Real-World Performance of Cryptographic Accumulators

Edward Tremel

Spring 2013

Abstract

Cryptographic accumulators have often been proposed for use in security protocols, and the theoretical runtimes of algorithms using them have been shown to be reasonably efficient, but their performance in the real world has rarely been measured. In this paper I analyze the performance differences between two cryptographic accumulator constructions, RSA accumulators and bilinear-map accumulators, based on a realistic practical implementation in C++. I first discuss the theoretical differences between the constructions and their runtimes, showing that both algorithms present the opportunity for parallel computation. Then I describe an experiment that measures the actual running time of these algorithms on current commodity hardware, and discuss the optimizations I was actually able to make in their code. Finally, I present and analyze the experimental results, which show that the bilinear-map accumulator performs faster than the RSA accumulator in almost all cases, and should be the preferred implementation for practical security systems as long as the size of the set to be accumulated can be given a reasonable upper bound.

1 Introduction

One-way accumulators are an important cryptographic primitive that form the basis of a large number of security systems. Similar to a one-way hash function, they provide a fixed-size digest representing an arbitrarily large set of inputs. More interestingly, a one-way accumulator can provide a fixed-size witness for any element of the set, which can be used in combination with the accumulated digest to verify that element's membership in the set. As a result of this ability to efficiently verify set membership, cryptographic accumulators have been used in security applications that require some form of authentication as an alternative to digital signatures. For example, accumulators are a component of anonymous credential systems, as in [14], e-cash schemes, as in [1], and authenticated data structures, as in [20].

Most existing work describing security algorithms that use accumulators, however, considers the performance of these algorithms only in the theoretical sense. Few publications include a working software implementation of their accumulator-based security system, describing the algorithms only in pseudocode. Unlike, for example, one-way hash functions, the average real-world performance and useability of cryptographic accumulators is not generally known, since they have not been widely adopted in current computer systems. The fact that accumulators have good asymptotic performance does not guarantee that they will be practical to use in real life, since the constants involved in their constant-time operations may turn out to be very large. In order for security schemes such as anonymous credential systems to be adopted by real users, it is important to determine whether accumulators demonstrate reasonably fast performance.

Another open question in the area of cryptographic accumulators is the choice of which version of the accumulator construct should be used when implementing a security system based on accumulators. Two different protocols have been described that fulfill the basic contract of a cryptographic accumulator, which are commonly referred to as the RSA accumulator and the bilinear-map accumulator. The RSA accumulator was first described by Beneloh and de Mare [5], when they introduced the concept of a cryptographic accumulator, and it was further developed and formalized by Barić and Pfitzmann [2]. The bilinear-map accumulator was introduced by Nguyen [19], and uses elliptic curve operations instead of modular exponentiation as the basis for computing accumulation values. While there are some theoretical differences

between these accumulators, they have largely the same capabilities, and both have been extended to dynamic accumulators (RSA accumulators in [9] and bilinear-map accumulators in [8]). In many constructions, such as authenticated data structures, they could be used interchangeably because they both provide the needed features, e.g. proving set membership. It then becomes important to consider which one has one provides faster performance when actually implemented, since the choice of a slow accumulator could make an otherwise efficient authenticated data structure impractical for use in real life.

This paper will therefore focus on benchmarking the performance of a concrete implementation of both an RSA accumulator and a bilinear-map accumulator, comparing the results to determine which accumulator performs faster. In Section 2 I provide the formal definitions of the two types of accumulators I am considering. In Section 3 I discuss the algorithms involved in using these accumulators from a theoretical perspective. This includes a discussion of the parts of each accumulator that can be computed in parallel, an optimization opportunity often overlooked in considering how security algorithms can be implemented on modern multicore processors. In Section 4 I describe the setup of my experiment, including implementation details of the accumulators and optimizations I was able to make to the accumulator algorithms. Section 5 contains the results of my experiments, and in section 6 I comment on these results and conclude that the bilinear-map accumulator is much faster than the RSA accumulator, placing it in the realm of practical runtimes.

1.1 Related Work

Little previous work has been done on measuring the real-world performance of cryptographic accumulators. The most potentially relevant work is an extensive study of the real-world performance of authenticated dictionary schemes by Crosby and Wallach [11]. In this paper, the authors use an authenticated dictionary based on RSA accumulators as one of their test systems, and conclude that the RSA accumulators introduce a significant performance overhead that make this authenticated dictionary too slow for practical use compared to digital signatures. However, the results do not show the specific amounts of time taken by RSA accumulator operations because they are formatted in terms of authenticated dictionary operations such as inserts and updates. Furthermore, the tests were all carried out on a single-core processor, so none of the available concurrency within the RSA accumulator's algorithms was exploited.

2 Background

The concept of one-way accumulators was first introduced by Benaloh and de Mare [5], who defined them as one-way hash functions with the property of being *quasi-commutative*. A quasi-commutative function is a function $f : X \times Y \to X$ such that, for all $x \in X$ and for all $y_1, y_2 \in Y$

$$f(f(x, y_1), y_2) = f(f(x, y_2), y_1) \tag{1}$$

If this function is also a one-way hash function, i.e. it is difficult for a polynomially-bounded adversary to invert, then it is a one-way accumulator. A one-way accumulator function h can thus be used to compute a secure digest z for a set of values $\{y_1, y_2, \ldots, y_m\} \in Y$ given a starting value x by applying h repeatedly to each y_i, and this value does not depend on the order in which the y_i are accumulated. It can also be used to generate a *witness* z_j for a value y_j in the set, by accumulating all y_i such that $i \neq j$. Since the order of accumulation does not matter, the only difference between z_j and z is that z_j has not yet accumulated y_j, so $h(z_j, y_j) = z$.

Barić and Pfitzmann [2] generalized the definition of an accumulator to any set of functions that can, given a security parameter k,

- Generate an *accumulator key* that can be used in all other functions
- Compute an accumulation value z for a set $\{y_1, y_2, \ldots, y_m\}$
- Compute a witness value w_i for an element y_i in the set, with respect to z

- Authenticate an element y_i using witness w_i and accumulation z.

They also introduced the concept of collision-free accumulators, a stronger guarantee than one-way accumulators, and note that Benaloh and de Mare's original implementation of accumulators with modular exponentiation is not collision-free. Briefly, an accumulator is collision-free if, for all set sizes N, it is difficult for a probabilistic polynomial-time adversary to find a set $\{y_1, \ldots, y_N\}$, a value y' (not in the set) and a witness w' such that y' is authenticated by w' and the accumulation value z for $\{y_1, \ldots, y_N\}$.

There are two different implementations of accumulators that satisfy Barić and Pfitzmann's definition. The first, RSA accumulators, were described by Barić and Pfitzmann themselves in the same paper in which they gave this definition.

2.1 The RSA Accumulator

The RSA accumulator is based on modular exponentiation with an RSA modulus. In its simplest form, it works as follows. The accumulator key is an RSA modulus $\mathbf{N} = pq$, where p and q are strong primes [17], and a base $x \in \mathbb{Z}_{\mathbf{N}}$. The modulus should be at least k bits, where k is the number of bits in the largest element that will be accumulated. The accumulation function computes the accumulation value for a set $\mathcal{P} = \{p_1, \ldots, p_n\}$ of prime numbers as

$$\mathsf{acc}(\mathcal{P}) = x^{p_1 \cdots p_n} \mod \mathbf{N} \tag{2}$$

The witness-generation function computes the witness $\mathsf{W}_{p_i,\mathcal{P}}$ for element p_i in $\mathcal{P}$ by accumulating all elements of $\mathcal{P}$ except p_i:

$$\mathsf{W}_{p_i,\mathcal{P}} = x^{p_1 \cdots p_{i-1} p_{i+1} \cdots p_n} \mod \mathbf{N} \tag{3}$$

Finally, the authentication function authenticates an element p_i and a witness $\mathsf{W}_{p_i,\mathcal{P}}$ with an accumulation $\mathsf{acc}(\mathcal{P})$ by testing

$$(\mathsf{W}_{p_i,\mathcal{P}})^{p_i} \stackrel{?}{\equiv} \mathsf{acc}(\mathcal{P}) \mod \mathbf{N} \tag{4}$$

Prime Representatives. It is important to note that the inputs to this accumulator must be restricted to prime numbers in order for it to be collision-free. Since most practical uses of accumulators need to be able to accumulate arbitrary integer values, it is necessary to compute a *prime representative* of each desired input to use as the actual input for the RSA accumulator.

One method of computing prime representatives, proposed by Sander, Ta-Shma, and Yung in [23] and described by Goodrich, Tamassia, and Hasić in [15], is based on two-universal hash functions (introduced by Carter and Wegman in [10]). It involves defining a two-universal function $h(x) = Fx$, where F is a $k \times 3k$ binary matrix, and searching for a prime $3k$-bit preimage of a k-bit element e by sampling $O(k^2)$ times from the set of inverses $h^{-1}(e)$. However, this method generates very large prime representatives and performs slowly in practice. In my experiments, I use a more practical method of computing prime representatives, which is more efficient and produces smaller representatives but is slightly less secure because it relies on the random oracle model.

The second method of computing prime representatives was described by Barić and Pfitzmann, also in [2]. They refer to it as the "RSA Accumulator with Random Oracle," but it is essentially the same as the standard RSA accumulator with a random oracle prime representative generator. Let $\Omega(y)$ be a random oracle; on input y, it returns a random number r and stores the pair (y, r), and if it receives y as input again it returns the same r. Using this oracle, the prime representative of composite element y is $2^t\Omega(y) + d$, where d is a t-bit number that, when appended to $\Omega(y)$, makes it prime. As Papamanthou, Tamassia, and Triandopoulos showed in [21], if a is the output of a b-bit random oracle, the interval $[2^t a, 2^t a + 2^t - 1]$ contains a prime with probability at least $1 - 2^{-b}$ provided $b \leq \lfloor \log(1 + \sqrt{2^t + 4e^{2^t - 1}}) - 1 \rfloor$. Therefore if t is of sufficient size given the size of the oracle's output, d can be found with high probability by incrementing from 1 to $2^t - 1$ until $2^t\Omega(y) + d$ is prime. This method can produce prime representatives of a fixed size regardless of the size (k) of the elements to be accumulated.

A more complete definition of the RSA accumulator, taking into account prime representatives, is the following. Suppose we have a set of k-bit elements $\mathcal{A}$, and a function $\mathbf{r}(a)$ for computing j-bit prime representatives for elements in $\mathcal{A}$. Let $\mathbf{N}$ be a k'-bit RSA modulus ($k' > j$), namely $\mathbf{N} = pq$, where p, q are strong primes [17]. The accumulation value of $\mathcal{A}$ is

$$\mathsf{acc}(\mathcal{A}) = g^{\prod_{a \in \mathcal{A}} \mathbf{r}(a)} \mod \mathbf{N}, \tag{5}$$

which is a k'-bit integer, where $g \in \mathbb{QR}_{\mathbf{N}}$. The RSA modulus $\mathbf{N}$, the exponentiation base g, and the prime representative generator $\mathbf{r}$ comprise the public key pk for the accumulator. The factorization of $\mathbf{N}$ is the accumulator's secret key and is kept hidden from any adversary. Subject to the accumulation $\mathsf{acc}(\mathcal{A})$, the witness for element $a_i \in A$ is

$$\mathsf{W}_{a_i,\mathcal{A}} = g^{\prod_{a \in \mathcal{A}-a_i} \mathbf{r}(a)} \mod \mathbf{N}. \tag{6}$$

More generally, the proof of *subset containment* for any set $\mathcal{B} \subseteq \mathcal{A}$ [22] is the subset witness $\mathsf{W}_{\mathcal{B},\mathcal{A}}$ where

$$\mathsf{W}_{\mathcal{B},\mathcal{A}} = g^{\prod_{a \in \mathcal{A}-\mathcal{B}} \mathbf{r}(a)} \mod \mathbf{N}. \tag{7}$$

Subset containment for a set $\mathcal{B}$ in set $\mathcal{A}$ can be authenticated by testing

$$\mathsf{W}_{\mathcal{B},\mathcal{A}}^{\prod_{b \in \mathcal{B}} \mathbf{r}(b)} \stackrel{?}{\equiv} \mathsf{acc}(\mathcal{A}) \mod \mathbf{N}, \tag{8}$$

which can be done by any verifier that has access to the correct accumulation value $\mathsf{acc}(\mathcal{A})$ and the public key.

This accumulator is collision-free under the Strong RSA Assumption, which was also defined by Barić and Pfitzmann:

Assumption 1 (Strong RSA assumption [2]) *Let k be the security parameter. Given a k-bit RSA modulus $\mathbf{N}$ and a random element $x \in \mathbb{Z}_{\mathbf{N}}^*$, there is no probabilistic polynomial-time algorithm that outputs $y > 1$ and β such that $\beta^y = x \mod \mathbf{N}$, except with probability $\mathsf{neg}(k)$[1].*

2.2 The Bilinear-Map Accumulator

The second well-known implementation of accumulators is the bilinear-map accumulator, which was first introduced by Nguyen in [19]. Nguyen describes them in terms of additive groups, but most subsequent work with them describes them in terms of multiplicative groups. I will be using the multiplicative group definition, as presented in e.g. [21] and [13], since it better shows the parallels with the RSA accumulator. First, it is necessary to define bilinear pairings. Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two cyclic multiplicative groups of prime order p generated by g_1 and g_2, for which there exists an isomorphism $\psi : \mathbb{G}_2 \to \mathbb{G}_1$ such that $\psi(g_2) = g_1$. If $\mathbb{G}_T$ is a cyclic multiplicative group with the same order p, then $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear pairing (or bilinear map) with the following properties:

1. Bilinearity: $e(P^a, Q^b) = e(P, Q)^{ab}$ for all $P \in \mathbb{G}_1, Q \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p^*$
2. Non-degeneracy: $e(g_1, g_2) \neq 1$
3. Computability: There is an efficient algorithm to compute $e(P, Q)$ for all $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$.

The groups and the pairing can be chosen by a *bilinear pairing instance generator*, which is a probabilistic polynomial time algorithm that takes a security parameter k and produces a tuple $\mathsf{t} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$ such that p grows exponentially with k. Most descriptions of the bilinear-map accumulator set $\mathbb{G}_1 = \mathbb{G}_2$, but in my experimental setup I keep them separate, since pairing functions are actually faster to compute when the input groups are distinct [18].

[1] Function $f : \mathbb{N} \to \mathbb{R}$ is $\mathsf{neg}(k)$ iff for any nonzero polynomial $p(k)$ there exists N such that for all $k > N$, $f(k) < 1/p(k)$.

Given an instance of a bilinear pairing, the bilinear-map accumulator is constructed as follows. Suppose we have a set of n elements $\mathcal{E} = \{e_1, e_2, \ldots, e_n\}$, all of which are in $\mathbb{Z}_p^*$ (where p is the prime order of the groups). Let s be a value randomly chosen from $\mathbb{Z}_p^*$. Then the accumulation value of $\mathcal{E}$ is

$$\mathsf{acc}(\mathcal{E}) = g_1^{(e_1+s)(e_2+s)\cdots(e_n+s)}, \tag{9}$$

which is an element of $\mathbb{G}_1$. Note that the exponent of g_1 can be seen as a polynomial on s of degree n, that is, $f_{\mathcal{E}}(s) = \prod_{e\in\mathcal{E}}(e+s)$. The witness for element $e_i \in \mathcal{E}$ with respect to the accumulator $\mathsf{acc}(\mathcal{E})$ is

$$\mathsf{W}_{e_i,\mathcal{E}} = g_2^{\prod_{e_j\in\mathcal{E}:e_j\neq e_i}(e_j+s)} \tag{10}$$

The exponent of g_2 here is also a polynomial on s, of degree $n-1$, and can be defined as $f'_{\mathcal{E},e_i}(s) = \frac{f_{\mathcal{E}}(s)}{e_i+s}$. The value s is the accumulator's secret key sk, and the set $\{g_1^{s^i}, g_2^{s^i} : 0 \leq i \leq q\}$ is the accumulator's public key pk, where q is an upper bound on n. A verifier with access only to pk, the correct accumulation value $\mathsf{acc}(\mathcal{E})$, and the bilinear pairing instance can authenticate element e_i by testing

$$e(g_1^{e_i} \cdot g_1^s, \mathsf{W}_{e_i,\mathcal{E}}) \stackrel{?}{=} e(\mathsf{acc}(\mathcal{E}), g_2) \tag{11}$$

since g_1, g_1^s, and g_2 are part of pk. This is mathematically equivalent to

$$\mathsf{W}_{e_i,\mathcal{E}}^{(e_i+s)} \stackrel{?}{=} \mathsf{acc}(\mathcal{E}) \tag{12}$$

but can be done without knowledge of s. Note that unlike the RSA accumulator, prime representatives are not required at any point; this accumulator can accept as input any integer less than p.

This accumulator is collision-free under the Strong Diffie-Hellman Assumption, which was introduced by Boneh and Boyen:

Assumption 2 (q-Strong Diffie-Hellman Assumption [7]) *Let G be a cyclic group of prime order p generated by g, and let $\kappa \in \mathbb{Z}_p^*$. Any probabilistic polynomial-time algorithm A that is given set $\{g^{\kappa^i} : 0 \leq i \leq q\}$ can find a pair $(x, g^{\frac{1}{x+\kappa}}) \in \mathbb{Z}_p^* \times G$ with probability at most $O(1/p)$.*

3 Accumulator Algorithms

Given these mathematical definitions, I will now consider the algorithms involved in using each type of accumulator. As they are defined above, both kinds of accumulators have a public key, pk, and a secret key, sk. Although it may not be apparent from the definition, in both accumulators there is a significant difference between computing an accumulation value or witness with access to the secret key and computing the same values with access to only the public key.

Consider the RSA accumulator. With access to the secret key $\mathsf{sk} = \{p, q\}$, i.e. the factorization of $\mathbf{N}$, it is possible to compute the totient $\phi(\mathbf{N}) = (p-1)(q-1)$. As a result of Euler's Totient Theorem, the exponent of g in the accumulation value is equivalent to $\prod_{a\in\mathcal{A}} \mathbf{r}(a) \mod \phi(\mathbf{N})$, so with knowledge of the totient it feasible to compute the entire exponent by reducing mod $\phi(\mathbf{N})$ after each multiplication. Only one modular exponentiation then needs to be done. Without sk, however, $\phi(\mathbf{N})$ remains unknown, so the exponent would need to be computed as an unbounded integer in order to compute its value all at once before doing the modular exponentiation. This quickly becomes infeasible as the size of $\mathcal{A}$ increases, because the product of $|\mathcal{A}|$ j-bit integers can be up to $j|\mathcal{A}|$ bits. Computing the accumulation value with only the public key therefore requires computing $g^{\prod_{a\in\mathcal{A}} \mathbf{r}(a)} \mod \mathbf{N}$ as $(((g^{\mathbf{r}(a_1)})^{\mathbf{r}(a_2)})^{\mathbf{r}(a_3)})^{\cdots} \mod \mathbf{N}$, doing a modular exponentiation for each element of $\mathcal{A}$. The same argument applies to computing a witness value, since the computation is the same except for the one element that is excluded from the exponent's product. Computing a witness value with access to sk can be done by computing the product in the exponent directly, reducing mod $\phi(\mathbf{N})$, while computing the same witness value with only pk requires turning the product into a series of modular exponentiations.

Now consider the bilinear-map accumulator. With the secret key $\mathsf{sk} = s$, the exponent of g_1 can be computed directly, using addition and multiplication mod p (operations in the exponent of group elements are always modulo the group order). Thus computing the accumulation value requires only a single group exponentiation operation. On the other hand, without sk, the exponent must be treated as a polynomial on s, which cannot be evaluated directly. Instead it is necessary to find the coefficients of this polynomial and use each coefficient as the power of a public-key element, since pk contains g_1 raised to powers of s. If $\{c_0, c_1, \ldots, c_n\}$ are the coefficients of $f_{\mathcal{E}}(s)$ in ascending order (i.e. c_1 is the coefficient of the s term), the accumulation value $\mathsf{acc}(\mathcal{E})$ is computed as $g_1^{c_0} \cdot (g_1^s)^{c_1} \cdot (g_1^{s^2})^{c_2} \cdots (g_1^{s^n})^{c_n}$. Due to the rules of exponentiation this is equivalent to evaluating $f_{\mathcal{E}}(s)$, but it requires a group multiplication and exponentiation operation for each coefficient of the polynomial, which is generally more computationally expensive than modular arithmetic. The same argument applies to computing a witness value: with sk, the desired exponent of g_2 can be computed directly, but without it the polynomial must be evaluated by pairing its coefficients with elements of pk. If $\{c_0, c_1, \ldots, c_n\}$ are the coefficients of $f'_{\mathcal{E},e_i}(s)$ in ascending order, the witness value $\mathsf{W}_{e_i,\mathcal{E}}$ is computed as $g_2^{c_0} \cdot (g_2^s)^{c_1} \cdot (g_2^{s^2})^{c_2} \cdots (g_2^{s^n})^{c_n}$.

These differences are very important in practice, as we will see in the experimental results. However, in theory, both accumulators still perform a constant number of operations per element to be accumulated, so for both accumulators the runtime of the "accumulate" function for a set of size n is $O(n)$ regardless of whether pk or sk is used. The runtime of the witness-generation function for both accumulators is also $O(n)$, and the runtime of the verification function for both accumulators is $O(1)$ (since verifying a single element against a witness requires a fixed number of operations and a comparison). In fact, the only algorithm for which these accumulators have different runtimes is the key-generation algorithm. Key generation for the RSA accumulator involves choosing an RSA modulus and an exponentiation base, and possibly generating a matrix for a two-universal hash function. Both of these operations are polynomial in k, the size in bits of an element of the set to be accumulated (this affects the number of probabilistic primality tests needed and the number of matrix bits that must be generated), but they are constant with respect to the size of the set (n). On the other hand, key generation for the bilinear-map accumulator involves generating the set $\{g_1^{s^i}, g_2^{s^i} : 0 \leq i \leq q\}$, which is $O(q)$. Since q must be an upper bound on the size of the set to be accumulated, this is affected by the size of the set to be accumulated; the runtime of key generation must be at least $O(n)$.

It would appear, as a result of the last conclusion, that the bilinear-map accumulator is less desirable than the RSA accumulator, because it has an asymptotically worse runtime for key generation. However, as the experimental results will show, this disadvantage is usually outweighed by the fact that the linear-time algorithms of the bilinear-map accumulator have much smaller constants than the RSA accumulator.

3.1 Parallel Computation

Another important aspect of the accumulator algorithms to consider is the degree to which they can be computed in parallel. Almost all computers in use today have multi-core processors, so any algorithm that can be parallelized will be able to exploit hardware concurrency to gain improved performance. Both kinds of accumulators have some easily available concurrency in their algorithms, which I will now describe.

RSA Accumulator. Each element in the set accumulated by the RSA accumulator needs a prime representative. However, the representatives do not need to be computed on-the-fly as the accumulation value is computed; they can be pre-computed and stored by the entity computing the accumulation values (although they will not be available to the verifier). Computing a set of prime representatives $\{r_1, \ldots, r_n\}$ for a set of composite values $\{a_1, \ldots, a_n\}$ can easily be done in parallel, since each $r_i = \mathbf{r}(a_i)$ can be computed independently of all the others. There are thus n independent tasks that can be performed in parallel by up to n threads, allowing the entire set to be computed in $O(\frac{n}{c})$, where c is the number of processor cores available.

When computing an accumulation value with access to the private key, the majority of the work is in computing the product $\prod_{a \in \mathcal{A}} \mathbf{r}(a) \mod \phi(\mathbf{N})$, with one modular exponentiation at the end. Since modular multiplication is associative, this product can be computed with the parallel prefix sum algorithm [6], a technique for mapping an associative operation over a set using a tree-organized grouping of operations. The

parallel prefix sum algorithm can compute the product of the entire set in $O(\frac{n}{c}+\log c)$ [6], which is dominated by $\frac{n}{c}$ when $\frac{n}{c} > \log c$. The same can be said for computing a single witness value with access to the private key, since that also involves modular multiplication of (almost) the entire set of prime representatives.

When computing accumulation values or witnesses with only the public key, parallel computation is not feasible because modular exponentiation is not associative – computing the value $(g^{r_1})^{r_2}$ in a right-associative manner would require reducing the power $r_1^{r_2} \bmod \phi(\mathbf{N})$, since it is to be used as an exponent for $g \bmod \mathbf{N}$. However, it is often the case that an entity needing to compute $\mathsf{acc}(\mathcal{A})$ also needs to compute a witness $\mathsf{W}_{a_i,\mathcal{A}}$ for every element of $\mathcal{A}$. In this case, the task of computing all of the witnesses can be easily parallelized, since computing each witness does not depend on any other witness and requires read-only access to the set of prime representatives of elements. As with prime representative generation, this means there are up to n independent tasks that can be performed in separate threads, so computing all n witnesses for a set of size n will take $O(\frac{n^2}{c})$ time.

Verification of elements with witnesses is already constant-time, but it may be worth noting that verifying a large number of elements could easily be done in parallel, since each witness-verification task would require only read access to the shared accumulator value.

Bilinear-Map Accumulator. Computing the public key for the bilinear-map accumulator requires two sets of exponentiations with the same powers of s: one to compute $g_1, g_1^s, g_1^{s^2}, \ldots, g_1^{s^q}$, and one to compute $g_2, g_2^s, g_2^{s^2}, \ldots, g_2^{s^q}$. These powers of s can first be computed in parallel using the parallel prefix sum algorithm, using as input a set of q copies of s and the multiplication operation, which will take $O(\frac{q}{c} + \log c)$. (The parallel prefix sum algorithm can also be used to compute all prefix sums of a set, not just the sum of the entire set). Then each of the group element exponentiations can be computed in parallel, with all threads sharing read-only access to the set of powers of s. There are up to $2q$ independent tasks, so this step will take $O(\frac{q}{c})$, and the total runtime of computing the public key is $O(\frac{q}{c} + \log c)$, where c is the number of processor cores available.

Similar to the RSA accumulator, computing an accumulation value or witness with access to the private key involves computing the modular product of a set of n (or $n-1$) values. This product can be computed with the parallel prefix sum algorithm in $O(\frac{n}{c} + \log c)$.

Computing an accumulation value with the public key also presents an opportunity for parallel computation because it involves a series of associative multiplications. First the product $\prod_{e\in\mathcal{E}}(e+s)$ must be turned into a set of polynomial coefficients by multiplying out the binomials. Polynomial multiplication could feasibly implmented as an associative operation, i.e. in a library where polynomials are represented with an encapsulated data type that does not care about term order, and a single-threaded library function uses Fast Fourier Transform to multiply two polynomial objects. In that case, finding the polynomial coefficients can be accomplished with the parallel prefix sum algorithm in $O(\frac{n}{c} + \log c)$. Once the coefficients of the polynomial representing the set to be accumulated have been computed, the accumulation value is $g_1^{c_0} \cdot (g_1^s)^{c_1} \cdot (g_1^{s^2})^{c_2} \cdots (g_1^{s^n})^{c_n}$. The group multiplication operation is associative, so this product can also be computed with the parallel prefix sum algorithm, with an additional step in which each exponent $(g_1^{s^i})^{c_i}$ is evaluated in parallel before the multiplication operations begin. The runtime of computing the product would then be $O(\frac{n}{c} + \log c)$ (acknowledging a larger constant term than usual for parallel prefix sum because of the exponentiation step), and the total runtime of computing an accumulation value is also $O(\frac{n}{c} + \log c)$ because the two steps have the same asymptotic runtime.

The same argument can be made for computing a witness value with the public key, since it is almost the same as computing an accumulation value, differing only by the one element that is not included in the polynomial. Thus computing a single witness takes $O(\frac{n}{c} + \log c)$. As with RSA accumulators, there is also parallelism in the common case where a witness must be computed for every element in the set $\mathcal{E}$, because each witness can be computed in its own thread. However, since computing a single witness already uses multiple threads, this would only provide an advantage when $c > \frac{n}{2}$, i.e. there are idle threads leftover after using the optimal number for the parallel prefix sum algorithm.

As with RSA accumulators, it may be worth noting that verifying a large batch of elements and witnesses could be done in parallel, since the shared accumulation value and public key elements are not modified by computing a verification.

4 Experimental Setup

Given this theoretical background, my research experiment was to measure and compare the actual running times of these accumulators as they performed each component function. In order to get the best possible performance, I implemented both accumulators in C++, using the fastest available libraries that were also thread-safe. For modular arithmetic over large integers (needed by both accumulators) and the polynomial operations needed by bilinear-map accumulators, I used FLINT [16], an open source C library for number theory computations. For the RSA modulus generation and SHA-256 hashing needed by the RSA accumulator, I used Crypto++ [12], a thread-safe C++ library implementing various cryptographic operations (the standard C library for cryptography, OpenSSL, is not thread-safe). Finally, I used DCLXVI [18], the fastest available library for elliptic curve computations (written in C), to implement the bilinear maps and cyclic groups needed for the bilinear-map accumulator.

I set up benchmark tests of these implementations to measure three different factors of their performance: how running time changes as the size of the input set changes, how running time changes as the number of available hardware threads changes, and the difference in running time between equivalent operations in the two types of accumulators. Each test was further broken down into seven parts, corresponding to the functions available in each accumulator:

1. Public/private key generation
2. Prime representative generation
3. Accumulation of a set using the private key
4. Accumulation of a set using the public key
5. Witness generation for each element in the set using the private key
6. Witness generation for each element in the set using the public key
7. Verification of elements with witnesses

Unfortunately, it turned out to be infeasible to run part 6 of this test on sets larger than 10,000 elements, or with fewer than 2 threads, because public-key-only witness generation can be several orders of magnitude slower than any other operation in the test. This is the reason why results for this test are not shown in all of the graphs below. The sets used as inputs for the accumulators were random sets of 256-bit numbers, which I pre-generated and saved for each desired set size from 1000 to 100,000 elements. Since DCLXVI and FLINT use different data formats to represent integers, the input format for the two accumulators was slightly different, and I had to generate separate random sets for each accumulator.

The tests were run on a 64-bit 3.4 GHz Intel Core i7 machine, with 8 hardware threads, an 8 MB cache, and 16 GB of RAM, running Sabayon Linux. To run the tests involving fewer threads, some processor cores were disabled using Linux kernel commands. The code was compiled using `g++` version 4.7 in C++11 mode, with native architecture tuning and optimization level 3 enabled. Libraries were installed as system packages and linked dynamically at runtime.

Timing for all of the tests was done using the computer's system clock. Although more accurate timing methods are possible, such as counting the number of processor cycles used by each function, the test machine was otherwise completely idle while the tests were running (i.e. not even a window manager was running), so wall-clock time should be a good measure of processor time.

4.1 Implementation Details

There are several ways in which my implementations of the accumulators differs from their theoretical definitions, and a few configuration choices I needed to make as I implemented these algorithms. Most of my implementation decisions were made with practicality in mind, and my goal was always to create a system reflecting the way an accumulator might actually be implemented. Whenever possible I made sure to use

the same design for the RSA accumulator and bilinear-map accumulator, so that the differences in their performance would not be the result of my failing to optimize one of them as well as the other.

As I mentioned in the background section, the standard method of computing prime representatives for composite elements in the RSA accumulator is to use the inverse of a two-universal hash function. However, my implementation uses the more practical method based on random oracles. A truly random oracle is, of course, impossible in practice, but there are several acceptable ways of approximating one with a hash function, some of which are described in [4]. I used the following construct for my random oracle: On input value x, the first 64 bits of x are used to seed a linear-congruential pseudorandom number generator, which is used to generate a random 16-bit number r. The random number is appended to x, and xr is used as input to the SHA-256 cryptographic hash function, which generates the oracle's output. This oracle is deterministic (the PRNG will always produce the same 16 bits given the same seed) but avoids the "structure" of the SHA-256 hash by salting its input.

Although I discussed several ways in which the accumulator algorithms could be parallelized in section 3, I did not end up implementing all of the parallel algorithms as they are described in theory. In particular, the parallel prefix-sum algorithm is difficult to implement in software because it requires fine-grained control over the operation of individual threads and the location of shared memory, and is most useful in real life when it is implemented on specialized hardware such as GPUs. The C++ threading library is better suited to parallel algorithms defined in terms of "task" functions that can be executed asynchronously. As a result, I did not use any concurrency for computing a single accumulation value with the private key in either type of accumulator. For computing all the witness values for a set with the private key, I did not use the parallel prefix sum algorithm for the products in the exponents, but I implemented a different parallel algorithm (in both accumulators) that takes advantage of the fact that any pair of witness values require almost the same multiplication, differing by only a single term.

This algorithm is, as far as I know, my own design, so I will describe it in detail. If we consider the exponent being computed to generate a witness value for either accumulator to be the product of a set of values $\{x_1, \ldots, x_n\}$ (prime representatives in the case of the RSA accumulator, sums $e + s$ in the case of the bilinear-map accumulator), then generating all the witnesses for a set requires a series of very similar multiplications:

$$\begin{aligned}
\text{Witness 1: } & x_2 x_3 x_4 x_5 \cdots x_n \\
\text{Witness 2: } & x_1 x_3 x_4 x_5 \cdots x_n \\
\text{Witness 3: } & x_1 x_2 x_4 x_5 \cdots x_n \\
& \vdots \\
\text{Witness } n\text{: } & x_1 x_2 x_3 x_4 \cdots x_{n-1}
\end{aligned}$$

These can be separated into a set of "left side" and a set of "right side" products, each of which has the property of being a sequence of partial products that adds one new factor at each iteration:

$$\begin{array}{ll}
1 & x_2 x_3 x_4 x_5 \cdots x_n \\
x_1 & x_3 x_4 x_5 \cdots x_n \\
x_1 x_2 & x_4 x_5 \cdots x_n \\
\vdots & \vdots \\
x_1 x_2 x_3 x_4 \cdots x_{n-2} & x_n \\
x_1 x_2 x_3 x_4 \cdots x_{n-1} & 1
\end{array}$$

The witness exponents are simply the product of an element from the left side and an element from the right side. Since each set can be computed by sequentially computing its largest element and saving each partial product, generating all of the exponents in this way will take $O(2n)$, whereas naively computing all of the witness exponents individually would take $O(n^2)$ (n multiplications for each of n witnesses). The parallel

component of the algorithm is the fact that each set can be computed independently of the other, so the algorithm can run in $O(n)$ time (eliminating the constant 2) with two parallel tasks.

In addition to this algorithm, I parallelized the compute-all-witnesses operation for both accumulators by running the exponentiation steps in parallel. After all of the witness exponents were computed, I created a separate parallel task for each witness to do the exponentiation operation. This is more analogous to the theoretical observation that generating each witness in a set of witnesses can be done in its own thread.

For computing the public key elements of the bilinear-map accumulator, I did not use the parallel prefix sum algorithm to compute the powers of s, but I did implement a more basic parallel algorithm using only two parallel tasks. Each task computes the powers $g^s, g^{s^2}, g^{s^3}, \ldots, g^{s^q}$ for either g_1 or g_2 by sequentially raising each partial power to s and storing the result (so after g^s is computed, g^{s^2} is computed as $(g^s)^s$). This still takes $O(q)$ time, but it is twice as fast as computing both sets of powers sequentially.

I also used a nonstandard parallel algorithm to compute accumulation values and witnesses with only the public key in the bilinear-map accumulator, as a result of the features of the DCLXVI library. This library provides a highly-optimized (but single-threaded) function for computing a large batch of group exponentiation and multiplication operations, i.e., it efficiently computes $g_a^x \cdot g_b^y \cdot g_c^z \cdots$ for a set of group elements $\{g_a, g_b, g_c, \ldots\}$ and a set of scalars $\{x, y, z, \ldots\}$. Since this is exactly what I need to compute in order to accumulate a set with the public key, my code takes advantage of it by splitting the polynomial coefficients and public-key elements into large batches (1000 coefficients each) and computing each batch in parallel. The results of the parallel calls to the batch function are multiplied together sequentially, since there are usually few enough of them that the effect of doing the multiplication in parallel would be negligible. I did not use the parallel prefix sum algorithm to compute the polynomial coefficients, since as I mentioned above it is difficult to implement in software, and testing showed that the public-key accumulation function was spending more time multiplying and exponentiating group elements than finding polynomial coefficients.

For both accumulators, I implemented the parallel algorithm I described in theory for generating all witnesses for a set using only the public key. Given a set, the witness for each element was computed as a separate parallel task, which simply invoked the public-key accumulation function with a subset excluding the desired element. For the RSA accumulator, I implemented the parallel algorithm I described in theory for precomputing the set of prime representatives in parallel, generating each prime representative in a separate task.

On the other hand, I did not implement any concurrency for verifying a large set of elements because I considered it unlikely that a single entity would use an accumulator to verify every element in a set. While many applications of accumulators involve some kind of server or authority generating a large set of witnesses, verification is usually done on an individual or per-client level. Keeping the verification algorithm single-threaded allowed me to accurately measure the amount of time it would take a single client to verify an element with a witness.

Note that in all of my parallel algorithms I refer to parallel tasks as opposed to threads. This is because my implementation uses a thread pool instead of explicit thread creation to handle concurrency. Since thread creation and deletion is expensive and could occur many times in the course of an accumulator's operation if each parallel algorithm created its own threads, a thread pool is a useful optimization that moves all thread creation to the beginning of the program and all thread deletion to the end. Code needing to run some operations in parallel can submit a "task" to the thread pool (specifically a function and its arguments), where it will be evaluated by the next available worker thread. In my program I initialize the thread pool with 16 threads, a number I chose to ensure there would be at least one software thread available to run on each hardware thread, even accounting for some threads being blocked, in systems with up to 8 cores.

The way in which I handled the computation and use of prime representatives in my test programs is a design decision worth mentioning. Technically, according to the way the RSA accumulator is defined, the prime representative of each element in the set must be computed before that element is used in each stage of the accumulator (accumulation, witness generation, and verification). However, in many cases the entity that computes the accumulation value for a set will also compute some or all of the witnesses for the same set, and will thus be able to re-use stored values of prime representatives instead of generating them again in the witness step. To allow for this case, and to make the RSA accumulator's tests more comparable to the

bilinear-map accumulator's, I separated the computation of prime representatives into a separate test and allowed all the accumulation and witness-generation tests to use the stored representatives. On the other hand, a verifier must always compute the prime representative of an element before verifying it (since the verifier is by definition distinct from the entity that computed the accumulator), so the verification function for the RSA accumulator does include computing a prime representative for the element to be verified.

Finally, an important design decision I had to make was choosing the security parameter of each accumulator so that they had equivalent levels of security. The performance comparison would be fairly useless without this condition, since the security parameter can significantly affect performance. Since the DCLXVI library has a hard-coded group order of 256 bits, this amounted to finding an RSA modulus size that would provide the same level of security. Different standards bodies have proposed different equivalence sizes for RSA moduli and elliptic curve orders; NIST, in [3], defines the 128-bit security level to include 256-bit-order elliptic curves and 3072-bit RSA moduli, while ECRYPT, in [24] claims that a 256-bit-order elliptic curve is equivalent to a 3248-bit RSA modulus, also at the 128-bit security level. I chose to follow NIST's slightly more permissive standard, since 3072 bits is also a common RSA modulus size, and I wanted to give the RSA accumulator a chance at performing as quickly as the bilinear-map accumulator.

4.2 Public Key and Witness Sizes

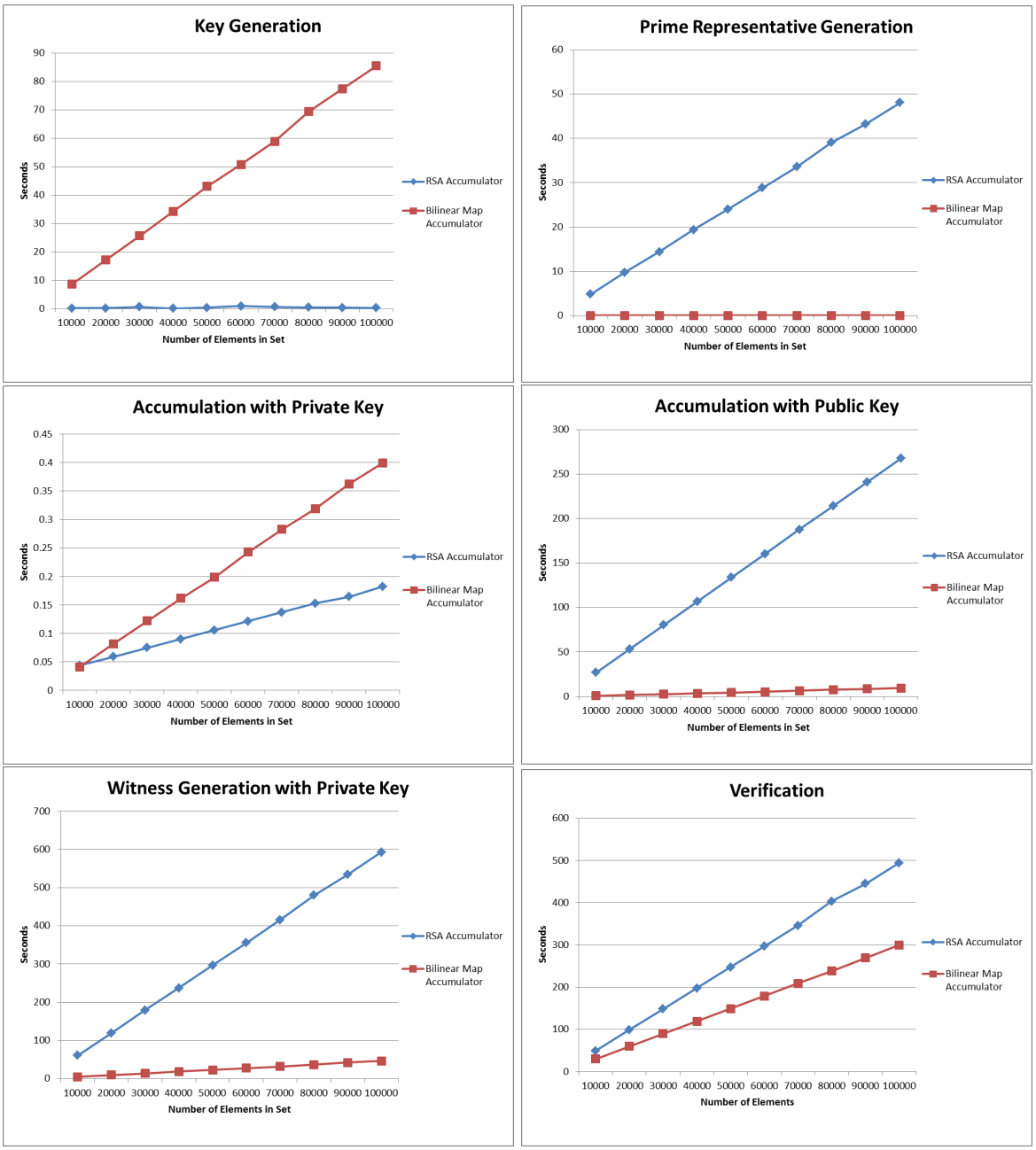
One important factor when comparing the practical usefulness of these two types of accumulators is the amount of information that must be transmitted between sources and verifiers under each scheme. Determining the size of a public key and of an element's witness could be considered an experimental result, but these sizes are constant across all of the tests I performed and hence do not appear in my results section. They are more like an implementation detail, since they are fixed once an implementation of RSA accumulators and bilinear-map accumulators has been chosen.

In my implementation, the RSA accumulator has a public key of size 6161 bits. This includes 3072 bits for the RSA modulus, 17 bits for the accumulator base (which is hard-coded to the standard value 65537) and 3072 bits to represent the modulus in the modular-integer data type used to store the base (integers modulo some p in my code are represented as two integer values, the mantissa and the modulus). Since the oracle algorithm used to compute prime representatives is deterministic and depends on no instance-specific data, I will assume that it is public knowledge and can be re-implemented by the verifier without needing to download code from the entity computing accumulations. The size of a witness for the RSA accumulator is up to 6144 bits, which is a single modular-integer value composed of a 3072-bit integer and a 3072-bit modulus. Since the large-integer data type in FLINT is constructed as a growing array of "limb" data types (integers the size of a machine word), the witness may be less than this size if its mantissa is significantly less than $\mathbf{N}$ and can be represented in fewer than 3072 bits.

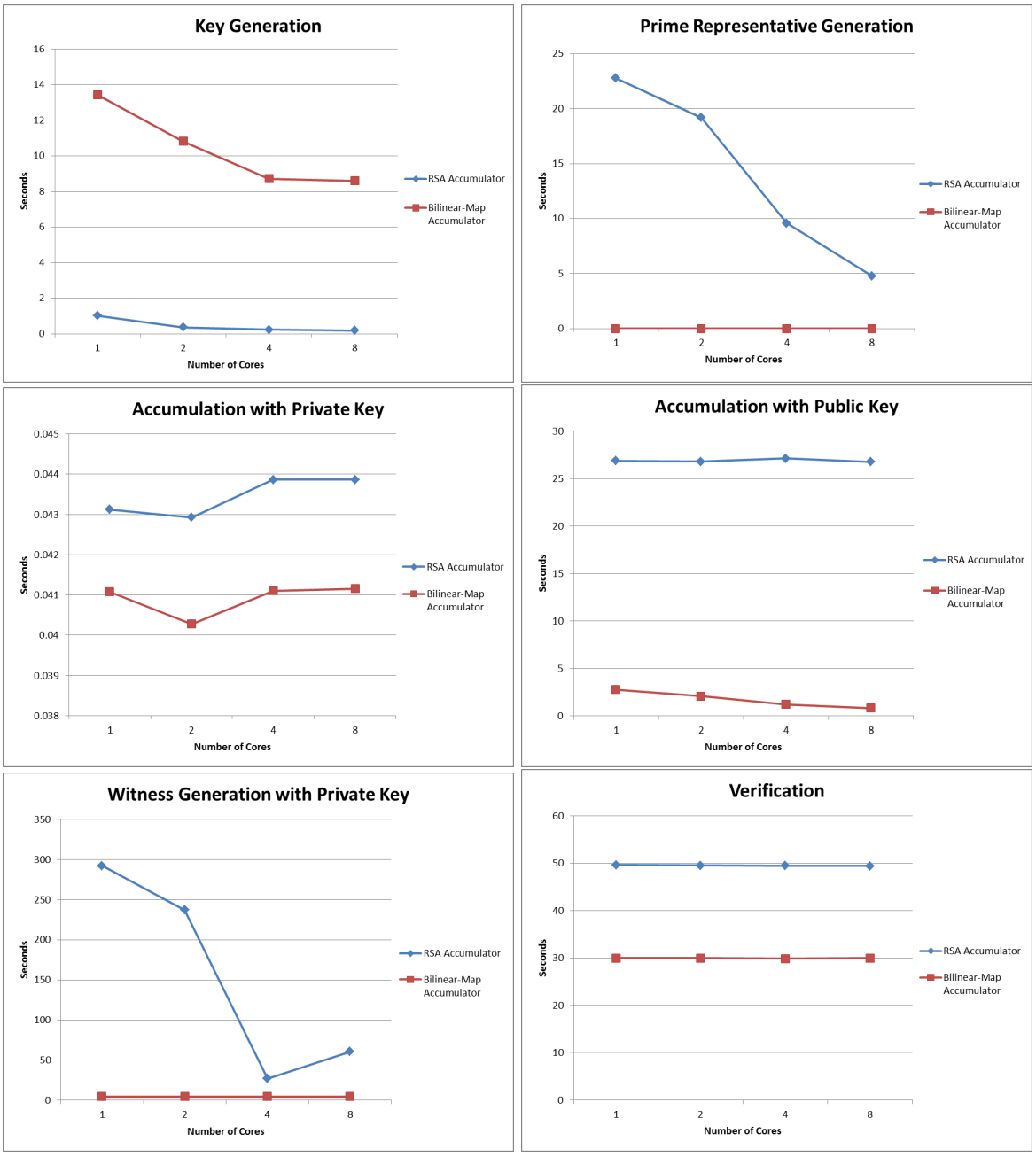
Meanwhile, the bilinear-map accumulator's public key size varies depending on the parameter q, which is an upper bound on the size of the set to be accumulated; it is $9216 \cdot q$ bits in length. The public key must contain q pairs of $\mathbb{G}_1$ and $\mathbb{G}_2$ elements, each $\mathbb{G}_1$ element is represented in 3072 bits as four 768-bit coordinates on the elliptic curve, and each $\mathbb{G}_2$ element is represented in 6144 bits as four 1536-bit coordinates on the elliptic twist (these representation details were determined by the DCLXVI library). The size of a witness for the bilinear-map accumulator is 6144 bits, since it is a single element of the group $\mathbb{G}_2$.

5 Results

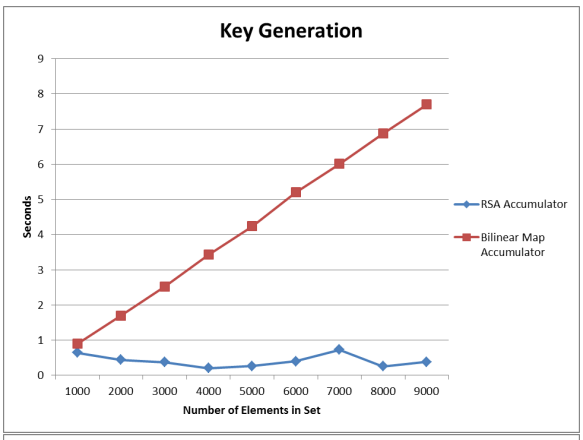
The following graphs show how the running time of each accumulator operation changes as the set size increases, running with 8 hardware threads available.

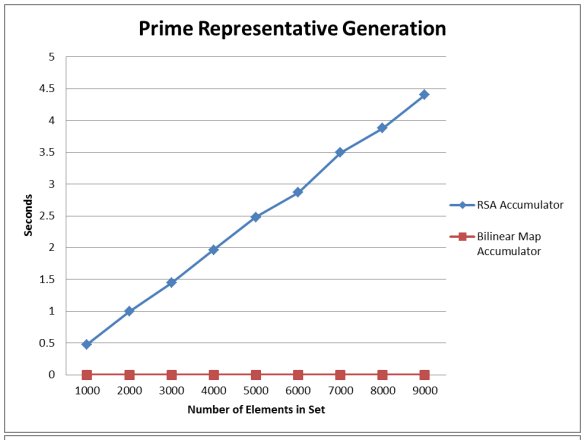


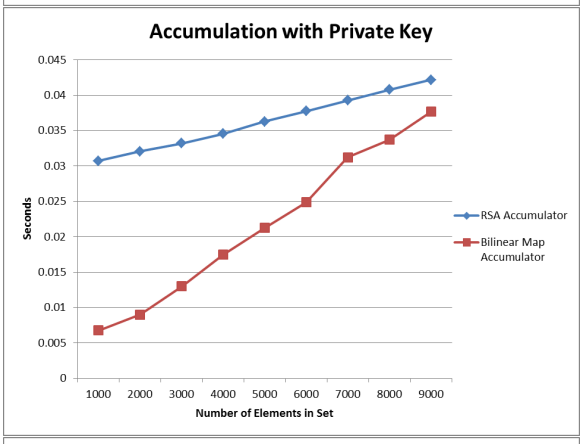
The following graphs show how the running time of each accumulator operation changes as the number of hardware threads available increases, assuming a set size of 10,000 elements. These tests were all run on the same machine, with some cores disabled.

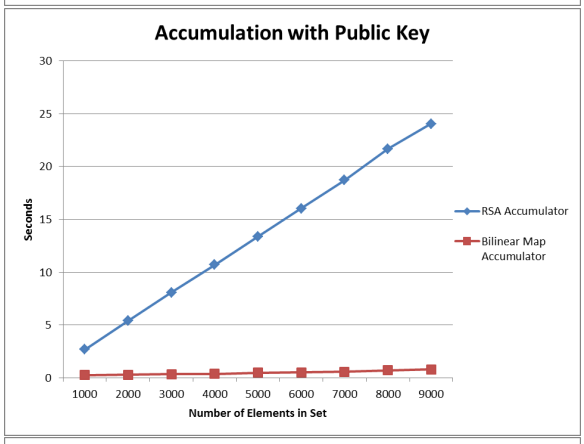


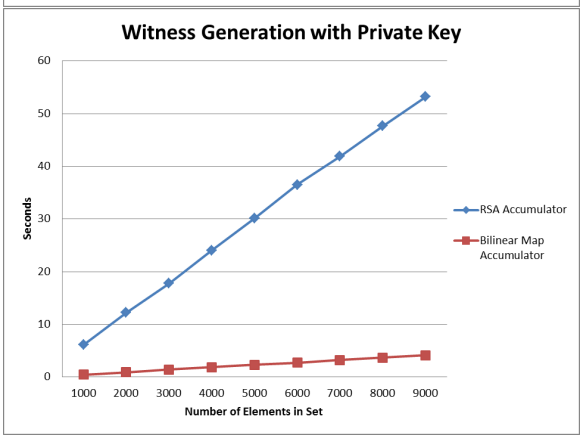
Finally, the following graphs show the running time of each accumulator operation as the set size increases, running with 8 hardware threads available, but with much smaller sets of elements that allowed the public-key witness computation algorithm to finish in a reasonable amount of time.

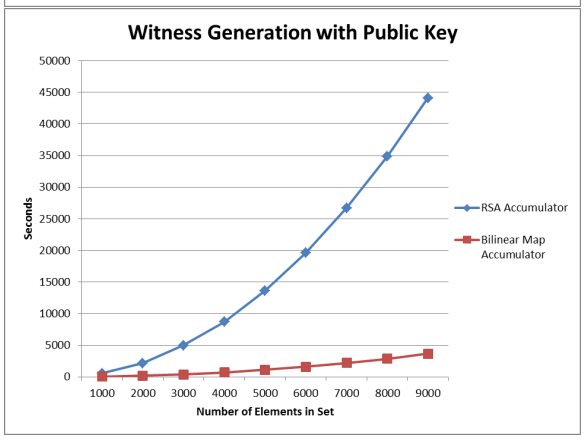

Key Generation
Seconds
Number of Elements in Set
RSA Accumulator
Bilinear Map Accumulator

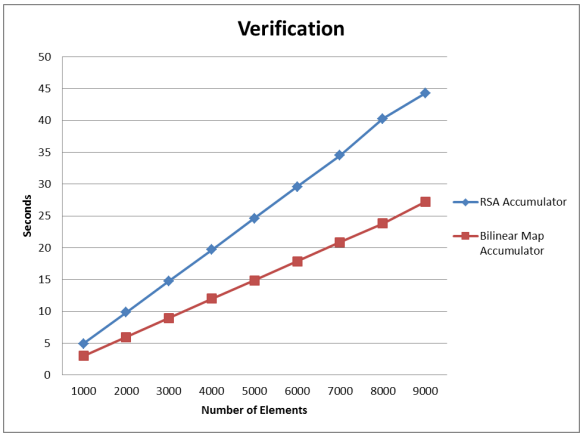

Prime Representative Generation
Seconds
Number of Elements in Set
RSA Accumulator
Bilinear Map Accumulator


Accumulation with Private Key
Seconds
Number of Elements in Set
RSA Accumulator
Bilinear Map Accumulator


Accumulation with Public Key
Seconds
Number of Elements in Set
RSA Accumulator
Bilinear Map Accumulator


Witness Generation with Private Key
Seconds
Number of Elements in Set
RSA Accumulator
Bilinear Map Accumulator


Witness Generation with Public Key
Seconds
Number of Elements in Set
RSA Accumulator
Bilinear Map Accumulator



Unfortunately, as a result of the exceedingly slow performance of public-key witness computation, it was not feasible to run these experiments with fewer cores enabled, so there is no data on the effect of available hardware threads on public-key witness computation. Finally, this table shows the average time per element, in seconds, for each operation, calculated by finding the slope of the regression line fitting each graph of the test with 8 threads and large sets of elements.

	Key Generation	Prime Representative Generation	Accumulation with Private Key	Accumulation with Public Key
RSA Accumulator	2.23852×10^{-6}	0.000481499	1.53345×10^{-6}	0.002675604
Bilinear-Map Accumulator	0.000857809	0	3.98997×10^{-6}	9.53584×10^{-5}

	Witness Generation with Private Key	Witness Generation with Public Key[2]	Verification
RSA Accumulator	0.005933464	5.4446684333	0.004967992
Bilinear-Map Accumulator	0.000461254	0.453206817	0.002990075

6 Discussion

At first glance, these results may seem to indicate that there is no clear advantage to using one accumulator over the other; the bilinear-map accumulator is faster in some operations, while the RSA accumulator is faster in others. However, a closer look reveals that not all operations are equally significant. Although the bilinear-map accumulator appears to be much slower at computing accumulations with the private key, the difference between it and the RSA accumulator even at the most extreme is only a few hundred milliseconds, spread over the course of 100,000 elements. The slopes of the lines only look significantly different because of the small scale of the y-axis; their actual values, as shown in the table, are actually within 3×10^{-6} of each other. For very small set sizes the bilinear-map accumulator is actually a bit faster, as the last set of tests showed. Meanwhile, the RSA accumulator is slower than the bilinear-map accumulator by a significant amount, with slopes that differ by at least an order of magnitude, in public-key accumulation, public-key witness generation, and private-key witness generation. The RSA accumulator is also slightly slower (within the same order of magnitude) at verification.

The bilinear-map accumulator has a definite disadvantage in the area of key generation, since its key-generation algorithm requires linear instead of constant time. This results in an average of .85 ms extra processing time per element when accumulating a set with the bilinear-map accumulator. However, the RSA accumulator has a definite disadvantage in the area of prime representative generation, which the bilinear-map accumulator doesn't need to do at all. This adds an average of .48 ms extra processing time per element when accumulating a set with the RSA accumulator. These asymmetric processing overheads almost cancel

[2]This is actually the slope of the graph for the small-sets test, since the tests with larger sets were unable to complete the public-key witnesses test. It is also less accurate because it is the result of a linear regression of a quadratic curve.

each other out; in fact, since prime representative generation may need to be done more than once (i.e. by an accumulation-computing entity and again by a separate witness-computing entity), while public-key generation will only ever need to be done once, the bilinear-map accumulator can come out ahead in some protocols.

The results of the tests across different number of hardware threads are less interesting and informative, though they do show approximately the expected behavior. Increasing the number of threads did significantly decrease the running times of the accumulator functions that were parallelized, while having no effect on the ones that were sequential: key generation for bilinear-map accumulators improved (while staying constant for RSA accumulators), prime representative generation for RSA accumulators improved, and public-key accumulation for bilinear-map accumulators improved (while staying constant for RSA accumulators). The results for private-key witness generation were somewhat odd, with only the RSA accumulator showing a speedup from multithreading despite the fact that the bilinear-map accumulator also used a parallel algorithm to compute witness exponents in bulk. However, this can probably be attributed to the fact that the modular multiplication operations being parallelized in the bilinear-map accumulator were over a much smaller modulus (the 256-bit group order rather than a 3072-bit RSA modulus), and thus were sufficiently inexpensive to make the impact of threading negligible. The RSA accumulator generally showed a more significant speedup from multithreaded computation, but only because it was (usually) so much slower than the bilinear-map accumulator to begin with.

7 Conclusion

Overall, the bilinear-map accumulator is generally faster than the RSA accumulator, and can be expected to perform better in most situations that call for accumulators. An instructive example of just how much faster the bilinear-map accumulator can be is to consider the expected average "round-trip" time of a single element, that is, the total time spent by the accumulator in accumulating, generating a witness for, and verifying that element. (This includes time spent generating a public key or computing prime representatives). Assuming accumulation and witness generation are done using the private key, the round-trip time per element for the RSA accumulator based on the table above is 11.4 ms, while the round-trip time per element for the bilinear-map accumulator is 4.3 ms. This makes the bilinear-map accumulator over twice as fast as the RSA accumulator. The difference is even more pronounced assuming accumulation and witness generation are done using the public key, in which case the round-trip time for the RSA accumulator is 5.4 seconds, while for the bilinear-map accumulator it is 0.45 seconds, roughly 12 times faster.

Some of the bilinear-map accumulator's speed advantage is undoubtedly due to the fact that the RSA accumulator was instantiated with a very large modulus, three times larger than the standard RSA modulus size used for many public-key cryptography applications today. The larger the modulus, the more computationally expensive the accumulator's modular arithmetic operations, particularly the modular exponentiations needed for public-key accumulation. The fact that such a large modulus was required in order to make the RSA accumulator as secure as the bilinear-map accumulator indicates the powerful security of elliptic-curve cryptosystems, which are a promising area of research for secure but practical cryptography.

There still remain some interesting questions to explore in the area of practical cryptographic accumulator performance. Future work could include an implementation that focuses more strongly on the parallel aspects of the accumulator algorithms, using the best available concurrent algorithms and fine-tuning the management of threads, to more fully investigate the degree to which each algorithm can benefit from multicore processors. Alternatively, this implementation could be extended to include features of dynamic accumulators, to determine if the additional functions introduced by the dynamic accumulator contract show the same performance differences as the ones tested here. Finally, it would be valuable to implement an entire accumulator-based security scheme using both types of accumulators and verify that the speed advantage of the bilinear-map accumulator holds out when it is used as part of a real system. An interesting experiment would be to find an accumulator-based system that has already been implemented and tested with RSA accumulators, and re-implement it with bilinear-map accumulators to see if its performance can be improved simply by substituting a different implementation for the general accumulator interface.

References

[1] Man Ho Au, Qianhong Wu, Willy Susilo, and Yi Mu. Compact e-cash from bounded accumulator. In Masayuki Abe, editor, *Topics in Cryptology – CT-RSA 2007*, number 4377 in Lecture Notes in Computer Science, pages 178–195. Springer Berlin Heidelberg, January 2006.

[2] Niko Barić and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *Advances in Cryptology – EUROCRYPT'97*, pages 480–494. Springer, 1997.

[3] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management – part 1: General (revision 3). Special Publication 800-57, NIST, Gaithersburg, MD, July 2012. `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf`.

[4] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 62–73, New York, NY, USA, 1993. ACM.

[5] Josh Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Advances in Cryptology – EUROCRYPT'93*, pages 274–285. Springer, 1994.

[6] Guy Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Carnegie Mellon University, School of Computer Science, November 1990.

[7] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In Matt Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, number 3152 in Lecture Notes in Computer Science, pages 41–55. Springer Berlin Heidelberg, January 2004.

[8] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In Stanisław Jarecki and Gene Tsudik, editors, *Public Key Cryptography – PKC 2009*, number 5443 in Lecture Notes in Computer Science, pages 481–500. Springer Berlin Heidelberg, January 2009.

[9] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, number 2442 in Lecture Notes in Computer Science, pages 61–76. Springer Berlin Heidelberg, January 2002.

[10] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[11] Scott A. Crosby and Dan S. Wallach. Authenticated dictionaries: Real-world costs and trade-offs. *ACM Transactions on Information and System Security (TISSEC)*, 14(2):17:1–17:30, September 2011.

[12] Wei Dai. Crypto++ library 5.6.2. `http://www.cryptopp.com`, February 2013.

[13] Ivan Damgåard and Nikos Triandopoulos. Supporting non-membership proofs with bilinear-map accumulators. Technical report, Cryptology ePrint Archive, Report 2008/538, 2008.

[14] Yevgeniy Dodis, Aggelos Kiayias, Antonio Nicolosi, and Victor Shoup. Anonymous identification in ad hoc groups. In *Advances in Cryptology – EUROCRYPT 2004*, pages 609–626. Springer, 2004.

[15] Michael T. Goodrich, Roberto Tamassia, and Jasminka Hasić. An efficient dynamic and distributed cryptographic accumulator. In Agnes Hui Chan and Virgil Gligor, editors, *Information Security*, number 2433 in Lecture Notes in Computer Science, pages 372–388. Springer Berlin Heidelberg, January 2002.

[16] William Hart. Fast library for number theory: an introduction. In *Mathematical Software – ICMS 2010: Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010, Proceedings*, volume 6327 of *Lecture Notes in Computer Science*, pages 88–91. Springer, 2010. `http://www.flintlib.org`.

[17] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, New York, 1997.

[18] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In Michel Abdalla and Paulo S.L.M. Barreto, editors, *Progress in Cryptology – LATINCRYPT 2010*, volume 6212 of *Lecture Notes in Computer Science*, pages 109–123. Springer-Verlag Berlin Heidelberg, 2010. Updated version at `http://cryptojedi.org/papers/#dclxvi`.

[19] Lan Nguyen. Accumulators from bilinear pairings and applications. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, number 3376 in Lecture Notes in Computer Science, pages 275–292. Springer Berlin Heidelberg, January 2005.

[20] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 437–448, New York, NY, USA, 2008. ACM.

[21] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Cryptographic accumulators for authenticated hash tables. Technical report, Cryptology ePrint Archive, Report 2009/625, 2009.

[22] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal verification of operations on dynamic sets. In *Advances in Cryptology – CRYPTO 2011 – 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 91–110, 2011.

[23] Tomas Sander, Amnon Ta-Shma, and Moti Yung. Blind, auditable membership proofs. In Yair Frankel, editor, *Financial Cryptography*, number 1962 in Lecture Notes in Computer Science, pages 53–71. Springer Berlin Heidelberg, January 2001.

[24] Nigel Smart, editor. *ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012)*. European Network of Excellence in Cryptology II, September 2012. `http://www.ecrypt.eu.org/documents/D.SPA.20.pdf`.