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., $t = h(k_1, m) \oplus \mathrm{AES}(k_2, n)$ 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].)