're using F16, which results in a 256-byte multiplication table). These tables were included in the code size, as were the braids, initial matrix, permutation, and t-values.

In a straightforward implementation, each E-Multiplication requires two branches, $2N$ or $3N$ finite-field multiplications, and N or $2N$ finite-field additions[3]. Based on either set of selected parameters, each block of λ input bits requires 16 E-Multiplications. This results in a total of $16 * \lceil \text{InputLength}/\lambda \rceil$ E-Multiplications.

During development we decided to elide several possible speed optimizations, including loop unrolling or writing the core functions in assembly language. We also decided to permute all the t-values each round instead of only the required entry for that particular braid generator. We are confident based on exploration of E-Multiplication in other avenues that we can achieve an addtional 2-3x speed improvement in C.

The implementation pseudocode can be found in Appendix B.

[3] We know that a hardware implementation of E-Multiplication can perform one E-Multiplication per clock cycle

7 Hickory Hash Performance

Using the implementation on the MSP, we ran several tests and measured the computation time to hash 64-byte messages. We tested both the $\lambda = 5$ and $\lambda = 7$ parameters on the MSP using a clock speed of 8MHz. As expected, the latter parameter set ran faster but used more ROM.

When we used the parameters for $\lambda = 5$, the code used a total of 2012 bytes of ROM (848 rodata, 1164 text) and 110 bytes of RAM. The runtime for a 64 byte message was 61ms, which is 1049 bytes per second.

When we set the program to use the paramters for $\lambda = 7$ the code space increased to 3546 bytes of ROM (2384 rodata, 1162 text), and RAM usage increased marginally to 112 bytes. However the runtime dropped to 44ms, yielding 1454 bytes per second.

As noted before, we are confident we could achieve an additional 2-3x speed improvement using techniques we've learned from different Algebraic Eraser implementations on this platform. This would increase our speeds to 2098-3147 bytes per second for $\lambda = 5$ and 2908-4362 bytes per second for $\lambda = 7$. Conversion to ASM should yield an additional speedup, and other additional speedups are possible.

Let's compare Hickory Hash to SHA2-256. According to Christian Wenzel-Benner *et al*, SHA2-256 on the MSP430 requires an area (4 * RAM bytes + ROM bytes) of approximately 4096 [16]. This is closer to the space required by the $\lambda = 7$ parameter set versus $\lambda = 5$.

8 Conclusions

The AEHash construction defines a family of hash algorithms based on the Algebraic Eraser E-Multiplication one-way function. We have instantiated two specific sets of parameters, one using $\lambda = 5$ and another using $\lambda = 7$, both providing a 256-bit hash, which we call Hickory Hash. Then we implemented the Hickory Hash with those parameters on a Texas Instruments MSP430 16-bit microcontroller.

Comparing Hickory Hash to SHA2, the size of Hickory Hash is smaller than SHA2. In particular the $\lambda = 5$ parameter set was 30% smaller than SHA2, and the $\lambda = 7$ parameter set was just slightly smaller than SHA2.

We believe there are several optimizations that can speed up the Hickory Hash implementation, including major changes like an implementation in assembler and minor changes like directly computing the permuted t-values.

Moreover, we believe a hardware implementation of Hickory Hash would be interesting when combined with other AE-based E-Multiplication constructions. This is because the underlying E-Multiplication engine can be re-used between the different constructions, reducing the amount of silicon and/or program ROM required. Previous experience with hardware implementations has shown a single clock cycle per E-Multiplication, which would result in a processing speed of λ bits per 16 cycles. A size/speed comparison with SHA2 in hardware would be interesting.

References

1. Anshel, Iris; Anshel, Michael; Goldfeld, Dorian; and Lemieux, Stephane, *Key agreement, the Algebraic Eraser^{TM}, and Lightweight Cryptography,* Algebraic methods in cryptography, Contemp. Math., vol. 418, Amer. Math. Soc., Providence, RI, 2006, pp. 1–34.
2. Anshel, Iris; Goldfeld, Dorian, *Cryptographic hash function,* US Patent number 8,972,715, March 3, 2015.
3. Blackburn, Simon R.; Robshaw, M.J.B.; *On the Security of the Algebraic Eraser Tag Authentication Protocol,* http://eprint.iacr.org/2016/091 (2016).
4. A. Ben-Zvi, S. Blackburn, and B. Tsaban, *A Practical Cryptanalysis of the Algebraic Eraser,* November, 2015.
5. A. Kalka, M. Teicher, and B. Tsaban, *Short expressions of permutations as products and cryptanalysis of the Algebraic Eraser*, Adv. in Appl. Math. **49** (2012), no. 1, 57–76.
6. Atkins, Derek; Goldfeld, Dorian, *Addressing the Algebraic Eraser Diffie–Hellman Over-the-Air Protocol,* http://eprint.iacr.org/2016/205 (2016).

7. Anshel, Iris; Atkins, Derek; Goldfeld, Dorian; Gunnells, Paul E., *Defeating the Ben-Zvi, Blackburn, and Tsaban Attack on the Algebraic Eraser,* http://arxiv.org/abs/1601.04780 (2016).

8. Anshel, Iris; Atkins, Derek; Goldfeld, Dorian; Gunnells, Paul E., *A Class of Hash Functions Based on the Algebraic Eraser*TM*,* preprint, Groups Complexity Cryptology, ISSN 1869-6104, April, 2016.

9. Birman, Joan; Ko, Ki Hyoung; Lee, Sang Jin; *A new approach to the word and conjugacy problems in the braid groups,* Adv. Math. 139 (1998), no. 2, 322–353.

10. Dehornoy, Patrick; *A fast method for comparing braids,* Adv. Math. 125 (1997), no. 2, 200–235.

11. Dehornoy, Patrick; *Braid-based cryptography,* http://www.math.unicaen.fr/∼dehornoy/Surveys/Dgw.pdf

12. Goldfeld, Dorian; Gunnells, Paul E. *Defeating the Kalka–Teicher–Tsaban linear algebra attack on the Algebraic Eraser,* http://arxiv.org/abs/1202.0598 (2012).

13. Ciaran Mulland, Boaz Tsaban; *SL2 homomorphic hash functions: Worst case to average case reduction and short collision search,* arXiv:1306.5646v3 [cs.CR] (2015).

14. National Institute of Standards and Technology; *NIST Statistical Test Suite,* available from `http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html`

15. G. Seroussi, Table of low-weight binary irreducible polynomials, Technical Report HP-98-135, Computer Systems Laboratory, Hewlett–Packard, 1998.

16. Wenzel-Benner, Christian; Gräf, Jens; Kaps, Jens-Peter; Pham, John, *XBX Benchmarking Results January 2012,* NIST SHA-3 Conference, March, 2012, available from `http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/March2012/documents/presentations/WENZEL_BENNER_presentation.pdf`

17. G. Zémor; *Hash functions and graphs with large girths,* Eurocrypt '91, Lecture Notes in Computer Science 547 (1991), 508–511.

Appendix A Hickory Hash Parameters

The following two sections detail two instantiations of the AEHash family that we call Hickory Hash. In the first instance λ is 5; in the second instance λ is 7. In both cases we selected a 256-bit hash result. The testing results show a time/space tradeoff. The $\lambda = 7$ case requires more space (to hold the 128 braids) however it requires fewer blocks to encode the message so requires fewer E-Multiplications, resulting in a faster execution. In all cases represent the finite field $\mathbb{F}_{16}$ as $\mathbb{F}_2[x]/(f)$, where f is the irreducible polynomial x^4+x+1 (cf. [15]). Elements of $\mathbb{F}_{16}$ are then represented as 4-bit numbers: the finite field element $a_3x^3 + a_2x^2 + \cdots + a_0 \bmod f$ is converted to the bitstring $a_3a_2 \cdots a_0$ (note that the coefficients of high degree monomials become the high-order bits in the bitstring).

A.1 $\lambda = 5$

- $N = 8$
- $q = 16$
- t-values: (4 15 8 4 8 4 7 12)
- Initial Permutation: (5 3 4 7 2 1 0 6)
- Initial Matrix:

$$\begin{pmatrix} 14 & 14 & 1 & 4 & 7 & 5 & 10 & 2 \\ 12 & 6 & 7 & 9 & 4 & 5 & 9 & 10 \\ 14 & 9 & 7 & 10 & 0 & 0 & 13 & 0 \\ 15 & 5 & 7 & 8 & 15 & 2 & 5 & 8 \\ 13 & 2 & 14 & 5 & 8 & 10 & 2 & 8 \\ 0 & 9 & 7 & 9 & 2 & 6 & 9 & 15 \\ 12 & 14 & 12 & 8 & 0 & 5 & 7 & 0 \\ 11 & 13 & 5 & 7 & 3 & 9 & 7 & 10 \end{pmatrix}$$

- Hash Braids:

0. $b_6b_7b_5b_6b_4b_5b_3b_4b_2b_3b_1b_2b_1b_2b_1b_2$
1. $b_6b_7b_5b_6b_4b_5b_3b_4b_2b_3b_2b_3b_2b_3b_1b_2$
2. $b_6b_7b_5b_6b_4b_5b_3b_4b_3b_4b_3b_4b_2b_3b_1b_2$
3. $b_6b_7b_5b_6b_4b_5b_4b_5b_4b_5b_3b_4b_2b_3b_1b_2$
4. $b_6b_7b_5b_6b_5b_6b_5b_6b_4b_5b_3b_4b_2b_3b_1b_2$
5. $b_6b_7b_6b_7b_6b_7b_5b_6b_4b_5b_3b_4b_2b_3b_1b_2$
6. $b_6b_7b_6b_7b_5b_6b_5b_6b_4b_5b_3b_4b_2b_3b_1b_2$
7. $b_6b_7b_6b_7b_5b_6b_4b_5b_4b_5b_3b_4b_2b_3b_1b_2$
8. $b_6b_7b_6b_7b_5b_6b_4b_5b_3b_4b_3b_4b_2b_3b_1b_2$
9. $b_6b_7b_6b_7b_5b_6b_4b_5b_3b_4b_2b_3b_2b_3b_1b_2$
10. $b_6b_7b_6b_7b_5b_6b_4b_5b_3b_4b_2b_3b_1b_2b_1b_2$
11. $b_6b_7b_5b_6b_5b_6b_4b_5b_4b_5b_3b_4b_2b_3b_1b_2$
12. $b_6b_7b_5b_6b_5b_6b_4b_5b_3b_4b_3b_4b_2b_3b_1b_2$
13. $b_6b_7b_5b_6b_5b_6b_4b_5b_3b_4b_2b_3b_2b_3b_1b_2$
14. $b_6b_7b_5b_6b_5b_6b_4b_5b_3b_4b_2b_3b_1b_2b_1b_2$
15. $b_6b_7b_5b_6b_4b_5b_4b_5b_3b_4b_3b_4b_2b_3b_1b_2$
16. $b_6b_7b_5b_6b_4b_5b_4b_5b_3b_4b_2b_3b_2b_3b_1b_2$
17. $b_6b_7b_5b_6b_4b_5b_4b_5b_3b_4b_2b_3b_1b_2b_1b_2$
18. $b_6b_7b_5b_6b_4b_5b_3b_4b_3b_4b_2b_3b_2b_3b_1b_2$
19. $b_6b_7b_5b_6b_4b_5b_3b_4b_3b_4b_2b_3b_1b_2b_1b_2$
20. $b_6b_7b_5b_6b_4b_5b_3b_4b_2b_3b_2b_3b_1b_2b_1b_2$
21. $b_6b_7b_6b_7b_6b_7b_4b_5b_4b_5b_3b_4b_2b_3b_1b_2$
22. $b_6b_7b_6b_7b_6b_7b_4b_5b_3b_4b_3b_4b_2b_3b_1b_2$
23. $b_6b_7b_6b_7b_6b_7b_4b_5b_3b_4b_2b_3b_2b_3b_1b_2$
24. $b_6b_7b_6b_7b_6b_7b_4b_5b_3b_4b_2b_3b_1b_2b_1b_2$
25. $b_6b_7b_5b_6b_5b_6b_5b_6b_3b_4b_3b_4b_2b_3b_1b_2$
26. $b_6b_7b_5b_6b_5b_6b_5b_6b_3b_4b_2b_3b_2b_3b_1b_2$
27. $b_6b_7b_5b_6b_5b_6b_5b_6b_3b_4b_2b_3b_1b_2b_1b_2$
28. $b_6b_7b_5b_6b_4b_5b_4b_5b_4b_5b_2b_3b_2b_3b_1b_2$
29. $b_6b_7b_5b_6b_4b_5b_4b_5b_4b_5b_2b_3b_1b_2b_1b_2$
30. $b_6b_7b_5b_6b_4b_5b_2b_3b_2b_3b_2b_3b_1b_2b_1b_2$
31. $b_6b_7b_6b_7b_5b_6b_3b_4b_3b_4b_2b_3b_1b_2b_1b_2$

- Test Case:
  - Input: "This is a test" (Hex: `54 68 69 73 20 69 73 20 61 20 74 65 73 74`)
  - Result: 256-bit packed result, in Hex:
    `2f 5b 19 73 9d c1 74 b6  50 ab 8d 0d aa 1f 36 28`
    `2d 20 69 44 b6 c7 88 04  45 57 c9 e2 c7 64 4b 78`

A.2 $\lambda = 7$

- $N = 8$
- $q = 16$
- t-values: (4 15 8 4 8 4 7 12)
- Initial Permutation: (5 3 4 7 2 1 0 6)
- Initial Matrix:

$$\begin{pmatrix} 14 & 14 & 1 & 4 & 7 & 5 & 10 & 2 \\ 12 & 6 & 7 & 9 & 4 & 5 & 9 & 10 \\ 14 & 9 & 7 & 10 & 0 & 0 & 13 & 0 \\ 15 & 5 & 7 & 8 & 15 & 2 & 5 & 8 \\ 13 & 2 & 14 & 5 & 8 & 10 & 2 & 8 \\ 0 & 9 & 7 & 9 & 2 & 6 & 9 & 15 \\ 12 & 14 & 12 & 8 & 0 & 5 & 7 & 0 \\ 11 & 13 & 5 & 7 & 3 & 9 & 7 & 10 \end{pmatrix}$$

- Hash Braids:

0. $b_6b_7b_6b_7b_6b_7b_5b_6b_4b_5b_3b_4b_2b_3b_1b_2$
1. $b_6b_7b_6b_7b_5b_6b_5b_6b_4b_5b_3b_4b_2b_3b_1b_2$
2. $b_6b_7b_5b_6b_5b_6b_5b_6b_4b_5b_3b_4b_2b_3b_1b_2$
3. $b_6b_7b_6b_7b_6b_7b_4b_5b_4b_5b_3b_4b_2b_3b_1b_2$
4. $b_6b_7b_6b_7b_5b_6b_4b_5b_4b_5b_3b_4b_2b_3b_1b_2$
5. $b_6b_7b_5b_6b_5b_6b_4b_5b_4b_5b_3b_4b_2b_3b_1b_2$
6. $b_6b_7b_6b_7b_4b_5b_4b_5b_4b_5b_3b_4b_2b_3b_1b_2$
7. $b_6b_7b_5b_6b_4b_5b_4b_5b_4b_5b_3b_4b_2b_3b_1b_2$
8. $b_6b_7b_6b_7b_6b_7b_5b_6b_3b_4b_3b_4b_2b_3b_1b_2$
9. $b_6b_7b_6b_7b_5b_6b_5b_6b_3b_4b_3b_4b_2b_3b_1b_2$
10. $b_6b_7b_5b_6b_5b_6b_5b_6b_3b_4b_3b_4b_2b_3b_1b_2$
11. $b_6b_7b_6b_7b_6b_7b_4b_5b_3b_4b_3b_4b_2b_3b_1b_2$
12. $b_6b_7b_6b_7b_5b_6b_4b_5b_3b_4b_3b_4b_2b_3b_1b_2$
13. $b_6b_7b_5b_6b_5b_6b_4b_5b_3b_4b_3b_4b_2b_3b_1b_2$
14. $b_6b_7b_6b_7b_4b_5b_4b_5b_3b_4b_3b_4b_2b_3b_1b_2$
15. $b_6b_7b_5b_6b_4b_5b_4b_5b_3b_4b_3b_4b_2b_3b_1b_2$
16. $b_6b_7b_6b_7b_5b_6b_3b_4b_3b_4b_3b_4b_2b_3b_1b_2$
17. $b_6b_7b_5b_6b_5b_6b_3b_4b_3b_4b_3b_4b_2b_3b_1b_2$
18. $b_6b_7b_6b_7b_4b_5b_3b_4b_3b_4b_3b_4b_2b_3b_1b_2$
19. $b_6b_7b_5b_6b_4b_5b_3b_4b_3b_4b_3b_4b_2b_3b_1b_2$
20. $b_6b_7b_6b_7b_6b_7b_5b_6b_4b_5b_2b_3b_2b_3b_1b_2$
21. $b_6b_7b_6b_7b_5b_6b_5b_6b_4b_5b_2b_3b_2b_3b_1b_2$
22. $b_6b_7b_5b_6b_5b_6b_5b_6b_4b_5b_2b_3b_2b_3b_1b_2$
23. $b_6b_7b_6b_7b_6b_7b_4b_5b_4b_5b_2b_3b_2b_3b_1b_2$
24. $b_6b_7b_6b_7b_5b_6b_4b_5b_4b_5b_2b_3b_2b_3b_1b_2$
25. $b_6b_7b_5b_6b_5b_6b_4b_5b_4b_5b_2b_3b_2b_3b_1b_2$
26. $b_6b_7b_6b_7b_4b_5b_4b_5b_4b_5b_2b_3b_2b_3b_1b_2$
27. $b_6b_7b_5b_6b_4b_5b_4b_5b_4b_5b_2b_3b_2b_3b_1b_2$
28. $b_6b_7b_6b_7b_6b_7b_5b_6b_3b_4b_2b_3b_2b_3b_1b_2$
29. $b_6b_7b_6b_7b_5b_6b_5b_6b_3b_4b_2b_3b_2b_3b_1b_2$
30. $b_6b_7b_5b_6b_5b_6b_5b_6b_3b_4b_2b_3b_2b_3b_1b_2$
31. $b_6b_7b_6b_7b_6b_7b_4b_5b_3b_4b_2b_3b_2b_3b_1b_2$
32. $b_6b_7b_6b_7b_5b_6b_4b_5b_3b_4b_2b_3b_2b_3b_1b_2$
33. $b_6b_7b_5b_6b_5b_6b_4b_5b_3b_4b_2b_3b_2b_3b_1b_2$
34. $b_6b_7b_6b_7b_4b_5b_4b_5b_3b_4b_2b_3b_2b_3b_1b_2$
35. $b_6b_7b_5b_6b_4b_5b_4b_5b_3b_4b_2b_3b_2b_3b_1b_2$
36. $b_6b_7b_6b_7b_5b_6b_3b_4b_3b_4b_2b_3b_2b_3b_1b_2$
37. $b_6b_7b_5b_6b_5b_6b_3b_4b_3b_4b_2b_3b_2b_3b_1b_2$
38. $b_6b_7b_6b_7b_4b_5b_3b_4b_3b_4b_2b_3b_2b_3b_1b_2$
39. $b_6b_7b_5b_6b_4b_5b_3b_4b_3b_4b_2b_3b_2b_3b_1b_2$
40. $b_6b_7b_6b_7b_6b_7b_4b_5b_2b_3b_2b_3b_2b_3b_1b_2$
41. $b_6b_7b_6b_7b_5b_6b_4b_5b_2b_3b_2b_3b_2b_3b_1b_2$
42. $b_6b_7b_5b_6b_5b_6b_4b_5b_2b_3b_2b_3b_2b_3b_1b_2$
43. $b_6b_7b_6b_7b_4b_5b_4b_5b_2b_3b_2b_3b_2b_3b_1b_2$
44. $b_6b_7b_5b_6b_4b_5b_4b_5b_2b_3b_2b_3b_2b_3b_1b_2$
45. $b_6b_7b_6b_7b_5b_6b_3b_4b_2b_3b_2b_3b_2b_3b_1b_2$
46. $b_6b_7b_5b_6b_5b_6b_3b_4b_2b_3b_2b_3b_2b_3b_1b_2$
47. $b_6b_7b_6b_7b_4b_5b_3b_4b_2b_3b_2b_3b_2b_3b_1b_2$
48. $b_6b_7b_5b_6b_4b_5b_3b_4b_2b_3b_2b_3b_2b_3b_1b_2$
49. $b_6b_7b_6b_7b_6b_7b_5b_6b_4b_5b_3b_4b_1b_2b_1b_2$
50. $b_6b_7b_6b_7b_5b_6b_5b_6b_4b_5b_3b_4b_1b_2b_1b_2$
51. $b_6b_7b_5b_6b_5b_6b_5b_6b_4b_5b_3b_4b_1b_2b_1b_2$
52. $b_6b_7b_6b_7b_6b_7b_4b_5b_4b_5b_3b_4b_1b_2b_1b_2$
53. $b_6b_7b_6b_7b_5b_6b_4b_5b_4b_5b_3b_4b_1b_2b_1b_2$
54. $b_6b_7b_5b_6b_5b_6b_4b_5b_4b_5b_3b_4b_1b_2b_1b_2$
55. $b_6b_7b_6b_7b_4b_5b_4b_5b_4b_5b_3b_4b_1b_2b_1b_2$
56. $b_6b_7b_5b_6b_4b_5b_4b_5b_4b_5b_3b_4b_1b_2b_1b_2$
57. $b_6b_7b_6b_7b_6b_7b_5b_6b_3b_4b_3b_4b_1b_2b_1b_2$
58. $b_6b_7b_6b_7b_5b_6b_5b_6b_3b_4b_3b_4b_1b_2b_1b_2$
59. $b_6b_7b_5b_6b_5b_6b_5b_6b_3b_4b_3b_4b_1b_2b_1b_2$
60. $b_6b_7b_6b_7b_6b_7b_4b_5b_3b_4b_3b_4b_1b_2b_1b_2$
61. $b_6b_7b_6b_7b_5b_6b_4b_5b_3b_4b_3b_4b_1b_2b_1b_2$
62. $b_6b_7b_5b_6b_5b_6b_4b_5b_3b_4b_3b_4b_1b_2b_1b_2$
63. $b_6b_7b_6b_7b_4b_5b_4b_5b_3b_4b_3b_4b_1b_2b_1b_2$
64. $b_6b_7b_5b_6b_4b_5b_4b_5b_3b_4b_3b_4b_1b_2b_1b_2$
65. $b_6b_7b_6b_7b_5b_6b_3b_4b_3b_4b_3b_4b_1b_2b_1b_2$
66. $b_6b_7b_5b_6b_5b_6b_3b_4b_3b_4b_3b_4b_1b_2b_1b_2$
67. $b_6b_7b_6b_7b_4b_5b_3b_4b_3b_4b_3b_4b_1b_2b_1b_2$
68. $b_6b_7b_5b_6b_4b_5b_3b_4b_3b_4b_3b_4b_1b_2b_1b_2$
69. $b_6b_7b_6b_7b_6b_7b_5b_6b_4b_5b_2b_3b_1b_2b_1b_2$
70. $b_6b_7b_6b_7b_5b_6b_5b_6b_4b_5b_2b_3b_1b_2b_1b_2$
71. $b_6b_7b_5b_6b_5b_6b_5b_6b_4b_5b_2b_3b_1b_2b_1b_2$

72. $b_6b_7b_6b_7b_6b_7b_4b_5b_4b_5b_2b_3b_1b_2b_1b_2$
73. $b_6b_7b_6b_7b_5b_6b_4b_5b_4b_5b_2b_3b_1b_2b_1b_2$
74. $b_6b_7b_5b_6b_5b_6b_4b_5b_4b_5b_2b_3b_1b_2b_1b_2$
75. $b_6b_7b_6b_7b_4b_5b_4b_5b_4b_5b_2b_3b_1b_2b_1b_2$
76. $b_6b_7b_5b_6b_4b_5b_4b_5b_4b_5b_2b_3b_1b_2b_1b_2$
77. $b_6b_7b_6b_7b_6b_7b_5b_6b_3b_4b_2b_3b_1b_2b_1b_2$
78. $b_6b_7b_6b_7b_5b_6b_5b_6b_3b_4b_2b_3b_1b_2b_1b_2$
79. $b_6b_7b_5b_6b_5b_6b_5b_6b_3b_4b_2b_3b_1b_2b_1b_2$
80. $b_6b_7b_6b_7b_6b_7b_4b_5b_3b_4b_2b_3b_1b_2b_1b_2$
81. $b_6b_7b_6b_7b_5b_6b_4b_5b_3b_4b_2b_3b_1b_2b_1b_2$
82. $b_6b_7b_5b_6b_5b_6b_4b_5b_3b_4b_2b_3b_1b_2b_1b_2$
83. $b_6b_7b_6b_7b_4b_5b_4b_5b_3b_4b_2b_3b_1b_2b_1b_2$
84. $b_6b_7b_5b_6b_4b_6b_4b_5b_3b_4b_2b_3b_1b_2b_1b_2$
85. $b_6b_7b_6b_7b_5b_6b_3b_4b_3b_4b_2b_3b_1b_2b_1b_2$
86. $b_6b_7b_5b_6b_5b_6b_3b_4b_3b_4b_2b_3b_1b_2b_1b_2$
87. $b_6b_7b_6b_7b_4b_5b_3b_4b_3b_4b_2b_3b_1b_2b_1b_2$
88. $b_6b_7b_5b_6b_4b_5b_3b_4b_3b_4b_2b_3b_1b_2b_1b_2$
89. $b_6b_7b_6b_7b_6b_7b_4b_5b_2b_3b_2b_3b_1b_2b_1b_2$
90. $b_6b_7b_6b_7b_5b_6b_4b_5b_2b_3b_2b_3b_1b_2b_1b_2$
91. $b_6b_7b_5b_6b_5b_6b_4b_5b_2b_3b_2b_3b_1b_2b_1b_2$
92. $b_6b_7b_6b_7b_4b_5b_4b_5b_2b_3b_2b_3b_1b_2b_1b_2$
93. $b_6b_7b_5b_6b_4b_5b_4b_5b_2b_3b_2b_3b_1b_2b_1b_2$
94. $b_6b_7b_6b_7b_5b_6b_3b_4b_2b_3b_2b_3b_1b_2b_1b_2$
95. $b_6b_7b_5b_6b_5b_6b_3b_4b_2b_3b_2b_3b_1b_2b_1b_2$
96. $b_6b_7b_6b_7b_4b_5b_3b_4b_2b_3b_2b_3b_1b_2b_1b_2$
97. $b_6b_7b_5b_6b_4b_5b_3b_4b_2b_3b_2b_3b_1b_2b_1b_2$
98. $b_6b_7b_6b_7b_6b_7b_5b_6b_3b_4b_1b_2b_1b_2b_1b_2$
99. $b_6b_7b_6b_7b_5b_6b_5b_6b_3b_4b_1b_2b_1b_2b_1b_2$
100. $b_6b_7b_5b_6b_5b_6b_5b_6b_3b_4b_1b_2b_1b_2b_1b_2$
101. $b_6b_7b_6b_7b_6b_7b_4b_5b_3b_4b_1b_2b_1b_2b_1b_2$
102. $b_6b_7b_6b_7b_5b_6b_4b_5b_3b_4b_1b_2b_1b_2b_1b_2$
103. $b_6b_7b_5b_6b_5b_6b_4b_5b_3b_4b_1b_2b_1b_2b_1b_2$
104. $b_6b_7b_6b_7b_4b_5b_4b_5b_3b_4b_1b_2b_1b_2b_1b_2$
105. $b_6b_7b_5b_6b_4b_5b_4b_5b_3b_4b_1b_2b_1b_2b_1b_2$
106. $b_6b_7b_6b_7b_5b_6b_3b_4b_3b_4b_1b_2b_1b_2b_1b_2$
107. $b_6b_7b_5b_6b_5b_6b_3b_4b_3b_4b_1b_2b_1b_2b_1b_2$
108. $b_6b_7b_6b_7b_4b_5b_3b_4b_3b_4b_1b_2b_1b_2b_1b_2$
109. $b_6b_7b_5b_6b_4b_5b_3b_4b_3b_4b_1b_2b_1b_2b_1b_2$
110. $b_6b_7b_6b_7b_6b_7b_4b_5b_2b_3b_1b_2b_1b_2b_1b_2$
111. $b_6b_7b_6b_7b_5b_6b_4b_5b_2b_3b_1b_2b_1b_2b_1b_2$
112. $b_6b_7b_5b_6b_5b_6b_4b_5b_2b_3b_1b_2b_1b_2b_1b_2$
113. $b_6b_7b_6b_7b_4b_5b_4b_5b_2b_3b_1b_2b_1b_2b_1b_2$
114. $b_6b_7b_5b_6b_4b_5b_4b_5b_2b_3b_1b_2b_1b_2b_1b_2$
115. $b_6b_7b_6b_7b_5b_6b_3b_4b_2b_3b_1b_2b_1b_2b_1b_2$
116. $b_6b_7b_5b_6b_5b_6b_3b_4b_2b_3b_1b_2b_1b_2b_1b_2$
117. $b_6b_7b_6b_7b_4b_5b_3b_4b_2b_3b_1b_2b_1b_2b_1b_2$
118. $b_6b_7b_5b_6b_4b_5b_3b_2b_2b_3b_1b_2b_1b_2b_1b_2$
119. $b_7b_7b_6b_5b_4b_3b_2b_1b_7b_6b_6b_5b_4b_3b_2b_1$
120. $b_7b_7b_6b_5b_4b_3b_2b_1b_7b_6b_5b_5b_4b_3b_2b_1$
121. $b_7b_7b_6b_5b_4b_3b_2b_1b_7b_6b_5b_4b_4b_3b_2b_1$
122. $b_7b_7b_6b_5b_4b_3b_2b_1b_7b_6b_5b_4b_3b_3b_2b_1$
123. $b_7b_7b_6b_5b_4b_3b_2b_1b_7b_6b_5b_4b_3b_2b_2b_1$
124. $b_7b_7b_6b_5b_4b_3b_2b_1b_7b_6b_5b_4b_3b_2b_1b_1$
125. $b_7b_6b_6b_5b_4b_3b_2b_1b_7b_6b_5b_5b_4b_3b_2b_1$
126. $b_7b_6b_5b_5b_4b_3b_2b_1b_7b_6b_5b_4b_4b_3b_2b_1$
127. $b_7b_6b_5b_4b_3b_3b_2b_1b_7b_6b_5b_4b_3b_2b_2b_1$

Appendix B Hickory Hash Pseudocode Implementation

The following is a C pseudocode implementation of AEHash. We call it pseudocode because it is not a fully compilable as-is; several obvious functions are elided for clarity, and the full set of parameters are elided for space considerations. The full parameter set can be found in Appendix A, and the elided functions are left as an exercise for the reader.

```
// These data are from the hash parameters.
// Some data is elided here for brevity
#define INDEX 8
static const int lambda = 5;
static const unsigned int tvalues[] = {4, 15, 8, 4, 8, 4, 7, 12};
static const unsigned int perm[] = {5, 3, 4, 7, 2, 1, 0, 6};
static const unsigned int matrix[INDEX][INDEX] = {
    { 14, 14, 1, 4, 7, 5, 10, 2 }, ... };
static const AeBraidlist braids[] = {
    { 16, { 5 6 4 5 3 4 2 3 1 2 0 1 0 1 0 1 } }, ... };

void ae_hash_emultiply_braid(AeHashCtx *ctx, const AeBraidlist braid)
{
    int i, nrow;
    unsigned int t, ti;
```

```
    unsigned int tempTvalues[INDEX];

    // Iterate down the braid generators
    for (i = 0; i < braid->len; i++) {
        unsigned int gen = braid->generator[i];

        // permute the t-values
        for (nrow=0; nrow < INDEX; nrow++)
            tempTvalues[nrow] = tvalues[ctx->perm[nrow];

        // Determine the t-value (and inverse) for this braid generator
        t = tempTvalues[ctx->perm[gen]];
        ti = FF_NEG(t);

        // Multiply in the CB Matrix for this generator
        for (nrow=0; nrow < INDEX; nrow++) {
            if (gen >= 1) {
                ctx->matrix[nrow][gen-1] =
                    FF_ADD(ctx->matrix[nrow][gen-1],
                        FF_MULT(ctx->matrix[nrow][gen], t));
            }
            if (gen < INDEX-1) {
                ctx->matrix[nrow][gen+1] =
                    FF_ADD(ctx->matrix[nrow][gen-1],
                        FF_MULT(ctx->matrix[nrow][gen], 1));
            }
            ctx->matrix[nrow][gen] =
                FF_MULT(ctx->matrix[nrow][gen], ti);
        }

        // Update the permutation
        t = ctx->perm[gen];
        ctx->perm[gen] = ctx->perm[gen+1];
        ctx->perm[gen+1] = t;
    }
}

void ae_hash(const unsigned char* data, size_t data_len,
             unsigned char* hash size_t *hashlen)
{
    AeHashCtx ctx;
    AePackCtx pack;
    unsigned int c;
    size_t i, j, len;

    // Initialize context with initial Matrix, Permutation, and t-values
    ae_hash_init_ctx(&ctx);

    // Initialize the packing context to pull out lamdba bits at a time
    ae_pack_init_ctx(&pack, data, data_len, lambda);
```

```
    // Compute the number of lambda blocks in data_len
    len = (data_len*8 + lambda - 1) / lambda;

    // Iterate over the input, E-multiply in the braid for
    // each lambda-length block of the input data
    for (i = 0; i < len; i++) {
        c = ae_pack_get_value(&pack);
        ae_hash_emultiply_braid(&ctx, braids[c]);
    }

    // output the hash by building a string of the matrix
    // re-initialize the packing context to build the result
    // (note that we're using F16, so there are 4 bits per entry)
    ae_pack_init_ctx(&pack, hash, *hashlen, 4);

    // Now iterate over the matrix and pack the result.
    for (i = 0; i < INDEX; i++) {
        for (j = 0; j < INDEX; j++) {
            ae_pack_add_value(&pack, ctx.matrix[i][j]);
        }
    }

    // And we're done
}
```

Appendix C Statistical Testing of the AE Hash Function

This section describes the various statistical tests performed on the AE Hash Function.

Bit Flip Test

In this test one first chooses a random message of length from 1 to 256 bytes. One computes the hash of the original message and then iterate through the full message, flipping one bit at a time. After each bitflip one computes another hash (and then reverts the bitflip before continuing on). Then one compares the Hamming distance between each "flipped" hash digest to the original hash.

Using the parameters in this paper we obtained the following test results.

Case 1: $\lambda = 5$ bit-flip test (2088408 checks):

min: 47
max: 166
mean: 127.87
median: 128
stddev: 8.24

Case 2: $\lambda = 7$ bit-flip test (2056088 checks):

min: 50
max: 165
mean: 127.91
median: 128
stddev: 8.14

For $N = 8$ and $q = 16 = 2^4$, the AEHash length is $N^2 \log_2(q) = (8^2) \cdot 4 = 256$. Hence, for a good hash function, the median of the Hamming distance (of the hash before and after bit flips) should be $256/2 = 128$.

Collision Test

This test is looking for collisions at specified offsets in the hash. In particular this test chooses the offset(s) into the hash output to check by iterating through every possible offset or offset-combination depending on the test configuration. Once the offset values are fixed it loops for a number of trials. In each trial it chooses two random messages of length twice that of a hash, then hashes them, and compares the hash values at the chosen offsets. We expect this to find a collision every 1 in 2^b trials, where $b = 8^o$ and o is the number of offset bytes checked, i.e., we expect to find a collision every 1 in 256 trials with a single offset, and every 1 in 65536 trials when checking for two offsets.

Using the parameters in this paper we obtained the following test results.

Case 1: $\lambda = 5$

- 1-octet collisions (25600 iterations per offset, so expecting 100):

  min: 81
  max: 124
  mean: 101.63
  median: 101.5
  stddev: 9.97

- 2-octet collisions (655360 iterations, expecting 10):

  min: 1
  max: 23
  mean: 9.96
  median: 10
  stddev is 3.16

Case 2: $\lambda = 7$

- 1-octet collisions (25600 iterations per offset, so expecting 100):

  min: 88
  max: 127
  mean: 100.16
  median: 97
  stddev: 8.99

- 2-octet collisions (655360 iterations, expecting 10):

  min: 1
  max: 22
  mean: 10.03
  median: 10
  stddev is 3.29

NIST Statistical Test Suite

We ran the AEHash through the NIST Statistical Test Suite [14]. This is a statistical package that was developed to test the randomness of (arbitrarily long) binary sequences produced by either hardware or software based cryptographic random or pseudorandom number generators. The tests focus on a variety of different types of non-randomness that could exist within a given sequence. The package then applies standard statistical significance methodology to produce a p-value for each test, which can then be compared to a (standard) level of significance α. For our tests, we chose $\alpha = 0.01$, as recommended by [14].

Altogether there are fifteen different tests in the suite, some of which are broken up into various subtests. Among the full set of tests are the following:

– The *Frequency (Monobit) Test*, which gives a most basic test of randomness of a sequence. It determines whether the number of ones and zeros in a sequence are approximately the same as would be expected for a truly random sequence.

– The *Runs Test*. By definition a run in a sequence is a subsequence of identical elements. This test decides whether the number of runs of ones and zeros of various lengths is as expected for a random sequence. In particular, it detects whether the oscillation between such zeros and ones is too fast or too slow.

– The *Binary Matrix Rank Test*, which checks for linear dependencies among fixed length substrings of the original sequence.

– The *Non-overlapping Template Matching Test*. This test counts the number of occurrences of a fixed set of pre-specified target strings, and rejects sequences exhibiting too many or too few occurrences of a given aperiodic pattern For a given target string of length m, an m-bit window is used to search for this pattern. If the pattern is not found, the window slides one bit position to the right, and the search is continued. If the pattern is found, then the window is reset to the bit after the found pattern, and again the search continues. The NIST suite uses a collection of 149 aperiodic patterns for this test.

For full details of all the tests in the suite, we refer to [14].

In our tests we generated the test data by creating an 8-byte input message using a counter and then hashing the input message (counter) until we had sufficient data to produce 10 streams of 100,000 bits each. These streams were run through the test suite. Overall the AE Hash function performed extremely well. For only 6 of the 149 non-overlapping template tests did any bitstream appear nonrandom according to the test, and in each case it was only 1 of the 10 bitstreams. All 161 computed p-values were above $\alpha = 0.01$ save for two, which were 0.0043 and 0.0088. To put these results in perspective, we also tested 10 bitstreams of 100,000 bits generated using output from `/dev/random`, which is a special file found on Unix-like systems that gathers environmental noise from device drivers and other sources to produce a blocking pseudorandom number generator. These bitstreams exhibited similar behavior, and in fact failed 22 of the non-overlapping template tests. Two p-values from the bitstreams from `/dev/random` were also found to be lower than our threshold.