SipHash: A Fast Short-Input PRF

Jean-Philippe Aumasson[1] and Daniel J. Bernstein[2]

[1] NAGRA
Switzerland
jeanphilippe.aumasson@gmail.com
[2] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7045, USA
djb@cr.yp.to

Abstract. SipHash is a family of pseudorandom functions optimized for short inputs. Target applications include network traffic authentication and hash-table lookups protected against hash-flooding denial-of-service attacks. SipHash is simpler than MACs based on universal hashing, and faster on short inputs. Compared to dedicated designs for hash-table lookup, SipHash has well-defined security goals and competitive performance. For example, SipHash processes a 16-byte input with a fresh key in 140 cycles on an AMD FX-8150 processor, which is much faster than state-of-the-art MACs. We propose that hash tables switch to SipHash as a hash function.

1 Introduction

A message-authentication code (MAC) produces a tag t from a message m and a secret key k. The security goal for a MAC is for an attacker, even after seeing tags for many messages (perhaps selected by the attacker), to be unable to guess tags for any other messages.

Internet traffic is split into short packets that require authentication. A 2000 note by Black, Halevi, Krawczyk, Krovetz, and Rogaway [11] reports that "a fair rule-of-thumb for the distribution on message-sizes on an Internet backbone is that roughly one-third of messages are 43 bytes (TCP ACKs), one-third are about 256 bytes (common PPP dialup MTU), and one-third are 1500 bytes (common Ethernet MTU)."

However, essentially all standardized MACs and state-of-the-art MACs are optimized for long messages, not for short messages. Measuring long-message performance hides the overheads caused by large MAC keys, MAC initialization, large MAC block sizes, and MAC finalization. These overheads are usually quite severe, as illustrated by the examples in the following paragraphs. Applications can compensate for these overheads by authenticating a concatenation of several packets instead of authenticating each packet separately, but then a single forged

This work was supported by the National Science Foundation under grant 1018836. Permanent ID of this document: b9a943a805fbfc6fde808af9fc0ecdfa. Date: 2012.09.17.

S. Galbraith and M. Nandi (Eds.): INDOCRYPT 2012, LNCS 7668, pp. 489–508, 2012.

packet forces several packets to be retransmitted, increasing the damage caused by denial-of-service attacks.

Our first example is HMAC-SHA-1, where overhead effectively adds between 73 and 136 bytes to the length of a message: for example, HMAC-SHA-1 requires two 64-byte compression-function computations to authenticate a short message. Even for long messages, HMAC-SHA-1 is not particularly fast: for example, the OpenSSL implementation takes 7.8 cycles per byte on Sandy Bridge, and 11.2 cycles per byte on Bulldozer. In general, building a MAC from a general-purpose cryptographic hash function appears to be a highly suboptimal approach: general-purpose cryptographic hash functions perform many extra computations for the goal of collision resistance on public inputs, while MACs have secret keys and do not need collision resistance.

Much more efficient MACs combine a large-input universal hash function with a short-input encryption function. A universal hash function h maps a long message m to a short hash $h(k_1, m)$ under a key k_1. "Universal" means that any two different messages almost never produce the same output when k_1 is chosen randomly; a typical universal hash function exploits fast 64-bit multipliers to evaluate a polynomial over a prime field. This short hash is then strongly encrypted under a second key k_2 to produce the authentication tag t. The original Wegman–Carter MACs [34] used a one-time pad for encryption, but of course this requires a very long key. Modern proposals such as UMAC version 2 [11], Poly1305-AES [5], and VMAC(AES) [25] [14] replace the one-time pad with outputs of AES-128: i.e., $t = h(k_1, m) \oplus \text{AES}(k_2, n)$ where n is a nonce. UMAC version 1 argued that "using universal hashing to reduce a very long message to a fixed-length one can be complex, require long keys, or reduce the quantitative security" [10, Section 1.2] and instead defined $t = \text{HMAC-SHA-1}(h(k, m), n)$ where $h(k, m)$ is somewhat shorter than m.

All of these MACs are optimized for long-message performance, and suffer severe overheads for short messages. For example, the short-message performance of UMAC version 1 is obviously even worse than the short-message performance of HMAC-SHA-1. All versions of UMAC and VMAC expand k_1 into a very long key (for example, 4160 bytes in one proposal), and are timed under the questionable assumptions that the very long key has been precomputed and preloaded into L1 cache. Poly1305-AES does not expand its key but still requires padding and finalization in h, plus the overhead of an AES call.

(We comment that, even for applications that emphasize long-message performance, the structure of these MACs often significantly complicates deployment. Typical universal MACs have lengthy specifications, are not easy to implement efficiently, and are not self-contained: they rely on extra primitives such as AES. Short nonces typically consume 8 bytes of data with each tag, and force applications to be stateful to ensure uniqueness; longer nonces consume even more space and require either state or random-number generation. There have been proposals of nonceless universal MACs, but those proposals are significantly slower than other universal MACs at the same security level; see, e.g., [4, Theorem 9.2].)

The short-input performance problems of high-security MACs are even more clear in another context. As motivation we point to the recent rediscovery of "hash flooding" denial-of-service attacks on Internet servers that store data in hash tables. These servers normally use public non-cryptographic hash functions, and these attacks exploit multicollisions in the hash functions to enforce worst-case lookup time. See Section 7 of this paper for further discussion.

Replacing the public non-cryptographic hash functions with strong small-output secret-key MACs would solve this problem. However, to compete with existing non-cryptographic hash functions, the MACs must be extremely fast for very short inputs, even shorter than the shortest common Internet packets. For example, Ruby on Rails applications are reported to hash strings shorter than 10 bytes on average. Recent hash-table proposals such as Google's CityHash [18] and Jenkins' SpookyHash [21] provide very fast hashing of short strings, but these functions were designed to have a close-to-uniform distribution, not to meet any particular cryptographic goals. For example, collisions were found in an initial version of CityHash128 [22], and the current version is vulnerable to a practical key-recovery attack when 64-bit keys are used.

This paper introduces the SipHash family of hash functions to address the needs for high-security short-input MACs. SipHash features include:

- **High security.** Our concrete proposal SipHash-2-4 was designed and evaluated to be a cryptographically strong PRF (pseudorandom function), i.e., indistinguishable from a uniform random function. This implies its strength as a MAC.
- **High speed.** SipHash-2-4 is much faster for short inputs than previous strong MACs (and PRFs), and is competitive in speed with popular non-cryptographic hash functions.
- **Key agility.** SipHash uses a 128-bit key. There is no key expansion in setting up a new key or hashing a message, and there is no hidden cost of loading precomputed expanded keys from DRAM into L1 cache.
- **Simplicity.** SipHash iterates a simple round function consisting of four additions, four xors, and six rotations, interleaved with xors of message blocks.
- **Autonomy.** No external primitive is required.
- **Small state.** The SipHash state consists of four 64-bit variables. This small state size allows SipHash to perform well on a wide range of CPUs and to fit into small hardware.
- **No state between messages.** Hashing is deterministic and doesn't use nonces.
- **No software side channels.** Many cryptographic functions, notably AES, encourage implementors to use secret load/store addresses or secret branch conditions, often allowing timing attacks. SipHash avoids this problem.
- **Minimal overhead.** Authenticated messages are just 8 bytes longer than original messages.

§2 presents a complete definition of SipHash; §3 makes security claims; §4 explains some design choices; §5 reports on our preliminary security analysis; §6 evaluates the efficiency of SipHash in software and hardware; §7 discusses the benefits of switching to SipHash for hash-table lookups.

2 Specification of SipHash

SipHash is a family of PRFs SipHash-c-d where the integer parameters c and d are the number of compression rounds and the number of finalization rounds. A compression round is identical to a finalization round and this round function is called SipRound. Given a 128-bit key k and a (possibly empty) byte string m, SipHash-c-d returns a 64-bit value SipHash-c-$d(k, m)$ computed as follows:

1. **Initialization:** Four 64-bit words of internal state v_0, v_1, v_2, v_3 are initialized as

$$\begin{aligned} v_0 &= k_0 \oplus \texttt{736f6d6570736575} \\ v_1 &= k_1 \oplus \texttt{646f72616e646f6d} \\ v_2 &= k_0 \oplus \texttt{6c7967656e657261} \\ v_3 &= k_1 \oplus \texttt{7465646279746573} \end{aligned}$$

   where k_0 and k_1 are the little-endian 64-bit words encoding the key k.
2. **Compression:** SipHash-c-d processes the b-byte string m by parsing it as $w = \lceil (b+1)/8 \rceil > 0$ 64-bit little-endian words $m_0, \ldots, m_{w-1}$ where m_{w-1} includes the last 0 through 7 bytes of m followed by null bytes and ending with a byte encoding the positive integer $b \bmod 256$. For example, the one-byte input string $m = \texttt{ab}$ is parsed as $m_0 = \texttt{01000000000000ab}$. The m_i's are iteratively processed by doing

$$v_3 \oplus = m_i$$

   and then c iterations of SipRound, followed by

$$v_0 \oplus = m_i$$

3. **Finalization:** After all the message words have been processed, SipHash-c-d xors the constant `ff` to the state:

$$v_2 \oplus = \texttt{ff}$$

   then does d iterations of SipRound, and returns the 64-bit value

$$v_0 \oplus v_1 \oplus v_2 \oplus v_3 \ .$$

Fig. 2.1 shows SipHash-2-4 hashing a 15-byte m.

The function SipRound transforms the internal state as follows (see also Fig.2.2):

$$\begin{array}{ll} v_0 += v_1 & v_2 += v_3 \\ v_1 \lll= 13 & v_3 \lll= 16 \\ v_1 \oplus = v_0 & v_3 \oplus = v_2 \\ v_0 \lll= 32 & \\ v_2 += v_1 & v_0 += v_3 \\ v_1 \lll= 17 & v_3 \lll= 21 \\ v_1 \oplus = v_2 & v_3 \oplus = v_0 \\ v_2 \lll= 32 & \end{array}$$

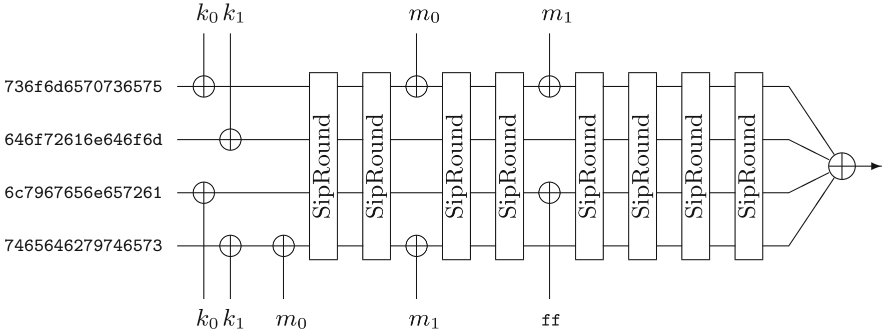

Fig. 2.1. SipHash-2-4 processing a 15-byte message. SipHash-2-4(k, m) is the output from the final $\oplus$ on the right.

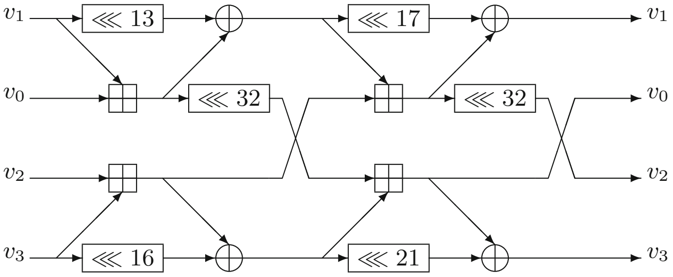

Fig. 2.2. The ARX network of SipRound

3 Expected Strength

SipHash-c-d with $c \geq 2$ and $d \geq 4$ is expected to provide the maximum PRF security possible (and therefore also the maximum MAC security possible) for any function with the same key size and output size. Our fast proposal is thus SipHash-2-4. We define SipHash-c-d for larger c and d to provide a higher security margin: our conservative proposal is SipHash-4-8, which is about half the speed of SipHash-2-4. We define SipHash-c-d for smaller c and d to provide targets for cryptanalysis. Cryptanalysts are thus invited to break

- SipHash-1-0, SipHash-2-0, SipHash-3-0, SipHash-4-0, etc.;
- SipHash-1-1, SipHash-2-1, SipHash-3-1, SipHash-4-1, etc.;
- SipHash-1-2, SipHash-2-2, SipHash-3-2, SipHash-4-2, etc.;

and so on.

Note that the standard PRF and MAC security goals allow the attacker access to the output of SipHash on messages chosen adaptively by the attacker. However, they do not allow access to any "leaked" information such as bits of the key or the internal state. They also do not allow “related keys”, “known keys”, “chosen keys”, etc.

Of course, security is limited by the key size (128 bits). In particular, attackers searching 2^s keys have chance 2^{s-128} of finding the SipHash key. This search is accelerated in standard ways by speedups in evaluation and partial evaluation of SipHash, for one key or for a batch of keys; by attacks against multiple targets; and by quantum computers.

Security is also limited by the output size (64 bits). In particular, when SipHash is used as a MAC, an attacker who blindly tries 2^s tags will succeed with probability 2^{s-64}.

We comment that SipHash is not meant to be, and (obviously) is not, collision-resistant.

4 Rationale

SipHash is an ARX algorithm, like the SHA-3 finalists BLAKE [**3**] and Skein [**16**]. SipHash follows BLAKE's minimalism (small code, small state) but borrows the two-input MIX from Skein, with two extra rotations to improve diffusion. SipHash's input injection is inspired by another SHA-3 finalist, JH [**36**].

Choice of Constants. The initial state constant corresponds to the ASCII string “somepseudorandomlygeneratedbytes”, big-endian encoded. There is nothing special about this value; the only requirement was some asymmetry so that the initial v_0 and v_1 differ from v_2 and v_3. This constant may be set to a “personalization string” but we have not evaluated whether it can safely be chosen as a “tweak”. Note that two nonzero words of initialization constants would have been as safe as four.

The other constant in SipHash is `ff`, as xored to v_2 in finalization. We could have chosen any other non-zero value. Without this constant, one can reach the internal state after finalization by just absorbing null words. We found no way to exploit this property, but we felt it prudent to avoid it given the low cost of the defense.

Choice of Rotation Counts. Finding really bad rotation counts for ARX algorithms turns out to be difficult. For example, randomly setting all rotations in BLAKE-512 or Skein to a value in $\{8, 16, 24, \ldots, 56\}$ may allow known attacks to reach slightly more rounds, but no dramatic improvement is expected.

The advantage of choosing such “aligned” rotation counts is that aligned rotation counts are much faster than unaligned rotation counts on many non-64-bit architectures. Many 8-bit microcontrollers have only 1-bit shifts of bytes, so rotation by (e.g.) 3 bits is particularly expensive; implementing a rotation by a mere permutation of bytes greatly speeds up ARX algorithms. Even 64-bit systems can benefit from alignment, when a sequence of shift-shift-xor can be replaced by SSSE3's `pshufb` byte-shuffling instruction. For comparison, implementing BLAKE-256's 16- and 8-bit rotations with `pshufb` led to a 20% speedup on Intel's Nehalem microarchitecture.

For SipHash, the rotation distances were chosen as a tradeoff between security and performance, with emphasis on the latter. We ran an automated search that picks random rotation counts, estimates the number of significant statistical biases on three SipRounds with respect to a specific significance threshold, and finally sorts the sets of rotation counts according to that metric. We then manually shortlisted a few sets, by choosing the ones with rotation counts the closest to multiples of eight. We changed some of those values to the closest multiple of eight and benchmarked them against our original security metric, and repeated this process several times until finding a satisfying set of rotation counts.

We chose counts 13, 16, 17, and 21 for the rotations in the two MIX layers: 13 and 21 are three bits away from a multiple of 8, whereas 17 is just one bit away, and 16 can be realized by byte permutation only. We aggressively set the two "asymmetric" rotation counts to 32 to minimize the performance penalty—it is just a swap of words on 32-bit systems. The 32-bit rotations significantly improve diffusion, and their position on the ARX network allows for an efficient scheduling of instructions.

Choice of Injection Structure. Like JH, SipHash injects input before and after each block, with the difference that SipHash leaves less freedom to attackers: whereas JH xors the message block to the two halves of the state before and after the permutation, SipHash xors a block to two quarters of the state. Any attack on the SipHash injection structure can be applied to the JH injection structure, so security proofs for the JH injection structure [30] also apply to the SipHash injection structure.

A basic advantage of the JH/SipHash injection structure compared to the sponge/Keccak [7] injection structure is that message blocks of arbitrary length (up to half the state) can be absorbed without reducing preimage security. A disadvantage is that each message block must be retained while the state is being processed, but for SipHash this extra storage is only a quarter of the state.

Choice of Padding Rule. SipHash's padding appends a byte encoding the message length modulo 256. We could have chosen a slightly simpler padding rule, such as appending a `80` byte followed by zeroes. However, our choice forces messages of different lengths modulo 256 to have different last blocks, which may complicate attacks on SipHash; the extra cost is negligible.

5 Preliminary Cryptanalysis

We first consider attacks that are independent of the SipRound algorithm, and thus that are independent of the c and d parameters. We then consider attacks on SipRound iterations, with a focus on our proposal SipHash-2-4.

Key-Recovery. Brute force will recover a key after on average 2^{127} evaluations of SipHash, given two input/output pairs (one being insufficient to uniquely identify the key). The optimal strategy is to work with 1-word padded messages, so that evaluating SipHash-c-d takes $c + d$ SipRounds.

State-Recovery. A simple strategy to attack SipHash is to choose three input strings identical except for their last word, query for their respective SipHash outputs, and then "guess" the state that produced the output $v_0 \oplus v_1 \oplus v_2 \oplus v_3$ for one of the two strings. The attacker checks the 192-bit guessed value against the two other strings, and eventually recovers the key. On average $d2^{191}$ evaluations of SipRound are computed.

Internal Collisions. As for any MAC with 256-bit internal state, internal collisions can be exploited to forge valid tags with complexity of the order of 2^{128} queries to SipHash. The padding of the message length forces attackers to search for collisions at the same position modulo 256 bytes.

Truncated Differentials. To assess the strength of SipRound, we applied the same techniques that were used [2] to attack Salsa20, namely a search for statistical biases in one or more bits of output given one or more differences in the input. We considered input differences in v_3 and sought biases in $v_0 \oplus v_1 \oplus v_2 \oplus v_3$ after iterating SipRound.

The best results were obtained by setting a 1-bit difference in the most significant bit of v_3. After three iterations of SipRound many biases are found. But after four or more iterations we did not detect any bias after experimenting with sets of 2^{30} samples.

To attempt to distinguish our fast proposal SipHash-2-4 by exploiting such statistical biases, one needs to find a bias on six rounds such that no input difference lies in the most significant byte of the last word (as this encodes the message length).

XOR-Linearized Characteristics. We considered an attacker who injects a difference in the first message word processed by SipHash-2-4, and then that guesses the difference in v_3 every two SipRounds in order to cancel it with the new message word processed. This ensures that at least a quarter of the internal state is free of difference when entering a new absorption phase. Note that such an omniscient attacker would require the leakage of v_3 every two SipRounds, and thus is not covered by our security claims in §3.

We used Leurent's ARX toolkit [27] to verify that our characteristics contain no obvious contradiction, and to obtain refined probability estimates. Table 5.1 shows the best characteristic we found: after two rounds there are 20 bit differences in the internal state, with differences in all four words. The message injection reduces this to 15 bit differences (with no difference in v_3), and after two more rounds there are 96 bit differences. The probability to follow this differential characteristic is estimated to be 2^{-134}. For comparison, Table 5.2 shows the characteristic obtained with the same input difference, but for an attacker who does not guess the difference in v_3: the probability to follow four rounds of the characteristic is estimated to be 2^{-159}.

Better characteristics may exist. However we expect that finding (collections of) characteristics that both have a high probability and are useful to attack SipHash is extremely difficult. SipRound has as many additions as xors, so

Table 5.1. For each SipRound, differences in v_0, v_1, v_2, v_3 before each half-round in the xor-linear model. Every two rounds a message word is injected that cancels the difference in v_3; the difference used is then xored to v_0 after the two subsequent rounds. The probability estimate is given for each round, with the cumulative value in parentheses.

Round	Differences				Prob.
1	`................`	`................`	`................`	`8...............`	1 (1)
	`................`	`................`	`8...............`	`8...........8...`	
2	`8...........8...`	`8...............`	`........8.......`	`8.....1...1.8...`	13 (14)
	`....8...........`	`............9...`	`8.....1.8.1.8...`	`8.1.......1.....`	
3	`..1.8.....1.....`	`8.....11a.1.1...`	`8.1.1...8.....1.`	`................`	33 (47)
	`a...1...8.1.8.11`	`8.12b413a2......`	`8.1.1...8.....1.`	`8.1.1...8.....1.`	
4	`2.1.......1.8..1`	`6825e.1322.1..35`	`22....1....2a413`	`2........2..82.3`	87 (134)
	`22118.344835e.13`	`f4378453.2172d3.`	`.2....1..2.2261.`	`.2...21.8..1.61.`	
5	`a..1..24c834e4.3`	`fe918.6d5a74e34f`	`..15.b2.f6378443`	`................`	145 (279)
	`924..74c5e9.8.49`	`6e9d2b.7.e29f89e`	`..15.b2.f6378443`	`..15.b2.f6378443`	
6	`9255.c6ca8a7.4.a`	`38863c74.922a1e7`	`f81e7cdd6e882.27`	`f64bca9c2.c7.6ab`	160 (439)
	`a185a5edaad33.18`	`6d5db13cf5b942fd`	`.e55b6414e4f268c`	`c4c9968648e4d.c7`	

linearization with respect to integer addition seems unlikely to give much better characteristics than xor-linearization.

Vanishing Characteristics. A particularly useful class of differential characteristics is that of vanishing characteristics: those start from a non-zero difference and yield an internal state with no difference, that is, an internal collision. Vanishing characteristics obviously do not exist for any iteration of SipRound; one has to consider characteristics for the function consisting of SipRound iterations followed by $v_0 \oplus = \Delta$, with an input difference Δ in v_3.

No vanishing characteristic exists for one SipRound, as a non-zero difference always propagates to v_2. We ensured that no vanishing xor-linear characteristic exists for iterations of two, three, or four SipRounds, by attempting to solve the corresponding linear system. For sequences of two words, we ensured that no sparse vanishing characteristic exists.

Other Attacks. We briefly examine the applicability of other attacks to attack SipHash:

- Rotational attacks are differential attacks with respect to the rotation operator; see, e.g., [6, Section 4] and [23]. Due to the asymmetry in the initial state—at most half of the initial state can be rotation-invariant—rotational attacks are ineffective against SipHash.
- Cube attacks [26] exploit a low algebraic degree in the primitive attacked. Due to the rapid growth of the degree in SipHash, as in other ARX primitives, cube attacks are unlikely to succeed.
- Rebound attacks [28] are not known to be relevant for keyed primitives.

Table 5.2. For each SipRound, differences in v_0, v_1, v_2, v_3 before each half-round in the xor-linear model. Every two rounds a message with no difference is injected. The probability estimate is given for each half-round, with the cumulative value in parentheses.

Round	Differences				Prob.
1	....	....	....	8...............	1 (1)
	....	....	8...............	8...........8...	
2	8...........8...	8...............	........8.......	8.....1...1.8...	13 (14)
	....8...........	............9...	8.....1.8.1.8...	8.1.......1.....	
3	..1.8.....1.....	8.....11a.1.1...	8.1.1...8.....1.	8.1.82.......2..	42 (56)
	a...1...8.1.8.11	8.12b413a2......	....92..8....21.	82..92..82..82..	
4	22..82...21..211	e835621322.1.235	22...21.8.122613	621.c21.42..42.3	103 (159)
	2.11..24ca35e.13	66778453..57bd22	4.1.c...c212641.	82..82..8.11.6..	
5	a21182244a24e613	2ec144fcb8.115dd	c245d93226674453	e2.18..48a34a6.3	152 (311)
	f225f3ce8cd.c6d8	a44f51d8d.9e5616	2.445936ac53e25.	a.4.d3.2.a5...51	
6	52652.cc868.c689	27baa9d2d.e.fcd8	7ccdb44684.b.8ee	32246acc8cb4ce93	187 (498)
	566.3a5175df891e	2.e5d3.249fb3ea6	4ee9de8a.8bfc67d	2425523ec62cf459	

Fixed Point. Any iteration of SipRound admits a trivial distinguisher: the zero-to-zero fixed-point. This may make theoretical arguments based on the "ideal permutation" assumption irrelevant. But exploiting this property to attack SipHash seems very hard, for

1. Hitting the all-zero state, although easy to verify, is expected to be as hard as hitting any other predefined state;
2. The ability to hit a predefined state implies the ability to recover the key, that is, to completely break SipHash.

That is, the zero-to-zero fixed point cannot be a significant problem for SipHash, for if it were, SipHash would have much bigger problems.

6 Performance

Lower Bounds for a 64-Bit Implementation. SipRound involves 14 64-bit operations, so SipHash-2-4 involves 30 64-bit operations for each 8 bytes of input, i.e., 3.75 operations per byte. A CPU core with 2 64-bit arithmetic units needs at least 1.875 cycles per byte for SipHash-2-4, and a CPU core with 3 64-bit arithmetic units needs at least 1.25 cycles per byte for SipHash-2-4. A CPU core with 4 64-bit arithmetic units needs at least 1 cycle per byte, since SipRound does not always have 4 operations to perform in parallel.

The cost of finalization cannot be ignored for short messages. For example, for an input of length between 16 and 23 bytes, a CPU core with 3 64-bit arithmetic units needs at least 49 cycles for SipHash-2-4.

Table 6.1. Speed measurements of SipHash-2-4 for short messages

Data byte length		8	16	24	32	40	48	56	64
“bulldozer”	Cycles	124	141	156	171	188	203	218	234
	Cycles per byte	15.50	8.81	6.50	5.34	4.70	4.23	3.89	3.66
“ishmael”	Cycles	123	134	145	158	170	182	192	204
	Cycles per byte	15.38	8.38	6.00	4.94	4.25	3.79	3.43	3.19
“latour”	Cycles	135	144	162	171	189	207	216	225
	Cycles per byte	16.88	10.29	6.75	5.34	4.50	4.31	3.86	3.52

Lower Bounds for a 32-Bit Implementation. 32-bit architectures are common in embedded systems, with for example processors of the ARM11 family implementing the ARMv6 architecture. To estimate SipHash's efficiency on ARM11, we can directly adapt the analysis of Skein's performance by Schwabe, Yang, and Yang [**31**, §7], which observes that six 32-bit instructions are sufficient to perform a MIX transform. Since SipRound consists of four MIX transforms—the 32-bit rotate is transparent—we obtain 24 instructions per SipRound, that is, a lower bound of $3c$ cycles per byte for SipHash on long messages. This is 6 cycles per byte for SipHash-2-4. An input of length between 16 and 23 bytes needs at least 240 cycles.

Implementation Results. We wrote a portable C implementation of SipHash, and ran preliminary benchmarks on three machines:

- “bulldozer”, a Linux desktop equipped with a processor from AMD's last generation (FX-8150, 4 × 3600 MHz, “Zambezi” core), using `gcc 4.5.2`;
- “ishmael”, a Linux laptop equipped with a processor from AMD's previous generation (Athlon II Neo Mobile, 1700 MHz, “Geneva” core), using `gcc 4.6.3`.
- “latour”, a Linux desktop equipped with an older Intel processor (Core 2 Quad Q6600, 2394 MHz, “Kentsfield” core), using `gcc 4.4.3`.

We used compiler options `-O3 -fomit-frame-pointer -funroll-loops`.

On “bulldozer”, our C implementation of SipHash-2-4 processes long messages at a speed of 1.96 cycles per byte. On “ishmael”, SipHash-2-4 reaches 1.44 cycles per byte; this is due to the Athlon II's K10 microarchitecture having three ALUs, against only two for the more recent Bulldozer. Similar comments apply to “latour”. These speeds are close to the lower bounds reported in §6, with respective gaps of approximately 0.10 and 0.20 cycles per byte.

Table 6.1 reports speeds on short messages. For comparison, the fastest SHA-3 finalist on “bulldozer” (BLAKE-512) takes approximately 1072 cycles to process 8 bytes, and 1280 cycles to process 64 bytes.

Figure 6.1 compares our implementation of SipHash on “bulldozer” with the optimized C++ and C implementations of CityHash (version CityHash64) and SpookyHash (version ShortHash) on short messages, as well as with OpenSSL's

MD5 implementation. Similar relative performance is observed on the other machines considered.

One can see from these tables that SipHash-2-4 is extremely fast, and competitive with non-cryptographic hashes. For example, hashing 16 bytes takes 141 Bulldozer cycles with SipHash-2-4, against 82 and 126 for CityHash and SpookyHash, and 600 for MD5. Our conservative proposal SipHash-4-8 is still twice as fast as MD5.

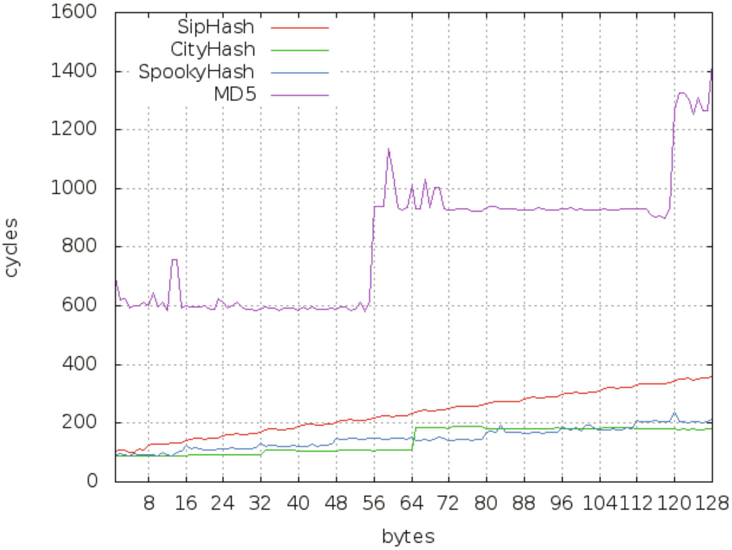

Fig. 6.1. Performance of SipHash-2-4 compared to non-cryptographic hash functions CityHash and SpookyHash and to MD5 on "bulldozer" (AMD FX-8150), for messages of $1, 2, \ldots, 128$ bytes. Curves on the right, from top to bottom, are MD5, SipHash, SpookyHash, and CityHash.

Automated Benchmarks. After the initial publication of SipHash, third-party applications were written in various programming languages, including C, C#, Javascript, Ruby, etc. In particular, Samuel Neves wrote optimized C implementations of SipHash compliant with the `crypto_auth` interface of the SUPERCOP benchmarking software. These implementations (`little`, `mmx`, `sse2-1`, `sse41`) as well as our reference implementation (`ref_le`) were added to SUPERCOP and benchmarked on various machines. A subset of the results are reported in Table 6.2.

Hardware Efficiency. ASICs can integrate SipHash with various degrees of area/throughput tradeoffs, with the following as extreme choices:

- **Compact architecture** with a circuit for a half-SipRound only, that is, two 64-bit full adders, 128 xors, and two rotation selectors. For SipHash-c-d this corresponds a latency of $c/4$ cycles per byte plus $2d$ cycles for the finalization.

Table 6.2. Performance of SipHash-2-4 on processors based on the `amd64` 64-bit architecture, in cycles per byte

Processor	Microarchitecture (core)	Long	64	8
AMD FX-8120	Bulldozer (Zambezi)	1.95	3.75	16.25
AMD E-450	Bobcat (Ontario)	2.03	4.88	22.88
AMD A8-3850	K10 (Llano)	1.44	3.61	26.50
AMD Athlon 64 X2	K8 (Windsor)	1.50	2.91	11.12
Intel Core i3-2310M	Sandy Bridge (206a7)	2.98	6.12	20.50
Intel Atom N435	Bonnell (Pineview)	2.19	4.50	20.00
Intel Xeon E5620	Nehalem (Westmere-EP)	1.63	2.81	11.50
Intel Core 2 Duo E8400	Core (Wolfdale)	1.69	3.38	13.50
VIA Nano U3500	Isaiah	2.38	4.53	17.50

- **High-speed architecture** with a circuit for $e = \max(c, d)$ rounds, that is, $4e$ 64-bit full adders and $256e$ xors. For SipHash-c-d this corresponds to a latency of 1/8 cycle per byte plus one cycle for finalization.

Both architectures require 256 D-flip-flops to store the internal state, plus 64 for the message blocks. For a technology with 8 gate-equivalents (GE) per full adder, 3 per xor, and 7 per D-flip-flop, this is a total of approximately 3700 GE for the compact architecture of SipHash-2-4, and 13500 GE for the high-speed architecture. With the compact architecture a 20-byte message is hashed by SipHash-2-4 in 20 cycles, against 4 cycles with the high-speed architecture. An architecture implementing $c = 2$ rounds of SipHash-2-4 would take approximately 7900 GE to achieve a latency of 1/8 cycles per byte plus two cycles for finalization, thus 5 cycles to process 20 bytes.

7 Application: Defense against Hash Flooding

We propose that hash tables switch to SipHash as a hash function. On startup a program reads a secret SipHash key from the operating system's cryptographic random-number generator; the program then uses SipHash for all of its hash tables. This section explains the security benefits of SipHash in this context.

The small state of SipHash also allows each hash table to have its own key with negligible space overhead, if that is more convenient. Any attacks must then be carried out separately for each hash table.

Review of Hash Tables. Storing n strings in a linked list usually takes a total of $\Theta(n^2)$ operations, and retrieving one of the n strings usually takes $\Theta(n)$ operations. This can be a crippling performance problem when n is large.

Hash tables are advertised as providing much better performance. The simplest type of hash table contains ℓ separate linked lists $L[0], L[1], \ldots, L[\ell - 1]$ and stores each string m inside the linked list $L[H(m) \bmod \ell]$, where H is a hash function and ℓ is a power of 2. Each linked list then has, on average, only n/ℓ strings. Normally this improves performance by a factor close to n if ℓ is chosen to be on the same scale as n: storing n strings usually takes only $\Theta(n)$ operations and retrieving a string usually takes $\Theta(1)$ operations.

There are other data structures that guarantee, e.g., $O(n \lg n)$ operations to store n strings and $O(\lg n)$ operations to retrieve one string. These data structures avoid all of the security problems discussed below. However, hash tables are perceived as being simpler and faster, and as a result are used pervasively throughout current programming languages, libraries, and applications.

Review of Hash Flooding. Hash flooding is a denial-of-service attack against hash tables. The attacker provides n strings m that have the same hash value $H(m)$, or at least the same $H(m) \bmod \ell$. The hash-table performance then deteriorates to the performance of one linked list.

The name "hash flooding" for this attack appeared in 1999, in the source code for the first release of the `dnscache` software from the second author of this paper:

```
    if (++loop > 100) return 0; /* to protect against hash flooding */
```

This line of code protects `dnscache` against the attack by limiting each linked list to 100 entries. However, this is obviously not a general-purpose solution to hash flooding. Caches can afford to throw away unusual types of data, but most applications need to store all incoming data.

Crosby and Wallach reintroduced the same attack in 2003 under the name "algorithmic complexity attack" [13] and explored its applicability to the Squid web cache, the Perl programming language, etc. Hash flooding made headlines again in December 2011, when Klink and Wälde [24] demonstrated its continued applicability to several commonly used web applications. For example, Klink and Wälde reported 500 KB of carefully chosen POST data occupying a PHP5 server for a full minute of CPU time.

Advanced Hash Flooding. Crosby and Wallach recommended replacing public functions H with secret functions, specifically universal hash functions, specifically the hash function $H(m_0, m_1, \ldots) = m_0 \cdot k_0 + m_1 \cdot k_1 + \cdots$ using a secret key $(k_0, k_1, \ldots)$. The idea is that an attacker has no way to guess which strings will collide.

We question the security of this approach. Consider, for example, a hash table containing one string m, where m is known to the attacker. Looking up another string m' will, with standard implementations, take longer if $H(m') \equiv H(m) \pmod{\ell}$ than if $H(m') \not\equiv H(m) \pmod{\ell}$. This timing information will often be visible to an attacker, and can be amplified beyond any level of noise if the application allows the attacker to repeatedly query m'. By guessing ℓ choices of strings $m' \neq m$ the attacker finds one with $H(m') \equiv H(m) \pmod{\ell}$. The linearity of the Crosby–Wallach choice of H then implies that adding any multiple of $m' - m$ to m will produce another colliding string. With twice as many guesses the attacker finds an independent string m'' with $H(m'') \equiv H(m) \pmod{\ell}$; then adding any combination of multiples of $m' - m$ and $m'' - m$

to m will produce even more collisions. With a moderate number of guesses the attacker finds enough information to solve for $(k_0 \bmod \ell, k_1 \bmod \ell, \ldots)$ by Gaussian elimination, and easily computes any number of strings with the same hash value.

One can blame the hash-table implementation for leaking information through timing; but it is not easy to build an efficient constant-time hash table. Even worse, typical languages and libraries allow applications to see all hash-table entries in order of hash value, and applications often expose this information to attackers. One could imagine changing languages and libraries to sort hash-table entries before enumerating them, but this would draw objections from applications that need the beginning of the enumeration to start quickly. One could also imagine changing applications to sort hash-table entries before exposing them to attackers, but ensuring this would require reviewing code in a huge number of applications.

We comment that many of the hash-flooding defenses proposed since December 2011 are vulnerable to the same attack. The most common public hash functions are of the form $m_0 \cdot k_0 + m_1 \cdot k_1 + \cdots$ where $k_0, k_1, \ldots$ are public, and many of the proposed defenses simply add some entropy to $k_0, k_1, \ldots$; but the attack works no matter how $k_0, k_1, \ldots$ are chosen. Many more of the proposed defenses are minor variations of this linear pattern and are broken by easy variants of the same attack.

We do not claim novelty for observing how much damage a single equation $H(m') \equiv H(m) \pmod{\ell}$ does to the unpredictability of this type of hash function; see, e.g., the attacks in [9] and [19] against related MACs. However, the fact that hash tables leak such equations through side channels does not seem to be widely appreciated.

Stopping Advanced Hash Flooding. The worst possible exposure of hash-table indices would simply show the attacker $H(m) \bmod \ell$ for any attacker-selected string m. We advocate protecting against this maximum possible exposure, so that applications do not have to worry about how much exposure they actually provide. The attacker's goal, given this exposure, is to find many strings m having a single value $H(m) \bmod \ell$.

We propose choosing H to be a cryptographically strong PRF. If H is a strong PRF then the truncation $H \bmod \ell$ is also a strong PRF (recall that ℓ is a power of 2), and therefore a strong MAC: even after seeing $H(m) \bmod \ell$ for selected strings m, the attacker cannot predict $H(m) \bmod \ell$ for any other string m. The strength of H as a PRF implies the same unpredictability even if the attacker is given hash values $H(m)$, rather than just hash-table indices $H(m) \bmod \ell$. Achieving this level of unpredictability does not appear to be significantly easier than achieving the full strong-PRF property.

Typical hash-table applications hash a large number of short strings, so the performance of H on short inputs is critical. We therefore propose choosing SipHash as H: we believe that SipHash is a strong PRF, and it provides excellent performance on short inputs. There are previous hash functions with competitive performance, and there are previous functions that have been proposed and evaluated for the same security standards, but as far as we know SipHash is the first function to have both of these features.

Of course, the attacker's inability to predict new hash values does not stop the attacker from exploiting old hash values. No matter how strong H is, the attacker will find two colliding strings after (on average) about $\sqrt{\ell}$ guesses, and then further strings with the same hash value for (on average) ℓ guesses per collision. However, finding n colliding strings in this way requires the attacker to communicate about $n\ell \approx n^2$ strings, so n—the CPU amplification factor of the denial-of-service attack—is limited to the square root of the volume of attacker communication. For comparison, weak secret hash functions and (weak or strong) public hash functions allow n to grow linearly with the volume of attacker communication. A strong secret hash function thus greatly reduces the damage caused by the attack.

The Python Hash Function. Versions 2.7.3 and 3.2.3 of the Python programming language (released in April 2012) introduced an option `-R` with the goal of protecting against hash flooding. According to the Python manual, this option "[turns] on 'hash randomization', so that the hash() values of str, bytes and datetime objects are 'salted' with an unpredictable pseudo-random value. ... This is intended to provide protection against a denial of service caused by carefully-chosen inputs ...". It is therefore expected that outputs of this hash function are unpredictable to parties who were not given the secret key (the salt); obviously, if this key is known, outputs are no longer unpredictable.

We point out that the keyed hashing introduced in Python 2.7.3 and 3.2.3 does not behave as an unpredictable function: the 128-bit key can be recovered efficiently given only two outputs of the keyed hash. The internal state contains only 64 bits, so multicollisions can be found efficiently with a meet-in-the-middle strategy once the key is known.

A proof-of-concept Python script is given in Appendix B. We verified our attack on Python 2.7.3 and 3.2.3, in each case successfully recovering the per-process key.

References

[1] — (no editor): 20th annual symposium on foundations of computer science. IEEE Computer Society, New York (1979). MR 82a:68004. See [33]

[2] Aumasson, J.-P., Fischer, S., Khazaei, S., Meier, W., Rechberger, C.: New features of Latin dances: analysis of Salsa, ChaCha, and Rumba. In: FSE 2008 [**29**], pp. 470–488 (2008), `http://eprint.iacr.org/2007/472`. Citations in this document: §5

[3] Aumasson, J.-P., Henzen, L., Meier, W., Phan, R.C.-W.: SHA-3 proposal BLAKE (version 1.3) (2010), `https://www.131002.net/blake/blake.pdf`. Citations in this document: §4

[4] Bernstein, D.J.: Floating-point arithmetic and message authentication (2004), `http://cr.yp.to/papers.html#hash127`. Citations in this document: §1

[5] Bernstein, D.J.: The Poly1305-AES message-authentication code. In: [**17**], pp. 32–49 (2005), `http://cr.yp.to/papers.html#poly1305`. Citations in this document: §1

[6] Bernstein, D.J.: Salsa20 security. eSTREAM report 2005/025 (2005), http://cr.yp.to/snuffle/security.pdf. Citations in this document: §5

[7] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The Keccak reference (version 3.0) (2011), http://keccak.noekeon.org/Keccak-reference-3.0.pdf. Citations in this document: §4

[8] Biham, E., Youssef, A.M. (eds.): Selected areas in cryptography, 13th international workshop, SAC 2006, Montreal, Canada, August 17–18, 2006, revised selected papers. LNCS, vol. 4356. Springer (2007). ISBN 978-3-540-74461-0. See [25]

[9] Black, J., Cochran, M.: MAC reforgeability. In: FSE 2009 [**15**], pp. 345–362 (2009), http://eprint.iacr.org/2006/095. Citations in this document: §7

[10] Black, J., Halevi, S., Krawczyk, H., Krovetz, T., Rogaway, P.: UMAC: fast and secure message authentication. In: Crypto '99 [**35**], pp. 216–233 (1999), http://fastcrypto.org/umac/umac_proc.pdf. Citations in this document: §1

[11] Black, J., Halevi, S., Krawczyk, H., Krovetz, T., Rogaway, P.: Update on UMAC fast message authentication (2000), http://fastcrypto.org/umac/update.pdf. Citations in this document: §1, §1

[12] Blahut, R.E., Costello Jr., D.J., Maurer, U., Mittelholzer, T. (eds.): Communications and cryptography: two sides of one tapestry. Springer (1994). See [26]

[13] Crosby, S.A., Wallach, D.S.: Denial of service via algorithmic complexity attacks. 12th USENIX Security Symposium (2003), http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003.pdf. Citations in this document: §7

[14] Dai, W., Krovetz, T.: VHASH security (2007), http://eprint.iacr.org/2007/338. Citations in this document: §1

[15] Dunkelman, O. (ed.): Fast software encryption, 16th international workshop, FSE 2009, Leuven, Belgium, February 22–25, 2009, revised selected papers. LNCS, vol. 5665. Springer (2009). ISBN 978-3-642-03316-2. See [9], [28]

[16] Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein hash function family (version 1.1) (2008), http://www.skein-hash.info/sites/default/files/skein1.1.pdf. Citations in this document: §4

[17] Gilbert, H., Handschuh, H. (eds.): Fast software encryption: 12th international workshop, FSE 2005, Paris, France, February 21–23, 2005, revised selected papers. LNCS, vol. 3557. Springer (2005). ISBN 3-540-26541-4. See [5]

[18] Google: The CityHash family of hash functions (2011), https://code.google.com/p/cityhash/. Citations in this document: §1

[19] Handschuh, H., Preneel, B.: Key-recovery attacks on universal hash function based MAC algorithms. In: CRYPTO 2008 [**32**], pp. 144–161 (2008), http://www.cosic.esat.kuleuven.be/publications/article-1150.pdf. Citations in this document: §7

[20] Hong, S., Iwata, T.: Fast software encryption, 17th international workshop, FSE 2010, Seoul, Korea, February 7–10, 2010, revised selected papers. LNCS, vol. 6147. Springer (2010). ISBN 978-3-642-13857-7. See [23]

[21] Jenkins, B.: SpookyHash: a 128-bit noncryptographic hash (2010), http://burtleburtle.net/bob/hash/spooky.html. Citations in this document: §1

[22] Jenkins, B.: Issue 4: CityHash128 isn't thorough enough (2011), https://code.google.com/p/cityhash/issues/detail?id=4&can=1. Citations in this document: §1

[23] Khovratovich, D., Nikolic, I.: Rotational cryptanalysis of ARX. In: FSE 2010 [**20**], pp. 333–346 (2010), http://www.skein-hash.info/sites/default/files/axr.pdf. Citations in this document: §5

[24] Klink, A., Wälde, J.: Efficient denial of service attacks on web application platforms (2011), http://events.ccc.de/congress/2011/Fahrplan/events/4680.en.html. Citations in this document: §7

[25] Krovetz, T.: Message authentication on 64-bit architectures. In: [8], pp. 327–341 (2007), http://eprint.iacr.org/2006/037. Citations in this document: §1

[26] Lai, X.: Higher order derivatives and differential cryptanalysis. In: [12], pp. 227–233 (1994). Citations in this document: §5

[27] Leurent, G.: The ARX toolkit (2012), http://www.di.ens.fr/~leurent/arxtools.html. Citations in this document: §5

[28] Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: The rebound attack: cryptanalysis of reduced Whirlpool and Grøstl. In: FSE 2009 [15], pp. 260–276 (2009), http://www2.mat.dtu.dk/people/S.Thomsen/MendelRST-fse09.pdf. Citations in this document: §5

[29] Nyberg, K. (ed.): Fast software encryption, 15th international workshop, FSE 2008, Lausanne, Switzerland, February 10–13, 2008, revised selected papers. LNCS, vol. 5086. Springer (2008). ISBN 978-3-540-71038-7. See [2]

[30] Paul, S.: Improved indifferentiability security bound for the JH mode. Third SHA-3 Conference (2012), http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/March2012/documents/papers/PAUL_paper.pdf. Citations in this document: §4

[31] Schwabe, P., Yang, B.-Y., Yang, S.-Y.: SHA-3 on ARM11 processors. In: Proceedings of Africacrypt 2012, to appear (2012), http://cryptojedi.org/papers/sha3arm-20120422.pdf. Citations in this document: §6

[32] Wagner, D. (ed.): Advances in cryptology—CRYPTO 2008, 28th annual international cryptology conference, Santa Barbara, CA, USA, August 17–21, 2008, proceedings. LNCS, vol. 5157. Springer (2008). ISBN 978-3-540-85173-8. See [19]

[33] Wegman, M.N., Lawrence Carter, J.: New classes and applications of hash functions. In: [1], pp. 175–182 (1979); see also newer version [34], http://cr.yp.to/bib/entries.html#1979/wegman

[34] Wegman, M.N., Lawrence Carter, J.: New hash functions and their use in authentication and set equality. Journal of Computer and System Sciences 22, 265–279 (1981); see also older version [33]. ISSN 0022-0000. MR 82i:68017, http://cr.yp.to/bib/entries.html#1981/wegman. Citations in this document: §1

[35] Wiener, M. (ed.): Advances in cryptology—CRYPTO '99. LNCS, vol. 1666. Springer (1999). ISBN 3-5540-66347-9. MR 2000h:94003. See [10]

[36] Wu, H.: The hash function JH (2011), http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf. Citations in this document: §4

A Test Values

This appendix shows intermediate values of SipHash-2-4 hashing the 15-byte string 000102···0c0d0e with the 16-byte key 000102···0d0e0f.

Initialization little-endian reads the key as

$$k_0 = \texttt{0706050403020100}$$
$$k_1 = \texttt{0f0e0d0c0b0a0908}$$

The key is then xored to the four constants to produces the following initial state (v_0 to v_3, left to right):

```
7469686173716475 6b617f6d656e6665 6b7f62616d677361 7b6b696e727e6c7b
```

The first message block 0706050403020100 is xored to v_3 to give

```
7469686173716475 6b617f6d656e6665 6b7f62616d677361 7c6d6c6a717c6d7b
```

and after two SipRounds the internal state is:

```
4d07749cdd0858e0 0d52f6f62a4f59a4 634cb3577b01fd3d a5224d6f55c7d9c8
```

Xoring the first message block to v_0 concludes the compression phase:

```
4a017198de0a59e0 0d52f6f62a4f59a4 634cb3577b01fd3d a5224d6f55c7d9c8
```

The second and last block is the last seven message bytes followed by the message's length, that is, 0f0e0d0c0b0a0908. After xoring this block to v_3, doing two SipRounds, xoring it to v_0 and xoring 00000000000000ff to v_2, the internal state is

```
3c85b3ab6f55be51 414fc3fb98efe374 ccf13ea527b9f442 5293f5da84008f82
```

After the four iterations of SipRound, the internal state is

```
f6bcd53893fecff1 54b9964c7ea0d937 1b38329c099bb55a 1814bb89ad7be679
```

and the four words are xored together to return a129ca6149be45e5.

B Computing the Key of the Python Hash Function

The Python script below can be used to compute the key used in the function hash() of a Python process. An example of usage is

```
$ python3.2 -R poc.py
128 candidate solutions
verified solution: 58df0aca50e7f48b 141f57f820cbfefe
verified solution: d8df0aca50e7f48b 941f57f820cbfefe
```

Note the equivalent keys, and the -R option.

```
solutions = []
mask = 0xffffffffffffffff

def bytes_hash( p, prefix, suffix ):
  if len(p) == 0: return 0
  x = prefix ^ (ord( p[0] )<<7)
  for i in range( len(p) ):
    x = ( ( x * 1000003 ) ^ ord(p[i]) ) & mask
  x ^= len(p) ^ suffix
  if x == -1: x = -2
  return x

def solvebit( h1, h2, prefix, bits ):
  f1 = 1000003
  f2 = f1*f1
  target = h1^h2^3
  if bits == 64:
    if ((f1*prefix)^(f2*prefix)^target) & mask: return
    suffix = h1^1^(f1*prefix)
    suffix&= mask
    solutions.append( (prefix,suffix) )
  else:
    if ((f1*prefix)^(f2*prefix)^target) & ((1<<bits)-1):
      return
    solvebit(h1,h2,prefix,bits + 1)
    solvebit(h1,h2,prefix + (1 << bits),bits + 1)
  pass

h1 = hash("\0")     & mask
h2 = hash("\0\0")   & mask
h3 = hash("python") & mask

solvebit( h1, h2, 0, 0 )

print("%d␣candidate␣solutions" % (len(solutions)))
for s in solutions:
  if bytes_hash("python",s[0],s[1]) == hash("python") & mask:
    ok=1
    for i in range(10)[1:]:
      if bytes_hash("\2"*i,s[0],s[1]) != hash("\2"*i) & mask:
        ok=0
    if ok: print("solution:␣%016x␣%016x" % (s[0],s[1]))
```