Gambit

A sponge based, memory hard key derivation function

Krisztián Pintér (pinterkr@gmail.com)

Pseudocode

S	a duplexed sponge
r	the number of words in the r part of the sponge
t	time/CPU cost
m	memory cost in words, *m<r*t*/4
salt	salt
pwd	the password as plain text
Mem[0 .. m-1]	an array of words. indexing modulo *m*.
ROM[0 .. rom-1]	read only random array, optional. indexing modulo *rom*.
R[0 .. r-1]	a temporary array of r words
^ and ^=	c style xor and xor onto
f	interleave factor, a small negative number modulo *m*
auth_key	the authentication key
dkid	identifier of the derived key, *r* words
key	derived key, any length
Trans	a word transformation function (or the identity function if not needed)

```
function Gambit(pwd, salt, t, m, dkid) returns key is
      S.Init
      Mem[0..m-1] := 0
      S.Absorb salt || pwd || pad
      loop i in 0 .. t-1
            R := S.Squeeze
            loop j in 0 .. r-1
                  Mem[i*r + j] ^= Trans(R[j])
                  S.Absorb (Mem[(i*r + j) * f] ^ ROM[i*r + j])
            end loop
      end loop
      // save S here
      S.AbsorbOvr dkid
      key := S.Squeeze
end
```

Explanation and details

S can be any sponge with a reasonably large state. Some parameter choices will be affected by the characteristics of the mixing function, namely by its invertibility. In this document, we use the example of Keccak[1600, c=256 or 512] to give figures about possible password lengths and performance.

Word can be of any size up to the r part of the sponge. A natural choice is 64 bit, to match the word size of modern PC architectures.

The **salt** and the **password** can be of any size, but for simplicity, we suggest a 128 bit salt, and a password with a maximum length that fits into one block. The password does not need to be zero terminated or include length. To deal with shorter passwords, we use padding. Padding can be any sponge compliant padding, for example 10*1. It is also possible to use zero (0*) padding, and disallowing zero characters in the password. With the example parameters, the password can be 119/151 bytes long (plus one byte padding) or 120/152 if zero padding is used.

ROM is an additional array of constant (and preferably high entropy) data. It can be skipped by declaring it an all zero array of any length. (Obviously, optimized implementations will simply omit the xor operation.) ROM can be used as a second factor in the authentication (“have”). And also an additional cost to the attacker, especially if a specialized hardware is used. The maximum effective size of ROM is t*r.

The indexing of the arrays Mem and ROM always assumed to be modulo the size of the array. Mem[k] means Mem[k mod m]. This does not require actual modulo calculations though, because we increase the index by constant, thus modulo can be calculated by one test and conditionally one addition.

There are no limitations on the **t** (time cost) parameter. The **m** (memory cost) parameter must be odd (see later) and must be at most t*r/2. In order to prevent some optimizations, it is advised to be at most t*r/4. More in the security discussion.

The **f** parameter should be chosen in conjunction with the m parameter. First, we want f to satisfy gcd(f, m) = 1 and gcd(f-1, m) = 1. This will ensure that all elements of Mem are used, and the time between writing and reading an element can be anything from 0 to m-1. That supposed to hinder the use of parallel running sponge instances. In addition, it is beneficial to choose f to be f <= (m-r), that is, to maximize the number of back dependencies. Further, we want f to be approximately m*p/(p+1), where p is the relative cost of the inverse of the sponge mixing function. If the sponge uses a one way mixing function, p can be considered infinite. If p is much bigger than 1, f will be close to m, or can be seen as a small negative number mod m. For Keccak, the actual p has to be established. Finding a suitable f in an elegant way is not

straightforward. For the time being, we propose starting with an initial value of floor(m*p/(p+1)), and scanning downwards for a suitable number. For more resource constrained systems, we can use a lookup table for some chosen m values in the used range.

Trans is a function with word domain and codomain. It can be used to further tax the system with some computation that is available on the target platform, but likely to introduce additional cost for the attacker. The identity function (one that gives back the parameter unaltered) can be used to skip that option. On a higher end CPU, Trans can be defined as f(W) = W*C (mod p), where C is any constant, and p is a prime that fits in the word. CPUs come with built in multiplication circuits, making this transformation almost free of charge. One can also define Trans as some floating point operation, even using advanced math functions. This option comes with the caveat that different platforms might have different rounding errors, hurting interoperability.

The **R** array is used only for the simplicity of the code and/or compatibility with libraries. In an optimized code, we can directly read and write the sponge state word by word, and then invoke the mixing function manually. The state words are accessed sequentially, the first word is read, and written to Mem, then xor-ed with the word read from Mem and possibly from ROM. Then we move to the next state word.

Finally the derived key identifier (**dkid**) is absorbed to the state in *overwrite mode*[1], then the key is squeezed. (In overwrite mode, the data is not xor-ed to the state, but the state gets overridden.) The dkid is always r words long. If only one key is needed, for example a verifier for authentication, we can simply choose an all zero dkid. If more than one key is needed, the internal state of the sponge must be saved at the point indicated in the pseudocode before absorbing the dkid, and restored for each derivation. This way we don't have to go through the entire costly part for each piece of derived data. The saved state is called the *derivation seed*.

Of course, any amount of additional data could be squeezed without this additional step, thus it appears to be unnecessary. Indeed, it is a valid option to simply squeeze a fixed amount of bits, and use those as derived keys, or to derive any additional keys by other means. The proposed way on the other hand has the following benefits.

First, it provides perfect future compatibility with not much added complexity. If a new key is needed, one just assigns it a new dkid. It is also possible to initialize a sponge with a dkid, and then use it for encryption[2] or any other purpose a sponge can be used for.

Second, it increases security. We might want to store the derivation seed for an extended period of time (for example the lifetime of a web session), and an attacker might compromise the state. Permutation based sponges (like Keccak) can be run backwards to reveal the password itself.

[1] http://eprint.iacr.org/2011/499.pdf 6.2 The mode Overwrite
[2] http://keccak.noekeon.org/KeccakDIAC2012.pdf

Although this is hindered by absorbing Mem, to be on the safe side, it is advisable to immediately destroy a significant part of the state right after the stretching. Since the next operation will be an overwrite mode absorb of size r, we can simply discard the entire r part of the state in advance.

Third, we don't have to store the entire sponge state, but only a small part. This might be significant on a web server under heavy traffic. With the example parameters, the derivation seed is 32/64 bytes.

Fourth, it allows easy server-relief. The client can do the stretching, and send the derivation seed to the server. The server derives a key with the authentication dkid, and compares it to the stored value. An attacker compromising the authentication database will not be able to find a derivation seed that gives a certain target authentication key.

Notes on security

The proposed hash function is in fact a single duplex sponge construction from the password absorbing to the squeezing of the derived key (or further). Therefore the preimage, second preimage and collision resistance is guaranteed by the sponge itself.

In addition, if the sponge is pseudorandom, the function is pseudorandom on the salt, the password and the ROM array and the dkid.

The design obeys the recommendations of Crypto Coding Standard[3], specifically:

- ✓ Avoid branchings controlled by secret data
- ✓ Avoid table look-ups indexed by secret data
- ✓ Avoid secret-dependent loop bounds
- ✓ Clean memory of secret data

The design does *not*, intrinsically, support the following, probably desired, characteristics

- × Do not write secret-dependent data to large volumes of RAM
- × Use standard cryptographic primitives (SHA3 standardization pending, no other standardized sponges)

Notes on attacking the cost

The sponge is sequential in nature, parallelization (above the "built-in" parallelization of the primitive) is not possible. This ensures the time/CPU cost.

Memory/CPU tradeoffs are acceptable as long as time*memory is not reduced. The way f is constructed intended to make optimizations hard. But we *don't* at the moment have a proof.

[3] https://cryptocoding.net/index.php/Coding_rules

Further analysis is necessary to establish that there is no optimized execution order that circumvents this requirement. The proposed method is most likely *not* sequential memory hard. The expected combined complexity is between $O(n^{3/2})$ and $O(n^2)$, increasing with the t/m ratio.

There is a straightforward optimization possibility based on the fact that half of the first m reads from Mem hit an element that has not been already written. These reads can be omitted. The same is true for half of the last m writes, they will never be used and can be omitted. It means that the actual memory requirement is growing from zero to m during the first m writes, and the reduces from m to zero during the last m reads. If multiple hashes have to be calculated, the next hash can be started before the previous one is finished, using the same Mem permuted. The reduction is proportional to m/t. If we require m<t*r/4, the gain can be pushed below 25%.

Keccak is known to be very fast in hardware, which opens up the path to highly optimized cheap circuits. But the same can be said about modern CPUs and GPUs, which closes the gap. Also, the simplicity of the proposed algorithm closes the gap between an actual implementation and an optimized implementation. We expect the actual implementations to be near perfectly optimized (with the possible exception of the memory cost).

Parameter interdependence

In this performance analysis, we use Keccak as a reference point. Keccak authors claim approximately 10-12 cycles/byte on a modern Intel CPU. This puts the sponge throughput at around 200MB per second at 2.5GHz. According to the security rule m<t*r/4, we use every location four times, resulting in 50MB/s. This means that with a cost parameter corresponding to 1s running time, we can use up to 50MB of RAM. The maximum memory is proportional to the running time.

It is possible to use reduced rounds in the stretching phase (but use a full round after the password absorbing and before key squeezing). For example 3 round Keccak-f() would grant up to 400MB/s memory-time ratio with a possibly reduced safety against collisions. It can also hinder compatibility with libraries.

Motivation

The proposed method is not expected to win the competition. It is more like an attempt to draw attention to some of its techniques and principles, namely: the use of a cryptographic sponge, extreme simplicity, incorporation of a keyfile (ROM), versatility in generating multiple keys, tight security (remains entirely inside the confines of a single primitive), conforming to coding standards and resilience to cache timing attacks.

Notes on the name

gam-bit, n.	A maneuver, stratagem, or ploy, especially one used at an initial stage.
Gambit (comics)	A mutant, Gambit can mentally create, control and manipulate pure kinetic energy to his every whim and desire. He is also incredibly knowledgeable and skilled in card-throwing.

Key derivation is usually the first step in a series of operations, and it expected to outsmart any opponents. Therefore, it is an opening move, gambit. Gambit also happens to be a famous character known to throw cards and perform card tricks on superhuman levels. The way the Gambit hash function throws and picks up words from the Mem array resembles a card trick or card game, words being the cards, the array is the table, and the mixing function is shuffling.

Legal notes

Author is unaware of any weaknesses not mentioned in this document.