High throughput asynchronous algorithms for message authentication

Scott A Crosby and Dan S. Wallach
Rice University

December 14, 2010

Abstract

Public-key digital signatures are a mainstay of cryptographic protocols and are widely used in practice. However, the performance gap between digital signatures and conventional cryptographic operations is significant, particularly with Internet messaging systems, chat systems, and the like, which would like to generate and verify digital signatures at high speeds. Consequently, our research considers hybrid schemes, derived from Merkle trees and newer hash-based data structures, that can optimize signing and verification of messages in bulk, while still preserving traditional single-message signature and verifications semantics. We show that hash trees can increase the signing throughput of *any* public key signature algorithm to tens of thousands of messages per second. In addition, our system's throughput degrades gracefully when the requested message signature rate saturates the CPU. Our techniques similarly create efficiencies when verifying groups of messages from the same source. In particular, by delaying any given signature verification until it's absolutely necessary, we can often verify multiple messages at the same time with minimal additional cost. We verify our system with detailed microbenchmarks of our hybrid signature algorithms as well as full-system simulations driven from traces taken from Google Wave.

1 Introduction

Public key signatures are widely used in the design and engineering secure systems and are built into a number of common cryptographic protocols including TLS [12]. Unfortunately, TLS only provides its authenticity, integrity, and privacy guarantees between two end points, which isn't a good fit for a variety of applications. When data is broadcast to multiple recipients, we need digital signatures that can stand alone and be independently verified. Many such systems, ranging from individual blogs to large aggregators like Twitter and Facebook, require their users to trust a central server to speak for the authenticity of individual messages. However, if that central server is compromised, by whatever means, all of its content becomes suspect.

A notable exception to this trend is Google's Wave service. Google Wave is a collaborative communication system supporting a federation protocol [17], where each document and user has a *home server*, and users across many home servers may concurrently edit the same document. To remain secure while enabling this functionality, Google Wave's federation protocol has a digital signature on every single message, where individual user keystrokes might well require individual signed messages. This has an obvious performance cost: over 75% of the runtime CPU time in the current "FedOne" implementation is spent on RSA signature generation. (While Google has announced it has no plan to continue developing Wave as a standalone product, the underlying Wave technology is expected to continue both within Google, other companies [29, 35], and through open-source distribution and adoption [17].)

In this paper, we investigate high throughput and low latency cryptographic primitives with semantics that are compatible with ordinary public key signatures, suitable for Wave and other comparable message-passing applications. For the purposes of this research, we are only interested in message integrity and authentication, not confidentiality.

The signing performance of any public key signature algorithm can be optimized by amortizing one public key signature over many messages by bundling multiple messages into a Merkle tree [24] and then computing a digital signature on the root. In Sections 5 and 6, we will discuss the performance of this approach.

Optimizing signature performance is not enough. Verification is also a requirement, as every message received must have its signature verified. If an attacker knew that a signature would never be verified, then the attacker would know that it could get away with forging a message. Consequently, in applications where messages fan out to a large number of users, signature verification will occur more frequently than signature generation.

While Merkle tree structures can easily enhance signature generation performance by batching messages together with cheap hashing operations, verification throughput is less likely to benefit from a simple approach like this because a given client may not necessarily be interested in the verification of multiple messages that happen to have been signed at the same time. In addition, batches will be generated frequently, keeping them small, in order to minimize latency. (A multiuser server would, presumably, queue up messages from all of its users and sign them in bulk. This minimizes the number of messages from any given user that would be in any given Merkle tree.)

Consequently, we must design new systems that enable fast verification of batched digital signatures. Toward this end, we present a new algorithm called *spliced signatures*: a hybrid containing both a cryptographic hash tree and a public key signature, preserving the semantics of a standalone public key signature. With spliced signatures, extra cryptographic hashes in the hash tree allow the cryptographic hash tree in one bundle of messages to have hashes of earlier messages *spliced* onto it. The newer public key signature can then be used to validate both old and new messages. Splicing is transitive, so a single public key verification can authenticate many messages sent by the same sender over time.

Splice signatures offer a latency-throughput tradeoff. If a message must be verified quickly, this can be done immediately at the cost of a single public key verification operation. However, increased verification latency can be acceptable, perhaps because users can tolerate the latency, or perhaps as a consequence of CPU saturation on the server, which simply cannot verify signatures at the message arrival rate.

We begin with a discussion of related work in Section 2 and background in Section 3, including a consideration of the design space for how multiple signatures may share locality, how signature latency may be tolerated, and how the programming semantics of signatures might be changed to take advantage of this. Section 4 introduces our spliced signature design. We evaluate our implementation with microbenchmarks in Section 5 and with a trace of Google Wave traffic in Section 6. Finally, we consider how our system might scale to large computing clusters in Section 7 and conclude in Section 8.

2 Related work

There have been many approaches to improve the performance of authentication algorithms. Merkle trees [24] were originally designed and are still used to aggregate several messages into a single root hash which is then signed. Similarly, linked-list style structures, as in Schneier and Kelsey [36, 37], allow a signature on a recent hash to validate earlier messages, and also allow a verifier to ensure that every message purported to be in a log is actually present. This has even been extended to allow network services to entangle with one another [23]. Data structures such as these, built from hash functions, have been designed for a wide variety of applications, including persistent authenticated dictionaries (see, e.g., [1, 19, 7]).

For this research, we only need semantics equivalent to standalone digital signatures, but we want throughput that's radically faster than traditional public key digital signature algorithms. Others have also pursued similar goals.

Improving public key algorithm performance We tested our algorithms with RSA and DSA implemented in software. We could have chosen other cryptographic primitives with different verification time, signing time, and size tradeoffs, or improved the performance of RSA and DSA by using external cryptographic hardware accelerators or computational cluster resources [6].

Algorithms offering faster signature verification than RSA or DSA include Rabin signatures [34], which require one modular multiplication to verify or Bernstein's RSA variant [5], which appears to be even faster.

Batched public key algorithms Many researchers have attempted to make public key algorithms that directly support batch signing and batch verification, which was formalized by Bellare et. al. [4]. Many of these algorithms have been found to be broken [8]. Camenisch et. al. [9] and Bellare et al. [4] have further discussion on this topic.

With Batch RSA [14], several messages can be combined together and signed in one exponentiation if they are to be verified with different public exponents. Optimizations for DSA signing have also been proposed [26]. Other approaches for increasing the throughput of public key algorithms include algorithms that can do parallel exponentiations of a constant g to random exponents by caching $g^{x_1} \dots g^{x_n}$ [25] for a 42%-85% improvement. Exponents can also be selected that offer efficient batch exponentiation [10].

Newer cryptographic primitives for batch verification have been proposed. Camenisch et. al. [9] present a signature scheme based on elliptic curve cryptography where a verifier can batch-verify signatures from many different signers. Several of these algorithms were evaluated for their performance [13]. Although batching lowers the per-message costs, the overall algorithm is still much slower than RSA or DSA.

While this class of algorithms has promise, they are primarily intended to allow one party (e.g., a web server which must perform a large volume of public key operations [38]) to compute results that are, to the remote party, no different than if they had been computed one-by-one. For our research, we are willing to make both parties aware of the batching strategy in return for improved throughput. Furthermore, the schemes we wish to consider are based around well-understood cryptographic primitives. Even if any particular algorithm is found to be weak, the Merkle construction and others like it are trivial to implement with any other algorithm having the common semantics expected of cryptographic hash functions and digital signatures.

Stream authentication Stream authentication [15, 33] addresses the challenge of authenticating a potentially infinite stream of messages, such as a stock ticker or multicast video stream. Stream authentication algorithms must support high throughput signature generation and verification and must function correctly even if the receiver does not receive every message (although there is an assumption that all receivers will get *most* messages in a stream).

Techniques for stream authentication include erasure codes [30], amortizing a public key signature over several packets using Merkle trees [40], and making signatures cheaper by using one-time-use signatures [15]. One system designed with efficiency in mind, TESLA [33, 32], releases successive pre-images of a hash function with each broadcast message, allowing for efficient authentication only at the time messages are seen rather than at any future time as with traditional digital signatures.

Our work differs from stream authentication schemes in that we do not require the receiver to be online nor do we require the receiver to need to see a substantial fraction of a stream. Each message in our work can be authenticated entirely by itself, at any future time, but we include optimizations to take advantage of stream-like locality in the data, when it's available.

3 Background

We first introduce some of the cryptographic data structures that we will use in this work, then we discuss some of the semantics for how we will use them.

3.1 Merkle trees

The structure of a Merkle tree is simple. A set of messages become the leaves of a tree. Then we build a binary tree, using cryptographic hash functions. Each node simply contains the hash of the concatenation of its two children. The root hash then represents a hash over all of the leaves of the tree, with all the same security semantics we would expect of a hash directly over the concatenation of the individual messages.

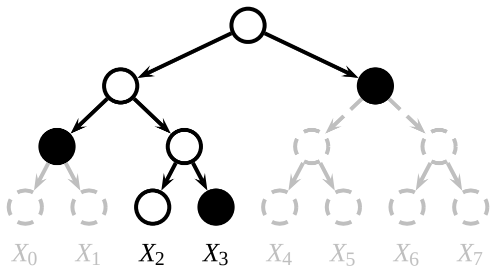

Figure 1: Graphical notation for a Merkle tree demonstrating the necessary hashes (solid black circles) to verify a Merkle signature on X_2. Open circles represent values that can be recomputed from the values below them. Grey circle nodes are unnecessary for the proof.

Applying a digital signature to this root hash is then equivalent to applying a digital signature to all of the elements.

Merkle trees gain their value when we wish to verify the signature over a specific leaf. Rather than needing every message to verify the signature, we only need a slice through the structure, as demonstrated in Figure 1. A *Merkle signature* [31] on X_2 would comprise the public-key digital signature on the root, X_2 itself, and the three internal hashes (the solid black circles). We call this a *pruned tree*, because it only contains a portion of the full hash tree. The verifier would then hash X_2, merging this result with the hash of X_3 and repeating two more times to yield the same value as was originally signed. If the signature on this root is valid, then the leaf node, X_2 is authentic. With Merkle signatures, delaying verification doesn't reduce the total work needed to verify the messages, but it can allow 'load averaging' where messages are queued up during an overload and verified when the overload ceases.

3.2 Locality

Locality is a measure of how often a verifier receives messages from the same signer. In some systems, the verifier is unlikely to receive messages from the same sender over time. In other systems, there is high locality. For instance, in Google Wave, an active user doing collaborative editing will be sending update messages several times per second. When there is locality, there is the prospect that a spliced signature can exploit it to amortize one public key verification across several messages.

3.3 Latency

We can examine the latency of a system using queuing theory. When a message arrives, there is both the time to sign or verify and the time that the message must wait before being processed. As the load increases, queuing delay will increase as the CPU gets behind. At 90% load, queuing delay will be 4.5 times the signing time or verification time. When overload occurs, the buffer of unprocessed messages grows faster than computation can drain it and the latency diverges to infinity.

To minimize latency, it is critical to prevent a system from entering overload. With discrete or *simple signatures*, where each message is individually signed with a public key algorithm such as DSA or RSA, the maximum throughput is that of the underlying public key algorithm. If we batch messages using a Merkle tree or other such structure before signing, we can amortize the costs of a public key signature over many messages. The peak throughput becomes the rate at which we can build and serialize the hash trees, not the rate at which the public key algorithm can sign messages. Further investigation of latency tradeoffs of batch signing is described in Korkmaz [20].

3.4 Asynchronous semantics

We follow the same general semantics of digital signatures, offering a signing operation and a verifying operation. Traditional digital signature APIs are synchronous, in that the signature bytes or validity of a signature comes back immediately when the signature of verification is requested. In order to use Merkle signature or other such structures, we need an asynchronous API, where messages can be submitted to be signed but the actual digital signature operation is delayed. In the case of Merkle trees, this is necessary to accumulate enough messages to fill out the leaves of the Merkle tree. Figure 2 shows a simple Java-like API for this, where outgoing messages are submitted as objects with callbacks that will be later invoked when the signature has been computed. Of course, this could be implemented in a variety of different lazy styles in other programming languages, but the result is the same. At some later point, the pending queue of messages is processed as a batch, and all the signatures for the batch are available at once.

In our corresponding asynchronous verification API, seen in Figure 3, the application submits messages to be verified at a later time. Messages can be *forced* at any time when the application requires that it must be verified, such as when a user logs in or opens a document. At that time, the verification module verifies the signature and invokes the callback with the result. The verification module may also invoke the callback on its own if it happens to verify messages that were not explicitly forced.

Asynchronous implementations, both when signing and when verifying messages, may also be more robust when the CPU is saturated. If the per-batch overhead is larger than the per-message overhead, a batch algorithm can compensate for higher arrival rates by using larger batches.

Asynchronous semantics have clear potential to improve signature throughput relative to discrete digital signatures, however, they are not beneficial for systems that require absolutely minimal verification latency when there is plenty of available computation relative to the desired message rate, or systems with poor locality.

3.5 Implementation details

Our verifier uses a single thread for all cryptography. That thread is responsible for reading *force* requests and incoming messages from mailboxes and sending them to an underlying cryptographic module that implements the asynchronous signature verification API.

```
class OutgoingMessage {
   byte []getMessageBytes();
   void sigBytesCallback(byte sigbytes[])
}

class SignQueue {
  void submit(OutgoingMessage msg);
  void process();
}
```

Figure 2: Asynchronous signature API.

```
class IncomingMessage {
   byte []getMessageBytes();
   byte []getSigBytes();
   void validityCallback(bool valid);
}

class VerifyQueue {
  void submit(IncomingMessage msg);
  void force(IncomingMessage msg);
  void forceAll();
}
```

Figure 3: Asynchronous verification API.

It first examines the incoming message mailbox. If there is an unprocessed message, it adds it to an internal buffer of unverified messages. If there are no incoming messages, but there is a force request, then we immediately verify that message and find other messages which may also be verified with the same digital signature, and notify all of them of the outcome. If there are no outstanding force requests or incoming messages, and thus the thread is idle, it forces the oldest unverified message. If there is nothing to do, it waits. In the case of batch-verification algorithms, handling all incoming messages before force requests can prevent overloads by allowing one batched verification to verify multiple forced messages.

Consequently, there will be two different conditions under which the verification system operates. If the message arrival rate is below what the CPU can natively handle, the verifier will simply verify every message eagerly, reducing the degree to which one message verification will benefit another; however, since there's excess CPU capacity, we should end up with lower latency verification. In the alternative case, where messages are arriving faster than the CPU can process them, the system will prioritize important messages to maintain the desired message throughput rate. Less important messages will be queued and opportunistically verified. If important messages arrive faster than they can be verified, batching can prevent overload, but latency will increase compared to the unloaded case. At some point, the message throughput will exceed the available CPU's ability to process it, regardless of the benefits of batch processing. At this limit, throughput cannot grow, so latency will necessarily suffer, and clustering algorithms of some sort must be used to preserve scalability. (See Section 7 for details.)

4 Spliced signatures

The idea of spliced signatures is simple. Say that Alice sends Bob three individually signed messages M_1, M_2, M_3. Bob's computer could authenticate each of them immediately at the cost of three public key verifications, but doesn't need to because Bob isn't logged in, or can't, because Bob's computer is overloaded. We would prefer for Bob's computer to do a public key verification of just one message, M_3, and then be able to authenticate all three messages.

4.1 History trees

Crosby and Wallach [11] introduced an improvement to Merkle trees that they called *history trees*. Where a Merkle tree is computed and then fixed for all time, a history tree allows for multiple *versions* of the tree, where later versions incrementally add to the hash trees of earlier versions.

Crosby and Wallach's history trees have semantics comparable to hash chains [37], where a recent hash is sufficient to verify older messages. Unlike hash chains, however, a linear scan over the chain is unnecessary. Instead, the creator of a history tree can produce a proof that a new root hash is consistent with an older root hash in $O(\log n)$ space. Likewise, the creator can be challenged to prove that any given leaf is in the history tree. As with the Merkle tree example in Section 3.1, the creator need only generate a logarithmic slice through the tree to the root node, when then can serve as a standalone proof that the message held by the verifier is the same as the one signed by the creator.

The core difference between our use of the history tree and its use as a tamper evident log is that when used by a tamper evident log is that with spliced signatures, we trust the signer. With a tamper evident log, we do not trust the signer.

We now present an overview of how history trees work. A filled history tree of depth d is simply a binary Merkle hash tree, storing 2^d events on the leaves. Successive leaf nodes store the hashes of successive messages. When a tree is not full, subtrees containing no messages are represented as $\square$ and have a hash value of $\emptyset$. This can be seen starting in Figure 4, a version-2 tree having three events. Figure 5 shows a version-6 tree, adding four additional events. Although the trees in our figures have a depth of 3 and can store up to 8 leaves, our design clearly extends to trees with greater depth and more leaves. When the tree is full, a new root, one level up, can be created with the old tree as its left child and an empty right child where new events can be added.

An interesting property of the history tree is the ability to efficiently reconstruct old versions or *views* of the tree. Consider the history tree given in Figure 5. The logger could reconstruct the root value C_2 analogous to the version-2 tree in Figure 4 by pretending that later nodes (marked with asterisks in Figure 5) were $\square$ and then recomputing the hashes for the interior nodes and the root. If the reconstructed C_2 had the same hash as a previously received hash of C_2, then both trees must have the same contents and commit the same events.

A proof that a message X_3 is committed is the *pruned tree P*, shown in Figure 6. It includes just enough of the full history tree from in Figure 5 to be able to validate X_3, validate X_2, reconstruct C_2, reconstruct C_3 and compute C_6. Unnecessary subtrees are elided out and replaced with stubs.

4.2 Implementing spliced signatures

The history tree is built in a batch fashion. Whenever the application requests that the set of outstanding messages be signed, the spliced signature module adds the new messages to the history tree. This requires $O(1)$ amortized hash operations per message. Then the root commitment C_n is generated, requiring $\log n$ hash operations, and signed. The spliced signature module then generates a pruned tree to authenticate each message, with each message's pruned tree being unique to that message.

Each batch of messages is appended to the same history tree, which could ostensibly grow indefinitely. To reduce RAM consumption in our implementation, we restart with an empty history tree every 100k messages. Signatures cannot be spliced across these epochs.

4.2.1 Authenticating X_3 in a batch

Consider the case where a signer adds 7 messages as a single batch into an empty history tree. The result is the history tree seen in Figure 5.

The minimum pruned tree that can reconstruct the root hash commitment C_6 is presented in Figure 7, containing a path to leaf X_6, the last leaf inserted. The pruned tree in the spliced signature that authenticates X_3 is seen in Figure 6. It takes the minimum pruned tree in Figure 7 and adds a path to leaf X_3, allowing that message to be authenticated. This tree, the public key signature on C_6, and the message X_3 is sufficient to authenticate X_3. To verify this spliced signature, the verifier hashes X_3, places the hash into the pruned tree, and recomputes the hash of the root of the tree, which should be equal to the one already signed. The verifier will also validate the public key signature on the root hash.

4.2.2 Splicing two history trees

In this example, we will assume that the signer runs two batches and generates two signatures. The first batch contains 3 messages and the second batch contains an additional 4 messages. The verifier receives two spliced signatures; the first contains the pruned tree seen in Figure 4, authenticating X_1, and having a signed root hash commitment C_2. For the second spliced signature, instead of sending the pruned tree seen in Figure 7, which authenticates message X_6 with a signed commitment of C_6 [1] the signer instead sends the pruned tree in Figure 6. This tree includes an extra path to leaf X_2. The second signature thus connects to the first one.

The verifier could verify these two message independently at the cost of two public key signatures. What if we first verified C_6's public key signature on the tree in Figure 6? We validate message X_6 from this tree. Observe that the pruned tree in Figure 6 also includes a path to X_2. Then, if we pretended that the asterisk-marked children in this tree were $\square$, we could reconstruct C_2, using hash operations. If the reconstructed hash C_2 was the same as the C_2 computed from Figure 4, then we have a valid splice. We have authenticated C_2 from the pruned tree in Figure 6 and the signature on C_6. Now that we have authenticated C_2, we can use the pruned tree in Figure 4 to authenticate X_1. In the end, we have validated two messages using $5 \cdot \log n$ hash operations and one public signature verification.

We call this a spliced signature because if the reconstructed C_2 is the same as the original C_2, the contents of the left children on the path to X_2 in Figure 4 are the same as the left children on the path to X_2 in Figure 6, and we could thus safely graft the tree in Figure 4 onto the tree in Figure 6.

In this example, the two messages being verified were in sequential batches. That is not necessary. What matters is that the verifier was able to use the latter pruned history tree in Figure 7 to reconstruct a commitment of the earlier history tree C_2 by virtue of Figure 7 including a path to leaf X_2.

Also note that for the recipient to be able to splice the two trees together, the signer had to send Figure 6, not Figure 7. It had to predict that the verifier would have previously received the spliced signature authenticating X_1. When this prediction is not possible, spliced signatures are no better than Merkle signatures because the signer will not know what prior messages to splice to.

[1]Figure 7 already includes a path to message X_6, and can authenticate that message without adding any additional nodes.

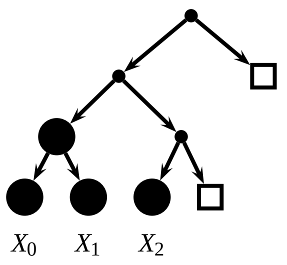

Figure 4: A version 2 history with commitment C_2.

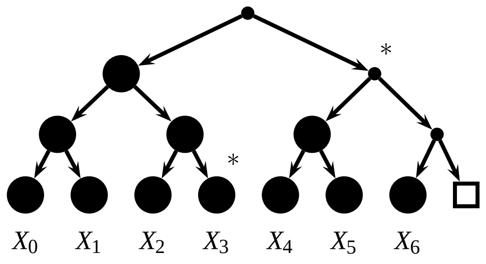

Figure 5: A version 6 history with commitment C_6.

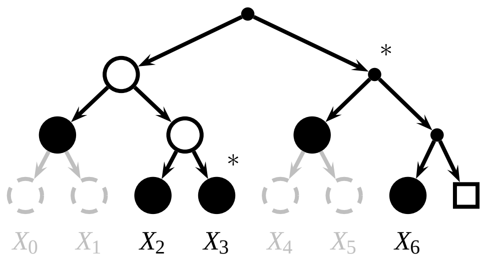

Figure 6: A pruned tree indicating a batch ending in version X_6 with commitment C_6 and including a path to leaf X_2. If that path is considered a splicepoint to C_2, this tree can be merged with the tree given in Figure 4. Solid black circles are included in the output. Open black circles need not be included as they can be computed from their children. Grey circles need not be included because they are not relevant to the proof being constructed. Small dots, like open circles, can be recomputed from their children, but can be expected to be different in future versions as more messages are inserted.

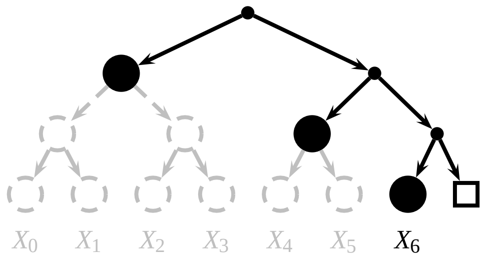

Figure 7: A pruned tree that authenticates message X_6 that was generated during the batch ending in version X_6. Solid black circles are included in the proof. Grey circles need not be included because they are not relevant to the proof being constructed. Small dots can be recomputed from their children, so they need not be included.

4.3 General splicing

In general, the pruned tree for a spliced signature P for a message X_i will contain at least two paths; it will contain a path from the root to X_j, where C_j is the commitment at the end of the batch, and a path to X_i, which is the particular message being authenticated. As the depth of the tree is $\log n$, the number of hashes in the two paths to the two leaves in the pruned tree is $2 \cdot \log n$.

Lets say the signing application later requests a spliced signature on message X_k. The minimum pruned tree Q in the spliced signature on X_k includes a path to the leaf X_k and a path to the leaf X_l, where X_l is the last message added in the batch.

Now lets say the signing application predicts that the verifier could take advantage of a splice from message X_i to X_k. The signer takes the pruned tree Q and adds in one additional path, to leaf X_j, the end of the batch for message X_i. That *splice to C_j* increases the size of Q by $\log n$ hashes.

A spliced signature is not limited to splicing to one prior commitment. It can splice to many prior commitments. Each splice requires $\log n$ hashes. For simplicity, we will assume that a spliced signature includes a list of the index numbers of all prior commitments to which it splices.

We could have included a path directly to message X_i. We spliced instead to C_j, the commitment that ends the batch containing X_i to simplify the design of the spliced signature verifier and reduce the number of distinct splicepoints, which increases the opportunities of opportunistic splicing, splicing that was not planned by the signing application.

Verifying spliced signatures is transitive. Lets say that the verifier receives P and Q and Q has splice to C_j. If the commitment C_l authenticating Q is authenticated, whether by having its public key signature verified directly, or being authenticated because it spliced into a later authenticated message, then Q is authenticated. If Q is authenticated, any prior commitments which splice into it are authenticated, as well as any messages authenticated by those commitments. This includes message X_k and commitment C_j, which authenticates P, and transitively, through earlier commitments which splice into P. With only a single public key verification, many messages can be authenticated through hash operations.

4.4 Implementing the asynchronous verification interface

There are two ways to authenticate a message. We can always ignore the splices entirely and validate a spliced signature by following the procedure outlined in Section 4.2.1, by using the pruned tree and the message being authenticated to generate the root commitment for that batch and then checking the public key signature on that commitment. This requires doing one public key verification for each verified message, the same as we would do with a standard Merkle tree.

To improve our efficiency we must leverage the asynchronous API and delay the verification in order to exploit any splicepoints between the spliced signatures. In Section 4.3 we described how to verify a splice between two messages. In this section, we describe the bookkeeping required in order to find and efficiently exploit splicepoints. In particular, we must track messages which may splice other messages, and we must remember when digital signatures, hashes, or splices have been verified. Because we must delay public key signature verification as long as possible—the whole point is to minimize public key signature verification—our bookkeeping stores unauthenticated messages and must properly reject forged messages.

It is not possible to splice two spliced signatures that were generated from different history trees. We can only splice signatures generated from different versions of the same history tree. To identify if two messages were generated in the same history tree, a spliced signature includes a tree-id, created randomly for each distinct history tree. For simplicity, in the rest of this section, we will assume that all messages are from the same history tree. Otherwise, the verifier could demultiplex the messages by using the tree-id and replicate this algorithm once for each distinct history tree.

If two messages have a verified splice between them, then validating the newer message will also validate the older one. To accomplish this, we must track these dependency relationships. Repeating this across all messages forms a logical dependency graph which we explicitly maintain for all messages. We place a node in this graph for each message, marking it as depending on the node representing the commitment for its batch. We also place a node in this graph for each end-of-batch commitment. We will add edges between nodes that represent end-of-batch commitments only if there is a verified splice between them.

For each pruned tree, we cache its root hash in a hash table $\text{Roots} = \{i_1 \to C_{i_1}\ i_2 \to C_{i_2} \ldots\}$ of already seen commitments.

Whenever a new spliced signature M arrives with commitment C_n, we determine if any existing messages splice into it. The incoming message contains a list of prior commitments $i_1, i_2, \ldots$ that splice into it. We use the pruned tree in M to reconstruct those prior hashes, $C_{i_1}, C_{i_2}, \ldots$. If they are the same as the hashes cached in Roots then we have validated the splice from C_n to C_{i_k} and add this edge into the dependency graph. If M contains a splice to a prior commitment that the verifier has not encountered, we record the potential splice in a separate table of prospective splices. We next determine if M splices into any existing messages. This is possible if messages arrive out of order. To do this, we see if M's commitment is in the table of prospective splices. If it is, we validate the splice and add the edge between the commitments into the dependency graph.

This algorithm verifies splices eagerly, as soon as messages arrive, and adds dependencies to the dependency graph. Determining parenthood—verifying a splice—requires $O(\log n)$ hash operations. For a spliced signature with r splices, the total time is $O(r \log n)$ hash operations.

When validating a message M with commitment C_n that has been forced, we could verify M's spliced signature, by using the pruned tree and the public key signature on that commitment. If the signature validates, instead of invoking the callback to mark just M as valid, we traverse the dependency graph for all descendants of C_n, which includes M, and invoke the callback on them.

However, we can further improve efficiency by finding a later message R which has M as a descendant, and verify R's commitment's public key signature. The best node R to use is a root of the dependency graph that has M as a descendant, which can be found with a depth-first-search (*DFS*).

Consider the error case where a public key signature on a root fails to verify. This could occur because the root signature (and its accompanying messages) are forgeries, or it could occur due to a transient failure of the signer. In either case, we must disregard the root signature and consider the messages that came with it to be unverified (as with any other new message). Rather than terminating or restarting the DFS, the DFS need only cache its search path and resume where it left off. By doing this, verification can be done in constant time per validated or rejected message.

5 Microbenchmark evaluation

We implemented all of our algorithms in Java OpenJDK 1.6.0_18 running a quad-core Intel Core i7-860 at 2.8GHz running Linux in 64-bit mode. Our implementation used Google Protocol Buffers [16] for serialization. We use BouncyCastle [22] for 2048-bit RSA signatures and 1024-bit DSA signatures. We used SHA2-256 for cryptographic hashing. We would have used 2048-bit DSA signatures except that they are not supported by our Java libraries. (Based on OpenSSL benchmarks, we would expect them to be about 3.5 times slower to sign and verify than 1024-bit DSA.) Except for DSA, our ciphers all operate at the 256-bit level of security [27].

Our benchmark harness tracks message latencies. We record when a message is generated, processed, forced, and ultimately verified. To minimize performance artifacts induced by our benchmarking harness, it runs in a separate thread on its own CPU core.

The Merkle and history trees used in our cryptographic primitives have a very specific structure, that of a complete binary tree where empty right leaves store nothing (i.e., a pointer to null). Just as a heap can be stored in an array to avoid object allocation overheads, we store our hash trees in an array, where a node's offset is assigned based on an in-order traversal.

In Tables 1 and 2 we present our Java and OpenSSL public key microbenchmarks. Except for DSA verification where OpenSSL is 6.5 times faster than Java, OpenSSL

	Sign	Verify
RSA-2048	10	.276
DSA-1024	.868	1.720
SHA2-256 on 64 bytes	.00125	

Table 1: CPU time for public key operations in Java in milliseconds.

	Sign	Verify
RSA-2048	2.72	.080
DSA-1024	.231	.267
ECDSA-256	.138	.606
SHA2-256 on 64 bytes	.0008	

Table 2: CPU time in OpenSSL 0.9.8o in milliseconds.

is 3.4-3.7 times faster than native Java. Clearly, Java systems that depend on crypto should consider using native methods to improve crypto performance.

Regardless, these performance numbers show an interesting trend. RSA, while being very expensive to compute a digital signature, is very efficient to verify. As we will discuss later, verification performance is essential to our system, while signature generation is less sensitive, so these benchmarks tend to lead us to prefer RSA over DSA signatures.

5.1 Hash tree microbenchmarks

We now focus on microbenchmarking the costs of Merkle trees and history trees. These are the foundation of the two asynchronous signature schemes that we present.

The critical serialized path in our hybrid signature algorithms for signing is generating the Merkle or history tree. Our current implementation does this in a single thread. In Figure 8 we summarize the costs of building these trees across different tree sizes, which tracks linearly at roughly 2ns per message.

In Figure 9 we plot overall *per-message* costs, including the cost of serializing a message's pruned tree for authenticating that message and the cost of building the tree over all messages in the batch. We find that history trees are more expensive than Merkle trees to generate because the membership proofs are approximately $2\log n$ nodes while Merkle tree signatures have $\log n$ nodes. This can also be seen in Figure 10, where we plot the serialized size of the generated pruned history or Merkle trees.

A realistic batch size for a Merkle tree is at most a few hundred nodes, with an amortized per-message costs of 15μs to generate or verify. History trees contain messages from several batches, growing until they hit a maximum size. At a 100k maximum size, the amortized per-message cost is 55μs.

The cost of either Merkle trees or history trees is much less than the digital signatures reported in Table 1. DSA is the fastest signing algorithm we benchmark, at .86ms per signature. This is the same time it takes to batch 90 messages into a Merkle tree and serialize 90 hybrid message signatures. Thus, in about 1.7ms, we can generate hy-

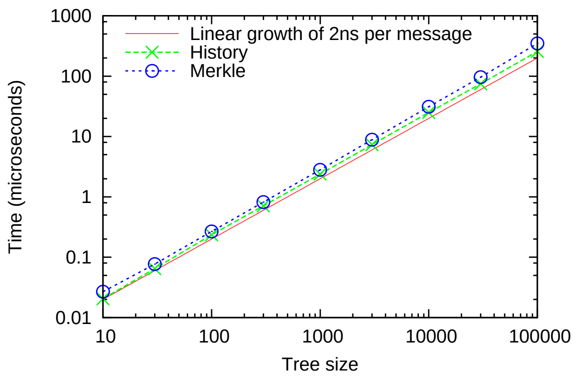

Figure 8: Time required, in microseconds, to build a Merkle or history tree, required before any signatures can be generated.

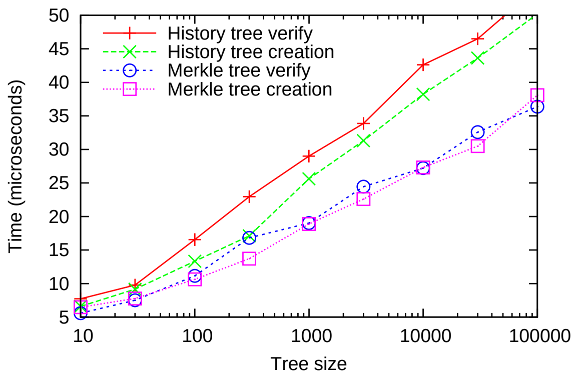

Figure 9: Amortized time, in microseconds, to generate and verify a membership proof in a Merkle or history tree as a function of the number of elements in the tree.

brid signatures over 90 messages, for an estimated throughput of 53000 messages per second. (In Table 3, we benchmark this algorithm as overloading at 52000 messages per second with a batch size of 108 messages.)

5.2 Peak throughput

Our previous microbenchmark examined the generation performance of Merkle trees and hash trees in isolation. In this microbenchmark, we analyze the performance of our asynchronous signing and verifying API by empirically determining the peak throughput before overload for signing and verifying messages in three styles: simply standalone signatures, Merkle tree signatures, and history-tree spliced signatures. We take each algorithm and progressively generate messages at a faster rate until the queue length (and latency) diverges to infinity.

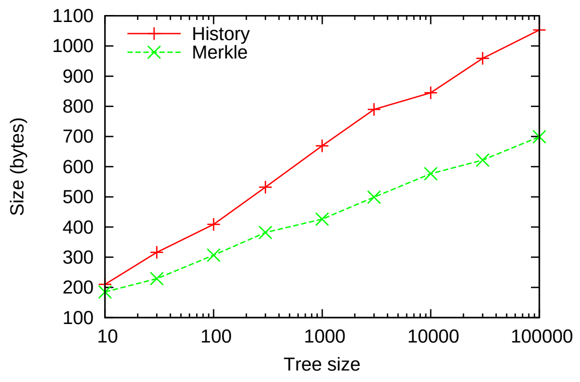

Figure 10: The number of bytes in the serialized SHA2-256 Merkle or history tree proof, as a function of the number of nodes in the tree, not including the signature on the root.

	RSA			DSA		
	Rate (/sec)	Latency (ms)	Batch Size	Rate (/sec)	Latency (ms)	Batch Size
Simple	80	65	1	800	< 1	1
Merkle	35000	38	1107	41000	3.0	108
Spliced	21500	47	846	23000	4.4	88

Table 3: Peak signature generation rate in messages/sec, average latency and batch size. Data reported is at 80% of the overload rate.

Our implementation has two threads, a signing thread and a message generating thread. The message generator thread generates messages at a given rate for 60 seconds, recording the latency from generation to processing of the message. When the queue exceeds more than one second of unprocessed work, we assume that the system has overloaded and terminate the experiment.

Java's garbage collection causes some artifacts in these results. When the system is very close to overload, a GC pause causes a sudden spike of unprocessed messages in the queue that instantly drives the system into overload. A production system faced with the same GC latency would similarly find itself overloaded, but production systems should hopefully address such performance concerns by spreading the workload across many separate machines (see Section 7). For our benchmarks, these results are still perfectly valid as we wish to measure the throughput of each signature technique at 80% of the fastest rate we were able to process messages without overloading (see Table 3), which we believe represents a more typical peak latency as might be seen in a production system. As the system gets closer to overload, latency rapidly increases to the point that it would be unacceptable. Our measurements show, for example, that at half of the overload rate, latency is about 20% less than at 80% of the overload rate.

In Table 4 we report the peak rate at which we could verify messages. Simple signa-

	RSA		DSA	
	Rate (/sec)	Latency (ms)	Rate (/sec)	Latency (ms)
Simple	2800	2.8	500	8.9
Merkle	2800	2.4	500	8.8
History	12700	7.5	11200	11.2

Table 4: Peak input for message verification. Data reported is at 80% of the overload rate.

tures and Merkle signatures are bottlenecked by the cost of a public key cryptographic operation, limiting their maximum throughput and setting their peak latencies.

History tree spliced signatures are a bit trickier to benchmark for verification. If the messages had no locality at all, the throughput and latency of spliced signatures would follow that of Merkle signatures. However, we expect real-world data to have better locality. For this experiment, we decided to measure something of a best-case scenario; we captured the peak throughput of spliced signatures with messages arriving as fast as possible, verified as fast as possible, and with maximum locality. In this scenario, the latest message's public key signature, combined with the history tree hashes accompanying prior messages, is sufficient to authenticate *all* prior unverified messages only using hash operations. The results are clear: under ideal situations, history tree spliced signatures are running 4.5-20 times faster than simple standalone digital signatures or Merkle trees. By tolerating a longer latency, significant gains can be found in verifying messages. Also, the faster verification performance of RSA signatures is now clear relative to DSA signatures.

6 Evaluation on Google Wave

In our previous sections, we benchmarked spliced signatures and Merkle signatures in an application-independent way to determine their peak throughput. We showed that spliced signatures and Merkle signatures have similar throughputs for signing. However, the verification performance of spliced signatures can be significantly higher if enough locality is present and the additional latency can be tolerated. In order to better characterize the performance of our designs, we next compare spliced signatures to Merkle signatures using traces from Google Wave to demonstrate the advantages of spliced signatures.

6.1 Google Wave

Google Wave is a realtime, concurrent, distributed system that allows users to concurrently edit shared documents (called "waves"). Waves can take several related forms: documents as in a wiki, discussion threads as in many web discussion forums, or status updates as in Twitter and other such systems. To support concurrent editing, Wave uses *operational transforms* [28], allowing for concurrent editing operations to be resolved to a consistent state, regardless of the order in which the operations are applied.

A wave is expressed as a sequence of delta operations that insert or remove characters, move a virtual cursor, and so forth. To support low latency concurrent editing, deltas are transmitted as fast as possible. Consequently, a fast-typing user can generate many deltas per second.

Wave resembles traditional email in that users have a home *wave server* which operates on their behalf, serializing concurrent updates to documents that it manages. Wave supports a federated protocol [21] [3], allowing one wave document to be edited and observed from users across many hosts in different trust domains. Unlike traditional email, a Wave server digitally signs every delta on behalf its users, allowing for third-party sharing of deltas, which can then still be authenticated. (For purposes of this research, we omit discussion of how Wave authenticates users, manages access control lists, and so forth. Some discussion appears in Tirsen [39].)

Google originally designed a Merkle tree-based bundling and signing mechanism for Wave [18], but this has not actually been implemented. Elements of the design were implemented in [2] but have not yet been integrated into any actual wave server. The present Google "FedOne" Wave server instead implements a linear hash chain within each wave. The design here is intended to improve the efficiency of the digital signatures used within this scheme.

In many circumstances, a wave server can tolerate long latencies between when a delta is received and when it's verified. For example, if no user on that server is presently viewing the wave, then delta validation can be delayed until one of them logs in. Also, for read-only observers of a wave, latencies of a few seconds could be perfectly tolerable.

6.2 Google Wave implementation details

With spliced signatures, the signer has to choose which splicepoints to include in the spliced signatures. One simple solution for which messages to splice would be to include a splice to the immediate prior batch, creating something analogous to a hash chain. In the case of Google Wave, however, the stream of messages being signed by a given server will contain a mix of messages for a variety of different waves on different home servers. Consequently, we can optimize the verification process by including splices, on a per-message basis, to the commitment of the previous batch containing a message destined for the same wave. This allows the subset of messages relevant to any given home server to be self-contained with respect to that server's ability to verify earlier messages from later ones.

6.3 Statistics on the Google Wave dataset

We received a trace of all updates on a subset of the individual waves running on Google's servers over the course of a day. Each update contains a millisecond-accurate timestamp, an anonymized userid, and an anonymized waveid. Our trace contains 50k users, 94k waves and 14M total deltas[2]. Half of the users in our trace only sent deltas

[2]Google has asked us not to disclose statistics that allow the reader to infer the total volume of Wave traffic handled by Google's servers.

Fraction of messages	Time since prior message (s)
12 %	.2
45 %	.5
55 %	1
91 %	2
95 %	5
96.8 %	10
97.8 %	20

Table 5: Cumulative distribution function time difference between updates sent by a user on a particular wave.

on one wave. 75% of waves only have a single user, 20% have two users, and only 5% have three or more. Despite this, the single-user waves only account for 4M of the 14M total deltas. Clearly, the multiuser waves are where the action is happening.

Table 5 describes how frequently deltas arrive from the same user. Our analysis shows that over half of the updates a user sends on a wave are within less than a second of the prior update the user sent on the same wave. This implies that spliced signatures would see some performance benefits from the message locality.

Simulation traces From our dataset of wave deltas, we need to construct a trace of messages to drive our algorithms. This means we must track both when a delta is generated and when it is verified. The former is already in the Google dataset, but the latter is one of the areas where we have some flexibility. Also, if a message has multiple recipients, we must generate events for every recipient. This process expands our original 14M message trace to 34M messages.

We simulated spliced signatures and Merkle signatures on a trace of Wave messages. Our simulations operate in a realtime fashion. Each message includes a timestamp. We inject messages into the asynchronous signing or verifying queue at the timestamp given in that message. It takes one hour to simulate a one-hour trace. The original dataset, if we were to run it in realtime following the timestamps, would not come anywhere near saturating the CPU of our test platform. Since our research wishes to consider what happens when a server reaches CPU saturation, we needed to artificially increase the message rate.

To create our final simulation traces, for each wave and each user, we determined in which hour we first saw a message for that wave or from that user, then extracted subsequent messages from that user. We then shifted this per-user trace by an integral number of hours, creating a new "virtual" user, distinct from the original user, doing the same things but effectively operating in a different timezone. This processing has the effect of generating a higher peak load and more concurrent editing. We use the first hour of this data as our *peak load trace*. It contains 3.8M messages from 47k users.

	8x	16x	32x
Signing trace	4.43M	8.45M	18.46M
Verification trace	2.16M	4.18M	9.01M

Table 6: Number of messages in the signing and verification traces for different replication factors.

6.4 Generating traffic

In our simulation, we want to model scenarios where users can log in and log off. While a user is logged-in, we immediately verify their messages. When a user logs in after an absence, we force any of their buffered, unverified messages to be immediately verified. Unfortunately, our trace does not include user login and logoff times. We inferred these times by assuming that a user was logged in for one minute before and after every message they authored and otherwise logged off. In our peak dataset, 5k users only send a single message and are treated as if they logged in for exactly 2 minutes and half of the users are logged in for 5 minutes or less.

Next, to model a group of Wave servers exchanging message traffic with one another, we must assign our simulated users and our simulated waves to distinct Wave servers acting as their homes. We randomly assign users to one of 8 servers, $S_0 \ldots S_7$. To compensate for the reduced load of dividing the users among 8 servers, we replicate the peak load trace making 8, 16, or 32 copies, assigning new user and wave IDs in the replicated data. We displace the copies by adding or subtracting up to one hour. These master traces are then used to derive our simulation traces.

We derived two sets of simulation traces. Our *signing trace* includes only messages sent from server S_0. Our *verification trace* includes messages received by server S_0 from all other servers. This trace is much smaller because it excludes updates by only one user, where the sender and recipient are the same and do not need any signature verification. Table 6 gives the sizes of the different traces.

While we would have preferred to stretch our simulation to an Internet scale, with tens of thousands of servers talking to one another, we lack the computational resources to run such a simulation. Instead, by creating a trace that emulates the load on one server from many of its peers, we can still capture the behavior of servers in these larger environments, albeit at a smaller scale.

6.5 Benchmark: Signature generation

To model a signature generation benchmark, we take the messages from our signing trace and play them into the simulated Wave server's processing queue in real-time based on the message timestamps. The server's cryptography thread continuously loops, fetching all unprocessed messages from the queue, building the hash tree data structures, signing the root cryptographic hash, and then serializing signatures. This is done for both Merkle tree signatures and history tree spliced signatures.

Our results mirror what we saw with our microbenchmarks. Our trace data have an average throughput of 1k-5k messages/second, which is notably less than the peak rate we benchmarked in Section 5.2. The CPU time to perform the underlying RSA or DSA public key signature dominated the runtime and the signing latency. For RSA with spliced signatures, the average signing latency was 18.7–21.2ms. Merkle signatures were a millisecond faster. DSA signature latency was 1.9ms–2.4ms for both Merkle signatures and spliced signatures. (We did not test standalone digital signatures because even our lightest 8x signing trace exceeded the DSA and RSA signing rate of a single CPU core running flat out.)

6.6 Benchmark: Signature verification

Creating a signature verification benchmark is more complicated than doing it for signature generation. To simulate a server verifying messages requires having a signed message trace that contains actual signed messages.

The only way to create realistic patterns of message signature generation and verification is to emulate our users running across a number of different servers. Our original verification trace assigned the users authoring a message to one of 8 servers. Spliced signatures would have been very effective with so few signers, so we needed to divide users among more servers. We only had enough memory to emulate 65 different signers, so we assigned users to that many signers at random. Each signing server is assumed to generate a new batch every simulated 60ms if the signing algorithm was RSA and 5ms if the signing algorithm was DSA. These epoch lengths were chosen to correspond to the latencies we measured in the peak throughput reported in Table 3. If any messages targeted server S_0 in that epoch, we added that message and 16 junk messages to that server in that epoch, simulating messages targeting servers other than S_0. This procedure simulates a signer with a signing rate from 1k-12k messages/sec. Each signing system then generates Merkle or spliced signatures as described earlier.

The resulting signed messages and their simulated timestamps became our signed message trace, which we then ran through our verification benchmark. Our trace replay thread used those message timestamps and user login and logoff times to force verification of messages when needed. We note that the event rate and the times at which messages are forced to be verified are the same, regardless of which signature algorithm is being simulated.

In Table 7 we report the results of verifying our signed message traces. Merkle signatures exhibit the lowest scalability. Because they lack sufficient locality to amortize public key operations over many messages, they become overloaded when used with DSA for all three of our traces, and with RSA for our 16x and 32x traces.

In contrast, with spliced signatures, we see more forced messages than we see public key signature verifications. This means that, in our simulation traces, forced messages arrived faster than they could be processed. Each public key verification had to be amortized across several forced messages to avoid overload.

If we compare the 16x and 32x traces, the number of messages in the trace and the corresponding number of forced messages doubles, but the number of public-key signatures verified only increased by 20–60%. Our spliced signatures compensated for the increased verification load by amortizing each public key verification over more forced messages. This increased message throughput comes at the cost of an increase in verification latency, which grows 60-130%.

PK alg	Tree alg	Trace size	Avg. latency (ms)	Total msgs	Forced msgs	Total PK verifs
RSA	History	16x	10.1	4.18M	2.26M	1.66M
RSA	History	32x	16.0	9.01M	4.73M	2.66M
DSA	History	8x	17.2	2.16M	1.11M	0.93M
DSA	History	16x	30.1	4.18M	2.26M	1.59M
DSA	History	32x	71.7	9.01M	4.73M	1.82M
RSA	Merkle	8x	13.7	2.16M	1.11M	2.16M
RSA	Merkle	16x	*overloaded*			
DSA	Merkle	8x	*overloaded*			

Table 7: For our Wave trace-driven verification benchmark, we report the average verification latency, the number of messages verified, the number of messages in the trace where verification was forced, and the number of public key signatures performed for a variety of different signature algorithm configurations.

The performance we observe of spliced signatures is made possible by the degree of locality that we observe in our Wave dataset, as seen in Table 5, where half of the messages a user sends to a wave are within less than a second of the prior message and 95% are within 5 seconds. Our spliced signatures can exploit this locality to adapt to increasing load. While message latency does increase, the increased time is still quite reasonable and far better than having the server fail due to overloading. In a production implementation, we would expect a larger number of servers to participate (see Section 7), keeping the load on any individual server at a more reasonable level. when a server or server cluster's resources are more heavily taxed, spliced signatures gracefully degrade latency while still scaling throughput. This property makes them attractive for a variety of real-world tasks.

7 Scaling

In this paper, we presented benchmarks that only used one CPU core for all signing and verification. Many applications will clearly require more throughput than one CPU core can provide. Large services, like Google Wave, may implement a single logical service used concurrently by millions of users, require a clustered architecture to obtain the necessary scalability for running the application and supporting its associated cryptographic costs.

7.1 Improving the throughput of a single server

There are several ways to improve the throughput of a single server by using more than one CPU core, exploiting the immutability of Merkle trees and the append-only property of history trees.

Merkle trees are amenable to parallel computation throughout their construction. The hashes for each subtree are completely independent of one another, making it trivial to delegate their computation to a pool of threads on the same CPU. Furthermore, once the root hash has been computed, the task of computing a public-key signature on it can be handled independently. Furthermore, the hash tree is immutable and each pruned tree can be derived independently from it, allowing for high concurrency. The only inherently serial process in computing a Merkle tree is assigning each message to its location in the leaves of the Merkle tree, and even that could potentially be parallelized by having concurrent queues which feed into different subtrees.

Similarly, when verifying Merkle signatures, each message can be treated independently. A naïve implementation would verify the public key signature on each message and then verify the hashes. This process would be completely independent from one message to the next, allowing exceptional speedups through parallel computation. To reuse the expensive public key verifications across messages that share the same public key signature, a concurrent hash table or comparable structure would track which signatures have been verified. Lock contention on this structure would seem unlikely to be a significant issue, and the worst case is merely that two or more threads will concurrently verify the same public key signature.

Like Merkle signatures, history trees and spliced signatures can also be generated in parallel. Spliced signatures on each message can be verified independently from each other for exceptional speedup. The potential concurrency of verifying spliced signatures is reduced, however, if we increase CPU efficiency by exploiting the splices between the spliced signatures and improve the number of messages verified per public key signature verification. To do this, spliced signatures are partitioned into sets based on which history tree they were built from. Each set is independent and can be verified concurrently with other sets. Within each set, the bookkeeping and dependency graph management would happen serially.

7.2 Scaling to large computational clusters

Industrial deployments of scalable web services inevitably run on large clusters of servers in order to support large numbers of users and store vast quantities of data. We now consider how Merkle and history tree techniques might scale to work in such environments.

7.2.1 Scaling Merkle signatures

If an application is distributed across several servers, each node can compute its own Merkle trees, independently, allowing for exceptional scaling without requiring any changes to how an external observer would verify messages. However, if no one application server generates messages at the peak signing rates in Table 3, each node would still preferably dedicate a CPU core to signing outgoing messages in order to minimize latency. A clustered deployment could reduce the number of signing CPUs needed by running them closer to the peak Merkle signature signing rate, directing the traffic from many application nodes toward a single signing server.

Similarly, when verifying Merkle signatures, each message can be treated independently. A naïve implementation would verify the public key signature on each message and then verify the hashes. This process would be completely independent from one message to the next, allowing for exceptional speedups.

Unlike a smaller scale system, which is unlikely to encounter two messages that happened to be in the same batch, a large scale system will more often encounter messages that were signed in the same batch. To reuse the expensive public key verifications across messages that share the same public key signature, the verification hosts could run a distributed cache to track previously verified signatures.

7.2.2 Scaling spliced signatures

With spliced signatures, if a signing cluster is distributed across several servers, each of the cluster nodes can compute its own history tree, independently, allowing for exceptional scaling. However, such a design would yield a series of disjoint histories, making it impossible to splice signatures computed on one node to signatures computed on another. (Such a design might also reveal the size of the cluster on which the service is running, which could be undesirable.) Just like Merkle signatures, a clustered deployment could reduce the number of signing CPUs needed by running them closer to the peak spliced signature signing rate, directing the traffic from many application nodes toward a single signing server.

The highest efficiency comes when the signer generates spliced signatures with useful splices. For instance, in Google Wave, if all of a user's messages are signed by the same server, at least over the space of a few minutes, then all of the messages from that user would at least appear in the same history tree, allowing verifiers to benefit from the message locality if they were verifying multiple messages from that user.

Achieving high levels of concurrency for spliced signature verification is also feasible. As with Merkle trees, a naïve solution would allow spliced signatures to be verified scalably just like Merkle signatures by ignoring the splices and treating each message independently. A more sophisticated implementation would take advantage of the splices. Each spliced signature is a member of a history tree. A system can direct signatures with the same tree-id to the same host and different systems can verify different trees concurrently.

8 Conclusion

This research has shown that a variation on Merkle trees, called history trees, can be used to create a system allowing for efficient batch signature generation and verification. We leverage this hash-based data structure to allow individual messages to be verified, all by themselves if desired, and to include splices to other messages, allowing for significant throughput gains. Our implementation trades off latency for throughput; by waiting for larger batches to arrive, we can compute or verify expensive digital signatures over larger numbers of messages, ensuring that message throughput stays strong, even when the CPU is saturated. We experimentally verified our design against synthetic microbenchmarks as well as traces taken from Google's Wave service. Wave needs to generate signed events at the granularity of individual users' keystrokes, making efficiency essential, and our techniques maintain high verification throughput (over 12K message per second on a single CPU core, versus a fifth of that using standalone digital signatures) with acceptable latency (10-16 milliseconds).

References

[1] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In *International Conference on Information Security (ISC)*, pages 379–393, Seoul, Korea, Dec. 2001.

[2] D. Balfanz. *Streamauth library for signing message streams*, July 2009. http://code.google.com/p/streamauth/.

[3] A. Baxter, J. Bekmann, D. Berlin, J. Gregorio, S. Lassen, and S. Thorogood. *Google Wave Federation Protocol Over XMPP*, July 2009. http://www.waveprotocol.org/protocol/draft-protocol-specs/draft-protocol-spec.

[4] M. Bellare, J. A. Garay, and T. Rabin. Fast batch verification for modular exponentiation and digital signatures. In *EuroCrypt '98*, pages 236–250, 1998.

[5] D. J. Bernstein. RSA signatures and Rabin-Williams signatures: The state of the art. http://cr.yp.to/sigs.html, Jan. 2008.

[6] T. Berson, D. Dean, M. Franklin, D. Smetters, and M. Spreitzer. Cryptography as a network service. In *Proceedings of the 2001 Network and Distributed System Security Symposium (NDSS '01)*, San Diego, CA, Feb. 2001.

[7] K. Blibech and A. Gabillon. CHRONOS: An authenticated dictionary based on skip lists for timestamping systems. In *Workshop on Secure Web Services*, pages 84–90, Fairfax, VA, Nov. 2005.

[8] C. Boyd and C. Pavlovski. Attacking and repairing batch verification schemes. In *Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '00, pages 58–71, London, UK, 2000.

[9] J. Camenisch, S. Hohenberger, and M. O. Pedersen. Batch verification of short signatures. In *Proceedings of the 26th annual international conference on Advances in Cryptology*, EuroCrypt '07, pages 246–263, Barcelona, Spain, 2007.

[10] J. H. Cheon and J. H. Yi. Fast batch verification of multiple signatures. In *Proceedings of the 10th International Conference on Practice and Theory in Public-key Cryptography (PKC'07)*, pages 442–457, Beijing, China, 2007.

[11] S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, Aug. 2009.

[12] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol, Version 1.2*. IETF, RFC 5246, Aug. 2008. http://tools.ietf.org/search/rfc5246.

[13] A. L. Ferrara, M. Green, S. Hohenberger, and M. O. Pedersen. Practical short signature batch verification. In *Cryptographers' Track at the RSA Conference (CT-RSA '09)*, pages 309–324, San Francisco, CA, 2009.

[14] A. Fiat. Batch RSA. In *CRYPTO '89*, pages 175–185, Santa Barbara, CA, 1989.

[15] R. Gennaro and P. Rohatgi. How to sign digital streams. In *CRYPTO '97*, pages 180–197, Santa Barbara, CA, Aug. 1997.

[16] Google. *Protocol Buffers*, 2010. http://code.google.com/p/protobuf/.

[17] Google. *Wave Protocol*, 2010. http://www.waveprotocol.org/.

[18] L. Kissner and B. Laurie. *General Verifiable Federation*. Google, May 2009. http://www.waveprotocol.org/protocol/whitepapers/wave-protocol-verification.

[19] P. C. Kocher. On certificate revocation and validation. In *International Conference on Financial Cryptography (FC '98)*, pages 172–177, Anguilla, British West Indies, Feb. 1998.

[20] T. Korkmaz. Analyzing response time of batch signing. In *International Conference on Computer Communications and Networks*, pages 1–6, San Francisco, CA, Aug. 2009.

[21] S. Lassen and S. Thorogood. *Google Wave Federation Architecture*, May 2009. http://www.waveprotocol.org/whitepapers/google-wave-architecture.

[22] Legion of the Bouncy Castle. *Bouncy Castle Crypto API*, 2010. http://www.bouncycastle.org/.

[23] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *USENIX Security Symposium*, San Francisco, CA, Aug. 2002.

[24] R. C. Merkle. A certified digital signature. In *CRYPTO '89*, pages 218–238, Santa Barbara, CA, 1989.

[25] D. M'Raïhi and D. Naccache. Batch exponentiation: a fast DLP-based signature generation strategy. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security (CCS '96)*, pages 58–61, New Delhi, India, 1996.

[26] D. Naccache, D. M'Raïhi, S. Vaudenay, and D. Raphaeli. Can DSA be improved? Complexity trade-offs with the digital signature standard. In *EuroCrypt '94*, pages 77–85, Perugia, Italy, May 1994.

[27] National Institute for Standards and Technology. *NIST Special Publication 800-57: Recommendation for Key Management — Part 1: General*, Mar. 2007.

[28] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *ACM Symposium on User Interface and Software Technology (UIST '95)*, pages 111–120, Pittsburgh, PA, 1995.

[29] Novell. *Vibe*, 2010. https://vibe.novell.com/.

[30] J. M. Park, E. K. P. Chong, and H. J. Siegel. Efficient multicast stream authentication using erasure codes. *ACM Transactions on Information Systems Security*, 6:258–285, May 2003.

[31] C. Pavlovski and C. Boyd. Efficient batch signature generation using tree structures. In *International Workshop on Cryptographic Techniques and E-Commerce*, Hong Kong, July 1999.

[32] A. Perrig, R. Canetti, D. Song, and J. Tygar. Efficient and secure source authentication for multicast. In *Network and Distributed System Security Symposium (NDSS'01)*, pages 35–46, San Diego, CA, Feb. 2001.

[33] A. Perrig, J. D. Tygar, D. Song, and R. Canetti. Efficient authentication and signing of multicast streams over lossy channels. In *IEEE Symposium on Security and Privacy*, pages 56–73, Berkeley, CA, May 2000.

[34] M. O. Rabin. *Digitalized signatures and public-key functions as intractable as factorization*. Massachusetts Institute of Technology, Cambridge, MA, 1979.

[35] SAP. *StreamWork*, 2010. http://www.sapstreamwork.com/.

[36] B. Schneier and J. Kelsey. Automatic event-stream notarization using digital signatures. In *Security Protocols Workshop*, pages 155–169, Cambridge, UK, Apr. 1996.

[37] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 1(3), 1999.

[38] H. Shacham and D. Boneh. Improving SSL handshake performance via batching. In *Cryptographers' Track at the RSA Conference (CT-RSA '01)*, pages 28–43, Apr. 2001.

[39] J. Tirsen. *Access Control in Google Wave*. Google, May 2009. http://www.waveprotocol.org/whitepapers/access-control.

[40] C. K. Wong and S. S. Lam. Digital signatures for flows and multicasts. *IEEE/ACM Trans. Netw.*, 7:502–513, Aug. 1999.