The Makwa Password Hashing Function

Specifications v1.0

Thomas Pornin, <pornin@bolet.org>

February 22, 2014

Abstract

We present the Makwa password hashing function, which turns variable-length input data into a fixed-sized output, suitable for storage as a password verification token, or use as a password-derived symmetric encryption key. In order to cope with the inherent weakness of human-chosen passwords, Makwa offers configurable slowness (with an adjustable work factor) and salting.

The most important feature of Makwa is that it supports *delegation*: the bulk of the processing cost can be offloaded to an external *untrusted* system; in some usage contexts, this allows for a much higher work factor, thus better resistance to password cracking attempts. Makwa also offers some other nice features: offline work factor increase (without needing access to the source password), and an optional fast path (subject to knowledge of a private key) which can be extended into an automatic escrow system.

1 Introduction

1.1 Password Hashing

Passwords are secret keys which fit in human brains. A human user *remembers* a password, and enters it in a device at some point. The two main usages for passwords are:

- **Password verification**: the system into which the password is entered must verify the correctness of that password with regards to some stored values.
- **Password-based cryptography**: the password is extended deterministically into other kinds of keys suitable for cryptographic algorithms (password-based symmetric encryption is the most common).

In both cases we want to hash passwords. Indeed, it is best to avoid storing cleartext passwords, because it would allow an attacker who got a partial, read-only glimpse of a server database to escalate into full user impersonation with associated write privileges. In Web contexts, read-only database dumps are the usual outcome of SQL injections, a very common type of attack which can be prevented only through sane programming practices on the server side. Purloined data from discarded hard disks and lost backup tapes is also a classic source of read-only leaks. Therefore, as a second line of defence, systems which perform password verification should only store password hashes, not the passwords themselves. As for password-based cryptography, it inherently implies some hash-like processing (usually called *key derivation*).

The one important property of passwords is that they are *weak*: since they rely on the ability of an average human to remember them, and the acceptance of the said human to type them on a regular basis, passwords cannot be really long or complex. Thus, average password entropy is low. Exact password entropy is very hard to pinpoint because it is a property of the password generation process, not of the generated password itself; it cannot be readily measured[1]. However, it seems fair to state that expecting more than 30 bits of entropy from a user-chosen password would be overoptimistic. Training and extra tools (e.g. password generators) may help, but can also be counterproductive by antagonizing users, prompting them to choose "smart" but non-random passwords, or to write them down (strict "password complexity rules" often backfire that way).

So passwords are vulnerable to exhaustive search. Our envisioned attacker has obtained a copy of one or several (possible millions of) hashed passwords from a database, and wants to find one or several passwords matching some of these hash values. As long as the hash function is ideally resistant to preimages, the best attack method is exhaustive search, that is hashing potential passwords until a match is found (this is often called a "dictionary attack"). Attacker's advantages are the following:

- *Dedicated hardware*: the attacker can use his budget on specific hardware (e.g. GPU arrays or even FPGA and ASIC) which specializes in computing the hash function, while the defender must still "run his business", of which user authentication is only a small part. This usually means that the defender must use a normal PC. This attacker's advantage is further enhanced by *parallelism*: the attacker has a lot of independent password hashing instances to compute, and can thus benefit from parallel hardware (typically, GPU).
- *Patience*: a human user expects to be authenticated within a few seconds at most; while the attacker can often afford to wait several hours or days before cracking a password.

[1]Some tools purport to be "password meters" and to estimate the entropy of a given password, but they really measure how fast they could break the password through exhaustive search. Since such tools are generic while good attackers optimize their search strategies based on their knowledge of the psychology of the victim, password meters tend to wildly overestimate effective entropy.

- *Cost sharing*: the attacker often has several passwords to crack and may share his costs between these different instances. This cost sharing can take the form of *precomputed tables* and thus extend over several cracking sessions, possibly by distinct cooperating attackers over distinct target systems.
- *Technological advances*: computers get faster over time; human brains do not. Thus, with each passing year, attackers can try more passwords per second at a given budget, while the average password entropy does not change, making the password weakness an increasingly worse issue. Moore's Law[20] is a well-known formulation of the unrelenting pace of computing power improvements.

Attacker's patience is a given; we cannot change it except possibly by rotating passwords at a very fast pace, e.g. a new password every day[2]. But very frequent renewal is hugely unpopular with users, who will actively fight against such policies.

The advantage of dedicated hardware can be somewhat lowered by defining the hash function to be especially efficient on PC hardware. This is the path taken by scrypt[27] and Catena[7], who intrinsically rely on the availability of a lot of fast RAM (RAM access is a bottleneck in GPU). We will see that though MAKWA does not especially aims at being optimized for a non-parallel PC, it still seems to fulfill the goal of making the defender's hardware the most cost-effective architecture for computing MAKWA, even in a massively parallel attack setup.

Cost sharing is countered by using *salts*. Formally, there is no longer a single hash function, but a complete family of hash functions, which do not output the same values. Each hashed password then uses its own hash function, and the cost cannot be shared between cracking attempts. The "salt" is an index in the hash function family. MAKWA supports salts. Good salt generation will be discussed in section A.2.

Technological advances can be coped with by using *configurable slowness*: make the hash function inherently slow. The goal is to make each password try expensive for the attacker; unfortunately, making the function slow makes it expensive for *everybody*. The defender cannot make the function arbitrarily slow, because he only has finite CPU and time budgets. User patience is a strong limit; and a busy server will also easily run out of available CPU. One may note that slow hashing tends to make servers more vulnerable to some sorts of Denial-of-Service attacks, which again limits the amount of slowness that can be accepted by the defender. In effect, slowness, through a *work factor*, may *at best* cancel the deleterious effects of Moore's Law.

[2]Rotating passwords does not change the probability for an attacker to guess a password in a given time, given the hash version of that password; however, it may reduce the window of opportunity between hash value interception and actual usage of the recovered password. Daily renewal can thus be modeled as an "impatient attacker" who always gives up after a dozen hours.

1.2 Delegation

MAKWA offers an extra feature, not normally provided by classic password hashing functions (like PBKDF2[13], bcrypt[28] and scrypt[27]). With MAKWA, the hashing can be delegated to an external, untrusted system. Specifically, the external system is assumed to be a potential passive attacker; that system will faithfully compute the functions it is supposed to execute, but it may also try to peek at the data. The idea is that, in some contexts, delegation may offer to the defender a huge increase in available computing power for password hashing, thus allowing for much higher work factors.

Among the scenarios where delegation applies are the following:

- **Password Authenticated Key Exchange and small clients**: in PAKE protocols like SRP[34], client and server mutually authenticate each other with regards to the shared password. What the server stores is not necessarily password-equivalent (i.e. stealing the server's secrets does not allow impersonating the client) but the mathematical structure inherent to PAKE more-or-less implies that the data stored on the server allows for a *fast* dictionary attack. This is fixed if the shared "password" is in fact a password hash (with a big work factor), but then the password hashing must necessarily occur on the client, which might not be up to the task (the client may be a low-power mobile device, or some Javascript in a Web browser). With MAKWA, the client may safely make the server itself do the bulk of the computation, even though the server is not yet authenticated at that point.

- **Feeble server and the Cloud**: a secure but limited server may obtain computing help from rented, cloud-based systems. Since these extra systems are not trusted with secret data, they can be considered to be outside of the security perimeter, and may provide CPU muscle for a much lower cost than if the said server had to be hosted in physically secured premises.

- **Heterogenous clients**: suppose that a server must handle many clients, some but not all being computationally powerful. For instance, this may be the central server for a massively multiplayer game, some clients being full-fledged PC, with all the raw power that a gaming machine may have, while other clients may use a much simpler Web-based interface. In that case, under heavy load, when many clients connect, the server may delegate the hashing of the passwords from the "small" clients to the "heavy" clients themselves. For instance, if at some point there are 200 waiting clients, 80 of which being heavy, the server may use the combined CPU power of the 80 heavy clients to compute the 200 hashes. In this example, the work factor could be such that a single hash takes 400 milliseconds on a PC, and yet all the clients would be served within one second; if the server had to do all the work by itself, then it would have to keep the individual hashing time under 5 milliseconds, i.e. a work factor 80 times smaller, which directly maps to an efficiency boost by a factor of 80 for the attacker[3].

1.3 MAKWA In A Nutshell

The core of MAKWA is a sequence of squarings modulo a composite (Blum) integer. The computational cost is proportional to the number of successive squarings. Work factor increase is as simple as computing some extra squarings.

The initial value consists in the input password, padded to the modulus length with at least 30 bytes obtained from a deterministic KDF, where both the salt value (an arbitrary sequence of bytes) and the input password are used as seed. The KDF is HMAC_DRBG, a well-known standard from NIST, relying on HMAC over a hash function (normally SHA-256). Figure 1 shows an overview of the structure of MAKWA.

It is not necessary to know the prime factors of the modulus in order to use MAKWA. However, knowledge of these factors allows for a "fast path" which bypasses the expensive operations, bringing down the cost to a level similar to that of a RSA private key operation. Knowledge of these factors also allows recovery of the password itself, which can be convenient in some specific contexts.

The algebraic structure of the squarings allows for delegation, in a way reminiscent of blind signatures. A number of *mask pairs* are precalculated; a random selection of these pairs is applied to each delegation job. This process ensures security against leakage with tolerable cost (similar to a RSA private key operation).

1.4 Outline

This document specifies MAKWA. The function itself is defined in section 2. Advanced features, such as a "fast path" processing, an escrow system, or offline work factor increase, are described in section 3. Methods for delegation are described in section 4. Some security analysis is presented in section 5.

Annex A discusses some implementation considerations (e.g. salt generation or password encoding). Annex B shows a detailed test vector, with intermediate values.

[3]These figures ignore I/O latency and assume that the server and the heavy clients are all identical; also, the server may want to submit the same hash to several clients in order to better resist client failure or disruption by evil clients. However, the main concept is still there: the accumulated power of the connected clients usually far exceeds that of the server alone, and delegation allows to enlist that power in the arms race against the attacker.

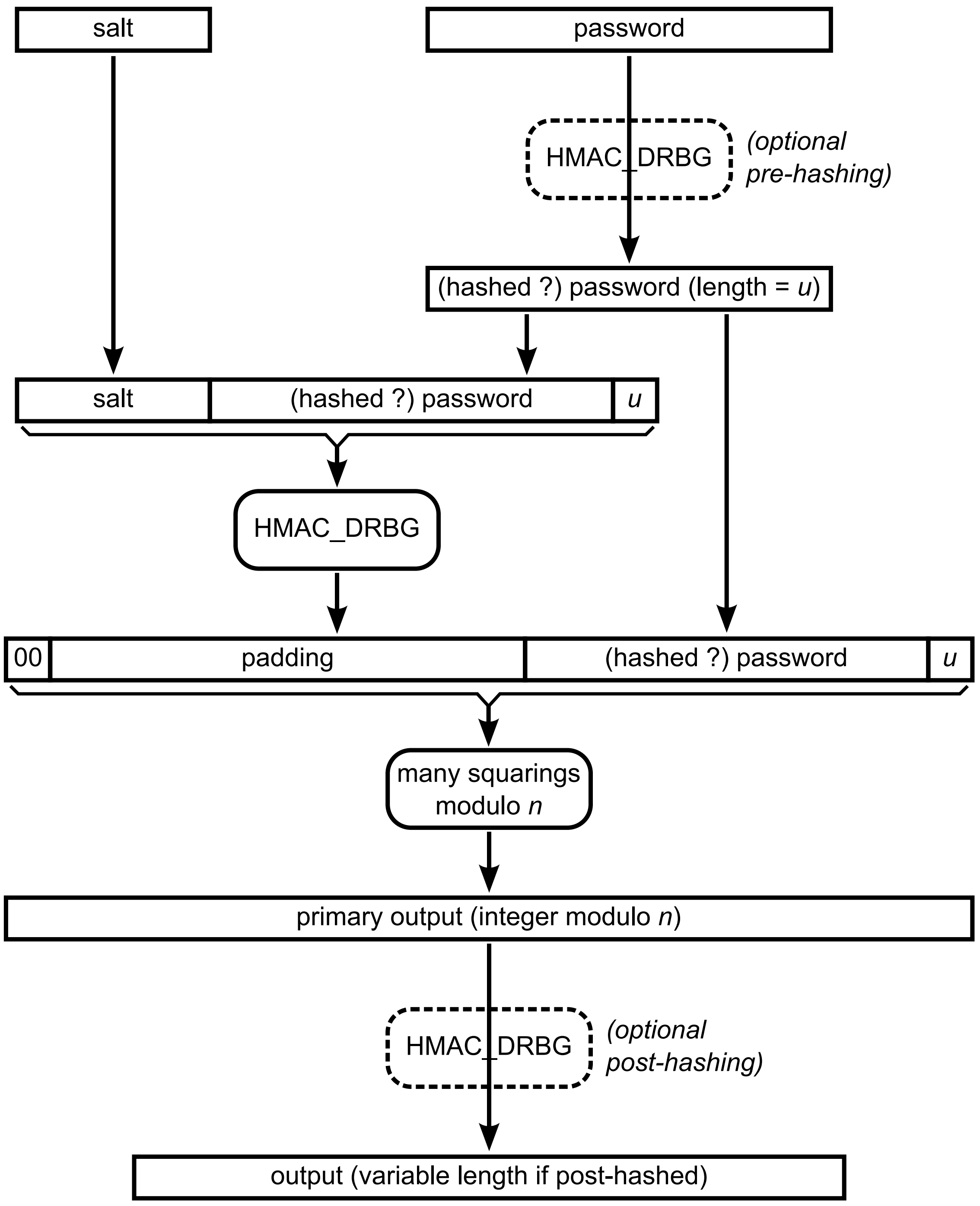

Figure 1: MAKWA overview: input password is optionally hashed, then padded deterministically (using the salt and a KDF), then repeatedly squared modulo a composite n. Final value is optionally hashed again to yield an unbiased output with configurable length.

2 Formal Specification

2.1 Notations

In all of the subsequent text, we call *bytes* what are, formally, *octets*, i.e. sequences of eight bits. A byte has a numerical value ranging from 0 to 255. Hexadecimal values are denoted by a "`0x`" header followed by two hexadecimal digits; e.g. `0x4F` represents a single byte of value 79. Exact mapping of byte values to bits is out of scope of this specification; MAKWA is meant for implementations on architectures where the "byte" is a natural data type.

A *byte sequence* is an ordered list of byte values. The first byte in the sequence is also called *leftmost*, while the last byte is *rightmost*. The *length* of a byte sequence is equal to the number of bytes in that sequence; the "empty sequence" has length 0. Concatenation is denoted by "$||$".

The *length of a positive integer* n is the unique integer a such that $2^{a-1} \leq n < 2^a$; this is the "length in bits". The *length in bytes* b is equal to $b = \lceil a/8 \rceil$.

2.2 Parameters

Let n be a *Blum integer*, i.e. the product $n = pq$ of two prime integers p and q such that:

$$\begin{aligned} p &= 3 \pmod 4 \\ q &= 3 \pmod 4 \end{aligned}$$

The n integer (called thereafter the *modulus*) is a parameter of the MAKWA function. The security of the function relies on the hardness of factoring n into p and q; therefore, p and q shall be generated with the same rules as those used for RSA key pairs. The normal size of n is 2048 bits (256 bytes). For this specification, the size of n MUST NOT be lower than 1273 bits (as we shall see, a larger n allows for longer input passwords; the minimal size is defined so that all passwords encoded over 128 bytes or less can be processed).

The modulus n is a parameter to the function. The p and q factors *are not used*; they need not be stored, except as part of the optional fast path and escrow system. Several systems using MAKWA may work with the same modulus n with no ill effect on security; however, it is expected that each deployment will generate its own modulus n, if only to avoid the possibility of the p and q factors being known by some third party.

The MAKWA function also uses as parameter a cryptographic hash function h. This function will be used in a hash-based deterministic KDF, described below. The output of h shall consist of an integral number of bytes; said otherwise, the output size of h, expressed in bits, must be a multiple of 8. The default function h is SHA-256[24]. The performance of h on any specific hardware architecture does not noticeably impact the overall execution time. h is used *only* as part of HMAC[17], within the construction described in the next section.

2.3 The KDF

The *Key-Derivation Function* H is a deterministic function which takes as input an arbitrary sequence of bytes, and outputs as many bytes as required. It actually is a specialized case of the HMAC_DRBG generator[23], using h as underlying hash function.

Let r be the output size of h, expressed in bytes (e.g. $r = 32$ if h is SHA-256). The KDF H is defined using three internal variables V, K and T which contain byte sequences.

On inputs m (a byte sequence of arbitrary length) and s (the size, in bytes, of the desired output), $H_s(m)$ is computed the following way:

1. Set:
   $V \leftarrow \texttt{0x01 0x01 0x01} \dots \texttt{0x01}$
   such that the length of V is equal to r (V is a sequence of r bytes who all have individual value 1).

2. Set:
   $K \leftarrow \texttt{0x00 0x00 0x00} \dots \texttt{0x00}$
   of length r (K is a sequence of r bytes who all have individual value 0).

3. Compute:
   $K \leftarrow \text{HMAC}_K(V \,\|\, \texttt{0x00} \,\|\, m)$
   where $\text{HMAC}_K(y)$ means “HMAC computed over input y with key K and hash function h”. In plain words, the HMAC input is the concatenation of, in that order: the current value of V, a single byte of value 0, and the KDF input m. The HMAC output is the new value for K.

4. Compute:
   $V \leftarrow \text{HMAC}_K(V)$

5. Compute:
   $K \leftarrow \text{HMAC}_K(V \,\|\, \texttt{0x01} \,\|\, m)$
   Take care that the “extra byte” between V and m has value 1 here, not 0.

6. Compute:
   $V \leftarrow \text{HMAC}_K(V)$

7. Set T to an empty sequence.
8. While the length of T is not at least equal to s (the requested output length), do the following:

   $V \leftarrow \mathrm{HMAC}_K(V)$

   $T \leftarrow T \,\|\, V$

   The output $H_s(m)$ then consists in the s leftmost bytes of T.

2.4 Integer Encoding

Let k_b be the length of the modulus n. If n was generated with the default recommended length, then $k_b = 2048$. Integers in the 0 to n range, i.e. n and all integers modulo n, are encoded into byte sequences of length $k = \lceil k_b/8 \rceil$ bytes, using big-endian convention (leftmost byte is most significant). In other words, value x is encoded as the sequence $x_0 \,\|\, x_1 \,\|\, x_2 \,\|\, ... \,\|\, x_{k-1}$, where the x_i are such that:

$$x = \sum_{i=0}^{k-1} x_i 2^{8(k-1-i)}$$

This encoding is unambiguous and deterministic. This is the exact same encoding as the one called “I2OSP” in the RSA standard PKCS#1[11]; we reuse that name in the context of this specification. Of note, the encoded length is always equal to the length of the modulus, even when the x value, as an integer, is considerably smaller.

The reverse process (decoding) is called “OS2IP” (again an import from PKCS#1).

2.5 Input Pre-Hashing

Pre-hashing is an optional feature of MAKWA. The input to MAKWA (the “password”) is, nominally, a sequence of bytes. Let π_0 be that sequence of bytes.

From that sequence is obtained another byte sequence π, which will be used as parameter for the rest of the processing:

- If pre-hashing is applied, then: $\pi = H_{64}(\pi_0)$
- Otherwise, without pre-hashing: $\pi = \pi_0$.

In other words, if pre-hashing is employed, then the input π_0 is replaced by the result of the application of the KDF to π_0 with a target length of 64 bytes.

Pre-hashing is optional. If it is applied, then the input sequence can have arbitrary length; without pre-hashing, the input sequence has a maximum length which is between 128 and 255 bytes (depending on the modulus length). On the other hand, pre-hashing is not compatible with the advanced "escrow" feature.

2.6 Core Hashing

Let a MAKWA instance be defined to use a modulus n of length k bytes (usually, $k = 256$; n must be such that $k \geq 160$, which means that the length of n is at least 1273 bits).

The inputs to MAKWA are:

- A "password" π; this is actually any byte sequence. If pre-hashing was applied, then π is the 64-byte KDF output. Let u be the length of π (in bytes); u must be such that $u \leq 255$ and $u \leq k - 32$. Thus, the maximum input size to MAKWA (when no pre-hashing is applied) is 255 bytes, or 32 bytes less than the modulus size, whichever is lower. Since we defined n to have length at least 160 bytes, input passwords of length 128 bytes or less are always supported. We will define in section A.1 standard rules for converting a *character string* into a byte sequence.
- A *work factor* w. This is a nonnegative integer. There is no formal upper limit to w, but typical values are expected to be less than one million. Processing time is proportional to w.
- A *salt* σ. The salt is a non-secret sequence of bytes, which is (as much as is possible) distinct for each password hash instance. In a typical system, a different salt value is selected for each hashed password; when a user changes his password, a new salt value is generated. More details on salts are given in section A.2.

The formal specification of MAKWA works with any $w \geq 0$. However, in order to ease some operations (output encoding, delegation), a MAKWA implementation may be restricted to work factors values such that $w = \zeta \cdot 2^\delta$, where $\zeta = 2$ or 3, and $\delta \geq 0$. This yields a choice of possible work factors which can accommodate most practical situations.

The processing of π goes thus:

1. Let S be the following byte sequence (called the *padding*):

$$S = H_{k-2-u}(\sigma \| \pi \| u)$$

In plain words, S is a sequence of $k-2-u$ bytes generated by the KDF, over an input consisting in the salt, followed by the password, followed by the password length u encoded as a single byte. Note that the conditions on u imply that the length of S is at least 30 bytes.

2. Let X be the following byte sequence:

$$X = \texttt{0x00} \,||\, S \,||\, \pi \,||\, u$$

   This is a sequence of exactly k bytes, consisting in, in that order: a single byte of value 0, the sequence S computed in the previous step with the KDF, the password π, and finally the length of π expressed as a single byte.

3. Let x be the integer obtained by decoding X with OS2IP. Since X starts with a byte of value 0 and the length of X is equal to the length of n, it is guaranteed that x lies in the 0 to $n-1$ range.

4. Compute:

$$y = x^{2^{w+1}} \pmod{n}$$

   This computation is normally performed by repeatedly squaring x modulo n; this is done $w+1$ times.

5. Encode y with I2OSP into the byte sequence Y of length k bytes.

The *primary output* of MAKWA is Y.

2.7 Post-Hashing

The primary output of MAKWA is Y; it is appropriate for storage as a password verification token. However, some usages may consider it to be inadequate:

- Y is an integer in the 1 to $n-1$ range; therefore, its leftmost bytes (most significant, in big-endian notation) are, as binary strings, sligthly biased.
- Y is bulky (typically 256 bytes, for a 2048-bit modulus), which can imply higher-than-optimal storage costs.

Optional post-hashing turns the primary output Y into a byte sequence of configurable length, unbiased, and suitable for economical storage and usage as key material for symmetric cryptographic algorithms. We thus define the MAKWA output to be τ:

- If post-hashing is applied, then $\tau = H_t(Y)$ for an integer t.

- Otherwise, without post-hashing: $\tau = Y$

If a post-hashed MAKWA output is stored for password verification purposes, then t must not be too small, because a wrong password would have probability 2^{-8t} to be accepted. As a rule of thumb, such a stored output should have length $t \geq 10$ bytes.

Application of pre- and/or post-hashing conditions the availability of some of the advanced features of MAKWA, as summarized in the table below:

Variant	Unlimited input	Short output	Offline work factor increase	Escrow
core MAKWA	no	no	yes	yes
pre-hashing	yes	no	yes	no
post-hashing	no	yes	no	no
pre- and post-hashing	yes	yes	no	no

The optional features will be described in the following sections. Note that the *work delegation* and *fast path* features are compatible with both pre- and post-hashing.

3 Advanced Features

3.1 Fast Path

In the definition of MAKWA, the modulus n is used “as is” and its factorization needs not be known to anybody. However, if p and q are known, then a fast path process can be applied. Since the fast path directly translates to a fast dictionary attack, the power to apply it, i.e. knowledge of p and q, should be restricted to “trusted” systems. In fact, knowledge of p and q yields even a bit more power, as will be described in the next section.

The core computation of MAKWA is:

$$y = x^{2^{w+1}} \pmod n$$

If we define x_p, x_q, y_p and y_q as:

$$\begin{aligned} x_p &= x \pmod p \\ x_q &= x \pmod q \\ y_p &= y \pmod p \\ y_q &= y \pmod q \end{aligned}$$

Then the following equations hold:

$$\begin{aligned} y_p &= x_p^{2^{w+1}} \pmod p \\ y_q &= x_q^{2^{w+1}} \pmod q \end{aligned}$$

Since p is prime, integers modulo p are a field, and the order of the group of invertible integers modulo p is $p-1$. It follows that we can compute the two integers e_p and e_q:

$$\begin{aligned} e_p &= 2^{w+1} \pmod{p-1} \\ e_q &= 2^{w+1} \pmod{q-1} \end{aligned}$$

and then compute y_p and y_q as:

$$\begin{aligned} y_p &= x_p^{e_p} \pmod p \\ y_q &= x_q^{e_q} \pmod q \end{aligned}$$

From y_p and y_q, y can be efficiently obtained using the Chinese Remainder Theorem[29], as is customarily done with RSA and described in PKCS#1. The computation involves the inverse of q modulo p, which can be precomputed when the p and q values are first generated.

The computation of e_p (respectively e_q) requires $O(\log w)$ multiplications modulo $p-1$, which is fast because w is a small integer (less than 20 bits in practice); e_q can similarly be computed efficiently. Moreover, e_p and e_q depend only on the work factor w, not on the actual password-dependent data element x, so their values can be cached. Since e_p (respectively e_q) is an integer modulo $p-1$ (respectively $q-1$), its binary length is no more than that of p (respectively q), which is about half the length of n, regardless of the actual value of the work factor w.

The cost of computing y from x, knowing p and q and using the equations above, is thus similar to the cost of a private RSA key operation. Using classic implementations, e.g. Montgomery's multiplication[19] and window-based optimizations on the square-and-multiply algorithm[4], the total cost is approximately equal to the cost of $k_b/4$ to $k_b/3$ multiplications modulo n, where k_b is the length of n in bits. In other words, with a normal-sized 2048-bit modulus, the fast path process outlined above reduces the cost below that of a work factor of 700, typically 100 to 1000 times faster than the normal MAKWA computation when the p and q factors are not known.

This fast path smoothly integrates in the normal computation of MAKWA and is thus fully compatible with pre-hashing and post-hashing.

Since the fast path uses secret values p and q in a RSA-like computation, side-channel leaks, in particular timing attacks[16], may be applicable to any specific implementation. A possible countermeasure is *masking*. Whenever the "fast path" is applied, do the following:

- Generate a random integer v in the 1 to $n-1$ range. We want v to be invertible modulo n; probability that a random v in the range is not relatively prime to n is negligible.
- Apply the fast path computation on v, yielding:
$$\dot{v} = v^{2^{w+1}} \pmod{n}$$
- Compute $\tilde{x} = xv \bmod n$ and apply the fast computation on $\tilde{x}$, yielding:
$$\tilde{y} = \tilde{x}^{2^{w+1}} \pmod{n}$$
- Compute $y = \tilde{y}/\dot{v} \bmod n$

The masking mechanism, unfortunately, doubles the cost of the "fast path" (it still is significantly faster than the "normal path", though). It might be possible to somehow reuse some mask values across several MAKWA fast path computations, possibly with some cheap transformations: given masking pair $(v, \dot{v})$, the pair $(v^2, \dot{v}^2)$ is another possible masking pair. To what extent such reuse can be performed without reducing the efficiency of the side-channel protection has not been analyzed yet.

3.2 Escrow

Escrowing the password π is about maintaining an encrypted storage of π, recoverable with the knowledge of a specific private key. For strict password *verification*, escrowing is not needed and is usually frowned upon, because passwords are considered private by human users. However, such a feature can become useful in some situations:

- When a password is used for user authentication but is *also* used (with a distinct password hashing function) to encrypt user data, a forgotten password implies data loss. Escrowing can avoid it by recovering the forgotten password.
- The cleartext password may be necessary to run some authentication protocols. For instance, an email server may want to offer a Web-based interface, and also email download through the POP3 protocol. The Web interface only needs to verify an incoming password; however, the POP3 subsystem must be able to access the actual password in order to support the APOP authentication method.
- Come what may, the possibility to recover user passwords when requested by law enforcement agencies may help achieve compliance with local legal frameworks.

Escrowing can always be applied, on any password hashing function, by the simple expedient of using asymmetric encryption on the password π whenever a new password is chosen; the encrypted value must then be stored along with the actual hash value. In MAKWA, the two processes can be merged: the MAKWA primary output Y *is* an almost-asymmetric encryption of π, and π can be recovered from Y through the process described below, as long as the p and q factors are known.

When p is a prime and $p = 3 \bmod 4$, then every non-zero quadratic residue d modulo p accepts two square roots, and exactly one of them is a quadratic residue. A square root of d modulo p is computed as:

$$c = d^{(p+1)/4} \pmod{p}$$

The formula also works if $d = 0$. Moreover, the square root thus obtained is always a quadratic residue itself. This leads to an efficient way to reverse a sequence of squarings modulo p. If we define integers x_p, x_q, y_p and y_q as in the description of the “fast path” mechanism, then we can compute exponents e'_p and e'_q:

$$\begin{aligned} e'_p &= \left(\frac{p+1}{4}\right)^{w+1} \pmod{p-1} \\ e'_q &= \left(\frac{q+1}{4}\right)^{w+1} \pmod{q-1} \end{aligned}$$

from which we can compute:

$$\begin{aligned} x'_p &= (y_p)^{e'_p} \pmod{p} \\ x'_q &= (y_q)^{e'_q} \pmod{q} \end{aligned}$$

The computed value x'_p is then equal to x_p if x_p is a quadratic residue, to $-x_p$ otherwise. Similarly, x'_q is equal to x_q or $-x_q$. This leads, through the Chinese Remainder Theorem, to four candidates for x. However, x has a very redundant format, in that its binary representation X must match the padding mechanism described in section 2.6. For a candidate x, the value of the rightmost byte of X is equal to the length of π, allowing for unambiguous extraction of π and the padding string S. S can then be recomputed with the KDF and the salt σ. Since there are at least 30 padding bytes obtained from the KDF, the probability of a wrong candidate having the "right" format is no more than 2^{-240}, which is negligible. Therefore, the correct value of x, and thus π, can be reliably recovered.

The computational cost of this password recovery system is, there again, roughly similar to that of a RSA decryption. Contrary to the "fast path", though, it is *not* compatible with pre-hashing and post-hashing: unescrowing can proceed only from Y, not $H_t(Y)$, so post-hashing prevents it. If pre-hashing is applied, but not post-hashing, then knowledge of p and q allows for recovery of $H_{64}(\pi)$, which then permits a fast dictionary attack of π, that can be further optimized through precomputed tables since that hashing process does not include the salt value; but π is not *immediately* obtained.

Similarly to the fast path, the unescrow mechanism implementation may be protected against side-channel attacks by using the same masking procedure:

- Generate a new random masking pair $(v, \dot{v})$.
- Compute $\tilde{y} = y\dot{v} \bmod n$.
- Apply the unescrow procedure to obtain the four candidate values $\tilde{x}$ which fulfill the equation:
$$\tilde{y} = (\tilde{x})^{2^{w+1}} \pmod{n}$$
- Divide each candidate $\tilde{x}$ by v (modulo n) to obtain the corresponding candidate for x.

It is easily seen that the masking+unmasking indeed yields the correct set for the four x candidates: if considered modulo p, the overall effect of masking (multiplication of y by $\dot{v}$) then unmasking (division of $\tilde{x}$ by v) is equivalent to multiplication by 1 or -1, depending on whether v_p is a quadratic residue modulo p or not.

There again, the creation of a new random masking pair incurs a cost similar to that of a RSA private key operation, making the overall unescrow process twice slower (but still considerably faster than the normal MAKWA computation).

3.3 Work Factor Increase

The work factor is meant to cancel technological improvements: w can be set so that password hashing reaches a specific cost on given hardware. When faster hardware becomes available, it suffices to raise w to maintain the cost target at its nominal value.

The tricky point is how to increase the work factor on *existing* password hashes. The generic method, applicable to all password hashing functions, is to wait for the user to log on, because, at that time, the actual password will be available, allowing for rehashing it with a higher work factor. However, MAKWA also supports *offline work factor increase*: the work factor of a stored hash can be increased without needing access to the password.

To increase the work factor from w to w', it suffices to decode the primary output Y into the integer y, then compute:

$$y' = y^{2^{w'-w}} \pmod{n}$$

and reencode y' as the new stored primary output Y'. This process, necessarily, works only for work factor *increments*.

It shall be noted that increasing the work factor works only if the primary output Y is available. Thus, if post-hashing was applied, then work factors cannot be increased in an offline way. Pre-hashing, on the other hand, does not impact the ability to increase work factors.

Decreasing the work factor is possible if the p and q factors are known; if $w' < w$, then it suffices to apply a partial unescrow with exponent $w - w'$. This feature is not expected to be used often; work factor changes are normally a method to cancel progresses in attack speed due to availability of new hardware, and this calls for an increase, not a decrease.

4 Delegation

Suppose that a system A uses MAKWA with modulus n and wants to delegate password hashing cost to the external system B. We assume that A uses homogenous work factors, i.e. all the hashes instances that A is interested in use the same work factor. This is usually not a hard requirement: in password-based authentication scenarios, the work factor is part of the server configuration, and naturally homogenous; and when it is changed, the update is done *en masse* over all stored hashes thanks to the capacity for offline work factor increase. Moreover, we deliberately allow MAKWA implementations to restrict the choice of possible work factors to $w = \zeta \cdot 2^\delta$ where $\zeta = 2$ or 3 and δ is an integer; therefore, there are only a few dozen possible work factor values which "make sense", so all precomputations involving w may possibly be conducted in advance for all such possible values w. In all of this section, we assume that w is fixed and known in advance.

Given n and the common work factor w, A can generate m pairs (α_i, β_i) such that, for all i, α_i is a random quadratic residue modulo n, and β_i is defined as:

$$\beta_i = \left(\alpha_i^{2^w}\right)^{-1} \pmod n$$

Each pair is easily produced by generating a random integer r_i modulo n, then computing α_i as the square of r_i, and then squaring α_i w times and finally inverting the result to obtain β_i. It shall be noted that this computation needs only be done once and can be cached; moreover, the α_i and β_i values need not be kept secret. Computation of the β_i can be performed quickly through the "fast path" mechanism; a good time to produce the (α_i, β_i) pairs is thus right after having generated p and q. For practical usage, consider that $m = 300$ (we need 300 pairs or so).

Let π a password to be hashed. Using the salt, π is padded into byte sequence X which is decoded into the integer value x. Delegation works the following way:

1. A generates m random bits (b_i) with a cryptographically strong PRNG.
2. A computes:
$$z = (x^2) \prod_{b_i=1} \alpha_i \pmod n$$
meaning that x^2 is multiplied with all the α_i for which b_i happens to be equal to 1 (on average $m/2$ of them).
3. A sends n, w and z to B.
4. B computes:
$$z' = z^{2^w} \pmod n$$

5. B returns z' to A.

6. A computes:
$$y = z' \prod_{b_i=1} \beta_i \pmod{n}$$

It is easily seen that the β_i in the last step cancel the α_i which were previously multiplied, so that the obtained value is indeed the primary output y.

The security of the delegation relies on the impossibility for the B system to use z as a test for the correctness of a password. The cornerstone is that the α_i values are a *generic multiplicative knapsack basis*; the best known algorithms for determining whether a given value is part of the set of values which can be generated from the basis have cost $2^{m/2}$, which is why we chose $m = 300$. This will be discussed in more details in section 5.5.

When delegating, the cost for system A is $2m$ multiplications modulo n. With the suggested parameters (n is a 2048-bit integer, $m = 300$), this cost is roughly equal to the cost of a RSA private-key operation on a modulus of the same size. This is also the cost for a "fast path" operation.

5 Security Analysis and Design Rationale

5.1 The KDF

The KDF H described in section 2.3 is actually the HMAC_DRBG pseudorandom number generator, specified in [23]. Its security has been analyzed[9]; roughly speaking, HMAC_DRBG output on an unkown seed is indistinguishable from randomness, up to the resistance of the HMAC_DRBG seed to exhaustive search, as long as HMAC itself is a PRF. Security of HMAC was also proven if the underlying hash function is a Merkle-Damgård hash whose compression function is a PRF. As Bellare puts it in [1]: "[this] helps explain the resistance-to-attack that HMAC has shown even when implemented with hash functions whose (weak) collision resistance is compromised". There is currently no known attack which would distinguish HMAC/MD5 from a PRF, though MD5 is thoroughly broken with regards to collisions and there is even a slightly better than generic attack for preimages against MD5.

5.2 Modulus and Squaring

Factoring n into p and q allows recovering π from the output of MAKWA. It is thus of paramount importance that integer factorization of n is hard. This specific question has been studied at length in the case of RSA, where factorization also breaks the algorithm. The current record for factorization of a RSA modulus is 768 bits[15]. Various people have produced estimates of robustness (see [14] for extensive comparisons and pointers) and how factorization compares to the effort of breaking a symmetric key through exhaustive search. Such comparisons are inevitably fuzzy because both kind of efforts are dissimilar (e.g. factorization requires a lot of RAM, brute-forcing a symmetric key does not), because it depends on the symmetric algorithm that is used for comparison, and because the one unifying measure for all costs (money) ceases to make sense beyond a certain threshold, making interpolation at best haphazard.

It can nonetheless be argued that a 2048-bit modulus ought to offer adequate resistance for now and for the foreseeable future. MAKWA was defined to use a minimal size of 1273 bits because of the design goal of allowing encoded passwords up to at least 128 bytes (without pre-hashing); it is also a size considered "acceptable for long-term protection against small organizations" by ECRYPT II[6].

Squaring modulo a composite integer n is akin to the Rabin asymmetric encryption[30]. It can be shown that the ability to extract square roots modulo n is equivalent to integer factorization[30]. Moreover, squaring modulo a Blum integer is a permutation when applied over quadratic residues, a fact that enables the MAKWA escrow mechanism to work;

successive squaring thus imply no reduction in the space of possible values. Applying successive modular squarings has been used in [2] as a “timed commitment”, following an idea in [31]. There appears to be no known shortcut for computing w successive squarings modulo n when the factorization of n is not known.

5.3 Padding

Padding is done on the left so that, with overwhelming probability, the resulting integer x is “big”. Indeed, while generally speaking, computing square roots modulo n is intractable, it becomes easy if the square root is small, trivial if $x \leq \sqrt{n}$. The padding bytes are added where they are most significant, due to the use of big-endian convention.

Since H is believed to behave like a random oracle, and since there are at least 30 padding bytes, probability of x to be small enough to make square rooting easy is at most 2^{-240} and cannot be forced to be that way with higher probability through deliberate choice of both salt and password.

When computing the padding S, the last byte of the input to H is the password length, so as to avoid spurious collisions (the input to H must be amenable to unambiguous split between the salt σ and the password π). The same last byte is used in the padded password X so that escrowing can recover the password without any hypothesis on the password contents, such as a fixed length or a lack of byte of value 0. Using a single byte limits input size to 255 bytes, which is considered not to be a problem (it is very improbable that a human user would accept to type a *password* that big). For arbitrary long input “passwords”, pre-hashing can be used.

5.4 Cost Sharing

The cost of computing MAKWA is almost entirely that of the $w+1$ squarings. An attacker has many passwords to try, and thus may want to share the squaring costs between several instances. The Rabin cryptosystem is malleable, meaning, in our context, that for all integers x_1 and x_2 modulo n, we have:

$$(x_1 x_2)^{2^{w+1}} = \left(x_1^{2^{w+1}}\right)\left(x_2^{2^{w+1}}\right) \pmod{n}$$

Therefore, if the attacker finds $j+k$ pairs (σ_i, π_i) and $j+k$ “small” integers ϕ_i such that:

- all the π_i are potential passwords;

- all the σ_i are actual salts used in the set of hash values that the attacker is currently trying to crack (with the same cost factor w);
- the following holds:
$$\prod_{i=1}^{j} x_i^{\phi_i} = \prod_{i=j+1}^{k} x_i^{\phi_i} \pmod{n}$$

then the attacker may "try" the $j+k$ passwords while paying only $j+k-1$ times the cost of $w+1$ squarings, and a few extra multiplications (depending on how "small" the ϕ_i integers are).

Fortunately, the definition of the padding S through the KDF H ought to prevent the attacker from practically coming upon such a relation. Indeed, being able to exhibit a relation of that kind is akin to solving a multiplicative knapsack problem in a set of size at least 2^{240} (because the size of S is at least 30 bytes), and the best known algorithms for that would have cost 2^{120}, which is not feasible (see the next section).

5.5 Knapsacks and Delegation

Let G be a finite group (with multiplicative notation). Let (g_i) be a set of j group elements (the "basis"). We are interested in two problems which have been designated by the term "knapsack":

- Given $g \in G$, decide whether there is a subset $J \subset [1..j]$ such that:
$$g = \prod_{i \in J} g_i$$
  This is the *knapsack decision problem* (KSDP). We call the set of values g for which the answer to the KSDP is "yes" the *knapsack generated by the basis*.
- Given an element g of the generated knapsack, compute a subset $J \subset [1..j]$ such that:
$$g = \prod_{i \in J} g_i$$
  This is the *knapsack computational problem* (KSCP).

The KSDP is easy if the basis contains j values and 2^j is much greater than the size of the group G, because in that case it becomes likely that the whole group G is covered, thus the answer would always be "yes". However, if j is such that 2^j is way smaller than the size of G, then the decision is not that simple.

Being able to solve the KSCP obviously implies being able to solve the KSDP. If the basis size is small (j such that 2^j is way smaller than the size of G) then the implication becomes an equivalence, by the following algorithm:

- Given g, solve the KSDP for g/g_1. If g/g_1 is in the knapsack generated by the basis, then infer that $g_1 \in J$ and replace g with g/g_1. Otherwise, infer that $g_1 \notin J$ and keep g unchanged.
- Do the same with g_2, then g_3, and so on.

On generic groups, both KSCP and KSDP are NP-complete[8]. Moreover, the best known algorithms have cost $\sqrt{N}$ where N is the size of the generated knapsack (hence about 2^j or the complete group size, whichever is smaller).

In cryptography, knapsack problems have been used with mixed results. Knapsack instances which have been partially or completely broken were *additive* knapsacks: the group G was integers modulo some value M with the addition as group law (the knapsack problem being then commonly called the "subset sum problem"). The attacks fall into mostly two categories:

- Many early asymmetric encryption systems used a superincreasing basis, where each g_i was greater (as an integer) than the sum of all previous basis elements. A superincreasing basis makes KSCP trivial. For security, the basis was "hidden" in some way, e.g. by multiplication with a secret integer (invertible modulo M). Lattice reduction techniques have proven very effective at unraveling such hiding mechanisms.
- When the modulus is a power of 2 ($M = 2^t$ for some integer t), the knapsack problem can be *dissected*[5] by considering the values modulo $2^{t'}$ for an adequate integer $t' < t$. Dissection allows for faster resolution, down to about $2^{j/4}$ instead of the theoretical $2^{j/2}$ for a generic group.

On the other hand, multiplicative knapsacks (group G consists in a subset of invertible integers modulo some value M with multiplication as group law) appear to resist cryptanalysis. An example is the Naccache-Stern cryptosystem[21]. When working modulo n, no "dissection" method is known for a multiplicative knapsack; at best, the Jacobi symbol[10] can separate integers modulo n into two classes: values which are quadratic residues modulo p but not modulo q, and values which are quadratic residues modulo q but not modulo p, have Jacobi symbol -1, while all other values have Jacobi symbol 1. We took care to use only quadratic residues for delegation: x^2 and all the α_i are all squares modulo n, thus have Jacobi symbol 1, never -1.

In the case of MAKWA, we use knapsacks for two security elements:

- Cost sharing attack methods, as described in the previous section, involve finding a "relation" between some group elements, which boils down to KSCP where the basis consists in all $(x_i/x_j)^\phi$ values, where each x_i comes from a pair (σ_i, π_i), σ_i being a salt from the set of target hash values, and π_i a potential password, and the ϕ are small integers (small enough for the relation to be actually worth the effort, i.e. less than w). The size of the knapsack group is low-bounded by the number of possible padding strings, which is at least 2^{240}, thus yielding a KSCP complexity of at least 2^{120}.
- The delegation mechanism relies on the KSDP. If the system B to which the computation is delegated is hostile, and B succeeds in finding a matching password based on what it saw, then it has solved the KSDP.

For the latter element, consider that B received the value z:

$$z = (x^2) \prod_{b_i=1} \alpha_i \pmod{n}$$

$z/(x^2)$ is part of the knapsack generated by the (α_i). However, we used only 300 values α_i or so, while we operate in a much larger group (there are about $n/4$ quadratic residues modulo n, and $n > 2^{1273}$). Therefore, the probability that, for any x' distinct from x, $z/(x'^2)$ is part of the knapsack is negligibly small. If the attacker can find the password, then he recovered the "right" x, and thus recognized the only value $z/(x^2)$ which was part of the knapsack.

5.6 Attack Speed Estimates

The cost of the operation where most CPU is spent in MAKWA (the $w + 1$ modular squarings) can be somehow extrapolated from published figures on RSA. This must be taken with a grain of salt, in particular because private key RSA operations normally use the CRT and work modulo p and q, while MAKWA is computed without knowledge of these factors.

As a rough approximation, a 2048-bit RSA private key operation uses two 1024-bit exponentiations (1024-bit exponent modulo a 1024-bit modulus factor), each of them requiring about 1260 multiplications, most of them being actually squarings. With numbers of that size, usual algorithms for modular multiplications and squarings have a quadratic cost, so the private key RSA operation will be roughly equal to 630 multiplications modulo a 2048-bit integer.

Exact speed performance does not matter much; as per the theory of password hashing, the important measure is how much computations can be accelerated by using specialized hardware, for a given budget. A 2011 study[22], exploring a novel GPU-based implementation of RSA and aggregating results from previous studies, found that while RSA could be efficiently implemented on the then-available NVIDIA GPU, the resulting performance was at best on par with what could be achieved with a "normal" CPU (AMD Phenom II 1090T), when performance was measured both per dollar and per Watt (throughput per dollar measures the cost of *buying* the hardware, while the throughput per Watt relates to the cost of *running* the hardware for a long time and on a large scale, where power and cooling prices tend to dominate). Assuming that these results are still meaningful three years later, and that RSA performance measured for 1024-bit RSA keys still applies to 2048-bit MAKWA, then we may infer that attackers trying to crack MAKWA-hashed passwords will *not* get a substantial performance boost over the defender by buying GPU.

Figures on RSA performance in FPGA and/or ASIC are much harder to come by. Dedicated cryptographic accelerator hardware (e.g. to "speed up SSL") such as Oracle's Crypto Accelerator 6000[26] and Thales' nShield Connect 6000 are typically priced in the 10k USD range and provide only average RSA performance, e.g. 13000 private key operation for the Oracle card, with 1024-bit RSA; a complete PC with a quad-core 3.1 GHz Xeon CPU will do twice as many for a tenth of the price. To be fair, these dedicated Hardware Security Modules (HSM) aim at tamper resistance against physical attacks, the raw performance being only a secondary goal; also, the price is artificially inflated by the necessity to recoup the very high costs of certification: these devices are certified FIPS 140-2 level 3 or more, and/or EAL4+.

On a rather speculative basis, we may assume that the cost of computing a modular square is mostly the cost of running all the elementary inner multiplications. If using words of s bits, then the number of words for a 2048-bit integer is $t = \lceil 2048/s \rceil$. A squaring entails $t + t(t-1)/2 + t^2$ multiplications of two s-bit unsigned words (the final t^2 is for Montgomery's reduction). A multiplier IP core from Xilinx[35] provides some figures for an optimized multiplier configurable for word size, circuit frequency, latency, and usage of resources (LUT/FF pairs and XtremeDSP slices). Reporting these figures into the biggest available Virtex-7 FPGA from Xilinx[36], the XC7V2000T, we can see that the best that could be achieved with this multiplier core is about 24 millions of 2048-bit squarings per second, assuming that all the extra operations (all the additions, and the data routing) can be done in negligible space, which is very optimistic. Since a 2048-bit RSA private key operation has been deemed equivalent to about 630 squarings, these figures imply about 38000 RSA private-key operations per second on that FPGA, roughly 11 times the performance of a quad-core Intel Xeon E3-1220 at 3.1 GHz[4] – but at almost 100 times the price ! Xilinx XC7V2000T currently sell for an hefty 20k USD per unit.

A "smaller" Virtex-7, the XC7VX690T, actually allows for a few more squarings per second (around 27 millions), thanks to its higher number of DSP units, and yet is cheaper (about 8k USD per unit). This is still more expensive than a PC-based cracking system. We

[4]Measured with OpenSSL-1.0.1e.

again emphasize that these are only wild estimates based on how many multipliers can fit on a single FPGA, ignoring all other costs and considering that placement and routing will be optimal and free (a rather optimistic assumption). It is also probable that the Virtex-7 series do not offer the best performance/cost ratio of FPGA for a MAKWA-cracking task. However, these figures still seem to indicate that it would be hard to beat PC performance with FPGA at a given budget. The technology is not at fault here; on the paper, FPGA ought to offer better performance than generic CPU because they give much more room for parallelism and need not pay for all the features that are present in a CPU but not used in the task at hand (e.g. the billions of transistors for extensive cache memory). But the hardware price has a lot to do with the market size, and the market for PC is *huge*, while high-end FPGA are still a niche.

For these reasons, we deem that *for the time being*, MAKWA fulfills its promises. The most cost-effective architecture to implement an exhaustive search over a MAKWA-hashed password is the kind of hardware that the defender is also most likely to use: a PC. This *may* change in the future, especially if GPU gain better abilities at computing over larger integer types. We must note that while GPU do not seem to offer a significant advantage over a “normal PC” for MAKWA computing, there is no huge gap either; MAKWA is not thoroughly hostile to GPU. Some so-called “memory-hard” password hashing functions, such as scrypt, are arguably better than MAKWA at ensuring that the defender’s average hardware is optimal.

Indeed, MAKWA was designed to allow for delegation, in some contexts, thus more than making up for a relative inefficiency: we would be ready to accept that specialized hardware yields, say, a tenfold speed boost to the attacker over the defender (for a fixed budget), if delegation enables us to harness the power of external third-party systems for a 30 times increase in available power. However, it turns out that MAKWA’s reliance on multiplications, one of the operations that PC are specially optimized to perform, makes it appropriate, by itself, as a stand-alone password hashing function.

5.7 Trivia

“Makwa” is the Ojibwe name[25] for the American black bear (*Ursus americanus*). Linguists classify the Ojibwe language as part of the Algonquian family.

6 Conclusion

We presented and specified the MAKWA password hashing function, which offers the following features:

- Arbitrary binary inputs can be processed, up to a maximum size which is at least 128 bytes.
- Optionally, longer inputs can be used with pre-hashing.
- Output size can optionally be reduced or expanded into sequences of pseudorandom bytes of arbitrary length.
- Processing is salted in order to defeat parallel attacks (including precomputations).
- Processing can be made arbitrarily expensive through a configurable work factor.
- Processing speed is mostly unimpacted by the input password length or the required output length.
- The work factor for hashed values can be increased from the value alone, in an offline manner, without knowledge of the original password.
- The bulk of the processing can be *delegated* to an external untrusted third party, which allows for much higher work factors to be practically used in a number of scenarios.

It seems that the most cost-efficient platform currently available to compute MAKWA is a common PC, which is exactly what an average defender will use anyway; in that sense, MAKWA does a correct job of password hashing. However, delegation of computations is what could potentially yield very significant security improvements, by enlisting considerable extra power, e.g. high-power clients currently waiting for authentication.

References

[1] *New Proofs for NMAC and HMAC: Security without Collision-Resistance*, M. Bellare, Proceedings of Crypto'2006, LNCS 4117, 2006.

[2] *Timed Commitments*, D. Boneh and M. Naor, Proceedings of Crypto'2000, LNCS 1880, 2000, pp. 236–254.

[3] *OpenPGP Message Format*, J. Callas, L. Donnerhacke, H. Finney, D. Shaw and R. Thayer, RFC 4880, 2007.

[4] *A Course in Computational Algebraic Number Theory*, H. Cohen, Springer-Verlag, 1993.

[5] *Efficient Dissection of Composite Problems, with Applications to Cryptanalysis, Knapsacks, and Combinatorial Search Problems*, I. Dinur, O. Dunkelman, N. Keller and A. Shamir, Proceedings of Crypto'2012, LNCS 7417, 2012, pp. 719–740.

[6] *Yearly Report on Algorithms and Keysizes (2012)*, D.SPA.20 Rev. 1.0, ICT-2007-216676 ECRYPT II, 2012.

[7] *Catena: A Memory-Consuming Password Scrambler*, C. Forler, S. Lucks and J. Wenzel, IACR Cryptology ePrint Archive 2013:525, 2013.

[8] *Computers and Intractability: A Guide to the Theory of NP-Completeness*, M. R. Gary and D. S. Johnson, W. H. Freeman, 1979.

[9] *Security Analysis of DRBG Using HMAC in NIST SP 800-90*, S. Hirose, Information Security Applications (WISA 2008), LNCS 5379, 2008.

[10] *Über die Kreisteilung und ihre Anwendung auf die Zahlentheorie*, C. G. J. Jacobi, Bericht Ak. Wiss. Berlin, 1837, pp. 127–136.

[11] *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications, Version 2.1*, J. Jonsson and B. Kaliski, RSA Laboratories, RFC 3447, 2003.

[12] *The Base16, Base32, and Base64 Data Encodings*, S. Josefsson, SJD, RFC 4648, 2006.

[13] *PKCS#5: Password-Based Cryptography Specification, Version 2.0*, B. Kaliski, RSA Laboratories, RFC 2898, 2000.

[14] *Cryptographic Key Length Recommendation*,
`http://www.keylength.com`

[15] *Factorization of a 768-bit RSA modulus*, T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev and P. Zimmermann, IACR Cryptology ePrint Archive 2010:006, 2010.

[16] *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems*, P. C. Kocher, Proceedings of Crypto'96, 1996, pp. 104–113.

[17] *HMAC: Keyed-Hashing for Message Authentication*, H. Krawczyk, M. Bellare and R. Canetti, RFC 2104, 1997.

[18] *A Universally Unique IDentifier (UUID) URN Namespace*, P. Leach, M. Mealling and R. Salz, RFC 4122, 2005.

[19] *Modular Multiplication without Trial Division*, P. L. Montgomery, Math. Computation, vol. 44, 1985, pp. 519–521.

[20] *Cramming more components onto integrated circuits*, G. Moore, Electronics Magazine, 1965.

[21] *A new public-key cryptosystem*, D. Naccache and J. Stern, Proceedings of Eucrocrypt'97, LNCS 1233, 1997, pp. 27–36.

[22] *On the Performance of GPU Public-Key Cryptography*, S. Neves and F. Araujo, 22nd IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2011, pp. 133–140.

[23] *Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised)*, National Institute of Standards and Technology, NIST Special Publication 800-90A, 2012.

[24] *Secure Hash Standard (SHS)*, National Institute of Standards and Technology, FIPS 180-4, 2012.

[25] *The Ojibwe People's Dictionary*, entry on "makwa",
`http://ojibwe.lib.umn.edu/main-entry/makwa-na`

[26] *Oracle's Sun Crypto Accelerator 6000 PCIe Card*, Oracle,
`http://www.oracle.com/us/products/networking/ethernet/crypto6000-pcie/overview/index.html`

[27] *Stronger Key Derivation via Sequential Memory-Hard Functions*, C. Percival, presented at BSDCan'09, May 2009, 2009.

[28] *A Future-Adaptable Password Scheme*, N. Provos and D. Mazieres, Proceedings of 1999 USENIX Annual Technical Conference, 1999, pp. 81–92.

[29] *Fast Decipherment Algorithm for RSA Public-Key Cryptosystem*, J.-J. Quisquater and C. Couvreur, Electronics Letters, 18 (21), 1982, pp. 905–907.

[30] *Digitalized Signatures and Public-Key Functions as Intractable as Factorization*, M. Rabin, MIT Laboratory for Computer Science, 1979.

[31] *Time-lock puzzles and timed-release Crypto*, R. L. Rivest, A. Shamir and D. A. Wagner, Massachusetts Institute of Technology, 1996.

[32] *nShield Connect*, Thales e-Security,
`http://www.thales-esecurity.com/products-and-services/products-and-services/hardware-security-modules/general-purpose-hsms/nshield-connect`

[33] *The Unicode Standard*,
`http://www.unicode.org/versions/latest/`

[34] *The Secure Remote Password Protocol*, T. Wu, Proceedings of the 1998 Internet Society Symposium on Network and Distributed Systems Security, 1998, pp. 97–111.

[35] *LogiCORE IP Multiplier v11.2*, Xilinx Inc.,
`http://www.xilinx.com/support/documentation/ip_documentation/mult_gen_ds255.pdf`

[36] *7 Series FPGAs Overview*, Xilinx Inc.,
`http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf`

A Programming Considerations

In this section, we explore some implementation aspects which are related to MAKWA but not part of the core function specification. In particular, we give recommendations for:

- password encoding;
- salt generation;
- flexible and robust API;
- output encoding into strings;
- serialization of modulus, private keys and delegation messages.

A.1 Password Encoding

MAKWA is primarily meant to deal with passwords, which usually consist in *characters*. Unicode[33] defines mappings from characters to *code points*, where each code point is an integer in the 0 to 1114111 range (that's `0x10FFFF`, the maximum allowed code point value). A sequence of code points can then be converted into a sequence of bytes using one of several conventions.

When MAKWA is used with character string inputs, UTF-8 encoding shall be used.

With UTF-8, every code point is converted into a sequence of 1 to 4 bytes; ASCII characters are encoded "as is" over one byte. Therefore, 128 bytes are enough to fit 128 ASCII characters, or at least 32 generic code points.

When dealing with Unicode and UTF-8, two common sources of vexation are the following:

- Some software platforms have taken to the habit of incorporating an extra leading code point called BYTE ORDER MARK (U+FEFF) when encoding text into UTF-8. The BOM was initially meant to disambiguate between little-endian and big-endian variants of UTF-16; since UTF-8 is byte-oriented, it has no endianness, making byte order detection irrelevant. The BOM can be useful when processing input text which may be UTF-8 or UTF-16 but has otherwise no guaranteed header structure; but, in our case, the BOM needlessly eats up 3 bytes of our precious input length. Therefore, MAKWA implementations shall not add any extra BOM when converting character string inputs into UTF-8.

- Some *glyphs* (the graphical concept that the human user sees) may yield several possible sequences of code points. For instance, the "é" letter can be encoded as a single code point U+00E9, yielding the UTF-8 byte sequence `0xC3 0xA9`, or it may be encoded as two successive code points U+0065 U+0301, for the UTF-8 byte sequence `0x65 0xCC 0x81`. Unicode defines *normalization forms* which specify which variant shall be used. Usually, Unicode data is expressed in NFC form, which is often the shortest and also is most compatible with legacy data (NFC form of "é" is U+00E9, not U+0065 U+0301). We therefore define MAKWA to use NFC by default.

It shall be noted that encoding issues are mostly out of scope of MAKWA. In general, any implementation of MAKWA, when given as input a character string already split into successive Unicode code points (as is customary in most modern programming languages), should encode these code points exactly, without trying to renormalize them and without adding or removing any code point (so that if the input starts with a BOM, then encode it, but don't add a BOM if there was none). However, an *overall system* which uses MAKWA to process user passwords must mind these encoding matters, and ensure that what the user thinks of as "his password" is always encoded into the same sequence of bytes; otherwise, the wrong hash output may be produced. In general, user input devices will return text as NFC, so there is no need to invoke bulky normalization libraries.

A.2 Salt Generation

The point of the salt is to ensure worldwide uniqueness. Any two distinct password hashing instances, regardless of whether they are for the same user or for two different users, shall use distinct salt values. This prevents cost sharing between attacks, in which attackers may try to crack N passwords for less cost than N times the cost of cracking one.

We may note that the *user name* (or any other local identifier) is a poor salt, for the following reason: while two distinct users on a given system will have distinct identifiers, another instance of the same system elsewhere may reuse the identifiers. In crude words, if the user identifier is used as salt, then it becomes worthwhile for attackers to generate rainbow tables for "admin". Moreover, when users change their password, they don't change their name, so this also leads to salt reuse[5]. A similar point can be made with database row identifiers and other local counters.

[5]Old passwords, though no longer active on the system, are still valuable targets for the attacker because typical human users reuse their passwords on all the systems they have access to. Users also tend to generate passwords as "series": when they must change their password, they jump from "P@ssw0rd42" to "P@ssw0rd43". The alternative is to maintain a file containing the 100+ different passwords for all the Web sites that the user occasionally uses; normal humans don't do that spontaneously. Password reuse prohibition can be edicted as a policy, but not really enforced. "Password wallet" tools can help reduce password reuse, but it would be inordinately optimistic to assume that reuse is rare.

On the other hand, there is no need for salts to be secret or unpredictable. Therefore, the easy way to generate a good salt is to obtain sufficiently many bytes from a good PRNG. If the salt length is at least 16 bytes and was generated with a cryptographically strong PRNG (e.g. `/dev/urandom` on Linux systems), then probability of salt reuse is sufficiently low that it can be neglected (it happens sufficiently rarely that attackers don't find it worth the effort to generate rainbow tables or run parallel attacks).

An alternative method is UUID[18] (also known as GUID). These are 16-byte values meant to achieve universal uniqueness, which is exactly what we want for a salt. There are several methods for producing UUID (e.g. by encoding local node characteristics, current time, a local counter...) but they all work well. Most importantly, many programming frameworks already offer functions to generate UUID (see `java.util.UUID` in Java, `System.Guid` in C#/.NET...). If you want to produce a new salt easily, then generate an UUID and encode it over 16 bytes.

The salt shall normally be stored along with the hash output, in a normal password verification system. When using password-based encryption, the data header will customarily contain the salt.

A.3 API

MAKWA is defined to process byte sequences, therefore a generic implementation should provide a function which takes as input a sequence of bytes. However, most usages of a password hashing function are, indeed, about hashing passwords, which are handled as character strings. Therefore, it is best if the MAKWA implementation *also* provides a function which uses a character string as input. An additional reason is that many programmers don't manage encodings well, so doing it in the MAKWA implementation ensures better consistency[6].

Salt generation can be botched. When the API expects the salt as an input parameter in its own right, users may generate the salt poorly, or even use a fixed salt. Allowing the salt to be specified explicitly is an important feature (e.g. to support separate storage of salt and hash value), but the *default* functions for password verification should work as follows:

- To generate the password hash (when the user chooses his initial password or changes it), the MAKWA implementation shall accept as inputs the password and the work factor only. A salt will be generated internally.

[6]For instance, many Java programmers will use `s.getBytes()` to encode string `s` into bytes, which will *usually* use UTF-8, but is locale dependent and thus may break when used in some countries. The correct method is `s.getBytes("UTF-8")`.

- The hash output will be encoded as a string (e.g. with a scheme based on Base64) and that string will also include the salt value and the work factor. We define such an output format in section A.4.
- When verifying a password, only the password and the stored string should be provided as inputs. The function internally decodes the string to recover the salt, the work factor, and the actual primary output with which the hashed password shall be compared.

Such rules have traditionally been used by bcrypt implementations and experience shows that they avoid trouble.

A.4 Output Format

In the interest of easier robust integration, we define in this section a standard format for encoding the MAKWA output and some of its parameters (salt, work factor...) into a single character string. Using such a format has proven effective for reduction of implementation errors; for instance, if the salt value is encoded into the output, then it becomes harder to mistakenly use a fixed salt for all password instances.

When MAKWA is used to produce a hashed password, for purposes of ulterior password verification, the following parameters impact the computation:

- The hash function h used in the KDF.
- The modulus n.
- The salt σ.
- The work factor w.
- Whether pre-hashing is applied.
- Whether post-hashing is applied, and the required output length t.

In a given system, it is expected that the same hash function h and modulus n are used; they need not be repeated in every produced string. However, an explicit checksum might be useful as a tool to quickly diagnose incorrect parameters; this is not a security feature but may help deployments.

A.4.1 Base64 Encoding

Base64[12] is a standard encoding scheme for turning arbitrary sequences of bytes into sequences of ASCII letters (uppercase and lowercase), digits, and "+" and "/" signs. Each sequence of three bytes yields four characters. If the input length is not a multiple of 3, then one or two "=" signs may be appended. In "true" Base64, newline characters are inserted at regular intervals as well. In the following, we *define* the `B64()` encoding function to be Base64 without any intervening newline, and without the "=" padding signs (if any). In the interest of clarity, we describe here the whole process:

- Let the input be the byte sequence M of length ϵ bytes.
- Let $M' \leftarrow M \,||\, Z$ where Z is a sequence of zero, one or two bytes, all of individual value `0x00`, so that the total length ϵ' of M' is a multiple of 3.
- Split M' into 3-byte chunks:

$$M' = m_1 \,||\, m_2 \,||\, ... \,||\, m_{\epsilon'/3}$$

- Convert each chunk m_i into a sequence s_i of four characters, as described below.
- Concatenate the s_i sequences in due order:

$$B' = s_1 \,||\, s_2 \,||\, ... \,||\, s_{\epsilon'/3}$$

- The encoded string B consists in the first (leftmost) θ characters of B', where:

$$\theta = 4 \cdot (\epsilon'/3) - (-\epsilon \bmod 3)$$

  In other words, if ϵ is not a multiple of 3, then we remove from B' its last (rightmost) character (if $\epsilon = 2 \bmod 3$) or its last two characters (if $\epsilon = 1 \bmod 3$).

A 3-byte sequence $b_1 \,||\, b_2 \,||\, b_3$ is converted into a 4-character sequence $c_1 \,||\, c_2 \,||\, c_3 \,||\, c_4$ by first computing four integers d_i:

$$\begin{aligned} d_1 &= \lfloor b_1/4 \rfloor \\ d_2 &= 16 \cdot (b_1 \bmod 4) + \lfloor b_2/16 \rfloor \\ d_3 &= 4 \cdot (b_2 \bmod 16) + \lfloor b_3/64 \rfloor \\ d_4 &= b_3 \bmod 64 \end{aligned}$$

Each d_i is an integer in the 0 to 63 range. Then each character c_i is the encoding of d_i where:

- uppercase letters "A" to "Z" encode values 0 to 25 (in that order);
- lowercase letters "a" to "z" encode values 26 to 51 (in that order);
- digits "0" to "9" encode values 52 to 61 (in that order);
- "+" encodes 62, and "/" encodes 63.

A.4.2 String Format

Consider a MAKWA computation. The modulus is n; the salt is σ; the work factor is $w = \zeta \cdot 2^{\delta}$; pre-hashing may be applied, or not; the output τ is either the MAKWA primary output Y (no post-hashing) or $H_t(Y)$ for some integer t (post-hashing applied).

The output of MAKWA is then encoded as a string with the following format:

$$\texttt{B64}(H_8(N)) \,\|\, \text{``\_''} \,\|\, F \,\|\, \text{``\_''} \,\|\, \texttt{B64}(\sigma) \,\|\, \text{``\_''} \,\|\, \texttt{B64}(\tau)$$

where:

- $H_8(N)$ is the result of the application of the KDF to N, where N is the encoding of the modulus n (by I2OSP).
- F consists in four characters:
  - first character is either "n" (no pre-hashing or post-hashing), "r" (pre-hashing, no post-hashing), "s" (post-hashing, no pre-hashing) or "b" (both pre-hashing and post-hashing);
  - second character is either "2" (if $\zeta = 2$) or "3" (if $\zeta = 3$);
  - third and fourth characters are the decimal representation, over two digits, of δ.

  For instance, if post-hashing is applied, but not pre-hashing, and the work factor is $w = 1536$, then F will be the string "s309".
- σ is the salt.
- τ is the output, whose length MUST be at least 10 bytes.

In other words, the "modulus checksum" ($H_8(N)$, which characterizes the modulus n and the underlying hash function), the "flags and work factor" (F), the salt (σ) and the output (τ) are encoded with B64() (our slightly modified Base64), and the result are concatenated in that order with underscore ("_") signs as separators.

Of note, if several password instances use constant-length salts and produce the same output length, then the encoded strings following this standard format will all have the same length:

- With 16-byte salts (e.g. UUID), a 2048-bit modulus, and no post-hashing, then the formatted string will have length 382 characters exactly.
- With 16-byte salts and post-hashing targeting an output size of exactly 16 bytes (a fine output length for password verification purposes), the formatted string will have length 62 characters exactly.

An important point is that the string-based output encoding supports only work factors w equal to 2 or 3, multiplied by a power of 2. Such work factor values are said to be "encodable". The smallest encodable work factors are $2, 3, 4, 6, 8, 12, 16, 24, 32$...

The restriction on output length is meant to avoid integration mistakes, such as an implementer mistakenly using a post-hashed output length of 2 or 3 bytes, which would imply weak authentication. With 10 bytes of post-hashed output, probability of a wrong password to be accepted is 2^{-80}, which is adequately low.

A.5 Parameter Encoding

In the interest of maximum interoperability, we define here standard formats for encoding a MAKWA modulus, private key or other elements into sequences of bytes.

We encode an integer with the MPI format used by OpenPGP[3]. Let m be a nonnegative integer of length k bytes; its value is:

$$m = \sum_{i=0}^{k-1} m_i 256^{k-1-i}$$

It is encoded as $k + 2$ bytes:

- The first byte has value $\lfloor k/256 \rfloor$.
- The second byte has value $k \bmod 256$.
- Then come the k bytes m_0 to m_{k-1}, in that order.

Thus, the MPI format consists in a two-byte header, which contains the integer length (big-endian convention), followed by the integer value itself in unsigned big-endian convention. There is no sign bit; only nonnegative integers can be encoded. Integers of length up to 65535 bytes (524280 bits) can be encoded. The encoding length is minimal (there is no leading byte of value `0x00` in the value field).

It is recommended that encoders are *strict* (they strictly follow the rules, in particular minimality of encoding length) but decoders are *lenient* (they ignore but do not reject leading bytes of value `0x00` in the value).

The modulus will be encoded as a four-byte header of value `55 41 4D 30` (hexadecimal notation), followed by the MPI encoding of the modulus n itself. Thus, a 2048-bit modulus yields an encoding of length 262 bytes exactly.

The private key will be encoded as a four-byte header of value `55 41 4D 31`, followed by the MPI encoding of the first factor p, then the MPI of the second factor q. By convention, the greater of the two factors comes first (i.e. $p > q$). For a private key such that both factors have length 1024 bits, the total encoded length will be 264 bytes.

A set of delegation parameters will be encoded as the concatenation, in that order, of:

- A four-byte header of value `55 41 4D 32`.
- The MPI of the modulus.
- The work factor for which the set was produced, as a 32-bit unsigned integer (big-endian convention over exactly four bytes).
- The number of (α_i, β_i) pairs, as a 16-bit unsigned integer (big-endian convention over exactly two bytes).
- The (α_i, β_i) pairs, each consisting of the MPI of α_i, followed by the MPI of β_i.

For the recommended parameters (2048-bit modulus, 300 pairs), the total length will be at most 155068 bytes, but may be slightly less because some of the α_i and β_i values may be shorter than the modulus, in application of the “minimal length” encoding rule.

A delegation request, containing the modulus n, the work factor w, and the value z to square w times, will be encoded as the concatenation, in that order, of:

- A four-byte header of value `55 41 4D 33`.
- The MPI of the modulus n.
- The work factor, as a 32-bit unsigned integer (big-endian convention over exactly four bytes).
- The MPI of the z value.

A delegation answer, containing the answer z' from a delegation server, will be encoded as a four-byte header of value `55 41 4D 34`, followed by the MPI of the value z'.

B Detailed Test Vector

In this section, we present a detailed MAKWA computation, with intermediate results.

B.1 Parameters

Let the input π be the UTF-8 encoding of the character string: "*Gego beshwaji'aaken awe makwa; onzaam naniizaanizi.*" (without the quotes). This is an Ojibwe saying which translates as: "Don't get friendly with the bear; he's too dangerous." In hexadecimal, π is the following sequence, of length 51 bytes:

```
pi = 47 65 67 6F 20 62 65 73 68 77 61 6A 69 27 61 61
     6B 65 6E 20 61 77 65 20 6D 61 6B 77 61 3B 20 6F
     6E 7A 61 61 6D 20 6E 61 6E 69 69 7A 61 61 6E 69
     7A 69 2E
```

Note the final "`2E`" which corresponds to the final "`.`" character in our input.

Let n be the following 2048-bit modulus:

```
n = C22C40BBD056BB213AAD7C830519101AB926AE18E3E9FC9699C806E0AE5C2594
    14A01AC1D52E873EC08046A68E344C8D74A508952842EF0F03F71A6EDC077FAA
    14899A79F83C3AE136F774FA6EB88F1D1AEA5EA02FC0CCAF96E2CE86F3490F49
    93B4B566C0079641472DEFC14BECCF48984A7946F1441EA144EA4C802A457550
    BA3DF0F14C090A75FE9E6A77CF0BE98B71D56251A86943E719D27865A489566C
    1DC57FCDEFACA6AB043F8E13F6C0BE7B39C92DA86E1D87477A189E73CE8E311D
    3D51361F8B00249FB3D8435607B14A1E70170F9AF36784110A3F2E67428FC18F
    B013B30FE6782AECB4428D7C8E354A0FBD061B01917C727ABEE0FE3FD3CEF761
```

Let the salt σ be the following 16-byte value:

```
sigma = C7 27 03 C2 2A 96 D9 99 2F 3D EA 87 64 97 E3 92
```

The hash function h is SHA-256. We do not apply pre-hashing. The work factor is $w = 4096$. We will apply post-hashing to get $t = 12$ bytes of output (such an output length is sufficient to verify passwords with a probability of accepting a wrong password equal to 2^{-96}, which is sufficiently low for most purposes).

B.2 Padding

Since the modulus length is $k = 256$ bytes, and the input length is $u = 51$ bytes, the length of the padding S is 203 bytes. The input to H for the computation of the padding S is the concatenation of the salt σ, the input sequence π, and the length of π:

```
s = C7 27 03 C2 2A 96 D9 99 2F 3D EA 87 64 97 E3 92
    47 65 67 6F 20 62 65 73 68 77 61 6A 69 27 61 61
    6B 65 6E 20 61 77 65 20 6D 61 6B 77 61 3B 20 6F
    6E 7A 61 61 6D 20 6E 61 6E 69 69 7A 61 61 6E 69
    7A 69 2E 33
```

The application of H on that sequence produces the following internal values of V and K (numeroted as per the steps in section 2.3):

```
V1 = 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
     01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01

K2 = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

K3 = A0 69 B8 5C EC A1 F3 F7 7D 53 31 28 C0 AD 21 CB
     0E B9 69 07 83 22 41 35 1A 84 77 F6 3E F5 51 2D

V4 = 49 02 28 BA 28 F9 06 55 5E 0F 74 5F 62 71 03 CC
     87 22 4D 99 81 F0 92 EE AB 76 21 C6 49 BD 1A 12

K5 = FB 14 B2 42 6D 97 42 6B 7A D9 7D 4E 65 AB 43 1F
     1E 68 25 70 A6 C7 23 1D C3 A1 37 B3 DA AA 60 A8

V6 = 93 FC 02 3A 13 7A 75 AF 44 08 78 C1 4A 8F 4E 0A
     7F C5 5A 38 82 2E 7A 22 D1 CA 05 AA 92 7B 0F BA
```

These values for K and V, obtained after steps 5 and 6, respectively, are then used to produce the padding string S:

```
S = 7A 84 CD 68 73 0A 09 45 C3 51 CC 73 68 C5 F0 C7
    DD 58 40 F5 D7 97 D8 F4 7B 21 E0 C0 BF E7 77 E3
    2E BB F6 4C 31 41 6A CD B8 B5 E4 93 7C EA A9 61
    89 61 41 A8 84 73 F4 0B 98 FA 5D 61 99 18 7E 78
    38 79 38 64 06 82 4A DF F9 A3 EA 99 0C 77 B1 B1
```

```
62 73 70 AA 73 10 2F 42 88 75 98 A7 1D 8B B0 D6
2D 9B A6 36 25 AF 26 DB 91 A2 91 D0 0F 0F 38 7B
55 33 83 84 BB 6D 94 D4 1B 30 F4 F8 23 E9 47 ED
39 B3 89 8C 91 F1 D5 9F 22 45 72 76 DA 38 5C DE
C6 EE AA F4 68 8D 00 CF DD 87 B7 DF CE B6 B0 DF
3C A8 6E 14 DF D0 75 DF 71 0E C1 21 07 2E 37 23
BD CA 64 E7 84 BC 08 2C BE 9B 34 92 BF 60 7E 58
EC 83 7F E0 0A DE 9E FE 2A 79 B3
```

B.3 Squarings

The padding, encoded password, and password length are then concatenated into the sequence X, which is reinterpreted as the integer x:

```
 x = 007A84CD68730A0945C351CC7368C5F0C7DD5840F5D797D8F47B21E0C0BFE777
     E32EBBF64C31416ACDB8B5E4937CEAA961896141A88473F40B98FA5D6199187E
     783879386406824ADFF9A3EA990C77B1B1627370AA73102F42887598A71D8BB0
     D62D9BA63625AF26DB91A291D00F0F387B55338384BB6D94D41B30F4F823E947
     ED39B3898C91F1D59F22457276DA385CDEC6EEAAF4688D00CFDD87B7DFCEB6B0
     DF3CA86E14DFD075DF710EC121072E3723BDCA64E784BC082CBE9B3492BF607E
     58EC837FE00ADE9EFE2A79B34765676F206265736877616A692761616B656E20
     617765206D616B77613B206F6E7A61616D206E616E69697A61616E697A692E33
```

The first squaring then yields the following integer:

```
x^2 = 000FA1CB767602F6A73DECCD6EF44B85AE509AD7F8891C45D138A7D6E2F63E8C
      55B3CFE430736B20FECBC19E804E77F7942AEF845BF4AAC599FECC6CD45744E0
      CD0421A330501DC48DD03846A531B9C9517594D4CFE56D5662D21CCF3BF474FF
      BF4F25C5178953C039107A256D8E50F58FD4D417FF9FB7B18719F769073DE422
      710086CB2B410E208D12757C4DF92C5F2025B2A9CB3759420737A01F44A9D756
      C73169DEAB27C888FEA69166F04805F7A54FE70EEC84673E2489FCE9D9A6217E
      57B4D5C5B4E41F67AD56B96B48DD717AB1608AC4427791DA9ED37A833E11378D
      3D01E0E1B6C1C92EB106AF7CEED1396F7EFD7118C689B8AB4A94A49746735309
```

The second squaring yields the following:

```
x^4 = A6F1EEACADF5BD2C35EFF5B83E9B2CB0BA33793C528C49FE0F9F6A083B99E9C1
      9DE7FF62233ED43BBFEE0BBCEFD38E1A7ED6E6045ED052A07C7660694CEA8141
      3C451ECA3A7E1FC742EFA34694264784B22ACEA4B2AD3D3996A72C8C5AF87E27
```

```
        85D3CB0FCEF717B9C77627768E16A6EA546A2258B1BDF87215C94EEE7F50AF2C
        106B9DDC80B9CB8090041BD5DED2379A503A51D89091B1A1F6B8C508D48D6A60
        B3A0E39DDD6C1631B540B8CD882E19EA17EAD3B1F76FDC0AB882E54A3B8D5DA9
        AA89718C6028921B57D2AD13762DB2228953049BC58B062C9076D3F68D6158BF
        24351BCAA1D2AFD9455002E8CB64EBAEF8B31EE6676214EAD9429D22DBE4C464
```

And so on. After the 4097 squarings, we obtain the integer y, whose big-endian encoding is the primary output Y:

```
    y = B480C605ECC513158353F828273E98D9483CFAA07DCAE863B425A65AA41EF5C9
        A069225F2A88687CDE3E950CF3C30FEB10E93D5BAC7B6AAE6356B95E31C1C442
        545E3EDEAADBF58E30483161D6876323CF7E43042E5ABBE8BF48FA36B8D80518
        104A34D965785959DF9F9155B977EDAF4F4F7648E59014613F583D7A7895FF6E
        2CCE1B6AEAEF69E8E8E27500DB850FE59FF720E37B03263CC2A3CC0B62301D9A
        0F865BAD35BA93AEB119A3BBE78B84FB8DFDE84615D53868309D66507531D6C8
        9AAC9E3F97904B4AFE367328AC5AD23ED76A87C06AF9CEC94A4A2A088DB87150
        7DF4A137F6C6BF15412E215BCF3BBA7A2EF0C4F5455B5ECC0FD1CE8957619E65
```

B.4 Post-Hashing

We defined that we want to apply post-hashing, to turn the primary output into a sequence τ of $t = 12$ bytes. The KDF steps produce the following intermediate K and V values:

```
V1 = 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
     01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01

K2 = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

K3 = 64 66 69 F4 9E DF 35 2D 9A 28 E5 AC 24 FE D2 B2
     7E FC F5 45 12 08 ED 80 0E ED A9 9F 63 25 8D BF

V4 = 7B CE F5 FA 21 45 04 9A 9A 8C 49 8E 1C 1F 01 97
     99 62 74 0E FE 53 59 74 60 8D 15 B3 81 6B 9B 18

K5 = DD 8E FE 2B DC 0A 8E BA 23 9F 07 2C E0 A5 83 89
     29 5C CA A2 80 82 7F 5F 0C BA BF C5 63 8A F6 A4

V6 = 79 BB 49 5E DC 72 6A 6D 71 FB 5C 04 F3 3A 0F CF
     E5 3E BF 18 3D 05 12 AC 2B 28 FC F7 41 5B 4A E2
```

Finally, the 12-byte output is:

```
out = C9 CE A0 E6 EF 09 39 3A B1 71 0A 08
```

With the string output encoding defined in section A.4, MAKWA produces the following string of 56 characters:

```
+RK3n5jz7gs_s211_xycDwiqW2ZkvPeqHZJfjkg_yc6g5u8JOTqxcQoI
```

Note the "flags" component: the "`s`" indicates that post-hashing was applied, but not pre-hashing; and the "`211`" encodes the work factor $w = 2 \cdot 2^{11} = 4096$.