W-OTS# - Shorter and Faster Winternitz Signatures

Abstract

A very simple modification to the standard W-OTS scheme is presented called W-OTS# that achieves a security enhancement similar to W-OTS+ but without the overhead of generating and transforming a randomization vector in every round of the chaining function. The idea proffered by W-OTS# is to simply thwart Birthday-attacks altogether by signing an HMAC of the message-digest (keyed with cryptographically random salt) rather than the message-digest itself. The signer thwarts a birthday attack by virtue of requiring that the attacker guess the salt bits in addition to the message-digest bits during the collision scanning process. By choosing a salt length matching the message-digest length, the security of W-OTS# reduces to that of the cryptographic hash function. This essentially doubles the security level of W-OTS and facilitates the use of shorter hash functions which provide shorter and faster signatures for same security. For example, W-OTS# 128-bit signatures have commensurate security to standard W-OTS 256-bit signatures yet are roughly half the size and twice as fast. It is proposed that Blake2b-128 and Winternitz parameter $w=4$ (i.e. base-16 digits) be adopted as the default parameter set for the W-OTS# scheme.

1. Birthday Attack

A birthday attack involves an attacker forging a signature for a "malicious" message $M$ by re-using a signature for an "agreed" message $m$. In this class of attack, the attacker has knowledge of a message $m$ that the victim is willing and intending to sign in the future. The attacker creates variations of $m$ as $\{m_1..m_k\}$ any of which will also be deemed "valid" and signed by the victim. Whilst the victim considers each message $m_i$ "identical", their hash digests are unique. This can be achieved by simply varying one or more nonce values or whitespace within $m$ to create this set.

The attacker simultaneously generates variations of a "malicious" message $M$ as the set $\{M_1..M_l\}$ and stops until a collision $H(m_i) = H(M_j)$ is found (where $H$ is the cryptographic hash function used in the scheme).

NOTE: the probability of finding such collisions is far more likely than a standard brute-force attack by virtue of the Birthday problem.

When a collision-pair $(m_i, M_j)$ is found, the attacker asks the victim to sign valid $m_i$ giving $s = \text{Sign}(m_i, \text{key}) = \text{SignDigest}(H(m_i), \text{key})$. The attacker then proceeds to forge a signature for invalid $M_i$ by simply re-using $s$, as follows:
Unbeknownst to the victim, by signing $m_i$, they have also signed $M_j$.

2. W-OTS & W-OTS+

The Winternitz scheme is a well-documented scheme whose description is beyond the scope of this document. However, of relevance is the relationship between the W-OTS "security parameter" $n$ (the bit-length of $H$) and it's "security level" which is generally $n/2$. This follows from the fact that if a brute-force attack on $H$ requires $2^n$ hash rounds then a birthday attack requires $2^{n/2}$ hash rounds. By eliminating the birthday attack, and assuming no such other class of attacks exist for $H$, the security level of the scheme is restored back to that of a brute-force attack on $H$ which is $n$.

W-OTS+ achieves a similar security enhancement through obfuscation of pre-images in the hashing chains, however they are performed during the chaining function which adds an overhead (significant in some implementations). W-OTS# is similar to W-OTS+ in this regard except it only obfuscates the message-digest once via an HMAC (keyed with the salt) and uses the standard W-OTS chaining function, which is faster than W-OTS+. Despite the concatenation of the salt to the signature, the overall signature size decreases by virtue of selecting a shorter hash function $H$.

3. W-OTS#

The W-OTS# construction is almost identical to a standard W-OTS construction for Winternitz parameter $w$ and cryptographic hash function $H$. The security parameter $n$ is inferred from the the bit-length of $H$.

In W-OTS, a message-digest $md$ is computed as $md=H(message)$. During signing, digits of base $2^w$ are read from $md$ and signed in a Winternitz chain. In W-OTS#, the message-digest $md$ is replaced with the "sig-mac" $smac$ defined as:

3.1 Signature Message Authentication Code (SMAC)

```
1:  smac = SMAC(m, salt)
2:    = HMAC(H(m), salt)
3:    = H(Salt || H(Salt || H(m)))
```

The $salt$ is concatenated to the signature and used to compute $smac$ during verification.

**NOTE** the checksum digits are calculated and signed identically as per W-OTS but derived from $smac$ not $md$. 
3.2 Salt

The salt is generated by the signer using cryptographic random number generator. The length of the salt is $n$ bits which is the minimum value required to nullify a birthday attack (proven below). The salt is defined as:

1: $\text{salt} = \{0,1\}^n$ (i.e. $n$ cryptographically random bits)

3.1.2 Proof

1. A birthday-collision is expected after $1.25 \times \sqrt{U}$ hashing rounds where $U$ is maximum hashing rounds ever required (non-repeating).

2. In W-OTS, $U = 2^n$ where $n$ is the security parameter (bits-length of $H$) and thus (1) becomes $1.25 \times 2^{n/2}$.

3. In W-OTS#, adding a $d$-bit salt hardens a birthday-collision to $A = 1.25 \times 2^{((n+d)/2)}$ rounds. This follows from the fact that an attacker must scan for collision $(\text{HMAC}(H(m_i), \text{salt}), \text{HMAC}(H(M_j), \text{salt}))$ which involves $d$ more bits (whereas in W-OTS they just scan for $(H(m_i), H(M_j))$).


5. We need to choose $d$ such that $A = B$, since we only need to harden a birthday attack to match that of a brute-force attack. Hardening beyond is redundant since the security level of the scheme is only as strong as the weakest attack vector.

6. Evaluating (5) gives $d = 2 \ln(0.8)/\ln(0.2) + n = 0.2773 + n$ which is approximately $n$.

7. Thus choosing $d=n$ is sufficient to thwart birthday-attack. QED.

4. Reference Implementation

This section contains snippets for the full reference implementation. The reference implementation is part of the PQC library within the Hydrogen Framework.

public class WOTSSharp : WOTS {

    public WOTSSharp()
    : this(WOTSSharp.Configuration.Default) {
    }

    public WOTSSharp(int w, bool usePublicKeyHashOptimization = false)
    : this(w, Configuration.Default.HashFunction, usePublicKeyHashOptimization) {
    }

    public WOTSSharp(int w, CHF hashFunction, bool usePublicKeyHashOptimization = false)
    : this(new Configuration(w, hashFunction, usePublicKeyHashOptimization)) {
    }
public WOTSSharp(Configuration config) :
    base(config) {
    }

public override byte[,] SignDigest(byte[,] privateKey, ReadOnlySpan<byte> digest) =>
    SignDigest(privateKey, digest,
    Tools.Crypto.GenerateCryptographicallyRandomBytes(digest.Length));

public byte[,] SignDigest(byte[,] privateKey, ReadOnlySpan<byte> digest,
    ReadOnlySpan<byte> seed) {
    Guard.Argument(seed.Length == digest.Length, nameof(seed), "Must be same size as digest");
    var wotsSig = base.SignDigest(privateKey, HMAC(digest, seed));
    Debug.Assert(wotsSig.Length == Config.SignatureSize.Length * Config.SignatureSize.Width);
    seed.CopyTo(wotsSig.GetRow(Config.SignatureSize.Length - 1)); // concat seed to sig
    return wotsSig;
}

public override bool VerifyDigest(byte[,] signature, byte[,] publicKey, ReadOnlySpan<byte> digest) {
    Debug.Assert(signature.Length == Config.SignatureSize.Length * Config.SignatureSize.Width);
    var seed = signature.GetRow(Config.SignatureSize.Length - 1);
    return base.VerifyDigest(signature, publicKey, HMAC(digest, seed));
}

private byte[] HMAC(ReadOnlySpan<byte> message, ReadOnlySpan<byte> seed) {
    using (Hashers.BorrowHasher(Config.HashFunction, out var hasher)) {
        hasher.Transform(seed);
        hasher.Transform(digest);
        var innerHash = hasher.GetResult();
        hasher.Transform(seed);
        hasher.Transform(innerHash);
        return hasher.GetResult();
    }
}

public new class Configuration : WOTS.Configuration {
    public new static readonly Configuration Default;

    static Configuration() {
        Default = new Configuration(4, CHF.Blake2b_128, true);
    }

    public Configuration()
5. References


