cryptography

MDPI

Article

The Definition and Software Performance of Hashstream, a Fast Length-Flexible PRF

Ted Krovetz

Computer Science Department, California State University, Sacramento, CA 95819, USA; tdk@csus.edu

Received: 13 September 2018; Accepted: 11 October 2018; Published: 15 October 2018

Abstract: Two of the fastest types of cryptographic algorithms are the stream cipher and the almost-universal hash function. There are secure examples of each that process data in software using less than one CPU cycle per byte. Hashstream combines the two types of algorithms in a straightforward manner yielding a PRF that can both consume inputs of and produce pseudorandom outputs of any desired length. The result is an object useful in many contexts: authentication, encryption, authenticated encryption, random generation, mask generation, etc. The HS1-SIV authenticated-encryption algorithm—a CAESAR competition second round selection—was based on Hashstream and showed the promise of such an approach by having provable security and topping the speed charts in several test configurations.

Keywords: pseudorandom function; length-flexible; high-speed; authenticated encryption; SIV; CAESAR

1. Introduction

The goal of this work is to introduce an easy to use, hard to misuse, provably secure, cryptographic pseudorandom function (PRF) useful in many contexts. Hashstream marries a well-known, extremely fast universal hash function (e.g., GHASH or Poly1305 [1,2]) with a well-known, extremely fast stream cipher (e.g., AES-CTR or Chacha [3,4]) into a length-flexible keyed pseudorandom function, which enjoys high security with or without the use of a nonce. The programming interface is simple: `hs_ctx_init(c,k)` takes a context structure, and a key and initializes the context; `hs_hash(c,s)` takes a context and an input string and stores the hash of the string in the context; and `hs_stream(c,n,l)` takes a context, nonce and output length and produces the requested number of pseudorandom bytes. As long as the hash result in the context and the nonce supplied to `hs_stream` have never been paired before, the pseudorandom output will be new. Thus, all a user needs to do to produce new pseudorandom outputs is update the nonce used with each application or, if that is not possible, limit the number of Hashstream applications so that it is unlikely that two intermediate hash values collide (the probability of which is governed by a birthday bound of the length of the internal hashes).

Hashstream has many attractive features. (i) Hashstream can be used in straightforward ways to achieve many goals: encryption, authentication, authenticated encryption, random generation, key derivation, mask generation, etc.; (ii) Hashstream is provably secure: because universal hash functions have provable combinatoric properties, Hashstream security depends solely on reasonable assumptions made on the stream cipher; (iii) Hashstream is misuse-resistant: nonces do not have to change between calls; keys can be any length; the programming interface specifies sensible defaults for usages that might easily cause trouble if not guarded against; (iv) Hashstream is fast in software: a version of Hashstream running on Intel's Skylake architecture consumes data at around 0.4 CPU cycles-per-byte (cpb) and produces pseudorandom bytes at around 0.6 cpb; together, this can yield authenticated encryption at around 1.0 cpb; (v) Hashstream is a simple abstraction: the programming

interface has a small number of functions, each taking a small number of parameters and performing a conceptually simple task.

It should be noted that this paper focuses on defining Hashstream constructions and their software performance. Security rationales are summarized, but formal proofs are omitted.

1.1. Notation

Throughout this paper, strings are considered sequences of bits with the first bit having Index 0. A substring is represented by $S[i, \ell]$, meaning the length ℓ substring beginning at index i. The length of string S is $|S|$ bits. Strings S and T are exclusive-or'd, indicated as $S \oplus T$, by first appending zeros to the end of the shorter string until they are the same length. When interpreting a non-negative integer as a string, $\langle i \rangle_\ell$ is the ℓ-bit big-endian binary representation of i. The symbol $\mathrm{Adv}^{\mathrm{prf}}_{\mathrm{abc}}$ is informally used to indicate the maximum advantage an adversary could achieve in distinguishing a randomly keyed algorithm "abc" from a random function with the same function signature when allowed q oracle queries and time t.

1.2. Hashstream Uses

Hashstream is a flexible tool. This paper is not focused on applications of Hashstream, but here we give some simple examples of its use. For illustrative purposes, let us say that $H(x, n, \ell)$ takes as input an arbitrary string x, a nonce n and an output length ℓ, and that for each distinct (x, n) pair, ℓ bytes of random output are produced by H. In other words, H behaves like Hashstream after it has been initialized with a random key.

Encryption: Message m is encrypted as $H(\varepsilon, n, |m|) \oplus m$, where ε is the empty string, n is a nonce that must be different for each encryption and $|m|$ is the length of m. The nonce and the result of the xor are bundled into a ciphertext. If a nonce is ever repeated for two encryptions, then an observer of the two ciphertexts can easily determine the xor of the corresponding plaintexts, so this application of Hashstream requires nonces to be non-repeating. Because the input string is empty, Hashstream's speed is similar to that of the stream cipher being used.

Authentication: An authentication tag of length ℓ for message m can be generated as $H(m, n, \ell)$. Changing the nonce n for each message and communicating it with the authentication tag is optional, but a birthday bound limits the number of authentication tags one can generate if the nonce is held constant. This is perfectly acceptable for systems where large numbers of authentications are not possible, but for protection against attacks involving large numbers of authentications, the nonce should be updated for each tag generated. Hashstream speed in this use case is expected to be similar to that of a Wegman-Carter MAC.

Authenticated encryption: The two methods above can be combined. To encrypt m, first generate length ℓ tag $t = H(m, n, \ell)$, and then, encrypt m as $c = H(t, n + 1, |m|) \oplus m$. (An alternative to using $n + 1$ as the encryption nonce is to use n a second time, but to throw out the first ℓ of the output before encryption.) The resulting ciphertext is the bundle (n, t, c). If a nonce is repeated, the only information that leaks is whether the corresponding messages are identical (i.e., with high probability $t = t'$ if and only if $m = m'$). This means that in situations where messages cannot be repeated or leaking whether messages repeat is not damaging, nonces need not be used. Extending this scheme to include authenticated data a is straightforward: only change the tag definition to $t = H(\mathsf{encode}(a||m), n, \ell)$ where encoding is done with any injective mapping on strings. This method of authenticated encryption is essentially the SIV mode of Rogaway and Shrimpton and is used in Hashstream-based CAESAR candidate HS1-SIV [5,6]. Later in the paper, timings are given for Hashstream in this SIV mode.

Random generation: If an entropy store x needs to be translated into a length ℓ pseudorandom output, it can be done as $H(x, n, \ell)$. If a simulation needs pseudorandomness, it can be generated as $H(\varepsilon, n, \ell)$. Since Hashstream is a deterministic algorithm, experiments can be repeated by reusing nonce n or rerun using a different nonce.

1.3. Hashstream Constructions

This paper considers two ways of constructing Hashstream. They both join a universal hash with a stream cipher, but in the first instance, an algorithm designed to be a stream cipher is used, whereas in the second instance, a block cipher in counter mode is used as the stream cipher.

The original idea for a Hashstream construction—the one forming the basis of CAESAR submission HS1-SIV—is to use a universal hash function to hash the Hashstream input and exclusive-or the hash result with the key of the stream cipher. This requires that the stream cipher be both secure against related-key attacks and be key-agile. Any stream cipher with these properties that accepts a nonce is suitable for Hashstream. Let $f(k, n)$ be a stream cipher that takes a key and nonce as input, and let $h(k, x)$ be an almost-universal hash function that takes a key and an arbitrary string as input and whose output is no longer than f's key. Then Hashstream can very simply be defined as:

$$\mathsf{Hashstream}_{f,h}((k_1, k_2), n, x) = f(k_1 \oplus h(k_2, x), n).$$

The key-agility of f ensures very little overhead beyond the hash and stream cipher computations, and if multiple output streams associated with x are needed, rehashing x is not necessary: only the nonce needs updating for each stream.

This paper also explores a second construction for Hashstream, one using a tweakable block cipher in counter-mode as the stream cipher. Let $E : \{0,1\}^\kappa \times \{0,1\}^b \to \{0,1\}^b$ be a block cipher, and let $h(k, x)$ be an xor-almost-universal hash function that takes a key and an arbitrary string as inputs and whose output is b bits. Then, we can define $\mathsf{Hashstream}_{E,h}((k_1, k_2), n, x) = Y$, where Y is computed as:

$$\begin{aligned} K &= (E(k_1, n[0, b-8] \,||\, \langle 1 \rangle_8) \,||\, E(k_1, n[0, b-8] \,||\, \langle 2 \rangle_8) \,||\, \cdots)[0, \kappa] \\ Y &= (E(K, (n[b-8, 8] \,||\, \langle 0 \rangle_{b-8}) \oplus h(k_2, x)) \oplus h(k_2, x)) \,|| \\ &\quad\; (E(K, (n[b-8, 8] \,||\, \langle 1 \rangle_{b-8}) \oplus h(k_2, x)) \oplus h(k_2, x)) \,|| \,\cdots\, . \end{aligned}$$

Here, K is an intermediate block cipher key determined by all but the last eight bits of the nonce, and Y is generated using K with the tweakable block cipher of Liskov, Rivest and Wagner used in counter mode with the last eight bits of the nonce as the initialization vector [7]. This is essentially using Naito's XKX beyond-birthday bound tweakable block cipher construction in counter-mode with a performance enhancement [8]. The XKX construction rekeys the block cipher for every nonce. The construction used here zeroes the low-order eight bits of the nonce and incorporates them instead as the high-order 8 bits of the counter-mode initialization vector. This means that when the Hashstream nonce is incremented as a counter, a new intermediate key is needed only once every 2^8 calls to Hashstream. The result is a stream cipher with beyond-birthday security and an amortized cost of little more than one block cipher call per blocklength of output. For clarity, the construction is given visually in Figure 1.

1.4. Hashstream Speed

Ever since Krawczyk and Halevi's MMH in 1997, it has been evident that very high speeds and provable security are not exclusive goals in cryptography [9]. MMH was the first universal hash function to be able to process large input data in software at a rate of close to one CPU cycle per byte. Since that time several universal hashes including UHASH (1999), Poly1305 (2005) and VHASH (2006) have all reported processing rates of well under one cpb [10,11]. Using specialized assembly instructions included on Intel processors since 2010, GHASH also belongs to this group of highly-efficient universal hashes. The OpenSSL cryptographic library has high-quality assembly implementations of both Poly1305 and GHASH, and they report peak speeds for large data on the Intel Skylake architecture of 0.51 cpb for Poly1305 and 0.36 cpb for GHASH [12]. OpenSSL implementations of these are nearly as fast on CPUs found in smartphones: Poly1305 runs at 0.72 cpb, and GHASH runs at 0.58 cpb on the Apple A7 processor.

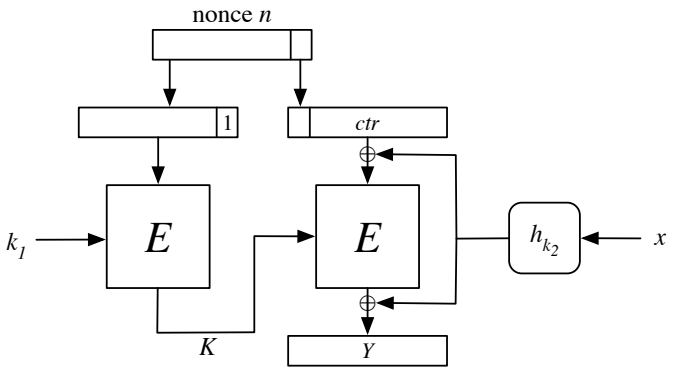

Figure 1. Hashstream instantiated with a block cipher *E*. The left *E* generates the key for the right *E* and is invoked whenever there is a change in the nonce outside of its last byte. The right side of the picture is a realization of a tweakable block cipher and should be repeated as many times as needed to produce as many output blocks as desired, each time with the counter incremented.

The production of cryptographic pseudorandom bits has long been more expensive than universal hashing, but the gap is narrowing significantly. Back when MMH was pushing the one cpb barrier for hashing, AES and RC4 were struggling to operate at 20 and 5 cpb, respectively. In the intervening years, CPU hardware has evolved in ways making cryptographic processing much faster. Intel processors now have vector registers eight-times as large and twice as plentiful as those available in 1997, and in 2010, assembly instructions accelerating AES by an order of magnitude were added to Intel's instruction set. Today, OpenSSL reports Skylake processing AES bytes at a rate of 0.63 cpb and the Chacha20 stream cipher producing output at a rate of 1.2 cpb.

These four algorithms—GHASH, Poly1305, AES and Chacha20—are core algorithms used frequently in TLS sessions, which means they are commonly found in cryptography libraries and highly-tuned for security and performance. This fact brings several benefits to Hashstream implementations. Programming Hashstream requires only "glue code" to assemble the constituent primitives together, reducing the chance of error. The Poly1305 and Chacha20 Hashstream implementation reported in this paper has its hash and stream code written in under 20 lines of C. The rest of the work is done by well-tested library code, and that library code can be fast. Table 1 shows sample speeds when using the OpenSSL cryptographic library to implement Hashstream.

To put these speeds into context, the Skylake section of the SUPERCOP benchmarking website lists over 240 authenticated-encryption algorithms [13]. Comparing the peak Hashstream SIV speeds in Table 1 against SUPERCOP benchmarks for a similar number of bytes processed, Hashstream with GHASH and AES would rank 15th and Hashstream with Poly1305 and Chacha20 would rank 29th. For the Cortex-A15, Hashstream with Poly1305 and Chacha20 would rank sixth in the SUPERCOP benchmark, while Hashstream with GHASH and AES would rank 39th. This is a significant result. These other algorithms are custom-designed authenticated-encryption algorithms, and the Hashstream SIV algorithm is not. Authenticated encryption is but one application of Hashstream. This shows that using a well-designed generalized tool like Hashstream does not necessarily require giving up much speed.

As can be seen in Table 1, when a processor has carryless multiplication and AES round instructions in the instruction set, as Intel Skylake does and the two ARM machines do not, the Hashstream version based on AES and GHASH is close to twice as fast as the version based on Poly1305 and Chacha20. When AES and GHASH are not accelerated in hardware, however, the advantage is reversed: the Poly1305 and Chacha20 version is close to three-times faster in this case. In an absolute sense, however, the two constructions are close in performance under Skylake, and the Poly1305 and Chacha20 version is much faster otherwise. If an application is known to run almost entirely on Intel CPUs and ARMv8 processors with cryptographic extensions, then it is a reasonable

choice to use the GHASH- and AES-based Hashstream, but if a wider variety of processors is expected, Poly1305 and Chacha20 make a better compromise choice.

Table 1. Hashstream throughput on Intel and ARM processors measured in CPU cycles per byte processed. The columns for the hash measure calls to `hs_hash`, which performs the universal hash and stores the result in the Hashstream context. The columns for the stream measure calls to `hs_stream`, which initializes the stream cipher and uses it to produce pseudorandom bytes. Hashing and streaming values can be added together to yield combined Hashstream throughput. The columns under SIV are for authenticated encryption using Hashstream in SIV mode. The Cortex-A5 is restricted to ARMv5 instructions to approximate a low-power 32-bit embedded processor.

	Hash			**Stream**			**SIV**		
Skylake	**64**	**256**	**1024**	**64**	**256**	**1024**	**64**	**256**	**1024**
Poly1305 + Chacha20	2.8	1.5	0.7	5.2	2.5	1.2	15.0	5.6	2.4
GHASH + AES-CTR	1.8	0.6	0.4	1.6	0.9	0.7	6.4	2.2	1.3
Cortex-A15	**64**	**256**	**1024**	**64**	**256**	**1024**	**64**	**256**	**1024**
Poly1305 + Chacha20	8.2	3.0	1.7	11.9	5.2	5.0	38.0	12.8	7.9
GHASH + AES-CTR	9.9	8.1	7.7	22.8	16.6	14.8	44.2	27.7	23.2
Cortex-A5 without NEON	**64**	**256**	**1024**	**64**	**256**	**1024**	**64**	**256**	**1024**
Poly1305 + Chacha20	16.4	8.9	7.0	21.4	20.0	19.4	75.1	38.4	28.8
GHASH + AES-CTR	52.3	43.7	41.5	49.2	46.7	46.1	141.2	100.3	90.2

1.5. Hashstream Security

Hashstream combines a universal hash function with a stream cipher. Because universal hash functions are combinatoric objects and have proven bounds, Hashstream security depends only on assumptions made about the steam cipher. In this paper, two stream ciphers are considered: a block cipher used in counter mode (with some modifications to achieve beyond-birthday security) and Chacha20.

Chacha stream cipher: The assumption made in this paper about Chacha is that it is a pseudorandom function mapping inputs $\{0,1\}^{256} \times \{0,1\}^{96}$ to strings of length 2^{38} bytes (Chacha's maximum output length). This is not the usual assumption made about stream ciphers, but several statements made by Bernstein support this view. In "Response to 'On the Salsa20 core function'", Bernstein claims that the Salsa "core" is designed "to eliminate all visible structure" [14]. In the Rumba compression function, adversaries are allowed to provide any chosen inputs to the Salsa core, and in both Salsa and Chacha, the cores are used simply in counter mode to produce their pseudorandom streams. All of this is consistent with the notion that the cores are simply pseudorandom functions, which immediately makes Salsa and Chacha pseudorandom functions as well, since they are simple counter wrappers around a core. These statements by Bernstein were originally about the Salsa stream cipher, but Chacha is designed as an incremental improvement of Salsa and is assumed to have inherited its relevant security properties. Under this assumption, Hashstream produces a different pseudorandom output for each distinct (internal hash result, nonce) pair presented to Chacha. When distinct nonces are in use, these pairs always differ, making any effective attack against Hashstream an effective attack against Chacha: $\mathrm{Adv}^{\mathrm{prf}}_{\mathrm{hs}} \leq \mathrm{Adv}^{\mathrm{prf}}_{\mathrm{chacha}}$. On the other hand, if nonces are allowed to repeat and Hashstream is using an ε-almost-universal hash function internally, then $\mathrm{Adv}^{\mathrm{prf}}_{\mathrm{hs}} \leq q^2\varepsilon + \mathrm{Adv}^{\mathrm{prf}}_{\mathrm{chacha}}$ when Hashstream is invoked q times. The $q^2\varepsilon$ term upper-bounds the chance that any two inputs hash to the same intermediate value. When, for example, Poly1305 is used as the universal hash function and Hashstream inputs are limited to no more than L bytes each, $\varepsilon \approx L/2^{106}$. Nonce repetition is acceptable in this scenario only if both q and L can be kept low, but security degrades quickly if either—especially q—becomes large. If q and L cannot be kept small, the use of nonces becomes essential for security. A higher security version of Hashstream could easily be constructed where

the universal hash is computed twice with different keys and the internal hash is considered the concatenation of the results. This causes the internal hash collision probability to be upper bounded at $q^2\varepsilon^2$ instead.

Counter-mode stream cipher: A Hashstream version employing a block-cipher-based stream cipher is desirable because of the wide proliferation of AES hardware. In many systems, an AES-hardware-assisted stream cipher will be faster than one executed only in software or one based on Chacha20. The stream cipher interface required by Hashstream receives a hash output and a nonce and supplies a long pseudorandom string as output. A natural idea to meet this requirement with a block cipher is to use Liskov, Rivest and Wagner's tweakable block cipher $E_K(M \oplus h(T)) \oplus h(T)$ [7]. In counter-mode with a nonce initializing the counter, this construction is a perfect syntactic fit (i.e., it accepts an arbitrary string to hash, and a fixed-size nonce can be its initialization vector). This construction, however, suffers too badly from a birthday bound. To improve security, we adopt the strategy, reported by Naito, of changing the block cipher key with changes in the nonce [8]. To avoid updating the internal block cipher key with every application of Hashstream, we only update it when there are changes to the nonce outside the low eight bits. This means that when the nonce is incremented as a counter, the internal key is only updated every 256 invocations of Hashstream.

This construction is not very resistant to failure when the nonce is held steady, in which case, security degrades to that of Liskov, Rivest and Wagner's tweakable block cipher, $\mathrm{Adv}_E^{\mathrm{prf}} + 3\varepsilon q^2$, where q is the number of Hashstream calls and $\mathrm{Adv}_E^{\mathrm{prf}}$ is over the total number of block cipher blocks output. With changing nonces, however, any distinguishing attack on Hashstream reduces to a distinguishing attack on Naito's construction, so we adopt Naito's security bounds. Using an ideal cipher analysis, he claims that when m different internal keys are used, n different hashes are produced per internal key, and the block cipher is on b bit blocks, then Naito's construction can be distinguished from a tweakable PRP with no more advantage than $n^2m/2^b$. This leads to Hashstream security $\mathrm{Adv}_{\mathrm{hs}}^{\mathrm{prf}} \leq m \cdot \mathrm{Adv}_E^{\mathrm{prf}} + n^2m/2^b$, where $\mathrm{Adv}_E^{\mathrm{prf}}$ is over the maximum number of block cipher blocks output per internal key (i.e., over 256 consecutive nonces).

A full security analysis will be the topic of another paper.

1.6. Hashstream Abstraction and API

The primitives used in symmetric cryptography provide a variety of abstractions: block ciphers provide a random bijection on a fixed block size; stream ciphers provide a random function from integers to infinite strings; cryptographic hash functions are public random functions from arbitrary strings to fixed-length strings, etc. Each of these abstractions are not terribly difficult to understand on their own, but understanding how to piece these abstractions together to provide cryptographic services is neither straightforward, nor easy to prove correct. One reason for this difficulty is that these abstractions are too low-level: the gap between the abstraction and the desired service is large enough that how to provide the service is neither obvious, nor obviously correct.

A major goal of this work is to provide a cryptographic object with a higher level of abstraction so as to make usage of the abstraction both easy to understand and easy to prove correct. By reducing the gap between cryptographic abstraction and cryptographic service, this work strives to increase security by reducing the likelihood of error. Most symmetric cryptography is accomplished using objects that simulate random mappings. Some of these mappings have fixed-length domains or ranges, while others have variable length domains or ranges. None of them have variable length domains and ranges, and that is precisely the gap that Hashstream fills. The benefit that Hashstream delivers is flexibility. It can consume either a variable-length input or a fixed-length one. Likewise, it can produce an output of any length. It is simply more likely to be a tool that can do what is needed in any particular situation than any other lower-level primitive.

Hashstream does incur some overhead as the cost for its versatility. A construction custom designed for a cryptographic service using lower-level abstractions is likely going to be more efficient than one based on Hashstream. However, Hashstream is designed to be as fast as possible given its

flexibility, and the assurances it gives may be worth the cost, especially for services that are easy to prove with Hashstream and more difficult otherwise.

Hashstream is designed to be a simple and powerful abstraction, and this is reflected in the suggested application programming interface (API) given in Appendix A. As an example of the versatility of the abstraction and API, here is code that implements authenticated encryption of a plaintext using Hashstream and Rogaway and Shrimpton's SIV mode [6].

```
/* siv_encrypt encrypts buf in-place then appends siv followed by nonce used */
void siv_encrypt(hs_ctx *ctx, unsigned char *buf, int nbytes, unsigned char *nonce)
{
unsigned char *nonce_used;
memset(buf+nbytes, 0, SIVLEN);
nonce_used = hs_hashstream(ctx, nonce, buf, nbytes, buf+nbytes, SIVLEN);
memcpy(buf+nbytes+SIVLEN, nonce_used, hs_nonce_nbytes());
hs_hashstream(ctx, NULL, buf+nbytes, SIVLEN, buf, nbytes);
}
```

The first call to `hs_hashstream` consumes the `nbytes` bytes of plaintext pointed at by `buf` and writes `SIVLEN` pseudorandom bytes just afterward. These bytes—called the synthetic initialization vector (SIV) and typically 16 bytes long—serve both as a MAC tag for the plaintext, but also as an initialization vector for encrypting it. The second call to `hs_hashstream` consumes the SIV and produces pseudorandom bytes that are then xor'd with the plaintext to produce the ciphertext. Because the API xor's `hs_hashstream` output with whatever is already in the output buffer, the buffer must be set to zero ahead of time if overwriting the buffer is the desired behavior.

1.7. Related Work

Cryptographic objects that map arbitrary length inputs to arbitrary length outputs go back at least to 1994 and Bellare and Rogaway's OAEP [15]. Their "generators", now commonly called mask generation functions, are unkeyed and typically based on cryptographic hash functions. In 2009, Boldyreva, Chenette, Lee and O'Neill created a "length-flexible PRF" for their work on order-preserving symmetric encryption [16]. Their construction uses a block-cipher-based MAC to consume input and then uses the MAC tag as the key in a block-cipher-based stream cipher. Neither of these examples indicated high-speed as a goal.

More recent constructions have focused more on speed. HS1, a precursor to the work in this paper, was introduced as part of the CAESAR submission HS1-SIV in 2014 [5,17]. Bernstein's HHFHFH construction, suggested at a workshop in 2016, composes a hash function with a stream cipher to achieve variable length input and output efficiently [18]. In 2017, the designers of Keccak published a design called Farfalle, which employs a permutation with key-dependent masks to process arbitrary length inputs and output [19]. The Farfalle design is careful to allow a high degree of parallelism, which results in good speed on systems with sufficient resources. The Farfalle authors also pointed out the utility of a length-flexible PRF and provided timing data for Farfalle when providing various services such as authentication, encryption and authenticated encryption.

Some newer authenticated-encryption schemes marry universal hashing with a stream cipher, most notably AES-GCM-SIV [20]. This particular scheme uses GHASH for hashing, AES-CTR for encryption and combines them in an SIV manner. That work differs from Hashstream in that it is designed to do only authenticated encryption and is not easily adapted to perform other tasks.

2. Results

The tangible contribution of this work is Hashstream software artifacts, which are placed in the public domain and available online [21]. This section provides a performance study of the

software. For the rest of the paper, Hashstream using Poly1305 and Chacha20 will be referred to as Hashstream/PC and Hashstream using GHASH and AES-CTR will be referred to as Hashstream/GA.

Hashstream was designed to incur very little overhead beyond the cost of a universal hash function computation to consume the variable length input and the cost of a stream cipher computation to produce as many bytes as requested. To achieve this low overhead, there must be no expensive computations between the two phases. This is achieved by careful selection of the stream cipher used.

In the case of Hashstream/PC, the interface is particularly simple. Chacha20 expects a key and nonce copied into a buffer, and immediately it is ready to start producing its output. There is no key setup beyond that. Because Chacha20 is essentially a PRF mapping (key, nonce)-pairs into pseudorandom outputs, all that is required of the Poly1305 output is that it gets xor-ed into the Chacha20 key before use. This means that the only work needed between hashing and streaming is 16 bytes of xor and 48 bytes of data copying, which is a nearly negligible amount of overhead.

In the case of Hashstream/GA, the integration of the GHASH output is simple: it gets xor-ed into the first and last AES round keys currently in use, and then, AES-CTR proceeds with producing output blocks. The chosen Hashstream nonce, however, can result in significant extra overhead. The AES key used for the counter mode is under the control of the Hashstream key and Hashstream nonce. Whenever any of the first 15 bytes of the Hashstream nonce changes, a new AES key is generated for use in the counter mode computation. This generation is fairly expensive: one or two AES invocations (depending on whether 128-bit AES keys are in use) and then an AES key setup using the result of those one or two AES invocations. There is potentially significant overhead associated with each stream. To mitigate this, it is recommended to increment the nonce as a counter, which will cause the first 15 bytes to see a change only once every 256 streams. Amortized over those 256 streams, there is 1/256 of this key generation process per stream. This is not quite negligible overhead, but still quite low; just a few cycles per stream. If nonces are chosen another way, such as randomly, then nonce overhead becomes more significant. Table 2 has a line marked "rekey", which demonstrates this effect. The data on that line were generated by always streaming with a nonce requiring an internal rekey. On the Skylake CPU, the result is an extra approximately 150 CPU cycles per stream.

Tables 2–4 show, respectively, Hashstream performance for streaming, hashing and, as an example application, authenticated encryption using SIV mode. Tables 2 and 3 closely mirror the performance characteristics for the cryptographic primitives they are based on. Although not shown in this paper, whenever Hashstream is used in a unified fashion (i.e., calling on hashing and streaming in a single call to the API), the computational cost is very close to the sum of the hash and stream portions, indicating that the interfacing efficiency goal has been achieved.

The Skylake, Ryzen, Cortex-A72 and Cortex-A53 are all 64-bit CPUs with assembly instructions for accelerating both AES and GHASH. The Cortex-A15 and Cortex-A5 are both 32-bit CPUs without such instructions. All systems were running Arch Linux and GCC 8.1.1. The Skylake was an Intel i5-6600, and the Ryzen was a Ryzen 7 1700.

Table 2. Hashstream streaming throughput on various output byte lengths (in CPU cycles per byte). The line marked "rekey" uses nonces forcing internal rekeying with each call, costing ≈150 cycles.

Skylake	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Poly1305 + Chacha20	22.3	12.0	5.2	4.9	2.5	1.3	1.2	1.2	1.2	1.2
GHASH + AES-CTR	5.5	3.0	1.6	1.1	0.9	0.7	0.7	0.7	0.6	0.6
GHASH + AES-CTR (rekey)	14.9	7.9	4.0	2.3	1.5	1.0	0.8	0.7	0.7	0.7
Ryzen	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Poly1305 + Chacha20	19.0	10.0	4.5	4.3	3.4	1.7	1.7	1.7	1.7	1.7
GHASH + AES-CTR	4.1	2.2	1.1	0.9	0.6	0.4	0.3	0.3	0.3	0.3

Table 2. *Cont.*

Cortex-A72	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Poly1305 + Chacha20	36.0	19.0	8.5	8.3	5.1	4.6	4.5	4.5	4.5	4.5
GHASH + AES-CTR	6.8	3.4	2.4	1.6	1.3	1.1	1.0	1.0	1.0	0.9
Cortex-A53	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Poly1305 + Chacha20	43.7	24.3	9.6	9.1	5.1	4.9	4.8	4.8	4.7	4.7
GHASH + AES-CTR	14.1	7.1	4.7	2.9	2.3	1.8	1.7	1.5	1.5	1.5
Cortex-A15	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Poly1305 + Chacha20	51.1	27.1	11.9	11.4	5.2	5.1	5.0	5.0	5.0	5.0
GHASH + AES-CTR	27.4	24.2	22.3	18.8	16.6	15.5	14.9	14.6	14.4	14.4
Cortex-A5	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Poly1305 + Chacha20	90.7	47.8	21.3	20.4	14.8	14.5	14.3	14.2	14.1	14.1
GHASH + AES-CTR	56.2	47.8	43.4	47.0	41.6	38.9	37.5	36.9	36.6	36.4
Cortex-A5 w/o NEON	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Poly1305 + Chacha20	91.6	48.8	21.2	20.4	19.9	19.6	19.4	19.3	19.3	19.3
GHASH + AES-CTR	58.8	52.0	48.4	46.6	45.6	45.2	44.9	44.8	44.8	44.8

Table 3. Hashstream hashing throughput on various input byte lengths (in CPU cycles per byte).

Skylake	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Poly1305 + Chacha20	6.5	4.6	2.8	2.4	1.5	1.0	0.7	0.6	0.6	0.5
GHASH + AES-CTR	7.2	3.6	1.8	0.9	0.6	0.5	0.4	0.4	0.4	0.4
Ryzen	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Poly1305 + Chacha20	5.3	3.0	2.0	2.5	1.7	1.3	1.1	1.0	1.0	1.0
GHASH + AES-CTR	3.2	1.8	1.1	0.7	0.5	0.4	0.4	0.4	0.4	0.4
Cortex-A72	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Poly1305 + Chacha20	9.8	6.2	4.5	4.0	2.5	1.8	1.5	1.3	1.2	1.2
GHASH + AES-CTR	7.2	3.8	2.1	1.4	1.1	0.9	0.8	0.8	0.7	0.7
Cortex-A53	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Poly1305 + Chacha20	16.2	9.5	6.1	5.4	3.4	2.4	2.0	1.7	1.6	1.5
GHASH + AES-CTR	7.3	4.5	2.4	1.7	1.3	1.1	1.0	0.9	0.9	0.9
Cortex-A15	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Poly1305 + Chacha20	18.0	10.5	8.2	4.8	3.0	2.1	1.7	1.5	1.4	1.3
GHASH + AES-CTR	17.0	12.4	9.9	8.7	8.1	7.8	7.7	7.6	7.5	7.5
Cortex-A5	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Poly1305 + Chacha20	40.0	22.4	18.3	10.8	6.9	4.9	4.0	3.5	3.2	3.1
GHASH + AES-CTR	42.0	29.9	23.9	20.9	19.3	18.6	18.2	18.0	17.9	17.9
Cortex-A5 w/o NEON	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Poly1305 + Chacha20	42.3	24.3	15.3	10.9	8.6	7.5	6.9	6.6	6.5	6.4
GHASH + AES-CTR	86.4	63.3	51.5	45.6	42.7	41.2	40.5	40.1	40.0	39.9

Table 4. Hashstream in SIV mode for authenticated encryption on various message byte lengths (in CPU cycles per byte). OCB and AES-GCM timings are taken from SUPERCOP [13]. Farfalle SAE timings are taken from [19].

Skylake	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**long**
Poly1305 + Chacha20	57.9	30.0	15.1	10.7	5.6	3.1	2.4	2.0	1.8	1.8
GHASH + AES-CTR	25.0	12.8	6.4	3.5	2.2	1.6	1.3	1.1	1.1	1.0
Farfalle SAE	88.8	44.4	22.2	11.1	–	–	–	–	1.9	1.4
AES-GCM	–	–	–	–	–	–	–	–	–	0.7
OCB	–	–	–	–	–	–	–	–	–	0.6
Ryzen	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**long**
Poly1305 + Chacha20	48.5	25.7	12.7	9.9	6.8	3.9	3.2	2.9	2.8	2.7
GHASH + AES-CTR	15.7	8.2	4.3	2.5	1.6	1.1	0.9	0.7	0.7	0.7
AES-GCM	–	–	–	–	–	–	–	–	–	1.1
OCB	–	–	–	–	–	–	–	–	–	0.4
Cortex-A72	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**long**
Poly1305 + Chacha20	92.2	49.1	25.2	18.3	10.7	7.9	6.8	6.2	5.9	5.8
GHASH + AES-CTR	28.1	14.2	8.1	4.8	3.3	2.4	2.1	1.8	1.7	1.7
AES-GCM	–	–	–	–	–	–	–	–	–	1.8
OCB	–	–	–	–	–	–	–	–	–	1.2
Cortex-A53	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**long**
Poly1305 + Chacha20	122.6	65.1	31.4	22.3	12.4	9.3	7.8	7.0	6.6	6.4
GHASH + AES-CTR	44.3	23.3	13.0	7.5	5.0	3.6	3.0	2.6	2.5	2.4
Cortex-A15	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**long**
Poly1305 + Chacha20	140.2	73.1	38.0	25.2	12.8	9.5	7.9	7.0	6.6	6.4
GHASH + AES-CTR	89.8	59.1	43.6	33.2	27.6	24.7	23.2	22.5	22.2	22.0
AES-GCM	–	–	–	–	–	–	–	–	–	37.6
OCB	–	–	–	–	–	–	–	–	–	20.5
Cortex-A5	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**long**
Poly1305 + Chacha20	269.4	140.1	74.3	48.4	30.4	23.7	20.4	18.7	17.8	17.4
GHASH + AES-CTR	199.8	128.5	93.1	80.7	67.5	60.7	57.4	55.7	54.9	54.5
AES-GCM	–	–	–	–	–	–	–	–	–	53.9
OCB	–	–	–	–	–	–	–	–	–	40.8
Cortex-A5 w/o NEON	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**long**
Poly1305 + Chacha20	276.1	144.2	71.8	49.1	37.5	31.5	28.5	27.1	26.3	26.0
GHASH + AES-CTR	294.2	190.9	137.7	111.1	97.7	91.1	87.8	86.2	85.4	84.9

3. Discussion

Hashstream should be viewed as a wrapper that bundles two lower-level primitives into a higher-level object with a simple abstraction. The new object largely preserves the speed advantages of the lower-level primitives while making them easier to access and harder to misuse. The cost of doing so is low. All of the primitives used in Hashstream are built into contemporary cryptographic libraries, and the wrapper code is short, resulting in just a few dozen lines of code providing the higher-level abstraction. A complete implementation of Hashstream/PC demonstrating this coding efficiency using OpenSSL is given in Appendix B.

When considering the cost of providing a new abstraction, efficiency is an important factor. It is clear from the performance data that, in some cases, speed is given-up by using the Hashstream abstraction. For example, peak performance on Skylake for authenticated encryption costs 0.7 cpb when using AES-GCM and 1.0 cpb when using Hashstream/GA in SIV mode (this speed disparity is due to the AES-GCM implementation interleaving encryption and authentication in a single pass over the data while SIV makes two passes by design). However, this misses the point. AES-GCM can

only easily do one thing. It cannot easily be the basis of a random number generator. It is not flexible with the size of its authentication tags. It cannot easily generate masks. It could likely be pressed into all of these duties, but it would be a complex, error-prone task and likely lose its speed advantage in the process. On the other hand, Hashstream can do them all easily and intuitively, without giving up much efficiency. Hashstream is a useful abstraction on which many services can be provided.

3.1. Nonces

Nonces can be a source of insecurity in cryptographic systems. Stateless systems, virtual-machine rollbacks, broken random generators and user's misunderstanding of requirements can all lead to nonce reuse.

One of Hashstream's goals is to make nonces optional. The user has the option of changing either the Hashstream nonce or the Hashstream input to achieve a new pseudorandom output. The user can hold the nonce constant, and as long as no two Hashstream inputs hash to the same value, a new pseudorandom output is produced. Because a birthday bound governs whether such collisions occur, it is more secure to update the nonce with each Hashstream invocation, but not essential when the number of applications is relatively low.

This nonce-optionality does not extend to all uses of Hashstream. The encryption scheme mentioned in the Introduction has an empty Hashstream input and relies on new nonces to produce the pseudorandom strings needed for encryption, for example.

The other given sample applications all benefit from nonces, yielding improved security, but do not fail when nonces are repeated. The authentication example has Hashstream behave as a one-input PRF when nonces are held constant, much as HMAC is. However, with the use of a nonce when authenticating, it is obscured whether messages being authenticated are repeated. The SIV mode of authenticated encryption also does not fail when nonces are held constant. When two messages are encrypted under the same nonce using an SIV authenticated-encryption scheme, it is leaked whether they are identical or not, but beyond that, full security is maintained.

It is worth repeating, however, that these examples and how they behave with or without nonces is a byproduct of the application's design and not a result of Hashstream itself. The Hashstream design allows good security in terms of its stated goal when nonces are reused and better security when they are updated with each invocation.

3.2. Relation to HS1-SIV

Another version of Hashstream called HS1 was developed in 2014 as part of the HS1-SIV submission to the CAESAR authenticated-encryption competition [5]. HS1 uses the same basic construction as Hashstream/PC, except that instead of Poly1305, it uses a custom hash function called HS1-Hash and allows reduced-round Chacha variants for situations where less security and higher speeds are required. The SUPERCOP benchmarking website reveals that HS1-SIV's performance is similar to that of Hashstream/PC when used in SIV mode. Romain Dolbeau contributed an AVX2 version of HS1-SIV with a peak speed on Skylake of 1.6 cpb (compared with 1.8 cpb for Hashstream/PC). Although HS1 may be slightly faster than Hashstream/PC, there are at least two reasons to prefer Hashstream/PC. Poly1305 and Chacha have been adopted for inclusion in TLS, meaning both will be specified in IETF RFCs and both will be available in numerous high-quality cryptographic libraries. Limiting the number of cryptographic primitives in common use is beneficial because it reduces coding requirements, increases confidence and improves interoperability. Not asking libraries to include another primitive is likely preferable. Furthermore, HS1-Hash requires over 200 bytes of internal key, while Poly1305 requires only 16. In an unconstrained environment where speed is paramount, HS1 may be a better choice than Hashstream/PC, but as a general choice, Hashstream/PC is nearly as performant and likely easier to develop.

3.3. Concluding Remarks and Future Work

Hashstream has a higher level of abstraction than most algorithms used in cryptography. It is not so high that it becomes difficult to use because of over-generality, but high enough that its applicability is wide and simple. At the same time, as the performance study in this paper shows, it is low-level and efficient enough to have exceptional speed on its own and in a wide variety of use cases. Hashstream brings to both universal hash functions and stream ciphers an approachability that neither currently enjoys.

Future possible improvements of Hashstream include developing versions with lower collision probabilities to make nonce use less important. This is likely to involve using a universal hash with a lower collision probability or using existing hashes multiple times with different keys. Furthermore, the concrete bounds for the block-cipher-based Hashstream construction still have a birthday bound adversarial success probability when the nonce is held steady or very long messages are encrypted. Investigating ways of improving this state is an open question. One possibility is changing the block cipher key not only when nonces change between Hashstream calls, but within a call as well. Perhaps the block cipher key in use can be governed by a combination of nonce and counter value. This would cut off a potential birthday bound attack.

4. Materials and Methods

Critical to Hashstream speed, security and correctness is access to quality implementations of the underlying cryptographic functions. Rather than write them from scratch for this paper, we adopt the best available open-source implementations. A benefit of this strategy is that the Hashstream code itself is short and easy to verify. This section discusses software availability, summarizes the steps to build the experiments used in this paper and outlines the timing methodology used.

4.1. Software Availability

The software used in this paper can be found in two places. The lower-level cryptographic algorithms Poly1305, Chacha20, GHASH and AES-CTR are taken from the open-source OpenSSL cryptographic library [12]. This library has many good assembly-language implementations of these algorithms tailored for the architectures of interest. Version 1.1.1 was used. Later or earlier versions may not work with the provided Hashstream code. None of the OpenSSL library routines that Hashstream relies on have public programming interfaces and are designed to only be used internally by OpenSSL. This means the OpenSSL authors are free to change the interface at any time to suit their needs. OpenSSL code is open-source and soon to be governed by the Apache v2.0 license.

The Hashstream code is freely available at a publicly available source code repository [21]. Table 5 lists the files used for the timing results in this paper. Hashstream code is in the public domain.

Table 5. Hashstream files found at [21].

File	Contents
`hashstream.h`	C header with programming interface and Doxygen documentation
`hspc.c`	Hashstream/PC implementation (Poly1305 and Chacha20)
`hsga.c`	Hashstream/GA implementation (GHASH and AES-CTR)
`hs_timer.c`	Program for producing Hashstream timings
`README.md`	Build instructions and more information
`LICENSE`	License file placing code into public domain

4.2. Building

To reproduce the results in this paper, complete the following steps.

1. Download OpenSSL anywhere on your system from https://www.openssl.org/source/ [12]. This paper was developed using Version 1.1.1.
2. Build OpenSSL: extract the OpenSSL archive, `cd`, into the new directory, run the configurator `./config -march=native -mtune=native`, and execute `make`. Depending on your architecture, you may need to change the `-march=native -mtune=native` to whatever is right for your machine. Adding `CC=clang` appears to work on Clang-based installations.
3. Compile your Hashstream application with the resulting `libcrypto.a` file. For example: `gcc -march=native -mtune=native -O3 hs_timer.c hspc.c openssl-1.1.1/libcrypto.a`.

4.3. Timing

The most important software CPU architectures currently are 64-bit Intel x86 and 64-bit ARMv8, which dominate among CPUs in laptops, desktops, servers, smartphones and tablets. Luckily, these two architectures have easily-accessed CPU cycle counters, making timing on both straightforward. The basic strategy for timing how many CPU cycles it takes to execute code X is: read cycle count; run X; read cycle count. Logically, the difference between the two counts is how many cycles running X consumed.

A read of a CPU's cycle count can, however, be nondeterministic. A CPU that is allowed to retire instructions out-of-order could return different instruction counts from run to run, which can be especially significant if X does not take many cycles. Process suspension by the operating system can also cause instructions to be counted that should not be attributed to the timing of X. Finally, reading cycle counts can cause the CPU pipeline to be flushed on some architectures, causing timing inaccuracies. To combat pipeline flushes and out-of-order variations, we run the algorithm 100 times between cycle count reads and divide the difference by 100 to get the average number of cycles per invocation (i.e., read cycle count; run X 100 times; read cycle count; divide difference by 100). This approach has the added advantage of allowing the inclusion of essential work external to X such as nonce incrementation.

The above method will give an accurate approximation of CPU cycles for running X as long as the OS does not run for a significant amount of time between cycle count reads. Because the timing of OS suspensions is unpredictable, we run the above experiment 100 times and report the tenth fastest one (i.e., run (read cycle count; run X 100 times; read cycle count; divide difference by 100) 100 times; report tenth fastest). Theoretically, the fastest of the 100 runs represents the experiment with the fewest OS interruptions and would be a legitimate result to report, but to be conservative, we throw out the ten fastest runs.

This timing method works with all x86 architecture models and with ARM architecture Cortex-A application processors since ARMv7. It does not work with ARM's lower-powered embedded Cortex-M processors, however, and so, we include results in our report from ARM's lowest-powered Cortex-A5 application processor with NEON extensions turned off as a proxy for 32-bit embedded processors.

The timing code is part of Hashstream's open software repository and can be found in the file `hs_timer.c`.

Funding: This research was funded by the United States National Science Foundation Grant Number 1314592 and by a sabbatical grant from California State University, Sacramento.

Conflicts of Interest: The author declares no conflict of interest. The funding sponsors had no role in the design of the study; in the collection, analyses or interpretation of data; in the writing of the manuscript; nor in the decision to publish the results.

Appendix A. Sample API

This Appendix presents the public documentation and API used for generating the results of this paper. It is the version current as of the submission of this paper. See the software repository for any corrections or errata [21].

```
/** @file hashstream.h
* @brief C header for Hashstream, a simple cryptographic
*    pseudorandom function
*
* Hashstream is a cryptographic construct that, once initialized,
* consumes arbitrary bytes of input and a nonce and produces
* as many pseudorandom bytes as requested. Each different
* input (data,nonce) pair should result in an independent
* pseudorandom output. The output bytes are most secure when the
* nonce changes with every call, but still have good security when
* the nonce is fixed and only the data changes.
*
* The length of the key and nonce you supply is implementation
* dependent, so you should consult your chosen implementation's
* documentation for these details.
*
* @see https://github.com/krovetz/Hashstream
* @date 1 July 2018
* @author Ted Krovetz (ted@krovetz.net)
*/

/** @brief Forward declaration of struct to represent a hashstream.
*
* Because `struct _hs_ctx` is declared but not defined, declarations
* of type `hs_ctx*` are allowed, but not of type `hs_ctx` because
* the compiler does not know its size. To dynamically allocate an
* `hs_ctx` call `malloc(hs_ctx_nbytes())` and cast the result to
* `(hs_ctx *)`. To auto-allocate an `hs_ctx` on the stack, define
* an array of `uint64_t` conatining enough bytes to accommodate an
* `hs_ctx` and cast the array to `(hs_ctx *)`. Use hs_ctx_nbytes()
* or read your implementation's documentation to discover how many
* bytes are needed. In either case, zero the `hs_ctx` before
* deallocating it to avoid leaking secrets to available memory.
*
* For example:
* @code
* hs_ctx *ctx = (hs_ctx *)malloc(hs_ctx_nbytes());
* ... Initialize and use ctx ...
* memset(ctx,0,hs_ctx_nbytes());
* free(ctx);
* @endcode
* or
* @code
* uint64_t ctx_storage[100]; // Change 100 to accommodate your hs_ctx.
* hs_ctx *ctx = (hs_ctx *)ctx_storage;
```

```
* ... Initialize and use ctx ...
* memset(ctx,0,hs_ctx_nbytes());
* @endcode
*/
typedef struct _hs_ctx hs_ctx;


/** @brief Initialize ‘hs_ctx‘ structure using ‘key‘.
*
* This function uses the supplied key to initialize an ‘hs_ctx‘
* structure. The bytes referred to by ‘key‘ should be random or
* pseudorandom, be of length ‘keybytes‘, and ‘keybytes‘ should be
* a value recommended by the implementation being used. If ‘keybytes‘
* is not a supported value, then the bytes you do supply will be
* truncated or stretched cryptographically to achieve a usable key.
* If ‘keybytes‘ is less than one, a publicly-known default key will
* be used, suitable only for non-security purposes. In any case, the
* context-embedded nonce and hash result are initialized to zeros.
*
* @param ctx Pointer to an ‘hs_ctx‘ structure.
* @param key Pointer to (pseudo)random key of length ‘keybytes‘.
* @param keybytes Number of bytes ‘key‘ points to. If less than one,
*   a publicly-known default is used. See implementation
*   documentation to find recommended values.
*/
void hs_ctx_init(hs_ctx *ctx, const unsigned char *key, int keybytes);


/** @brief Hash ‘inbytes‘ bytes pointed to by ‘in‘.
*
* This function hashes ‘inbytes‘ bytes pointed to by ‘in‘ using a
* key placed in the ‘hs_ctx‘ by ‘hs_ctx_init‘. The result is held
* in the ‘hs_ctx‘ for later use.
*
* A pointer to the hash result is returned as a convenience in case
* you want to use the hash capability of Hashstream independent of the
* streaming capability.
*
* @param ctx Pointer to an initialized ‘hs_ctx‘ structure.
* @param in Pointer to bytes to be hashed.
* @param inbytes Number of bytes to be hashed.
* @return Pointer to hash result (inside of the ‘hs_ctx‘ pointed to
*   by ‘ctx‘). See implementation documentation or call
*   ‘hs_hash_nbytes()‘ to find its length.
*/
unsigned char* hs_hash(hs_ctx *ctx, const unsigned char *in, int inbytes);


/** @brief Write ‘outbytes‘ bytes to buffer pointed to by ‘out‘.
*
```

```
* This function employs a stream cipher using a key and the hash
* embedded in the ‘hs_ctx‘ structure pointed to by ‘ctx‘, along with
* a ‘nonce‘, to produce any number of pseudorandom bytes. The produced
* bytes are xor-ed with the bytes already in the output buffer. If you
* simply want random bytes in your buffer, zero your buffer before
* calling ‘hs_stream‘.
*
* For best security, ‘nonce‘ should be different for each call. Each
* time you explicitly supply a nonce, a copy is kept in the ‘hs_ctx‘.
* If NULL is supplied for ‘nonce‘, the copy in the ‘hs_ctx‘ is updated
* by adding 1 (mod 2^64) to the last 8 bytes of the nonce as if it
* were a big-endian integer, and the updated nonce is used for the
* ‘hs_hash‘ invocation. A pointer to the used nonce copy in the
* ‘hs_ctx‘ is returned as a convenience.
*
* @param ctx Pointer to an initialized ‘hs_ctx‘ structure.
* @param nonce Pointer to a nonce, or NULL to indicate
*   autoincrement of prior nonce.
* @param out Pointer to buffer to be xor-ed with pseudorandom bytes.
* @param outbytes Number of bytes to write to the buffer.
* @return Pointer to the copy of the nonce used. See implementation
*   documentation or call ‘hs_nonce_nbytes()‘ to find its length.
*/
unsigned char* hs_stream(hs_ctx *ctx,
const unsigned char *nonce,
unsigned char *out, int outbytes);


/** @brief Composition of ‘hs_hash‘ followed by ‘hs_stream‘.
*
* This is a convenience function defined exactly as follows.
*
* @code
* unsigned char* hs_hashstream(hs_ctx *ctx,
*     const unsigned char *nonce,
*     const unsigned char *in, int inbytes,
*     unsigned char *out, int outbytes) {
*   hs_hash(ctx, in, inbytes);
*   return hs_stream(ctx, nonce, out, outbytes);
* }
* @endcode
*
* @param ctx Pointer to an initialized ‘hs_ctx‘ structure.
* @param nonce Pointer to a nonce, or NULL to indicate
*   autoincrement of last nonce.
* @param in Pointer to bytes to be hashed.
* @param inbytes Number of bytes to be hashed.
* @param out Pointer to buffer to be xor-ed with pseudorandom bytes.
* @param outbytes Number of bytes to write to the buffer.
* @return Pointer to the copy of the nonce used. See implementation
```

```
*   documentation or call ‘hs_nonce_nbytes()‘ to find its length.
*/
unsigned char* hs_hashstream(hs_ctx *ctx,
const unsigned char *nonce,
const unsigned char *in, int inbytes,
unsigned char *out, int outbytes);

/** @brief Returns the length in bytes of nonces. */
int hs_nonce_nbytes();

/** @brief Returns the length in bytes of hashes. */
int hs_hash_nbytes();

/** @brief Returns the length in bytes of ‘hs_ctx‘ structures. */
int hs_ctx_nbytes();
```

Appendix B. Sample Implementation: Hashstream/PC

This Appendix presents the C code implementation of Hashstream/PC as a demonstration of how little code is necessary to implement the Hashstream abstraction when using OpenSSL. Similar effort would be needed if using another cryptographic library. This code is the version current as of the submission of this paper. See the software repository for any corrections or errata [21].

```
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include "hashstream.h"

/* ------------------------------------------------------------------- */
/* Private OpenSSL interfaces (from 1.1) which could change in the future. */

void OPENSSL_cpuid_setup(void);
void Poly1305_Init(void *ctx, const unsigned char key[32]);
void Poly1305_Update(void *ctx, const unsigned char *inp, size_t len);
void Poly1305_Final(void *ctx, unsigned char mac[16]);
void ChaCha20_ctr32(unsigned char *out, const unsigned char *inp,
size_t len, const uint32_t key[8],
const uint32_t counter[4]);

/* ------------------------------------------------------------------- */
/* Each field is at least 8B aligned for both stack and malloc allocation. */

struct _hs_ctx {
uint64_t nonce[2];      /* mem order, uint64_t forces 8B align */
uint8_t hashkey[16];    /* mem order                           */
uint8_t hashresult[16]; /* mem order                           */
uint32_t key[8];        /* 16B set for LE reads, 16B mem order */
};

/* ------------------------------------------------------------------- */

static const union { unsigned x; unsigned char e; } l = { 1 };
```

```
static void bswap4x32_if_be(uint32_t *p) {
int i; if ( ! l.e ) for (i=0 ; i<4 ; i++) p[i] = __builtin_bswap32(p[i]);
}
static uint64_t load64_be(uint64_t *p){return l.e?__builtin_bswap64(*p):*p;}
static void store64_be(uint64_t *p,uint64_t x){*p=l.e?__builtin_bswap64(x):x;}
static void xor128(void *d, void *s) {
uint64_t *dp=(uint64_t *)d, *sp=(uint64_t *)s, t=sp[0];
dp[1]^=sp[1]; dp[0]^=t;
}

/* ------------------------------------------------------------------------ */

int hs_nonce_nbytes() { return 12; }
int hs_hash_nbytes() { return 16; }
int hs_ctx_nbytes() { return (int)sizeof(hs_ctx); }

/* ------------------------------------------------------------------------ */

void hs_ctx_init(hs_ctx *ctx, const unsigned char *key, int keybytes) {
/* Default key (WolframAlpha): IntegerPart[(pi - 3) * 2^128] to hex */
uint8_t def[16] = {0x24,0x3f,0x6a,0x88,0x85,0xa3,0x08,0xd3,
0x13,0x19,0x8a,0x2e,0x03,0x70,0x73,0x44};
OPENSSL_cpuid_setup();
if (key==NULL || keybytes==0) {
memcpy(ctx->hashkey, def, 16);
memset(ctx->key,0,32);
} else if (keybytes>=48) {
memcpy(ctx->hashkey,key,16);
memcpy(ctx->key,key+16,32);
} else {
uint32_t zero[4] = {0};
memset(ctx,0,sizeof(hs_ctx)); /* Use entire ctx as scratch space */
memcpy(ctx->nonce,key,(keybytes>=32?32:keybytes));
bswap4x32_if_be((uint32_t *)ctx->nonce);
bswap4x32_if_be((uint32_t *)ctx->hashkey);
ChaCha20_ctr32(ctx->hashresult, ctx->hashresult, 48, ctx->key, zero);
memcpy(ctx->hashkey, ctx->hashresult, 16);
}
memset(ctx->nonce,0,16);
memset(ctx->hashresult,0,16);
bswap4x32_if_be(ctx->key);
}

/* ------------------------------------------------------------------------ */

unsigned char* hs_hash(hs_ctx *ctx, const unsigned char *in, int inbytes) {
uint64_t poly_ctx[32]; /* 256 bytes/8-byte aligned, enough for OpenSSL */
memset(ctx->hashresult,0,16);
Poly1305_Init(poly_ctx, ctx->hashkey);
```

```
Poly1305_Update(poly_ctx, in, inbytes);
Poly1305_Final(poly_ctx, ctx->hashresult);  /* OpenSSL zeros poly_ctx */
return ctx->hashresult;
}

/* ---------------------------------------------------------------------- */

unsigned char* hs_stream(hs_ctx *ctx,
const unsigned char *nonce,
unsigned char *out, int outbytes) {
if (nonce) memcpy((uint32_t *)ctx->nonce+1,nonce,12);
else       store64_be(ctx->nonce+1, load64_be(ctx->nonce+1) + 1);
xor128(ctx->key+4, ctx->hashresult);
bswap4x32_if_be((uint32_t *)(ctx->nonce));
bswap4x32_if_be(ctx->key+4);
ChaCha20_ctr32(out, out, outbytes, ctx->key, (uint32_t *)ctx->nonce);
bswap4x32_if_be((uint32_t *)(ctx->nonce));
bswap4x32_if_be(ctx->key+4);
xor128(ctx->key+4, ctx->hashresult);
return (unsigned char *)(ctx->nonce);
}

/* ---------------------------------------------------------------------- */

unsigned char* hs_hashstream(hs_ctx *ctx,
const unsigned char *nonce,
const unsigned char *in, int inbytes,
unsigned char *out, int outbytes) {
hs_hash(ctx, in, inbytes);
return hs_stream(ctx, nonce, out, outbytes);
}

/* ---------------------------------------------------------------------- */
```

References

1. Bernstein, D.J. The Poly1305-AES message-authentication code. In *Fast Software Encryption*; Gilbert, H., Handschuh, H., Eds.; Lecture Notes in Computer Science 3557; Springer: Berlin/Heidelberg, Germany, 2005; pp. 32–49.
2. McGrew, D.A.; Viega, J. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In *Progress in Cryptology—INDOCRYPT 2004*; Canteaut, A., Viswanathan, K., Eds.; Lecture Notes in Computer Science 3348; Springer: Berlin/Heidelberg, Germany, 2004; pp. 343–355.
3. Bernstein, D.J. ChaCha, A Variant of Salsa20. Presented at SASC 2008: The State of the Art of Stream Ciphers, Lausanne, Switzerland. 2008. Available online: http://www.ecrypt.eu.org/stvl/sasc2008/ (accessed on 1 June 2018).
4. Dworkin, M. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*; SP 800-38A; NIST: Gaithersburg, MD, USA, 2001.
5. Krovetz, T. HS1-SIV (v2). CAESAR Submissions. 2015. Available online: https://competitions.cr.yp.to/round2/hs1sivv2c.pdf (accessed on 1 June 2018).
6. Rogaway, R.; Shrimpton, T. A provable-security treatment of the keywrap problem. In *Advances in Cryptology—EUROCRYPT 2006*; Vaudenay, S., Ed.; Lecture Notes in Computer Science 4004; Springer: Berlin/Heidelberg, Germany, 2006; pp. 373–390.

7. Liskov, M.; Rivest, R.L.; Wagner, D. Tweakable Block Ciphers. In *Advances in Cryptology—CRYPTO 2002*; Yung, M., Ed.; Lecture Notes in Computer Science 2442; Springer: Berlin/Heidelberg, Germany, 2002; pp. 31–46.
8. Naito, Y. Tweakable blockciphers for efficient authenticated encryptions with beyond the birthday-bound security. *IACR Trans. Symmetric Cryptol.* **2017**, *2*, 1–26. [CrossRef]
9. Halevi, S.; Krawczyk, H. MMH: Software message authentication in the Gbit/second rates. In *Fast Software Encryption*; Biham, E., Youssef, A.M., Eds.; Lecture Notes in Computer Science 4356; Springer: Berlin/Heidelberg, Germany, 2007; pp. 172–189.
10. Black, J.; Halevi, S.; Krawczyk, H.; Krovetz, T.; Rogaway, P. UMAC: Fast and secure message authentication. In *Advances in Cryptology—CRYPTO '99*; Wiener, M., Ed.; Lecture Notes in Computer Science 1666; Springer: Berlin/Heidelberg, Germany, 1999; pp. 216–233.
11. Krovetz, T. Message authentication on 64-bit architectures. In *Fast Software Encryption*; Gilbert, H., Handschuh, H., Eds.; Lecture Notes in Computer Science 3557; Springer: Berlin/Heidelberg, Germany, 2005; pp. 327–341.
12. OpenSSL: Cryptography and SSL/TLS Toolkit. Available online: https://www.openssl.org (accessed on 1 June 2018).
13. Bernstein, D.J.; Lange, T. (Eds.) eBACS: ECRYPT Benchmarking of Cryptographic Systems. Available online: https://bench.cr.yp.to (accessed on 1 June 2018).
14. Bernstein, D.J. Response to "On the Salsa20 Core Function". 2008. Available online: https://cr.yp.to/snuffle/reoncore-20080224.pdf (accessed on 1 June 2018).
15. Bellare, M.; Rogaway, R. Optimal asymmetric encryption. In *Advances in Cryptology—EUROCRYPT '94*; De Santis, A., Ed.; Lecture Notes in Computer Science 950; Springer: Berlin/Heidelberg, Germany, 1995; pp. 92–111.
16. Boldyreva, A.; Chenette, N.; Lee, Y.; O'Neill, A. Order-preserving symmetric encryption. In *Advances in Cryptology—EUROCRYPT 2009*; Joux, A., Ed.; Lecture Notes in Computer Science 5479; Springer: Berlin/Heidelberg, Germany, 2009; pp. 224–241.
17. CAESAR. Competition for Authenticated Encryption, Security, Applicability, and Robustness. Available online: https://competitions.cr.yp.to/caesar.html (accessed on 1 June 2018).
18. Bernstein, D.J. Some Challenges in Heavyweight Cipher Design. Presented at Dagstuhl Seminar on Symmetric Encryption, Dagstuhl, Germany, 15 January 2016. Available online: https://cr.yp.to/talks/2016.01.15/slides-djb-20160115-a4.pdf (accessed on 1 June 2018).
19. Bertoni, G.; Daemen, J.; Hoffert, S.; Peeters, M.; Van Assche, G.; Van Keer, R. Farfalle: parallel permutation-based cryptography. *IACR Trans. Symmetric Cryptol.* **2017**, *4*, 1–38. [CrossRef]
20. Gueron, S; Langley, A.; Lindell Y. AES-GCM-SIV: Specification and Analysis, 2017. Cryptology ePrint Archive. Available online: https://eprint.iacr.org/2017/168Report2017/168 (accessed on 1 June 2018).
21. Krovetz, T. Hashstream Code. GitHub Repository. 2018. Available online: https://github.com/krovetz/Hashstream (accessed on 1 June 2018).