Internet Research Task Force
Internet-Draft
Intended status: Informational
Expires: August 01, 2015
January 28, 2015

S.H. Shin
K. Kobara
AIST

Augmented Password-Authenticated Key Exchange (AugPAKE)
draft-irtf-cfrg-augpake-03

Abstract

This document describes a secure and highly-efficient augmented password-authenticated key exchange (AugPAKE) protocol where a user remembers a low-entropy password and its verifier is registered in the intended server. In general, the user password is chosen from a small set of dictionary whose space is within the off-line dictionary attacks. The AugPAKE protocol described here is secure against passive attacks, active attacks and off-line dictionary attacks (on the obtained messages with passive/active attacks). Also, this protocol provides resistance to server compromise in the context that an attacker, who obtained the password verifier from the server, must at least perform off-line dictionary attacks to gain any advantage in impersonating the user. The AugPAKE protocol is not only provably secure in the random oracle model but also the most efficient over the previous augmented PAKE protocols (SRP and AMP).

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 01, 2015.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 2
1.1. Keywords 3
2. AugPAKE Specification 4
2.1. Underlying Group 4
2.2. Notation 4
2.2.1. Password Processing 6
2.3. Protocol 6
2.3.1. Initialization 7
2.3.2. Actual Protocol Execution 7
3. Security Considerations 9
3.1. General Assumptions 9
3.2. Security against Passive Attacks 9
3.3. Security against Active Attacks 10
3.3.1. Impersonation Attacks on User U 10
3.3.2. Impersonation Attacks on Server S 10
3.3.3. Man-in-the-Middle Attacks 11
3.4. Security against Off-line Dictionary Attacks 11
3.5. Resistance to Server Compromise 12
4. Implementation Consideration 13
5. IANA Considerations 13
6. References 13
6.1. Normative References 13
6.2. Informative References 14
Appendix A. Features of AugPAKE 15
Appendix B. Test Vector of AugPAKE 16
Appendix C. AugPAKE over EC Groups 18
Authors' Addresses 20

1. Introduction

In the real world, many applications such as web mail, Internet banking/shopping/trade require secure channels between participating parties. Such secure channels can be established by using an authenticated key exchange (AKE) protocol, which allows the involving parties to authenticate each other and to generate a temporary session key. The temporary session key is used to protect the subsequent communications between the parties.

Until now, password-only AKE (called, PAKE) protocols have attracted much attention because password-only authentication is very convenient to the users. However, it is not trivial to design a secure PAKE protocol due to the existence of off-line dictionary attacks on passwords. These attacks are possible since passwords are chosen from a relatively-small dictionary that allows for an attacker to perform the exhaustive searches. This problem was brought forth by Bellovin and Merritt [BM92], and many following works have been conducted in the literature (see some examples in [IEEEP1363.2]). A PAKE protocol is said to be secure if the best attack an active attacker can take is restricted to the on-line dictionary attacks, which allow to check a guessed password only by interacting with the honest party.

An augmented PAKE protocol (e.g., [BM93], [RFC2945], [ISOIEC11770-4], [IEEEP1363.2]) provides extra protection for server compromise in the sense that an attacker, who obtained a password verifier from a server, cannot impersonate the corresponding user without performing off-line dictionary attacks on the password verifier. This additional security is known as "resistance to server compromise". The AugPAKE protocol described in this document is an augmented PAKE which also achieves highly efficiency over the previous works (SRP [RFC2945], [ISOIEC11770-4], and AMP [ISOIEC11770-4]). In summary, 1) The AugPAKE protocol is secure against passive attacks, active attacks and off-line dictionary attacks on the obtained messages with passive/active attacks (see [SKI10] for the formal security proof), and 2) It provides resistance to server compromise. At the same time, the AugPAKE protocol has similar computational efficiency to the plain Diffie-Hellman key exchange [DH76] that does not provide authentication by itself. Specifically, the user and the server need to compute 2 and 2.17 modular exponentiations, respectively, in the AugPAKE protocol. After excluding pre-computable costs, the user and the server are required to compute only 1 and 1.17 modular exponentiations, respectively. Compared with SRP [RFC2945], [ISOIEC11770-4], and AMP [ISOIEC11770-4], the AugPAKE protocol is more efficient 1) than SRP in terms of the user's computational costs and 2) than AMP in terms of the server's computational costs.

1.1. Keywords

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. AugPAKE Specification

2.1. Underlying Group

The AugPAKE protocol can be implemented over the following group.

- o Let p and q be sufficiently large primes such that q is a divisor of ((p - 1) / 2) and every factors of ((p - 1) / 2) are also primes comparable to q in size. This p is called a "secure" prime. We denote by G a multiplicative subgroup of prime order q over the field GF(p), the integers modulo p. Let g be a generator for the subgroup G so that all the subgroup elements are generated by g. The group operation is denoted multiplicatively (in modulo p).

By using a secure prime p, the AugPAKE protocol has computational efficiency gains. Specifically, it does not require the order check of elements, received from the counterpart party. Note that the groups, defined in Discrete Logarithm Cryptography [SP800-56A] and RFC 5114 [RFC5114], are not necessarily the above secure prime groups.

Alternatively, one can implement the AugPAKE protocol over the following groups.

- o Let p and q be sufficiently large primes such that p = (2 * q) + 1. This p is called a "safe" prime. We denote by G a multiplicative subgroup of prime order q over the field GF(p), the integers modulo p. Let g be any element of G other than 1. For example, g = h^2 mod p where h is a primitive element. The group operation is denoted multiplicatively (in modulo p).

- o Let p and q be sufficiently large primes such that q is a divisor of ((p - 1) / 2). We denote by G a multiplicative subgroup of prime order q over the field GF(p), the integers modulo p. Let g be a generator for the subgroup G so that all the subgroup elements are generated by g. The group operation is denoted multiplicatively (in modulo p). If p is not a "secure" prime, the AugPAKE protocol MUST perform the order check of received elements.

2.2. Notation

The AugPAKE protocol is a two-party protocol where a user and a server authenticate each other and generate a session key. The following notation is used in this document:

U
: The user's identity (e.g., defined in [RFC4282]). It is a string in {0,1}^* where {0,1}^* indicates a set of finite binary strings.

S
: The server's identity. It is a string in {0,1}^*.

b = H(a)
: A binary string a is given as input to a secure one-way hash function H (e.g., SHA-2 family [FIPS180-3]) which produces a fixed-length output b. The hash function H maps {0,1}^* to {0,1}^k where {0,1}^k indicates a set of binary strings of length k and k is a security parameter.

b = H'(a)
: A binary string a is given as input to a secure one-way hash function H' which maps the input a in {0,1}^* to the output b in Z_q^* where Z_q^* is a set of positive integers modulo prime q.

a | b
: It denotes a concatenation of binary strings a and b in {0,1}^*.

0x
: A hexadecimal value is shown preceded by "0x".

X * Y mod p
: It indicates a multiplication of X and Y modulo prime p.

X = g^x mod p
: The g^x indicates a multiplication computation of g by x times. The resultant value modulo prime p is assigned to X. The discrete logarithm problem says that it is computationally hard to compute the discrete logarithm x from X, g and p.

w
: The password remembered by the user. This password may be used as an effective password (instead of itself) in the form of H'(0x00 | U | S | w).

W

The password verifier registered in the server. This password verifier is computed as follows: $W = g^{w} \bmod p$ where the user's password w is used itself, or $W = g^{w'} \bmod p$ where the effective password $w' = H'(0x00 \mid U \mid S \mid w)$ is used.

bn2bin(X)
: It indicates a conversion of a multiple precision integer X to the corresponding binary string. If X is an element over GF(p), its binary representation MUST have the same bit length as the binary representation of prime p.

U -> S: msg
: It indicates a message transmission that the user U sends a message msg to the server S.

U:
: It indicates a local computation of user U (without any out-going messages).

2.2.1. Password Processing

The input password MUST be processed according to the rules of the [RFC4013] profile of [RFC3454]. The password SHALL be considered a "stored string" per [RFC3454] and unassigned code points are therefore prohibited. The output SHALL be the binary representation of the processed UTF-8 character string. Prohibited output and unassigned code points encountered in SASLprep pre-processing SHALL cause a failure of pre-processing and the output SHALL NOT be used with the AugPAKE protocol.

The following table shows examples of how various character data is transformed by the rules of the [RFC4013] profile.

#	Input	Output	Comments
1	I<U+00AD>X	IX	SOFT HYPHEN mapped to nothing
2	user	user	no transformation
3	USER	USER	case preserved, will not match #2
4	<U+00AA>	a	output is NFKC, input in ISO 8859-1
5	<U+2168>	IX	output is NFKC, will match #1
6	<U+0007>		Error - prohibited character
7	<U+0627><U+0031>		Error - bidirectional check

2.3. Protocol

The AugPAKE protocol consists of two phases: initialization and actual protocol execution. The initialization phase SHOULD be

finished in a secure manner between the user and the server, and it is performed all at once. Whenever the user and the server need to establish a secure channel, they can run the actual protocol execution through an open network (i.e., the Internet) in which an active attacker exists.

2.3.1. Initialization

U -> S: (U, W)

The user U computes W = g^w mod p (instead of w, the effective password w’ may be used), and transmits W to the server S. The W is registered in the server as the password verifier of user U. Of course, user U just remembers the password w only.

If resistance to server compromise is not necessary, the server can store w’ instead of W. In either case, server S SHOULD NOT store any plaintext passwords.

As noted above, this phase SHOULD be performed securely and all at once.

2.3.2. Actual Protocol Execution

The actual protocol execution of the AugPAKE protocol allows the user and the server to share an authenticated session key through an open network (see Figure 1).

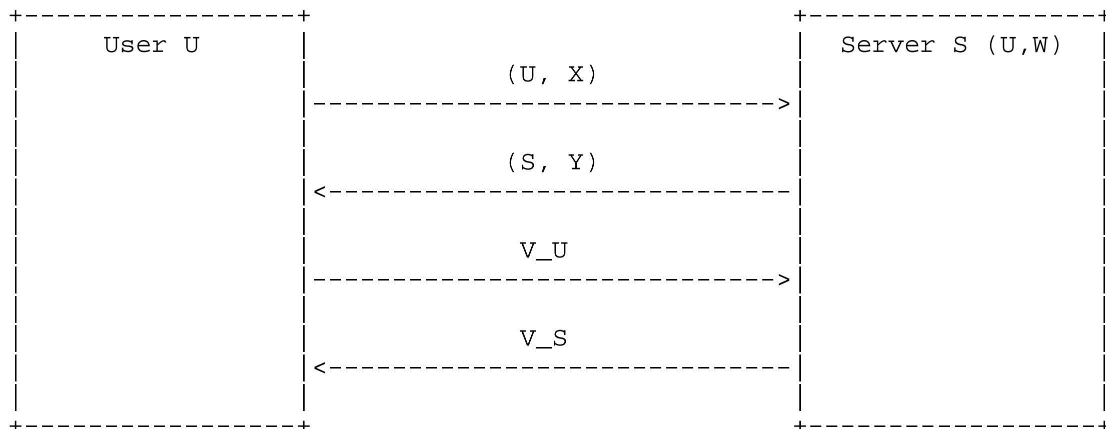


Figure 1: Actual Protocol Execution of AugPAKE

U -> S: (U, X)

The user U chooses a random element x from Z_q^* and computes its Diffie-Hellman public value X = g^x mod p. The user sends the first message (U, X) to the server S.

S -> U: (S, Y)

If the received X from user U is 0, 1 or -1 (mod p), server S MUST terminate the protocol execution. Otherwise, the server chooses a random element y from Z_q^* and computes Y = (X * (W^r))^y mod p where r = H'(0x01 | U | S | bn2bin(X)). Note that X^y * g^(w * r * y) mod p can be computed from y and (w * r * y) efficiently using Shamir's trick [MOV97]. Then, server S sends the second message (S, Y) to the user U.

U -> S: V_U

If the received Y from server S is 0, 1 or -1 (mod p), user U MUST terminate the protocol execution. Otherwise, the user computes K = Y^z mod p where z = 1 / (x + (w * r)) mod q and r = H'(0x01 | U | S | bn2bin(X)). Also, user U generates an authenticator V_U = H(0x02 | U | S | bn2bin(X) | bn2bin(Y) | bn2bin(K)). Then, the user sends the third message V_U to the server S.

S -> U: V_S

If the received V_U from user U is not equal to H(0x02 | U | S | bn2bin(X) | bn2bin(Y) | bn2bin(K)) where K = g^y mod p, server S MUST terminate the protocol execution. Otherwise, the server generates an authenticator V_S = H(0x03 | U | S | bn2bin(X) | bn2bin(Y) | bn2bin(K)) and a session key SK = H(0x04 | U | S | bn2bin(X) | bn2bin(Y) | bn2bin(K)). Then, server S sends the fourth message V_S to the user U.

U:

If the received V_S from server S is not equal to H(0x03 | U | S | bn2bin(X) | bn2bin(Y) | bn2bin(K)), user U MUST terminate the protocol execution. Otherwise, the user generates a session key SK = H(0x04 | U | S | bn2bin(X) | bn2bin(Y) | bn2bin(K)).

In the actual protocol execution, the sequential order of message exchanges is very important in order to avoid any possible attacks. For example, if the server S sends the second message (S, Y) and the fourth message V_S together, any attacker can easily derive the correct password w with off-line dictionary attacks.

The session key SK, shared only if the user and the server authenticate each other successfully, MAY be generated by using a key derivation function (KDF) [SP800-108]. After generating SK, the user and the server MUST delete all the internal states (e.g., Diffie-Hellman exponents x and y) from memory.

For the formal proof [SKI10] of the AugPAKE protocol, we need to change slightly the computation of Y (in the above S -> U: (S, Y)) and K (in the above S -> U: V_S) as follows: Y = (X * (W^r))^y' and K = g^y' where y' = H'(0x05 | bn2bin(y)).

3. Security Considerations

This section shows why the AugPAKE protocol (i.e., the actual protocol execution) is secure against passive attacks, active attacks and off-line dictionary attacks, and also provides resistance to server compromise.

3.1. General Assumptions

- An attacker is computationally-bounded.
- Any hash functions, used in the AugPAKE protocol, are secure in terms of pre-image resistance (one-wayness), second pre-image resistance and collision resistance.

3.2. Security against Passive Attacks

An augmented PAKE protocol is said to be secure against passive attacks in the sense that an attacker, who eavesdrops the exchanged messages, cannot compute an authenticated session key (shared between the honest parties in the protocol).

In the AugPAKE protocol, an attacker can get the messages (U, X), (S, Y), V_U, V_S by eavesdropping, and then wants to compute the session key SK. That is, the attacker's goal is to derive the correct K from the obtained messages X and Y because the hash functions are secure and the only secret in the computation of SK is K = g^y mod p. Note that

X =   g^x mod p and

Y =   (X * (W^r))^y = X^y * W^(r * y) = X^y * (g^y)^t = X^y * K^t

hold where t = w * r mod q. Though t is determined from possible password candidates and X, the only way for the attacker to extract K from X and Y is to compute X^y. However, the probability for the attacker to compute X^y is negligible in the security parameter for

the underlying groups since both x and y are random elements chosen from Z_q^*. Therefore, the AugPAKE protocol is secure against passive attacks.

3.3. Security against Active Attacks

An augmented PAKE protocol is said to be secure against active attacks in the sense that an attacker, who completely controls the exchanged messages, cannot compute an authenticated session key (shared with the honest party in the protocol) with the probability better than that of on-line dictionary attacks. In other words, the probability for an active attacker to compute the session key is restricted by the on-line dictioinary attacks where it grows linearly to the number of interactions with the honest party.

In the AugPAKE protocol, the user (resp., the server) computes the session key SK only if the received authenticator V_S (resp., V_U) is valid. There are three cases to be considered in the active attacks.

3.3.1. Impersonation Attacks on User U

When an attacker impersonates the user U, the attacker can compute the same SK (to be shared with the server S) only if the authenticator V_U is valid. For a valid authenticator V_U, the attacker has to compute the correct K from X and Y because the hash functions are secure. In this impersonation attack, the attacker of course knows the discrete logarithm x of X and guesses a password w'' from the password dictionary. So, the probability for the attacker to compute the correct K is bounded by the probability of w = w''. That is, this impersonation attack is restricted by the on-line dictionary attacks where the attacker can try a guessed password communicating with the honest server S. Therefore, the AugPAKE protocol is secure against impersonation attacks on user U.

3.3.2. Impersonation Attacks on Server S

When an attacker impersonates the server S, the attacker can compute the same SK (to be shared with the user U) only if the authenticator V_S is valid. For a valid authenticator V_S, the attacker has to compute the correct K from X and Y because the hash functions are secure. In this impersonation attack, the attacker chooses a random element y and guesses a password w'' from the password dictionary so that

```
Y =     (X * (W'^r))^y = X^y * W'^(r * y) = X^y * (g^y)^t'
```

where t' = w'' * r mod q. The probability for the attacker to compute the correct K is bounded by the probability of w = w''.

Also, the attacker knows whether the guessed password is equal to w or not by seeing the received authenticator V_U. However, when w is not equal to w'', the probability for the attacker to compute the correct K is negligible in the security parameter for the underlying groups since the attacker has to guess the discrete logarithm x (chosen by user U) as well. That is, this impersonation attack is restricted by the on-line dictionary attacks where the attacker can try a guessed password communicating with the honest user U. Therefore, the AugPAKE protocol is secure against impersonation attacks on server S.

3.3.3. Man-in-the-Middle Attacks

When an attacker performs the man-in-the-middle attack, the attacker can compute the same SK (to be shared with the user U or the server S) only if one of the authenticators V_U, V_S is valid. Note that if the attacker relays the exchanged messages honestly, it corresponds to the passive attacks. In order to generate a valid authenticator V_U or V_S, the attacker has to compute the correct K from X and Y because the hash functions are secure. So, the attacker is in the same situation as discussed above. Though the attacker can test two passwords (one with user U and the other with server S), it does not change the fact that this attack is restricted by the on-line dictionary attacks where the attacker can try a guessed password communicating with the honest party. Therefore, the AugPAKE protocol is also secure against man-in-the-middle attacks.

3.4. Security against Off-line Dictionary Attacks

An augmented PAKE protocol is said to be secure against off-line dictionary attacks in the sense that an attacker, who completely controls the exchanged messages, cannot reduce the possible password candidates better than on-line dictionary attacks. Note that, in the on-line dictionary attacks, an attacker can test one guessed password by running the protocol execution (i.e., communicating with the honest party).

As discussed in Section 3.2, an attacker in the passive attacks does not compute X^y (and the correct K = g^y mod p) from the obtained messages X, Y. This security analysis also indicates that, even if the attacker can guess a password, the K is derived independently from the guessed password. Next, we consider an active attacker whose main goal is to perform the off-line dictionary attacks in the AugPAKE protocol. As in Section 3.3, the attacker can 1) test one guessed password by impersonating the user U or the server S, or 2) test two guessed passwords by impersonating the server S (to the honest user U) and impersonating the user U (to the honest server S) in the man-in-the-middle attacks. Whenever the honest party receives

an invalid authenticator, the party terminates the actual protocol execution without sending any message. In fact, this is important to prevent an attacker from testing more than one password in the active attacks. Since passive attacks and active attacks cannot remove the possible password candidates efficiently than on-line dictionary attacks, the AugPAKE protocol is secure against off-line dictionary attacks.

3.5. Resistance to Server Compromise

We consider an attacker who has obtained a (user's) password verifier from a server. In the (augmented) PAKE protocols, there are two limitations [BJKMRSW00]: 1) the attacker can find out the correct password from the password verifier with the off-line dictionary attacks because the verifier has the same entropy as the password; and 2) if the attacker impersonates the server with the password verifier, this attack is always possible because the attacker has enough information to simulate the server. An augmented PAKE protocol is said to provide resistance to server compromise in the sense that the attacker cannot impersonate the user without performing off-line dictionary attacks on the password verifier.

In order to show resistance to server compromise in the AugPAKE protocol, we consider an attacker who has obtained the password verifier W and then tries to impersonate the user U without off-line dictionary attacks on W. As a general attack, the attacker chooses two random elements c and d from Z_q^*, and computes

```
X =     (g^c) * (W^d) mod p
```

and sends the first message (U, X) to the server S. In order to impersonate user U successfully, the attacker has to compute the correct K = g^y mod p where y is randomly chosen by server S. After receiving Y from the server, the attacker's goal is to find out a value e satisfying Y^e = K mod p. That is,

```
        log_g (Y^e) = log_g K mod q

        (c + (w * d) + (w * r)) * y * e = y mod q

        (c + w * (d + r)) * e = 1 mod q
```

where log_g K indicates the logarithm of K to the base g. Since there is no off-line dictionary attacks on W, the above solution is that e = 1 / c mod q and d = -r mod q. However, the latter is not possible since r is determined by X (i.e., r = H'(0x01 | U | S | bn2bin(X))) and H' is a secure hash function. Therefore, the AugPAKE protocol provides resistance to server compromise.

4. Implementation Consideration

As discussed in Section 3, the AugPAKE protocol is secure against passive attacks, active attacks and off-line dictionary attacks, and provides resistance to server compromise. However, an attacker in the on-line dictionary attacks can check whether one password (guessed from the password dictionary) is correct or not by interacting with the honest party. Let N be a dictionary size of passwords. Certainly, the attacker's success probability grows with the probability of (I / N) where I is the number of interactions with the honest party. In order to provide a reasonable security margin, implementation SHOULD take a countermeasure to the on-line dictionary attacks. For example, it would take about 90 years to test 2^(25.5) passwords with one minute lock-out for 3 failed password guesses (see Appendix A in [SP800-63]).

5. IANA Considerations

This document has no request to IANA.

6. References

6.1. Normative References

[FIPS180-3]
Information Technology Laboratory, , "Secure Hash Standard (SHS)", NIST FIPS Publication 180-3, October 2008, <http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3454] Hoffman, P. and M. Blanchet, "Preparation of Internationalized Strings ("stringprep")", RFC 3454, December 2002.

[RFC4013] Zeilenga, K., "SASLprep: Stringprep Profile for User Names and Passwords", RFC 4013, February 2005.

[RFC4282] Aboba, B., Beadles, M., Arkko, J., and P. Eronen, "The Network Access Identifier", RFC 4282, December 2005.

[SP800-108]
Chen, L., "Recommendation for Key Derivation Using Pseudorandom Functions (Revised)", NIST Special Publication 800-108, October 2009, <http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf>.

6.2. Informative References

[BJKMRSW00]
Bellare, M., Jablon, D., Krawczyk, H., MacKenzie, P., Rogaway, P., Swaminathan, R., and T. Wu, "Proposal for P1363 Study Group on Password-Based Authenticated-Key-Exchange Methods", IEEE P1363.2: Password-Based Public-Key Cryptography , Submissions to IEEE P1363.2 , February 2000, <http://grouper.ieee.org/groups/1363/passwdPK/contributions/p1363-pw.pdf>.

[BM92] Bellovin, S. M. and M. Merritt, "Encrypted Key Exchange: Password-based Protocols Secure against Dictionary Attacks", Proceedings of the IEEE Symposium on Security and Privacy , IEEE Computer Society , 1992.

[BM93] Bellovin, S. M. and M. Merritt, "Augmented Encrypted Key Exchange: A Password-based Protocol Secure against Dictionary Attacks and Password File Compromise", Proceedings of the 1st ACM Conference on Computer and Communication Security , ACM Press , 1993.

[DH76] Diffie, W. and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory Volume IT-22, Number 6, 1976.

[IEEE1363.2]
IEEE 1363.2, , "IEEE Standard Specifications for Password-Based Public-Key Cryptographic Techniques", IEEE Std 1363.2, IEEE Computer Society , January 2009.

[IEEEP1363.2]
IEEE P1363.2, , "Password-Based Public-Key Cryptography", Submissions to IEEE P1363.2 , , <http://grouper.ieee.org/groups/1363/passwdPK/submissions.html>.

[ISOIEC11770-4]
ISO/IEC 11770-4, , "Information technology -- Security techniques -- Key management -- Part 4: Mechanisms based on weak secrets", International Standard ISO/IEC 11770-4:2006, May 2006, <http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39723>.

[MOV97] Menezes, A. J., Oorschot, P. C., and S. A. Vanstone, "Simultaneous Multiple Exponentiation", in Handbook of Applied Cryptography , CRC Press , 1997.

[RFC2945] Wu, T., "The SRP Authentication and Key Exchange System", RFC 2945, September 2000.

[RFC5114] Lepinski, M. and S. Kent, "Additional Diffie-Hellman Groups for Use with IETF Standards", RFC 5114, January 2008.

[SKI10] Shin, S. H., Kobara, K., and H. Imai, "Security Proof of AugPAKE", Cryptology ePrint Archive: Report 2010/334, June 2010, <http://eprint.iacr.org/2010/334>.

[SP800-56A] Barker, E., Johnson, D., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised)", NIST Special Publication 800-56A, March 2007, <http://csrc.nist.gov/publications/nistpubs/800-56A/SP800-56A_Revision1_Mar08-2007.pdf>.

[SP800-63] Burr, W. E., Dodson, D. F., and W. T. Polk, "Electronic Authentication Guideline", NIST Special Publication 800-63 Version 1.0.2, April 2006, <http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63V1_0_2.pdf>.

Appendix A. Features of AugPAKE

Below are some features of the AugPAKE protocol.

Security:

- AugPAKE is zero knowledge (password) proof. It is secure against passive/active/off-line dictionary attacks. It is also resistant to server-compromise impersonation attacks.
- AugPAKE provides Perfect Forward Secrecy (PFS) and is secure against Denning-Sacco attack.
- Any cryptographically secure Diffie-Hellman groups can be used.
- The formal security proof of AugPAKE can be found at [SKI10].
- AugPAKE can be easily used with strong credentials.
- In the case of server compromise, an attacker has to perform off-line dictionary attacks while computing modular exponentiation with a password candidate.

Intellectual Property:

- o AugPAKE was publicly disclosed on Oct. 2008.

- o AIST applied for a patent in Japan on July 10, 2008. AIST would provide royal-free license of AugPAKE.

- o IPR disclosure (see https://datatracker.ietf.org/ipr/2037/)

Miscellaneous:

- o The user needs to compute only 2 modular exponentiation computations while the server needs to compute 2.17 modular exponentiation computations. AugPAKE needs to exchange 2 group elements and 2 hash values. This is almost the same computation/ communication costs as the plain Diffie-Hellman key exchange. If we use a large (e.g., 2048/3072-bits) parent group, the hash size would be relatively small.

- o AugPAKE can be easily converted to 'balanced' PAKE.

- o AugPAKE has the same performance for any type of secret.

- o Internationalization of character-based passwords can be supported.

- o AugPAKE can be implemented over any ECP (Elliptic Curve Group over GF[P]), EC2N (Elliptic Curve Group over GF[2^N]), and MODP (Modular Exponentiation Group) groups.

- o AugPAKE has request/response nature.

- o No Trusted Third Party (TTP) and clock synchronization

- o No additional primitive (e.g., Full Domain Hash (FDH) and/or ideal cipher) is needed.

- o Easy implementation. We already implemented AugPAKE and have been testing in AIST.

Appendix B. Test Vector of AugPAKE

Here is a test vector of the AugPAKE protocol.

```
p = 0x FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
       4300000000000000000000000000000000000000000000000000000000000000
       0000000000000000000000000000000000000000000000000000000000000000
       0000000000000000000000000000000000000000000000000000000000000000
```

```
          0000000000000000000000000000000000000000000000000000000000000000
          0000000000000000000000000000000000000000000000000000000000000000
          0000000000000000000000000000000000000000000000000000000000000000
          0000000000000000000000000000000000000000000000000000000000000000
          0000000000000000000000000000000000000000000000000000000000000000
          0000000000000000000000000000000000000000000000000000000000000000
          0000000000000000000000000000000000000000000000000000000000000000
          0000000000000000330A0DFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
          FFFFFFFFFFFFFFFFDA5193AB

   q = 0x FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
          43

   g = 0x F1AC99884ABBBBCC9BAA19BF375607FD14570B3019A03871147032445ADA7F
          A5B8BDC399C1889BBDA197ADB1E3939D55361241F5CD5ED529B0ADD921B274
          44BD2EB698DC962A9F7D202EAB98BC0C8CC950CA13BC6B1E632D0876A4E796
          26FDE85F06A46C9991EB02A6D6096E0DF6BCA2CAA12E838BEC47A7CB4AF2B0
          D94107B9CDBD67327238ECAF84DF292E776AF0F76288B39F9D9E4DDF3A9731
          CC832D70F150A0F29E7A1E193D1D21CBE8A84B56B0A4692CB39D3048086782
          85A23F08F9DB402487746F7E2A19CAF2171E55C76337E359217516213FF3BF
          616F8B20586A8B3168DA444AEA862BB76B9EA2BF8CB84773D29D4EFE511C53
          95F89CB547EFBBAE333E0BDB22DA40CE0B942A59841A12790910CC1332699D
          64BBF667E0DF3791C4E29CEB48E8397D50C72F7765C5A18809E3497F6BD374
          F5D185BBC8F57E36051E11E8DD0C5DD385A9DA442F22598111960CC2B83CBA
          0A1D980745562F6C62DD6D81B7BAEA7650B1E6E57AB9CC4C95EF17256A79B1
          31859E1BAC81FF1E

   U =    augpakeuser@aist.go.jp

   S =    augpakeserver@aist.go.jp

   X = 0x AF261CB6FD99E2E3B4FCDE8A9C538A872EBB54CC82845E8038EA34804DD739
          90689F2191BF6436BFFD13459DC5FF1FBD56734CA6347ADF52AFAB85EC98A0
          4D27C122EBAF1840229C2F74E9AEEB79C27ECAF6C8B88FCD5FB3F9C8F20F8F
          88D45CEBB56D6CAB251DC34D3656DB73E4DF77B71420565D4696DBF8D27023
          36E909F4C33B2F14605BDEB535423F02B779D7AF0CCB4F3705E177C4334F14
          707140218197845C708E6ABC025BCA954AA2FDAA352D284DE41D33F7CCFCFD
          99CAB3CFD34361B6D0551C6D8AEE604316F99758A9C58A96C39AA811E8BA10
          2E4D21D8F5EFC26920B3F95E6C800D8EBE9914B53144B038BE0E0B43D86479
          E3CBE623FD148CC90784548C1C7C522C6C71FD4A095B15CBF3EB74C7759493
          38AC7691A3763C0A3AB97E9E2B439192C49470BF90F292C4A8D336D86780FD
          DC7DC3C2CC04A6E3AB632788D97D4FDDEE5B117DCF76D91CEA1794FAD91CA1
          7C9715D83589A00CBB72DD513F6A30E4DB31ACA8DA5D6ECD0E20307D70D6DD
          0C2D6D3D7962694B5E1B0848

   Y = 0x 39033C00E1C30FF1A945DD699C516D3BB194FABBB46ADF54A2ED0820FEE0A4
          DE52128E2126A4393E3040E07D90432B8C19433EC920274E5A7C165DC89AA2
          C6D77E801A0C7676BC71955A52E74775C57C63B97304C73211C2035BD7B6D1
```

8844A28AC50DD2DA96B257236A6090CCB08D2006BF57AC69EA14ABC5D71BA0
6A6BD4093A2EF27C74D5D9189BB2EC865E6321D0DFECDE2D9AC537E8254E98
C38AE00BE2554F71FE6EFBBDBE8D7038128957E98C99206998B0B4E578EC47
957205C228FAC57B9A1589B8FF2B134980504F56B8A84809B8FF70EFF67520
2B255F0724DC0F76F3802D8A42ACC33D349A7FAF249BFBFEB324C3966D2B30
6093C32A928A8BEB99AC301D20372E95BB8A3E500778B4651EB8A19B162666
8DDFB77D0DF4C1932F1FE63389F3B1F29AE99F34BC39EF0AD04BC3A6A129DE
E66E50B6768EDECC529F06FE5F7AD3825E8ECFCB12DB579C40F19D12BF6F60
4621F60413DAEB77FE48C136518C57D02A2C6BB596EDFA0DACC127C2FD5FE1
9B72580A722307C3F86C0EB1

V_U = 0x 490C7CE33DCC3EBE8D0406EEB97CA154882DCBBA0A728F3B870263BCA36
9DB6

V_S = 0x D70D2CAA821B9D84E29D75EB5E9B2DB038BA1256ECFC35C553832743A6E
36F

Appendix C. AugPAKE over EC Groups

The AugPAKE protocol can be implemented over elliptic curve groups as follows.

Let p and q be sufficiently large primes, and let m be some positive integer. An elliptic curve E is defined by one of the following two curve equations

$y^2 = x^3 + a * x + b$ over the prime field GF(p) or

$y^2 + x * y = x^3 + a * x^2 + b$ over the binary field GF(2^m)

together with the point at infinity 0_E where x, y, and two coefficients a and b are elements of GF(p) or GF(2^m). Let #E be the number of points on E, and prime q be the order of the desired group. The cofactor k is the value (#E / q) satisfying k = 2^n * q_1 * q_2 ... q_t where n = {0,1,2,3} and every primes q_i > q for i = 1, 2, ..., t. Optionally, k = 2^n. Also, n can be 3 for good performance and security. Let G be a generator for a subgroup of q points on E so that all the subgroup elements are generated by G. The group operation is denoted additively. For example, (X = [x] * G) indicates that an addition computation of G by x times and the resultant value is assigned to X.

By using the above elliptic curve groups, the AugPAKE protocol has computational efficiency gains. Specifically, it does not require the order check of elements, received from the counterpart party.

The AugPAKE protocol consists of two phases: initialization and actual protocol execution. The initialization phase SHOULD be

finished in a secure manner between the user and the server, and it is performed all at once. Whenever the user and the server need to establish a secure channel, they can run the actual protocol execution through an open network (i.e., the Internet) in which an active attacker exists.

Initialization

U -> S: (U, W)
The user U computes W = [w] * G (instead of w, the effective password w' may be used), and transmits W to the server S. The W is registered in the server as the password verifier of user U. Of course, user U just remembers the password w only.

Actual Protocol Execution

U -> S: (U, X)
The user U chooses a random element x from Z_q^* and computes its elliptic curve Diffie-Hellman public value X = [x] * G. The user sends the first message (U, X) to the server S.

S -> U: (S, Y)
If the received X from user U is not a point on E or [2^n] * X = 0_E, server S MUST terminate the protocol execution. Otherwise, the server chooses a random element y from Z_q^* and computes Y = [y] * (X + ([r] * W)) where r = H'(0x01 | U | S | bn2bin(X)). Then, server S sends the second message (S, Y) to the user U.

U -> S: V_U
If the received Y from server S is not a point on E or [2^n] * Y = 0_E, user U MUST terminate the protocol execution. Otherwise, the user computes K = [z] * Y where z = 1 / (x + (w * r)) mod q and r = H'(0x01 | U | S | bn2bin(X)). Also, user U generates an authenticator V_U = H(0x02 | U | S | bn2bin(X) | bn2bin(Y) | bn2bin(K)). Then, the user sends the third message V_U to the server S.

S -> U: V_S
If the received V_U from user U is not equal to H(0x02 | U | S | bn2bin(X) | bn2bin(Y) | bn2bin(K)) where K = [y] * G, server S MUST terminate the protocol execution. Otherwise, the server generates an authenticator V_S = H(0x03 | U | S | bn2bin(X) | bn2bin(Y) | bn2bin(K)) and a session key SK = H(0x04 | U | S | bn2bin(X) | bn2bin(Y) |

bn2bin(K)). Then, server S sends the fourth message V_S to the user U.

U:

If the received V_S from server S is not equal to H(0x03 | U | S | bn2bin(X) | bn2bin(Y) | bn2bin(K)), user U MUST terminate the protocol execution. Otherwise, the user generates a session key SK = H(0x04 | U | S | bn2bin(X) | bn2bin(Y) | bn2bin(K)).

Authors' Addresses

SeongHan Shin
AIST
Central 2, 1-1-1, Umezono
Tsukuba, Ibaraki 305-8568
JP

Phone: +81 29-861-5284
Email: seonghan.shin@aist.go.jp

Kazukuni Kobara
AIST

Email: kobara_conf-ml@aist.go.jp

CFRG
Internet-Draft
Intended status: Informational
Expires: August 1, 2015

A. Langley
Google
R. Salz
Akamai Technologies
S. Turner
IECA, Inc.

January 28, 2015

Elliptic Curves for Security
draft-irtf-cfrg-curves-01

Abstract

This memo describes an algorithm for deterministically generating parameters for elliptic curves over prime fields offering high practical security in cryptographic applications, including Transport Layer Security (TLS) and X.509 certificates. It also specifies a specific curve at the ~128-bit security level.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 1, 2015.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 2
2. Requirements Language 3
3. Security Requirements 3
4. Notation 3
5. Parameter Generation 4
  5.1. Edwards Curves 4
  5.2. Twisted Edwards Curves 5
6. Recommended Curves 6
7. The curve25519 function 7
  7.1. Test vectors 10
8. Diffie-Hellman 11
  8.1. Test vectors 11
9. Acknowledgements 11
10. References 12
  10.1. Normative References 12
  10.2. Informative References 12
Authors' Addresses 13

1. Introduction

Since the initial standardization of elliptic curve cryptography (ECC) in [SEC1] there has been significant progress related to both efficiency and security of curves and implementations. Notable examples are algorithms protected against certain side-channel attacks, various 'special' prime shapes which allow faster modular arithmetic, and a larger set of curve models from which to choose. There is also concern in the community regarding the generation and potential weaknesses of the curves defined in [NIST].

This memo describes a deterministic algorithm for generating cryptographic elliptic curves over a given prime field. The constraints in the generation process produce curves that support constant-time, exception-free scalar multiplications that are resistant to a wide range of side-channel attacks including timing and cache attacks, thereby offering high practical security in cryptographic applications. The deterministic algorithm operates without any input parameters that would permit manipulation of the resulting curves. The selection between curve models is determined by choosing the curve form that supports the fastest (currently known) complete formulas for each modularity option of the underlying field prime. Specifically, the Edwards curve $x^2 + y^2 = 1 + dx^2y^2$

is used with primes p with p = 3 mod 4, and the twisted Edwards curve $-x^2 + y^2 = 1 + dx^2y^2$ is used when p = 1 mod 4.

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

3. Security Requirements

For each curve at a specific security level:

1. The domain parameters SHALL be generated in a simple, deterministic manner, without any secret or random inputs. The derivation of the curve parameters is defined in Section 5.

2. The trace of Frobenius MUST NOT be in {0, 1} in order to rule out the attacks described in [Smart], [AS], and [S], as in [EBP].

3. MOV Degree: the embedding degree k MUST be greater than (r - 1) / 100, as in [EBP].

4. CM Discriminant: discriminant D MUST be greater than 2^{100}, as in [SC].

4. Notation

Throughout this document, the following notation is used:

p Denotes the prime number defining the underlying field.

GF(p) The finite field with p elements.

d An element in the finite field GF(p), not equal to -1 or zero.

Ed An Edwards curve: an elliptic curve over GF(p) with equation $x^2 + y^2 = 1 + dx^2y^2$.

tEd A twisted Edwards curve where a=-1: an elliptic curve over GF(p) with equation $-x^2 + y^2 = 1 + dx^2y^2$.

oddDivisor The largest odd divisor of the number of GF(p)-rational points on a (twisted) Edwards curve.

oddDivisor' The largest odd divisor of the number of GF(p)-rational points on the non-trivial quadratic twist of a (twisted) Edwards curve.

cofactor The cofactor of the subgroup of order oddDivisor in the group of GF(p)-rational points of a (twisted) Edwards curve.

cofactor' The cofactor of the subgroup of order oddDivisor in the group of GF(p)-rational points on the non-trivial quadratic twist of a (twisted) Edwards curve.

trace The trace of Frobenius of Ed or tEd such that #Ed(GF(p)) = p + 1 - trace or #tEd(GF(p)) = p + 1 - trace, respectively.

P A generator point defined over GF(p) of prime order oddDivisor on Ed or tEd.

X(P) The x-coordinate of the elliptic curve point P.

Y(P) The y-coordinate of the elliptic curve point P.

5. Parameter Generation

This section describes the generation of the curve parameter, namely d, of the elliptic curve. The input to this process is p, the prime that defines the underlying field. The size of p determines the amount of work needed to compute a discrete logarithm in the elliptic curve group and choosing a precise p depends on many implementation concerns. The performance of the curve will be dominated by operations in GF(p) and thus carefully choosing a value that allows for easy reductions on the intended architecture is critical. This document does not attempt to articulate all these considerations.

5.1. Edwards Curves

For p = 3 mod 4, the elliptic curve Ed in Edwards form is determined by the non-square element d from GF(p) (not equal to -1 or zero) with smallest absolute value such that #Ed(GF(p)) = cofactor * oddDivisor, #Ed'(GF(p)) = cofactor' * oddDivisor', cofactor = cofactor' = 4, and both subgroup orders oddDivisor and oddDivisor' are prime. In addition, care must be taken to ensure the MOV degree and CM discriminant requirements from Section 3 are met.

These cofactors are chosen because they are minimal.

```
Input: a prime p, with p = 3 mod 4
Output: the parameter d defining the curve Ed
1. Set d = 0
2. repeat
     repeat
       if (d > 0) then
         d = -d
       else
         d = -d + 1
       end if
     until d is not a square in GF(p)

     Compute oddDivisor, oddDivisor', cofactor and cofactor' where #Ed(GF(p)) =
     cofactor * oddDivisor, #Ed'(GF(p)) = cofactor' * oddDivisor', cofactor and
     cofactor' are powers of 2 and oddDivisor, oddDivisor' are odd.
   until ((cofactor = cofactor' = 4), oddDivisor is prime and oddDivisor' is pri
me)
3. Output d
```

GenerateCurveEdwards

5.2. Twisted Edwards Curves

For a prime p = 1 mod 4, the elliptic curve tEd in twisted Edwards form is determined by the non-square element d from GF(p) (not equal to -1 or zero) with smallest absolute value such that #tEd(GF(p)) = cofactor * oddDivisor, #tEd'(GF(p)) = cofactor' * oddDivisor', cofactor = 8, cofactor' = 4 and both subgroup orders oddDivisor and oddDivisor' are prime. In addition, care must be taken to ensure the MOV degree and CM discriminant requirements from Section 3 are met.

These cofactors are chosen so that they are minimal such that the cofactor of the main curve is greater than the cofactor of the twist. For 1 mod 4 primes, the cofactors are never equal. If the cofactor of the twist is larger than the cofactor of the curve, algorithms may be vulnerable to a small-subgroup attack if a point on the twist is incorrectly accepted.

```
Input: a prime p, with p = 1 mod 4
Output: the parameter d defining the curve tEd
1. Set d = 0
2. repeat
     repeat
       if (d > 0) then
         d = -d
       else
         d = -d + 1
       end if
     until d is not a square in GF(p)

     Compute oddDivisor, oddDivisor', cofactor, cofactor' where #tEd(GF(p)) =
     cofactor * oddDivisor, #tEd'(GF(p)) = cofactor' * oddDivisor', cofactor
     and cofactor' are powers of 2 and oddDivisor, oddDivisor' are odd.
   until (cofactor = 8 and cofactor' = 4 and rd is prime and rd' is prime)
3. Output d
```

GenerateCurveTEdwards

6. Recommended Curves

For the ~128-bit security level, the prime 2^255-19 is recommended for performance on a wide-range of architectures. This prime is congruent to 1 mod 4 and the above procedure results in the following twisted Edwards curve, called "intermediate25519":

p 2^255-19

d 121665

order 2^252 + 0x14def9dea2f79cd65812631a5cf5d3ed

cofactor 8

In order to be compatible with widespread existing practice, the recommended curve is an isogeny of this curve. An isogeny is a "renaming" of the points on the curve and thus cannot affect the security of the curve:

p 2^255-19

d 370957059346694393431380835087545651895421138798432190163887855330
85940283555

order 2^252 + 0x14def9dea2f79cd65812631a5cf5d3ed

cofactor 8

X(P) 15112221349535400772501151409588531511454012693041857206046113283949847762202

Y(P) 4631683569492647816942839400347516314130799386625622561578303360316525185596 0

The d value in this curve is much larger than the generated curve and this might slow down some implementations. If this is a problem then implementations are free to calculate on the original curve, with small d, as the isogeny map can be merged into the affine transform without any performance impact.

The latter curve is isomorphic to a Montgomery curve defined by v^2 = u^3 + 486662u^2 + u where the maps are:

```
(u, v) = ((1+y)/(1-y), sqrt(-1)*sqrt(486664)*u/x)
(x, y) = (sqrt(-1)*sqrt(486664)*u/v, (u-1)/(u+1)
```

The base point maps onto the Montgomery curve such that u = 9, v = 14781619447589544791020593568409986887264606134616475288964881837755586237401.

The Montgomery curve defined here is equal to the one defined in [curve25519] and the isomorphic twisted Edwards curve is equal to the one defined in [ed25519].

7. The curve25519 function

The "curve25519" function performs scalar multiplication on the Montgomery form of the above curve. (This is used when implementing Diffie-Hellman.) The function takes a scalar and a u-coordinate as inputs and produces a u-coordinate as output. Although the function works internally with integers, the inputs and outputs are 32-byte strings and this specification defines their encoding.

U-coordinates are elements of the underlying field GF(2^255-19) and are encoded as an array of bytes, u, in little-endian order such that u[0] + 256 * u[1] + 256^2 * u[2] + ... + 256^n * u[n] is congruent to the value modulo p and u[n] is minimal. When receiving such an array, implementations MUST mask the most-significant bit in the final byte. This is done to preserve compatibility with point formats which reserve the sign bit for use in other protocols and to increase resistance to implementation fingerprinting.

For example, the following functions implement this in Python, although the Python code is not intended to be performant nor side-channel free:

```
def decodeLittleEndian(b):
    return sum([b[i] << 8*i for i in range(32)])

def decodeUCoordinate(u):
    u_list = [ord(b) for b in u]
    u_list[31] &= 0x7f
    return decodeLittleEndian(u_list)

def encodeUCoordinate(u):
    u = u % p
    return ''.join([chr((u >> 8*i) & 0xff) for i in range(32)])
```

(EDITORS NOTE: draft-turner-thecurve25519function also says "Implementations MUST reject numbers in the range [2^255-19, 2^255-1], inclusive." but I'm not aware of any implementations that do so.)

Scalars are assumed to be randomly generated bytes. In order to decode 32 bytes into an integer scalar, set the three least significant bits of the first byte and the most significant bit of the last to zero, set the second most significant bit of the last byte to 1 and, finally, decode as little-endian. This means that resulting integer is of the form 2^254 + 8 * {0, 1, ..., 2^(251) - 1}.

```
def decodeScalar(k):
    k_list = [ord(b) for b in k]
    k_list[0] &= 248
    k_list[31] &= 127
    k_list[31] |= 64
    return decodeLittleEndian(k_list)
```

To implement the "curve25519(k, u)" function (where "k" is the scalar and "u" is the u-coordinate) first decode "k" and "u" and then perform the following procedure, taken from [curve25519] and based on formulas from [montgomery]. All calculations are performed in GF(p), i.e., they are performed modulo p. The constant a24 is (486662 - 2) / 4 = 121665.

```
x_1 = u
x_2 = 1
z_2 = 0
x_3 = u
z_3 = 1
swap = 0

For t = 254 down to 0:
    k_t = (k >> t) & 1
    swap ^= k_t
    // Conditional swap; see text below.
    (x_2, x_3) = cswap(swap, x_2, x_3)
    (z_2, z_3) = cswap(swap, z_2, z_3)
    swap = k_t

    A = x_2 + z_2
    AA = A^2
    B = x_2 - z_2
    BB = B^2
    E = AA - BB
    C = x_3 + z_3
    D = x_3 - z_3
    DA = D * A
    CB = C * B
    x_3 = (DA + CB)^2
    z_3 = x_1 * (DA - CB)^2
    x_2 = AA * BB
    z_2 = E * (AA + a24 * E)

// Conditional swap; see text below.
(x_2, x_3) = cswap(swap, x_2, x_3)
(z_2, z_3) = cswap(swap, z_2, z_3)
Return x_2 * (z_2^(p - 2))
```

(TODO: Note the difference in the formula from Montgomery's original paper. See https://www.ietf.org/mail-archive/web/cfrg/current/msg05872.html.)

Finally, encode the resulting value as 32 bytes in little-endian order.

When implementing this procedure, due to the existence of side-channels in commodity hardware, it is important that the pattern of memory accesses and jumps not depend on the values of any of the bits of "k". It is also important that the arithmetic used not leak information about the integers modulo p (such as having b*c be distinguishable from c*c).

The cswap instruction SHOULD be implemented in constant time (independent of "swap") as follows:

```
cswap(swap, x_2, x_3):
      dummy = swap * (x_2 - x_3)
      x_2 = x_2 - dummy
      x_3 = x_3 + dummy
      Return (x_2, x_3)
```

where "swap" is 1 or 0. Alternatively, an implementation MAY use the following:

```
cswap(swap, x_2, x_3):
      dummy = mask(swap) AND (x_2 XOR x_3)
      x_2 = x_2 XOR dummy
      x_3 = x_3 XOR dummy
      Return (x_2, x_3)
```

where "mask(swap)" is the all-1 or all-0 word of the same length as x_2 and x_3, computed, e.g., as mask(swap) = 1 - swap. The latter version is often more efficient.

7.1. Test vectors

```
Input scalar:
  a546e36bf0527c9d3b16154b82465edd62144c0ac1fc5a18506a2244ba449ac4
Input scalar as a number (base 10):
  31029842492115040904895560451863089656472772604678260265531221036453811406496
Input U-coordinate:
  e6db6867583030db3594c1a424b15f7c726624ec26b3353b10a903a6d0ab1c4c
Input U-coordinate as a number:
  34426434033919594451155107781188821651316167215306631574996226621102155684838
Output U-coordinate:
  c3da55379de9c6908e94ea4df28d084f32eccf03491c71f754b4075577a28552

Input scalar:
  4b66e9d4d1b4673c5ad22691957d6af5c11b6421e0ea01d42ca4169e7918ba0d
Input scalar as a number (base 10):
  35156891815674817266734212754503633747128614016119564763269015315466259359304
Input U-coordinate:
  e5210f12786811d3f4b7959d0538ae2c31dbe7106fc03c3efc4cd549c715a493
Input U-coordinate as a number:
  8883857351183929894090759386610649319417338800022198945255395922347792736741
Output U-coordinate:
  95cbde9476e8907d7aade45cb4b873f88b595a68799fa152e6f8f7647aac7957
```

8. Diffie-Hellman

The "curve25519" function can be used in an ECDH protocol as follows:

Alice generates 32 random bytes in f[0] to f[31] and transmits K_A = curve25519(f, 9) to Bob, where 9 is the u-coordinate of the base point and is encoded as a byte with value 9, followed by 31 zero bytes.

Bob similarly generates 32 random bytes in g[0] to g[31] and computes K_B = curve25519(g, 9) and transmits it to Alice.

Alice computes curve25519(f, K_B); Bob computes curve25519(g, K_A) using their generated values and the received input.

Both now share K = curve25519(f, curve25519(g, 9)) = curve25519(g, curve25519(f, 9)) as a shared secret. Alice and Bob can then use a key-derivation function, such as hashing K, to compute a key.

Note that this Diffie-Hellman protocol is not contributory, e.g. if the u-coordinate is zero then the output will always be zero. A contributory Diffie-Hellman function would ensure that the output was unpredictable no matter what the peer's input. This is not a problem for the vast majority of cases but, if a contributory function is specifically required, then "curve25519" should not be used.

8.1. Test vectors

```
Alice's private key, f:
  77076d0a7318a57d3c16c17251b26645df4c2f87ebc0992ab177fba51db92c2a
Alice's public key, curve25519(f, 9):
  8520f0098930a754748b7ddcb43ef75a0dbf3a0d26381af4eba4a98eaa9b4e6a
Bob's private key, g:
  5dab087e624a8a4b79e17f8b83800ee66f3bb1292618b6fd1c2f8b27ff88e0eb
Bob's public key, curve25519(g, 9):
  de9edb7d7b7dc1b4d35b61c2ece435373f8343c85b78674dadfc7e146f882b4f
Their shared secret, K:
  4a5d9d5ba4ce2de1728e3bf480350f25e07e21c947d19e3376f09b3c1e161742
```

9. Acknowledgements

This document merges "draft-black-rpgecc-01" and "draft-turner-thecurve25519function-01". The following authors of those documents wrote much of the text and figures but are not listed as authors on this document: Benjamin Black, Joppe W. Bos, Craig Costello, Patrick Longa, Michael Naehrig and Watson Ladd.

The authors would also like to thank Tanja Lange and Rene Struik for their reviews.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

10.2. Informative References

[AS] Satoh, T. and K. Araki, "Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves", 1998.

[EBP] ECC Brainpool, "ECC Brainpool Standard Curves and Curve Generation", October 2005, <http://www.ecc-brainpool.org/download/Domain-parameters.pdf>.

[NIST] National Institute of Standards, "Recommended Elliptic Curves for Federal Government Use", July 1999, <http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf>.

[S] Semaev, I., "Evaluation of discrete logarithms on some elliptic curves", 1998.

[SC] Bernstein, D. and T. Lange, "SafeCurves: choosing safe curves for elliptic-curve cryptography", June 2014, <http://safecurves.cr.yp.to/>.

[SEC1] Certicom Research, "SEC 1: Elliptic Curve Cryptography", September 2000, <http://www.secg.org/collateral/sec1_final.pdf>.

[Smart] Smart, N., "The discrete logarithm problem on elliptic curves of trace one", 1999.

[curve25519] Bernstein, D., "Curve25519 -- new Diffie-Hellman speed records", 2006, <http://www.iacr.org/cryptodb/archive/2006/PKC/3351/3351.pdf>.

[ed25519] Bernstein, D., Duif, N., Lange, T., Schwabe, P., and B. Yang, "High-speed high-security signatures", 2011, <http://ed25519.cr.yp.to/ed25519-20110926.pdf>.

[montgomery]
Montgomery, P., "Speeding the Pollard and elliptic curve methods of factorization", 1983, <http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/S0025-5718-1987-0866113-7.pdf>.

Authors' Addresses

Adam Langley
Google
345 Spear St
San Francisco, CA 94105
US

Email: agl@google.com

Rich Salz
Akamai Technologies
8 Cambridge Center
Cambridge, MA 02142
US

Email: rsalz@akamai.com

Sean Turner
IECA, Inc.
3057 Nutley Street
Suite 106
Fairfax, VA 22031
US

Email: turners@ieca.com

Network Working Group C. Percival
Internet-Draft Tarsnap
Intended status: Informational S. Josefsson
Expires: July 30, 2015 SJD AB
January 26, 2015

The scrypt Password-Based Key Derivation Function
draft-josefsson-scrypt-kdf-02

Abstract

This document specifies the password-based key derivation function scrypt. The function derives one or more secret keys from a secret string. It is based on memory-hard functions which offer added protection against attacks using custom hardware. The document also provides an ASN.1 schema.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on July 30, 2015.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction 2
2. The Salsa20/8 Core Function 3
3. The scryptBlockMix Algorithm 3
4. The scryptROMix Algorithm 4
5. The scrypt Algorithm 5
6. ASN.1 Syntax 6
6.1. ASN.1 Module 7
7. Test Vectors for Salsa20/8 Core 8
8. Test Vectors for scryptBlockMix 8
9. Test Vectors for scryptROMix 9
10. Test Vectors for PBKDF2 with HMAC-SHA-256 9
11. Test Vectors for scrypt 10
12. Copying Conditions 11
13. Acknowledgements 11
14. IANA Considerations 11
15. Security Considerations 11
16. References 11
16.1. Normative References 12
16.2. Informative References 12
Authors' Addresses 12

1. Introduction

Password-based key derivation functions are used in cryptography for deriving one or more secret keys from a secret value. Over the years, several password-based key derivation functions have been used, including the original DES-based UNIX Crypt-function, FreeBSD MD5 crypt, PKCS#5 PBKDF2 [RFC2898] (typically used with SHA-1), GNU SHA-256/512 crypt, Windows NT LAN Manager (NTLM) hash, and the Blowfish-based bcrypt. These algorithms are based on similar techniques that employ a cryptographic primitive, salting and/or iteration. The iteration count is used to slow down the computation.

Providing that the number of iterations used is increased as computer systems get faster, this allows legitimate users to spend a constant amount of time on key derivation without losing ground to an attackers' ever-increasing computing power - as long as attackers are limited to the same software implementations as legitimate users. However, as Bernstein pointed out in the context of integer factorization, while parallelized hardware implementations may not change the number of operations performed compared to software implementations, this does not prevent them from dramatically changing the asymptotic cost, since in many contexts - including the

embarrassingly parallel task of performing a brute-force search for a passphrase - dollar-seconds are the most appropriate units for measuring the cost of a computation. As semiconductor technology develops, circuits do not merely become faster; they also become smaller, allowing for a larger amount of parallelism at the same cost. Consequently, existing key derivation algorithms, even when the iteration count is increased so that the time taken to verify a password remains constant, the cost of finding a password by using a brute force attack implemented in hardware drops each year.

The scrypt function aims to reduce the advantage which attackers can gain by using custom-designed parallel circuits for breaking password-based key derivation functions.

For further background, see the original scrypt paper [SCRYPT].

The rest of this document is divided into sections that each describe algorithms needed for the final "scrypt" algorithm.

2. The Salsa20/8 Core Function

Salsa20/8 Core is a round-reduced variant of the Salsa20 Core. It is a hash function from 64-octet strings to 64-octet strings. Note that Salsa20/8 Core is not a cryptographic hash function since it is not collision-resistant. See section 8 of [SALSA20SPEC] for its specification, and [SALSA20CORE] for more information.

3. The scryptBlockMix Algorithm

The scryptBlockMix algorithm is the same as the BlockMix algorithm described in [SCRYPT] but with Salsa20/8 Core used as the hash function H. Below, Salsa(T) corresponds to the Salsa20/8 Core function applied to the octet vector T.

```
   Algorithm scryptBlockMix

   Parameters:
            r       Block size parameter.

   Input:
            B[0], ..., B[2 * r - 1]
                    Input vector of 2 * r 64-octet blocks.

   Output:
            B'[0], ..., B'[2 * r - 1]
                    Output vector of 2 * r 64-octet blocks.

   Steps:

     1. X = B[2 * r - 1]

     2. for i = 0 to 2 * r - 1 do
          T = X xor B[i]
          X = Salsa (T)
          Y[i] = X
        end for

     3. B' = (Y[0], Y[2], ..., Y[2 * r - 2],
              Y[1], Y[3], ..., Y[2 * r - 1])
```

4. The scryptROMix Algorithm

The scryptROMix algorithm is the same as the ROMix algorithm described in [SCRYPT] but with scryptBlockMix used as the hash function H and the Integerify function explained inline.

```
   Algorithm scryptROMix

   Input:
            r       Block size parameter.
            B       Input octet vector of length 128 * r octets.
            N       CPU/Memory cost parameter, must be larger than 1,
                    a power of 2 and less than 2^(128 * r / 8).

   Output:
            B'      Output octet vector of length 128 * r octets.

   Steps:

     1. X = B

     2. for i = 0 to N - 1 do
          V[i] = X
          X = scryptBlockMix (X)
        end for

     3. for i = 0 to N - 1 do
          j = Integerify (X) mod N
                 where Integerify (B[0] ... B[2 * r - 1]) is defined
                 as the result of interpreting B[2 * r - 1] as a
                 little-endian integer.
          T = X xor V[j]
          X = scryptBlockMix (T)
        end for

     4. B' = X
```

5. The scrypt Algorithm

The PBKDF2-HMAC-SHA-256 function used below denote the PBKDF2 algorithm [RFC2898] used with HMAC-SHA-256 [RFC6234] as the PRF. The HMAC-SHA-256 function generates 32 octet outputs.

```
   Algorithm scrypt

   Input:
            P       Passphrase, an octet string.
            S       Salt, an octet string.
            N       CPU/Memory cost parameter, must be larger than 1,
                    a power of 2 and less than 2^(128 * r / 8).
            r       Block size parameter.
            p       Parallelization parameter, a positive integer
                    less than or equal to ((2^32-1) * hLen) / MFLen
                    where hLen is 32 and MFlen is 128 * r.
            dkLen   Intended output length in octets of the derived
                    key; a positive integer less than or equal to
                    (2^32 - 1) * hLen where hLen is 32.

   Output:
            DK      Derived key, of length dkLen octets.

   Steps:

     1. B[0] || B[1] || ... || B[p - 1] =
          PBKDF2-HMAC-SHA256 (P, S, 1, p * 128 * r)

     2. for i = 0 to p - 1 do
          B[i] = scryptROMix (r, B[i], N)
        end for

     3. DK = PBKDF2-HMAC-SHA256 (P, B[0] || B[1] || ... || B[p - 1],
                                 1, dkLen)
```

6. ASN.1 Syntax

This section defines ASN.1 syntax for the scrypt key derivation function. The intended application of these definitions includes PKCS #8 and other syntax for key management. (Various aspects of ASN.1 are specified in several ISO/IEC standards.)

The object identifier id-scrypt identifies the scrypt key derivation function.

```
id-scrypt OBJECT IDENTIFIER ::= {1 3 6 1 4 1 11591 4 11}
```

The parameters field associated with this OID in an AlgorithmIdentifier shall have type scrypt-params:

```
scrypt-params ::= SEQUENCE {
       salt OCTET STRING,
       costParameter INTEGER (1..MAX),
       blockSize INTEGER (1..MAX),
       parallelizationParameter INTEGER (1..MAX),
       keyLength INTEGER (1..MAX) OPTIONAL }
```

The fields of type scrypt-params have the following meanings:

- salt specifies the salt value. It shall be an octet string.

- costParameter specifies the CPU/Memory cost parameter N.

- blockSize specifies the block size parameter r.

- parallelizationParameter specifies the parallelization parameter.

- keyLength, an optional field, is the length in octets of the derived key. The maximum key length allowed depends on the implementation; it is expected that implementation profiles may further constrain the bounds. This field only provides convenience; the key length is not cryptographically protected.

6.1. ASN.1 Module

For reference purposes, the ASN.1 syntax is presented as an ASN.1 module here.

```
-- scrypt ASN.1 Module

scrypt-0 {1 3 6 1 4 1 11591 4 10}

DEFINITIONS ::= BEGIN

id-scrypt OBJECT IDENTIFIER ::= {1 3 6 1 4 1 11591 4 11}

scrypt-params ::= SEQUENCE {
    salt OCTET STRING,
    costParameter INTEGER (1..MAX),
    blockSize INTEGER (1..MAX),
    parallelizationParameter INTEGER (1..MAX),
    keyLength INTEGER (1..MAX) OPTIONAL
}

END
```

7. Test Vectors for Salsa20/8 Core

Below is a sequence of octets to illustrate input and output values for the Salsa20/8 Core. The octets are hex encoded and whitespace is inserted for readability. The value corresponds to the first input and output pair generated by the first scrypt test vector below.

```
INPUT:
7e 87 9a 21 4f 3e c9 86 7c a9 40 e6 41 71 8f 26
ba ee 55 5b 8c 61 c1 b5 0d f8 46 11 6d cd 3b 1d
ee 24 f3 19 df 9b 3d 85 14 12 1e 4b 5a c5 aa 32
76 02 1d 29 09 c7 48 29 ed eb c6 8d b8 b8 c2 5e

OUTPUT:
a4 1f 85 9c 66 08 cc 99 3b 81 ca cb 02 0c ef 05
04 4b 21 81 a2 fd 33 7d fd 7b 1c 63 96 68 2f 29
b4 39 31 68 e3 c9 e6 bc fe 6b c5 b7 a0 6d 96 ba
e4 24 cc 10 2c 91 74 5c 24 ad 67 3d c7 61 8f 81
```

8. Test Vectors for scryptBlockMix

Below is a sequence of octets to illustrate input and output values for scryptBlockMix. The test vector uses an r value of 1. The octets are hex encoded and whitespace is inserted for readability. The value corresponds to the first input and output pair generated by the first scrypt test vector below.

```
INPUT
B[0] =  f7 ce 0b 65 3d 2d 72 a4 10 8c f5 ab e9 12 ff dd
        77 76 16 db bb 27 a7 0e 82 04 f3 ae 2d 0f 6f ad
        89 f6 8f 48 11 d1 e8 7b cc 3b d7 40 0a 9f fd 29
        09 4f 01 84 63 95 74 f3 9a e5 a1 31 52 17 bc d7

B[1] =  89 49 91 44 72 13 bb 22 6c 25 b5 4d a8 63 70 fb
        cd 98 43 80 37 46 66 bb 8f fc b5 bf 40 c2 54 b0
        67 d2 7c 51 ce 4a d5 fe d8 29 c9 0b 50 5a 57 1b
        7f 4d 1c ad 6a 52 3c da 77 0e 67 bc ea af 7e 89

OUTPUT
B'[0] = a4 1f 85 9c 66 08 cc 99 3b 81 ca cb 02 0c ef 05
        04 4b 21 81 a2 fd 33 7d fd 7b 1c 63 96 68 2f 29
        b4 39 31 68 e3 c9 e6 bc fe 6b c5 b7 a0 6d 96 ba
        e4 24 cc 10 2c 91 74 5c 24 ad 67 3d c7 61 8f 81

B'[1] = 20 ed c9 75 32 38 81 a8 05 40 f6 4c 16 2d cd 3c
        21 07 7c fe 5f 8d 5f e2 b1 a4 16 8f 95 36 78 b7
        7d 3b 3d 80 3b 60 e4 ab 92 09 96 e5 9b 4d 53 b6
        5d 2a 22 58 77 d5 ed f5 84 2c b9 f1 4e ef e4 25
```

9. Test Vectors for scryptROMix

Below is a sequence of octets to illustrate input and output values for scryptROMix. The test vector uses an r value of 1 and an N value of 16. The octets are hex encoded and whitespace is inserted for readability. The value corresponds to the first input and output pair generated by the first scrypt test vector below.

```
INPUT:
B = f7 ce 0b 65 3d 2d 72 a4 10 8c f5 ab e9 12 ff dd
    77 76 16 db bb 27 a7 0e 82 04 f3 ae 2d 0f 6f ad
    89 f6 8f 48 11 d1 e8 7b cc 3b d7 40 0a 9f fd 29
    09 4f 01 84 63 95 74 f3 9a e5 a1 31 52 17 bc d7
    89 49 91 44 72 13 bb 22 6c 25 b5 4d a8 63 70 fb
    cd 98 43 80 37 46 66 bb 8f fc b5 bf 40 c2 54 b0
    67 d2 7c 51 ce 4a d5 fe d8 29 c9 0b 50 5a 57 1b
    7f 4d 1c ad 6a 52 3c da 77 0e 67 bc ea af 7e 89

OUTPUT:
B = 79 cc c1 93 62 9d eb ca 04 7f 0b 70 60 4b f6 b6
    2c e3 dd 4a 96 26 e3 55 fa fc 61 98 e6 ea 2b 46
    d5 84 13 67 3b 99 b0 29 d6 65 c3 57 60 1f b4 26
    a0 b2 f4 bb a2 00 ee 9f 0a 43 d1 9b 57 1a 9c 71
    ef 11 42 e6 5d 5a 26 6f dd ca 83 2c e5 9f aa 7c
    ac 0b 9c f1 be 2b ff ca 30 0d 01 ee 38 76 19 c4
    ae 12 fd 44 38 f2 03 a0 e4 e1 c4 7e c3 14 86 1f
    4e 90 87 cb 33 39 6a 68 73 e8 f9 d2 53 9a 4b 8e
```

10. Test Vectors for PBKDF2 with HMAC-SHA-256

Below is a sequence of octets illustring input and output values for PBKDF2-HMAC-SHA-256. The octets are hex encoded and whitespace is inserted for readability. The test vectors below can be used to verify the PBKDF2-HMAC-SHA-256 [RFC2898] function. The password and salt strings are passed as sequences of ASCII [RFC0020] octets.

```
PBKDF2-HMAC-SHA-256 (P="passwd", S="salt",
                    c=1, dkLen=64) =
55 ac 04 6e 56 e3 08 9f ec 16 91 c2 25 44 b6 05
f9 41 85 21 6d de 04 65 e6 8b 9d 57 c2 0d ac bc
49 ca 9c cc f1 79 b6 45 99 16 64 b3 9d 77 ef 31
7c 71 b8 45 b1 e3 0b d5 09 11 20 41 d3 a1 97 83
```

```
PBKDF2-HMAC-SHA-256 (P="Password", S="NaCl",
                     c=80000, dkLen=64) =
4d dc d8 f6 0b 98 be 21 83 0c ee 5e f2 27 01 f9
64 1a 44 18 d0 4c 04 14 ae ff 08 87 6b 34 ab 56
a1 d4 25 a1 22 58 33 54 9a db 84 1b 51 c9 b3 17
6a 27 2b de bb a1 d0 78 47 8f 62 b3 97 f3 3c 8d
```

11. Test Vectors for scrypt

For reference purposes, we provide the following test vectors for scrypt, where the password and salt strings are passed as sequences of ASCII [RFC0020] octets.

The parameters to the scrypt function below are, in order, the password P (octet string), the salt S (octet string), the CPU/Memory cost parameter N, the block size parameter r, and the parallelization parameter p, and the output size dkLen. The output is hex encoded and whitespace is inserted for readability.

```
scrypt (P="", S="",
        N=16, r=1, p=1, dklen=64) =
77 d6 57 62 38 65 7b 20 3b 19 ca 42 c1 8a 04 97
f1 6b 48 44 e3 07 4a e8 df df fa 3f ed e2 14 42
fc d0 06 9d ed 09 48 f8 32 6a 75 3a 0f c8 1f 17
e8 d3 e0 fb 2e 0d 36 28 cf 35 e2 0c 38 d1 89 06

scrypt (P="password", S="NaCl",
        N=1024, r=8, p=16, dkLen=64) =
fd ba be 1c 9d 34 72 00 78 56 e7 19 0d 01 e9 fe
7c 6a d7 cb c8 23 78 30 e7 73 76 63 4b 37 31 62
2e af 30 d9 2e 22 a3 88 6f f1 09 27 9d 98 30 da
c7 27 af b9 4a 83 ee 6d 83 60 cb df a2 cc 06 40

scrypt (P="pleaseletmein", S="SodiumChloride",
        N=16384, r=8, p=1, dkLen=64) =
70 23 bd cb 3a fd 73 48 46 1c 06 cd 81 fd 38 eb
fd a8 fb ba 90 4f 8e 3e a9 b5 43 f6 54 5d a1 f2
d5 43 29 55 61 3f 0f cf 62 d4 97 05 24 2a 9a f9
e6 1e 85 dc 0d 65 1e 40 df cf 01 7b 45 57 58 87

scrypt (P="pleaseletmein", S="SodiumChloride",
        N=1048576, r=8, p=1, dkLen=64) =
21 01 cb 9b 6a 51 1a ae ad db be 09 cf 70 f8 81
ec 56 8d 57 4a 2f fd 4d ab e5 ee 98 20 ad aa 47
8e 56 fd 8f 4b a5 d0 9f fa 1c 6d 92 7c 40 f4 c3
37 30 40 49 e8 a9 52 fb cb f4 5c 6f a7 7a 41 a4
```

12. Copying Conditions

The authors agree to grant third parties the irrevocable right to copy, use and distribute this entire document or any portion of it, with or without modification, in any medium, without royalty, provided that, unless separate permission is granted, redistributed modified works do not contain misleading author, version, name of work, or endorsement information.

13. Acknowledgements

Text in this document was borrowed from [SCRYPT] and [RFC2898].

Feedback on this document were received from Dmitry Chestnykh, Alexander Klink, Rob Kendrick, Royce Williams Ted Rolle, Jr., and Eitan Adler.

14. IANA Considerations

None.

15. Security Considerations

This document specifies a cryptographic algorithm. The reader must follow cryptographic research of published attacks. ROMix has been proven sequential memory-hard under the Random Oracle model for the hash function. The security of scrypt relies on the assumption that BlockMix with Salsa20/8 Core does not exhibit any "shortcuts" which would allow it to be iterated more easily than a random oracle. For other claims about the security properties see [SCRYPT].

Passwords and other sensitive data, such as intermediate values, may continue to be stored in memory, core dumps, swap areas, etc, for a long time after the implementation has processed them. This makes attacks on the implementation easier. Thus, implementation should consider storing sensitive data in protected memory areas. How to achieve this is system dependent.

By nature and depending on parameters, running the scrypt algorithm may require large amounts of memory. Systems should protect against a denial of service attack resulting from attackers presenting unreasonably large parameters.

16. References

16.1. Normative References

[RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, September 2000.

[RFC6234] Eastlake, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, May 2011.

[SALSA20SPEC] Bernstein, D., "Salsa20 specification", WWW http://cr.yp.to/snuffle/spec.pdf, April 2005.

[SALSA20CORE] Bernstein, D., "The Salsa20 Core", WWW http://cr.yp.to/salsa20.html, March 2005.

16.2. Informative References

[RFC0020] Cerf, V., "ASCII format for network interchange", RFC 20, October 1969.

[SCRYPT] Percival, C., "Stronger key derivation via sequential memory-hard functions", BSDCan’09 http://www.tarsnap.com/scrypt/scrypt.pdf, May 2009.

Authors’ Addresses

Colin Percival
Tarsnap

Email: cperciva@tarsnap.com

Simon Josefsson
SJD AB

Email: simon@josefsson.org
URI: http://josefsson.org/

Crypto Forum Research Group
Internet-Draft
Intended status: Informational
Expires: September 10, 2015

A. Huelsing
TU Eindhoven
D. Butin
TU Darmstadt
S. Gazdag
genua mbH

March 9, 2015

XMSS: Extended Hash-Based Signatures
draft-xmss-00

Abstract

This note describes the eXtended Merkle Signature Scheme (XMSS), a hash-based digital signature system. It follows existing descriptions in scientific literature. The note specifies the WOTS+ one-time signature scheme, a single-tree (XMSS) and a multi-tree variant (XMSS^MT) of XMSS. Both variants use WOTS+ as a main building block. XMSS provides cryptographic digital signatures without relying on the conjectured hardness of mathematical problems. Instead, it is proven that it only relies on the properties of cryptographic hash functions. XMSS provides strong security guarantees and, besides some special instantiations, is even secure when the collision resistance of the underlying hash function is broken. It is suitable for compact implementations, relatively simple to implement, and naturally resists side-channel attacks. Unlike most other signature systems, hash-based signatures withstand attacks using quantum computers.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 10, 2015.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction   3
  1.1. Conventions Used In This Document   5
2. Notation   5
  2.1. Data Types   5
  2.2. Operators   5
  2.3. Functions   6
  2.4. Strings of Base-w Numbers   6
  2.5. Member Functions   7
3. Primitives   8
  3.1. WOTS+ One-Time Signatures   8
    3.1.1. WOTS+ Parameters   8
      3.1.1.1. WOTS+ Hashing Functions   9
    3.1.2. WOTS+ Chaining Function   9
    3.1.3. WOTS+ Private Key   9
    3.1.4. WOTS+ Public Key 10
    3.1.5. WOTS+ Signature Generation 10
    3.1.6. WOTS+ Signature Verification 11
    3.1.7. Pseudorandom Key Generation 12
4. Schemes 12
  4.1. XMSS: eXtended Merkle Signature Scheme 13
    4.1.1. XMSS Parameters 13
    4.1.2. XMSS Hash Functions 14
    4.1.3. XMSS Private Key 14
    4.1.4. L-Trees 14
    4.1.5. TreeHash 15
    4.1.6. XMSS Public Key 15
    4.1.7. XMSS Signature 16
    4.1.8. XMSS Signature Generation 18
    4.1.9. XMSS Signature Verification 19
    4.1.10. Pseudorandom Key Generation 20
    4.1.11. Free Index Handling and Partial Secret Keys 21

4.2. XMSS^MT: Multi-Tree XMSS 21
4.2.1. XMSS^MT Parameters 21
4.2.2. XMSS Algorithms Without Message Hash 22
4.2.3. XMSS^MT Private Key 22
4.2.4. XMSS^MT Public Key 22
4.2.5. XMSS^MT Signature 23
4.2.6. XMSS^MT Signature Generation 24
4.2.7. XMSS^MT Signature Verification 25
4.2.8. Pseudorandom Key Generation 26
4.2.9. Free Index Handling and Partial Secret Keys 26
5. Parameter Sets 27
5.1. Zero Bitmasks 27
5.2. WOTS+ Parameters 28
5.3. XMSS Parameters 29
5.3.1. XMSS Parameters 29
5.3.1.1. XMSS Parameters with AES and SHA3 29
5.3.1.2. XMSS Parameters with SHA3 30
5.3.2. XMSS Parameters With Empty Bitmasks 31
5.4. XMSS^MT Parameters 32
5.4.1. XMSS^MT Parameters 32
5.4.1.1. XMSS^MT Parameters with AES and SHA3 32
5.4.1.2. XMSS^MT Parameters with SHA3 33
5.4.2. XMSS^MT Parameters With Empty Bitmasks 35
6. Rationale 38
7. IANA Considerations 38
8. Security Considerations 49
8.1. Security Proofs 50
8.2. Security Assumptions 51
8.3. Post-Quantum Security 51
9. Acknowledgements 51
10. References 51
10.1. Normative References 51
10.2. Informative References 52
Appendix A. WOTS+ XDR Formats 53
Appendix B. XMSS XDR Formats 55
Appendix C. XMSS^MT XDR Formats 65
Authors' Addresses 87

1. Introduction

A (cryptographic) digital signature scheme provides asymmetric message authentication. The key generation algorithm produces a key pair consisting of a private and a public key. A message is signed using a private key to produce a signature. A message/signature pair can be verified using a public key. A One-Time Signature (OTS) scheme allows us to use a key pair to sign exactly one message securely. A many-time signature system can be used to sign multiple messages.

One-Time Signature schemes, and Many-Time Signature (MTS) schemes composed of them, were proposed by Merkle in 1979 [Merkle79]. They were well-studied in the 1990s and have regained interest from 2006 onwards because of their resistance against quantum-computer-aided attacks. These kinds of signature schemes are called hash-based signature schemes as they are built out of a cryptographic hash function. Hash-based signature schemes generally feature small private and public keys as well as fast signature generation and verification but large signatures and a relatively slow key generation. In addition, they are suitable for compact implementations that benefit various applications and are naturally resistant to most kinds of side-channel attacks.

Some progress has already been made toward standardizing and introducing hash signatures. McGrew and Curcio have published an Internet-Draft [DC14] specifying the "textbook" Lamport-Diffie-Winternitz-Merkle (LDWM) scheme based on early publications. Independently, Buchmann, Dahmen and Huelsing have proposed XMSS [BDH11], the "eXtended Merkle Signature Scheme," offering better efficiency and a modern security proof. Very recently, SPHINCS, a stateless hash-based signature scheme was introduced [BHH15], with the intent of being easier to deploy in current applications. A reasonable next step toward introducing hash signatures would seem to complete the specifications of the basic algorithms - LDWM, XMSS, SPHINCS and/or variants [Kaliski15].

The eXtended Merkle Signature Scheme (XMSS) [BDH11] is the latest hash-based signature scheme. It has the smallest signatures out of such schemes and comes with a multi-tree variant that solves the problem of slow key generation. Moreover, it can be shown that XMSS is secure, making only mild assumptions on the underlying hash function. Especially, it is not required that the cryptographic hash function is collision-resistant for the security of XMSS.

This note describes a single-tree and a multi-tree variant of the eXtended Merkle Signature Scheme (XMSS) [BDH11]. It also describes WOTS+, a variant of the Winternitz OTS scheme introduced in [Huelsing13] that is used by XMSS. The schemes are described with enough specificity to ensure interoperability between implementations.

This note is structured as follows. Notation is introduced in Section 2. Section 3 describes the WOTS+ signature system. Many time signature schemes are defined in Section 4: the eXtended Merkle Signature Scheme (XMSS) in Section 4.1, and its Multi-Tree variant (XMSS^MT) in Section 4.2. Parameter sets are described in Section 5. Section 6 describes the rationale behind choices in this note. The IANA registry for these signature systems is described in Section 7. Finally, security considerations are presented in Section 8.

1.1. Conventions Used In This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Notation

2.1. Data Types

Bytes and byte strings are the fundamental data types. A byte is a sequence of eight bits. A single byte is denoted as a pair of hexadecimal digits with a leading "0x". A byte string is an ordered sequence of zero or more bytes and is denoted as an ordered sequence of hexadecimal characters with a leading "0x". For example, 0xe534f0 is a byte string of length 3. An array of byte strings is an ordered, indexed set starting with index 0 in which all byte strings have identical length.

2.2. Operators

When a and b are integers, mathematical operators are defined as follows:

^ : a ^ b denotes the result of a raised to the power of b.

* : a * b denotes the product of a and b. This operator is sometimes used implicitly in the absence of ambiguity, as in usual mathematical notation.

/ : a / b denotes the quotient of a by b.

% : a % b denotes the non-negative remainder of the integer division of a by b.

+ : a + b denotes the sum of a and b.

- : a - b denotes the difference of a and b.

The standard order of operations is used when evaluating arithmetic expressions.

Arrays are used in the common way, where the ith element of an array A is denoted A[i]. Byte strings are treated as arrays of bytes where necessary: If X is a byte string, then X[i] denotes its ith byte, where X[0] is the leftmost byte. In addition, bytes(X, i, j) with i < j denotes the range of bytes from the ith to the jth byte in X, inclusively. For example, if X = 0x01020304, then X[0] is 0x01 and bytes(X, 1, 2) is 0x0203.

If A and B are byte strings of equal length, then:

A AND B denotes the bitwise logical conjunction operation.

A XOR B denotes the bitwise logical exclusive disjunction operation.

When B is a byte and i is an integer, then B >> i denotes the logical right-shift operation. Similarly, B << i denotes the logical left-shift operation.

If X is a x-byte string and Y a y-byte string, then X || Y denotes the concatenation of X and Y, with X || Y = X[0]...X[x-1]Y[0]...Y[y-1].

2.3. Functions

If x is a non-negative real number, then we define the following functions:

ceil(x) : returns the smallest integer greater or equal than x.

floor(x) : returns the largest integer less or equal than x.

lg(x) : returns the base-2 logarithm of x.

If x, y, and z are real numbers, then we define the functions max(x, y) and max(x, y, z) which return the maximum value of the set {x, y} and {x, y, z}, respectively.

2.4. Strings of Base-w Numbers

A byte string can be considered as a string of base-w numbers, i.e. integers in the set {0, ... , w - 1}. The correspondence is defined by the function base_w(X, w) as follows. If X is a m-byte string, w is a member of the set {4, 8, 16}, then base_w(X, w) outputs a length ceil(8m/lg(w)) array of integers between 0 and w - 1. In case lg(w)

does not divide 8 * m without a remainder, X is virtually padded with a sufficient amount of zero bits.

Algorithm 1: base_w(X, w)

```
i_byte = 0;
i_bit = 0;
for ( i=0; i < ceil(8m/lg(w)); i++ ){
  if( i_bit + lg(w) <= 8 ){
    basew[i] = ((X[i_byte] << i_bit) >> (8-lg(w))) AND (w-1);
    i_bit += lg(w);
    if ( i_bit == 8 ){
      i_bit = 0;
      i_byte = i_byte + 1;
    }
  } else {
    basew[i] = ((X[i_byte] << i_bit) >> (8-lg(w))) AND (w-1);
    i_byte = i_byte + 1;
    if ( i_byte < m ){
      basew[i] += (X[i_byte] >> (8-(i_bit + lg(w)-8))) AND (w-1);
      i_bit = i_bit + lg(w)-8;
    }
  }
}
return basew;
```

For example, if X is 0x1234, then base_w(X, 8) returns the array {0, 4, 4, 3, 2, 0}.

```
                  X (represented as bits)
      +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
      | 0| 0| 0| 1| 0| 0| 1| 0| 0| 0| 1| 1| 0| 1| 0| 0|
      +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

                  X (padded with zeros)
      +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
      | 0| 0| 0| 1| 0| 0| 1| 0| 0| 0| 1| 1| 0| 1| 0| 0| 0| 0|
      +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

             X (represented as base-w numbers)
      +--------+--------+--------+--------+--------+--------+
      |    0   |    4   |    4   |    3   |    2   |    0   |
      +--------+--------+--------+--------+--------+--------+
```

2.5. Member Functions

To simplify algorithm descriptions, we assume the existence of member functions. If a complex data structure like a public key PK contains

a value X then getX(PK) returns the value of X for this public key. Accordingly, setX(PK, X, Y) sets value X in PK to the value hold by Y.

3. Primitives

3.1. WOTS+ One-Time Signatures

This section describes the WOTS+ one-time signature system, as defined in [Huelsing13]. WOTS+ is a one-time signature scheme; while a private key can be used to sign any message, each private key MUST be used only once to sign a single message. In particular, if a secret key is used to sign two different messages, the scheme becomes insecure.

The section starts with an explanation of parameters. Afterwards, the so-called chaining function, which forms the main building block of the WOTS+ scheme, is explained. It follows a description of the algorithms for key generation, signing and verification. Finally, pseudorandom key generation is discussed.

3.1.1. WOTS+ Parameters

WOTS+ uses the parameters m, n, and w; they all take positive integer values. These parameters are summarized as follows:

m : the message length in bytes

n : the length, in bytes, of a secret key, public key, or signature element

w : the Winternitz parameter; it is a member of the set {4, 8, 16}

The parameters are used to compute values l, l_1 and l_2:

l : the number of n-byte string elements in a WOTS+ secret key, public key, and signature. It is computed as l = l_1 + l_2, with l_1 = ceil(8m/lg(w)) and l_2 = floor(lg(l_1*(w-1))/lg(w)) + 1

The value of n is determined by the cryptographic hash function used for WOTS+. The hash function is chosen to ensure an appropriate level of security. The value of m is often the length of a message digest. The parameter w can be chosen from the set {4,8,16}. A larger value of w results in shorter signatures but slower overall signing operations; it has little effect on security. Choices of w are limited to the values 4, 8 and 16 since these values yield optimal trade-offs.

3.1.1.1. WOTS+ Hashing Functions

The WOTS+ algorithm uses a cryptographic hash function F. F accepts and returns byte strings of length n. Security requirements on F are discussed in Section 8.

3.1.2. WOTS+ Chaining Function

The chaining function (Algorithm 2) computes an iteration of F on an n-byte input using a vector of n-byte strings called bitmasks. In each iteration, a bitmask is first XORed to an intermediate result before it is processed by F. In the following, bm is an array of at least w-2 n-byte strings (that contains the bitmasks). The chaining function takes as input an n-byte string X, a start index i, a number of steps s, and the bitmasks bm. The chaining function returns as output the value obtained by iterating F for s times on input X, using the bitmasks from bm starting at index i.

Algorithm 2: Chaining Function

```
if s is equal to 0 then
  return X;
end
if (i+s) > w-1 then
  return NULL;
end
byte[n] tmp = chain(X, i, s-1, bm);
tmp = F(tmp XOR bm[i+s-1]);
return tmp;
```

3.1.3. WOTS+ Private Key

The private key in WOTS+, denoted by sk, is a length l array of n-byte strings. This private key MUST be only used to sign exactly one message. Each n-byte string MUST either be selected randomly from the uniform distribution or using a cryptographically secure pseudorandom procedure. In the latter case, the security of the used procedure MUST at least match that of the WOTS+ parameters used. For a further discussion on pseudorandom key generation see the end of this section. The following pseudocode (Algorithm 3) describes an algorithm for generating sk.

Algorithm 3: Generating a WOTS+ Private Key

```
for ( i = 0; i < l; i = i + 1 ) {
  set sk[i] to a uniformly random n-byte string
}
return sk
```

3.1.4. WOTS+ Public Key

A WOTS+ key pair defines a virtual structure that consists of l hash chains of length w. The l n-byte strings in the secret key each define the start node for one hash chain. The public key consists of the end nodes of these hash chains. Therefore, like the secret key, the public key is also a length l array of n-byte strings. To compute the hash chain, the chaining function (Algorithm 2) is used. The bitmasks have to be provided by the calling algorithm. The same bitmasks are used for all chains. The following pseudocode (Algorithm 4) describes an algorithm for generating the public key pk, where sk is the private key.

Algorithm 4 (WOTS_genPK): Generating a WOTS+ Public Key From a Private Key

```
for ( i = 0; i < l; i = i + 1 ) {
  pk[i] = chain(sk[i], 0, w-1, bm);
}
return pk;
```

3.1.5. WOTS+ Signature Generation

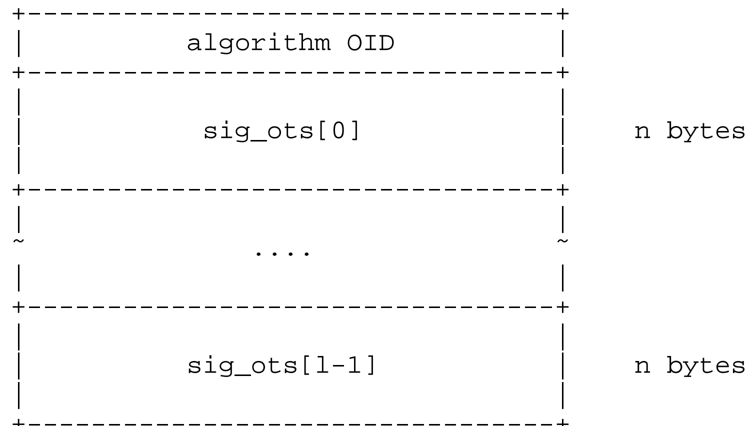
A WOTS+ signature is a length l array of n-byte strings. The WOTS+ signature is generated by mapping a message to l integers between 0 and w - 1. To this end, the message is transformed into base w numbers using the base_w function defined in Section 2.4. Next, a checksum is computed and appended to the transformed message as base w numbers using base_w(). Each of the base w integers is used to select a node from a different hash chain. The signature is formed by concatenating the selected nodes. The pseudocode for signature generation is shown below (Algorithm 5), where M is the message and sig is the resulting signature.

Algorithm 5 (WOTS_sign): Generating a signature from a private key and a message

```
csum = 0;
// convert message to base w
msg = base_w(M,w)
// compute checksum
for ( i = 0; i < l_1; i = i + 1 ) {
  csum = csum + w - 1 - msg[i]
}
// Convert csum to base w
msg = msg || base_w(csum, w);
for ( i = 0; i < l; i = i + 1 ) {
  sig[i] = chain(sk[i], 0, msg[i], bm)
}
return sig
```

The data format for a signature is given below.

WOTS+ Signature



```
+---------------------------------+
|          algorithm OID          |
+---------------------------------+
|                                 |
|           sig_ots[0]            |    n bytes
|                                 |
+---------------------------------+
|                                 |
~              ....               ~
|                                 |
+---------------------------------+
|                                 |
|          sig_ots[l-1]           |    n bytes
|                                 |
+---------------------------------+
```

3.1.6. WOTS+ Signature Verification

In order to verify a signature sig on a message M, the verifier computes a WOTS+ public key value from the signature. This can be done by "completing" the chain computations starting from the signature values, using the base-w values of the message hash and its checksum. This step, called WOTS_pkFromSig, is described below in Algorithm 6. The result of WOTS_pkFromSig is then compared to the given public key. If the values are equal, the signature is accepted. Otherwise, the signature is rejected.

Algorithm 6 (WOTS_pkFromSig): Computing a WOTS+ public key from a message and its signature

```
csum = 0;
// convert message to base w
msg = base_w(M,w)
// compute checksum
for ( i = 0; i < l_1; i = i + 1 ) {
  csum = csum + w - 1 - msg[i]
}
// Convert csum to base w
msg = msg || base_w(csum, w);
for ( i = 0; i < l; i = i + 1 ) {
  tmp_pk[i] = chain(sig[i], msg[i], w-1-msg[i], bm)
}
return tmp_pk
```

Note: XMSS uses WOTS_pkFromSig to compute a public key value and delays the comparison to a later point.

3.1.7. Pseudorandom Key Generation

An implementation MAY use a cryptographically secure pseudorandom method to generate the secret key from a single n-byte value. For example, the method suggested in [BDH11] and explained below MAY be used. Other methods MAY be used. The choice of a pseudorandom method does not affect interoperability, but the cryptographic strength MUST match that of the used WOTS+ parameters.

The advantage of generating the secret key elements from a random n-byte string is that only this n-byte string needs to be stored instead of the full secret key. The key can be regenerated when needed. The suggested method from [BDH11] uses a pseudorandom function G(K,M) that takes an n-byte key and an n-byte message. During key generation a uniformly random n-byte string S is sampled from a secure source of randomness. The secret key elements are computed as sk[i] = G(S,i) whenever needed. The second parameter of G is i, represented as n-byte string in the common way. To implement G, an implementation MAY use the hash function F in PRF mode. When WOTS+ is used within XMSS or XMSS^MT, an implementation SHOULD use PRF_m, taking the first n bytes from the output.

4. Schemes

In this section, the extended Merkle signature scheme (XMSS) is described using WOTS+. XMSS comes in two flavours: First, a single-tree variant (XMSS) and second a multi-tree variant (XMSS^MT). Both allow combining a large number of WOTS+ key pairs under a single small public key. The main ingredient added is a binary hash tree construction. XMSS uses a single hash tree while XMSS^MT uses a tree of XMSS key pairs.

4.1. XMSS: eXtended Merkle Signature Scheme

XMSS is a method for signing a potentially large but fixed number of messages. It is based on the Merkle signature scheme. XMSS uses four cryptographic components: WOTS+ as OTS method, two additional cryptographic hash functions H and H_m, and a pseudorandom function PRF_m. One of the main advantages of XMSS with WOTS+ is that it does not rely on the collision resistance of the used hash functions but on weaker properties. Each XMSS public/private key pair is associated with a perfect binary tree, every node of which contains an n-byte value. Each tree leaf contains a special tree hash of a WOTS+ public key value. Each non-leaf tree node is computed by first concatenating the values of its child nodes, computing the XOR with a bitmask, and applying the hash function H to the result. The value corresponding to the root of the XMSS tree forms the XMSS public key together with the bitmasks.

To generate a key pair that can be used to sign 2^h messages, a tree of height h is used. XMSS is a stateful signature scheme, meaning that the secret key changes after every signature. To prevent one-time secret keys from being used twice, the WOTS+ key pairs are numbered from 0 to (2^h)-1 according to the related leaf, starting from index 0 for the leftmost leaf. The secret key contains an index that is updated after every signature, such that it contains the index of the next unused WOTS+ key pair.

A signature consists of the index of the used WOTS+ key pair, the WOTS+ signature on the message and the so-called authentication path. The latter is a vector of tree nodes that allow a verifier to compute a value for the root of the tree. A verifier computes the root value and compares it to the respective value in the XMSS public key. If they match, the signature is valid. The XMSS secret key consists of all WOTS+ secret keys and the actual index. To reduce storage, a pseudorandom key generation procedure, as described in [BDH11], MAY be used. The security of the used method MUST at least match the security of the XMSS instance.

4.1.1. XMSS Parameters

XMSS has the following parameters:

h : the height (number of levels - 1) of the tree

n : the length in bytes of each node

m : the length of the message digest

w : the Winternitz parameter as defined for WOTS+ in Section 3.1

There are N = 2^h leaves in the tree. XMSS uses num_bm = max{2 * (h + ceil(lg(l))), w - 2} bitmasks produced during key generation.

For XMSS and XMSS^MT, secret and public keys are denoted by SK and PK. For WOTS+, secret and public keys are denoted by sk and pk, respectively. XMSS and XMSS^MT signatures are denoted by Sig. WOTS+ signatures are denoted by sig.

4.1.2. XMSS Hash Functions

Besides the cryptographic hash function F required by WOTS+, XMSS uses three more functions:

A cryptographic hash function H. H accepts byte strings of length (2 * n) and returns an n-byte string.

A cryptographic hash function H_m. H_m accepts byte strings of arbitrary length and returns an m-byte string.

A pseudorandom function PRF_m. PRF_m accepts byte strings of arbitrary length and an m-byte key and returns an m-byte string.

4.1.3. XMSS Private Key

An XMSS private key contains N = 2^h WOTS+ private keys, the leaf index idx of the next WOTS+ private key that has not yet been used and SK_PRF, an m-byte key for the PRF. The leaf index idx is initialized to zero when the XMSS private key is created. The PRF key SK_PRF MUST be sampled from a secure source of randomness that follows the uniform distribution. The WOTS+ secret keys MUST be generated as described in Section 3.1. To reduce the secret key size, a cryptographic pseudorandom method MAY be used as discussed at the end of this section. For the following algorithm descriptions, the existence of a method getWOTS_SK(SK,i) is assumed. This method takes as inputs an XMSS secret key SK and an integer i and outputs the ith WOTS+ secret key of SK.

4.1.4. L-Trees

To compute the leaves of the binary hash tree, a so-called L-tree is used. An L-tree is an unbalanced binary hash tree, distinct but similar to the main XMSS binary hash tree. The algorithm ltree (Algorithm 7) takes as input a WOTS+ public key pk and compresses it to a single n-byte value pk[0]. The algorithm uses the first (2 * ceil(log(l))) of the num_bm n-byte bitmasks bm.

Algorithm 7: ltree

```
unsigned int l' = l
unsigned int j = 0
while ( l' > 1 ) {
  for ( i = 0; i < floor(l' / 2); i = i + 1 ) {
    pk[i] = H((pk[2i] XOR bm[j]) || (pk[2i + 1] XOR bm[j + 1]))
  }
  if ( l' is equal to 1 % 2 ) {
    pk[floor(l' / 2) + 1] = pk[l']
  }
  l' = ceil(l' / 2)
  j = j + 2
}
return pk[0]
```

4.1.5. TreeHash

For the computation of the internal n-byte nodes of a Merkle tree, the subroutine treeHash (Algorithm 8) accepts an XMSS secret key SK, an unsigned integer s (the start index), an unsigned integer h (the target node height) and the bitmasks bm. The treeHash algorithm returns the root node of a tree of height h with the leftmost leaf being the hash of the WOTS+ pk with index s. The treeHash algorithm uses a stack holding up to (h-1) n-byte strings, with the usual stack functions push() and pop().

Algorithm 8: treeHash

```
for ( i = 0; i < 2^h; i = i + 1 ) {
  pk = WOTS_genPK (getWOTS_SK(SK, s+i), bm)
  node = ltree(pk, bm)
  while ( Top node on Stack has same height h' as node ) {
    node = H((Stack.pop() XOR bm[2l + 2h']) ||
             (node XOR bm[2l + 2h' + 1]))
  }
  Stack.push(node)
}
return Stack.pop()
```

4.1.6. XMSS Public Key

The XMSS public key is computed as described in XMSS_genPK (Algorithm 9). The algorithm takes the num_bm n-byte bitmasks bm, the XMSS secret key SK, and the tree height h. The XMSS public key PK consists of the root of the binary hash tree and the bitmasks bm.

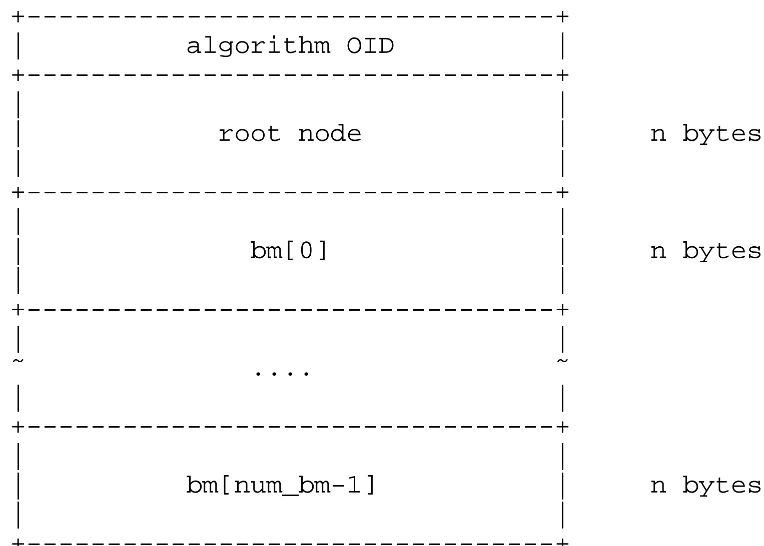
Algorithm 9: XMSS_genPK - Generate an XMSS public key from an XMSS private key

```
for ( i = 0; i < num_bm; i = i + 1 ) {
   set bm[i] to a uniformly random n-byte string
}
root = treeHash(SK, 0, h, bm)
PK = root || bm
return PK
```

Public and private key generation MAY be interleaved to save space. Especially, when a pseudorandom method is used to generate the secret key, generation MAY be done when the respective WOTS+ key pair is needed by treeHash.

The format of an XMSS public key is given below.

XMSS Public Key



```
+---------------------------------+
|          algorithm OID          |
+---------------------------------+
|                                 |
|            root node            |     n bytes
|                                 |
+---------------------------------+
|                                 |
|              bm[0]              |     n bytes
|                                 |
+---------------------------------+
|                                 |
~              ....               ~
|                                 |
+---------------------------------+
|                                 |
|          bm[num_bm-1]           |     n bytes
|                                 |
+---------------------------------+
```

4.1.7. XMSS Signature

An XMSS signature is a (4 + m + (l + h) * n)-byte string consisting of

the index idx_sig of the used WOTS+ key pair (4 bytes),

a byte string r used for randomized hashing (m bytes),

a WOTS+ signature sig_ots (l * n bytes),

the so called authentication path 'auth' for the leaf associated with the used WOTS+ key pair (h * n bytes).

The authentication path is an array of h n-byte strings. It contains the siblings of the nodes on the path from the used leaf to the root. It does not contain the nodes on the path itself. These nodes are needed by a verifier to compute a root node for the tree from the WOTS+ public key. A node Node is addressed by its position in the tree. Node(x,y) denotes the xth node on level y with x = 0 being the leftmost node on a level. The leaves are on level 0, the root is on level h. An authentication path contains exactly one node on every layer 0 <= x <= h-1. For the ith WOTS+ key pair, counting from zero, the jth authentication path node is

Node(j, floor(i / (2^j)) + 1) if floor(i / (2^j)) is even or

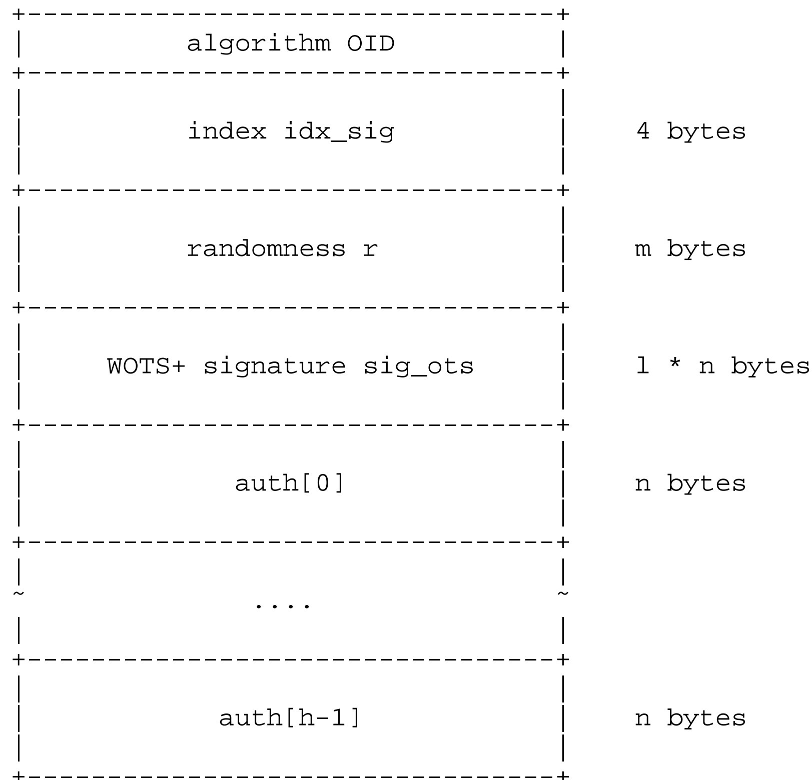
Node(j, floor(i / (2^j)) - 1) if floor(i / (2^j)) is odd.

Given an XMSS secret key SK and bitmasks bm, all nodes in a tree are determined. Their value is defined in terms of treeHash (Algorithm 8):

Node(x,y) = treeHash(SK, x * 2^y, y, bm).

The data format for a signature is given below.

XMSS Signature



4.1.8. XMSS Signature Generation

To compute the XMSS signature of a message M with an XMSS private key, the signer first computes a randomized message digest. Then a WOTS+ signature of the message is computed using the next unused WOTS+ private key. Next, the authentication path is computed. Finally, the secret key is updated, i.e. idx is incremented. An implementation MUST NOT output the signature before the updated private key.

The node values of the authentication path MAY be computed in any way. This computation is assumed to be performed by the subroutine buildAuth for the function XMSS_sign, as below. The fastest alternative is to store all tree nodes and set the array in the signature by copying them, respectively. The least storage-intensive alternative is to recompute all nodes for each signature online. There exist several algorithms in between, with different time/ storage trade-offs. For an overview see [BDS09]. Note that the details of this procedure are not relevant to interoperability; it is not necessary to know any of these details in order to perform the

signature verification operation. As a consequence, buildAuth is not specified here.

The algorithm XMSS_sign (Algorithm 10) described below calculates an updated secret key SK and a signature on a message M. XMSS_sign takes as inputs a message M of an arbitrary length, an XMSS secret key SK and bitmasks bm. It returns the byte string containing the concatenation of the updated secret key SK and the signature Sig.

Algorithm 10: XMSS_sign - Generate an XMSS signature and update the XMSS secret key

```
idx_sig = getIdx(SK)
auth = buildAuth(SK, bm, idx_sig)
byte[m] r = PRF_m(getSK_PRF(SK), M)
byte[m] M' = H_m(r || M)
sig_ots = WOTS_sign(getWOTS_SK(SK, idx_sig), M', bm)
Sig = (idx_sig || r || sig_ots || auth)
setIdx(SK, idx_sig + 1)
return (SK || Sig)
```

4.1.9. XMSS Signature Verification

An XMSS signature is verified by first computing the message digest using randomness r and a message M. Then the used WOTS+ public key pk_ots is computed from the WOTS+ signature using WOTS_pkFromSig. The WOTS+ public key in turn is used to compute the corresponding leaf using an L-tree. The leaf, together with index idx_sig, authentication path auth and bitmasks bm is used to compute an alternative root value for the tree. These first steps are done by XMSS_rootFromSig (Algorithm 11). The verification succeeds if and only if the computed root value matches the one in the XMSS public key. In any other case it MUST return fail.

The main part of XMSS signature verification is done by the function XMSS_rootFromSig (Algorithm 11) described below. XMSS_rootFromSig takes as inputs an XMSS signature Sig, a message M, and the bitmasks bm. XMSS_rootFromSig returns an n-byte string holding the value of the root of a tree defined by the input data.

Algorithm 11: XMSS_rootFromSig - Compute a root node using an XMSS signature, a message, and bitmasks bm

```
byte[m] M' = H_m(r || M)
pk_ots = WOTS_pkFromSig(sig_ots, M', bm)
byte[n][2] node
node[0] = ltree(pk_ots, bm)
for ( k = 1; k < h; k = k + 1 ) {
  if ( floor(i / (2^k)) % 2 is equal to 0 ) {
    node[1] = H((node[0] XOR bm[2l + 2k]) ||
                (auth[k - 1] XOR bm[2l + 2k + 1]))
  } else {
    node[1] = H((auth[k - 1] XOR bm[2l + 2k]) ||
                (node[0] XOR bm[2l + 2k + 1]))
  }
  node[0] = node[1]
}
return node[0]
```

The full XMSS signature verification is depicted below for completeness. XMSS^MT uses only XMSS_rootFromSig and delegates the comparison to a later comparison of data depending on its output.

Algorithm 12: XMSS_verify - Verify an XMSS signature using an XMSS signature, the corresponding XMSS public key and a message

```
byte[n] node = XMSS_rootFromSig(Sig, M, getBM(PK))
if ( node is equal to root in PK ) {
  return true
} else {
  return false
}
```

4.1.10. Pseudorandom Key Generation

An implementation MAY use a cryptographically secure pseudorandom method to generate the XMSS secret key from a single n-byte value. For example, the method suggested in [BDH11] and explained below MAY be used. Other methods MAY be used. The choice of a pseudorandom method does not affect interoperability, but the cryptographic strength MUST match that of the used XMSS parameters.

For XMSS a similar method than the one used for WOTS+ can be used. The suggested method from [BDH11] uses a pseudorandom function G(K,M) that takes an n-byte key and an n-byte message. During key generation a uniformly random n-byte string S is sampled from a secure source of randomness. This seed S is used to generate an n-byte value S_ots for each WOTS+ key pair. This n-byte value can then be used to compute the respective WOTS+ secret key using the method described in Section 3.1.7. The seeds for the WOTS+ key pairs are computed as S_ots[i] = G(S,i). The second parameter of G is the

index i of the WOTS+ key pair, represented as n-byte string in the common way. To implement G an implementation SHOULD use PRF_m, taking the first n bytes from the output. An advantage of this method is that a WOTS+ key can be computed using only l+1 evaluations of G when S is given.

4.1.11. Free Index Handling and Partial Secret Keys

Some applications might require to work with partial secret keys or copies of secret keys. Examples include delegation of signing rights / proxy signatures, and load balancing. Such applications MAY use their own key format and MAY use a signing algorithm different from the one described above. The index in partial secret keys or copies of a secret key MAY be manipulated as required by the applications. However, applications MUST establish means that guarantee that each index and thereby each WOTS+ key pair is used to sign only a single message.

4.2. XMSS^MT: Multi-Tree XMSS

XMSS^MT is a method for signing a large but fixed number of messages. It was first described in [HRB13]. It builds on XMSS. XMSS^MT uses a tree of several layers of XMSS trees. The trees on top and intermediate layers are used to sign the root nodes of the trees on the respective layer below. Trees on the lowest layer are used to sign the actual messages. All XMSS trees have equal height.

Consider an XMSS^MT tree of total height h that has d layers of XMSS trees of height h / d. Then layer d - 1 contains one XMSS tree, layer d - 2 contains 2^(h / d) XMSS trees, and so on. Finally, layer 0 contains 2^(h - h / d) XMSS trees.

4.2.1. XMSS^MT Parameters

In addition to all XMSS parameters, an XMSS^MT system requires the number of tree layers d, specified as an integer value that divides h without remainder. The same tree height h / d and the same Winternitz parameter w are used for all tree layers.

All the trees on higher layers sign root nodes of other trees which are n-byte strings. Hence, no message compression is needed and WOTS+ is used to sign the root nodes themselves instead of their hash values. Hence the WOTS+ message length for these layers is n not m. Accordingly, the values of l_1, l_2 and l change for these layers. The parameters l_1_n, l_2_n, and l_n denote the respective values computed using n as message length for WOTS+.

4.2.2. XMSS Algorithms Without Message Hash

As all XMSS trees besides those on layer 0 are used to sign short fixed length messages, the initial message hash can be omitted. In the description below XMSS_sign_wo_hash and XMSS_rootFromSig_wo_hash are versions of XMSS_sign and XMSS_rootFromSig, respectively, that omit the initial message hash. They are obtained by setting M' = M in the above algorithms. Accordingly, the evaluations of H_m and PRF_m SHOULD be omitted. This also means that no randomization element r for the message hash is required. XMSS signatures generated by XMSS_sign_wo_hash and verified by XMSS_rootFromSig_wo_hash MUST NOT contain a value r.

4.2.3. XMSS^MT Private Key

An XMSS^MT private key SK_MT consists of one reduced XMSS private key for each XMSS tree. These reduced XMSS private keys contain no pseudorandom function key and no index. Instead, SK_MT contains a single m-byte pseudorandom function key SK_PRF and a single (ceil(h / 8))-byte index idx_MT. The index is a global index over all WOTS+ key pairs of all XMSS trees on layer 0. It is initialized with 0. It stores the index of the last used WOTS+ key pair on the bottom layer, i.e. a number between 0 and 2^h - 1.

The algorithm descriptions below uses a function getXMSS_SK(SK, x, y) that outputs the reduced secret key of the xth XMSS tree on the yth layer.

4.2.4. XMSS^MT Public Key

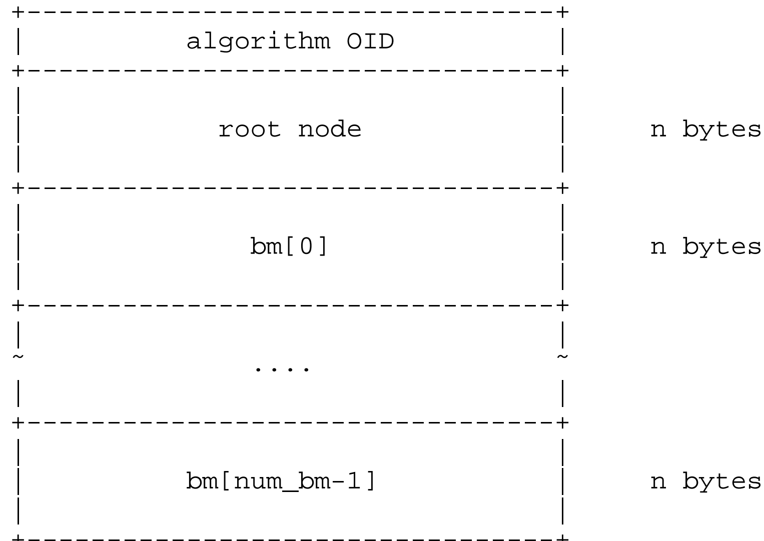
The XMSS^MT public key PK_MT contains the root of the single XMSS tree on layer d-1 and the bitmasks. The same bitmasks are used for all XMSS tress. Algorithm 13 shows pseudocode to generate PK_MT. First, num_bm = max{ 2 * (h / d + ceil(lg(l))), 2 * (h / d + ceil(lg(l_n))), w - 2 } n-byte bitmasks bm are chosen uniformly at random. The n-byte root node of the top layer tree is computed using treeHash. The algorithm XMSSMT_genPK takes the XMSS^MT secret key SK_MT as an input and outputs an XMSS^MT public key PK_MT.

Algorithm 13: XMSSMT_genPK - Generate an XMSS^MT public key from an XMSS^MT private key

```
for ( i = 0; i < num_bm; i = i + 1 ) {
   set bm[i] to a uniformly random n-byte string
}
root = treeHash(getXMSS_SK(SK_MT, 0, d - 1), 0, h / d, bm)
PK_MT = root || bm
return PK_MT
```

The format of an XMSS^MT public key is given below.

XMSS^MT Public Key



4.2.5. XMSS^MT Signature

An XMSS^MT signature Sig_MT is a byte string of length (ceil(h / 8) + m + (h + l + (d - 1) * l_n) * n). It consists of

the index idx_sig of the used WOTS+ key pair on the bottom layer (ceil(h / 8) bytes),

a byte string r used for randomized hashing (m bytes),

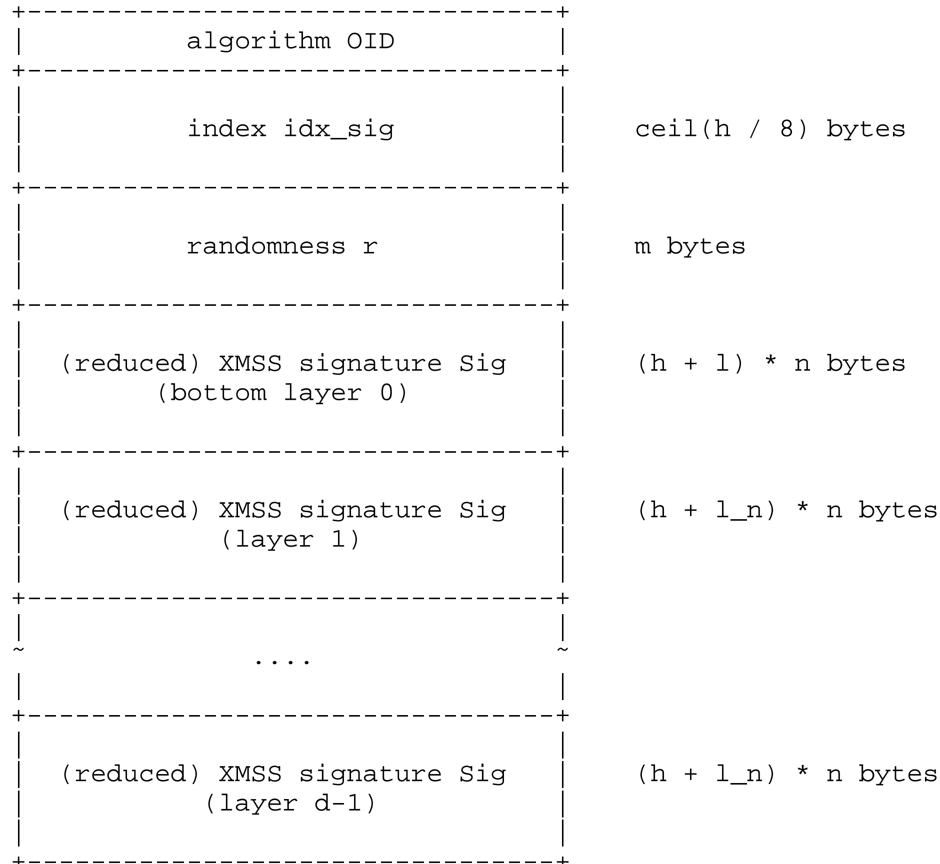
one reduced XMSS signature ((h + l) * n bytes),

d-1 reduced XMSS signatures with message length n ((h + l_n) * n bytes).

The reduced XMSS signatures contain no index idx and no byte string r. They only contain a WOTS+ signature sig_ots and an authentication path auth. The first reduced XMSS signature contains a WOTS+ signature that consists of l n-byte elements. The remaining reduced XMSS signatures contain a WOTS+ signature on an n-byte message and hence consist of l_n n-byte elements.

The data format for a signature is given below.

XMSS^MT signature



4.2.6. XMSS^MT Signature Generation

To compute the XMSS^MT signature Sig_MT of a message M using an XMSS^MT private key SK_MT and bitmasks bm, XMSSMT_sign (Algorithm 14) described below uses XMSS_sign and XMSS_sign_wo_hash as defined in Section 4.2.2. First, the signature index is set to idx. Next, PRF_m is used to compute a pseudorandom m-byte string r. This m-byte string is then used to compute a randomized message digest of length m. The message digest is signed using the WOTS+ key pair on the bottom layer with absolute index idx. The authentication path for the WOTS+ key pair is computed as well as the root of the containing XMSS tree. The root is signed by the parent XMSS tree. This is repeated until the top tree is reached.

Algorithm 14: XMSSMT_sign - Generate an XMSS^MT signature and update the XMSS^MT secret key

```
SK_PRF = getSK_PRF(SK_MT)
idx_sig = getIdx(SK_MT)
setIdx(SK_MT, idx_sig + 1)
Sig_MT = idx_sig
unsigned int idx_tree = (h - h / d) most significant bits of idx_sig
unsigned int idx_leaf = (h / d) least significant bits of idx_sig
SK = idx_leaf || SK_PRF || getXMSS_SK(SK_MT, idx_tree, 0)
Sig_tmp = XMSS_sign(M, SK, bm)
Sig_tmp = Sig_tmp without idx
Sig_MT = Sig_MT || Sig_tmp
for ( j = 1; j < d; j = j + 1 ) {
   root = treeHash(SK, 0, h / d, bm)
   idx_leaf = (h / d) least significant bits of idx_tree
   idx_tree = (h - j * (h / d)) most significant bytes of idx_tree
   SK = idx_leaf || SK_PRF || getXMSS_SK(SK_MT, idx_tree, j)
   Sig_tmp = XMSS_sign_wo_hash(root, SK, bm) with idx removed
   Sig_MT = Sig_MT || Sig_tmp
}
return SK_MT || Sig_MT
```

Algorithm 14 is only one method to compute XMSS^MT signatures. Especially, there exist time-memory trade-offs that allow to reduce the signing time to less than the signing time of an XMSS scheme with tree height h / d. These trade-offs prevent certain values from being recomputed several times by keeping a state and distribute all computations over all signature generations. Details can be found in [Huelsing13a].

4.2.7. XMSS^MT Signature Verification

XMSS^MT signature verification (Algorithm 15) can be summarized as d XMSS signature verifications with small changes. First, only the message is hashed. The remaining XMSS signatures are on the root nodes of trees which have a fixed length. Second, instead of comparing the computed root node to a given value, a signature on the root is verified. Only the root node of the top tree is compared to the value in the XMSS^MT public key. XMSSMT_verify uses XMSS_rootFromSig and XMSS_rootFromSig_wo_hash. XMSSMT_verify takes as inputs an XMSS^MT signature Sig^MT, a message M and a public key PK_MT. It outputs a boolean.

Algorithm 15: XMSSMT_verify - Verify an XMSS^MT signature Sig_MT on a message M using an XMSS^MT public key PK_MT

```
idx = getIdx(Sig_MT)
unsigned int idx_leaf = (h / d) least significant bits of idx
unsigned int idx_tree = (h - h / d) most significant bits of idx
Sig' = leaf || setR(Sig_MT) || getXMSSSignature(Sig, 0)
byte[n] node = XMSS_rootFromSig(Sig', M, getBm(PK_MT))
for ( j = 1; j < d; j = j + 1 ) {
   idx_leaf = (h / d) least significant bytes of idx_tree
   idx_tree = (h - j * h / d) most significant bytes of idx_tree
   Sig' = idx_leaf || getXMSSSignature(Sig, j)
   node = XMSS_rootFromSig_wo_hash(Sig', node, getBm(PK_MT))
}
if ( node is equal to getRoot(PK_MT) ) {
  return true
} else {
  return false
}
```

4.2.8. Pseudorandom Key Generation

Like for XMSS, an implementation MAY use a cryptographically secure pseudorandom method to generate the XMSS^MT secret key from a single n-byte value. For example, the method explained below MAY be used. Other methods MAY be used. The choice of a pseudorandom method does not affect interoperability, but the cryptographic strength MUST match that of the used XMSS parameters.

For XMSS^MT a method similar to that for XMSS and WOTS+ can be used. The method uses a pseudorandom function G(K,M) that takes an n-byte key and an n-byte message. During key generation a uniformly random n-byte string S_MT is sampled from a secure source of randomness. This seed S_MT is used to generate one n-byte value S for each XMSS key pair. This n-byte value can be used to compute the respective XMSS secret key using the method described in Section 4.1.10. Let S[x][y] be the seed for the xth XMSS secret key on layer y. The seeds are computed as S[x][y] = G(G(S, y), x). The second parameter of G is the index x (resp. level y), represented as n-byte string in the common way. To implement G an implementation SHOULD use PRF_m, taking the first n bytes from the output.

4.2.9. Free Index Handling and Partial Secret Keys

The content of Section 4.1.11 also applies to XMSS^MT.

5. Parameter Sets

This note provides a first basic set of parameter sets which are assumed to cover most relevant applicants. Parameter sets for three classical security levels are defined: 128, 256 and 512 bits. Function output sizes are n = 16, 32 and 64 bytes and m = 32, 64, respectively. While m = n is used for n = 32 and n = 64, m = 32 is used for the n = 16 case. Considering quantum-computer-aided attacks, these output sizes yield post-quantum security of 64, 128 and 256 bits, respectively. The n = 16 parameter sets are included to encourage adoption in the pre-quantum era as they lead to smaller signatures and faster runtimes than other parameter sets. The n = 64 parameter sets are provided to support post-quantum scenarios.

For the n = 16 setting, this note only defines parameter sets with AES-based hash functions. The reason is that they benefit from hardware acceleration on many modern platforms. Let AES(K,M) denote evaluation of AES-128 with 128 bit key K and 128 bit message M. Define the 16-byte string IV = 0x0001020304050607080910111213141516. Then F and H are implemented as

```
F(X) = AES(IV,X) XOR X

H(X) = AES( AES(IV, X1) XOR X1, X2) XOR X2
```

where X = X1 || X2, i.e. X1 denotes the most significant 16 bytes of X and X2 the least significant 16 bytes. For these parameter sets H_m is implemented as SHA3-256 and PRF_m as SHA3-256 in PRF/MAC mode.

For the n = m = 32 and n = m = 64 settings, all functions are implemented using SHA3-256 and SHA3-512, respectively.

5.1. Zero Bitmasks

For applications that require a very small public key this note additionally defines zero bitmasks parameter sets. For these parameter sets the bitmasks are set to an all-zero string. The XMSS and XMSS^MT public keys for these parameter sets contain no bitmasks. Instead, they only contain the single n-byte value holding the root node. When handling zero bitmasks parameter sets, implementations MAY internally use an all-zero string as bitmasks and stick to the same algorithms as for the other parameter sets. Implementations MAY omit the XOR with an all-zero bitmask. Zero bitmasks parameter sets are only defined for n = 32 and n = 64, as formal security reductions require the used hash functions to be collision-resistant in this case. Hence, the estimated classical security levels are 128 and 256 bits for n = 32 and n = 64 with zero bitmasks, respectively. The corresponding post-quantum security levels are approximately 85 and 170 bits, respectively.

5.2. WOTS+ Parameters

To fully describe a WOTS+ signature method, the parameters m, n, and w, as well as the function F MUST be specified. This section defines several WOTS+ signature systems, each of which is identified by a name. Values for l are provided for convenience.

Name	F	m	n	w	l
WOTSP_AES128_M32_W4	AES128	32	16	4	133
WOTSP_AES128_M32_W8	AES128	32	16	8	90
WOTSP_AES128_M32_W16	AES128	32	16	16	67
WOTSP_SHA3-256_M32_W4	SHA3	32	32	4	133
WOTSP_SHA3-256_M32_W8	SHA3	32	32	8	90
WOTSP_SHA3-256_M32_W16	SHA3	32	32	16	67
WOTSP_SHA3-512_M64_W4	SHA3	64	64	4	261
WOTSP_SHA3-512_M64_W8	SHA3	64	64	8	175
WOTSP_SHA3-512_M64_W16	SHA3	64	64	16	131

Table 1

Here SHA3 denotes the NIST standard hash function, also known as Keccak [DRAFTFIPS202]. XDR formats for WOTS+ are listed in Appendix A.

5.3. XMSS Parameters

To fully describe an XMSS signature method, the parameters m, n, w, and h, as well as the functions F, H, H_m and PRF_m MUST be specified. This section defines several XMSS signature systems, each of which is identified by a name.

The XDR formats for XMSS are listed in Appendix B.

5.3.1. XMSS Parameters

We first define XMSS signature methods as described in Section 4.1. We define parameter sets that implement the functions using AES and SHA3 as described above as well as pure SHA3 parameter sets.

5.3.1.1. XMSS Parameters with AES and SHA3

The following XMSS signature methods implement the functions F, H, H_m and PRF_m using AES and SHA3 as described above.

Name	m	n	w	l	h
XMSS_AES128_M32_W4_H10	32	16	4	133	10
XMSS_AES128_M32_W4_H16	32	16	4	133	16
XMSS_AES128_M32_W4_H20	32	16	4	133	20
XMSS_AES128_M32_W8_H10	32	16	8	90	10
XMSS_AES128_M32_W8_H16	32	16	8	90	16
XMSS_AES128_M32_W8_H20	32	16	8	90	20
XMSS_AES128_M32_W16_H10	32	16	16	67	10
XMSS_AES128_M32_W16_H16	32	16	16	67	16
XMSS_AES128_M32_W16_H20	32	16	16	67	20

Table 2

5.3.1.2. XMSS Parameters with SHA3

The following XMSS signature methods implement the functions F, H, H_m and PRF_m solely using SHA3 as described above.

Name	m	n	w	l	h
XMSS_SHA3-256_M32_W4_H10	32	32	4	133	10
XMSS_SHA3-256_M32_W4_H16	32	32	4	133	16
XMSS_SHA3-256_M32_W4_H20	32	32	4	133	20
XMSS_SHA3-256_M32_W8_H10	32	32	8	90	10
XMSS_SHA3-256_M32_W8_H16	32	32	8	90	16
XMSS_SHA3-256_M32_W8_H20	32	32	8	90	20
XMSS_SHA3-256_M32_W16_H10	32	32	16	67	10
XMSS_SHA3-256_M32_W16_H16	32	32	16	67	16
XMSS_SHA3-256_M32_W16_H20	32	32	16	67	20
XMSS_SHA3-512_M64_W4_H10	64	64	4	261	10
XMSS_SHA3-512_M64_W4_H16	64	64	4	261	16
XMSS_SHA3-512_M64_W4_H20	64	64	4	261	20
XMSS_SHA3-512_M64_W8_H10	64	64	8	175	10
XMSS_SHA3-512_M64_W8_H16	64	64	8	175	16
XMSS_SHA3-512_M64_W8_H20	64	64	8	175	20
XMSS_SHA3-512_M64_W16_H10	64	64	16	131	10
XMSS_SHA3-512_M64_W16_H16	64	64	16	131	16
XMSS_SHA3-512_M64_W16_H20	64	64	16	131	20

Table 3

5.3.2. XMSS Parameters With Empty Bitmasks

We now define XMSS signature methods for the zero bitmasks special case described in Section 5.1. For this setting all signature methods implement the functions F, H, H_m and PRF_m solely using SHA3 as described above.

Name	m	n	w	l	h
XMSS_SHA3-256_M32_W4_H10_z	32	32	4	133	10
XMSS_SHA3-256_M32_W4_H16_z	32	32	4	133	16
XMSS_SHA3-256_M32_W4_H20_z	32	32	4	133	20
XMSS_SHA3-256_M32_W8_H10_z	32	32	8	90	10
XMSS_SHA3-256_M32_W8_H16_z	32	32	8	90	16
XMSS_SHA3-256_M32_W8_H20_z	32	32	8	90	20
XMSS_SHA3-256_M32_W16_H10_z	32	32	16	67	10
XMSS_SHA3-256_M32_W16_H16_z	32	32	16	67	16
XMSS_SHA3-256_M32_W16_H20_z	32	32	16	67	20
XMSS_SHA3-512_M64_W4_H10_z	64	64	4	261	10
XMSS_SHA3-512_M64_W4_H16_z	64	64	4	261	16
XMSS_SHA3-512_M64_W4_H20_z	64	64	4	261	20
XMSS_SHA3-512_M64_W8_H10_z	64	64	8	175	10
XMSS_SHA3-512_M64_W8_H16_z	64	64	8	175	16
XMSS_SHA3-512_M64_W8_H20_z	64	64	8	175	20
XMSS_SHA3-512_M64_W16_H10_z	64	64	16	131	10
XMSS_SHA3-512_M64_W16_H16_z	64	64	16	131	16
XMSS_SHA3-512_M64_W16_H20_z	64	64	16	131	20

Table 4

5.4. XMSS^MT Parameters

To fully describe an XMSS^MT signature method, the parameters m, n, w, h, and d, as well as the functions F, H, H_m and PRF_m MUST be specified. This section defines several XMSS^MT signature systems, each of which is identified by a name.

XDR formats for XMSS^MT are listed in Appendix C.

5.4.1. XMSS^MT Parameters

We first define XMSS^MT signature methods as described in Section 4.2. We define parameter sets that implement the functions using AES and SHA3 as described above as well as pure SHA3 parameter sets.

5.4.1.1. XMSS^MT Parameters with AES and SHA3

The following XMSS^MT signature methods implement the functions F, H, H_m and PRF_m using AES and SHA3 as described above.

Name	m	n	w	l	h	d
XMSSMT_AES128_M32_W4_H20_D2	32	16	4	133	20	2
XMSSMT_AES128_M32_W4_H20_D4	32	16	4	133	20	4
XMSSMT_AES128_M32_W4_H40_D2	32	16	4	133	40	2
XMSSMT_AES128_M32_W4_H40_D4	32	16	4	133	40	4
XMSSMT_AES128_M32_W4_H40_D8	32	16	4	133	40	8
XMSSMT_AES128_M32_W4_H60_D3	32	16	4	133	60	3
XMSSMT_AES128_M32_W4_H60_D6	32	16	4	133	60	6
XMSSMT_AES128_M32_W4_H60_D12	32	16	4	133	60	12
XMSSMT_AES128_M32_W8_H20_D2	32	16	8	90	20	2
XMSSMT_AES128_M32_W8_H20_D4	32	16	8	90	20	4
XMSSMT_AES128_M32_W8_H40_D2	32	16	8	90	40	2
XMSSMT_AES128_M32_W8_H40_D4	32	16	8	90	40	4

XMSSMT_AES128_M32_W8_H40_D8	32	16	8	90	40	8
XMSSMT_AES128_M32_W8_H60_D3	32	16	8	90	60	3
XMSSMT_AES128_M32_W8_H60_D6	32	16	8	90	60	6
XMSSMT_AES128_M32_W8_H60_D12	32	16	8	90	60	12
XMSSMT_AES128_M32_W16_H20_D2	32	16	16	67	20	2
XMSSMT_AES128_M32_W16_H20_D4	32	16	16	67	20	4
XMSSMT_AES128_M32_W16_H40_D2	32	16	16	67	40	2
XMSSMT_AES128_M32_W16_H40_D4	32	16	16	67	40	4
XMSSMT_AES128_M32_W16_H40_D8	32	16	16	67	40	8
XMSSMT_AES128_M32_W16_H60_D3	32	16	16	67	60	3
XMSSMT_AES128_M32_W16_H60_D6	32	16	16	67	60	6
XMSSMT_AES128_M32_W16_H60_D12	32	16	16	67	60	12

Table 5

5.4.1.2. XMSS^MT Parameters with SHA3

The following XMSS^MT signature methods implement the functions F, H, H_m and PRF_m solely using SHA3 as described above.

Name	m	n	w	l	h	d
XMSSMT_SHA3-256_M32_W4_H20_D2	32	32	4	133	20	2
XMSSMT_SHA3-256_M32_W4_H20_D4	32	32	4	133	20	4
XMSSMT_SHA3-256_M32_W4_H40_D2	32	32	4	133	40	2
XMSSMT_SHA3-256_M32_W4_H40_D4	32	32	4	133	40	4
XMSSMT_SHA3-256_M32_W4_H40_D8	32	32	4	133	40	8
XMSSMT_SHA3-256_M32_W4_H60_D3	32	32	4	133	60	3
XMSSMT_SHA3-256_M32_W4_H60_D6	32	32	4	133	60	6

XMSSMT_SHA3-256_M32_W4_H60_D12	32	32	4	133	60	12
XMSSMT_SHA3-256_M32_W8_H20_D2	32	32	8	90	20	2
XMSSMT_SHA3-256_M32_W8_H20_D4	32	32	8	90	20	4
XMSSMT_SHA3-256_M32_W8_H40_D2	32	32	8	90	40	2
XMSSMT_SHA3-256_M32_W8_H40_D4	32	32	8	90	40	4
XMSSMT_SHA3-256_M32_W8_H40_D8	32	32	8	90	40	8
XMSSMT_SHA3-256_M32_W8_H60_D3	32	32	8	90	60	3
XMSSMT_SHA3-256_M32_W8_H60_D6	32	32	8	90	60	6
XMSSMT_SHA3-256_M32_W8_H60_D12	32	32	8	90	60	12
XMSSMT_SHA3-256_M32_W16_H20_D2	32	32	16	67	20	2
XMSSMT_SHA3-256_M32_W16_H20_D4	32	32	16	67	20	4
XMSSMT_SHA3-256_M32_W16_H40_D2	32	32	16	67	40	2
XMSSMT_SHA3-256_M32_W16_H40_D4	32	32	16	67	40	4
XMSSMT_SHA3-256_M32_W16_H40_D8	32	32	16	67	40	8
XMSSMT_SHA3-256_M32_W16_H60_D3	32	32	16	67	60	3
XMSSMT_SHA3-256_M32_W16_H60_D6	32	32	16	67	60	6
XMSSMT_SHA3-256_M32_W16_H60_D12	32	32	16	67	60	12
XMSSMT_SHA3-512_M64_W4_H20_D2	64	64	4	261	20	2
XMSSMT_SHA3-512_M64_W4_H20_D4	64	64	4	261	20	4
XMSSMT_SHA3-512_M64_W4_H40_D2	64	64	4	261	40	2
XMSSMT_SHA3-512_M64_W4_H40_D4	64	64	4	261	40	4
XMSSMT_SHA3-512_M64_W4_H40_D8	64	64	4	261	40	8
XMSSMT_SHA3-512_M64_W4_H60_D3	64	64	4	261	60	3
XMSSMT_SHA3-512_M64_W4_H60_D6	64	64	4	261	60	6

Name	m	n	w	l	h	d
XMSSMT_SHA3-512_M64_W4_H60_D12	64	64	4	261	60	12
XMSSMT_SHA3-512_M64_W8_H20_D2	64	64	8	175	20	2
XMSSMT_SHA3-512_M64_W8_H20_D4	64	64	8	175	20	4
XMSSMT_SHA3-512_M64_W8_H40_D2	64	64	8	175	40	2
XMSSMT_SHA3-512_M64_W8_H40_D4	64	64	8	175	40	4
XMSSMT_SHA3-512_M64_W8_H40_D8	64	64	8	175	40	8
XMSSMT_SHA3-512_M64_W8_H60_D3	64	64	8	175	60	3
XMSSMT_SHA3-512_M64_W8_H60_D6	64	64	8	175	60	6
XMSSMT_SHA3-512_M64_W8_H60_D12	64	64	8	175	60	12
XMSSMT_SHA3-512_M64_W16_H20_D2	64	64	16	131	20	2
XMSSMT_SHA3-512_M64_W16_H20_D4	64	64	16	131	20	4
XMSSMT_SHA3-512_M64_W16_H40_D2	64	64	16	131	40	2
XMSSMT_SHA3-512_M64_W16_H40_D4	64	64	16	131	40	4
XMSSMT_SHA3-512_M64_W16_H40_D8	64	64	16	131	40	8
XMSSMT_SHA3-512_M64_W16_H60_D3	64	64	16	131	60	3
XMSSMT_SHA3-512_M64_W16_H60_D6	64	64	16	131	60	6
XMSSMT_SHA3-512_M64_W16_H60_D12	64	64	16	131	60	12

Table 6

5.4.2. XMSS^MT Parameters With Empty Bitmasks

We now define XMSS^MT signature methods for the zero bitmasks special case described in Section 5.1. For this setting all signature methods implement the functions F, H, H_m and PRF_m solely using SHA3 as described above.

Name	m	n	w	l	h	d

XMSSMT_SHA3-256_M32_W4_H20_D2_z	32	32	4	133	20	2
XMSSMT_SHA3-256_M32_W4_H20_D4_z	32	32	4	133	20	4
XMSSMT_SHA3-256_M32_W4_H40_D2_z	32	32	4	133	40	2
XMSSMT_SHA3-256_M32_W4_H40_D4_z	32	32	4	133	40	4
XMSSMT_SHA3-256_M32_W4_H40_D8_z	32	32	4	133	40	8
XMSSMT_SHA3-256_M32_W4_H60_D3_z	32	32	4	133	60	3
XMSSMT_SHA3-256_M32_W4_H60_D6_z	32	32	4	133	60	6
XMSSMT_SHA3-256_M32_W4_H60_D12_z	32	32	4	133	60	12
XMSSMT_SHA3-256_M32_W8_H20_D2_z	32	32	8	90	20	2
XMSSMT_SHA3-256_M32_W8_H20_D4_z	32	32	8	90	20	4
XMSSMT_SHA3-256_M32_W8_H40_D2_z	32	32	8	90	40	2
XMSSMT_SHA3-256_M32_W8_H40_D4_z	32	32	8	90	40	4
XMSSMT_SHA3-256_M32_W8_H40_D8_z	32	32	8	90	40	8
XMSSMT_SHA3-256_M32_W8_H60_D3_z	32	32	8	90	60	3
XMSSMT_SHA3-256_M32_W8_H60_D6_z	32	32	8	90	60	6
XMSSMT_SHA3-256_M32_W8_H60_D12_z	32	32	16	67	60	12
XMSSMT_SHA3-256_M32_W16_H20_D2_z	32	32	16	67	20	2
XMSSMT_SHA3-256_M32_W16_H20_D4_z	32	32	16	67	20	4
XMSSMT_SHA3-256_M32_W16_H40_D2_z	32	32	16	67	40	2
XMSSMT_SHA3-256_M32_W16_H40_D4_z	32	32	16	67	40	4
XMSSMT_SHA3-256_M32_W16_H40_D8_z	32	32	16	67	40	8
XMSSMT_SHA3-256_M32_W16_H60_D3_z	32	32	16	67	60	3
XMSSMT_SHA3-256_M32_W16_H60_D6_z	32	32	16	67	60	6
XMSSMT_SHA3-256_M32_W16_H60_D12_z	32	32	16	67	60	12

XMSSMT_SHA3-512_M64_W4_H20_D2_z	64	64	4	261	20	2
XMSSMT_SHA3-512_M64_W4_H20_D4_z	64	64	4	261	20	4
XMSSMT_SHA3-512_M64_W4_H40_D2_z	64	64	4	261	40	2
XMSSMT_SHA3-512_M64_W4_H40_D4_z	64	64	4	261	40	4
XMSSMT_SHA3-512_M64_W4_H40_D8_z	64	64	4	261	40	8
XMSSMT_SHA3-512_M64_W4_H60_D3_z	64	64	4	261	60	3
XMSSMT_SHA3-512_M64_W4_H60_D6_z	64	64	4	261	60	6
XMSSMT_SHA3-512_M64_W4_H60_D12_z	64	64	4	261	60	12
XMSSMT_SHA3-512_M64_W8_H20_D2_z	64	64	8	175	20	2
XMSSMT_SHA3-512_M64_W8_H20_D4_z	64	64	8	175	20	4
XMSSMT_SHA3-512_M64_W8_H40_D2_z	64	64	8	175	40	2
XMSSMT_SHA3-512_M64_W8_H40_D4_z	64	64	8	175	40	4
XMSSMT_SHA3-512_M64_W8_H40_D8_z	64	64	8	175	40	8
XMSSMT_SHA3-512_M64_W8_H60_D3_z	64	64	8	175	60	3
XMSSMT_SHA3-512_M64_W8_H60_D6_z	64	64	8	175	60	6
XMSSMT_SHA3-512_M64_W8_H60_D12_z	64	64	8	175	60	12
XMSSMT_SHA3-512_M64_W16_H20_D2_z	64	64	16	131	20	2
XMSSMT_SHA3-512_M64_W16_H20_D4_z	64	64	16	131	20	4
XMSSMT_SHA3-512_M64_W16_H40_D2_z	64	64	16	131	40	2
XMSSMT_SHA3-512_M64_W16_H40_D4_z	64	64	16	131	40	4
XMSSMT_SHA3-512_M64_W16_H40_D8_z	64	64	16	131	40	8
XMSSMT_SHA3-512_M64_W16_H60_D3_z	64	64	16	131	60	3
XMSSMT_SHA3-512_M64_W16_H60_D6_z	64	64	16	131	60	6
XMSSMT_SHA3-512_M64_W16_H60_D12_z	64	64	16	131	60	12

Table 7

6. Rationale

The goal of this note is to describe the WOTS+, XMSS and XMSS^MT algorithms following the scientific literature. Other signature methods are out of scope and may be an interesting follow-on work. The description is done in a modular way that allows to base a description of stateless hash-based signature algorithms like SPHINCS [BHH15] on it.

The parameter w is constrained to powers of 2 to support simpler and more efficient implementations. Furthermore, w is restricted to the set {4, 8, 16}. No bigger values are included since the decrease in signature size then becomes less significant. The value w = 2 was not included since w = 4 leads to similar runtimes but a halved signature size. This is the case because while chains get twice as long, thereby increasing runtime, the number of chains is roughly halved. For instance, assuming m = n = 32, one obtains l = 38 for w = 2 and l = 19 for w = 4.

The signature and public key formats are designed so that they are easy to parse. Each format starts with a 32-bit enumeration value that indicates all of the details of the signature algorithm and hence defines all of the information that is needed in order to parse the format.

The enumeration values used in this note are palindromes, which have the same byte representation in either host order or network order. This fact allows an implementation to omit the conversion between byte order for those enumerations. Note however that the idx field used in XMSS and XMSS^MT signatures and secret keys must be properly converted to and from network byte order; this is the only field that requires such conversion. There are 2^32 XDR enumeration values, 2^16 of which are palindromes, which is adequate for the foreseeable future. If there is a need for more assignments, non-palindromes can be assigned.

7. IANA Considerations

The Internet Assigned Numbers Authority (IANA) is requested to create three registries: one for WOTS+ signatures as defined in Section 3, one for XMSS signatures and one for XMSS^MT signatures; the latter two being defined in Section 4. For the sake of clarity and convenience, the first sets of WOTS+, XMSS, and XMSS^MT parameter sets are defined in Section 5. Additions to these registries require that a specification be documented in an RFC or another permanent and readily available reference in sufficient details to make

interoperability between independent implementations possible. Each entry in the registry contains the following elements:

a short name, such as "XMSS_SHA3-512_M64_W16_H20",

a positive number, and

a reference to a specification that completely defines the signature method test cases that can be used to verify the correctness of an implementation.

Requests to add an entry to the registry MUST include the name and the reference. The number is assigned by IANA. These number assignments SHOULD use the smallest available palindromic number. Submitters SHOULD have their requests reviewed by the IRTF Crypto Forum Research Group (CFRG) at cfrg@ietf.org. Interested applicants that are unfamiliar with IANA processes should visit http://www.iana.org.

The numbers between 0xDDDDDDDD (decimal 3,722,304,989) and 0xFFFFFFFF (decimal 4,294,967,295) inclusive, will not be assigned by IANA, and are reserved for private use; no attempt will be made to prevent multiple sites from using the same value in different (and incompatible) ways [RFC2434].

The WOTS+ registry is as follows.

Name	Reference	Numeric Identifier
WOTSP_AES128_M32_W4	Section 5.2	0x01000001
WOTSP_AES128_M32_W8	Section 5.2	0x02000002
WOTSP_AES128_M32_W16	Section 5.2	0x03000003
WOTSP_SHA3-256_M32_W4	Section 5.2	0x04000004
WOTSP_SHA3-256_M32_W8	Section 5.2	0x05000005
WOTSP_SHA3-256_M32_W16	Section 5.2	0x06000006
WOTSP_SHA3-512_M64_W4	Section 5.2	0x07000007
WOTSP_SHA3-512_M64_W8	Section 5.2	0x08000008
WOTSP_SHA3-512_M64_W16	Section 5.2	0x09000009

Table 8

The XMSS registry is as follows.

Name	Reference	Numeric Identifier
XMSS_SHA3-256_M32_W4_H10_Z	Section 5.3	0x01000001
XMSS_SHA3-256_M32_W4_H16_Z	Section 5.3	0x02000002
XMSS_SHA3-256_M32_W4_H20_Z	Section 5.3	0x03000003
XMSS_SHA3-256_M32_W8_H10_Z	Section 5.3	0x04000004
XMSS_SHA3-256_M32_W8_H16_Z	Section 5.3	0x05000005
XMSS_SHA3-256_M32_W8_H20_Z	Section 5.3	0x06000006
XMSS_SHA3-256_M32_W16_H10_Z	Section 5.3	0x07000007
XMSS_SHA3-256_M32_W16_H16_Z	Section 5.3	0x08000008
XMSS_SHA3-256_M32_W16_H20_Z	Section 5.3	0x09000009
XMSS_SHA3-512_M64_W4_H10_Z	Section 5.3	0x0a00000a
XMSS_SHA3-512_M64_W4_H16_Z	Section 5.3	0x0b00000b
XMSS_SHA3-512_M64_W4_H20_Z	Section 5.3	0x0c00000c
XMSS_SHA3-512_M64_W8_H10_Z	Section 5.3	0x0d00000d
XMSS_SHA3-512_M64_W8_H16_Z	Section 5.3	0x0e00000e
XMSS_SHA3-512_M64_W8_H20_Z	Section 5.3	0x0f00000f
XMSS_SHA3-512_M64_W16_H10_Z	Section 5.3	0x01010101
XMSS_SHA3-512_M64_W16_H16_Z	Section 5.3	0x02010102
XMSS_SHA3-512_M64_W16_H20_Z	Section 5.3	0x03010103
XMSS_AES128_M32_W4_H10	Section 5.3	0x04010104
XMSS_AES128_M32_W4_H16	Section 5.3	0x05010105
XMSS_AES128_M32_W4_H20	Section 5.3	0x06010106

XMSS_AES128_M32_W8_H10	Section 5.3	0x07010107
XMSS_AES128_M32_W8_H16	Section 5.3	0x08010108
XMSS_AES128_M32_W8_H20	Section 5.3	0x09010109
XMSS_AES128_M32_W16_H10	Section 5.3	0x0a01010a
XMSS_AES128_M32_W16_H16	Section 5.3	0x0b01010b
XMSS_AES128_M32_W16_H20	Section 5.3	0x0c01010c
XMSS_SHA3-256_M32_W4_H10	Section 5.3	0x0d01010d
XMSS_SHA3-256_M32_W4_H16	Section 5.3	0x0e01010e
XMSS_SHA3-256_M32_W4_H20	Section 5.3	0x0f01010f
XMSS_SHA3-256_M32_W8_H10	Section 5.3	0x01020201
XMSS_SHA3-256_M32_W8_H16	Section 5.3	0x02020202
XMSS_SHA3-256_M32_W8_H20	Section 5.3	0x03020203
XMSS_SHA3-256_M32_W16_H10	Section 5.3	0x04020204
XMSS_SHA3-256_M32_W16_H16	Section 5.3	0x05020205
XMSS_SHA3-256_M32_W16_H20	Section 5.3	0x06020206
XMSS_SHA3-512_M64_W4_H10	Section 5.3	0x07020207
XMSS_SHA3-512_M64_W4_H16	Section 5.3	0x08020208
XMSS_SHA3-512_M64_W4_H20	Section 5.3	0x09020209
XMSS_SHA3-512_M64_W8_H10	Section 5.3	0x0a02020a
XMSS_SHA3-512_M64_W8_H16	Section 5.3	0x0b02020b
XMSS_SHA3-512_M64_W8_H20	Section 5.3	0x0c02020c
XMSS_SHA3-512_M64_W16_H10	Section 5.3	0x0d02020d
XMSS_SHA3-512_M64_W16_H16	Section 5.3	0x0e02020e
XMSS_SHA3-512_M64_W16_H20	Section 5.3	0x0f02020f

Table 9

The XMSS^MT registry is as follows.

Name	Reference	Numeric Identifier
XMSSMT_SHA3-256_M32_W4_H20_D2_Z	Section 5.4	0x01000001
XMSSMT_SHA3-256_M32_W4_H20_D4_Z	Section 5.4	0x02000002
XMSSMT_SHA3-256_M32_W4_H40_D2_Z	Section 5.4	0x03000003
XMSSMT_SHA3-256_M32_W4_H40_D4_Z	Section 5.4	0x04000004
XMSSMT_SHA3-256_M32_W4_H40_D8_Z	Section 5.4	0x05000005
XMSSMT_SHA3-256_M32_W4_H60_D3_Z	Section 5.4	0x06000006
XMSSMT_SHA3-256_M32_W4_H60_D6_Z	Section 5.4	0x07000007
XMSSMT_SHA3-256_M32_W4_H60_D12_Z	Section 5.4	0x08000008
XMSSMT_SHA3-256_M32_W8_H20_D2_Z	Section 5.4	0x09000009
XMSSMT_SHA3-256_M32_W8_H20_D4_Z	Section 5.4	0x0a00000a
XMSSMT_SHA3-256_M32_W8_H40_D2_Z	Section 5.4	0x0b00000b
XMSSMT_SHA3-256_M32_W8_H40_D4_Z	Section 5.4	0x0c00000c
XMSSMT_SHA3-256_M32_W8_H40_D8_Z	Section 5.4	0x0d00000d

XMSSMT_SHA3-256_M32_W8_H60_D3_Z	Section 5.4	0x0e00000e
XMSSMT_SHA3-256_M32_W8_H60_D6_Z	Section 5.4	0x0f00000f
XMSSMT_SHA3-256_M32_W8_H60_D12_Z	Section 5.4	0x00010100
XMSSMT_SHA3-256_M32_W16_H20_D2_Z	Section 5.4	0x01010101
XMSSMT_SHA3-256_M32_W16_H20_D4_Z	Section 5.4	0x02010102
XMSSMT_SHA3-256_M32_W16_H40_D2_Z	Section 5.4	0x03010103
XMSSMT_SHA3-256_M32_W16_H40_D4_Z	Section 5.4	0x04010104
XMSSMT_SHA3-256_M32_W16_H40_D8_Z	Section 5.4	0x05010105
XMSSMT_SHA3-256_M32_W16_H60_D3_Z	Section 5.4	0x06010106
XMSSMT_SHA3-256_M32_W16_H60_D6_Z	Section 5.4	0x07010107
XMSSMT_SHA3-256_M32_W16_H60_D12_Z	Section 5.4	0x08010108
XMSSMT_SHA3-512_M64_W4_H20_D2_Z	Section 5.4	0x09010109
XMSSMT_SHA3-512_M64_W4_H20_D4_Z	Section 5.4	0x0a01010a
XMSSMT_SHA3-512_M64_W4_H40_D2_Z	Section 5.4	0x0b01010b
XMSSMT_SHA3-512_M64_W4_H40_D4_Z	Section 5.4	0x0c01010c
XMSSMT_SHA3-512_M64_W4_H40_D8_Z	Section 5.4	0x0d01010d

XMSSMT_SHA3-512_M64_W4_H60_D3_Z	Section 5.4	0x0e01010e
XMSSMT_SHA3-512_M64_W4_H60_D6_Z	Section 5.4	0x0f01010f
XMSSMT_SHA3-512_M64_W4_H60_D12_Z	Section 5.4	0x00020200
XMSSMT_SHA3-512_M64_W8_H20_D2_Z	Section 5.4	0x01020201
XMSSMT_SHA3-512_M64_W8_H20_D4_Z	Section 5.4	0x02020202
XMSSMT_SHA3-512_M64_W8_H40_D2_Z	Section 5.4	0x03020203
XMSSMT_SHA3-512_M64_W8_H40_D4_Z	Section 5.4	0x04020204
XMSSMT_SHA3-512_M64_W8_H40_D8_Z	Section 5.4	0x05020205
XMSSMT_SHA3-512_M64_W8_H60_D3_Z	Section 5.4	0x06020206
XMSSMT_SHA3-512_M64_W8_H60_D6_Z	Section 5.4	0x07020207
XMSSMT_SHA3-512_M64_W8_H60_D12_Z	Section 5.4	0x08020208
XMSSMT_SHA3-512_M64_W16_H20_D2_Z	Section 5.4	0x09020209
XMSSMT_SHA3-512_M64_W16_H20_D4_Z	Section 5.4	0x0a02020a
XMSSMT_SHA3-512_M64_W16_H40_D2_Z	Section 5.4	0x0b02020b
XMSSMT_SHA3-512_M64_W16_H40_D4_Z	Section 5.4	0x0c02020c
XMSSMT_SHA3-512_M64_W16_H40_D8_Z	Section 5.4	0x0d02020d

XMSSMT_SHA3-512_M64_W16_H60_D3_Z	Section 5.4	0x0e02020e
XMSSMT_SHA3-512_M64_W16_H60_D6_Z	Section 5.4	0x0f02020f
XMSSMT_SHA3-512_M64_W16_H60_D12_Z	Section 5.4	0x00030300
XMSSMT_AES128_M32_W4_H20_D2	Section 5.4	0x01030301
XMSSMT_AES128_M32_W4_H20_D4	Section 5.4	0x02030302
XMSSMT_AES128_M32_W4_H40_D2	Section 5.4	0x03030303
XMSSMT_AES128_M32_W4_H40_D4	Section 5.4	0x04030304
XMSSMT_AES128_M32_W4_H40_D8	Section 5.4	0x05030305
XMSSMT_AES128_M32_W4_H60_D3	Section 5.4	0x06030306
XMSSMT_AES128_M32_W4_H60_D6	Section 5.4	0x07030307
XMSSMT_AES128_M32_W4_H60_D12	Section 5.4	0x08030308
XMSSMT_AES128_M32_W8_H20_D2	Section 5.4	0x09030309
XMSSMT_AES128_M32_W8_H20_D4	Section 5.4	0x0a03030a
XMSSMT_AES128_M32_W8_H40_D2	Section 5.4	0x0b03030b
XMSSMT_AES128_M32_W8_H40_D4	Section 5.4	0x0c03030c
XMSSMT_AES128_M32_W8_H40_D8	Section 5.4	0x0d03030d

XMSSMT_AES128_M32_W8_H60_D3	Section 5.4	0x0e03030e
XMSSMT_AES128_M32_W8_H60_D6	Section 5.4	0x0f03030f
XMSSMT_AES128_M32_W8_H60_D12	Section 5.4	0x00040400
XMSSMT_AES128_M32_W16_H20_D2	Section 5.4	0x01040401
XMSSMT_AES128_M32_W16_H20_D4	Section 5.4	0x02040402
XMSSMT_AES128_M32_W16_H40_D2	Section 5.4	0x03040403
XMSSMT_AES128_M32_W16_H40_D4	Section 5.4	0x04040404
XMSSMT_AES128_M32_W16_H40_D8	Section 5.4	0x05040405
XMSSMT_AES128_M32_W16_H60_D3	Section 5.4	0x06040406
XMSSMT_AES128_M32_W16_H60_D6	Section 5.4	0x07040407
XMSSMT_AES128_M32_W16_H60_D12	Section 5.4	0x08040408
XMSSMT_SHA3-256_M32_W4_H20_D2	Section 5.4	0x09040409
XMSSMT_SHA3-256_M32_W4_H20_D4	Section 5.4	0x0a04040a
XMSSMT_SHA3-256_M32_W4_H40_D2	Section 5.4	0x0b04040b
XMSSMT_SHA3-256_M32_W4_H40_D4	Section 5.4	0x0c04040c
XMSSMT_SHA3-256_M32_W4_H40_D8	Section 5.4	0x0d04040d

XMSSMT_SHA3-256_M32_W4_H60_D3	Section 5.4	0x0e04040e
XMSSMT_SHA3-256_M32_W4_H60_D6	Section 5.4	0x0f04040f
XMSSMT_SHA3-256_M32_W4_H60_D12	Section 5.4	0x00050500
XMSSMT_SHA3-256_M32_W8_H20_D2	Section 5.4	0x01050501
XMSSMT_SHA3-256_M32_W8_H20_D4	Section 5.4	0x02050502
XMSSMT_SHA3-256_M32_W8_H40_D2	Section 5.4	0x03050503
XMSSMT_SHA3-256_M32_W8_H40_D4	Section 5.4	0x04050504
XMSSMT_SHA3-256_M32_W8_H40_D8	Section 5.4	0x05050505
XMSSMT_SHA3-256_M32_W8_H60_D3	Section 5.4	0x06050506
XMSSMT_SHA3-256_M32_W8_H60_D6	Section 5.4	0x07050507
XMSSMT_SHA3-256_M32_W8_H60_D12	Section 5.4	0x08050508
XMSSMT_SHA3-256_M32_W16_H20_D2	Section 5.4	0x09050509
XMSSMT_SHA3-256_M32_W16_H20_D4	Section 5.4	0x0a05050a
XMSSMT_SHA3-256_M32_W16_H40_D2	Section 5.4	0x0b05050b
XMSSMT_SHA3-256_M32_W16_H40_D4	Section 5.4	0x0c05050c
XMSSMT_SHA3-256_M32_W16_H40_D8	Section 5.4	0x0d05050d

XMSSMT_SHA3-256_M32_W16_H60_D3	Section 5.4	0x0e05050e
XMSSMT_SHA3-256_M32_W16_H60_D6	Section 5.4	0x0f05050f
XMSSMT_SHA3-256_M32_W16_H60_D12	Section 5.4	0x00060600
XMSSMT_SHA3-512_M64_W4_H20_D2	Section 5.4	0x01060601
XMSSMT_SHA3-512_M64_W4_H20_D4	Section 5.4	0x02060602
XMSSMT_SHA3-512_M64_W4_H40_D2	Section 5.4	0x03060603
XMSSMT_SHA3-512_M64_W4_H40_D4	Section 5.4	0x04060604
XMSSMT_SHA3-512_M64_W4_H40_D8	Section 5.4	0x05060605
XMSSMT_SHA3-512_M64_W4_H60_D3	Section 5.4	0x06060606
XMSSMT_SHA3-512_M64_W4_H60_D6	Section 5.4	0x07060607
XMSSMT_SHA3-512_M64_W4_H60_D12	Section 5.4	0x08060608
XMSSMT_SHA3-512_M64_W8_H20_D2	Section 5.4	0x09060609
XMSSMT_SHA3-512_M64_W8_H20_D4	Section 5.4	0x0a06060a
XMSSMT_SHA3-512_M64_W8_H40_D2	Section 5.4	0x0b06060b
XMSSMT_SHA3-512_M64_W8_H40_D4	Section 5.4	0x0c06060c
XMSSMT_SHA3-512_M64_W8_H40_D8	Section 5.4	0x0d06060d

XMSSMT_SHA3-512_M64_W8_H60_D3	Section 5.4	0x0e06060e
XMSSMT_SHA3-512_M64_W8_H60_D6	Section 5.4	0x0f06060f
XMSSMT_SHA3-512_M64_W8_H60_D12	Section 5.4	0x00070700
XMSSMT_SHA3-512_M64_W16_H20_D2	Section 5.4	0x01070701
XMSSMT_SHA3-512_M64_W16_H20_D4	Section 5.4	0x02070702
XMSSMT_SHA3-512_M64_W16_H40_D2	Section 5.4	0x03070703
XMSSMT_SHA3-512_M64_W16_H40_D4	Section 5.4	0x04070704
XMSSMT_SHA3-512_M64_W16_H40_D8	Section 5.4	0x05070705
XMSSMT_SHA3-512_M64_W16_H60_D3	Section 5.4	0x06070706
XMSSMT_SHA3-512_M64_W16_H60_D6	Section 5.4	0x07070707
XMSSMT_SHA3-512_M64_W16_H60_D12	Section 5.4	0x08070708

Table 10

An IANA registration of a signature system does not constitute an endorsement of that system or its security.

8. Security Considerations

A signature system is considered secure if it prevents an attacker from forging a valid signature. More specifically, consider a setting in which an attacker gets a public key and can learn signatures on arbitrary messages of his choice. A signature system is secure if, even in this setting, the attacker can not produce a message signature pair of his choosing such that the verification algorithm accepts.

Preventing an attacker from mounting an attack means that the attack is computationally too expensive to be carried out. There exist various estimates when a computation is too expensive to be done. For that reason, this note only describes how expensive it is for an attacker to generate a forgery. Parameters are accompanied by a bit security value. The meaning of bit security is as follows. A parameter set grants b bits of security if the best attack takes at least 2^(b-1) bit operations to achieve a success probability of 1/2. Hence, to mount a successful attack, an attacker needs to perform 2^b bit operations on average. How the given values for bit security were estimated is described below.

8.1. Security Proofs

There exist formal security proofs for the schemes described here in the literature [Huelsing13a]. These proofs show that an attacker has to break at least one out of certain security properties of the used hash functions and PRFs to forge a signature. The proofs in [Huelsing13a] do not consider the initial message compression. For the scheme without initial message compression, these proofs show that an attacker has to break certain minimal security properties. In particular, it is not sufficient to break the collision resistance of the hash functions to generate a forgery.

It is a folklore that one can securely combine a secure signature scheme for fixed length messages with an initial message digest. It is easy to proof that an attacker either must break the security of the fixed-input-length signature scheme or the collision resistance of the used hash function. XMSS and XMSS^MT use a known trick to prevent the applicability of collision attacks. Namely, the schemes use a randomized message hash. For technical reasons, it is not possible to formally prove that the resulting scheme is secure if the hash function is not collision-resistant but fulfills some weaker security properties.

The given bit security values were estimated based on the complexity of the best known generic attacks against the required security properties of the used hash functions and PRFs.

8.2. Security Assumptions

The security assumptions made to argue for the security of the described schemes are minimal. Any signature algorithm that allows arbitrary size messages relies on the security of a cryptographic hash function. For the schemes described here this is already sufficient to be secure. In contrast, common signature schemes like RSA, DSA, and ECDSA additionally rely on the conjectured hardness of certain mathematical problems.

8.3. Post-Quantum Security

A post-quantum cryptosystem is a system that is secure against attackers with access to a reasonably sized quantum computer. At the time of writing this note, whether or not it is feasible to build such machine is an open conjecture. However, significant progress was made over the last few years in this regard.

In contrast to RSA, DSA, and ECDSA, the described signature systems are post-quantum-secure if they are used with an appropriate cryptographic hash function. In particular, for post-quantum security, the size of m and n must be twice the size required for classical security. This is in order to protect against quantum square root attacks due to Grover's algorithm. It has been shown that Grover's algorithm is optimal for finding preimages and collisions.

9. Acknowledgements

We would like to thank Burt Kaliski, David McGrew, and Aziz Mohaisen for their help.

10. References

10.1. Normative References

[DRAFTFIPS202]
National Institute of Standards and Technology, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", Draft FIPS 202, 2014.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 2434, October 1998.

[RFC4506] Eisler, M., "XDR: External Data Representation Standard", STD 67, RFC 4506, May 2006.

10.2. Informative References

[BDH11] Buchmann, J., Dahmen, E., and A. Huelsing, "XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions", Lecture Notes in Computer Science volume 7071. Post-Quantum Cryptography, 2011.

[BDS09] Buchmann, J., Dahmen, E., and M. Szydlo, "Hash-based Digital Signature Schemes", Book chapter Post-Quantum Cryptography, Springer, 2009.

[BHH15] Bernstein, D., Hopwood, D., Huelsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., and Z. Wilcox-O'Hearn, "SPHINCS: practical stateless hash-based signatures", To appear. Advances in Cryptology - EUROCRYPT, 2015.

[DC14] McGrew, D. and M. Curcio, "Hash-based signatures", draft-mcgrew-hash-sigs-02 (work in progress), July 2014.

[HRB13] Huelsing, A., Rausch, L., and J. Buchmann, "Optimal Parameters for XMSS^MT", Lecture Notes in Computer Science volume 8128. CD-ARES, 2013.

[Huelsing13]
Huelsing, A., "W-OTS+ - Shorter Signatures for Hash-Based Signature Schemes", Lecture Notes in Computer Science volume 7918. Progress in Cryptology - AFRICACRYPT, 2013.

[Huelsing13a]
Huelsing, A., "Practical Forward Secure Signatures using Minimal Security Assumptions", PhD thesis TU Darmstadt, 2013.

[Kaliski15]
Kaliski, B., "Shoring up the Infrastructure: A Strategy for Standardizing Hash Signatures", Post Quantum NIST Workshop on Cybersecurity in a Post-Quantum World, 2015.

[Merkle79]
Merkle, R., "Secrecy, Authentication, and Public Key Systems", Stanford University Information Systems Laboratory Technical Report 1979-1, 1979.

Appendix A. WOTS+ XDR Formats

The WOTS+ signature and public key formats are formally defined using XDR [RFC4506] in order to provide an unambiguous, machine readable definition. Though XDR is used, these formats are simple and easy to parse without any special tools. To avoid the need to convert to and from network / host byte order, the enumeration values are all palindromes.

WOTS+ parameter sets are defined using XDR syntax as follows:

```
/* ots_algorithm_type identifies a particular
   signature algorithm */

enum ots_algorithm_type {
  wotsp_reserved         = 0x00000000,
  wotsp_aes128_m32_w4    = 0x01000001,
  wotsp_aes128_m32_w8    = 0x02000002,
  wotsp_aes128_m32_w16   = 0x03000003,
  wotsp_sha3-256_m32_w4  = 0x04000004,
  wotsp_sha3-256_m32_w8  = 0x05000005,
  wotsp_sha3-256_m32_w16 = 0x06000006,
  wotsp_sha3-512_m64_w4  = 0x07000007,
  wotsp_sha3-512_m64_w8  = 0x08000008,
  wotsp_sha3-512_m64_w16 = 0x09000009,
};
```

WOTS+ signatures are defined using XDR syntax as follows:

```
/* Byte strings */

typedef opaque bytestring32[32];
typedef opaque bytestring64[64];

union ots_signature switch (ots_algorithm_type type) {
  case wotsp_aes128_m32_w4:
  case wotsp_sha3-256_m32_w4:
    bytestring32 ots_sig_m32_l133[133];

  case wotsp_aes128_m32_w8:
  case wotsp_sha3-256_m32_w8:
    bytestring32 ots_sig_m32_l90[90];

  case wotsp_aes128_m32_w16:
  case wotsp_sha3-256_m32_w16:
    bytestring32 ots_sig_m32_l67[67];

  case wotsp_sha3-512_m64_w4:
    bytestring64 ots_sig_m64_l261[261];

  case wotsp_sha3-512_m64_w8:
    bytestring64 ots_sig_m64_l75[175];

  case wotsp_sha3-512_m64_w16:
    bytestring64 ots_sig_m64_l18[131];

  default:
    void;   /* error condition */
};
```

WOTS+ public keys are defined using XDR syntax as follows:

```
      union ots_pubkey switch (ots_algorithm_type type) {
        case wotsp_aes128_m32_w4:
        case wotsp_sha3-256_m32_w4:
          bytestring32 ots_pubk_m32_l133[133];

        case wotsp_aes128_m32_w8:
        case wotsp_sha3-256_m32_w8:
          bytestring32 ots_pubk_m32_l90[90];

        case wotsp_aes128_m32_w16:
        case wotsp_sha3-256_m32_w16:
          bytestring32 ots_pubk_m32_l67[67];

        case wotsp_sha3-512_m64_w4:
          bytestring64 ots_pubk_m64_l261[261];

        case wotsp_sha3-512_m64_w8:
          bytestring64 ots_pubk_m64_l75[175];

        case wotsp_sha3-512_m64_w16:
          bytestring64 ots_pubk_m64_l18[131];

        default:
          void;   /* error condition */
      };
```

Appendix B. XMSS XDR Formats

XMSS parameter sets are defined using XDR syntax as follows:

```
      /* Byte strings */

      typedef opaque bytestring4[4];
      typedef opaque bytestring16[16];

      /* Definition of parameter sets */

      enum xmss_algorithm_type {
        xmss_reserved                = 0x00000000,

        /* Empty bitmasks */

        /* 128 bit classical security, 85 bit post-quantum security */

        xmss_sha3-256_m32_w4_h10_z  = 0x01000001,
        xmss_sha3-256_m32_w4_h16_z  = 0x02000002,
```

```
xmss_sha3-256_m32_w4_h20_z  = 0x03000003,

xmss_sha3-256_m32_w8_h10_z  = 0x04000004,
xmss_sha3-256_m32_w8_h16_z  = 0x05000005,
xmss_sha3-256_m32_w8_h20_z  = 0x06000006,

xmss_sha3-256_m32_w16_h10_z = 0x07000007,
xmss_sha3-256_m32_w16_h16_z = 0x08000008,
xmss_sha3-256_m32_w16_h20_z = 0x09000009,

/* 256 bit classical security, 170 bit post-quantum security */

xmss_sha3-512_m64_w4_h10_z  = 0x0a00000a,
xmss_sha3-512_m64_w4_h16_z  = 0x0b00000b,
xmss_sha3-512_m64_w4_h20_z  = 0x0c00000c,

xmss_sha3-512_m64_w8_h10_z  = 0x0d00000d,
xmss_sha3-512_m64_w8_h16_z  = 0x0e00000e,
xmss_sha3-512_m64_w8_h20_z  = 0x0f00000f,

xmss_sha3-512_m64_w16_h10_z = 0x01010101,
xmss_sha3-512_m64_w16_h16_z = 0x02010102,
xmss_sha3-512_m64_w16_h20_z = 0x03010103,

/* Non-empty bitmasks */

/* 128 bit classical security, 64 bit post-quantum security */

xmss_aes128_m32_w4_h10        = 0x04010104,
xmss_aes128_m32_w4_h16        = 0x05010105,
xmss_aes128_m32_w4_h20        = 0x06010106,

xmss_aes128_m32_w8_h10        = 0x07010107,
xmss_aes128_m32_w8_h16        = 0x08010108,
xmss_aes128_m32_w8_h20        = 0x09010109,

xmss_aes128_m32_w16_h10       = 0x0a01010a,
xmss_aes128_m32_w16_h16       = 0x0b01010b,
xmss_aes128_m32_w16_h20       = 0x0c01010c,

/* 256 bit classical security, 128 bit post-quantum security */

xmss_sha3-256_m32_w4_h10      = 0x0d01010d,
xmss_sha3-256_m32_w4_h16      = 0x0e01010e,
xmss_sha3-256_m32_w4_h20      = 0x0f01010f,

xmss_sha3-256_m32_w8_h10      = 0x01020201,
xmss_sha3-256_m32_w8_h16      = 0x02020202,
```

```
    xmss_sha3-256_m32_w8_h20      = 0x03020203,

    xmss_sha3-256_m32_w16_h10     = 0x04020204,
    xmss_sha3-256_m32_w16_h16     = 0x05020205,
    xmss_sha3-256_m32_w16_h20     = 0x06020206,

    /* 512 bit classical security, 256 bit post-quantum security */

    xmss_sha3-512_m64_w4_h10      = 0x07020207,
    xmss_sha3-512_m64_w4_h16      = 0x08020208,
    xmss_sha3-512_m64_w4_h20      = 0x09020209,

    xmss_sha3-512_m64_w8_h10      = 0x0a02020a,
    xmss_sha3-512_m64_w8_h16      = 0x0b02020b,
    xmss_sha3-512_m64_w8_h20      = 0x0c02020c,

    xmss_sha3-512_m64_w16_h10     = 0x0d02020d,
    xmss_sha3-512_m64_w16_h16     = 0x0e02020e,
    xmss_sha3-512_m64_w16_h20     = 0x0f02020f,
  };
```

XMSS signatures are defined using XDR syntax as follows:

```
  /* Authentication path types */

  union xmss_path switch (xmss_algorithm_type type) {
    case xmss_sha3-256_m32_w4_h10_z:
    case xmss_sha3-256_m32_w8_h10_z:
    case xmss_sha3-256_m32_w16_h10_z:
    case xmss_sha3-256_m32_w4_h10:
    case xmss_sha3-256_m32_w8_h10:
    case xmss_sha3-256_m32_w16_h10:
      bytestring32 path_n32_t10[10];

    case xmss_sha3-256_m32_w4_h16_z:
    case xmss_sha3-256_m32_w8_h16_z:
    case xmss_sha3-256_m32_w16_h16_z:
    case xmss_sha3-256_m32_w4_h16:
    case xmss_sha3-256_m32_w8_h16:
    case xmss_sha3-256_m32_w16_h16:
      bytestring32 path_n32_t16[16];

    case xmss_sha3-256_m32_w4_h20_z:
    case xmss_sha3-256_m32_w8_h20_z:
    case xmss_sha3-256_m32_w16_h20_z:
    case xmss_sha3-256_m32_w4_h20:
```

```
        case xmss_sha3-256_m32_w8_h20:
        case xmss_sha3-256_m32_w16_h20:
          bytestring32 path_n32_t20[20];

        case xmss_sha3-512_m64_w4_h10_z:
        case xmss_sha3-512_m64_w8_h10_z:
        case xmss_sha3-512_m64_w16_h10_z:
        case xmss_sha3-512_m64_w4_h10:
        case xmss_sha3-512_m64_w8_h10:
        case xmss_sha3-512_m64_w16_h10:
          bytestring64 path_n64_t10[10];

        case xmss_sha3-512_m64_w4_h16_z:
        case xmss_sha3-512_m64_w8_h16_z:
        case xmss_sha3-512_m64_w16_h16_z:
        case xmss_sha3-512_m64_w4_h16:
        case xmss_sha3-512_m64_w8_h16:
        case xmss_sha3-512_m64_w16_h16:
          bytestring64 path_n64_t16[16];

        case xmss_sha3-512_m64_w4_h20_z:
        case xmss_sha3-512_m64_w8_h20_z:
        case xmss_sha3-512_m64_w16_h20_z:
        case xmss_sha3-512_m64_w4_h20:
        case xmss_sha3-512_m64_w8_h20:
        case xmss_sha3-512_m64_w16_h20:
          bytestring64 path_n64_t20[20];

        case xmss_aes128_m32_w4_h10:
        case xmss_aes128_m32_w8_h10:
        case xmss_aes128_m32_w16_h10:
          bytestring16 path_n16_t10[10];

        case xmss_aes128_m32_w4_h16:
        case xmss_aes128_m32_w8_h16:
        case xmss_aes128_m32_w16_h16:
          bytestring16 path_n16_t16[16];

        case xmss_aes128_m32_w4_h20:
        case xmss_aes128_m32_w8_h20:
        case xmss_aes128_m32_w16_h20:
          bytestring16 path_n16_t20[20];

        default:
          void;     /* error condition */
      };

      /* Types for XMSS random strings */
```

```
      union random_string_xmss switch (xmss_algorithm_type type) {
        case xmss_sha3-256_m32_w4_h10_z:
        case xmss_sha3-256_m32_w4_h16_z:
        case xmss_sha3-256_m32_w4_h20_z:
        case xmss_sha3-256_m32_w8_h10_z:
        case xmss_sha3-256_m32_w8_h16_z:
        case xmss_sha3-256_m32_w8_h20_z:
        case xmss_sha3-256_m32_w16_h10_z:
        case xmss_sha3-256_m32_w16_h16_z:
        case xmss_sha3-256_m32_w16_h20_z:
        case xmss_sha3-256_m32_w4_h10:
        case xmss_sha3-256_m32_w4_h16:
        case xmss_sha3-256_m32_w4_h20:
        case xmss_sha3-256_m32_w8_h10:
        case xmss_sha3-256_m32_w8_h16:
        case xmss_sha3-256_m32_w8_h20:
        case xmss_sha3-256_m32_w16_h10:
        case xmss_sha3-256_m32_w16_h16:
        case xmss_sha3-256_m32_w16_h20:
        case xmss_aes128_m32_w4_h10:
        case xmss_aes128_m32_w4_h16:
        case xmss_aes128_m32_w4_h20:
        case xmss_aes128_m32_w8_h10:
        case xmss_aes128_m32_w8_h16:
        case xmss_aes128_m32_w8_h20:
        case xmss_aes128_m32_w16_h10:
        case xmss_aes128_m32_w16_h16:
        case xmss_aes128_m32_w16_h20:
          bytestring32 rand_m32;

        case xmss_sha3-512_m64_w4_h10_z:
        case xmss_sha3-512_m64_w4_h16_z:
        case xmss_sha3-512_m64_w4_h20_z:
        case xmss_sha3-512_m64_w8_h10_z:
        case xmss_sha3-512_m64_w8_h16_z:
        case xmss_sha3-512_m64_w8_h20_z:
        case xmss_sha3-512_m64_w16_h10_z:
        case xmss_sha3-512_m64_w16_h16_z:
        case xmss_sha3-512_m64_w16_h20_z:
        case xmss_sha3-512_m64_w4_h10:
        case xmss_sha3-512_m64_w4_h16:
        case xmss_sha3-512_m64_w4_h20:
        case xmss_sha3-512_m64_w8_h10:
        case xmss_sha3-512_m64_w8_h16:
        case xmss_sha3-512_m64_w8_h20:
        case xmss_sha3-512_m64_w16_h10:
        case xmss_sha3-512_m64_w16_h16:
        case xmss_sha3-512_m64_w16_h20:
```

```
          bytestring64 rand_m64;

        default:
          void;     /* error condition */
      };

      /* Corresponding WOTS+ type for given XMSS type */

      union xmss_ots_signature switch (xmss_algorithm_type type) {
        case xmss_sha3-256_m32_w4_h10_z:
        case xmss_sha3-256_m32_w4_h16_z:
        case xmss_sha3-256_m32_w4_h20_z:
          wotsp_sha3-256_m32_w4;

        case xmss_sha3-256_m32_w8_h10_z:
        case xmss_sha3-256_m32_w8_h16_z:
        case xmss_sha3-256_m32_w8_h20_z:
          wotsp_sha3-256_m32_w8;

        case xmss_sha3-256_m32_w16_h10_z:
        case xmss_sha3-256_m32_w16_h16_z:
        case xmss_sha3-256_m32_w16_h20_z:
          wotsp_sha3-256_m32_w16

        case xmss_sha3-512_m64_w4_h10_z:
        case xmss_sha3-512_m64_w4_h16_z:
        case xmss_sha3-512_m64_w4_h20_z:
          wotsp_sha3-512_m64_w4;

        case xmss_sha3-512_m64_w8_h10_z:
        case xmss_sha3-512_m64_w8_h16_z:
        case xmss_sha3-512_m64_w8_h20_z:
          wotsp_sha3-512_m64_w8;

        case xmss_sha3-512_m64_w16_h10_z:
        case xmss_sha3-512_m64_w16_h16_z:
        case xmss_sha3-512_m64_w16_h20_z:
          wotsp_sha3-512_m64_w16;

        case xmss_aes128_m32_w4_h10:
        case xmss_aes128_m32_w4_h16:
        case xmss_aes128_m32_w4_h20:
          wotsp_aes128_m32_w4;

        case xmss_aes128_m32_w8_h10:
        case xmss_aes128_m32_w8_h16:
        case xmss_aes128_m32_w8_h20:
          wotsp_aes128_m32_w8;
```

```
          case xmss_aes128_m32_w16_h10:
          case xmss_aes128_m32_w16_h16:
          case xmss_aes128_m32_w16_h20:
            wotsp_aes128_m32_w16;

          case xmss_sha3-256_m32_w4_h10:
          case xmss_sha3-256_m32_w4_h16:
          case xmss_sha3-256_m32_w4_h20:
            wotsp_sha3-256_m32_w4;

          case xmss_sha3-256_m32_w8_h10:
          case xmss_sha3-256_m32_w8_h16:
          case xmss_sha3-256_m32_w8_h20:
            wotsp_sha3-256_m32_w8;

          case xmss_sha3-256_m32_w16_h10:
          case xmss_sha3-256_m32_w16_h16:
          case xmss_sha3-256_m32_w16_h20:
            wotsp_sha3-256_m32_w16;

          case xmss_sha3-512_m64_w4_h10:
          case xmss_sha3-512_m64_w4_h16:
          case xmss_sha3-512_m64_w4_h20:
            wotsp_sha3-512_m64_w4;

          case xmss_sha3-512_m64_w8_h10:
          case xmss_sha3-512_m64_w8_h16:
          case xmss_sha3-512_m64_w8_h20:
            wotsp_sha3-512_m64_w8;

          case xmss_sha3-512_m64_w16_h10:
          case xmss_sha3-512_m64_w16_h16:
          case xmss_sha3-512_m64_w16_h20:
            wotsp_sha3-512_m64_w16;

          default:
            void;     /* error condition */
        };

        /* XMSS signature structure */

        struct xmss_signature {
          /* WOTS+ key pair index */
          bytestring4 idx_sig;
          /* Random string for randomized hashing */
          random_string_xmss rand_string;
          /* WOTS+ signature */
          xmss_ots_signature sig_ots;
```

```
        /* authentication path */
        xmss_path nodes;
      };
```

When no bitmasks are used, XMSS public keys are defined using XDR syntax as follows:

```
      /* Types for XMSS root node */

      union xmss_root switch (xmss_algorithm_type type) {
        case xmss_sha3-256_m32_w4_h10_z:
        case xmss_sha3-256_m32_w4_h16_z:
        case xmss_sha3-256_m32_w4_h20_z:
        case xmss_sha3-256_m32_w8_h10_z:
        case xmss_sha3-256_m32_w16_h10_z:
        case xmss_sha3-256_m32_w8_h16_z:
        case xmss_sha3-256_m32_w16_h16_z:
        case xmss_sha3-256_m32_w8_h20_z:
        case xmss_sha3-256_m32_w16_h20_z:
          bytestring32 root_n32;

        case xmss_sha3-512_m64_w4_h10_z:
        case xmss_sha3-512_m64_w4_h16_z:
        case xmss_sha3-512_m64_w4_h20_z:
        case xmss_sha3-512_m64_w8_h10_z:
        case xmss_sha3-512_m64_w16_h10_z:
        case xmss_sha3-512_m64_w8_h16_z:
        case xmss_sha3-512_m64_w16_h16_z:
        case xmss_sha3-512_m64_w8_h20_z:
        case xmss_sha3-512_m64_w16_h20_z:
          bytestring64 root_n64;

        default:
          void;     /* error condition */
      };

      /* XMSS public key structure */

      struct xmss_public_key {
        xmss_root root;  /* Root node */
      };
```

When bitmasks are used, XMSS public keys are defined using XDR syntax as follows:

```
/* Types for XMSS bitmasks */

union xmss_bm switch (xmss_algorithm_type type) {
  case xmss_aes128_m32_w4_h10:
    bytestring16 bm_n16_bm36[36];

  case xmss_aes128_m32_w4_h16:
    bytestring16 bm_n16_bm48[48];

  case xmss_aes128_m32_w4_h20:
    bytestring16 bm_n16_bm56[56];

  case xmss_aes128_m32_w8_h10:
  case xmss_aes128_m32_w16_h10:
    bytestring16 bm_n16_bm34[34];

  case xmss_aes128_m32_w8_h16:
  case xmss_aes128_m32_w16_h16:
    bytestring16 bm_n16_bm46[46];

  case xmss_aes128_m32_w8_h20:
  case xmss_aes128_m32_w16_h20:
    bytestring16 bm_n16_bm54[54];

  case xmss_sha3-256_m32_w4_h10:
    bytestring32 bm_n32_bm36[36];

  case xmss_sha3-256_m32_w4_h16:
    bytestring32 bm_n32_bm48[48];

  case xmss_sha3-256_m32_w4_h20:
    bytestring32 bm_n32_bm56[56];

  case xmss_sha3-256_m32_w8_h10:
  case xmss_sha3-256_m32_w16_h10:
    bytestring32 bm_n32_bm34[34];

  case xmss_sha3-256_m32_w8_h16:
  case xmss_sha3-256_m32_w16_h16:
    bytestring32 bm_n32_bm46[46];

  case xmss_sha3-256_m32_w8_h20:
  case xmss_sha3-256_m32_w16_h20:
    bytestring32 bm_n32_bm54[54];

  case xmss_sha3-512_m64_w4_h10:
    bytestring64 bm_n64_bm38[38];
```

```
        case xmss_sha3-512_m64_w4_h16:
          bytestring64 bm_n64_bm50[50];

        case xmss_sha3-512_m64_w4_h20:
          bytestring64 bm_n64_bm58[58];

        case xmss_sha3-512_m64_w8_h10:
        case xmss_sha3-512_m64_w16_h10:
          bytestring64 bm_n64_bm36[36];

        case xmss_sha3-512_m64_w8_h16:
        case xmss_sha3-512_m64_w16_h16:
          bytestring64 bm_n64_bm48[48];

        case xmss_sha3-512_m64_w8_h20:
        case xmss_sha3-512_m64_w16_h20:
          bytestring64 bm_n64_bm56[56];

        default:
          void;     /* error condition */
      };

      /* Types for XMSS root node */

      union xmss_root switch (xmss_algorithm_type type) {
        case xmss_aes128_m32_w4_h10:
        case xmss_aes128_m32_w4_h16:
        case xmss_aes128_m32_w4_h20:
        case xmss_aes128_m32_w8_h10:
        case xmss_aes128_m32_w16_h10:
        case xmss_aes128_m32_w8_h16:
        case xmss_aes128_m32_w16_h16:
        case xmss_aes128_m32_w8_h20:
        case xmss_aes128_m32_w16_h20:
          bytestring16 root_n16;

        case xmss_sha3-256_m32_w4_h10:
        case xmss_sha3-256_m32_w4_h16:
        case xmss_sha3-256_m32_w4_h20:
        case xmss_sha3-256_m32_w8_h10:
        case xmss_sha3-256_m32_w16_h10:
        case xmss_sha3-256_m32_w8_h16:
        case xmss_sha3-256_m32_w16_h16:
        case xmss_sha3-256_m32_w8_h20:
        case xmss_sha3-256_m32_w16_h20:
          bytestring32 root_n32;

        case xmss_sha3-512_m64_w4_h10:
```

```
        case xmss_sha3-512_m64_w4_h16:
        case xmss_sha3-512_m64_w4_h20:
        case xmss_sha3-512_m64_w8_h10:
        case xmss_sha3-512_m64_w16_h10:
        case xmss_sha3-512_m64_w8_h16:
        case xmss_sha3-512_m64_w16_h16:
        case xmss_sha3-512_m64_w8_h20:
        case xmss_sha3-512_m64_w16_h20:
          bytestring64 root_n64;

        default:
          void;     /* error condition */
      };

      /* XMSS public key structure */

      struct xmss_public_key {
        xmss_bm bm;  /* Bitmasks */
        xmss_root root;  /* Root node */
      };
```

Appendix C. XMSS^MT XDR Formats

XMSS^MT parameter sets are defined using XDR syntax as follows:

```
      /* Byte strings */

      typedef opaque bytestring3[3];
      typedef opaque bytestring5[5];
      typedef opaque bytestring8[8];

      /* Definition of parameter sets */

      enum xmssmt_algorithm_type {
        xmssmt_reserved                 = 0x00000000,

        /* Empty bitmasks */

        /* 128 bit classical security, 85 bit post-quantum security */

        xmssmt_sha3-256_m32_w4_h20_d2_z   = 0x01000001,
        xmssmt_sha3-256_m32_w4_h20_d4_z   = 0x02000002,
        xmssmt_sha3-256_m32_w4_h40_d2_z   = 0x03000003,
        xmssmt_sha3-256_m32_w4_h40_d4_z   = 0x04000004,
        xmssmt_sha3-256_m32_w4_h40_d8_z   = 0x05000005,
        xmssmt_sha3-256_m32_w4_h60_d3_z   = 0x06000006,
```

```
        xmssmt_sha3-256_m32_w4_h60_d6_z   = 0x07000007,
        xmssmt_sha3-256_m32_w4_h60_d12_z  = 0x08000008,

        xmssmt_sha3-256_m32_w8_h20_d2_z   = 0x09000009,
        xmssmt_sha3-256_m32_w8_h20_d4_z   = 0x0a00000a,
        xmssmt_sha3-256_m32_w8_h40_d2_z   = 0x0b00000b,
        xmssmt_sha3-256_m32_w8_h40_d4_z   = 0x0c00000c,
        xmssmt_sha3-256_m32_w8_h40_d8_z   = 0x0d00000d,
        xmssmt_sha3-256_m32_w8_h60_d3_z   = 0x0e00000e,
        xmssmt_sha3-256_m32_w8_h60_d6_z   = 0x0f00000f,
        xmssmt_sha3-256_m32_w8_h60_d12_z  = 0x00010100,

        xmssmt_sha3-256_m32_w16_h20_d2_z  = 0x01010101,
        xmssmt_sha3-256_m32_w16_h20_d4_z  = 0x02010102,
        xmssmt_sha3-256_m32_w16_h40_d2_z  = 0x03010103,
        xmssmt_sha3-256_m32_w16_h40_d4_z  = 0x04010104,
        xmssmt_sha3-256_m32_w16_h40_d8_z  = 0x05010105,
        xmssmt_sha3-256_m32_w16_h60_d3_z  = 0x06010106,
        xmssmt_sha3-256_m32_w16_h60_d6_z  = 0x07010107,
        xmssmt_sha3-256_m32_w16_h60_d12_z = 0x08010108,

        /* 256 bit classical security, 170 bit post-quantum security */

        xmssmt_sha3-512_m64_w4_h20_d2_z   = 0x09010109,
        xmssmt_sha3-512_m64_w4_h20_d4_z   = 0x0a01010a,
        xmssmt_sha3-512_m64_w4_h40_d2_z   = 0x0b01010b,
        xmssmt_sha3-512_m64_w4_h40_d4_z   = 0x0c01010c,
        xmssmt_sha3-512_m64_w4_h40_d8_z   = 0x0d01010d,
        xmssmt_sha3-512_m64_w4_h60_d3_z   = 0x0e01010e,
        xmssmt_sha3-512_m64_w4_h60_d6_z   = 0x0f01010f,
        xmssmt_sha3-512_m64_w4_h60_d12_z  = 0x00020200,

        xmssmt_sha3-512_m64_w8_h20_d2_z   = 0x01020201,
        xmssmt_sha3-512_m64_w8_h20_d4_z   = 0x02020202,
        xmssmt_sha3-512_m64_w8_h40_d2_z   = 0x03020203,
        xmssmt_sha3-512_m64_w8_h40_d4_z   = 0x04020204,
        xmssmt_sha3-512_m64_w8_h40_d8_z   = 0x05020205,
        xmssmt_sha3-512_m64_w8_h60_d3_z   = 0x06020206,
        xmssmt_sha3-512_m64_w8_h60_d6_z   = 0x07020207,
        xmssmt_sha3-512_m64_w8_h60_d12_z  = 0x08020208,

        xmssmt_sha3-512_m64_w16_h20_d2_z  = 0x09020209,
        xmssmt_sha3-512_m64_w16_h20_d4_z  = 0x0a02020a,
        xmssmt_sha3-512_m64_w16_h40_d2_z  = 0x0b02020b,
        xmssmt_sha3-512_m64_w16_h40_d4_z  = 0x0c02020c,
        xmssmt_sha3-512_m64_w16_h40_d8_z  = 0x0d02020d,
        xmssmt_sha3-512_m64_w16_h60_d3_z  = 0x0e02020e,
        xmssmt_sha3-512_m64_w16_h60_d6_z  = 0x0f02020f,
```

```
        xmssmt_sha3-512_m64_w16_h60_d12_z = 0x00030300,

        /* Non-empty bitmasks */

        /* 128 bit classical security, 64 bit post-quantum security */

        xmssmt_aes128_m32_w4_h20_d2         = 0x01030301,
        xmssmt_aes128_m32_w4_h20_d4         = 0x02030302,
        xmssmt_aes128_m32_w4_h40_d2         = 0x03030303,
        xmssmt_aes128_m32_w4_h40_d4         = 0x04030304,
        xmssmt_aes128_m32_w4_h40_d8         = 0x05030305,
        xmssmt_aes128_m32_w4_h60_d3         = 0x06030306,
        xmssmt_aes128_m32_w4_h60_d6         = 0x07030307,
        xmssmt_aes128_m32_w4_h60_d12        = 0x08030308,

        xmssmt_aes128_m32_w8_h20_d2         = 0x09030309,
        xmssmt_aes128_m32_w8_h20_d4         = 0x0a03030a,
        xmssmt_aes128_m32_w8_h40_d2         = 0x0b03030b,
        xmssmt_aes128_m32_w8_h40_d4         = 0x0c03030c,
        xmssmt_aes128_m32_w8_h40_d8         = 0x0d03030d,
        xmssmt_aes128_m32_w8_h60_d3         = 0x0e03030e,
        xmssmt_aes128_m32_w8_h60_d6         = 0x0f03030f,
        xmssmt_aes128_m32_w8_h60_d12        = 0x00040400,

        xmssmt_aes128_m32_w16_h20_d2        = 0x01040401,
        xmssmt_aes128_m32_w16_h20_d4        = 0x02040402,
        xmssmt_aes128_m32_w16_h40_d2        = 0x03040403,
        xmssmt_aes128_m32_w16_h40_d4        = 0x04040404,
        xmssmt_aes128_m32_w16_h40_d8        = 0x05040405,
        xmssmt_aes128_m32_w16_h60_d3        = 0x06040406,
        xmssmt_aes128_m32_w16_h60_d6        = 0x07040407,
        xmssmt_aes128_m32_w16_h60_d12       = 0x08040408,

        /* 256 bit classical security, 128 bit post-quantum security */

        xmssmt_sha3-256_m32_w4_h20_d2       = 0x09040409,
        xmssmt_sha3-256_m32_w4_h20_d4       = 0x0a04040a,
        xmssmt_sha3-256_m32_w4_h40_d2       = 0x0b04040b,
        xmssmt_sha3-256_m32_w4_h40_d4       = 0x0c04040c,
        xmssmt_sha3-256_m32_w4_h40_d8       = 0x0d04040d,
        xmssmt_sha3-256_m32_w4_h60_d3       = 0x0e04040e,
        xmssmt_sha3-256_m32_w4_h60_d6       = 0x0f04040f,
        xmssmt_sha3-256_m32_w4_h60_d12      = 0x00050500,

        xmssmt_sha3-256_m32_w8_h20_d2       = 0x01050501,
        xmssmt_sha3-256_m32_w8_h20_d4       = 0x02050502,
        xmssmt_sha3-256_m32_w8_h40_d2       = 0x03050503,
        xmssmt_sha3-256_m32_w8_h40_d4       = 0x04050504,
```

```
        xmssmt_sha3-256_m32_w8_h40_d8       = 0x05050505,
        xmssmt_sha3-256_m32_w8_h60_d3       = 0x06050506,
        xmssmt_sha3-256_m32_w8_h60_d6       = 0x07050507,
        xmssmt_sha3-256_m32_w8_h60_d12      = 0x08050508,

        xmssmt_sha3-256_m32_w16_h20_d2      = 0x09050509,
        xmssmt_sha3-256_m32_w16_h20_d4      = 0x0a05050a,
        xmssmt_sha3-256_m32_w16_h40_d2      = 0x0b05050b,
        xmssmt_sha3-256_m32_w16_h40_d4      = 0x0c05050c,
        xmssmt_sha3-256_m32_w16_h40_d8      = 0x0d05050d,
        xmssmt_sha3-256_m32_w16_h60_d3      = 0x0e05050e,
        xmssmt_sha3-256_m32_w16_h60_d6      = 0x0f05050f,
        xmssmt_sha3-256_m32_w16_h60_d12     = 0x00060600,

        /* 512 bit classical security, 256 bit post-quantum security */

        xmssmt_sha3-512_m64_w4_h20_d2       = 0x01060601,
        xmssmt_sha3-512_m64_w4_h20_d4       = 0x02060602,
        xmssmt_sha3-512_m64_w4_h40_d2       = 0x03060603,
        xmssmt_sha3-512_m64_w4_h40_d4       = 0x04060604,
        xmssmt_sha3-512_m64_w4_h40_d8       = 0x05060605,
        xmssmt_sha3-512_m64_w4_h60_d3       = 0x06060606,
        xmssmt_sha3-512_m64_w4_h60_d6       = 0x07060607,
        xmssmt_sha3-512_m64_w4_h60_d12      = 0x08060608,

        xmssmt_sha3-512_m64_w8_h20_d2       = 0x09060609,
        xmssmt_sha3-512_m64_w8_h20_d4       = 0x0a06060a,
        xmssmt_sha3-512_m64_w8_h40_d2       = 0x0b06060b,
        xmssmt_sha3-512_m64_w8_h40_d4       = 0x0c06060c,
        xmssmt_sha3-512_m64_w8_h40_d8       = 0x0d06060d,
        xmssmt_sha3-512_m64_w8_h60_d3       = 0x0e06060e,
        xmssmt_sha3-512_m64_w8_h60_d6       = 0x0f06060f,
        xmssmt_sha3-512_m64_w8_h60_d12      = 0x00070700,

        xmssmt_sha3-512_m64_w16_h20_d2      = 0x01070701,
        xmssmt_sha3-512_m64_w16_h20_d4      = 0x02070702,
        xmssmt_sha3-512_m64_w16_h40_d2      = 0x03070703,
        xmssmt_sha3-512_m64_w16_h40_d4      = 0x04070704,
        xmssmt_sha3-512_m64_w16_h40_d8      = 0x05070705,
        xmssmt_sha3-512_m64_w16_h60_d3      = 0x06070706,
        xmssmt_sha3-512_m64_w16_h60_d6      = 0x07070707,
        xmssmt_sha3-512_m64_w16_h60_d12     = 0x08070708,
      };
```

XMSS^MT signatures are defined using XDR syntax as follows:

```
/* Type for XMSS^MT key pair index */
/* Depends solely on h */

union idx_sig_xmssmt switch (xmss_algorithm_type type) {
  case xmssmt_sha3-256_m32_w4_h20_d2_z:
  case xmssmt_sha3-256_m32_w4_h20_d4_z:
  case xmssmt_sha3-256_m32_w8_h20_d2_z:
  case xmssmt_sha3-256_m32_w8_h20_d4_z:
  case xmssmt_sha3-256_m32_w16_h20_d2_z:
  case xmssmt_sha3-256_m32_w16_h20_d4_z:
  case xmssmt_sha3-512_m64_w4_h20_d2_z:
  case xmssmt_sha3-512_m64_w4_h20_d4_z:
  case xmssmt_sha3-512_m64_w8_h20_d2_z:
  case xmssmt_sha3-512_m64_w8_h20_d4_z:
  case xmssmt_sha3-512_m64_w16_h20_d2_z:
  case xmssmt_sha3-512_m64_w16_h20_d4_z:
  case xmssmt_aes128_m32_w4_h20_d2:
  case xmssmt_aes128_m32_w4_h20_d4:
  case xmssmt_aes128_m32_w8_h20_d2:
  case xmssmt_aes128_m32_w8_h20_d4:
  case xmssmt_aes128_m32_w16_h20_d2:
  case xmssmt_aes128_m32_w16_h20_d4:
  case xmssmt_sha3-256_m32_w4_h20_d2:
  case xmssmt_sha3-256_m32_w4_h20_d4:
  case xmssmt_sha3-256_m32_w8_h20_d2:
  case xmssmt_sha3-256_m32_w8_h20_d4:
  case xmssmt_sha3-256_m32_w16_h20_d2:
  case xmssmt_sha3-256_m32_w16_h20_d4:
  case xmssmt_sha3-512_m64_w4_h20_d2:
  case xmssmt_sha3-512_m64_w4_h20_d4:
  case xmssmt_sha3-512_m64_w8_h20_d2:
  case xmssmt_sha3-512_m64_w8_h20_d4:
  case xmssmt_sha3-512_m64_w16_h20_d2:
  case xmssmt_sha3-512_m64_w16_h20_d4:
    bytestring3 idx3;

  case xmssmt_sha3-256_m32_w4_h40_d2_z:
  case xmssmt_sha3-256_m32_w4_h40_d4_z:
  case xmssmt_sha3-256_m32_w4_h40_d8_z:
  case xmssmt_sha3-256_m32_w8_h40_d2_z:
  case xmssmt_sha3-256_m32_w8_h40_d4_z:
  case xmssmt_sha3-256_m32_w8_h40_d8_z:
  case xmssmt_sha3-256_m32_w16_h40_d2_z:
  case xmssmt_sha3-256_m32_w16_h40_d4_z:
  case xmssmt_sha3-256_m32_w16_h40_d8_z:
  case xmssmt_sha3-512_m64_w4_h40_d2_z:
  case xmssmt_sha3-512_m64_w4_h40_d4_z:
  case xmssmt_sha3-512_m64_w4_h40_d8_z:
```

```
        case xmssmt_sha3-512_m64_w8_h40_d2_z:
        case xmssmt_sha3-512_m64_w8_h40_d4_z:
        case xmssmt_sha3-512_m64_w8_h40_d8_z:
        case xmssmt_sha3-512_m64_w16_h40_d2_z:
        case xmssmt_sha3-512_m64_w16_h40_d4_z:
        case xmssmt_sha3-512_m64_w16_h40_d8_z:
        case xmssmt_aes128_m32_w4_h40_d2:
        case xmssmt_aes128_m32_w4_h40_d4:
        case xmssmt_aes128_m32_w4_h40_d8:
        case xmssmt_aes128_m32_w8_h40_d2:
        case xmssmt_aes128_m32_w8_h40_d4:
        case xmssmt_aes128_m32_w8_h40_d8:
        case xmssmt_aes128_m32_w16_h40_d2:
        case xmssmt_aes128_m32_w16_h40_d4:
        case xmssmt_aes128_m32_w16_h40_d8:
        case xmssmt_sha3-256_m32_w4_h40_d2:
        case xmssmt_sha3-256_m32_w4_h40_d4:
        case xmssmt_sha3-256_m32_w4_h40_d8:
        case xmssmt_sha3-256_m32_w8_h40_d2:
        case xmssmt_sha3-256_m32_w8_h40_d4:
        case xmssmt_sha3-256_m32_w8_h40_d8:
        case xmssmt_sha3-512_m64_w4_h40_d2:
        case xmssmt_sha3-512_m64_w4_h40_d4:
        case xmssmt_sha3-512_m64_w4_h40_d8:
        case xmssmt_sha3-256_m32_w16_h40_d2:
        case xmssmt_sha3-256_m32_w16_h40_d4:
        case xmssmt_sha3-256_m32_w16_h40_d8:
        case xmssmt_sha3-512_m64_w8_h40_d2:
        case xmssmt_sha3-512_m64_w8_h40_d4:
        case xmssmt_sha3-512_m64_w8_h40_d8:
        case xmssmt_sha3-512_m64_w16_h40_d2:
        case xmssmt_sha3-512_m64_w16_h40_d4:
        case xmssmt_sha3-512_m64_w16_h40_d8:
          bytestring5 idx5;

        case xmssmt_sha3-256_m32_w4_h60_d3_z:
        case xmssmt_sha3-256_m32_w4_h60_d6_z:
        case xmssmt_sha3-256_m32_w4_h60_d12_z:
        case xmssmt_sha3-256_m32_w8_h60_d3_z:
        case xmssmt_sha3-256_m32_w8_h60_d6_z:
        case xmssmt_sha3-256_m32_w8_h60_d12_z:
        case xmssmt_sha3-256_m32_w16_h60_d3_z:
        case xmssmt_sha3-256_m32_w16_h60_d6_z:
        case xmssmt_sha3-256_m32_w16_h60_d12_z:
        case xmssmt_sha3-512_m64_w4_h60_d3_z:
        case xmssmt_sha3-512_m64_w4_h60_d6_z:
        case xmssmt_sha3-512_m64_w4_h60_d12_z:
        case xmssmt_sha3-512_m64_w8_h60_d3_z:
```

```
        case xmssmt_sha3-512_m64_w8_h60_d6_z:
        case xmssmt_sha3-512_m64_w8_h60_d12_z:
        case xmssmt_sha3-512_m64_w16_h60_d3_z:
        case xmssmt_sha3-512_m64_w16_h60_d6_z:
        case xmssmt_sha3-512_m64_w16_h60_d12_z:
        case xmssmt_aes128_m32_w4_h60_d3:
        case xmssmt_aes128_m32_w4_h60_d6:
        case xmssmt_aes128_m32_w4_h60_d12:
        case xmssmt_aes128_m32_w8_h60_d3:
        case xmssmt_aes128_m32_w8_h60_d6:
        case xmssmt_aes128_m32_w8_h60_d12:
        case xmssmt_aes128_m32_w16_h60_d3:
        case xmssmt_aes128_m32_w16_h60_d6:
        case xmssmt_aes128_m32_w16_h60_d12:
        case xmssmt_sha3-256_m32_w4_h60_d3:
        case xmssmt_sha3-256_m32_w4_h60_d6:
        case xmssmt_sha3-256_m32_w4_h60_d12:
        case xmssmt_sha3-256_m32_w8_h60_d3:
        case xmssmt_sha3-256_m32_w8_h60_d6:
        case xmssmt_sha3-256_m32_w8_h60_d12:
        case xmssmt_sha3-256_m32_w16_h60_d3:
        case xmssmt_sha3-256_m32_w16_h60_d6:
        case xmssmt_sha3-256_m32_w16_h60_d12:
        case xmssmt_sha3-512_m64_w4_h60_d3:
        case xmssmt_sha3-512_m64_w4_h60_d6:
        case xmssmt_sha3-512_m64_w4_h60_d12:
        case xmssmt_sha3-512_m64_w8_h60_d3:
        case xmssmt_sha3-512_m64_w8_h60_d6:
        case xmssmt_sha3-512_m64_w8_h60_d12:
        case xmssmt_sha3-512_m64_w16_h60_d3:
        case xmssmt_sha3-512_m64_w16_h60_d6:
        case xmssmt_sha3-512_m64_w16_h60_d12:
          bytestring8 idx8;

        default:
          void;     /* error condition */
      };

      union random_string_xmssmt switch (xmssmt_algorithm_type type) {
        case xmssmt_aes128_m32_w4_h20_d2:
        case xmssmt_aes128_m32_w4_h20_d4:
        case xmssmt_aes128_m32_w4_h40_d2:
        case xmssmt_aes128_m32_w4_h40_d4:
        case xmssmt_aes128_m32_w4_h40_d8:
        case xmssmt_aes128_m32_w4_h60_d3:
        case xmssmt_aes128_m32_w4_h60_d6:
        case xmssmt_aes128_m32_w4_h60_d12:
        case xmssmt_aes128_m32_w8_h20_d2:
```

```
        case xmssmt_aes128_m32_w8_h20_d4:
        case xmssmt_aes128_m32_w8_h40_d2:
        case xmssmt_aes128_m32_w8_h40_d4:
        case xmssmt_aes128_m32_w8_h40_d8:
        case xmssmt_aes128_m32_w8_h60_d3:
        case xmssmt_aes128_m32_w8_h60_d6:
        case xmssmt_aes128_m32_w8_h60_d12:
        case xmssmt_aes128_m32_w16_h20_d2:
        case xmssmt_aes128_m32_w16_h20_d4:
        case xmssmt_aes128_m32_w16_h40_d2:
        case xmssmt_aes128_m32_w16_h40_d4:
        case xmssmt_aes128_m32_w16_h40_d8:
        case xmssmt_aes128_m32_w16_h60_d3:
        case xmssmt_aes128_m32_w16_h60_d6:
        case xmssmt_aes128_m32_w16_h60_d12:
        case xmssmt_sha3-256_m32_w4_h20_d2_z:
        case xmssmt_sha3-256_m32_w4_h20_d4_z:
        case xmssmt_sha3-256_m32_w4_h40_d2_z:
        case xmssmt_sha3-256_m32_w4_h40_d4_z:
        case xmssmt_sha3-256_m32_w4_h40_d8_z:
        case xmssmt_sha3-256_m32_w4_h60_d3_z:
        case xmssmt_sha3-256_m32_w4_h60_d6_z:
        case xmssmt_sha3-256_m32_w4_h60_d12_z:
        case xmssmt_sha3-256_m32_w8_h20_d2_z:
        case xmssmt_sha3-256_m32_w8_h20_d4_z:
        case xmssmt_sha3-256_m32_w8_h40_d2_z:
        case xmssmt_sha3-256_m32_w8_h40_d4_z:
        case xmssmt_sha3-256_m32_w8_h40_d8_z:
        case xmssmt_sha3-256_m32_w8_h60_d3_z:
        case xmssmt_sha3-256_m32_w8_h60_d6_z:
        case xmssmt_sha3-256_m32_w8_h60_d12_z:
        case xmssmt_sha3-256_m32_w16_h20_d2_z:
        case xmssmt_sha3-256_m32_w16_h20_d4_z:
        case xmssmt_sha3-256_m32_w16_h40_d2_z:
        case xmssmt_sha3-256_m32_w16_h40_d4_z:
        case xmssmt_sha3-256_m32_w16_h40_d8_z:
        case xmssmt_sha3-256_m32_w16_h60_d3_z:
        case xmssmt_sha3-256_m32_w16_h60_d6_z:
        case xmssmt_sha3-256_m32_w16_h60_d12_z:
        case xmssmt_sha3-256_m32_w4_h20_d2:
        case xmssmt_sha3-256_m32_w4_h20_d4:
        case xmssmt_sha3-256_m32_w4_h40_d2:
        case xmssmt_sha3-256_m32_w4_h40_d4:
        case xmssmt_sha3-256_m32_w4_h40_d8:
        case xmssmt_sha3-256_m32_w4_h60_d3:
        case xmssmt_sha3-256_m32_w4_h60_d6:
        case xmssmt_sha3-256_m32_w4_h60_d12:
        case xmssmt_sha3-256_m32_w8_h20_d2:
```

```
        case xmssmt_sha3-256_m32_w8_h20_d4:
        case xmssmt_sha3-256_m32_w8_h40_d2:
        case xmssmt_sha3-256_m32_w8_h40_d4:
        case xmssmt_sha3-256_m32_w8_h40_d8:
        case xmssmt_sha3-256_m32_w8_h60_d3:
        case xmssmt_sha3-256_m32_w8_h60_d6:
        case xmssmt_sha3-256_m32_w8_h60_d12:
        case xmssmt_sha3-256_m32_w16_h20_d2:
        case xmssmt_sha3-256_m32_w16_h20_d4:
        case xmssmt_sha3-256_m32_w16_h40_d2:
        case xmssmt_sha3-256_m32_w16_h40_d4:
        case xmssmt_sha3-256_m32_w16_h40_d8:
        case xmssmt_sha3-256_m32_w16_h60_d3:
        case xmssmt_sha3-256_m32_w16_h60_d6:
        case xmssmt_sha3-256_m32_w16_h60_d12:
          bytestring32 rand_m32;

        case xmssmt_sha3-512_m64_w4_h20_d2_z:
        case xmssmt_sha3-512_m64_w4_h20_d4_z:
        case xmssmt_sha3-512_m64_w4_h40_d2_z:
        case xmssmt_sha3-512_m64_w4_h40_d4_z:
        case xmssmt_sha3-512_m64_w4_h40_d8_z:
        case xmssmt_sha3-512_m64_w4_h60_d3_z:
        case xmssmt_sha3-512_m64_w4_h60_d6_z:
        case xmssmt_sha3-512_m64_w4_h60_d12_z:
        case xmssmt_sha3-512_m64_w8_h20_d2_z:
        case xmssmt_sha3-512_m64_w8_h20_d4_z:
        case xmssmt_sha3-512_m64_w8_h40_d2_z:
        case xmssmt_sha3-512_m64_w8_h40_d4_z:
        case xmssmt_sha3-512_m64_w8_h40_d8_z:
        case xmssmt_sha3-512_m64_w8_h60_d3_z:
        case xmssmt_sha3-512_m64_w8_h60_d6_z:
        case xmssmt_sha3-512_m64_w8_h60_d12_z:
        case xmssmt_sha3-512_m64_w16_h20_d2_z:
        case xmssmt_sha3-512_m64_w16_h20_d4_z:
        case xmssmt_sha3-512_m64_w16_h40_d2_z:
        case xmssmt_sha3-512_m64_w16_h40_d4_z:
        case xmssmt_sha3-512_m64_w16_h40_d8_z:
        case xmssmt_sha3-512_m64_w16_h60_d3_z:
        case xmssmt_sha3-512_m64_w16_h60_d6_z:
        case xmssmt_sha3-512_m64_w16_h60_d12_z:
        case xmssmt_sha3-512_m64_w4_h20_d2:
        case xmssmt_sha3-512_m64_w4_h20_d4:
        case xmssmt_sha3-512_m64_w4_h40_d2:
        case xmssmt_sha3-512_m64_w4_h40_d4:
        case xmssmt_sha3-512_m64_w4_h40_d8:
        case xmssmt_sha3-512_m64_w4_h60_d3:
        case xmssmt_sha3-512_m64_w4_h60_d6:
```

```
        case xmssmt_sha3-512_m64_w4_h60_d12:
        case xmssmt_sha3-512_m64_w8_h20_d2:
        case xmssmt_sha3-512_m64_w8_h20_d4:
        case xmssmt_sha3-512_m64_w8_h40_d2:
        case xmssmt_sha3-512_m64_w8_h40_d4:
        case xmssmt_sha3-512_m64_w8_h40_d8:
        case xmssmt_sha3-512_m64_w8_h60_d3:
        case xmssmt_sha3-512_m64_w8_h60_d6:
        case xmssmt_sha3-512_m64_w8_h60_d12:
        case xmssmt_sha3-512_m64_w16_h20_d2:
        case xmssmt_sha3-512_m64_w16_h20_d4:
        case xmssmt_sha3-512_m64_w16_h40_d2:
        case xmssmt_sha3-512_m64_w16_h40_d4:
        case xmssmt_sha3-512_m64_w16_h40_d8:
        case xmssmt_sha3-512_m64_w16_h60_d3:
        case xmssmt_sha3-512_m64_w16_h60_d6:
        case xmssmt_sha3-512_m64_w16_h60_d12:
          bytestring64 rand_m64;

        default:
          void;     /* error condition */
      };

      struct xmss_reduced_bottom {
        xmss_ots_signature sig_ots; /* WOTS+ signature */
        xmss_path nodes;            /* authentication path */
      };

      /* Type for individual reduced XMSS signatures on higher layers */

      union xmss_reduced_others (xmss_algorithm_type type) {
        case xmssmt_aes128_m32_w4_h20_d2:
        case xmssmt_aes128_m32_w4_h20_d4:
          bytestring16 xmss_reduced_n16_t88[88];

        case xmssmt_aes128_m32_w4_h40_d2:
        case xmssmt_aes128_m32_w4_h40_d4:
        case xmssmt_aes128_m32_w4_h40_d8:
          bytestring16 xmss_reduced_n16_t108[108];

        case xmssmt_aes128_m32_w4_h60_d3:
        case xmssmt_aes128_m32_w4_h60_d6:
        case xmssmt_aes128_m32_w4_h60_d12:
          bytestring16 xmss_reduced_n16_t128[128];

        case xmssmt_aes128_m32_w8_h20_d2:
        case xmssmt_aes128_m32_w8_h20_d4:
          bytestring16 xmss_reduced_n16_t66[66];
```

```
        case xmssmt_aes128_m32_w8_h40_d2:
        case xmssmt_aes128_m32_w8_h40_d4:
        case xmssmt_aes128_m32_w8_h40_d8:
          bytestring16 xmss_reduced_n16_t86[86];

        case xmssmt_aes128_m32_w8_h60_d3:
        case xmssmt_aes128_m32_w8_h60_d6:
        case xmssmt_aes128_m32_w8_h60_d12:
          bytestring16 xmss_reduced_n16_t106[106];

        case xmssmt_aes128_m32_w16_h20_d2:
        case xmssmt_aes128_m32_w16_h20_d4:
          bytestring16 xmss_reduced_n16_t55[55];

        case xmssmt_aes128_m32_w16_h40_d2:
        case xmssmt_aes128_m32_w16_h40_d4:
        case xmssmt_aes128_m32_w16_h40_d8:
          bytestring16 xmss_reduced_n16_t75[75];

        case xmssmt_aes128_m32_w16_h60_d3:
        case xmssmt_aes128_m32_w16_h60_d6:
        case xmssmt_aes128_m32_w16_h60_d12:
          bytestring16 xmss_reduced_n16_t95[95];

        case xmssmt_sha3-256_m32_w4_h20_d2_z:
        case xmssmt_sha3-256_m32_w4_h20_d4_z:
        case xmssmt_sha3-256_m32_w4_h20_d2:
        case xmssmt_sha3-256_m32_w4_h20_d4:
          bytestring32 xmss_reduced_n32_t153[153];

        case xmssmt_sha3-256_m32_w4_h40_d2_z:
        case xmssmt_sha3-256_m32_w4_h40_d4_z:
        case xmssmt_sha3-256_m32_w4_h40_d8_z:
        case xmssmt_sha3-256_m32_w4_h40_d2:
        case xmssmt_sha3-256_m32_w4_h40_d4:
        case xmssmt_sha3-256_m32_w4_h40_d8:
          bytestring32 xmss_reduced_n32_t173[173];

        case xmssmt_sha3-256_m32_w4_h60_d3_z:
        case xmssmt_sha3-256_m32_w4_h60_d6_z:
        case xmssmt_sha3-256_m32_w4_h60_d12_z:
        case xmssmt_sha3-256_m32_w4_h60_d3:
        case xmssmt_sha3-256_m32_w4_h60_d6:
        case xmssmt_sha3-256_m32_w4_h60_d12:
          bytestring32 xmss_reduced_n32_t193[193];

        case xmssmt_sha3-256_m32_w8_h20_d2_z:
        case xmssmt_sha3-256_m32_w8_h20_d4_z:
```

```
        case xmssmt_sha3-256_m32_w8_h20_d2:
        case xmssmt_sha3-256_m32_w8_h20_d4:
          bytestring32 xmss_reduced_n32_t110[110];

        case xmssmt_sha3-256_m32_w8_h40_d2_z:
        case xmssmt_sha3-256_m32_w8_h40_d4_z:
        case xmssmt_sha3-256_m32_w8_h40_d8_z:
        case xmssmt_sha3-256_m32_w8_h40_d2:
        case xmssmt_sha3-256_m32_w8_h40_d4:
        case xmssmt_sha3-256_m32_w8_h40_d8:
          bytestring32 xmss_reduced_n32_t130[130];

        case xmssmt_sha3-256_m32_w8_h60_d3_z:
        case xmssmt_sha3-256_m32_w8_h60_d6_z:
        case xmssmt_sha3-256_m32_w8_h60_d12_z:
        case xmssmt_sha3-256_m32_w8_h60_d3:
        case xmssmt_sha3-256_m32_w8_h60_d6:
        case xmssmt_sha3-256_m32_w8_h60_d12:
          bytestring32 xmss_reduced_n32_t150[150];

        case xmssmt_sha3-256_m32_w16_h20_d2_z:
        case xmssmt_sha3-256_m32_w16_h20_d4_z:
        case xmssmt_sha3-256_m32_w16_h20_d2:
        case xmssmt_sha3-256_m32_w16_h20_d4:
          bytestring32 xmss_reduced_n32_t87[87];

        case xmssmt_sha3-256_m32_w16_h40_d2_z:
        case xmssmt_sha3-256_m32_w16_h40_d4_z:
        case xmssmt_sha3-256_m32_w16_h40_d8_z:
        case xmssmt_sha3-256_m32_w16_h40_d2:
        case xmssmt_sha3-256_m32_w16_h40_d4:
        case xmssmt_sha3-256_m32_w16_h40_d8:
          bytestring32 xmss_reduced_n32_t107[107];

        case xmssmt_sha3-256_m32_w16_h60_d3_z:
        case xmssmt_sha3-256_m32_w16_h60_d6_z:
        case xmssmt_sha3-256_m32_w16_h60_d12_z:
        case xmssmt_sha3-256_m32_w16_h60_d3:
        case xmssmt_sha3-256_m32_w16_h60_d6:
        case xmssmt_sha3-256_m32_w16_h60_d12:
          bytestring32 xmss_reduced_n32_t127[127];

        case xmssmt_sha3-512_m64_w4_h20_d2_z:
        case xmssmt_sha3-512_m64_w4_h20_d4_z:
        case xmssmt_sha3-512_m64_w4_h20_d2:
        case xmssmt_sha3-512_m64_w4_h20_d4:
          bytestring64 xmss_reduced_n64_t281[281];
```

```
        case xmssmt_sha3-512_m64_w4_h40_d2_z:
        case xmssmt_sha3-512_m64_w4_h40_d4_z:
        case xmssmt_sha3-512_m64_w4_h40_d8_z:
        case xmssmt_sha3-512_m64_w4_h40_d2:
        case xmssmt_sha3-512_m64_w4_h40_d4:
        case xmssmt_sha3-512_m64_w4_h40_d8:
          bytestring64 xmss_reduced_n64_t301[301];

        case xmssmt_sha3-512_m64_w4_h60_d3_z:
        case xmssmt_sha3-512_m64_w4_h60_d6_z:
        case xmssmt_sha3-512_m64_w4_h60_d12_z:
        case xmssmt_sha3-512_m64_w4_h60_d3:
        case xmssmt_sha3-512_m64_w4_h60_d6:
        case xmssmt_sha3-512_m64_w4_h60_d12:
          bytestring64 xmss_reduced_n64_t321[321];

        case xmssmt_sha3-512_m64_w8_h20_d2_z:
        case xmssmt_sha3-512_m64_w8_h20_d4_z:
          bytestring64 xmss_reduced_n64_t195[195];

        case xmssmt_sha3-512_m64_w8_h40_d2_z:
        case xmssmt_sha3-512_m64_w8_h40_d4_z:
        case xmssmt_sha3-512_m64_w8_h40_d8_z:
        case xmssmt_sha3-512_m64_w8_h40_d2:
        case xmssmt_sha3-512_m64_w8_h40_d4:
        case xmssmt_sha3-512_m64_w8_h40_d8:
          bytestring64 xmss_reduced_n64_t215[215];

        case xmssmt_sha3-512_m64_w8_h60_d3_z:
        case xmssmt_sha3-512_m64_w8_h60_d6_z:
        case xmssmt_sha3-512_m64_w8_h60_d12_z:
        case xmssmt_sha3-512_m64_w8_h60_d3:
        case xmssmt_sha3-512_m64_w8_h60_d6:
        case xmssmt_sha3-512_m64_w8_h60_d12:
          bytestring64 xmss_reduced_n64_t235[235];

        case xmssmt_sha3-512_m64_w16_h20_d2_z:
        case xmssmt_sha3-512_m64_w16_h20_d4_z:
        case xmssmt_sha3-512_m64_w16_h20_d2:
        case xmssmt_sha3-512_m64_w16_h20_d4:
          bytestring64 xmss_reduced_n64_t151[151];

        case xmssmt_sha3-512_m64_w16_h40_d2_z:
        case xmssmt_sha3-512_m64_w16_h40_d4_z:
        case xmssmt_sha3-512_m64_w16_h40_d8_z:
        case xmssmt_sha3-512_m64_w16_h40_d2:
        case xmssmt_sha3-512_m64_w16_h40_d4:
        case xmssmt_sha3-512_m64_w16_h40_d8:
```

```
          bytestring64 xmss_reduced_n64_t171[171];

        case xmssmt_sha3-512_m64_w16_h60_d3_z:
        case xmssmt_sha3-512_m64_w16_h60_d6_z:
        case xmssmt_sha3-512_m64_w16_h60_d12_z:
        case xmssmt_sha3-512_m64_w16_h60_d3:
        case xmssmt_sha3-512_m64_w16_h60_d6:
        case xmssmt_sha3-512_m64_w16_h60_d12:
          bytestring64 xmss_reduced_n64_t191[191];

        default:
          void;     /* error condition */
      };

      /* xmss_reduced_array depends on d */

      union xmss_reduced_array (xmss_algorithm_type type) {
        case xmssmt_sha3-256_m32_w4_h20_d2_z:
        case xmssmt_sha3-256_m32_w8_h20_d2_z:
        case xmssmt_sha3-256_m32_w16_h20_d2_z:
        case xmssmt_sha3-512_m64_w4_h20_d2_z:
        case xmssmt_sha3-512_m64_w8_h20_d2_z:
        case xmssmt_sha3-512_m64_w16_h20_d2_z:
        case xmssmt_aes128_m32_w4_h20_d2:
        case xmssmt_aes128_m32_w8_h20_d2:
        case xmssmt_aes128_m32_w16_h20_d2:
        case xmssmt_sha3-256_m32_w4_h20_d2:
        case xmssmt_sha3-256_m32_w8_h20_d2:
        case xmssmt_sha3-256_m32_w16_h20_d2:
        case xmssmt_sha3-512_m64_w4_h20_d2:
        case xmssmt_sha3-512_m64_w8_h20_d2:
        case xmssmt_sha3-512_m64_w16_h20_d2:
        case xmssmt_sha3-256_m32_w4_h40_d2_z:
        case xmssmt_sha3-256_m32_w8_h40_d2_z:
        case xmssmt_sha3-256_m32_w16_h40_d2_z:
        case xmssmt_sha3-512_m64_w4_h40_d2_z:
        case xmssmt_sha3-512_m64_w8_h40_d2_z:
        case xmssmt_sha3-512_m64_w16_h40_d2_z:
        case xmssmt_aes128_m32_w4_h40_d2:
        case xmssmt_aes128_m32_w8_h40_d2:
        case xmssmt_aes128_m32_w16_h40_d2:
        case xmssmt_sha3-256_m32_w4_h40_d2:
        case xmssmt_sha3-256_m32_w8_h40_d2:
        case xmssmt_sha3-512_m64_w4_h40_d2:
        case xmssmt_sha3-256_m32_w16_h40_d2:
        case xmssmt_sha3-512_m64_w8_h40_d2:
        case xmssmt_sha3-512_m64_w16_h40_d2:
          xmss_reduced_others xmss_red_arr_d2[1];
```

```
        case xmssmt_sha3-256_m32_w4_h60_d3_z:
        case xmssmt_sha3-256_m32_w8_h60_d3_z:
        case xmssmt_sha3-256_m32_w16_h60_d3_z:
        case xmssmt_sha3-512_m64_w4_h60_d3_z:
        case xmssmt_sha3-512_m64_w8_h60_d3_z:
        case xmssmt_sha3-512_m64_w16_h60_d3_z:
        case xmssmt_aes128_m32_w4_h60_d3:
        case xmssmt_aes128_m32_w8_h60_d3:
        case xmssmt_aes128_m32_w16_h60_d3:
        case xmssmt_sha3-256_m32_w4_h60_d3:
        case xmssmt_sha3-256_m32_w8_h60_d3:
        case xmssmt_sha3-256_m32_w16_h60_d3:
        case xmssmt_sha3-512_m64_w4_h60_d3:
        case xmssmt_sha3-512_m64_w8_h60_d3:
        case xmssmt_sha3-512_m64_w16_h60_d3:
          xmss_reduced_others xmss_red_arr_d3[2];

        case xmssmt_sha3-256_m32_w4_h20_d4_z:
        case xmssmt_sha3-256_m32_w8_h20_d4_z:
        case xmssmt_sha3-256_m32_w16_h20_d4_z:
        case xmssmt_sha3-512_m64_w4_h20_d4_z:
        case xmssmt_sha3-512_m64_w8_h20_d4_z:
        case xmssmt_sha3-512_m64_w16_h20_d4_z:
        case xmssmt_aes128_m32_w4_h20_d4:
        case xmssmt_aes128_m32_w8_h20_d4:
        case xmssmt_aes128_m32_w16_h20_d4:
        case xmssmt_sha3-256_m32_w4_h20_d4:
        case xmssmt_sha3-256_m32_w8_h20_d4:
        case xmssmt_sha3-256_m32_w16_h20_d4:
        case xmssmt_sha3-512_m64_w4_h20_d4:
        case xmssmt_sha3-512_m64_w8_h20_d4:
        case xmssmt_sha3-512_m64_w16_h20_d4:
        case xmssmt_sha3-256_m32_w4_h40_d4_z:
        case xmssmt_sha3-256_m32_w8_h40_d4_z:
        case xmssmt_sha3-256_m32_w16_h40_d4_z:
        case xmssmt_sha3-512_m64_w4_h40_d4_z:
        case xmssmt_sha3-512_m64_w8_h40_d4_z:
        case xmssmt_sha3-512_m64_w16_h40_d4_z:
        case xmssmt_aes128_m32_w4_h40_d4:
        case xmssmt_aes128_m32_w8_h40_d4:
        case xmssmt_aes128_m32_w16_h40_d4:
        case xmssmt_sha3-256_m32_w4_h40_d4:
        case xmssmt_sha3-256_m32_w8_h40_d4:
        case xmssmt_sha3-512_m64_w4_h40_d4:
        case xmssmt_sha3-256_m32_w16_h40_d4:
        case xmssmt_sha3-512_m64_w8_h40_d4:
        case xmssmt_sha3-512_m64_w16_h40_d4:
          xmss_reduced_others xmss_red_arr_d4[3];
```

```
    case xmssmt_sha3-256_m32_w4_h60_d6_z:
    case xmssmt_sha3-256_m32_w8_h60_d6_z:
    case xmssmt_sha3-256_m32_w16_h60_d6_z:
    case xmssmt_sha3-512_m64_w4_h60_d6_z:
    case xmssmt_sha3-512_m64_w8_h60_d6_z:
    case xmssmt_sha3-512_m64_w16_h60_d6_z:
    case xmssmt_aes128_m32_w4_h60_d6:
    case xmssmt_aes128_m32_w8_h60_d6:
    case xmssmt_aes128_m32_w16_h60_d6:
    case xmssmt_sha3-256_m32_w4_h60_d6:
    case xmssmt_sha3-256_m32_w8_h60_d6:
    case xmssmt_sha3-256_m32_w16_h60_d6:
    case xmssmt_sha3-512_m64_w4_h60_d6:
    case xmssmt_sha3-512_m64_w8_h60_d6:
    case xmssmt_sha3-512_m64_w16_h60_d6:
      xmss_reduced_others xmss_red_arr_d6[5];

    case xmssmt_sha3-256_m32_w4_h40_d8_z:
    case xmssmt_sha3-256_m32_w8_h40_d8_z:
    case xmssmt_sha3-256_m32_w16_h40_d8_z:
    case xmssmt_sha3-512_m64_w4_h40_d8_z:
    case xmssmt_sha3-512_m64_w8_h40_d8_z:
    case xmssmt_sha3-512_m64_w16_h40_d8_z:
    case xmssmt_aes128_m32_w4_h40_d8:
    case xmssmt_aes128_m32_w8_h40_d8:
    case xmssmt_aes128_m32_w16_h40_d8:
    case xmssmt_sha3-256_m32_w4_h40_d8:
    case xmssmt_sha3-256_m32_w8_h40_d8:
    case xmssmt_sha3-512_m64_w4_h40_d8:
    case xmssmt_sha3-256_m32_w16_h40_d8:
    case xmssmt_sha3-512_m64_w8_h40_d8:
    case xmssmt_sha3-512_m64_w16_h40_d8:
      xmss_reduced_others xmss_red_arr_d8[7];

    case xmssmt_sha3-256_m32_w4_h60_d12_z:
    case xmssmt_sha3-256_m32_w8_h60_d12_z:
    case xmssmt_sha3-256_m32_w16_h60_d12_z:
    case xmssmt_sha3-512_m64_w4_h60_d12_z:
    case xmssmt_sha3-512_m64_w8_h60_d12_z:
    case xmssmt_sha3-512_m64_w16_h60_d12_z:
    case xmssmt_aes128_m32_w4_h60_d12:
    case xmssmt_aes128_m32_w8_h60_d12:
    case xmssmt_aes128_m32_w16_h60_d12:
    case xmssmt_sha3-256_m32_w4_h60_d12:
    case xmssmt_sha3-256_m32_w8_h60_d12:
    case xmssmt_sha3-256_m32_w16_h60_d12:
    case xmssmt_sha3-512_m64_w4_h60_d12:
    case xmssmt_sha3-512_m64_w8_h60_d12:
```

```
        case xmssmt_sha3-512_m64_w16_h60_d12:
          xmss_reduced_others xmss_red_arr_d12[11];

        default:
          void;     /* error condition */
      };

      /* XMSS^MT signature structure */

      struct xmssmt_signature {
        /* WOTS+ key pair index */
        idx_sig_xmssmt idx_sig;
        /* Random string for randomized hashing */
        random_string_xmssmt randomness;
        /* Reduced bottom layer XMSS signature */
        xmss_reduced_bottom;
        /* Array of reduced XMSS signatures with message length n */
        xmss_reduced_array;
      };
```

When no bitmasks are used, XMSS^MT public keys are defined using XDR syntax as follows:

```
      /* Types for XMSS^MT root node */

      union xmssmt_root switch (xmssmt_algorithm_type type) {
        case xmssmt_sha3-256_m32_w4_h20_d2_z:
        case xmssmt_sha3-256_m32_w4_h20_d4_z:
        case xmssmt_sha3-256_m32_w4_h40_d2_z:
        case xmssmt_sha3-256_m32_w4_h40_d4_z:
        case xmssmt_sha3-256_m32_w4_h40_d8_z:
        case xmssmt_sha3-256_m32_w4_h60_d3_z:
        case xmssmt_sha3-256_m32_w4_h60_d6_z:
        case xmssmt_sha3-256_m32_w4_h60_d12_z:
        case xmssmt_sha3-256_m32_w8_h20_d2_z:
        case xmssmt_sha3-256_m32_w8_h20_d4_z:
        case xmssmt_sha3-256_m32_w8_h40_d2_z:
        case xmssmt_sha3-256_m32_w8_h40_d4_z:
        case xmssmt_sha3-256_m32_w8_h40_d8_z:
        case xmssmt_sha3-256_m32_w8_h60_d3_z:
        case xmssmt_sha3-256_m32_w8_h60_d6_z:
        case xmssmt_sha3-256_m32_w8_h60_d12_z:
        case xmssmt_sha3-256_m32_w16_h20_d2_z:
        case xmssmt_sha3-256_m32_w16_h20_d4_z:
        case xmssmt_sha3-256_m32_w16_h40_d2_z:
        case xmssmt_sha3-256_m32_w16_h40_d4_z:
```

```
      case xmssmt_sha3-256_m32_w16_h40_d8_z:
      case xmssmt_sha3-256_m32_w16_h60_d3_z:
      case xmssmt_sha3-256_m32_w16_h60_d6_z:
      case xmssmt_sha3-256_m32_w16_h60_d12_z:
        bytestring32 root_n32;

      case xmssmt_sha3-512_m64_w4_h20_d2_z:
      case xmssmt_sha3-512_m64_w4_h20_d4_z:
      case xmssmt_sha3-512_m64_w4_h40_d2_z:
      case xmssmt_sha3-512_m64_w4_h40_d4_z:
      case xmssmt_sha3-512_m64_w4_h40_d8_z:
      case xmssmt_sha3-512_m64_w4_h60_d3_z:
      case xmssmt_sha3-512_m64_w4_h60_d6_z:
      case xmssmt_sha3-512_m64_w4_h60_d12_z:
      case xmssmt_sha3-512_m64_w8_h20_d2_z:
      case xmssmt_sha3-512_m64_w8_h20_d4_z:
      case xmssmt_sha3-512_m64_w8_h40_d2_z:
      case xmssmt_sha3-512_m64_w8_h40_d4_z:
      case xmssmt_sha3-512_m64_w8_h40_d8_z:
      case xmssmt_sha3-512_m64_w8_h60_d3_z:
      case xmssmt_sha3-512_m64_w8_h60_d6_z:
      case xmssmt_sha3-512_m64_w8_h60_d12_z:
      case xmssmt_sha3-512_m64_w16_h20_d2_z:
      case xmssmt_sha3-512_m64_w16_h20_d4_z:
      case xmssmt_sha3-512_m64_w16_h40_d2_z:
      case xmssmt_sha3-512_m64_w16_h40_d4_z:
      case xmssmt_sha3-512_m64_w16_h40_d8_z:
      case xmssmt_sha3-512_m64_w16_h60_d3_z:
      case xmssmt_sha3-512_m64_w16_h60_d6_z:
      case xmssmt_sha3-512_m64_w16_h60_d12_z:
        bytestring64 root_n64;

      default:
        void;     /* error condition */
    };

    /* XMSS^MT public key structure */

    struct xmssmt_public_key {
      xmssmt_root root;  /* Root node */
    };
```

When bitmasks are used, XMSS^MT public keys are defined using XDR syntax as follows:

```
    /* Types for XMSS^MT bitmasks */
```

```
      union xmssmt_bm switch (xmssmt_algorithm_type type) {
        case xmssmt_aes128_m32_w4_h20_d2:
        case xmssmt_aes128_m32_w4_h40_d4:
        case xmssmt_aes128_m32_w4_h60_d6:
          bytestring16 bm_n16_t36[36];

        case xmssmt_aes128_m32_w4_h60_d3:
        case xmssmt_aes128_m32_w4_h40_d2:
          bytestring16 bm_n16_t36[56];

        case xmssmt_aes128_m32_w4_h20_d4:
        case xmssmt_aes128_m32_w4_h40_d8:
        case xmssmt_aes128_m32_w4_h60_d12:
          bytestring16 bm_n16_t26[26];

        case xmssmt_aes128_m32_w8_h20_d2:
        case xmssmt_aes128_m32_w8_h40_d4:
        case xmssmt_aes128_m32_w8_h60_d6:
        case xmssmt_aes128_m32_w16_h20_d2:
        case xmssmt_aes128_m32_w16_h40_d4:
        case xmssmt_aes128_m32_w16_h60_d6:
          bytestring16 bm_n16_t34[34];

        case xmssmt_aes128_m32_w8_h20_d4:
        case xmssmt_aes128_m32_w8_h40_d8:
        case xmssmt_aes128_m32_w8_h60_d12:
        case xmssmt_aes128_m32_w16_h20_d4:
        case xmssmt_aes128_m32_w16_h40_d8:
        case xmssmt_aes128_m32_w16_h60_d12:
          bytestring16 bm_n16_t24[24];

        case xmssmt_aes128_m32_w8_h40_d2:
        case xmssmt_aes128_m32_w8_h60_d3:
        case xmssmt_aes128_m32_w16_h40_d2:
        case xmssmt_aes128_m32_w16_h60_d3:
          bytestring16 bm_n16_t54[54];

        case xmssmt_sha3-256_m32_w4_h20_d2:
        case xmssmt_sha3-256_m32_w4_h40_d4:
        case xmssmt_sha3-256_m32_w4_h60_d6:
          bytestring32 bm_n32_t36[36];

        case xmssmt_sha3-256_m32_w4_h20_d4:
        case xmssmt_sha3-256_m32_w4_h40_d8:
        case xmssmt_sha3-256_m32_w4_h60_d12:
          bytestring32 bm_n32_t26[26];

        case xmssmt_sha3-256_m32_w4_h40_d2:
```

```
        case xmssmt_sha3-256_m32_w4_h60_d3:
          bytestring32 bm_n32_t56[56];

        case xmssmt_sha3-256_m32_w8_h20_d2:
        case xmssmt_sha3-256_m32_w8_h40_d4:
        case xmssmt_sha3-256_m32_w8_h60_d6:
        case xmssmt_sha3-256_m32_w16_h20_d2:
        case xmssmt_sha3-256_m32_w16_h40_d4:
        case xmssmt_sha3-256_m32_w16_h60_d6:
          bytestring32 bm_n32_t34[34];

        case xmssmt_sha3-256_m32_w8_h20_d4:
        case xmssmt_sha3-256_m32_w8_h40_d8:
        case xmssmt_sha3-256_m32_w8_h60_d12:
        case xmssmt_sha3-256_m32_w16_h20_d4:
        case xmssmt_sha3-256_m32_w16_h40_d8:
        case xmssmt_sha3-256_m32_w16_h60_d12:
          bytestring32 bm_n32_t24[24];

        case xmssmt_sha3-256_m32_w8_h40_d2:
        case xmssmt_sha3-256_m32_w8_h60_d3:
        case xmssmt_sha3-256_m32_w16_h40_d2:
        case xmssmt_sha3-256_m32_w16_h60_d3:
          bytestring32 bm_n32_t54[54];

        case xmssmt_sha3-512_m64_w4_h20_d2:
        case xmssmt_sha3-512_m64_w4_h40_d4:
        case xmssmt_sha3-512_m64_w4_h60_d6:
          bytestring64 bm_n64_t38[38];

        case xmssmt_sha3-512_m64_w4_h20_d4:
        case xmssmt_sha3-512_m64_w4_h40_d8:
        case xmssmt_sha3-512_m64_w4_h60_d12:
          bytestring64 bm_n64_t28[28];

        case xmssmt_sha3-512_m64_w4_h40_d2:
        case xmssmt_sha3-512_m64_w4_h60_d3:
          bytestring64 bm_n64_t58[58];

        case xmssmt_sha3-512_m64_w8_h20_d2:
        case xmssmt_sha3-512_m64_w8_h40_d4:
        case xmssmt_sha3-512_m64_w8_h60_d6:
        case xmssmt_sha3-512_m64_w16_h20_d2:
        case xmssmt_sha3-512_m64_w16_h40_d4:
        case xmssmt_sha3-512_m64_w16_h60_d6:
          bytestring64 bm_n64_t36[36];

        case xmssmt_sha3-512_m64_w8_h20_d4:
```

```
        case xmssmt_sha3-512_m64_w8_h40_d8:
        case xmssmt_sha3-512_m64_w8_h60_d12:
        case xmssmt_sha3-512_m64_w16_h20_d4:
        case xmssmt_sha3-512_m64_w16_h40_d8:
        case xmssmt_sha3-512_m64_w16_h60_d12:
          bytestring64 bm_n64_t26[26];

        case xmssmt_sha3-512_m64_w8_h40_d2:
        case xmssmt_sha3-512_m64_w8_h60_d3:
        case xmssmt_sha3-512_m64_w16_h40_d2:
        case xmssmt_sha3-512_m64_w16_h60_d3:
          bytestring64 bm_n64_t56[56];

        default:
          void;     /* error condition */
      };

      /* Types for XMSS^MT root node */

      union xmssmt_root switch (xmssmt_algorithm_type type) {
        case xmssmt_aes128_m32_w4_h20_d2:
        case xmssmt_aes128_m32_w4_h20_d4:
        case xmssmt_aes128_m32_w4_h40_d2:
        case xmssmt_aes128_m32_w4_h40_d4:
        case xmssmt_aes128_m32_w4_h40_d8:
        case xmssmt_aes128_m32_w4_h60_d3:
        case xmssmt_aes128_m32_w4_h60_d6:
        case xmssmt_aes128_m32_w4_h60_d12:
        case xmssmt_aes128_m32_w8_h20_d2:
        case xmssmt_aes128_m32_w8_h20_d4:
        case xmssmt_aes128_m32_w8_h40_d2:
        case xmssmt_aes128_m32_w8_h40_d4:
        case xmssmt_aes128_m32_w8_h40_d8:
        case xmssmt_aes128_m32_w8_h60_d3:
        case xmssmt_aes128_m32_w8_h60_d6:
        case xmssmt_aes128_m32_w8_h60_d12:
        case xmssmt_aes128_m32_w16_h20_d2:
        case xmssmt_aes128_m32_w16_h20_d4:
        case xmssmt_aes128_m32_w16_h40_d2:
        case xmssmt_aes128_m32_w16_h40_d4:
        case xmssmt_aes128_m32_w16_h40_d8:
        case xmssmt_aes128_m32_w16_h60_d3:
        case xmssmt_aes128_m32_w16_h60_d6:
        case xmssmt_aes128_m32_w16_h60_d12:
          bytestring16 root_n16;

        case xmssmt_sha3-256_m32_w4_h20_d2:
        case xmssmt_sha3-256_m32_w4_h20_d4:
```

```
        case xmssmt_sha3-256_m32_w4_h40_d2:
        case xmssmt_sha3-256_m32_w4_h40_d4:
        case xmssmt_sha3-256_m32_w4_h40_d8:
        case xmssmt_sha3-256_m32_w4_h60_d3:
        case xmssmt_sha3-256_m32_w4_h60_d6:
        case xmssmt_sha3-256_m32_w4_h60_d12:
        case xmssmt_sha3-256_m32_w8_h20_d2:
        case xmssmt_sha3-256_m32_w8_h20_d4:
        case xmssmt_sha3-256_m32_w8_h40_d2:
        case xmssmt_sha3-256_m32_w8_h40_d4:
        case xmssmt_sha3-256_m32_w8_h40_d8:
        case xmssmt_sha3-256_m32_w8_h60_d3:
        case xmssmt_sha3-256_m32_w8_h60_d6:
        case xmssmt_sha3-256_m32_w8_h60_d12:
        case xmssmt_sha3-256_m32_w16_h20_d2:
        case xmssmt_sha3-256_m32_w16_h20_d4:
        case xmssmt_sha3-256_m32_w16_h40_d2:
        case xmssmt_sha3-256_m32_w16_h40_d4:
        case xmssmt_sha3-256_m32_w16_h40_d8:
        case xmssmt_sha3-256_m32_w16_h60_d3:
        case xmssmt_sha3-256_m32_w16_h60_d6:
        case xmssmt_sha3-256_m32_w16_h60_d12:
          bytestring32 root_n32;

        case xmssmt_sha3-512_m64_w4_h20_d2:
        case xmssmt_sha3-512_m64_w4_h20_d4:
        case xmssmt_sha3-512_m64_w4_h40_d2:
        case xmssmt_sha3-512_m64_w4_h40_d4:
        case xmssmt_sha3-512_m64_w4_h40_d8:
        case xmssmt_sha3-512_m64_w4_h60_d3:
        case xmssmt_sha3-512_m64_w4_h60_d6:
        case xmssmt_sha3-512_m64_w4_h60_d12:
        case xmssmt_sha3-512_m64_w8_h20_d2:
        case xmssmt_sha3-512_m64_w8_h20_d4:
        case xmssmt_sha3-512_m64_w8_h40_d2:
        case xmssmt_sha3-512_m64_w8_h40_d4:
        case xmssmt_sha3-512_m64_w8_h40_d8:
        case xmssmt_sha3-512_m64_w8_h60_d3:
        case xmssmt_sha3-512_m64_w8_h60_d6:
        case xmssmt_sha3-512_m64_w8_h60_d12:
        case xmssmt_sha3-512_m64_w16_h20_d2:
        case xmssmt_sha3-512_m64_w16_h20_d4:
        case xmssmt_sha3-512_m64_w16_h40_d2:
        case xmssmt_sha3-512_m64_w16_h40_d4:
        case xmssmt_sha3-512_m64_w16_h40_d8:
        case xmssmt_sha3-512_m64_w16_h60_d3:
        case xmssmt_sha3-512_m64_w16_h60_d6:
        case xmssmt_sha3-512_m64_w16_h60_d12:
```

```
          bytestring64 root_n64;

        default:
          void;     /* error condition */
      };

      /* XMSS^MT public key structure */

      struct xmssmt_public_key {
        xmssmt_bm bm;  /* Bitmasks */
        xmssmt_root root;  /* Root node */
      };
```

Authors' Addresses

Andreas Huelsing
TU Eindhoven
P.O. Box 513
Eindhoven 5600 MB
The Netherlands

Email: a.t.huelsing@tue.nl

Denis Butin
TU Darmstadt
Hochschulstrasse 10
Darmstadt 64289
Germany

Email: dbutin@cdc.informatik.tu-darmstadt.de

Stefan-Lukas Gazdag
genua mbH
Domagkstrasse 7
Kirchheim bei Muenchen 85551
Germany

Email: stefan-lukas_gazdag@genua.eu