Faster Binary-Field Multiplication and Faster Binary-Field MACs

Daniel J. Bernstein[1,2] and Tung Chou[2]

[1] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7053, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, the Netherlands
blueprint@crypto.tw

Abstract. This paper shows how to securely authenticate messages using just 29 bit operations per authenticated bit, plus a constant overhead per message. The authenticator is a standard type of "universal" hash function providing information-theoretic security; what is new is computing this type of hash function at very high speed.

At a lower level, this paper shows how to multiply two elements of a field of size 2^{128} using just $9062 \approx 71 \cdot 128$ bit operations, and how to multiply two elements of a field of size 2^{256} using just $22164 \approx 87 \cdot 256$ bit operations. This performance relies on a new representation of field elements and new FFT-based multiplication techniques.

This paper's constant-time software uses just 1.89 Core 2 cycles per byte to authenticate very long messages. On a Sandy Bridge it takes 1.43 cycles per byte, without using Intel's PCLMULQDQ polynomial-multiplication hardware. This is much faster than the speed records for constant-time implementations of GHASH without PCLMULQDQ (over 10 cycles/byte), even faster than Intel's best Sandy Bridge implementation of GHASH with PCLMULQDQ (1.79 cycles/byte), and almost as fast as state-of-the-art 128-bit prime-field MACs using Intel's integer-multiplication hardware (around 1 cycle/byte).

Key words: Performance · FFTs · Polynomial multiplication · Universal hashing · Message authentication

1 Introduction

NIST's standard AES-GCM authenticated-encryption scheme uses GHASH to authenticate ciphertext (produced by AES in counter mode) and to authenticate additional data. GHASH converts its inputs into a polynomial and evaluates that

This work was supported by the National Science Foundation under grant 1018836 and by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005. Permanent ID of this document: 393655eb413348f4e17c7ec451b9e159. Date: 2020.12.28.

polynomial at a secret element of $\mathbb{F}_{2^{128}} = \mathbb{F}_2[x]/(x^{128}+x^7+x^2+x+1)$, using one multiplication in $\mathbb{F}_{2^{128}}$ for each 128-bit input block. The cost of GHASH is an important part of the cost of GCM, and it becomes almost the entire cost when large amounts of non-confidential data are being authenticated without being encrypted, or when a denial-of-service attack is sending a flood of forgeries to consume all available processing time.

Most AES-GCM software implementations rely heavily on table lookups and presumably leak their keys to cache-timing attacks. Käsper and Schwabe [35] (CHES 2009) addressed this problem by introducing a constant-time implementation of AES-GCM using 128-bit vector instructions. Their GHASH implementation takes 14.4 cycles/byte on one core of an Intel Core 2 processor. On a newer Intel Sandy Bridge processor the same software takes 13.1 cycles/byte. For comparison, HMAC-SHA1, which is widely used in Internet applications, takes 6.74 Core 2 cycles/byte and 5.18 Sandy Bridge cycles/byte.

1.1 Integer-Multiplication Hardware. Much better speeds than GHASH were already provided by constant-time MACs that used integer multiplication rather than multiplication of polynomials mod 2. Examples include UMAC [15], Poly1305 [5], and VMAC [36]. Current Poly1305 software from [22] runs at 1.89 Core 2 cycles/byte and 1.22 Sandy Bridge cycles/byte. VMAC, which uses "pseudo dot products" (see Section 4), is even faster than Poly1305.

CPUs include large integer-multiplication units to support many different applications, so it is not a surprise that these MACs are much faster in software than GHASH (including non-constant-time GHASH software; see [35]). However, integer multiplication uses many more bit operations than multiplication of polynomials mod 2, so for hardware designers these MACs are much less attractive. MAC choice is a continuing source of tension between software designers and hardware designers.

1.2 New Speeds for Binary-Field MACs. This paper introduces Auth256, an $\mathbb{F}_{2^{256}}$-based MAC at a 2^{255} security level; and a constant-time software implementation of Auth256 running at just 1.89 cycles/byte on a Core 2. We also tried our software on a Sandy Bridge; it runs at just 1.43 cycles/byte. We also have a preliminary Cortex-A8 implementation below 14 cycles/byte.

This new binary-field MAC is not quite as fast as integer-multiplication MACs. However, the gap is quite small, while the hardware advantages of binary fields are quite important. We plan to put our software into the public domain.

Caveat: All of the above performance figures ignore short-message overhead, and in particular our software has very large overhead, tens of thousands of cycles. For 32-, 64-, 128-kilobyte messages, our software takes 3.07, 2.44, 2.14 Core 2 cycles per byte. and 2.85, 2.09, 1.74 Sandy Bridge cycles per byte. This software is designed primarily for authenticating large files, not for authenticating network packets. However, a variant of Auth256 ($b = 1$ in Section 6) takes only 0.81 additional cycles/byte and has much smaller overhead. We also expect that, compared to previous MAC designs, this variant will allow significantly lower area for high-throughput hardware, as explained below.

1.3 New Bit-Operation Records for Binary-Field Multiplication. The software speed advantage of Auth256 over GHASH, despite the much higher security level of Auth256, is easily explained by the following comparison. Schoolbook multiplication would take 128^2 ANDs and approximately 128^2 XORs for each 128 bits of GHASH input, i.e., approximately 256 bit operations per authenticated bit. Computing a 256-bit authenticator in the same way would use approximately 512 bit operations per authenticated bit. Auth256 uses just 29 bit operations per authenticated bit.

Of course, Karatsuba's method saves many bit operations at this size. See, e.g., [4], [42], [18], [48], [45], [41], [8], [9], and [21]. Bernstein's Karatsuba/Toom combination in [9] multiplies 256-bit polynomials using only about $133 \cdot 256$ bit operations. Multiplying 256-bit field elements has only a small overhead. However, 133 bit operations is still much larger than 29 bit operations.

Our improved number of bit operations is a combination of four factors. The first factor is faster multiplication: we reduce the cost of multiplication in $\mathbb{F}_{2^{256}}$ from $133 \cdot 256$ bit operations to just $22292 \approx 87 \cdot 256$ bit operations. The second factor, which we do not take credit for, is the use of pseudo dot products to reduce the number of multiplications by a factor of 2, reducing 87 below 44. The third factor, which reduces 44 to 32, is an extra speedup from an interaction between the structure of pseudo dot products and the structure of the multiplication algorithms that we use. The fourth factor, which reduces 32 to just 29, is to use a different field representation for the input to favor the fast multiplication algorithm we use.

Specifically, we use a fast Fourier transform (FFT) to multiply polynomials in $\mathbb{F}_{2^8}[x]$. The FFT is advertised in algorithm courses as using an essentially linear number of field additions and multiplications but is generally believed to be much slower than other multiplication methods for cryptographic sizes. Bernstein, Chou, and Schwabe showed at CHES 2013 [11] that a new "additive FFT" saves time for decryption in McEliece's code-based public-key cryptosystem, but the smallest FFT sizes in [11] were above 10000 bits (evaluation at every element in $\mathbb{F}_{2^m}$, where $m \geq 11$). We introduce an improved additive FFT that uses fewer bit operations than any previously known multiplier for fields as small as $\mathbb{F}_{2^{64}}$, provided that the fields contain $\mathbb{F}_{2^8}$. Our additive FFT, like many AES hardware implementations, relies heavily on a tower-field representation of $\mathbb{F}_{2^8}$, but benefits from this representation in different ways from AES. The extra speedup inside pseudo dot products comes from merging inverse FFTs, which requires breaking the standard encapsulation of polynomial multiplication; see Section 4.

The fact that we are optimizing bit operations is also the reason that we expect our techniques to produce low area for high-throughput hardware. Optimizing the area of a fully unrolled hardware multiplier is highly correlated with optimizing the number of bit operations. We do not claim relevance to very small serial multipliers.

1.4 Polynomial-Multiplication Hardware: PCLMULQDQ. Soon after [35], in response to the performance and security problems of AES-GCM soft-

ware, Intel added "AES New Instructions" to some of its CPUs. These instructions include PCLMULQDQ, which computes a 64-bit polynomial multiplication in $\mathbb{F}_2[x]$.

Krovetz and Rogaway reported in [37] that GHASH takes 2 Westmere cycles/byte using PCLMULQDQ. Intel's Shay Gueron reported in [26] that heavily optimized GHASH implementations using PCLMULQDQ take 1.79 Sandy Bridge cycles/byte. Our results are faster at a higher security level, although they do require switching to a different authenticator.

Of course, putting sufficient resources into a hardware implementation will beat any software implementation. To quantify this, consider what is required for GHASH to run faster than 1.43 cycles/byte using PCLMULQDQ. GHASH performs one multiplication for every 16 bytes of input, so it cannot afford more than 22.88 cycles for each multiplication. If PCLMULQDQ takes t cycles and t is not very small then presumably Karatsuba is the best approach to multiplication in $\mathbb{F}_{2^{128}}$, taking $3t$ cycles plus some cycles for latency, additions, and reductions.

The latest version of Fog's well-known performance survey [23] indicates that $t = 7$ for AMD Bulldozer, Piledriver, and Steamroller and that $t = 8$ for Intel Sandy Bridge and Ivy Bridge; on the other hand, $t = 2$ for Intel Haswell and $t = 1$ for AMD Jaguar. Gueron, in line with this analysis, reported 0.40 Haswell cycles/byte for GHASH.

It is quite unclear what to expect from future CPUs. Intel did not put PCLMULQDQ hardware into its low-cost "Core i3" lines of Sandy Bridge, Ivy Bridge, and Haswell CPUs; and obviously Intel is under pressure from other manufacturers of small, low-cost CPUs. To emphasize the applicability of our techniques to a broad range of CPUs, we have avoided PCLMULQDQ in our software.

2 Field Arithmetic in $\mathbb{F}_{2^8}$

This section reports optimized circuits for field arithmetic in $\mathbb{F}_{2^8}$. We write "circuit" here to mean a fully unrolled combinatorial circuit consisting of AND gates and XOR gates. Our main cost metric is the total number of bit operations, i.e., the total number of AND gates and XOR gates, although as a secondary metric we try to reduce the number of registers required in our software.

Subsequent sections use these circuits as building blocks. The techniques also apply to larger $\mathbb{F}_{2^s}$, but $\mathbb{F}_{2^8}$ is large enough to support the FFTs that we use in this paper.

2.1 Review of Tower Fields. We first construct $\mathbb{F}_{2^2}$ in the usual way as $\mathbb{F}_2[x_2]/(x_2^2 + x_2 + 1)$. We write α_2 for the image of x_2 in $\mathbb{F}_{2^2}$, so $\alpha_2^2 + \alpha_2 + 1 = 0$. We represent elements of $\mathbb{F}_{2^2}$ as linear combinations of 1 and α_2, where the coefficients are in $\mathbb{F}_2$. Additions in $\mathbb{F}_{2^2}$ use 2 bit operations, namely 2 XORs.

We construct $\mathbb{F}_{2^4}$ as $\mathbb{F}_{2^2}[x_4]/(x_4^2 + x_4 + \alpha_2)$, rather than using a polynomial basis for $\mathbb{F}_{2^4}$ over $\mathbb{F}_2$. We write α_4 for the image of x_4 in $\mathbb{F}_{2^4}$. We represent elements of $\mathbb{F}_{2^4}$ as linear combinations of 1 and α_4, where the coefficients are in $\mathbb{F}_{2^2}$. Additions in $\mathbb{F}_{2^4}$ use 4 bit operations.

Finally, we construct $\mathbb{F}_{2^8}$ as $\mathbb{F}_{2^4}[x_8]/(x_8^2 + x_8 + \alpha_2\alpha_4)$; write α_8 for the image of x_8 in $\mathbb{F}_{2^8}$; and represent elements of $\mathbb{F}_{2^8}$ as $\mathbb{F}_{2^4}$-linear combinations of 1 and α_8. Additions in $\mathbb{F}_{2^8}$ use 8 bit operations.

2.2 Variable Multiplications. A variable multiplication is the computation of ab given $a, b \in \mathbb{F}_{2^s}$ as input. We say "variable multiplication" to distinguish this operation from multiplication by a constant; we will optimize constant multiplication later.

For variable multiplication in $\mathbb{F}_{2^2}$, we perform a multiplication of $a_0+a_1x, b_0+b_1x \in \mathbb{F}_2[x]$ and reduction modulo x^2+x+1. Here is a straightforward sequence of 7 operations using schoolbook polynomial multiplication: $c_0 \leftarrow a_0 \otimes b_0; c_1 \leftarrow a_0 \otimes b_1; c_2 \leftarrow a_1 \otimes b_0; c_3 \leftarrow a_1 \otimes b_1; c_4 \leftarrow c_1 \oplus c_2; c_5 \leftarrow c_0 \oplus c_3; c_6 \leftarrow c_4 \oplus c_3$. The result is c_5, c_6.

For $\mathbb{F}_{2^4}$ and $\mathbb{F}_{2^8}$ we use 2-way Karatsuba. Note that since the irreducible polynomials are of the form $x^2 + x + \alpha$ the reductions involve a different type of multiplication described below: multiplication of a field element with a constant.

We end up with just 110 bit operations for variable multiplication in $\mathbb{F}_{2^8}$. For comparison, Bernstein [9] reported 100 bit operations to multiply 8-bit polynomials in $\mathbb{F}_2[x]$, but reducing modulo an irreducible polynomial costs many extra operations. A team led by NIST [19], improving upon various previous results such as [31], reported 117 bit operations to multiply in $\mathbb{F}_2[x]$ modulo the AES polynomial $x^8 + x^4 + x^3 + x + 1$.

2.3 Constant Multiplications. A constant multiplication in $\mathbb{F}_{2^s}$ is the computation of αb given $b \in \mathbb{F}_{2^s}$ as input for some constant $\alpha \in \mathbb{F}_{2^s}$. This is trivial for $\alpha \in \mathbb{F}_2$ so we focus on $\alpha \in \mathbb{F}_{2^s} \backslash \mathbb{F}_2$. One can substitute a specific α into our 110-gate circuit for variable multiplication to obtain a circuit for constant multiplication, and then shorten the circuit by eliminating multiplications by 0, multiplications by 1, additions of 0, etc.; but for small fields it is much better to use generic techniques to optimize the cost of multiplying by a constant matrix.

Our linear-map circuit generator combines various features of Paar's greedy additive common-subexpression elimination algorithm [40] and Bernstein's two-operand "xor-largest" algorithm [10]. For $\alpha \in \mathbb{F}_{2^8} \backslash \mathbb{F}_2$ our constant-multiplication circuits use 14.83 gates on average. Compared to Paar's results, this is slightly more gates but is much better in register use; compared to Bernstein's results, it is considerably fewer gates.

The real importance of the tower-field construction for us is that constant multiplications become much faster when the constants are in subfields. Multiplying an element of $\mathbb{F}_{2^8}$ by a constant $\alpha \in \mathbb{F}_{2^4} \backslash \mathbb{F}_2$ takes only 7.43 gates on average, and multiplying an element of $\mathbb{F}_{2^8}$ by a constant $\alpha \in \mathbb{F}_{2^2} \backslash \mathbb{F}_2$ takes only 4 gates on average. The constant multiplications in our FFT-based multiplication algorithms for $\mathbb{F}_{2^{256}}$ (see Section 3) are often in subfields of $\mathbb{F}_{2^8}$, and end up using only 9.02 gates on average.

2.4 A Circuit Generator for Invertible Linear Maps over $\mathbb{F}_2$. Given an invertible linear map $\phi : \mathbb{F}_2^n \to \mathbb{F}_2^n$, our circuit generator generates a circuit, represented as a sequence of XOR gates, that carries out the map. That is, the

resulting circuit, given an n-bit string representing $\vec{v} \in \mathbb{F}_2^n$, computes an n-bit string representing $\vec{w} = \phi(\vec{v}) \in \mathbb{F}_2^n$.

Such sequence must exist since the map is the composition of a sequence of elementary row (if $\mathbb{F}_2^n$ is treated as a space of column vectors) operations that add one row to another, and each of such operations can be carried out by a XOR gate. Here we ignore the row swapping operations for a moment, which will be justified later.

Assume a circuit for the map takes as input a bit string $a_1, a_2, \ldots, a_n$. Each bit in the output bit string is then $\sum_{i \in I} a_i$ for some $I \subseteq \{1, \ldots, n\}$. Each output bit is thus represented as a polynomial $\sum_{i \in I} x_i \in \mathbb{F}_2[x_1, x_2, \ldots, x_n]$; the resulting array of polynomials is called the **final state**. The input can also be expressed in similar way: simply express a_i as x_i; the resulting array of polynomials is called the **initial state**.

The circuit generator starts with initializing an array $\mathcal{A}$ to the final state. The goal is to move from the final state to the initial state by a sequence of in-place additions. Each in-place addition $\mathcal{A}[i] \leftarrow \mathcal{A}[i] + \mathcal{A}[j]$ represents a "gate" $a_i \leftarrow a_i \oplus a_j$; When the goal is reached the sequence of gates is reversed. With proper renaming of the inputs and outputs of the gates the circuit is obtained.

The sequence finding is done by finding operations in the sequence one-by-one. To generate the next operation, all $n(n-1)$ possibilities are considered and evaluated. Now let $|s|$ denote the number of unknowns appearing in $s \in \mathbb{F}_2[x_1, x_2, \ldots, x_n]$. The evaluation is based on how much progress the operation brings, which is quantified as $\sum_i |\mathcal{A}'[i]| - \sum_i |\mathcal{A}[i]|$, where $\mathcal{A}'$ is the result of applying the operation on $\mathcal{A}$. This makes sense since the smaller $\sum_i |\mathcal{A}[i]|$ is, the closer $\mathcal{A}$ is to the initial state: the minimum value is exactly $n = \sum_i |x_i|$. After evaluating all the possibilities, we pick at random one of the operations that makes the most progress. The same step is repeated until $\sum_i |\mathcal{A}[i]| = n$.

Here are some details about the code generator:

- Our circuit generator takes a greedy approach; it does not necessarily generate optimal result. Also since randomness is involved in our circuit generator, it can generate a different result each time. Sometimes it can not even reach the initial state and thus fails to generate a circuit. We always run the generator several times and pick the best result.
- Precisely speaking the circuit generator move from the final state to the initial state but up to some permutation. This is something that needs to be taken care of by the renaming.

2.5 Subfields and Decomposability. A further advantage of the tower-field construction, beyond the number of bit operations, is that it allows constant multiplications by subfield elements to be decomposed into independent subcomputations. For example, when an $\mathbb{F}_{2^8}$ element in this representation is multiplied by a constant in $\mathbb{F}_{2^2}$, the computation decomposes naturally into 4 independent subcomputations, each of which takes 2 input bits to 2 output bits.

Decomposability is a nice feature for software designers; it guarantees a smaller working set, which in general implies easier optimization, fewer memory operations and cache misses, etc. The ideal case is when the working set can fit into

registers; in this case the computation can be done using the minimum number of memory accesses. Section 5 gives an example of how decomposability can be exploited to help optimization of a software implementation.

The decomposition of multiplication by a constant in a subfield has the extra feature that the subcomputations are identical. This allows extra possibilities for efficient vectorization in software, and can also be useful in hardware implementations that reuse the same circuit several times. Even when subcomputations are not identical, decomposability increases flexibility of design and is desirable in general.

3 Faster Additive FFTs

Given a 2^{m-1}-coefficient polynomial f with coefficients in $\mathbb{F}_{2^8}$, a size-2^m additive FFT computes $f(0), f(\beta_m), f(\beta_{m-1}), f(\beta_m + \beta_{m-1}), f(\beta_{m-2})$, etc., where $\beta_m, \ldots, \beta_2, \beta_1$ are $\mathbb{F}_2$-linearly independent elements of $\mathbb{F}_{2^8}$ specified by the algorithm. We always choose a "Cantor basis", i.e., elements $\beta_m, \ldots, \beta_2, \beta_1$ satisfying $\beta_{i+1}^2 + \beta_{i+1} = \beta_i$ and $\beta_1 = 1$; specifically, we take $\beta_1 = 1$, $\beta_2 = \alpha_2$, $\beta_3 = \alpha_4 + 1$, $\beta_4 = \alpha_2\alpha_4$, $\beta_5 = \alpha_8$, and $\beta_6 = \alpha_2\alpha_8 + \alpha_2\alpha_4 + \alpha_2 + 1$. We do not need larger FFT sizes in this paper.

Our additive FFT is an improvement of the Bernstein–Chou–Schwabe [11] additive FFT, which in turn is an improvement of the Gao–Mateer [24] additive FFT. This section presents details of our size-4, size-8, and size-16 additive FFTs over $\mathbb{F}_{2^8}$. All of our improvements are already visible for size 16. At the end of the section gate counts for all sizes are collected and compared with state-of-the-art Karatsuba/Toom-based methods.

3.1 Size-4 FFTs: the Lowest Level of Recursion. Given a polynomial $f = a + bx \in \mathbb{F}_{2^8}[x]$, the size-4 FFT computes $f(0) = a, f(\beta_2) = a + \beta_2 b, f(1) = a + b, f(\beta_2 + 1) = a + (\beta_2 + 1)b$. Recall that $\beta_2 = \alpha_2$ so $\beta_2^2 + \beta_2 + 1 = 0$. The size-4 FFT is of interest because it serves as the lowest level of recursion for larger-size FFTs.

As mentioned in Section 2, since $\beta_2 \in \mathbb{F}_{2^2}$, the size-4 FFT can be viewed as a collection of 4 independent pieces, each dealing with only 2 out of the 8 bits.

Let a_0, a_1 be the first 2 bits of a; similarly for b. Then a_0, a_1 and b_0, b_1 represent $a_0 + a_1\beta_2, b_0 + b_1\beta_2 \in \mathbb{F}_{2^2}$. Since $\beta_2(a_0 + a_1\beta_2) = a_1 + (a_0 + a_1)\beta_2$, a 6-gate circuit that carries out the size-4 FFTs operations on the first 2 bits is $c_{00} \leftarrow a_0; c_{01} \leftarrow a_1; c_{20} \leftarrow a_0 \oplus b_0; c_{21} \leftarrow a_1 \oplus b_1; c_{10} \leftarrow a_0 \oplus b_1; c_{31} \leftarrow a_1 \oplus b_0; c_{11} \leftarrow c_{31} \oplus b_1; c_{30} \leftarrow c_{10} \oplus b_0$. Then c_{00}, c_{01} is the 2-bit result of a; c_{10}, c_{11} is the 2-bit result of $a + \beta_2 b$; similarly for c_{20}, c_{21} and c_{30}, c_{31}. In conclusion, a size-4 FFT can be carried out using a $6 \cdot 4 = 24$-gate circuit.

The whole computation costs the same as merely 3 additions in $\mathbb{F}_{2^8}$. This is the result of having evaluation points to be in the smallest possible subfield, namely $\mathbb{F}_{2^2}$, and the use of tower field construction for $\mathbb{F}_{2^8}$.

3.2 The Size-8 FFTs: the First Recursive Case. Given a polynomial $f = f_0 + f_1x + f_2x^2 + f_3x^3 \in \mathbb{F}_{2^8}[x]$, the size-8 FFT computes $f(0), f(\beta_3), f(\beta_2), f(\beta_2 +$

$\beta_3), f(1), f(\beta_3+1), f(\beta_2+1), f(\beta_2+\beta_3+1)$. Recall that $\beta_3 = \alpha_4 + 1$ so $\beta_3^2 + \beta_3 + \beta_2 = 0$. The size-8 FFT is of interest because it is the smallest FFT that involves recursion.

In general, a recursive size-2^m FFT starts with a radix conversion that computes $f^{(0)}$ and $f^{(1)}$ such that $f = f^{(0)}(x^2+x) + xf^{(1)}(x^2+x)$. When f is a 2^{m-1}-coefficient polynomial we call this a size-2^{m-1} radix conversion. Since the size-4 radix conversion can be viewed as a change of basis in $\mathbb{F}_2^4$, each coefficient in $f^{(0)}$ and $f^{(1)}$ is a subset sum of f_0, f_1, f_2, and f_3. In fact, $f^{(0)} = f_0 + (f_2+f_3)x$ and $f^{(1)} = (f_1+f_2+f_3) + f_3x$ can be computed using exactly 2 additions.

After the radix conversion, 2 size-4 FFTs are invoked to evaluate $f^{(0)}$, $f^{(1)}$ at $0^2+0 = 0$, $\beta_3^2+\beta_3 = \beta_2$, $\beta_2^2+\beta_2 = 1$, and $(\beta_2+\beta_3)^2+(\beta_2+\beta_3) = \beta_2+1$. Each of these size-4 FFTs takes 24 bit operations.

Note that we have

$$f(\alpha) = f^{(0)}(\alpha^2+\alpha) + \alpha f^{(1)}(\alpha^2+\alpha),$$
$$f(\alpha+1) = f^{(0)}(\alpha^2+\alpha) + (\alpha+1) f^{(1)}(\alpha^2+\alpha).$$

Starting from $f^{(0)}(\alpha^2+\alpha)$ and $f^{(1)}(\alpha^2+\alpha)$, Gao and Mateer multiply $f^{(1)}(\alpha^2+\alpha)$ by α and add $f^{(0)}(\alpha^2+\alpha)$ to obtain $f(\alpha)$, and then add $f^{(1)}(\alpha^2+\alpha)$ with $f(\alpha)$ to obtain $f(\alpha+1)$. We call this a **muladdadd** operation.

The additive FFT thus computes all the pairs $f(\alpha), f(\alpha+1)$ at once: given $f^{(0)}(0)$ and $f^{(1)}(0)$ apply muladdadd to obtain $f(0)$ and $f(1)$, given $f^{(0)}(\beta_2)$ and $f^{(1)}(\beta_2)$ apply muladdadd operation to obtain $f(\beta_3)$ and $f(\beta_3+1)$, and so on.

The way that the output elements form pairs is a result of having 1 as the last basis element. In general the Gao–Mateer FFT is able to handle the case where 1 is not in the basis with some added cost, but here we avoid the cost by making 1 the last basis element.

Generalizing this to the case of size-2^m FFTs implies that the i-th output element of $f^{(0)}$ and $f^{(1)}$ work together to form the ith and $(i+2^{m-1})$th output element for f. We call the collection of muladdadds that are used to combine 2 size-2^{m-1} FFT outputs to form a size-2^m FFT output a **size-2^m combine routine**.

We use our circuit generator introduced in Section 2 to generate the circuits for all the constant multiplications. The muladdadds take a total of 76 gates. Therefore, a size-8 FFT can be carried out using $2 \cdot 8 + 2 \cdot 24 + 76 = 140$ gates.

Note that for a size-8 FFT we again benefit from the special basis and the $\mathbb{F}_{2^8}$ construction. The recursive calls still use the good basis $\beta_2, 1$ so that there are only constant multiplications by $\mathbb{F}_{2^2}$ elements. The combine routine, although not having only constant multiplications by $\mathbb{F}_{2^2}$ elements, at least has only constant multiplications by $\mathbb{F}_{2^4}$ elements.

3.3 The Size-16 FFTs: Saving Additions for Radix Conversions. The size-16 FFT is the smallest FFT in which non-trivial radix conversions happen in recursive calls. Gao and Mateer presented an algorithm performing a size-2^n radix conversion using $(n-1)2^{n-1}$ additions. We do better by combining additions across levels of recursion.

The size-8 radix conversion finds $f^{(0)}$, $f^{(1)}$ such that $f = f^{(0)}(x^2+x) + xf^{(1)}(x^2+x)$. The two size-4 radix conversion in size-8 FFT subroutines find $f^{(i0)}$, $f^{(i1)}$ such that $f^{(i)} = f^{(i0)}(x^2+x)+xf^{(i1)}(x^2+x)$ for $i \in \{0,1\}$. Combining all these leads to $f = f^{(00)}(x^4+x) + (x^2+x)f^{(01)}(x^4+x) + xf^{(10)}(x^4+x) + x(x^2+x)f^{(11)}(x^4+x)$.

In the end the size-8 and the two size-4 radix conversions together compute from f the following: $f^{(00)} = f_0 + (f_4+f_7)x$, $f^{(01)} = (f_2+f_3+f_5+f_6) + (f_6+f_7)x$, $f^{(10)} = (f_1+f_2+f_3+f_4+f_5+f_6+f_7) + (f_5+f_6+f_7)x$, and $f^{(11)} = (f_3+f_6) + f_7x$. The Gao–Mateer algorithm takes 12 additions for this computation, but one sees by hand that 8 additions suffice. One can also obtain this result by applying the circuit generator introduced in Section 2. Here is an 8-addition sequence generated by the circuit generator: $f_0^{(00)} \leftarrow f_0$; $f_1^{(11)} \leftarrow f_7$; $f_1^{(00)} \leftarrow f_4 + f_7$; $f_0^{(01)} \leftarrow f_2 + f_5$; $f_0^{(11)} \leftarrow f_3 + f_6$; $f_1^{(01)} \leftarrow f_6 + f_7$; $f_0^{(10)} \leftarrow f_1 + f_1^{(00)}$; $f_1^{(10)} \leftarrow f_5 + f_1^{(01)}$; $f_0^{(01)} \leftarrow f_0^{(01)} + f_0^{(11)}$; $f_0^{(10)} \leftarrow f_0^{(10)} + f_0^{(01)}$.

We applied the circuit generator for larger FFTs and found larger gains. A size-32 FFT, in which the input is a size-16 polynomial, requires 31 rather than 48 additions for radix conversions. A size-64 FFT, in which the input is a size-32 polynomial, requires 82 rather than 160 additions for radix conversions.

We also applied our circuit generator to the muladdadds, obtaining a 170-gate circuit for the size-16 combined routine and thus a size-16 FFT circuit using $8 \cdot 8 + 4 \cdot 24 + 2 \cdot 76 + 170 = 482$ gates.

3.4 Size-16 FFTs Continued: Decomposition at Field-Element Level. The size-16 FFT also illustrates the decomposability of the combine routines of a FFT. Consider the size-16 and size-8 combine routines; the computation takes as input the FFT outputs for the $f^{(ij)}$'s to compute the FFT output for f.

Let the output for f be $a_0, a_1, \ldots, a_{15}$, the output for $f^{(i)}$ be $a_0^{(i)}, a_1^{(i)}, \ldots, a_7^{(i)}$, and similarly for $f^{(ij)}$. For $k \in \{0,1,2,3\}$, a_k, a_{k+8} are functions of $a_k^{(0)}$ and $a_k^{(1)}$, which in turn are functions of $a_k^{(00)}$, $a_k^{(01)}$, $a_k^{(10)}$, and $a_k^{(11)}$; a_{k+4}, a_{k+12} are functions of $a_{k+4}^{(0)}$ and $a_{k+4}^{(1)}$, which in turn are functions of the same 4 elements. We conclude that $a_k, a_{k+4}, a_{k+8}, a_{k+12}$ depend only on $a_k^{(00)}$, $a_k^{(01)}$, $a_k^{(10)}$, and $a_k^{(11)}$. In this way, the computation is decomposed into 4 independent parts; each takes as input 4 field elements and outputs 4 field elements. Note that here the decomposition is at the field-element level, while Section 2 considered decomposability at the bit level.

More generally, for size-2^m FFTs we suggest decomposing k levels of combine routines into 2^{m-k} independent pieces, each taking 2^k $\mathbb{F}_{2^8}$ elements as input and producing 2^k $\mathbb{F}_{2^8}$ elements as output.

3.5 Improvements: a Summary. We have two main improvements to the additive FFT: reducing the cost of multiplications and reducing the number of additions in radix conversion. We also use these ideas to accelerate size-32 and size-64 FFTs, and obviously they would also save time for larger FFTs.

The reduction in the cost of multiplications is a result of (1) choosing a "good" basis for which constant multiplications use constants in the smallest possible

b	forward	pointwise	inverse	total	competition
16	$2 \cdot 24$	$4 \cdot 110$	60	$448 \approx 14 \cdot 2 \cdot 16$	$350 \approx 10.9 \cdot 2 \cdot 16$
32	$2 \cdot 140$	$8 \cdot 110$	228	$1388 \approx 21.7 \cdot 2 \cdot 32$	$1158 \approx 18.1 \cdot 2 \cdot 32$
64	$2 \cdot 482$	$16 \cdot 110$	746	$3470 \approx 27.1 \cdot 2 \cdot 64$	$3682 \approx 28.8 \cdot 2 \cdot 64$
128	$2 \cdot 1498$	$32 \cdot 110$	2066	$8582 \approx 33.5 \cdot 2 \cdot 128$	$11486 \approx 44.9 \cdot 2 \cdot 128$
256	$2 \cdot 4068$	$64 \cdot 110$	5996	$21172 \approx 41.4 \cdot 2 \cdot 256$	$34079 \approx 66.6 \cdot 2 \cdot 256$

Table 1. Cost of multiplying $b/8$-coefficient polynomials over $\mathbb{F}_{2^8}$. "Forward" is the cost of two size-$b/4$ FFTs with size-$b/8$ inputs. "Pointwise" is the cost of pointwise multiplication. "Inverse" is the cost of an inverse size-$b/4$ FFT. "Total" is the sum of forward, pointwise, and inverse. "Competition" is the cost from [9] of an optimized Karatsuba/Toom multiplication of b-coefficient polynomials over $\mathbb{F}_2$; note that slight improvements appear in [21].

subfield; (2) using a tower-field representation to accelerate those constant multiplications; and (3) searching for short sequences of additions. The reduction of additions for radix conversion is a result of (1) merging radix conversion at different levels of recursion and again (2) searching for short sequences of additions.

3.6 Polynomial Multiplications: a Comparison with Karatsuba and Toom. Just like other FFT algorithms, any additive FFT can be used to multiply polynomials. Given two 2^{m-1}-coefficient polynomials in $\mathbb{F}_{2^s}$, we apply a size-2^m additive FFT to each polynomial, a pointwise multiplication consisting of 2^m variable multiplications in $\mathbb{F}_{2^s}$, and a size-2^m inverse additive FFT, i.e., the inverse of an FFT with both input and output size 2^m. An FFT (or inverse FFT) with input and output size 2^m is slightly more expensive than an FFT with input size 2^{m-1} and output size 2^m: input size 2^{m-1} is essentially input size 2^m with various 0 computations suppressed.

Table 1 summarizes the number of bit operations required for multiplying b-bit (i.e., $b/8$-coefficient) polynomials in $\mathbb{F}_{2^8}[x]$. Field multiplication is slightly more expensive than polynomial multiplication. For $\mathbb{F}_{2^{256}}$ we use the polynomial $x^{32} + x^{17} + x^2 + \alpha_8$; reduction costs 992 bit operations. However, as explained in Section 4, in the context of Auth256 we can almost eliminate the inverse FFT and the reduction, and eliminate many operations in the forward FFTs, making the additive FFT even more favorable than Karatsuba.

4 The Auth256 Message-Authentication Code: Major Features

Auth256, like GCM's GHASH, follows the well-known Wegman–Carter [47] recipe for building an MAC with (provably) information-theoretic security. The recipe is to apply a (provably) "δ-xor-universal hash" to the message and encrypt the result with a one-time pad. Every forgery attempt then (provably) has success probability at most δ, no matter how much computer power the attacker used.

Of course, real attackers do not have unlimited computer power, so GCM actually replaces the one-time pad with counter-mode AES output to reduce key size. This is safe against any attacker who cannot distinguish AES output from uniform random; see, e.g., [33, comments after Corollary 3]. Similarly, it is safe to replace the one-time pad in Auth256 with cipher output.

This section presents two important design decisions for Hash256, the hash function inside Auth256. Section 4.1 describes the advantages of the Hash256 output size. Section 4.2 describes the choice of pseudo dot products inside Hash256, and the important interaction between FFTs and pseudo dot products. Section 4.3 describes the use of a special field representation for inputs to reduce the cost of FFTs.

Section 6 presents, for completeness, various details of Hash256 and Auth256 that are not relevant to this paper's performance evaluation.

4.1 Output size: Bigger-Birthday-Bound Security. Hash256 produces 256-bit outputs, as its name suggests, and Auth256 produces 256-bit authenticators. Our multiplication techniques are only slightly slower per bit for $\mathbb{F}_{2^{256}}$ than for $\mathbb{F}_{2^{128}}$, so Auth256 is only slightly slower than an analogous Auth128 would be. An important advantage of an increased output size is that one can safely eliminate nonces.

Encrypting a hash with a one-time pad, or with a stream cipher such as AES in counter mode, requires a nonce, and becomes insecure if the user accidentally repeats a nonce; see, e.g., [30]. Directly applying a PRF (as in HMAC) or PRP (as in WMAC) to the hash, without using a nonce, is much more resilient against misuse but becomes insecure if hashes collide, so b-bit hashes are expected to be broken within $2^{b/2}$ messages (even with an optimal $\delta = 2^{-b}$) and already provide a noticeable attack probability within somewhat fewer messages.

This problem has motivated some research into "beyond-birthday-bound" mechanisms for authentication and encryption that can safely be used for more than $2^{b/2}$ messages. See, e.g., [38]. Hash256 takes a different approach, which we call "bigger-birthday-bound" security: simply increasing b to 256 (and correspondingly reducing δ) eliminates all risk of collisions. For the same reason, Hash256 provides extra strength inside other universal-hash applications, such as wide-block disk encryption; see, e.g., [28].

In applications with space for only 128-bit authenticators, it is safe to simply truncate the Hash256 and Auth256 output from 256 bits to 128 bits. This increases δ from 2^{-255} to 2^{-127}.

4.2 Pseudo Dot Products and FFT Addition. Hash256 uses the same basic construction as NMH [29, Section 5], UMAC [15], Badger [16], and VMAC [36]: the hash of a message with blocks $m_1, m_2, m_3, m_4, \ldots$ is $(m_1 + r_1)(m_2 + r_2) + (m_3 + r_3)(m_4 + r_4) + \cdots$. Halevi and Krawczyk [29] credit this hash to Carter and Wegman; Bernstein [6] credits it to Winograd and calls it the "pseudo dot product". The pseudo-dot-product construction of Hash256 gives $\delta < 2^{-255}$; see Appendix A for the proof.

A simple dot product $m_1r_1 + m_2r_2 + m_3r_3 + m_4r_4 + \cdots$ uses one multiplication per block. The same is true for GHASH and many other polynomial-evaluation

hashes. The basic advantage of $(m_1 + r_1)(m_2 + r_2) + (m_3 + r_3)(m_4 + r_4) + \cdots$ is that there are only 0.5 multiplications per block.

For Auth256 each block contains 256 bits, viewed as an element of the finite field $\mathbb{F}_{2^{256}}$. Our cost of Auth256 per 512 authenticated bits is $29 \cdot 512 = 58 \cdot 256$ bit operations, while the our cost for a multiplication in $\mathbb{F}_{2^{256}}$ is $87 \cdot 256$ bit operations. We now explain one of the two sources of this gap.

FFT-based multiplication of two polynomials $f_1 f_2$ has several steps: apply an FFT to evaluate f_1 at many points; apply an FFT to evaluate f_2 at many points; compute the corresponding values of the product $f_1 f_2$ by pointwise multiplication; and apply an inverse FFT to reconstruct the coefficients of $f_1 f_2$. FFT-based multiplication of field elements has the same steps plus a final reduction step.

These steps for $\mathbb{F}_{2^{256}}$, with our optimizations from Section 3, cost 4068 bit operations for each forward FFT, $64 \cdot 110$ bit operations for pointwise multiplication, 5996 bit operations for the inverse FFT (the forward FFT is less expensive since more polynomial coefficients are known to be 0), and 992 bit operations for the final reduction. Applying these steps to each 512 bits of input would cost approximately 15.89 bit operations per bit for the two forward FFTs, 13.75 bit operations per bit for pointwise multiplication, 11.71 bit operations per bit for the inverse FFT, and 1.94 bit operations per bit for the final reduction, plus 1.5 bit operations per bit for the three additions in the pseudo dot product.

We do better by exploiting the structure of the pseudo dot product as a sum of the form $f_1 f_2 + f_3 f_4 + f_5 f_6 + \cdots$. Optimizing this computation is not the same problem as optimizing the computation of $f_1 f_2$. Specifically, we apply an FFT to each f_i and compute the corresponding values of $f_1 f_2$, $f_3 f_4$, etc., but we then add these values *before* applying an inverse FFT. See Figure 1. There is now only one inverse FFT (and one final reduction) per message, rather than one inverse FFT for every two blocks. Our costs are now 15.89 bit operations per bit for the two forward FFTs, 13.75 bit operations per bit for pointwise multiplication, 1 bit operation per bit for the input additions in the pseudo dot product, and 1 bit operation per bit for the pointwise additions, for a total of 31.64 bit operations per bit, plus a constant (not very large) overhead per message.

This idea is well known in the FFT literature (see, e.g., [7, Section 2]) but we have never seen it applied to message authentication. It reduces the cost of FFT-based message authentication by a factor of nearly 1.5. Note that this also reduces the cutoff between FFT and Karatsuba.

UMAC and VMAC actually limit the lengths of their pseudo dot products, to limit key size. This means that longer messages produce two or more hashes; these hashes are then further hashed in a different way (which takes more time per byte but is applied to far fewer bytes). For simplicity we instead use a key as long as the maximum message length. We have also considered the small-key hashes from [6] but those hashes obtain less benefit from merging inverse FFTs.

4.3 Embedding Invertible Linear Operations into FFT Inputs. Section 4.2 shows how to achieve 31.64 bit operations per message bit by skipping the inverse FFTs for almost all multiplications in the pseudo dot product. Now

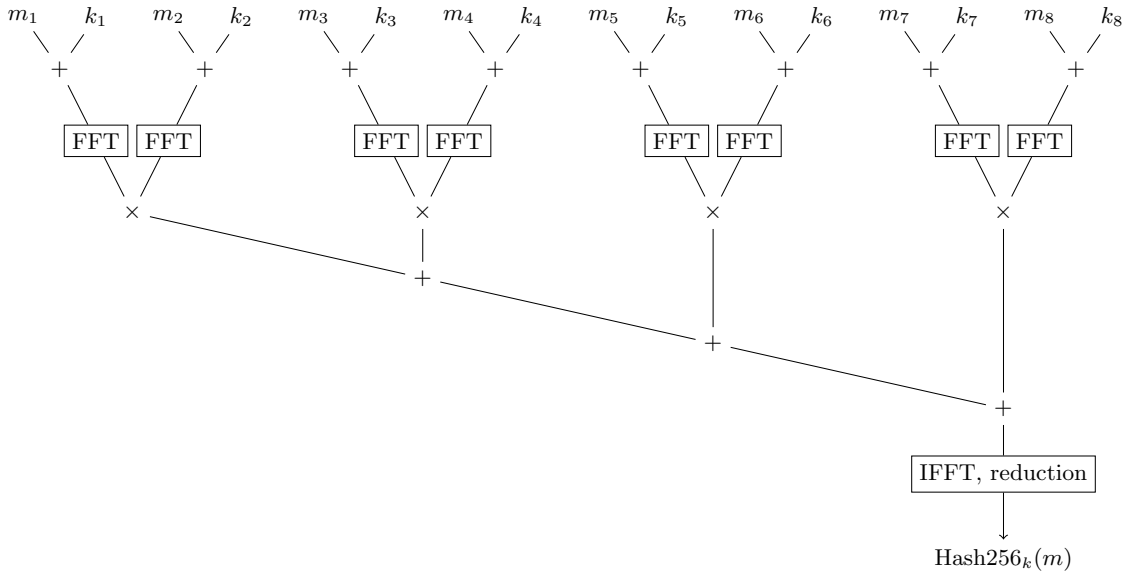

Fig. 1. Hash256 flowchart

we show how Auth256 achieves 29 bit operations per message bit by skipping operations in the forward FFTs.

Section 3.3 shows that the radix conversions can be merged into one invertible $\mathbb{F}_{2^8}$-linear (actually $\mathbb{F}_2$-linear) map, which takes place before all other operations in the FFT. The input is a $\mathbb{F}_{2^{256}}$ element which is represented as coefficients in $\mathbb{F}_{2^8}$ with respect to a polynomial basis. Applying an invertible linear map on the coefficients implies a change of basis. In other words, the radix conversions convert the input into another 256-bit representation. If we define the input to be elements in this new representation, all the radix conversions can simply be skipped. Note that the authenticator still uses the original representation. See Section 6.2 for a definition of the new representation.

This technique saves a significant fraction of the operations in the forward FFT. As shown in Section 3, one forward FFT takes 4068 bit operations, where $82 \cdot 8 = 656$ of them are spent on radix conversions. Eliminating all radix conversions then gives the 29 bit operations per message bit.

The additive FFTs described so far are "2-way split" FFTs since they require writing the input polynomial $f(x)$ in the form $f^{(0)}(x^2 + x) + xf^{(1)}(x^2 + x)$. It is easy to generalize this to a "2^k-way split" in which $f(x)$ is written as $\sum_{i=0}^{2^k-1} x^i f^{(i)}(\psi^k(x))$, where $\psi(x) = x^2+x$. In particular, Gao and Mateer showed how to perform $2^{2^{k-1}}$-way-split FFTs for polynomials in $\mathbb{F}_{2^{2^k}}[x]$. The technique of changing input representation works for any 2^k-way split. In fact we found that with 8-way-split FFTs, the number of bit operations per message bit can be slightly better than 29. However, for simplicity, Auth256 is defined in a way that favors 2-way-split FFTs.

5 Software Implementation

Our software implementation uses bitslicing. This means that we convert each bit in previous sections into w bits, where w is the register width on the machine; we convert each AND into a bitwise w-bit AND instruction; and we convert each XOR into a bitwise w-bit XOR instruction.

Bitslicing is efficient only if there is adequate parallelism in the algorithm. Fortunately, the pseudo-dot-product computation is naturally parallelizable: we let the jth bit position compute the sum of all products $(m_{2i+1}+r_{2i+1})(m_{2i+2}+r_{2i+2})$ where $i \equiv j \pmod{w}$. After all the products are processed, the results in all bit positions are summed up to get the final value.

The detailed definition of Auth256 (see Section 6) has a parameter b. Our software takes $b = w$, allowing it to simply pick up message blocks as vectors. If b is instead chosen as 1 then converting to bitsliced form requires a transposition of message blocks; in our software this transposition costs an extra 0.81 cycles/byte.

5.1 Minimizing Memory Operations in Radix Conversions. We exploit the decomposability of additions to minimize memory operations for a radix conversion. When dealing with size-2^k radix conversions with $k \leq 4$, we decompose at bit level the computation into 2^k parts, each of which deals with $16/2^k$ bit positions. This minimizes the number of loads and stores. The same technique applies for a radix conversion combined with smaller-size radix conversions in the FFT subroutines.

Our implementation uses the size-16 FFT as a subroutine. Inside a size-16 FFT the size-8 radix conversion is combined with the 2 size-4 radix conversions in FFT subroutines. Our bit-operation counts handle larger radix conversions in the same way, but in our software we sacrifice some of the bit operations saved here to improve instruction-level parallelism and register utilization. For size-16 radix conversion the decomposition method is adopted. For size-32 radix conversion the decomposition method is used only for the size-16 recursive calls.

5.2 Minimizing Memory Operations in Muladdadd Operations. For a single muladdadd operation $a \leftarrow a + \alpha b; b \leftarrow b + a$, each of a and b consumes 8 vectors; evidently at least 16 loads and 16 stores are required. While we showed how the sequence of bit operations can be generated, it does not necessarily mean that there are enough registers to carry out the bit operations using the minimum number of loads and stores.

Here is one strategy to maintain both the number of bit operations and the lower bound on number of loads and stores. First load the 8 bits of b into 8 registers $t_0, t_1, \ldots, t_7$. Use the sequence of XORs generated by the code generator, starting from the t_i's, to compute the 8 bits of αb, placing them in the other 8 registers $s_0, s_1, \ldots, s_7$. Then perform $s_i \leftarrow s_i \oplus a[i]$, where $a[i]$ is the corresponding bit of a in memory, to obtain $a + \alpha b$. After that overwrite a with the s_i's. Finally, perform $t_i \leftarrow t_i \oplus s_i$ to obtain $a + (\alpha + 1)b$, and overwrite b with the t_i's.

In our software muladdadd operations are handled one by one in size-64 and size-32 combine routines. See below for details about how muladdadds in smaller size combine routines are handled.

5.3 Implementing the Size-16 Additive FFT. In our size-16 FFT implementation the size-8 radix conversion is combined with the two size-4 ones in the FFT subroutines using the decomposition method described earlier in this section. Since the size-4 FFTs deal with constants in $\mathbb{F}_{2^2}$, we further combine the radix conversions with size-4 FFTs.

At the beginning of one of the 4 rounds of the whole computation, the $2 \cdot 8 = 16$ bits of the input for size-8 radix conversion are loaded. Then the logic operations are carried out in registers, and eventually the result is stored in $2 \cdot 16 = 32$ bits of the output elements. The same routine is repeated 4 times to cover all the bit positions.

The size-16 and size-32 combine routines are also merged as shown in Section 3. The field-level decomposition is used together with a bit-level decompostion: in size-16 FFT all the constants are in $\mathbb{F}_{2^4}$, so it is possible to decompose any computation that works on field elements into a 2-round procedure and handle 4 bit positions in each round. In conclusion, the field-level decomposition turns the computation into 4 pieces, and the bit-level decompostion further decomposes each of these into 2 smaller pieces. In the end, we have an 8-round procedure.

At the beginning of one of the 8 rounds of the whole computation, the $4 \cdot 4 = 16$ bits of the outputs of the size-4 FFTs are loaded. Then the logic operations are carried out in registers, and eventually the result is stored in $4 \cdot 4 = 16$ bits of the output elements. The same routine is repeated 8 times to cover all the bit positions.

6 Auth256: Minor Details

To close we fill in, for completeness, the remaining details of Hash256 and Auth256.

6.1 Review of Wegman–Carter MACs. Wegman–Carter MACs work as follows. The authenticator of the nth message m is $H(r, m) \oplus s_n$. The key consists of independent uniform random $r, s_1, s_2, s_3, \ldots$; the pad is $s_1, s_2, s_3, \ldots$.

The hash function H is designed to be "δ-xor-universal", i.e., to have "differential probability at most δ". This means that, for every message m, every message $m' \neq m$, and every difference Δ, a uniform random r has $H(r, m) \oplus H(r, m') = \Delta$ with probability at most δ.

6.2 Field Representation. We represent an element of $\mathbb{F}_{2^s}$ as a sequence of s bits. If we construct $\mathbb{F}_{2^s}$ as $\mathbb{F}_{2^t}[x]/\phi$ then we recursively represent the element $c_0 + c_1 x + \cdots + c_{t/s-1} x^{t/s-1}$ as the concatenation of the representations of $c_0, c_1, \ldots, c_{t/s-1}$. At the bottom of the recursion, we represent an element of $\mathbb{F}_2$ as 1 bit in the usual way. See Sections 2 and 3.6 for the definition of ϕ for $\mathbb{F}_{2^2}$, $\mathbb{F}_{2^4}$, $\mathbb{F}_{2^8}$, and $\mathbb{F}_{2^{256}}$.

As mentioned in Section 4.3, we do not use the polynomial basis $1, x, \ldots, x^{31}$ for $\mathbb{F}_{2^{256}}$ inputs. Here we define the representation for them. Let $y_{(b_{k-1}b_{k-2}\cdots b_0)_2} = \prod_{i=0}^{k-1}(\psi^i(x))^{b_i}$, where $\psi(x)$ follows the definition in Section 4.3. Then each $\mathbb{F}_{2^{256}}$ input $c_0y_0 + c_1y_1 + \cdots + c_{31}y_{31}$ is defined as the concatenation of the representations of $c_0, c_1, \ldots, c_{31}$. One can verify that $y_0, y_1, \ldots, y_{31}$ is the desired basis by writing down the equation between $f(x)$ and $f^{(00000)}(x)$, $f^{(00001)}(x)$, $\ldots$, $f^{(111111)}(x)$ as in Section 3.3.

If $s \geq 8$ then we also represent an element of $\mathbb{F}_{2^s}$ as a sequence of $s/8$ bytes, i.e., $s/8$ elements of $\{0, 1, \ldots, 255\}$. The 8-bit sequence $b_0, b_1, \ldots, b_7$ is represented as the byte $b = \sum_i 2^i b_i$.

6.3 Hash256 Padding and Conversion. Hash256 views messages as elements of $K^0 \cup K^2 \cup K^4 \cup \cdots$, i.e., even-length strings of elements of K, where K is the finite field $\mathbb{F}_{2^{256}}$. It is safe to use a single key with messages of different lengths.

In real applications, messages are strings of bytes, so strings of bytes need to be encoded invertibly as strings of elements of K. The simplest encoding is standard "10*" padding, where a message is padded with a 1 byte and then as many 0 bytes as necessary to obtain a multiple of 64 bytes. Each 32-byte block is then viewed as an element of K.

We define a more general encoding parametrized by a positive integer b; the encoding of the previous paragraph has $b = 1$. The message is padded with a 1 byte and then as many 0 bytes as necessary to obtain a multiple of $64b$ bytes, say $64bN$ bytes. These bytes are split into $2N$ segments $M_0, M'_0, M_1, M'_1, \ldots, M_{N-1}, M'_{N-1}$, where each segment contains $32b$ consecutive bytes. Each segment is then transposed into b elements of K: segment M_i is viewed as a column-major bit matrix with b rows and 256 columns, and row j of this matrix is labeled c_{bi+j}, while c'_{bi+j} is defined similarly using M'_i. This produces $2bN$ elements of K, namely $m_0, m'_0, m_1, m'_1, m_2, m'_2, \ldots, m_{bN-1}, m'_{bN-1}$.

The point of this encoding is to allow a simple bitsliced vectorized implementation; see Section 5. Our 1.59 cycle/byte implementation uses $b = 256$. We have also implemented $b = 1$, which costs 0.81 cycles/byte extra for transposition and is compatible with efficient handling of much shorter messages. An interesting intermediate possibility is to take, e.g., $b = 8$, eliminating the most expensive (non-bytewise) transposition steps while still remaining suitable for authentication of typical network packets.

6.4 Hash256 and Auth256 Keys and Authenticators. The Hash256 key is a uniform random byte string of the same length as a maximum-length padded message, representing elements $r_0, r'_0, r_1, r'_1, \ldots$ of K. If the key is actually defined as, e.g., counter-mode AES output then the maximum length does not need to be specified in advance: extra key elements can be computed on demand and cached for subsequent use.

The Hash256 output is $(m_0 + r_0)(m'_0 + r'_0) + (m_1 + r_1)(m'_1 + r'_1) + \cdots$. This is an element of K.

The Auth256 key is a Hash256 key together with independent uniform random elements $s_1, s_2, \ldots$ of K. The Auth256 authenticator of the nth message m_n is $\mathrm{Auth256}(r, m_n) \oplus s_n$.

References

[1] — (no editor), *Information theory workshop, 2006. ITW '06 Chengdu*, IEEE, 2006. See [45].

[2] — (no editor), *Design, automation & test in Europe conference & exhibition, 2007. DATE '07*, IEEE, 2007. See [41].

[3] — (no editor), *Proceedings of the 6th WSEAS world congress: applied computing conference (ACC 2013)*, WSEAS, 2013. See [21].

[4] Daniel J. Bernstein, *Fast multiplication* (2000). URL: http://cr.yp.to/talks.html#2000.08.14. Citations in this document: §1.3.

[5] Daniel J. Bernstein, *The Poly1305-AES message-authentication code*, in FSE 2005 [**25**] (2005), 32–49. URL: http://cr.yp.to/papers.html#poly1305. Citations in this document: §1.1.

[6] Daniel J. Bernstein, *Polynomial evaluation and message authentication* (2007). URL: http://cr.yp.to/papers.html#pema. Citations in this document: §4.2, §4.2.

[7] Daniel J. Bernstein, *Fast multiplication and its applications*, in [**17**] (2008), 325–384. URL: http://cr.yp.to/papers.html#multapps. Citations in this document: §4.2.

[8] Daniel J. Bernstein, *Batch binary Edwards*, in Crypto 2009 [**27**] (2009), 317–336. URL: http://cr.yp.to/papers.html#bbe. Citations in this document: §1.3.

[9] Daniel J. Bernstein, *Minimum number of bit operations for multiplication* (2009). URL: http://binary.cr.yp.to/m.html. Citations in this document: §1.3, §1.3, §2.2, §1, §1.

[10] Daniel J. Bernstein, *Optimizing linear maps modulo 2*, in Workshop Record of SPEED-CC: Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers (2009), 3–18. URL: http://cr.yp.to/papers.html#linearmod2. Citations in this document: §2.3.

[11] Daniel J. Bernstein, Tung Chou, Peter Schwabe, *McBits: fast constant-time code-based cryptography*, in CHES 2013 [**12**] (2013), 250–272. Citations in this document: §1.3, §1.3, §3.

[12] Guido Bertoni, Jean-Sébastien Coron (editors), *Cryptographic hardware and embedded systems—CHES 2013—15th international workshop, Santa Barbara, CA, USA, August 20–23, 2013, proceedings*, Lecture Notes in Computer Science, 8086, Springer, 2013. ISBN 978-3-642-40348-4. See [11].

[13] Eli Biham (editor), *Fast software encryption '97*, Lecture Notes in Computer Science, 1267, Springer, 1997. ISBN 3-540-63247-6. See [29].

[14] Eli Biham, Amr M. Youssef (editors), *Selected areas in cryptography, 13th international workshop, SAC 2006, Montreal, Canada, August 17–18, 2006, revised selected papers*, Lecture Notes in Computer Science, 4356, Springer, 2007. ISBN 978–3–540–74461–0. See [36].

[15] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, Phillip Rogaway, *UMAC: fast and secure message authentication*, in Crypto 1999 [**49**] (1999), 216–233. URL: http://www.cs.ucdavis.edu/~rogaway/umac/. Citations in this document: §1.1, §4.2.

[16] Martin Boesgaard, Ove Scavenius, Thomas Pedersen, Thomas Christensen, Erik Zenner, *Badger—a fast and provably secure MAC*, in [**32**] (2005), 176–191. Citations in this document: §4.2.

[17] Joe P. Buhler, Peter Stevenhagen (editors), *Surveys in algorithmic number theory*, Mathematical Sciences Research Institute Publications, 44, Cambridge University Press, New York, 2008. See [7].

[18] Nam Su Chang, Chang Han Kim, Young-Ho Park, Jongin Lim, *A non-redundant and efficient architecture for Karatsuba-Ofman algorithm*, in [**50**] (2005), 288–299. Citations in this document: §1.3.

[19] Circuit Minimization Team, *Multiplication circuit for GF(256) with irreducible polynomial* $X^8 + X^4 + X^3 + X + 1$ (2010). URL: http://cs-www.cs.yale.edu/homes/peralta/CircuitStuff/slp_84310.txt. Citations in this document: §2.2.

[20] Christophe Clavier, Kris Gaj (editors), *Cryptographic hardware and embedded systems—CHES 2009, 11th international workshop, Lausanne, Switzerland, September 6–9, 2009, proceedings*, Lecture Notes in Computer Science, 5747, Springer, 2009. ISBN 978-3-642-04137-2. See [35].

[21] Davide D'Angella, Chiara Valentina Schiavo, Andrea Visconti, *Tight upper bounds for polynomial multiplication*, in [**3**] (2013). URL: http://www.wseas.us/e-library/conferences/2013/Nanjing/ACCIS/ACCIS-03.pdf. Citations in this document: §1.3, §1, §1.

[22] Andrew M. "Floodyberry", *Optimized implementations of Poly1305, a fast message-authentication-code* (2014). URL: https://github.com/floodyberry/poly1305-opt. Citations in this document: §1.1.

[23] Agner Fog, *Instruction tables* (2014). URL: http://www.agner.org/optimize/instruction_tables.pdf. Citations in this document: §1.4.

[24] Shuhong Gao, Todd Mateer, *Additive fast Fourier transforms over finite fields*, IEEE Transactions on Information Theory **56** (2010), 6265–6272. URL: http://www.math.clemson.edu/~sgao/pub.html. Citations in this document: §3.

[25] Henri Gilbert, Helena Handschuh (editors), *Fast software encryption: 12th international workshop, FSE 2005, Paris, France, February 21–23, 2005, revised selected papers*, Lecture Notes in Computer Science, 3557, Springer, 2005. ISBN 3-540-26541-4. See [5].

[26] Shay Gueron, *AES-GCM software performance on the current high end CPUs as a performance baseline for CAESAR* (2013). URL: http://2013.diac.cr.yp.to/slides/gueron.pdf. Citations in this document: §1.4.

[27] Shai Halevi (editor), *Advances in cryptology—CRYPTO 2009, 29th annual international cryptology conference, Santa Barbara, CA, USA, August 16–20, 2009, proceedings*, Lecture Notes in Computer Science, 5677, Springer, 2009. See [8].

[28] Shai Halevi, *Invertible universal hashing and the TET encryption mode*, in Crypto 2007 [**39**] (2007), 412–429. URL: http://eprint.iacr.org/2007/014. Citations in this document: §4.1.

[29] Shai Halevi, Hugo Krawczyk, *MMH: software message authentication in the Gbit/second rates*, in FSE 1997 [**13**] (1997), 172–189. URL: http://www.research.ibm.com/people/s/shaih/pubs/mmh.html. Citations in this document: §4.2, §4.2.

[30] Helena Handschuh, Bart Preneel, *Key-recovery attacks on universal hash function based MAC algorithms*, in [**46**] (2008), 144–161. Citations in this document: §4.1.

[31] José Luis Imaña, Román Hermida, Francisco Tirado, *Low-complexity bit-parallel multipliers based on a class of irreducible pentanomials*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems **14** (2006), 1388–1393. Citations in this document: §2.2.

[32] John Ioannidis, Angelos D. Keromytis, Moti Yung, *Applied cryptography and network security, third international conference, ACNS 2005, New York, NY, USA, June 7-10, 2005, proceedings*, Lecture Notes in Computer Science, 3531, 2005. ISBN 3-540-26223-7. See [16].

[33] Tetsu Iwata, Keisuke Ohashi, Kazuhiko Minematsu, *Breaking and repairing GCM security proofs*, in Crypto 2012 [**44**] (2012), 31–49. URL: http://eprint.iacr.org/2012/438. Citations in this document: §4.

[34] Antoine Joux (editor), *Fast software encryption—18th international workshop, FSE 2011, Lyngby, Denmark, February 13–16, 2011, revised selected papers*, Lecture Notes in Computer Science, 6733, Springer, 2011. ISBN 978-3-642-21701-2. See [37].

[35] Emilia Käsper, Peter Schwabe, *Faster and timing-attack resistant AES-GCM*, in CHES 2009 [**20**] (2009), 1–17. URL: http://eprint.iacr.org/2009/129. Citations in this document: §1, §1.1, §1.4.

[36] Ted Krovetz, *Message authentication on 64-bit architectures*, in SAC 2006 [**14**] (2007), 327–341. Citations in this document: §1.1, §4.2.

[37] Ted Krovetz, Philip Rogaway, *The software performance of authenticated-encryption modes*, in FSE 2011 [**34**] (2011), 306–327. URL: http://www.cs.ucdavis.edu/~rogaway/papers/ae.pdf. Citations in this document: §1.4.

[38] Will Landecker, Thomas Shrimpton, R. Seth Terashima, *Tweakable blockciphers with beyond birthday-bound security*, in Crypto 2012 [**44**] (2012), 14–30. Citations in this document: §4.1.

[39] Alfred Menezes (editor), *Advances in cryptology—CRYPTO 2007, 27th annual international cryptology conference, Santa Barbara, CA, USA, August 19–23, 2007, proceedings*, Lecture Notes in Computer Science, 4622, Springer, 2007. ISBN 978-3-540-74142-8. See [28].

[40] Christof Paar, *Optimized arithmetic for Reed–Solomon encoders* (1997). URL: http://www.emsec.rub.de/media/crypto/veroeffentlichungen/2011/01/19/cnst.ps. Citations in this document: §2.3.

[41] Steffen Peter, Peter Langendörfer, *An efficient polynomial multiplier in GF(2^m) and its application to ECC designs*, in DATE 2007 [**2**] (2007). URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?isnumber=4211749&arnumber=4211979&count=305&index=229. Citations in this document: §1.3.

[42] Francisco Rodríguez-Henríquez, Çetin Kaya Koç, *On fully parallel Karatsuba multipliers for GF(2^m)*, in [**43**] (2003), 405–410. Citations in this document: §1.3.

[43] S. Sahni (editor), *Proceedings of the international conference on computer science and technology*, Acta Press, 2003. See [42].

[44] Reihaneh Safavi-Naini, Ran Canetti (editors), *Advances in cryptology—CRYPTO 2012—32nd annual cryptology conference, Santa Barbara, CA, USA, August 19–23, 2012, proceedings*, Lecture Notes in Computer Science, 7417, Springer, 2012. ISBN 978-3-642-32008-8. See [33], [38].

[45] Joachim von zur Gathen, Jamshid Shokrollahi, *Fast arithmetic for polynomials over $\mathbf{F}_2$ in hardware*, in ITW 2006 [**1**] (2006), 107–111. Citations in this document: §1.3.

[46] David Wagner (editor), *Advances in cryptology—CRYPTO 2008, 28th annual international cryptology conference, Santa Barbara, CA, USA, August 17–21, 2008, proceedings*, Lecture Notes in Computer Science, 5157, Springer, 2008. ISBN 978-3-540-85173-8. See [30].

[47] Mark N. Wegman, J. Lawrence Carter, *New hash functions and their use in authentication and set equality*, Journal of Computer and System Sciences **22** (1981), 265–279. ISSN 0022-0000. MR 82i:68017. Citations in this document: §4.

[48] André Weimerskirch, Christof Paar, *Generalizations of the Karatsuba algorithm for efficient implementations* (2006). URL: http://eprint.iacr.org/2006/224. Citations in this document: §1.3.

[49] Michael Wiener (editor), *Advances in cryptology—CRYPTO '99*, Lecture Notes in Computer Science, 1666, Springer, 1999. ISBN 3-5540-66347-9. MR 2000h:94003. See [15].

[50] Jianying Zhou, Javier Lopez, Robert H. Deng, Feng Bao (editors), *Information security, 8th international conference, ISC 2005, Singapore, September 20–23, 2005, proceedings*, Lecture Notes in Computer Science, 3650, Springer, 2005. ISBN 3-540-29001-X. See [18].

A Security Proof

This appendix proves that Hash256 has differential probability smaller than 2^{-255}. This is not exactly the same as the proofs for the pseudo-dot-product portions of UMAC and VMAC: UMAC and VMAC specify fixed lengths for their pseudo dot products, whereas we allow variable lengths.

Theorem 1. *Let K be a finite field. Let ℓ, ℓ', k be nonnegative integers with $\ell \le k$ and $\ell' \le k$. Let $m_1, m_2, \ldots, m_{2\ell-1}, m_{2\ell}$ be elements of K. Let $m'_1, m'_2, \ldots, m'_{2\ell'-1}$, $m'_{2\ell'}$ be elements of K. Assume that $(m_1, m_2, \ldots, m_{2\ell}) \neq (m'_1, m'_2, \ldots, m'_{2\ell'})$. Let Δ be an element of K. Let $r_1, r_2, \ldots, r_{2k}$ be independent uniform random elements of k. Let p be the probability that $h = h' + \Delta$, where*

$$\begin{aligned} h &= (m_1 + r_1)(m_2 + r_2) + (m_3 + r_3)(m_4 + r_4) + \cdots \\ &\quad + (m_{2\ell-1} + r_{2\ell-1})(m_{2\ell} + r_{2\ell}), \\ h' &= (m'_1 + r_1)(m'_2 + r_2) + (m'_3 + r_3)(m'_4 + r_4) + \cdots \\ &\quad + (m'_{2\ell'-1} + r_{2\ell'-1})(m'_{2\ell'} + r_{2\ell'}). \end{aligned}$$

Then $p < 2/\#K$. If $\ell = \ell'$ then $p \le 1/\#K$, and if $\ell \neq \ell'$ then $p < 1/\#K + 1/\#K^{|\ell-\ell'|}$.

Proof. Case 1: $\ell = \ell'$. Then $h = h' + \Delta$ if and only if

$$\begin{aligned} &r_1(m_2 - m'_2) + r_2(m_1 - m'_1) + r_3(m_4 - m'_4) + r_4(m_3 - m'_3) + \cdots \\ &\quad = \Delta + m'_1 m'_2 - m_1 m_2 + m'_3 m'_4 - m_3 m_4 + \cdots . \end{aligned}$$

This is a linear equation in $r_1, r_2, \ldots, r_{2k}$. This linear equation is nontrivial: by hypothesis $(m_1, m_2, \ldots, m_{2\ell}) \neq (m'_1, m'_2, \ldots, m'_{2\ell'})$, so there must be some i for which $m_i - m'_i \neq 0$. Consequently there are most $\#K^{2k-1}$ solutions to the equation out of the $\#K^{2k}$ possibilities for r; i.e., $p \le 1/\#K$ as claimed.

Case 2: $\ell < \ell'$ and $(m_1, \ldots, m_\ell) \neq (m'_1, \ldots, m'_\ell)$. Define

$$f = (m'_{2\ell+1} + r_{2\ell+1})(m_{2\ell+2} + r_{2\ell+2}) + \cdots + (m'_{2\ell'-1} + r_{2\ell'-1})(m'_{2\ell'} + r_{2\ell'}).$$

Then $h = h' + \Delta$ if and only if

$$\begin{aligned} &r_1(m_2 - m'_2) + r_2(m_1 - m'_1) + r_3(m_4 - m'_4) + r_4(m_3 - m'_3) + \cdots \\ &\qquad + r_{2\ell-1}(m_{2\ell} - m'_{2\ell}) + r_{2\ell}(m_{2\ell-1} - m'_{2\ell-1}) \\ &\quad = f + \Delta + m'_1 m'_2 - m_1 m_2 + m'_3 m'_4 - m_3 m_4 + \cdots + m'_{2\ell-1} m'_{2\ell} - m_{2\ell-1} m_{2\ell}. \end{aligned}$$

This is a linear equation in $r_1, \ldots, r_{2\ell}$, since f is independent of $r_1, \ldots, r_{2\ell}$. For each choice of $(r_{2\ell+1}, r_{2\ell+2}, \ldots, r_{2k})$, there are at most $\#K^{2\ell-1}$ choices of $(r_1, \ldots, r_{2\ell})$ satisfying this linear equation. Consequently $p \leq 1/\#K$ as above.

Case 3: $\ell < \ell'$ and $(m_1, \ldots, m_\ell) = (m'_1, \ldots, m'_\ell)$. Then $h = h' + \Delta$ if and only if $0 = f + \Delta$, where f is defined as above. This is a linear equation in $r_{2\ell+2}, r_{2\ell+4}, \ldots, r_{2\ell'}$ for each choice of $r_{2\ell+1}, r_{2\ell+3}, \ldots, r_{2\ell'-1}$. The linear equation is nontrivial except when $r_{2\ell+1} = -m'_{2\ell+1}$, $r_{2\ell+3} = -m'_{2\ell+3}$, and so on through $r_{2\ell'-1} = -m'_{2\ell'-1}$. The linear equation thus has at most $\#K^{\ell'-\ell-1}$ solutions $(r_{2\ell+2}, r_{2\ell+4}, \ldots, r_{2\ell'})$ for $\#K^{\ell'-\ell} - 1$ choices of $(r_{2\ell+1}, r_{2\ell+3}, \ldots, r_{2\ell'-1})$, plus at most $\#K^{\ell'-\ell}$ solutions $(r_{2\ell+2}, r_{2\ell+4}, \ldots, r_{2\ell'})$ for 1 exceptional choice of $(r_{2\ell+1}, r_{2\ell+3}, \ldots, r_{2\ell'-1})$, for a total of $\#K^{2\ell'-2\ell-1} - \#K^{\ell'-\ell-1} + \#K^{\ell'-\ell} < \#K^{2\ell'-2\ell}(1/\#K + 1/\#K^{\ell'-\ell})$ solutions. Consequently $p < 1/\#K + 1/\#K^{\ell'-\ell}$ as claimed.

Case 4: $\ell' < \ell$. Exchanging ℓ, m with ℓ', m' produces Case 2 or Case 3. □