STRICT MEMORY HARD HASHING FUNCTIONS (PRELIMINARY V0.3, 01-19-14)

SERGIO DEMIAN LERNER

Abstract. We introduce the concept of strict memory hard functions. Strict memory hard functions are an extension of memory hard functions such that a slight reduction in the memory available for computation, compared to a predefined optimal memory size, makes the function evaluation exponentially slower or infeasible. The main application of strict memory hard functions is to prove a certain amount of memory is used during a certain time interval or in a certain computation. This in turn can be used to attest that areas of memory of devices do not contain hidden data. Other applications are password hashing and proof of work. We present SeqMemoHash and RandMemoHash, two sequential memory hard functions under the random oracle model.

1. Introduction

Cryptographic hash functions are easy to evaluate but practically difficult to invert and while preserving much of the source entropy. Cryptographic hash functions serve as building blocks for other useful functions, such as key derivation functions (KDF). A key derivation function derives one or more secret keys from a secret value such as a master key using a pseudo-random function. Password-based KDFs (PKDF) (also known as password hashing functions) use a password or pass-phrase as the master key. PKDFs are often used for key stretching. Key stretching is used to achieve a more secure password authentication with a website. The website does not store the original password, but a hashed keys, using the PKDF. If an attacker gets access to the list of derived key for all the website users, he may try to perform a dictionary attack or a brute-force attack. This requires evaluating the PKDF for each password candidate and comparing the result to the hashed values. To reduce the usefulness of these attacks, key stretching functions are designed so that they require either a costly computation or a high use of RAM. A costly computation can be achieved by xor-ing the digests of recursively hashing a password and a salt, such as in the PBKDF2 scheme (the xor operation ensures that entropy is minimally reduced). The scrypt algorithm [1] is an example of a key stretching algorithm that can be configured to require the use of certain amount of RAM in order to be computed efficiently and each memory access depends on the previous one. This makes hardware implementations of a cracking machine less attractive since this memory access pattern cannot be paralellized nor pipelined. Nevertheless the scrypt function can still be computed fast even if the memory available during computation is less than the optimal value. Password hashing functions do not need to prevent the function to be computed with less memory: the desired property is that computing the function with less than optimal memory makes a evaluation slow enough so that there is no economic benefit in creating a cracking machine

Key words and phrases. scrypt kdf.

with such properties. In scrypt, computing time increases linearly with the amount of memory removed from the optimum, until some very small constant minimum memory is left, where it turns infeasible (e.g. exponential). If more memory is taken away, the computation turns impossible. In comparison, in strict memory hard functions (SMHF) if more than a constant small amount of memory is removed from the optimum, the computation becomes immediately infeasible or even impossible. We present SeqMemoHash and RandMemoHash, two strict memory hard functions under the random oracle model. These functions can be used as password hashing functions, but they may not be the achieve the highest throughput, since they require a hashing function to support the strict property, while scrypt only needs a uniform strongly-ordered function (a function whose output differs with high probability if the operations are evaluated in a different order). Nevertheless, it's possible to modify our schemes by reducing the number of rounds of the hashing function to achieve higher throughput (e.g.) without affecting the practical security as long as inverting the compression function is expensive compared to add the additional memory required for optimal performance. For example SHA-256 reduced to 16 rounds is 4 times faster than Salsa20/8.

2. Application to Software attestation

Suppose a verifier wishes to attest the software on a target computer and already has an attestation method which verifies that the running or installed software is the same as a predefined base (e.g. by means of hashing). But the attestation process must also verify that nothing is hiding in the free memory, either in volatile or non-volatile storage. The trivial but far from optimal solution is that the verifier sends a truly random number sequence to fill the unused memory, then performs the known software attestation phase, and finally checks for the presence of the same sequence in memory (possible by requesting a hash digest prefixed by a challenge). SMHF can provide the same functionality without the network transmission of the random values, which may constitute a prohibitively high overhead. A verifier provides a seed to the attested computer and then the computer computes a pseudo-random number sequence that fills unused memory using a multiple rounds of a SMHF. This filling process should take a measurable time higher than the communication latency (e.g. 1 second). Then the target computer sends back a hash digest of the last SMHF round, which is verified by the verifier. After the known software attestation phase is concluded, the verifier sends a challenge to the target computer. The computer must reply with a the hash of the challenge concatenated with the output of the last round of the SMHF (that should be stored in memory). If memory is not filled with this data, and the target computer attempts to recompute the SMHF, it will be detected by the verifier because of a high delay in the expected response.

3. Strict memory-hard functions

Def. 1: An algorithm for Random Access Machine is RAM-fast for a memory size n if it uses $T(n)$ operations, where $T(n) = O(n)$.

Def. 2: A function is RAM-fast if it can be computed by a RAM-fast algorithm for a Random Access Machine.

Def. 3: A memory-hard algorithm on a Random Access Machine is an algorithm which uses n space and $T(n)$ operations, where $n = \Omega(T(n)^{1-\epsilon})$..

Def. 3: A function is strict-memory-hard if:

- is RAM-fast and can be feasibly computed by a memory-hard algorithm on a Random Access Machine in n space and
- It cannot be feasibly computed on a Parallel Random Access Machine with any number of processors and n' space, if $n' < n - x$ for a fixed value $x \geq 0$ for any sufficiently large parameter n.

4. SeqMemoHash

SeqMemoHash is our first SMHF proposal. Let H be a one-way compression function. Let D be a hash digest size. Let s be the master-secret whose size equals D. Let $M[i]$ for $0 \leq i < N$ be the memory array, where the memory cell holds D bytes. Let R be the number of rounds of the function. The algorithm computes the function in-place, the output is the memory array M. For PoW or password hashing, the last z blocks of memory ($M[N-z]..M[N-1]$) are taken and hashed again with a secure cryptographic hashing function to obtain the final result.

```
**SeqMemoHash**$(s, R, N)$
  (1) Set $M[0] := s$
  (2) For $i := 1$ to $N-1$ do set $M[i] := H(M[i-1])$
  (3) For $r := 1$ to $R$ do
      (a) For $b := 0$ to $N-1$ do
          (i) $M[b] :=$H$(M[(b-1+N) \bmod N] \mid\mid M[b])$
```

If we set $R \geq N+1$, then SeqMemoHash is strict memory hard. Now we attempt an informal proof. Let c be working state space of the compression function. If the available memory is lower than $N * D + c$ then computing the last round requires storing a temporary state of size D and evaluating the previous round at least twice. The second time the evaluation would have to be performed with $N * D + c - D$ space. The same argument can be applied to the round before the last, but with the reduced memory. After N rounds, the compression function would need to be evaluated without enough temporary space for the working state, and the computation becomes infeasible. Also, for $N \geq 128$, $R = 128$ is enough for 128-bit equivalent infeasibility, since the number of times the first round will be evaluated will be higher than 2^{128}. A more careful analysis shows that the number of backtracking calls each round performs increases with the backtracking depth because each round requires the storage of a temporary hash digest and reduces the caller round temporary storage space. This problem is similar to a register allocation problem, and the exact recurrence that gives the number of hashes that need to be computed for the optimal solution could not be found by the author.

5. RandMemoHash, adding unpredictable memory accesses

The inputs of each hashing step of SeqMemoHash are known in advance, so it is possible to use an optimal register allocation algorithm to reuse as many intermediate results as possible. In fact, a greedy algorithm that always removes the oldest block in the chain seems optimal. In this section we propose a slight modification of our previous function that prevents any register allocation algorithm to know in advance which registers (or memory blocks, in our context) will be needed in the future. We force that the index of one of the blocks to hash depend on the last hash of the chain. This gives a highly uniform random distribution of the indexes. It is still possible that when a recomputation is required, and a backtrack is executed, the current block index is previously stored and passed to the backtracking subroutine as argument, so it can dynamically optimize the state of memory after the backtracking function returns. Nevertheless, the underling problem seems to be NP-complete, so it's possible that only a suboptimal solution is found for a sufficiently large N. Our simulations show that if less than the optimal memory is provided, the number of hashing steps performed by a greedy algorithm grows with the factorial of the number of rounds. If we are able to prove this is true for any algorithm, then it would mean that if $R \geq 35$ the computation turns infeasible for 128-bit equivalent security.

We define RandMemoHash as follows:

```
RandMemoHash($s, R, N$)
(1) Set $M[0] := s$
(2) For $i := 1$ to $N-1$ do set $M[i] := H(M[i-1])$
(3) For $r := 1$ to $R$ do
    (a) For $b := 0$ to $N-1$ do
          (i) $p := (b - 1 + N) \bmod N$
         (ii) $q :=$AsInteger$(M[p]) \bmod (N-1)$
        (iii) $j := (b + q) \bmod N$
         (iv) $M[b] :=$H$(M[p] \mid\mid M[j])$
```

Note that if inputs are taken at random order instead of sequentially, it reduces the utility of CPU caches, which is generally an advantage.

To use RandMemoHash as a password hashing function, the function H can be replaced by a reduced round version (e.g. SHA-256 with 16 rounds) if the final result is fully hashed. Also the seed should be obtained by a standard fast key derivation function from the password and the result M should be passed through another key derivation function to output a shorter keys.

6. Performance optimizations

When RandMemoHash is used for proof of work, it can prevent the use of GPUs or ASICs for spamming or obtaining a speedup over standard computers. By requiring 1 MB of memory, computing SeqMemoHash on a start-of-the-art GPUs (as of 2013) becomes slower than using a standard computer. The inner hashing function can be reduced in rounds, as long as finding a pre-image for the reduced round function is harder than computing RandMemoHash at full.

A suggested configuration for the use of RandMemoHash as PoW is this:

- RandMemoHash is used with 4 rounds (R=4)
- The inner hash function is SHA-256 reduced to 16 rounds
- $N = 2^{16}$, so 2 MB of memory are required to optimally evaluate SeqMemoHash.
- Computing SeqMemoHash requires at least 2^{18} reduced hash function evaluations, which is equivalent to 2^{16} full SHA-256 evaluations, which takes approximately 30 msec in a standard computer.
- Assuming each SHA-256 round requires 4 steps, the total number of steps performed for a RandMemoHash evaluation is 2^{24}.

For this configuration, computing RandMemoHash with half of the memory is conjectured to require the recomputation of approximately 2^{60} hash digests.

Also for use in PoW there is no need to hide the initial seed, not any intermediate state. Full SeqMemoHash pre-image attacks are prevented by hashing the final block with full SHA-256. An attacker will not be willing to break an internal reduced hash function for every nonce, even if it may be computationally feasible. For example, if we assume that breaking pre-image resistance of SHA-2 reduced to 16 rounds requires only 2^{32} steps (breaking the pre-image resistance of SHA-256 reduced to 24 rounds requires 2^{240} steps [3]). Then the cost of computing RandMemoHash with 32 bytes less memory than the optimal size (an hash digest less) requires at least performing these 2^{32} additional steps, so RandMemoHash will require 256 times more than before.

7. Gradual verification

When SeqMemoHash or RandMemoHash are used as PoW, an attacker may try a DoS attack by cheating on the difficulty of the PoW, and forcing the verifier to invest CPU resources in computing the (invalid) MemoHash digest. One way of protecting from this attack is by creating a PoW that consist of the concatenation of all intermediate results produced at steps that are power of two (e.g. at hashing steps 1,2,4,8, ..), and the final result. For the configuration given in the previous section, this requires 17 intermediate hash digests and the final hash digest (totaling 576 bytes). The verifier must check each intermediate state against the given values during the computation. This protection assures that the attacker must have performed at least half of the operations performed by the verifier.

8. Conclusion

We introduce the concept of strict memory hard functions (SMHF) and present SeqMemoHash and RandMemoHash, two SMHF candidates that we conjecture that are strict memory hard under the random oracle model.

9. References

[1] Colin Percival, STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS

[2] Takanori Isobe and Kyoji Shibutani. Preimage attacks on reduced Tiger and SHA-2. In Fast Software Encryption âĂŞ FSE 2009, LNCS. Springer, 2009. to appear

E-mail address: sergiolerner@certimix.com