The EARWORM Password Hashing Algorithm

Password Hashing Competition Submission Document (Round 1)

Daniel Franke <dfoxfranke@gmail.com>

March 31, 2014

Contents

1 Overview

1.1 Required statements

No deliberately-hidden weaknesses

The attacks against EARWORM discussed in this submission document are the strongest that are known to its author. EARWORM contains no back doors or other deliberately-hidden weaknesses.

Intellectual property statement

EARWORM is, and will remain, available worldwide on a royalty-free basis.

EARWORM utilizes SHA-256, whose implementation is subject to U.S. Patent 6,829,355. This patent was issued to The United States of America as represented by the National Security Agency on December 7, 2004. The National Security Agency has made U.S. Patent 6,829,355 available royalty-free [1].

The designer is unaware of any other patent or patent application that covers the use or implementation of EARWORM.

1.2 Design goals

One-wayness

It is a design goal of EARWORM that it should satisfy the standard cryptographic notion of a one-way function. In practical terms, this means that a dictionary attack should be the most efficient way to recover the preimage of an EARWORM hash.

CPU-friendliness

It is a design goal of EARWORM that a commodity x86-64 PC, equipped with support for the AES-NI instruction set but no other distinguishing features (particularly, no GPGPU support), should constitute, as nearly as possible, the maximally economically-efficient platform for computing EARWORM hashes, measured on any of three axes:

1. Latency-fixed-cost: No device cheaper than a typical x86-64-based server should be capable of performing a single EARWORM computation with significantly less latency.
2. Throughput-fixed-cost: No device cheaper than a typical x86-64-based server should be capable of performing EARWORM computations at significantly greater throughput.
3. Operating cost: No device should be capable of performing EARWORM computations at significantly lower operating cost (including electricity, cooling, depreciation, etc.) than a typical x86-64-based server.

Security at low time-cost

A password hash can be made expensive to attack by requiring significant time, significant memory, or both; the defender must trade off computational burden against security margin. EARWORM aims to maximize the flexibility of this trade-off, particularly in the direction of low time and high memory. By taking a large, pre-initialized array as an input, EARWORM can impose an arbitrarily-high memory cost with an arbitrarily-low time cost, potentially requiring more memory than it is possible to address in the duration of a single hash computation.

1.3 Non-goals

Second-preimage resistance

EARWORM is not designed to resist second-preimage or (by implication) collision attacks.

Sequential memory-hardness

EARWORM is not a sequential memory-hard function as defined in [2]. It instead derives its difficulty from its demand on memory bandwidth, or, in the terminology of [3], "ROM-port hardness".

Operation on constrained platforms

EARWORM is intended to be implemented on servers. Its dependence on a large ROM and AES-NI makes it impractical to offload EARWORM computations to constrained platforms such as browsers or smartphones.

Integration with hardware security modules

EARWORM provides no explicit support for integration with hardware security modules (HSMs). This does not imply that no such integration is possible. For example, [4] proposes a scheme for integrating a KDF with a YubiHSM[1] in a manner that is independent of the choice of KDF. Given the existence of this scheme, and the ease with which similar ones can be invented, I perceive no benefit to complicating the design of EARWORM by explicitly accounting for such considerations.

DDoS mitigation

Prevention of online dictionary attacks without denying service to legitimate users remains an open problem. Rate-limiting login attempts on a per-user basis makes it trivial to maliciously lock users out of their accounts. Rate-limiting on a per-IP-address basis, already a dubious proposition in 2014, will grow increasingly useless as the world migrates toward IPv6 and IP addresses cease to become a scare resource.

Expensive password hashes like EARWORM, intended to frustrate offline attacks, make online attacks even more problematic than before because they now threaten to deny service to *all* of a site's users by by exhausting the server's computational resources. EARWORM attempts no new contributions toward addressing this problem.

My comments on [5] (which I wrote prior to any of my work on password hashing) suggest JavaScript-based proof-of-work systems as a possible solution. Any approach along these lines, should one be found practical, can treat password-hashing schemes as an entirely orthogonal consideration.

Applicability as anything other than a password hash

EARWORM is strictly a password hash and is not designed to be suitable for any other purpose, such as key derivation.

[1] `https://www.yubico.com/products/yubihsm/`

2 Specification

2.1 Notation

This specification deals in four sorts of mathematical object: natural numbers, octet strings, arrays, and first-order functions.

Literal natural numbers will be represented in base ten and typeset in ordinary numerals. The notation $x : \mathbb{N}$ indicates that the variable x ranges over the natural numbers. Operations defined on natural numbers include the basic arithmetic functions of addition ($+$), subtraction ($-$), multiplication ($\cdot$), division ($/$), and modulus (mod). In all cases where subtraction is used, the difference will always be positive. In all cases where division is used, the divisor will always evenly divide the dividend. Moduli are non-negative, *i.e.*, $n \bmod m$ ranges from 0 to $m - 1$ inclusive. Natural numbers are defined with infinite range — arithmetic operations do not overflow — but in practice no natural number larger than $2^{128} - 1$ needs to be handled, and all arithmetic operations can be implemented easily and efficiently on any 32-bit platform, without the need for any sophisticated BIGNUM algorithms.

Literal octet strings will be represented in base sixteen and typeset in `monospace` with spaces between each octet. The notation $x : \mathbb{B}^n$ indicates that the variable x ranges over the set of octet strings of length n. The notation $x : \mathbb{B}^*$ indicates that the variable x ranges over the set of all octet strings (including the zero-length string). $|x|$ denotes the length of an octet string x. Operations defined on octet strings include bitwise exclusive-or ($\oplus$), concatenation ($\|$), and slicing. Xor is defined only on octet strings of equal length. For slicing, the notation $x_{m..n}$ represents the substring of x consisting of the mth thru nth octets, zero-indexed and inclusive of both endpoints. The zeroth octet is the one in the leftmost position of a string. Strings will never be decomposed into units smaller than a single octet, so "bit-endianness" is left undefined. The notation $(\texttt{00})^n$ represents an all-zero octet string of length n. The notation `nil` represents the zero-length octet string.

Array elements are either natural numbers or octet strings. The notation $x : \mathbb{N}[m][n]$ indicates that the variable x ranges over the set of $m \times n$ arrays of natural numbers. The notation $x[a][b]$, where $0 \leq a < m$ and $0 \leq b < n$, represents the natural number appearing in the ath row and bth column of that array.

Function names will be represented in SMALL CAPS. The notation $\textsc{f}(x, y)$ represents the result of calling the function F with the arguments x and y. Functions are pure, obviating any distinction between call-by-value and call-by-reference.

The boundaries of for loops are inclusive of both endpoints. All other pseudocode constructs hopefully are self-explanatory.

2.2 Encoding and decoding of natural numbers

The function

$$\text{BE32DEC}(a : \mathbb{B}^4) : \mathbb{N}$$

converts its argument to a natural number in the range $0..2^{32}-1$ by interpreting it in big-endian order. Examples follow.

$$\begin{aligned}
\text{BE32DEC}(\texttt{00 00 00 00}) &= 0 \\
\text{BE32DEC}(\texttt{00 00 00 0f}) &= 15 \\
\text{BE32DEC}(\texttt{00 00 01 00}) &= 256 \\
\text{BE32DEC}(\texttt{ff ff ff ff}) &= 4294967295
\end{aligned}$$

The function

$$\text{BE32ENC}(n : \mathbb{N}) : \mathbb{B}^4$$

is the inverse of BE32DEC, converting a natural number in the domain $0..2^{32}-1$ to a big-endian octet string.

$$\begin{aligned}
\text{BE32ENC}(0) &= \texttt{00 00 00 00} \\
\text{BE32ENC}(15) &= \texttt{00 00 00 0f} \\
\text{BE32ENC}(256) &= \texttt{00 00 01 00} \\
\text{BE32ENC}(4294967295) &= \texttt{ff ff ff ff}
\end{aligned}$$

The functions

$$\begin{aligned}
\text{BE64DEC}(a : \mathbb{B}^8) &: \mathbb{N} \\
\text{BE128DEC}(a : \mathbb{B}^{16}) &: \mathbb{N}
\end{aligned}$$

are analogous to BE32DEC but take as input octet strings of length 8 and 16 respectively, returning output in the range $0..2^{64}-1$ and $0..2^{128}-1$ respectively. Their inverses are

$$\begin{aligned}
\text{BE64ENC}(n : \mathbb{N}) &: \mathbb{B}^8 \\
\text{BE128ENC}(n : \mathbb{N}) &: \mathbb{B}^{16}
\end{aligned}$$

2.3 The EWPRF function

The functions

$$\text{SHA256}(in : \mathbb{B}^*) : \mathbb{B}^{32}$$

and

$$\text{HMAC-SHA256}(key : \mathbb{B}^*, in : \mathbb{B}^*) : \mathbb{B}^{32}$$

are as defined in [6].

The function

$$\text{PBKDF2}_{\text{PRF}(key:\mathbb{B}^*,in:\mathbb{B}^*):\mathbb{B}^*}(P : \mathbb{B}^*, S : \mathbb{B}^*, c : \mathbb{N}, dkLen : \mathbb{N}) : \mathbb{B}^{dkLen}$$

is as defined in [7].

The function

$$\textsc{ewprf}(secret : \mathbb{B}^*, salt : \mathbb{B}^*, outlen : \mathbb{N}) : \mathbb{B}^{outlen}$$

is defined as

$$\text{PBKDF2}_{HMAC-SHA256}(secret, salt, 1, outlen)$$

where *outlen* must not exceed $32 \cdot (2^{32} - 1)$.

$\textsc{ewprf}(\text{“passwd”}, \text{“salt”}, 64) =$

```
55 ac 04 6e 56 e3 08 9f ec 16 91 c2 25 44 b6 05
f9 41 85 21 6d de 04 65 e6 8b 9d 57 c2 0d ac bc
49 ca 9c cc f1 79 b6 45 99 16 64 b3 9d 77 ef 31
7c 71 b8 45 b1 e3 0b d5 09 11 20 41 d3 a1 97 83
```

where “passwd” and “salt” represent the octet strings

```
70 61 73 73 77 64
```

and

```
73 61 6c 74
```

respectively. This test vector comes from [8].

N.b.: In the reference implementation, EWPRF is just called PRF. The EW prefix is included here in order to clearly disambiguate EWPRF from the PBKDF2 parameter named PRF.

2.4 The AESRound function

The AESRound function computes a single internal round of AES encryption. Before specifying it further, a slight digression to take care of some semantic housekeeping is necessary. Inconveniently for our purposes, the AES specification, FIPS-197, makes extensive use of impure “functions” which mutate their arguments, which this document seeks to avoid dealing with. For example, FIPS-197's SubBytes function has no return value, but takes a pointer argument named *state*, and the 16-byte memory region to which it points gets modified. The ShiftRows and MixColumns functions behave similarly. Here instead we let these functions return the updated state rather than modify their argument in-place. Thus their new signatures are

$$\textsc{SubBytes}(state : \mathbb{B}^{16}) : \mathbb{B}^{16}$$
$$\textsc{ShiftRows}(state : \mathbb{B}^{16}) : \mathbb{B}^{16}$$
$$\textsc{MixColumns}(state : \mathbb{B}^{16}) : \mathbb{B}^{16}$$

FIPS-197's AddRoundKey function is denoted here simply as xor ($\oplus$).

The function

$$\text{AESRound}(roundKey : \mathbb{B}^{16}, block : \mathbb{B}^{16}) : \mathbb{B}^{16}$$

is defined as

$$\text{MixColumns}(\text{ShiftRows}(\text{SubBytes}(block))) \oplus roundKey.$$

The operation of AESRound is identical to that of the Intel® `AESENC` instruction. The following test vector is taken from [9]:

AESRound(`5d 6e 6f 72 65 75 47 5b 29 79 61 68 53 28 69 48`,
`5d 47 53 5d 72 6f 74 63 65 56 74 73 65 54 5b 7b`) =
`95 e5 d7 de 58 4b 10 8b c5 a3 db 9f 2f 1c 31 a8`

2.5 The WORKUNIT function

The WORKUNIT function is the "meat" of EARWORM. Its specification depends on three constants:

1. W is the "chunk width". It determines how much internal parallelism is available. This constant is set at $W = 4$.
2. L is the "chunk length". With W, it determines how many sequential memory accesses occur after each random access. This constant is set at $L = 64$. The product LW is called the "chunk area".
3. D is the "workunit depth". It determines how many random memory accesses occur per workunit. This constant is set at $D = 256$.

WORKUNIT takes as input a secret (passphrase), a salt, the desired output length, a memory cost parameter, and a site-local parameter called the *arena*. The arena is a large, randomly-initialized, read-only array of $2^{m\_cost} \times L \times W$ 128-bit blocks used as AES round keys. The necessity of storing the arena and providing high-bandwidth access to it is what makes EARWORM hashes expensive to compute.

WORKUNIT is defined here for secrets and salts of arbitrary length, but implementations may impose length limits. The reference implementation of EARWORM accepts secrets of unlimited length (up to the amount of addressable memory on the target platform), but limits the salt to 36 octets. This limitation allows it to avoid dynamic memory allocation within the WORKUNIT function. Furthermore, due to a limitation inherited from PBKDF2, *outlen* must not exceed $32 \cdot (2^{32} - 1)$. *m_cost* must not exceed 128; implementations may, and likely must, impose a much smaller limit.

1: **procedure** WORKUNIT(
secret : $\mathbb{B}^*$,
salt : $\mathbb{B}^*$,

```
        m_cost : ℕ,
        arena : 𝔹^16[2^m_cost][L][W],
        outlen : ℕ )
2:    var index_a : ℕ
3:    var index_b : ℕ
4:    var index_tmpbuf : 𝔹^32
5:    var scratchpad : 𝔹^16[W]
6:    var scratchpad_tmpbuf : 𝔹^16W
7:
8:    index_tmpbuf ← EWPRF(secret, 00‖salt, 32)
9:    index_a ← BE128DEC(index_tmpbuf_0..15) mod 2^m_cost
10:   index_b ← BE128DEC(index_tmpbuf_16..31) mod 2^m_cost
11:
12:   scratchpad_tmpbuf ← EWPRF(secret, 01‖salt, 16W)
13:   for i from 0 to W − 1 do
14:       scratchpad[i] ← scratchpad_tmpbuf_16i..16i+15
15:   end for
16:
17:   for d from 0 to D/2 − 1 do
18:       for l from 0 to L − 1 do
19:           for w from 0 to W − 1 do
20:               scratchpad[w] ←
                      AESROUND(arena[index_a][l][w], scratchpad[w])
21:           end for
22:       end for
23:       index_a ← BE128DEC(scratchpad[0]) mod 2^m_cost
24:       for l from 0 to L − 1 do
25:           for w from 0 to W − 1 do
26:               scratchpad[w] ←
                      AESROUND(arena[index_b][l][w], scratchpad[w])
27:           end for
28:       end for
29:       index_b ← BE128DEC(scratchpad[0]) mod 2^m_cost
30:   end for
31:
32:   for i from 0 to W − 1 do
33:       scratchpad_tmpbuf_16i..16i+15 ← scratchpad[i]
34:   end for
35:   return EWPRF(scratchpad_tmpbuf, 02‖salt, outlen)
36: end procedure
```

2.6 The EARWORM function

The complete EARWORM function computes *t_cost* workunits with distinct salts, and xor's together their results.

```
1: procedure EARWORM(
```

$secret : \mathbb{B}^*$,
$salt : \mathbb{B}^*$,
$t\_cost : \mathbb{N}$,
$m\_cost : \mathbb{N}$,
$arena : \mathbb{B}^{16}[2^{m\_cost}][L][W]$,
$outlen : \mathbb{N}$)

2: **var** $out : \mathbb{B}^{outlen}$
3: $out \leftarrow (\texttt{00})^{outlen}$
4: **for** i **from** 0 **to** $t\_cost - 1$ **do**
5: $out \leftarrow out \oplus \textsc{workunit}(secret, \textsc{be32enc}(i) \| salt, m\_cost, arena, outlen)$
6: **end for**
7: **return** out
8: **end procedure**

t_cost must not exceed $2^{32} - 1$, $outlen$ must not exceed $32 \cdot (2^{32} - 1)$, and m_cost must not exceed 128. Implementations may impose further limits on m_cost and the length of $secret$ and $salt$; the reference implementation limits $salt$ to 32 octets.

2.7 The testArena function

This section defines the function

$$\textsc{testArena}(m\_cost : \mathbb{N}) : \mathbb{B}^{16}[2^{m\_cost}][L][W],$$

which is not *per se* a part of a EARWORM, but is used to provide test vectors.

The function

$$\text{AES256}\textsc{Encrypt}(key : \mathbb{B}^{32}, block : \mathbb{B}^{16}) : \mathbb{B}^{16}$$

is the complete AES-256 block cipher as defined by [10].

The constant

$$test\_key : \mathbb{B}^{32}$$

is

```
64 6f 6e 2c 74 20 75 73 65 20 74 68 69 73 20 6b
65 79 20 69 6e 20 70 72 6f 64 75 63 74 69 6f 6e,
```

which is the ASCII encoding of "don't use this key in production".

Then, testArena is defined as follows.

1: **procedure** testArena($m\_cost : \mathbb{N}$)
2: **var** $n : \mathbb{N}$
3: **var** $arena : \mathbb{B}^{16}[2^{m\_cost}][L][W]$
4: $n \leftarrow 0$
5: **for** i **from** 0 **to** $2^{m\_cost} - 1$ **do**
6: **for** l **from** 0 **to** $L - 1$ **do**

```
            for w from 0 to W − 1 do
                arena[i][l][w] ← AES256Encrypt(test_key, be128enc(n))
                n ← n + 1
            end for
        end for
    end for
    return arena
end procedure
```

Essentially, we just populate the arena with an AES-CTR keystream.

WARNING: This function exists *strictly* for testing purposes! Production use of an arena generated from a known seed will seriously degrade earworm's security.

2.8 The PHS function and test vectors

The function

$$\mathrm{PHS}(secret : \mathbb{B}^*, salt : \mathbb{B}^*, t\_cost : \mathbb{N}, m\_cost : \mathbb{N}, outlen : \mathbb{N}) : \mathbb{B}^{outlen}$$

exists only to conform to the Password Hashing Contest's API requirements and to produce test vectors. It should not be used in production. It is defined as

$$\textsc{earworm}(secret, salt, t\_cost, m\_cost, \textsc{testArena}(m\_cost), outlen)$$

and test vectors are as follows:

$\mathrm{PHS}(\text{“secret”}, \text{“salt”}, 1, 12, 16) =$

```
d6 62 fa 90 b9 a9 d7 d2 71 3a fb c0 9d ef e2 2f
```

$\mathrm{PHS}(\text{“secret”}, \text{“salt”}, 10000, 16, 16) =$

```
2b 48 60 81 f7 d3 2c 19 97 67 ef 28 9e be dd c4
```

$\mathrm{PHS}(\text{“secret”}, \text{“salt”}, 10000, 16, 64) =$

```
2b 48 60 81 f7 d3 2c 19 97 67 ef 28 9e be dd c4
40 f7 ef c7 9c ea 40 06 82 29 b4 70 65 2f 08 20
71 d2 0d 09 31 0f 94 0c 0b 84 49 c7 23 15 94 b1
a1 5b 02 31 99 73 4a 21 f2 ec 84 1a da 9a da d3
```

$\mathrm{PHS}(\texttt{nil}, \texttt{nil}, 10000, 16, 16) =$

```
e7 d6 6f 5d 9e f2 05 13 34 09 aa 25 ad ee f0 61
```

$\mathrm{PHS}(\texttt{00 01 02 ... ff}, \texttt{00 01 02 ... 1f}, 10000, 16, 16) =$

```
95 53 15 fc 0e 69 e2 08 f7 b5 68 d4 59 45 0b 5d
```

The strings “secret” and “salt” represent the (ASCII) octet strings

```
73 65 63 72 65 74
```

and

```
73 61 6c 74
```

respectively.

3 Security analysis

3.1 Behavior as a pseudo-random function

EARWORM has a second-preimage property which makes it trivially distinguishable from a random function. This property is a consequence of the way in which HMAC deals with long keys, "long" meaning that their length exceeds the output length of the underlying hash function. Ordinarily, HMAC is defined by

$$\mathrm{HMAC}_H(K, m) = H(K \oplus opad \| H((K \oplus ipad) \mid m))$$

but for long K, it is instead

$$\mathrm{HMAC}_H(K, m) = H(H(K) \oplus opad \| H((H(K) \oplus ipad) \mid m)).$$

As a result, HMAC has the property that for long K,

$$\mathrm{HMAC}_H(K, m) = \mathrm{HMAC}_H(K, m).$$

PBKDF2-HMAC-SHA256 uses the password as an HMAC key and therefore inherits this property. EARWORM likewise inherits it from PBKDF2.

It would have been trivial to avoid this property by specifying EWPRF such that it passes SHA256(*secret*) rather than *secret* to PBKDF2. I have specifically chosen not to do this because second-preimage resistance is not a necessary or useful property for password hashes. Any application of EARWORM which relies on second-preimage resistance is an abuse, and patching over this particular attack by adding an initial hashing step is unlikely to make anyone safer.

When a random oracle is substituted in place of HMAC-SHA256, I conjecture EARWORM to be a weakly secure KDF according to the definition given in [11]. Providing a proof of this conjecture, with concrete bounds on the adversary's advantage, is a goal of future work.

3.2 Network timing attacks

At lines 9, 10, 23, and 29 of EARWORM's WORKUNIT function, *index_a* and *index_b* are are assigned secret-dependent values and then used as array indices at lines 20 and 26. These operations may take non-constant time as a result of the host system's cache architecture, opening up the possibility of timing-based side-channel attacks.

Consider the following scenario. A user transmits his username and password to a server, tunneled within a TLS session or some other protocol providing a similar confidentiality assurance. The server hashes the password using EARWORM, compares it to a stored authenticator, and sends the user a reply indicating a successful login attempt. An attacker eavesdropping on this transaction measures how long it took the server to produce this reply. Later, the attacker compromises the server steals the user's stored password hash and associated parameters (the salt and arena). Let us consider how the attacker

can use his earlier timing measurement to speed up an offline dictionary attack against the password.

Each EARWORM workunit performs D secret-dependent memory accesses which may take variable time, so a complete computation performs a total of $D \cdot t\_cost$ such accesses; we will call this number N. For simplicity, let us assume the following:

1. The host system has only one level of cache, so each of these memory accesses results in either a hit or a miss, with no other result possible.

2. All cache hits take the same amount of time. All cache misses take the same amount of time.

3. Hashing the same password always results in the same pattern of hits and misses.

4. Each memory access results in a miss with probability p. Therefore, the number of misses during computation of a randomly-chosen password follows the binomial distribution $B(N, p)$.

5. The attacker's timing measurement is noiseless, so a single measurement is sufficient to reveal precisely how many cache misses occurred.

These assumptions are not very realistic, but they all favor the attacker.

The attacker can use his knowledge of the number of cache misses which occurred when the user submitted the correct password to speed up his attack by aborting some EARWORM computations early, once too many or too few cache misses have occurred in order for the guessed password to possibly be correct. The algorithm that the attacker can apply here may be more-intuitively understood in the context of the following balls-and-boxes probability problem.

Mallory is handed two bags, each of which contains N balls, some of which are white and the rest of which are black. The first bag contains n white balls; the second contains m. Mallory knows *a priori* that each of the bags was filled by randomly drawing from an infinite reservoir of balls that are white with probability p. He wishes to determine whether $m = n$.

The number of white balls in the first bag is analogous to the number of cache misses that are triggered when hashing the user's correct password. The number of white balls in the second bag is analogous to the number of cache misses which occur when Mallory hashes a guessed password. If $m \neq n$, then as soon as Mallory can conclude that this is the case he can abort that hash computation because the guess cannot possibly be correct.

Mallory proceeds as follows. First, he counts all the balls in the first bag in order to determine n; this represents taking the network timing measurement. Then, he counts balls out of the second bag until one of the following stopping conditions occurs.

- Case I: He has counted $n + 1$ white balls, and so concludes that $m > n$.

- Case II: He has counted $N - n + 1$ black balls, and so concludes that $m < n$.
- Case III: He has emptied the bag without reaching cases I or II, and so concludes that $m = n$.

Given particular values of n and m with $n \neq m$, the number of balls that Mallory will count before stopping can be modeled by a negative hypergeometric distribution. If $m > n$, then the distribution mean is $(n+1)(N+1)(m+1)^{-1}$, and if $m < n$, then the distribution mean is $(N-n+1)(N+1)(N-m+1)^{-1}$. So, in general, the following function describes Mallory's expected stopping time for given values of N, n, m:

$$g(N,n,m) = \begin{cases} N & : \quad m = n \\ \frac{(n+1)(N+1)}{m+1} & : \quad m > n \\ \frac{(N-n+1)(N+1)}{N-m+1} & : \quad m < n \end{cases}$$

Since it is given that $n, m \sim B(N,p)$, Mallory's overall expectation is characterized by

$$f(N,p) = \sum_{n=0}^{N} \sum_{m=0}^{N} \left[\binom{N}{n} p^n (1-p)^{N-n} \right] \left[\binom{N}{m} p^m (1-p)^{N-m} \right] g(N,n,m).$$

Mallory's percentage-wise time savings is then represented by

$$1 - \frac{f(N,p)}{N}.$$

For $N = 256$ (representative of a $t\_cost = 1$ computation) and $p = .5$, this evaluates to approximately a mere 6.3% savings. As N and $|p - .5|$ increase, the savings decreases further, asymptotically approaching zero.

Mallory can do somewhat better if he is willing to accept some risk of improperly rejecting a correct password, but in this case, determining the ideal strategy and how much time it saves seems to be very complicated. A conjecture due to Colin Percival (personal correspondence) and the Stack Exchange user "fedja"[12] is that if Mallory accepts a false-rejection rate of α, then as N grows large, his expected time savings approaches $\sqrt{\alpha}$. For example, accepting a 25% risk of rejecting a correct password provides an expected time savings of 50%.

3.3 Local timing attacks

I make no claims regarding EARWORM's resistance to timing or other side-channel attacks in situations where adversaries have the ability to manipulate CPU cache lines while a EARWORM computation is in progress. Users concerned about such attacks should avoid using EARWORM on systems that timeshare hardware with untrusted parties.

3.4 Reliance on AES and SHA-256

Although EARWORM incorporates SHA-256 and the AES round function, the properties of these functions which it relies on are weaker than those which they were originally designed to ensure.

SHA-256 is designed to be, and as of this writing, is widely believed to be, a collision-resistant hash function. However, EARWORM only relies on it to be first-preimage-resistant. It is not necessary that a password hash be resistant to collisions or second preimages, and indeed, due to the previously-discussed issue concerning long HMAC keys, EARWORM is not. It is not apparent that the ability to find SHA-256 collisions would lead to any sort of new attack against EARWORM.

EARWORM relies on the AES round function to behave in a way that prevents the main loop of the workunit function from being rewritten in order to consume less memory bandwidth, *i.e.*, to ensure that the algorithm described in the pseudocode specification of WORKUNIT is the most efficient one possible. The following definition of a *shortcut-free function* is an attempt to formalize this property.

Let

$$f_n(k : \mathbb{B}^n, m : \mathbb{B}^n) : \mathbb{B}^n$$

be a family of functions, assumed to be computable in time polynomial in n. Let $L : \mathbb{N}$, and let

$$g_n(l : \mathbb{N}, K : \mathbb{B}^n[L], m : \mathbb{B}^n) : \mathbb{B}^n$$

, defined only when $l < L$, compute l iterations of f_n, i.e.,

$$\begin{aligned} g_n(0, K, m) &= m \\ g_n(l, K, m) &= f_n(K[l-1], g_n(l-1, K, m)) \end{aligned}$$

Consider the following game. The defender fixes $n : \mathbb{N}$ and $L : \mathbb{N}$ and then selects

$$K \xleftarrow{r} \mathbb{B}^n[L]$$

and

$$m \xleftarrow{r} \mathbb{B}^n$$

uniformly at random. The defender outputs n, L, and K, keeping m secret. The adversary then outputs a circuit description

$$h(m : \mathbb{B}^n) : \mathbb{B}^n.$$

$\{f_n\}$ is defined to be a *shortcut-free family of functions* if for all adversaries A such that

1. A operates in probabilistic polynomial time, and
2. The circuit-size complexity of A's output is $o(L)$,

the probability that $h(m) = g_n(L, K, m)$ is bound by a negligible function $\epsilon(n)$.

A particular function f_n may informally be said to be a *shortcut-free function* if $\epsilon(n)$ is "sufficiently" small.

The gist of this definition is that the attacker cannot do any precomputation which usefully "compresses" K for the purpose of computing g_n. The first restriction on the set of adversaries asserts that the precomputation must be tractable, and the second restriction asserts that it produces a useful result.

For a simple example of a function family which is *not* shortcut-free, consider addition, where f_n adds two integers modulo 2^n, so g_n computes $m + \sum K$. In this case the adversary can precompute $\sum K$ and output a circuit that performs just the final addition of m. The complexity of this circuit can be $O(\log n)$ and independent of L. In fact, this strategy works for any associative and commutative function.

The security of EARWORM relies on the conjecture that AESROUND is a shortcut-free function.

4 Performance analysis

4.1 Performance on CPUs

EARWORM's performance on CPUs supporting AES-NI is typically determined almost entirely by memory bandwidth. Since computing one workunit requires accessing $16LWD$ bytes = 1MiB of arena memory, a typical system is able to sustain one workunit computation per second per MiB/s of memory bandwidth, minus a negligible amount of overhead imposed by the the initial and final EWPRF computations. Multiple CPU cores may be needed in order to achieve bandwidth saturation.

A system's effective memory bandwidth is not always limited by the bandwidth of the memory modules. My development system, for example, contains an AMD SR5650 north bridge chipset, supporting quad-channel DDR3-1600 DIMMs. When all DIMM slots are in use, the memory modules should have a maximum theoretical bandwidth of 51.2GiB/s. However, the CPU accesses memory via a 16-bit-wide connection to a version 3.0 HyperTransport® bus, which is capable of 5.2GiT/s (gibi-transfers per second), resulting in a 10.4GiB/s bottleneck on EARWORM's access to its arena. Running EARWORM on two cores of the system's 2.3GHz Opteron™ 6376 CPU is sufficient to achieve bus saturation.

4.2 Performance on GPUs

GPUs typically have access to greater memory bandwidth than CPUs, making it conceivable that they could outperform them. However, my (limited) attempts at an efficient GPGPU implementation of EARWORM have so far been stymied by an issue similar to the one that has historically plagued GPGPU implementations of bcrypt. The AES round function, in the way that it is typically

implemented, makes random accesses to each of four 1KiB tables. The current generation of GPUs lack sufficient low-latency memory to provide all cores with fast access to these tables. Although storing the tables in high-latency global memory allows all cores to be utilized, the performance penalty due to latency is severe.

The use of bitslicing techniques enables high-performance implementations of AES that do not rely on table lookups [13][14][15]. EARWORM's somewhat irregular use of the AES round function means that these results are not immediately applicable, but nonetheless further research is necessary to determine whether or not bitslicing presents a viable means of efficiently implementing EARWORM on GPUs.

As a consequence of EARWORM's copious external parallelism, any breakthrough in attacking EARWORM with GPUs is likely to be equally applicable toward defensive GPU use.

4.3 Performance on custom hardware

FPGAs generally lack sufficient I/O bandwidth to be a viable means of attacking EARWORM. Those FPGAs which are even potentially capable of matching the performance of a commodity PC are also far more expensive than a PC.

Something approaching an ideal platform for attacking EARWORM would be a graphics card augmented with AES-NI support. Putting an AES-NI circuit on every core of a modern GPU would likely make it possible to fully utilize the card's memory bandwidth, thus outperforming a typical CPU-based platform by roughly one order of magnitude.

5 Usage considerations

EARWORM's need for a large site-local parameter (the arena) creates some challenges with respect to providing site administrators with an acceptable user experience. Loading a multi-gigabyte arena from disk into memory can take much longer than an EARWORM computation itself, making it a practical necessity that the arena be kept resident in main memory at all times. A straightforward and programming-language-agnostic way to accomplish this would be to implement EARWORM in a daemon listening on a loopback port or UNIX domain socket, providing an RPC API via, e.g., FastCGI[16].

The need to keep backups of the entire arena or copy it from one server to another can be alleviated by using a CSPRNG to generate it from a seed. However, it then becomes very important to protect the seed from compromise, because this would allow the attacker to make a favorable time/memory tradeoff by generating portions of the arena on-the-fly rather than storing the entire thing. A suitably foolproof program design might involve reading a seed from `/dev/urandom` or other system facility, using it to generate and store the arena, and then printing the seed to the screen with instructions to write it down on paper and store it in a physically-secure location.

References

[1] W. T. Newbill, "The United States of America as represented by the National Security Agency's general license statement," 2007. [Online]. Available: https://datatracker.ietf.org/ipr/858/

[2] C. Percival, "Stronger key derivation via sequential memory-hard functions," 2009. [Online]. Available: http://www.tarsnap.com/scrypt/scrypt.pdf

[3] Solar Designer, "New developments in password hashing: ROM-port-hard functions," November 2012. [Online]. Available: http://www.openwall.com/presentations/ZeroNights2012-New-In-Password-Hashing/ZeroNights2012-New-In-Password-Hashing.pdf

[4] Solar Designer and S. Marechal, "Password security: past, present, and future," December 2012. [Online]. Available: http://www.openwall.com/presentations/Passwords12-The-Future-Of-Hashing/Passwords12-The-Future-Of-Hashing.pdf

[5] N. Lawson, "Final post on Javascript crypto," December 2010. [Online]. Available: http://rdist.root.org/2010/11/29/final-post-on-javascript-crypto

[6] D. Eastlake and T. Hansen, "US secure hash algorithms (SHA and HMAC-SHA)," Internet Engineering Task Force, RFC 4634, July 2006. [Online]. Available: https://tools.ietf.org/html/rfc4634

[7] B. Kaliski, "PKCS #5: Password-based crytography specification version 2.0," Internet Engineering Task Force, RFC 2898, September 2000. [Online]. Available: http://tools.ietf.org/html/rfc2898

[8] S. Josefsson, "The scrypt password-based key derivation function," Internet Engineering Task Force, Internet-Draft, September 2012, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-josefsson-scrypt-kdf-00#section-10

[9] S. Gueron, *Intel® Advanced Encryption Standard (AES) New Instructions Set*, 3rd ed., Intel Corporation, September 2012. [Online]. Available: http://download-software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf

[10] "Announcing the Advanced Encryption Standard (AES)," National Institute of Standards and Technology, FIPS 197, 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[11] F. F. Yao and Y. L. Yin, "Design and analysis of password-based key derivation functions," in *Topics in Cryptology — CT-RSA 2005*, 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.4846&rep=rep1&type=pdf#page=256

[12] D. Franke, "Expected running time of a statistical test procedure," August 2013, Stack Exchange post. [Online]. Available: http://math.stackexchange.com/questions/469499/expected-running-time-of-a-statistical-test-procedure

[13] C. Rebeiro, D. Lelvakumar, and A. S. L. Devi, "Bitslice implementation of AES," *Cryptology and Network Security*, pp. 203–212, 2006.

[14] D. J. Bernstein and P. Schwabe, "New AES software speed records," in *Progress in Cryptology — INDOCRYPT 2008*, November 2008. [Online]. Available: http://cr.yp.to/aes-speed/aesspeed-20080926.pdf

[15] J. W. Bos, D. A. Osvik, and D. Stefan, "Fast implementations of AES on various platforms," 2009. [Online]. Available: http://eprint.iacr.org/2009/501.pdf

[16] M. R. Brown, "FastCGI specification," Open Market, Inc., Tech. Rep., April 1996. [Online]. Available: http://www.fastcgi.com/devkit/doc/fcgi-spec.html