Catfish – A Tunable Password Hashing Algorithm with Server-Specific Computational Shortcut (Version 1.0)

Bo Zhu, Xinxin Fan, Qi Chai and Guang Gong

University of Waterloo, Canada,
{bo.zhu,x5fan,q3chai,ggong}@uwaterloo.ca

March 31, 2014

Abstract

Password-based authentication has been widely deployed in practice due to its simplicity and efficiency, and how to securely store passwords and derive cryptographic keys from passwords are crucial for many security systems. However, the choices of well-studied password hashing algorithms are quite limited, as their security requirements and design principles are different from common cryptosystems.

In this paper, we propose a simple and secure password hashing algorithm, called Catfish. The design of Catfish is built upon several well-understood cryptographic algorithms, which combines advantages of symmetric and asymmetric primitives. By using a collision-free hash function based on discrete logarithm, Catfish provides better security and especially a server-specific computational shortcut if certain private information is known. It is also sequential memory-hard, in order to thwart large-scale password cracking by parallel hardware such as GPUs, FPGAs and ASICs. Moreover, the total computation and memory usage of Catfish are tunable through its cost parameters.

Chapter 1

Introduction

Password-based authentication is probably the most widely used security mechanism across all information systems thanks to its cost-effectiveness and efficiency. However, there are two fundamental limitations of password-based authentication: (1) Users routinely pick poor passwords which are particularly subject to brute-force attacks; and (2) A device or server storing a large number of passwords is consistently a juicy target for attackers and how to keep it secretly and minimize the damage if it has been breached is non-trivial. As a countermeasure, all passwords should be obscured together with user-specific, public, random and high-entropy salts by applying a one-way function (a.k.a. password derivation function or password hashing function) before storing them. During the authentication, the user's input is processed in the same way and the obscured input is compared with the obscured password stored for the purpose of authentication.

Generally speaking, there are several requirements that a good password hashing algorithm should fulfill:

- Similar as most cryptographic primitives, the algorithm should behave as a random function that ensures *one-wayness*, *collision-resistance*, resilience of side-channel attacks and invulnerable to known cryptanalytic technologies such as time-memory tradeoff [9, 14], differential/linear analysis [6, 17];
- Different from most cryptographic primitives, the algorithm should be *heavyweight* in computation and memory usage to slow down brute-force attacks to a certain degree and make the attacks economically difficult. Note that the desired heavyweightness is expected to be roughly consistent for all platforms, no matter software or hardware;
- *Server-specific shortcut* is an optional but very attractive feature. When this feature is enabled, the server can obscure passwords using less computation (call it *server-specific computational shortcut*) and/or less memory (namely *server-specific memory shortcut*) thanks to the private information that can be leveraged to accelerate the derivation.

1.1 Related Work

Due to these uncommon and demanding requirements, there are only limited choices such as PBKDF2 [15], bcrypt [22] and scrypt [21], despite that password hashing is the foundation of many real-world cryptosystems. PBKDF2 is a key derivation function designed by RSA Laboratories, standardized in RFC 2898, has been the subject of intense research and still remains the best

conservative choice. PBKDF2 is more a conventional design that mainly relies on iterating a pseudorandom function (which is usually HMAC-SHA1) a certain number of times. However, the iterative design leads to quite unaggressive usage of memory, which makes the parallel computing possible [11]. bcrypt is designed by Provos and Mazières, based on the Blowfish cipher [24] with a purposefully expensive key schedule. Thanks to the adaptive iteration count that can be increased to make it slower, bcrypt could remain resistant to brute-force search attacks even with increasing computation power. Like PBKDF2, bcrypt works in-place in memory and performs poorly in thwarting attacks using dedicated hardware. scrypt, designed by Percival, is a proposal that not only offers stronger security from a theoretical point of view than the other two but also allows to configure the amount of space in memory needed to compute the obscured password. This additional feature reduces the advantage which attackers can gain by using the custom-designed parallel circuits and has been extensively evaluated when scrypt was selected as the underlying proof-of-principle function for many cryptocurrencies, e.g., Litecoin, Dogecoin and Mastercoin.

1.2 Our Contributions

In this paper, we propose a new tunable password hashing algorithm named Catfish based upon well-studied cryptographic structures and primitives. The novelty in the design of Catfish is the combination of asymmetric and symmetric components that offer twofold benefits: (1) Since the tools to cryptanalyzing asymmetric algorithms are quite different from those of symmetric ones, the composition of asymmetric and symmetric components often makes the cryptanalysis harder. This is analogous to the design of ARX based cryptographic primitives [2, 4, 19] and the block cipher IDEA [16] where mixed operations are used; (2) The asymmetric component not only makes our scheme more robust but also enables the server-specific computational shortcut as a result of faster exponentiation via the Chinese Remainder Theorem when certain private information is available. In addition to describing the Catfish design in great detail, we also prove the one-wayness and collision-resistance of its underlying hash function in theory.

1.3 Organization

The organization of the paper is as follows. Chapter 2 describes the cryptographic primitives that Catfish employs. The Catfish algorithm is specified in Chapter 3. We discuss the design rationale and provide the initial security analysis of Catfish in Chapter 4. Efficiency analysis is presented in Chapter 5 and we conclude this paper in Chapter 6.

Chapter 2

Ingredients of Catfish

This chapter briefly introduces several cryptographic primitives, which are employed to build Catfish.

2.1 Provably One-Way and Collision-Free Hash Function

Gibson proved that if factoring n is hard, the following discrete logarithm based hash function

$$\mathcal{DL}(x) = g^x \bmod n$$

is one-way and collision-free [12][1], where x is a positive integer and g is a generator of the multiplicative group of integers modulo n. The security of this hash function is reduced to the hardness of integer factorization, since a collision will lead to the factorization of n.

To obtain a hard-to-factor modulus n, one can utilize the same approach as that for generating moduli for the RSA algorithm, i.e., randomly generating two large prime numbers p and q, and using their product $n = p \cdot q$ as a modulus.

2.2 Sponge Construction based Hash Function Keccak

Keccak is the winner of the SHA-3 cryptographic hash function competition held by NIST [20]. It is designed by Bertoni, Daemen, Peeters, and Van Assche [5]. Keccak is based on a unique construction, namely *sponge construction*, which can *absorb* an arbitrary-length binary string as input, and then *squeeze out* a binary string of the required length.

In our password hashing design, Keccak is adopted to:

- Fully mix password and salt strings;
- Expend short input strings to the large space of $\mathcal{DL}$;
- Alternately apply to intermediate states with $\mathcal{DL}$ to gain more cryptanalytic strength;
- Process the final state to produce hash tags of the required lengths.

If not specified, default parameters of Keccak should be used, e.g., $r = 1024$ and $c = 576$.

[1] It is claimed in [25] that this hash function was proposed by Shamir, and a simple proof was given by Rivest.

2.3 Sequential Memory-Hard Construction ROMix

The password-based key derivation function scrypt was proposed by Percival in order to thwart parallel brute-force attacks using GPU, FPGA or ASIC chips on passwords and has been widely used by cryptocurrencies. One of core functions of scrypt, namely ROMix, is proven to be *sequential memory-hard*. One important feature of being sequential memory-hard is that parallel algorithm cannot asymptotically achieve efficiency advantage [21]. For a more detailed definition of sequential memory-hard, the reader is referred to [21].

We list ROMix below as it is highly relevant to our scheme. In ROMix, H is a cryptographic hash function, B is an input binary string, M is called the CPU/memory cost parameter which must be larger than 1, and Integerify is a bijective function that maps binary strings to integers.

Algorithm 1 $\mathsf{ROMix}_H(B, M)$

1: $x \leftarrow B$
2: **for** $i \leftarrow 0$ to $M-1$ **do**
3: $\quad v_i \leftarrow x$
4: $\quad x \leftarrow H(x)$
5: **end for**
6: **for** $i \leftarrow 0$ to $M-1$ **do**
7: $\quad j \leftarrow \mathsf{Integerify}(x) \pmod{M}$
8: $\quad x \leftarrow H(x \oplus v_j)$
9: **end for**
10: **return** x

Chapter 3

Specification of Catfish

Following notations are used in this Chapter:

- $\mathsf{int}(\cdot)$ means converting a binary string into a non-negative integer, where the little-endian convention is used, i.e. the left-most (lowest address) bit is the least significant bit of the integer;
- $\mathsf{str}_b(\cdot)$ converts a non-negative integer back to a binary string, and may append zeros to the string in order to achieve a total length of b bits;
- For a given binary string s, we use $\mathsf{len}(s)$ to denote the bit-length of s, and $\mathsf{pad}(s)$ to append zeros to s and output a 128-byte (1024-bit) string;
- $||$ concatenates two binary strings.

Letting $\mathcal{DL}_b$ be $\mathcal{DL}$ whose modulus n is b-bit long and $\textsc{Keccak}_b$ be a Keccak instance that produces only b bits as output, we define a new hash function

$$\mathcal{H}_N(x) = \mathsf{str}_N(\mathcal{DL}_N(\mathsf{int}(\textsc{Keccak}_N(x)))),$$

for a given positive integer N. To be secure, N should be at least 1024 bits, or preferably larger than 3072 bits, according to [3]. Moreover, in order to be easily implemented in software, N is highly recommended to be a multiple of 8.

The hash function $\mathcal{H}_N$ should be initialized by the following steps, which generate a proper modulus n and the corresponding generator g for $\mathcal{DL}_N$:

1. Randomly generate two secure prime numbers p and q, such that $\mathsf{len}(p \cdot q) = N$.
2. Randomly choose a generator g for the modulus $n = p \cdot q$.
3. Discard p and q, or store them securely if the server-specific computational shortcut is desired.
4. Return n and g as the parameters to be used in $\mathcal{DL}_N$ of $\mathcal{H}_N$.

Steps 1 and 3 can be skipped if p and q are preset or given as input parameters. The initialization process is only required to be performed once, since n and g can be stored for later usage. The detailed procedure for generating the parameters p, q, n and g is described in Appendix A.

Given a 128-bit binary string *salt* as a randomly generated salt, a variable-length ($\leq$ 128 bytes) binary string *pass* as a user password, a positive integer *tcost* as the time cost parameter,

Algorithm 2 $\mathsf{Catfish}_{\mathcal{H}_N}(salt, pass, tcost, mcost, hsize)$

1: $ctr \leftarrow 0$
2: $x \leftarrow salt||\mathsf{str}_{128}(\mathsf{len}(pass))||\mathsf{pad}(pass)$
3: **for** $i \leftarrow 0$ to $tcost - 1$ **do**
4: $\quad x \leftarrow \mathcal{H}_N(x)$
5: $\quad$ **for** $j \leftarrow 0$ to $mcost - 1$ **do**
6: $\quad\quad v_j \leftarrow x$
7: $\quad\quad ctr \leftarrow ctr + 1$
8: $\quad\quad x \leftarrow \mathcal{H}_N(x \oplus \mathsf{str}_N(ctr))$
9: $\quad$ **end for**
10: $\quad$ **for** $j \leftarrow 0$ to $mcost - 1$ **do**
11: $\quad\quad k \leftarrow \mathsf{int}(x) \pmod{mcost}$
12: $\quad\quad ctr \leftarrow ctr + 1$
13: $\quad\quad x \leftarrow \mathcal{H}_N(x \oplus v_k \oplus \mathsf{str}_N(ctr))$
14: $\quad$ **end for**
15: $\quad ctr \leftarrow ctr + 1$
16: **end for**
17: $tag \leftarrow \textsc{Keccak}_{hsize}(x \oplus \mathsf{str}_N(ctr))$
18: **return** tag

a positive integer *mcost* as the memory cost parameter, and a hash tag bit-length *hsize*, our new password hashing algorithm Catfish is defined by Algorithm 2.

Lines 5-14 of Algorithm 2 are essentially the same as ROMix, except that the input to $\mathcal{H}_N$ is always XORed with an incremental counter *ctr*.

Chapter 4

Design Rationale and Security Analysis

The design rationale of Catfish is to inherit the existing structure of scrypt that is proved to be sequential memory-hard and improve its inner components to provide better security and asymmetry in computation as desired. In terms of the security, our design combines public-key and symmetric-key algorithms, and alters the operation sequence to make analysis harder. This is analogous to the design of some ARX ciphers and IDEA where mixed operations are used. In terms of the asymmetry in computation, the public-key algorithm in our design can be computed faster by the ones (usually the authorized servers) who have the private information. In what follows, we discuss several security properties of Catfish in detail.

4.1 One-Wayness

One of the most important security goals of designing password hashing is one-wayness, i.e., attackers should not be able to devise any methods faster than brute-force search for reversing the hashing algorithm and finding original passwords.

The crypographic hash Keccak and the discrete logarithm based function $\mathcal{DL}$ are applied to the intermediate state x alternatively, and we believe that there are no newly emerging weaknesses when combining these two algorithms together, so in order to reverse $\mathcal{H}$ the attackers might have to analyze Keccak and $\mathcal{DL}$ separately. Even if the one-wayness of Keccak is completely broken, say replacing Keccak by an identity function, the one-wayness of $\mathcal{H}$ is still guaranteed by $\mathcal{DL}$, i.e., the hardness of integer factorization. More formally, we give the following definition.

Definition 1. *For a given function f and a randomly chosen output y, we define the advantage of an adversary $\mathcal{A}$ to find the preimage of y (reversing f) as*

$$\mathbf{Adv}_f^{\mathrm{Pre}}(\mathcal{A}) \stackrel{\mathrm{def}}{=} \Pr[x \leftarrow \mathcal{A}^{f,y} : f(x) = y].$$

Then we can reduce the preimage security of $\mathcal{H}$ to the one of $\mathcal{DL}$, as described in the following theorem.

Theorem 1 (One-wayness of $\mathcal{H}$)**.** *If $\mathcal{H}_N$ and $\mathcal{DL}_N$ are using the same parameters n and g, then for any adversary $\mathcal{A}$, we will have*

$$\mathbf{Adv}_{\mathcal{H}_N}^{\mathrm{Pre}}(\mathcal{A}) \leq \mathbf{Adv}_{\mathcal{DL}_N}^{\mathrm{Pre}}(\mathcal{A}).$$

Proof. Once a preimage of y of $\mathcal{H}_N$ is found, e.g., $y = \mathcal{H}_N(x)$, we let $x' = \mathsf{int}(\textsc{Keccak}(x))$ and $y' = \mathsf{int}(y)$, and then x' is a preimage of y' of $\mathcal{DL}_N$. □

Note that there is a negligible chance that Keccak may produce N-bit all-zero strings, and computing $\mathcal{DL}$ of 0 will always return the same value, i.e. 1, regardless of the values of n and g. Moreover, the proofs of $\mathcal{DL}$ in the paper [12] are only for the cases that the input is a positive integer. However, in practice, the probability of Keccak producing all-zero strings should be negligible, and even if it happens it will likely disappear when $\mathcal{H}$ is iterated for multiple times with an incremental counter. Henceforth, the security level of the entire design of Catfish will not be influenced.

4.2 Collision and Second-Preimage Resistances

Collision and second-preimage resistances are also desirable when designing a password hashing scheme. In this context, an incident of collision may result in two passwords being hashed to the same tag, whereas a second preimage implies that given a password *pass1* one may find the second one *pass2* that produces the same tag. It is easy to see that if there exists an algorithm for constructing second preimages, then it can also be used to generate collisions, so collision-resistant implies second-preimage-resistant.

Once a collision of Keccak is found, then it will also result in a collision of $\mathcal{H}$. Thus the collision probability of $\mathcal{H}$ is not upper-bounded by $\mathcal{DL}$.

Formally, we give the security definition related to collision attacks.

Definition 2. *For a given function f, we define the advantage of an adversary $\mathcal{A}$ to find a collision of f as*

$$\mathbf{Adv}_f^{\text{Coll}}(\mathcal{A}) \stackrel{\text{def}}{=} \Pr[x_1, x_2 \leftarrow \mathcal{A}^f : f(x_1) = f(x_2)].$$

Then we have the following theorem about collision attacks on $\mathcal{H}$.

Theorem 2 (Collision-resistance of $\mathcal{H}$)**.** *If $\mathcal{H}_N$ and $\mathcal{DL}_N$ are using the same parameters n and g, then for any adversary $\mathcal{A}$, we will have*

$$\mathbf{Adv}_{\mathcal{H}_N}^{\text{Coll}}(\mathcal{A}) \leq \mathbf{Adv}_{\mathcal{DL}_N}^{\text{Coll}}(\mathcal{A}) + \mathbf{Adv}_{\textsc{Keccak}_N}^{\text{Coll}}(\mathcal{A}).$$

Proof. Suppose a colliding pair x_1 and x_2 of $\mathcal{H}_N$ are found, i.e. $\mathcal{H}_N(x_1) = \mathcal{H}_N(x_2)$. Let $s_1 = \textsc{Keccak}_N(x_1)$ and $s_2 = \textsc{Keccak}_N(x_2)$. Then we have two cases:

- If $s_1 = s_2$, then a collision of $\textsc{Keccak}_N$ is found;
- If $s_1 \neq s_2$, then $\mathsf{int}(s_1)$ and $\mathsf{int}(s_2)$ form a colliding pair of $\mathcal{DL}_N$.

Therefore, the theorem holds. □

Even if a collision attack is discovered on Keccak, it may still be computationally infeasible to be applied to Catfish, since Keccak has been iterated for multiple times.

4.3 Thwarting Brute-Force Attacks using Parallel Hardware

Although the design of Catfish may be secure for random inputs in theory, users' passwords are usually weak and easily crackable by using parallel computation based on GPUs, FPGAs and ASICs. Thus password hashing designs should try to thwart such attacks as much as possible.

The hardware such as GPUs, FPGAs and ASICs can feature thousands of cores for parallel computing, but each core only has very restrained memory space. By using the structure of ROMix, Catfish inherits scrypt's security property of being sequential memory-hard. The design of Catfish provides a tunable memory parameter *mcost* to increase its memory cost as desired.

Please note that in scrypt, a structure called BlockMix is used to build an internal hash function with wide input/output from a small function Salsa20 core [4]. However, BlockMix may not be necessary for Catfish since the input/output length of $\mathcal{H}_N$ should be relatively large. As a side benefit of omitting BlockMix, our scheme is simpler and easier to be analyzed when compared to scrypt.

4.4 Preventing Potential Self-Similarity Attacks

An incremental counter *ctr* is XORed with the intermediate states of Catfish before each invocation of $\mathcal{H}$ and Keccak, in order to prevent potential self-similarity attacks, such as iterative patterns of $\mathcal{H}$. The similar technique is used in many other cryptographic designs, such as Keccak, PRESENT [7] and PRINCE [8].

Chapter 5

Initial Efficiency Analysis

This chapter focuses on the analysis of Catfish's time and memory cost, which includes the server-specific computational shortcut and the potential method to transform hash tags into larger cost settings.

5.1 Tunable Time and Memory Costs

The design of Catfish provides two parameters, *tcost* and *mcost*, for applications to tune its time and memory consumption.

The parameter *mcost* adjusts the amount of memory that needs to be present during the computation of ROMix in Catfish. The memory usage is expected to be around

$$N \cdot mcost$$

bits. Due to the sequential memory-hardness property of ROMix [21], without having such amount of memory, the computation time of Catfish will increase dramatically.

The parameter *tcost* has limited ability to adjust the time usage of Catfish, since the total time cost also relies on the memory usage. To complete a full computation of Catfish, it requires

$$tcost \cdot (2 \cdot mcost + 1)$$

invocations of $\mathcal{H}$, plus one final computation of Keccak.

5.2 Efficiency of Software Implementation

Catfish is built upon well-established cryptographic primitives, and their implementations have been studied for years. For example, the discrete logarithm problem is the basis for Diffie-Hellman key exchange algorithm [10] widely used in TLS/SSL. Keccak is designed to be efficient in both software and hardware. Overall we may have efficient software implementations for Catfish as well.

Although slowness is somehow desirable in password hashing designs for thwarting large-scale password cracking, we should consistently improve the time efficiency of the implementations of Catfish. For example, by reducing the computation time of $\mathcal{H}$, we will have more flexibility for the time parameter *tcost*. On the other hand, attackers are always trying to speed up their cracking methods, so there is no reason why we should stick to under-optimized implementations.

5.3 Server-Specific Shortcut with Private Information

It would be very attractive if password hashing algorithms could support private parameters or keys to speed up hashing computations. For example, legitimate servers with certain private information may compute or verify hash tags faster than attackers who have obtained only salts and hash tags. In this way, servers will save a lot of computation cost without risking too much about the overall security.

Due to the nature of the modular exponentiation function $\mathcal{DL}$, if we know the factorization of the modulus n, i.e., knowing p and q, $\mathcal{DL}$ can be evaluated faster by using the Chinese Remainder Theorem. p and q should be kept securely in this case, e.g., being encrypted and stored separately with hash tags.

Note that even if p and q are leaked to attackers, the overall security of Catfish still has Keccak as a "fail-safe". With this private information, attackers can compute Catfish as fast as legitimate servers could but the brute-force search for original passwords is probably still a must.

5.4 Transforming Existing Hashes to Larger Cost Settings

If we set $hsize = N$ in Algorithm 2, the hash tag tag can be processed by $\mathcal{DL}_N$ again and re-enter the algorithm from Line 5, which is equivalent to increasing the time cost parameter $tcost$ by one. During the additional computations, we can also choose larger numbers for the memory parameter $mcost$. Under such circumstance, hash tags can be updated according to new cost settings without the knowledge of the original passwords.

Chapter 6

Concluding Remarks

We propose a simple password hashing algorithm, Catfish, which is built upon well-studied cryptographic primitives, such as a provably secure hash algorithm based on discrete logarithm and integer factorization, the SHA-3 hash competition winner Keccak, and the sequential memory-hard construction ROMix. We prove that Catfish inherits the similar security properties as discrete logarithm, Keccak and ROMix, such as one-wayness and memory-hardness. More interestingly, a computational shortcut exists if certain private information is known, which is especially useful for large authentication servers responsible for handling thousands of requests simultaneously. The design of Catfish provides two parameters *tcost* and *mcost* that can be tuned for different application environments.

In order to fully utilize the memory-hardness property of ROMix, the internal hash function $\mathcal{H}$ should be as fast as possible, since during a fixed time period the total amount of memory that can be consumed is limited by the computational speed of $\mathcal{H}$. The current design of $\mathcal{H}$ is based on modular exponentiation of big integers, so it may not be efficient enough for certain devices with constrained CPU power. One possible solution for this issue is to replace $\mathcal{DL}$ by more efficient public-key algorithms, such as ones based on elliptic curves or the Rabin's public-key encryption algorithm [23]. For example, Shamir proposed an efficient authentication algorithm for RFID tags [26], which is based an optimized version of the Rabin's algorithm. The other solution involves in removing public-key algorithms from the *mcost* loop, i.e., the inside of ROMix, and only applying $\mathcal{DL}$ at the beginning or the end of the *tcost* loop. However, the security of these potential solutions needs futher investigations.

It is encouraging to design password hashing by combining asymmetric and symmetric algorithms, since the combined algorithm offers provable security and possibilities of server-specific computational shortcuts. We expect more password hashing designs consisting of asymmetric and symmetric components to appear.

Intellectual Property Statement

The scheme is and will remain available worldwide on a royalty free basis, and that the designers are unaware of any patent or patent application that covers the use or implementation of the submitted algorithm.

Consent

We, the designers of Catfish, faithfully declare that we have not inserted any hidden weaknesses in the submitted scheme.

Bibliography

[1] American National Standards Institute, ANSI X9.31-1998: Public key cryptography using reversible algorithms for the financial services industry (rDSA), 1998.

[2] J. P. Aumasson and D. J. Bernstein, SipHash: a fast short-input PRF, Progress in Cryptology, INDOCRYPT 2012, LNCS 7668, pp. 489–508, 2012.

[3] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, Recommendation for key management, part 1: general (revision 3), *NIST Special Publication 800-57*, 2012.

[4] D. J. Bernstein, The Salsa20 family of stream ciphers, *New Stream Cipher Designs*, LNCS 4986, pp. 8–97, 2008.

[5] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, The Keccak SHA-3 submission. *Submission to NIST (Round 3)*, 2011.

[6] E. Biham and A. Shamir, Differential cryptanalysis of DES-like cryptosystems, *Journal of CRYPTOLOGY*, vol. 4, no. 1, pp. 3–72, 1991.

[7] A. Bogdanov, L. R. Knudsen, G. Leander, and C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, PRESENT: an ultra-lightweight block cipher, *Cryptographic Hardware and Embedded Systems, CHES 2007*, LNCS 4727, pp 450–466, 2007.

[8] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, *et al.*, PRINCE – A low-latency block cipher for pervasive computing applications, *Advances in Cryptology - ASIACRYPT 2012*, LNCS 7658, pp. 208–225, 2012.

[9] W. Diffie, Special feature exhaustive cryptanalysis of the NBS data encryption standard, vol. 10, no. 6, pp. 74-84, *IEEE Computer* 1977.

[10] W. Diffie and M. E. Hellman, New directions in cryptography, *Information Theory, IEEE Transactions*, 22, no. 6, pp 644–654, 1976

[11] M. Dürmuth, T. Güneysu, M. Kasper, C. Paar, T. Yalcin and R. Zimmermann, Evaluation of standardized password-based key derivation against parallel processing platforms, *Computer Security, ESORICS 2012*, LNCS 7417, pp. 716–733, 2012.

[12] J. K. Gibson, Discrete logarithm hash function that is collision free and one way, *IEE Proceedings E (Computers and Digital Techniques)*, 138, no. 6, pp 407–410, 1991.

[13] J. A. Gordon, Strong primes are easy to find, *Advances in Cryptology - Eurocrypt 1984*, LNCS 209, pp. 216–223, 1985.

[14] M. E. Hellman, A cryptanalytic time-memory trade-off, *Information Theory, IEEE Transactions on*, vol. 26, no. 4, pp. 401–406, 1980.

[15] B. Kaliski, PKCS# 5: Password-based cryptography specification version 2.0, *RFC 2898, available at* `http://www.ietf.org/rfc/rfc2898.txt`, 2000.

[16] X. Lai and J. L. Massey, A Proposal for a New Block Encryption Standard, LNCS 473, pp. 389–404, 1991.

[17] M. Matsui, Linear cryptanalysis method for DES cipher, *Advances in Cryptology, EUROCRYPT 1993*, pp. 386–397, 1994.

[18] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press Series on Discrete Mathematics and Its Applications. CRC Press, Boca Raton, FL, 1997.

[19] R. M. Needham and D. J. Wheeler, TEA extensions, *available at* `http://www.movable-type.co.uk/scripts/xtea.pdf`, 1997.

[20] NIST Computer Security Division, The SHA-3 cryptographic hash algorithm competition, *available at* `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

[21] C. Percival, Stronger key derivation via sequential memory-hard functions, *BSDCan*, 2009.

[22] N. Provos and D. Mazieres, A future-adaptable password scheme, *USENIX Annual Technical Conference, USENIX 1999*, pp.81–91, 1999.

[23] M. O. Rabin, Digitalized signatures and public-Key functions as intractable as factorization, *Technical Report*, MIT, 1979.

[24] B. Schneier, Description of a new variable-length key, 64-bit block cipher (Blowfish), *Fast Software Encryption, FSE 1994*, LNCS 809, pp. 191–204, 1994.

[25] R. Senderek, A discrete logarithm hash function for RSA signatures, *available at* `http://senderek.com/SDLH/discrete-logarithm-hash-for-RSA-signatures.ps`, 2003.

[26] A. Shamir, SQUASH – A new MAC with provable security properties for highly constrained devices such as RFID tags, *Fast Software Encryption, FSE 2008*, LNCS 5086, pp 144-157, 2008

Appendix A

Generation of Domain Parameters p, q, n and g

Strong primes are required by the ANSI X9.31 standard [1] for use in generating RSA keys for digital signatures. Therefore, we first generate two strong primes p and q. A prime p is said to be a strong prime if the following conditions are satisfied:

- $p-1$ has a large prime factor r;
- $p+1$ has a large prime factor s;
- $r-1$ has a large prime factor t.

Gordon [13] proposed the following algorithm for generating strong primes.

Algorithm 3 Gordon's Algorithm for Strong Prime Generation

1: Generate two large random primes s and t of approximately the same size
2: $i \leftarrow 1$
3: **repeat**
4: $\quad r \leftarrow 2it + 1$
5: $\quad i \leftarrow i + 1$
6: **until** r is a prime
7: $z \leftarrow s^{r-2} \pmod{r}$
8: $p' \leftarrow 2zs - 1$
9: $j \leftarrow 1$
10: **repeat**
11: $\quad p \leftarrow p' + 2jrs$
12: $\quad j \leftarrow j + 1$
13: **until** p is a prime
14: **return** p

Using Gordon's algorithm we can find two strong primes p and q for a pre-defined security level. Then the modulus n is set to be $n = p \cdot q$. In order to obtain a random generator g of Z_n^*, we first need to find a generator g_p (resp. g_q) of Z_p^* (resp. Z_q^*), followed by computing g using the Chinese Remainder Theorem. The above procedure is described in Algorithm 4.83 of [18].

Appendix B

Test Vectors

For testing, we generate a set of domain parameters at the 80-bit security level as follows, where p and q are 512-bit strong primes, i.e. $N = 1024$.

p	6ee498d8d2351442d8af696f71cbd0e479368f22304c53ed1480fe416095175b
	86359b7086707ab192cfdf6c12cddc57d16f0f29f0ba4e7774f0f07117670085
q	9b188e9e7b3ba224084430c259110e29f52e9247d6867f996c799a29beececa3
	8412e70553fe3b99b255f5bc9a2e6be230cdfe7c7b0b86ba07d828c6cff9dcef
n	432f0bc10eda1d0538e16fd4c1882c01736d2ebb4a21ca00f8de508551c6aff6
	78ba19e7a9162c00c993ce4fb251bab10981dabf84154ddf7c53c7b3cc987165
	bcffce39a4fcbad83e1a8bbb11a6ff928e09aac05037a21636a6671acfed77d7
	e109b243f5ba9b5cce8aa56e85914785096f33bca9ebac78d074e52a69f8c82b
g	1f91b2b37b136f3886df5c6871551eb9c8977ade58b77693999a30d8ad11fc00
	43d9ad55451dfc2f805d46dea4b5d3b631ac841c8d46cae8cdd40e1ee8e036a6
	c6acdad4e7f06523203195fd78a9d541240a18afe024dc9e0f241017e99d5904
	5d8d3b8cb3d91859a71f027e22773b0b14057d59383c4eaad0b49382bbfa5665

These numbers are represented in their hexadecimal formats, e.g., `6ee498d8` denotes the integer 1860475096, and long numbers are written in multiple lines.

Then we give the following input parameters and their corresponding hash tag as reference. Please note that the hexadecimal numbers here represents byte strings, e.g., `546865` means a string *The*, which is different from the ones in the above table.

salt	4c880aa553669c3869f62b389c2c3499
pass	54686520717569636b2062726f776e20666f78206a756d7073206f7665722074
	6865206c617a7920646f67
tcost	2
mcost	1024
tag	f19f0330e9b10929edd99fb53a8344eebb909db6ebaafe36d5999a8a2e2608e0

More test vectors, along with reference implementations, are available at `https://github.com/ComSec/Catfish`.