CS 478 Lab Reports

Ekstedt, Andrew
ekstedta@oregonstate.edu

May 20, 2018

1 Lab 1: Puzzles

1.1 Introduction

We implemented a simple client-server puzzle scheme to mitigate denial of service attacks, based on brute-forcing n bits of a hash [1].

1.2 Benchmarks

We measured how long it took the client to solve puzzles as the number n of puzzle bits increased. For each n we made 20 requests to the server and timed how long they took. Up to 7 bits there is approximately no visible effect, but after 7 the time to solve each puzzle goes up by about a factor of two for each bit, which is what we expect. See Table 1.

Bits	Average solve time	Requests per second
4	0.5ms	1845
5	0.55ms	1753
6	0.70ms	1346
7	0.70ms	1348
8	1.0ms	992
9	1.7ms	557
10	2.1ms	470
11	4.3ms	227.8
12	8.2ms	121.5
13	14.2ms	70.1
14	29.2ms	34.2
15	65.1ms	15.3
16	86.7ms	11.5
17	207ms	4.8

Table 1: Benchmark results for client-server puzzles.

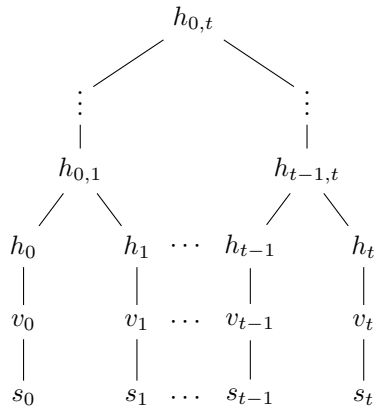

Figure 1: A Merkle tree on top of HORS. The bottom two rows of the tree are the HORS public key and private key elements. The remaining rows are intermediate nodes in the Merkle tree.

2 Lab 2: HORS

2.1 Introduction

We implemented PQ, a stateful many-time hashed-based signature scheme based on HORS. See algorithm 1 for the complete algorithm.

The main limitation of HORS is that each key pair can only be used once; we extend HORS and turn it into a d-time signature scheme. The main idea behind PQ is to generate d HORS keys, but instead of transmitting all $d \times t$ elements of the signature, we compute a Merkle tree over the leaves and use the root as the public key.

PQ contains three main improvements over HORS:

1. We use a Merkle tree to verify signatures, allowing us to shrink the size of the public key.
2. We use a PRF to generate the leaves of the Merkle tree, allowing us to shrink the size of the private key.
3. The Merkle tree is generated over $d \cdot t$ leaves instead of just t leaves, allowing us to sign d messages.

We present a diagram of a Merkle tree on top of a single HORS key in Figure 1. The root of the tree is capable of verifying all the leaves, allowing us to distribute just the root as the public key. The secret elements s_i are generated from a seed value as $s_i = F_z(i)$. Figure 2 shows the tree in PQ. This can be view as a series of d independent, virtual HORS signature trees, connected together with another Merkle tree on top, or it can be viewed as a single Merkle tree of $d \times t$ elements.

2.2 Tradeoffs

PQ trades increased signing and verification time for a smaller key size.

The use of a Merkle tree gives us constant-time public keys at the cost of an additional factor of $\log t$ during verification, and an additional $\log t$ factor for signature size due to the need to transmit the Merkle path for each signature element. The advantage is that public keys are constant-size, compared to $d * k$ for a d-time HORS scheme.

The biggest disadvantage is the increased signing time. Signing a message requires traversing the entire Merkle tree for each signature element, at a cost of $k * d * t$ compared to t for basic HORS.

Another disadvantage is the need to keep state on the signer side. This is because we must not reuse a HORS key that we have already used.

A table of the big O numbers for key size, signing time, and verification time are given in Table 2

Algorithm 1: The PQ signature scheme

Let $F : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^{k \log_2 t}$ be a secure PRF.
Let $H : \{0,1\}^* \to \{0,1\}^\lambda$ be a secure hash function.
Let t be the number of HORS key elements.
Let k be the number of HORS signature elements.
Let d be the maximum number of messages.
func *PQ.Keygen*
    $z \leftarrow \{0,1\}^\lambda$
    $st \leftarrow 1$
    for i *from* 1 *to* $d \times t$ **do**
        $s_i \leftarrow F_z(i)$
        $v_i \leftarrow H(s_i)$
    end
    $R \leftarrow Merkle.Create(v_1, \ldots, v_{d \times t})$
    return $sk = (z, st, d), pk = (R, d)$
end
func *PQ.Sign(m,sk)*
    if $st > d$ **then**
        abort
    end
    $(h_1 \| \ldots \| h_k) \leftarrow H(m)$
    for $j \leftarrow 1$ *to* k **do**
        $i_j \leftarrow (st - 1) \times t + h_j$
        $s_{i_j} \leftarrow F_z(i_j)$
        $v_{i_j} \leftarrow H(s_{i_j})$
        $P_{i_j} \leftarrow Merkle.Path(v_{i_j})$
    end
    $st \leftarrow st + 1$
    return $\sigma = (s_{i_1}, P_1, \ldots, s_{i_j}, P_{i_j})$
end
func *PQ.Verify(m,σ,pk)*
    $(h_1 \| \ldots \| h_k) \leftarrow H(m)$
    for $j \leftarrow 1$ *to* k **do**
        $v_j \leftarrow H(s_{i_j})$
        $b_i \leftarrow Merkle.Verify(v_{i_j})$
    end
    $b \leftarrow \underset{i \in 1..j}{\text{all}}\ b_i = 1$
    return b
end

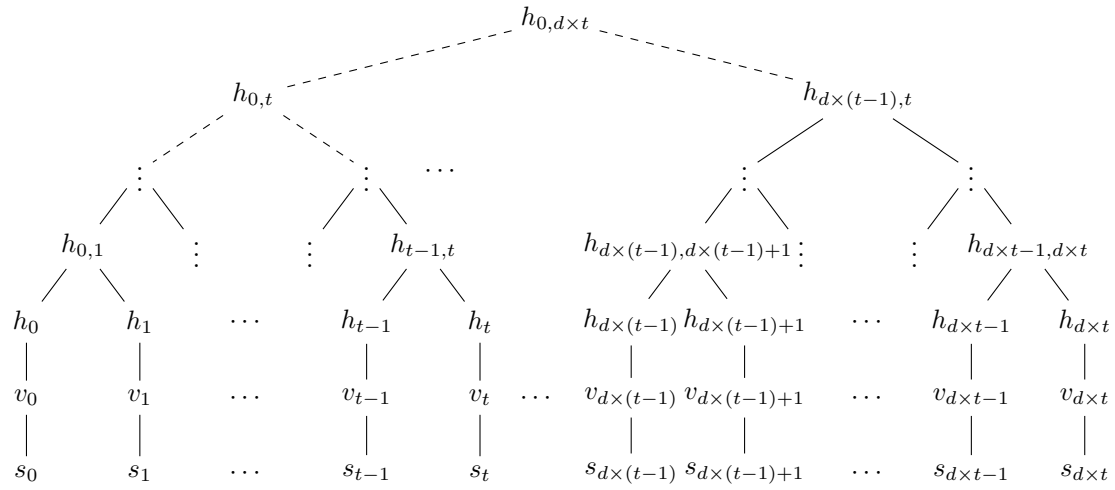

Figure 2: A Merkle tree in PQ. We can view this as d independent Merkle trees with t elements, with another Merkle tree on top.

	Secret key size	Public key size	Signature Size	Keygen time	Signing time	Verify time
HORS	$\mathcal{O}(t)$	$\mathcal{O}(t)$	$\mathcal{O}(k)$	$\mathcal{O}(t)$	$\mathcal{O}(k)$	$O(k)$
d-HORS	$\mathcal{O}(dt)$	$\mathcal{O}(dt)$	$\mathcal{O}(k)$	$\mathcal{O}(t)$	$\mathcal{O}(k)$	$O(k)$
PQ	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(k \log t)$	$\mathcal{O}(dt)$	$\mathcal{O}(kdt)$	$O(k \log t)$
HORS	8192 bits	8192 bits				
PQ (d=2)	256 bits	256 bits		10ms	136ms	0.77ms
(d=4)	⋮	⋮		21ms	271ms	0.84ms
(d=8)				42ms	546ms	0.91ms
(d=16)				83ms	1052ms	0.95ms
(d=32)				166ms	2145ms	1ms

Table 2: Comparison of HORS and PQ. d-HORS is a trivial variant of HORS which generates d separate keys to sign d messages. Assuming H is a 256-bit hash, t=1024 and k=32.

2.3 Benchmarks

We benchmarked our implementation of PQ and compared it to straight HORS signatures. See Table 2

3 Lab 3: Digital Forensic Tool

For lab 3, we implemented a mini digital forensic tool which synthesizes a number of different networking and cryptography techniques that we covered in class.

There are 6 properties that we desire from the tool. We enumerate those properties below, along with a rationale for how the tool satisfies that property.

- **Compromise resiliency**. The sensor should be secure against compromise. This is, if an adversary is able to take over the tool, we do not want stored messages to be revealed. We achieve this by continually refreshing the key stored on the device by hashing it with a secure hash. This way, even if the adversary learns the current key, they cannot recover old keys and so cannot decrypt old messages.
- **All-or-nothing verification** Messages are sent in batches; an adversary should not be able to selectively delete messages from a batch without being detected (nothing can prevent an adversary from blocking all messages). We achieve this property by sending a single authentication tag which covers all the messages in a batch. If even one message is deleted or changed, verification will fail.
- **Authentication and integrity** An adversary should not be able to spoof messages. We achieve this property by including an authentication tag with the messages, as above.
- **Compression** Stored messages should be compressed to save space. We achieve this property by using Huffman coding to compress each message.
- **Aggregate authentication** The authentication tag should use only a constant-size amount of storage on the device, even if there are many stored messages. We achieve this property by computing aggregating the authentication tags for each message in a hash chain. Only the last value in the chain is stored and sent to the collector.
- **Packet loss resiliency** The protocol should be usable on a noisy network; this is, the collector should be able to recover the messages even if a few packets are lost. We achieve this property by using Rabin information dispersal to split the messages into t packets such that the message can be recovered even if only n of t are received.

We apply operations in the following order when sending (and the reverse order when receiving).

1. Compression
2. Encryption
3. Authentication
4. Redundancy

Compression must come before encryption because encrypted data cannot be compressed (it is indistinguishable from random data). Authentication should come after encryption because Encrypt-Then-MAC is the only order that has been proven CCA-secure. Redundancy must be applied after encryption (before decryption) because you cannot decrypt data that is missing or corrupted.

3.1 Benchmarks

We timed how long it took to send messages with this scheme for a variety of message sizes. Results are summarized in Table 3

References

[1] A. Juels and J. Brainard, "Client puzzles: A cryptographic countermeasure against connection depletion attacks," 1999.

Message	Message size	Decryption time
64k bytes of the letter "a"	64000 B	3ms
64k bytes of the letter "a" (1000 messages)	64000000	3200ms
64k bytes of the unix dictionary	64000 B	23ms
entire unix dictionary	3571894 B	1100ms
jpeg image	3475052 B	1200ms

Table 3: Benchmark results for lab 3. Rabin parameters: $t = 8$, $l = 4$