Fast Batch Verification for Modular Exponentiation and Digital Signatures

Mihir Bellare[1], Juan A. Garay[2], and Tal Rabin[2]

[1] Dept. of Computer Science & Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, California 92093, USA. E-Mail: mihir@cs.ucsd.edu URL: http://www-cse.ucsd.edu/users/mihir

[2] IBM T.J. Watson Research Center, PO Box 704, Yorktown Heights, New York 10598, USA. E-mail: {garay,talr}@watson.ibm.com

Abstract. Many tasks in cryptography (e.g., digital signature verification) call for verification of a basic operation like modular exponentiation in some group: given (g, x, y) check that $g^x = y$. This is typically done by re-computing g^x and checking we get y. We would like to do it differently, and faster.

The approach we use is batching. Focusing first on the basic modular exponentiation operation, we provide some probabilistic batch verifiers, or tests, that verify a sequence of modular exponentiations significantly faster than the naive re-computation method. This yields speedups for several verification tasks that involve modular exponentiations.

Focusing specifically on digital signatures, we then suggest a weaker notion of (batch) verification which we call "screening." It seems useful for many usages of signatures, and has the advantage that it can be done very fast; in particular, we show how to screen a sequence of RSA signatures at the cost of one RSA verification plus hashing.

1 Introduction

It is a consequence of the "adversarial" nature of cryptography that many of its computational tasks are for the purpose of "verifying" some property or computation. For example, signatures need to be verified; the opening of a bit-commitment needs to be verified; in protocols, various claims about generated values and their relations need to be verified.

These tasks are computationally important; for example, signature verification is likely to be done much more often than signature generation, as certificates and signed documents are circulated.

At the heart of many of these verification tasks is the problem of verifying a basic computational operation like modular exponentiation in some group: given (g, x, y) check that $g^x = y$. The naive way to verify such a claim is to redo the operation and check we get back the same value: namely, re-compute g^x and check it equals y. We would like to find means of verification, for such basic operations, that are faster than re-computation, and thereby speed up any verification process using such operations.

In this paper we investigate the use of batching for the purpose of speeding up such verification. This is a natural idea since we often have to verify many instances simultaneously. For example, a certificate chain can contain many signatures to check; a bank can be signing coins and we have many coins to verify; ZK proofs use many bit commitments, whose decommitments need to be verified.

We consider batching for verification in several contexts. The first is very general, namely batch verification for modular exponentiation itself. We provide several *batch verifiers* for modular exponentiation. These are probabilistic tests that verify the correctness of a batch of exponentiations much faster than doing each verification individually. We specify several uses for these tests, but there are probably more. Next we suggest a new notion called "signature screening," which provides "weak but fast" verification for signatures, and show how to implement it very efficiently for RSA signatures.

We also suggest a notion of batch program instance checking, and provide fast batch verification methods for degrees of polynomials which have applications in verifiable secret sharing and other robust distributed tasks. These, together with some applications of the results here, and all proofs, are omitted from this abstract, and can be found in our full paper [2] which is available on the web.

Following a brief discussion of previous work, we will look at all the above in more detail.

PREVIOUS WORK. The modular exponentiation operation itself can be made more efficient via pre-processing [9, 13] or addition chain heuristics [8, 18, 16]. What we are saying is that performing modular exponentiation is only one way to perform verification, and if the interest is verification, one can do better than any of these ways. In particular, our batch verifiers will perform better than the naive re-computation based verifier, even when the latter uses the best known exponentiation methods. In fact, better exponentiation methods only make our batch verifiers even faster, because we use these methods as subroutines.

The idea of batching in cryptography is of course not new: some previous instances are [11, 15, 6, 14]. However, there seems to have been no previous systematic look at the general problem of batch verification for modular exponentiation, and our first set of results indicates that by putting oneself above specific applications one can actually find general speed-up tools that apply to them; in particular, we improve some of the mentioned works.

1.1 Batch verification

Let R be a boolean relation. (Meaning $R(inst) \in \{0,1\}$ for any instance $inst$ of R. For example, $R(x,y) = 1$ iff $g^x = y$ in some group of which g is a generator, or R might be a signature verification algorithm with respect to some fixed public key.) The verification problem for R is: given an instance $inst$, check whether $R(inst) = 1$. In the batch verification problem we are given a sequence $inst_1, \ldots, inst_n$ of instances and asked to verify that for *all* $i = 1, \ldots, n$ we have $R(inst_i) = 1$. The naive way is to compute $R(inst_i)$, and check it is 1, for all $i = 1, \ldots, n$. We want to do it faster. To do this, we allow probabilism and an error probability. A *batch verifier* (also called a *test*) is a probabilistic algorithm

Test	No. of multiplications
Naive	$\mathit{ExpCost}_G^n(k_1)$
RANDOM SUBSET (RS)	$nl/2 + \mathit{ExpCost}_G^l(k_1)$
SMALL EXPONENTS (SE)	$l + nl/2 + \mathit{ExpCost}_G(k_1)$
BUCKET	$\min_{m\geq 2} \left\lceil \frac{l}{m-1} \right\rceil \cdot (n + m + 2^{m-1}m + \mathit{ExpCost}_G(k_1))$

Fig. 1. *Performance of algorithms for batch verification of modular exponentiation.* We indicate the number of multiplications each method uses to get error 2^{-l}. See the text for explanations of the parameters.

V which takes $inst_1, \ldots, inst_n$ and produces a bit as output. We ask that when $R(inst_i) = 1$ for all $i = 1, \ldots, n$, this output be 1. On the other hand, if there is even a single i for which $R(inst_i) = 0$ then we want that $V(inst_1, \ldots, inst_n) = 1$ with very low probability. Specifically, we let l be a security parameter and ask that this probability be at most 2^{-l}.

We stress that if even a single one of the n instances is "wrong" the verifier should detect it, except with probability 2^{-l}. Yet we want this verifier to run faster than the time to do n computations of R.

1.2 Batch verifiers for modular exponentiation

Let g be a generator of a (cyclic) group G, and let q denote the order of G. The modular exponentiation function is $x \mapsto g^x$, where $x \in Z_q$. Define the exponentiation relation $\mathrm{EXP}_{G,g}(x, y) = 1$ iff $g^x = y$, for $x \in Z_q$ and $y \in G$.

We design batch verifiers for this relation. As per the above, such a verifier is given a sequence $(x_1, y_1), \ldots, (x_n, y_n)$ and wants to verify that $\mathrm{EXP}_{G,g}(x_i, y_i) = 1$ for all $i = 1, \ldots, n$. The naive test is to compute g^{x_i} and test it equals y_i, for all $i = 1, \ldots, n$, having cost n exponentiations. We want to do better.

Three tests, the RANDOM SUBSET TEST, the SMALL EXPONENTS TEST and the BUCKET TEST are presented, with analysis of correctness, in Section 3. Their performance is summarized in Table 1, with the naive test listed for comparison. We explain the notation used in the table: $k_1 = \lg(|G|)$; $\mathit{ExpCost}_G(k_1)$ is the number of multiplications required to compute an exponentiation a^b for $a \in G$ and b an integer of k_1 bits; and $\mathit{ExpCost}_G^s(k_1)$ is the cost of computing s different such exponentiations. (Under the normal square-and-multiply method, $\mathit{ExpCost}_G(k_1) \approx 1.5k_1$ multiplications in the group, but it could be less [9, 13, 8]. Obviously $\mathit{ExpCost}_G^s(k_1) \leq s \cdot \mathit{ExpCost}_G(k_1)$, but there are ways to make it strictly less [9, 13, 8], which is why it is a separate parameter. See Section 2.3 for more information.) We treat costs of basic operations like exponentiation as a parameter to stress that our tests can make use of any method for the task. In particular, this explains why standard methods of speeding up modular exponentiation such as those mentioned above are not "competitors" of our schemes; rather our batch verifiers will always do better by using these methods as subroutines.

Table 3 in Section 3.6 looks at some example parameter values and computes the speed-ups. We see where are the cross over points in performance: for small values of n the SMALL EXPONENTS TEST is better, while for larger values, BUCKET TEST wins. Notice that even for quite small values of n we start getting appreciable speed-ups over the naive method, meaning the benefits of batching kick in even when the number of instances to batch is quite small.

Asymptotically more efficient tests can be constructed by recursively applying the tests we have presented, but the gains kick in at values of n that seem too high to be useful, so we don't discuss this.

SOME APPLICATIONS. Applications are relatively obvious, namely to any discrete logarithm based protocol in which discrete exponentiation needs to be verified. In some cases, we need to tweak the techniques.

DSS signatures [12] are a particularly attractive target for batch verification because signing is fast and verification is slow. Naccache *et al.* [15] give some batch verification algorithms for a slight variant of DSS. We can adapt our tests to apply to this variant, and get faster batch verification algorithms. See [2].

In many ZK or witness-hiding proofs, discrete exponentiation may be used to implement bit commitment, and there are lots of such commitments. Our batch verifiers will speed-up verification of the de-commitments. We can also improve the discrete log based n-party signature protocols of Brickell *et al.* [10]. See [2].

EXPONENTIATION WITH COMMON EXPONENT. The version of the exponentiation problem that underlies RSA is different from the above in that the exponent, not the base, is fixed. The results discussed above don't apply to this version. Batch verification of RSA signatures can be done via screening as we now discuss.

1.3 Screening: Fast but weak verification for signatures

For the particular case of signature verification, we suggest a different notion of batch verification, called screening, which has weaker guarantees but can be achieved at much lower cost. In certain usages of signatures, it is adequate and useful.

Fix some signature scheme and a public key *pk* for it. Let $\mathit{Verify}_{pk}(\cdot,\cdot)$ be the verification algorithm of this scheme, meaning a signature x of message M is valid if $\mathit{Verify}_{pk}(M,x) = 1$. A batch instance for signature verification consists of a sequence $(M_1,x_1),\ldots,(M_n,x_n)$ where x_i is a purported signature of M_i relative to *pk*. Batch verification in the sense we have been discussing so far would mean batch verification for the relation $\mathit{Verify}_{pk}(\cdot,\cdot)$: the test would reject with high probability if there was any $i \in \{1,\ldots,n\}$ for which $\mathit{Verify}_{pk}(M_i,x_i) = 0$. In screening, what we ask is that if the batch instance $(M_1,x_1),\ldots,(M_n,x_n)$ contains a forgery —meaning there is some i such that M_i was never signed by the signer— then our batch verifier will reject, with high probability. However, if the signer has in the past signed all the messages $M_1,\ldots,M_n$, then our test might accept even if for some i the string x_i is in fact not a valid signature of M_i.

In other words, screening is the task of determining whether the signer has at some point authenticated the text M_i, rather than the task of checking that the particular string x_i provided is a valid signature of M_i. The rationale is that in many applications, all that counts is whether or not M_i is authentic. Take for example a case where the M_i are electronic coins. We may only really care whether the coin is valid, not whether we actually hold a correct signature demonstrating the validity of this particular coin.

In Section 4 we show how RSA signatures generated under the standard "hash-then-decrypt" paradigm can be very efficiently screened: the cost of batch verification is that of one exponentiation with the public RSA exponent plus some hashing.

1.4 Batch program instance checking and other results

The notion of batch verification has on the face of it nothing to do with program checking since there is no program in the picture that one is trying to check. Nonetheless, we apply this notion to do program checking in a novel way. Our approach, called batch program instance checking, permits fast checking, and also permits instance checking, not just program checking, in the sense that (in contrast to standard program checking [7]), a correct result is not rejected just because the program might be wrong on some other instance. We can do batch program instance checking for any function f whose corresponding graph (the relation $R_f(x,y) = 1$ iff $f(x) = y$) has efficient batch verifiers, so that the main technical problem is the construction of batch verifiers. See [2] for more information including explanations of how this differs from other notions like batch program checking [17].

The idea of batch verification introduced here was applied in [1] in the domain of fault-tolerant distributed computing. They design a batch verifiable secret sharing protocol and use it to construct "distributed pseudo-random bit generators," which are efficient ways of generating shared distributed coins.

An invited talk on batch verification including the material presented in this paper was given at *LATIN '98* [3].

2 Definitions

Here we provide formal definitions of the main new notions underlying this work, extending the discussion in Section 1.

2.1 Batch verification

Let $R(\cdot)$ be a boolean relation, meaning $R(\cdot) \in \{0,1\}$. An *instance* for the relation is an input *inst* on which the relation is evaluated. A *batch instance* for relation R is a sequence $inst_1, \ldots, inst_n$ of instances for R. (We call n the size of the instance, and also call this an n-instance for R.) We say that the batch instance is *correct* if $R(inst_i) = 1$ for all $i = 1, \ldots, n$, and *incorrect* if there is some $i \in \{1, \ldots, n\}$ for which $R(inst_i) = 0$.

Definition 1. *A* batch verifier *for relation R is a probabilistic algorithm V that takes as input (possibly a description of R), a batch instance $X = (inst_1, \ldots, inst_n)$ for R, and a security parameter l provided in unary. It satisfies:*

(1) *If X is correct then V outputs 1.*
(2) *If X is incorrect then the probability that V outputs 1 is at most 2^{-l}.*
The probability is over the coin tosses of V only.

Obvious extensions can be made, such as allowing a slight error in the first case. We stress that if there is even a single i for which $R(inst_i) \neq 1$, the verifier must reject, except with probability 2^{-l}. Variants such as batch verification over a distribution, or computational batch verification, are easily defined.

The *naive batch verifier*, or *naive test*, consists of computing $R(inst_i)$ for each $i = 1, \ldots, n$, and checking that each of these n values is 1.

In practice, setting l to be about 60, meaning an error of 2^{-60}, should suffice.

2.2 Signature screening: weak verification

SIGNATURES. A *digital signature scheme*, $(Gen, Sign, Verify)$, consists of a *key generation algorithm*, a *signing algorithm*, and a *verification algorithm*. The first is probabilistic; the second may be; the third is not. A matching pair of public and secret keys can be generated via $(pk, sk) \overset{R}{\leftarrow} Gen(1^k)$ where k is the security parameter. A message is signed via $x \overset{R}{\leftarrow} Sign_{sk}(M)$. A candidate message-signature pair (M, x) is verified by making sure $Verify_{pk}(M, x) = 1$.

SCREENING. The notion was discussed in Section 1.3. We now provide the formalization. Fix a signature scheme $(Gen, Sign, Verify)$. Recall that a batch instance for signature verification consists of a sequence $(M_1, x_1), \ldots, (M_n, x_n)$ where x_i is a purported signature of M_i relative to some given public key pk. Let *ScreenTest* be a (possibly probabilistic) algorithm, where $ScreenTest_{pk}((M_1, x_1), \ldots, (M_n, x_n))$ outputs a bit. We want to say what it means for this algorithm to be a good screening algorithm for the signature scheme.

An attacker A is given the public key pk. It tries to produce a batch instance, and is said to be successful if the batch instance contains an unauthenticated message but still passes the screening test. To make the notion strong, the attacker is allowed a chosen-message attack. So the game is like this. A has oracle access to $Sign_{sk}(\cdot)$. After making some number of signing queries it outputs a batch instance $(M_1, x_1), \ldots, (M_n, x_n)$. We say that M_i is a *not legally signed* if it was not previously a query to the $Sign_{sk}(\cdot)$ oracle. We say that A is *successful* if the batch instance $(M_1, x_1), \ldots, (M_n, x_n)$ contains a message M_i that was not legally signed, but $ScreenTest_{pk}((M_1, x_1), \ldots, (M_n, x_n)) = 1$. We let $Succ(A)$ denote the success probability of A. The probability is over the choice of keys, the coins of the signing algorithm, the coins of A, and the coins of the screening algorithm. Intuitively, the screening algorithm is good if $Succ(A)$ is small for any A whose computation time is not extraordinarily high. In the theorems (cf. Theorem 4) we will be more precise, quantifying the success probability as a function of the running time and allowed number of oracle queries of the adversary.

Whenever we talk about the running time of an algorithm, it is the sum of the actual running time (on some fixed RAM model of computation) and the size of the code.

2.3 Costs of Multiplication and Exponentiation

Let G be a (multiplicative) group. Many of our algorithms are in cryptographic groups like Z_N^* or subgroups thereof (N could be composite or prime). We measure cost in terms of the number of group operations, here multiplications, and discuss these costs below.

Given $a \in G$ and an integer b, the standard square-and-multiply method computes $a^b \in G$ at a cost of $1.5|b|$ multiplications on the average. Using the windowing method based on addition chains [8, 18], the cost can be reduced to about $1.2|b|$; pre-computation methods have been proposed to reduce the number of multiplications further at the expense of storage for the pre-computed values [9, 13] (a range of values can be obtained here; we give some numerical examples in Section 3.6). Accordingly it is best to treat the cost of exponentiation as a parameter. We let $\mathit{ExpCost}_G(k_1)$ denote the time to compute a^b in group G when $k_1 = |b|$, and express the costs of our algorithms in terms of this.

Suppose we need to compute $a^{b_1}, \ldots, a^{b_n}$, exponentiations in a common base a but with changing exponents. Say each exponent is t bits long. We can certainly do this with $n \cdot \mathit{ExpCost}_G(t)$ multiplications. However, it is possible to do better, via the techniques of [9, 13], because in this case the pre-computation can be done on-line and still yield an overall savings. Accordingly, we treat the cost of this operation as a parameter too, denoting it $\mathit{ExpCost}_G^n(t)$.

Note squaring can be performed faster than general multiplication.

3 Batch Verification for Modular Exponentiation

Let G be a group, and let $q = |G|$ be the order of G. Let g be a primitive element of G. Hence, for each $y \in G$ there is a unique $i \in Z_q$ such that $y = g^i$. This i is the discrete logarithm of y to the base g and is denoted $\log_g(y)$. Define relation $\mathrm{EXP}_{G,g}(x, y)$ to be true iff $g^x = y$. (Equivalently, $x = \log_g(y)$.) We let k_1 denote the length (number of bits) of q, and k_2 the length of g. With G, g fixed we want to construct fast batch verifiers for the relation $\mathrm{EXP}_{G,g}$.

3.1 Random subset test

The first thing that one might think of is to compute $x = \sum_{i=1}^n x_i \bmod q$ and $y = \prod_{i=1}^n y_i$ (the multiplications are in G) and check that $g^x = y$. However it is easy to see this doesn't work: for example, the batch instance $(x + \alpha, g^x), (x - \alpha, g^x)$ passes the test for any $\alpha \in Z_q$, but is clearly not a correct instance when $\alpha \neq 0$. A natural fix that comes to mind is to do the above test on a random subset of the instances: pick a random subset S of $\{1, \ldots, n\}$, compute $x = \sum_{i \in S} x_i \bmod q$ and $y = \prod_{i \in S} y_i$ and check that $g^x = y$. (The idea is that randomizing "splits" any "bad pairs" such as those of the example above.) We call this the ATOMIC RANDOM SUBSET TEST. It works in the sense of the following lemma, whose proof can be found in [2].

Lemma 1. *Given a group G and a generator g of G. Suppose $(x_1, y_1), \ldots, (x_n, y_n)$ is an incorrect batch instance of the batch verification problem for* $\mathrm{EXP}_{G,g}(\cdot, \cdot)$. *Then the* ATOMIC RANDOM SUBSET TEST *accepts $(x_1, y_1), \ldots, (x_n, y_n)$ with probability at most* $1/2$.

But $1/2$ is not a low enough error. (One can show the analysis is tight, so no better is expected.) To lower the error to the desired 2^{-l} we must repeat the atomic test independently l times, yielding the RANDOM SUBSET TEST of Figure 2. However, the repetition is costly: the total cost is now $nl/2 + \mathit{ExpCost}_G^l(k_1)$ multiplications. This is not so good, and, in many practical instances may even be *worse* than the naive test, for example if $n \leq l$. (Since l should be at least 60 this is not unlikely.)

The conclusion is that repeating many times some atomic test which itself has constant error can be costly even if the atomic test is efficient. Thus, in what follows we will look for ways to *directly* get low error. First, lets summarize the results we just discussed in a theorem.

Theorem 1. *Given a group G, a generator g of G. The* RANDOM SUBSET TEST *is a batch verifier for the relation* $\mathrm{EXP}_{G,g}(\cdot,\cdot)$ *with cost $nl/2 + \mathit{ExpCost}_G^l(k_1)$ multiplications, where $k_1 = \lceil \lg(|G|) \rceil$.*

3.2 Computing a product of powers

Before presenting the next test, we present a general algorithm we will use as a subroutine. Suppose $a_1, \ldots, a_n \in G$. Suppose $b_1, \ldots, b_n$ are integers in the range $0, \ldots, 2^t - 1 < |G|$. We write them all as strings of length t, so that $b_i = b_i[t] \ldots b_i[1]$. The problem is to compute the product $a = \prod_{i=1}^n a_i^{b_i}$, the operations being in G. The naive way to do this is to compute $c_i = a_i^{b_i}$ for $i = 1, \ldots, n$ and then compute $a = \prod_{i=1}^n c_i$. This takes $\mathit{ExpCost}_G^n(t) + n - 1$ multiplications, where k_2 is the size of the representation of an element of G. (Using square-and-multiply exponentiation, for example, this works out to $3ntk_2/2 + n - 1$ multiplications; with a faster exponentiation it may be a bit less.) However, drawing on some ideas from [9], we can do better, as follows:

```
Algorithm FastMult($(a_1,b_1), \ldots, (a_n,b_n)$)
    $a := 1$;
    for $j = t$ downto 1 do
        for $i = 1$ to $n$ do if $b_i[j] = 1$ then $a := a \cdot a_i$;
        $a := a^2$
return $a$
```

This algorithm does t multiplications in the outer loop and $nt/2$ multiplications on the average for the inner loop. Hence, for computing y we get a total of $t + nt/2$ multiplications.

3.3 The Small Exponents Test

We can view the ATOMIC RANDOM SUBSET TEST in a different way. Namely, pick bits $s_1, \ldots, s_n \in \{0, 1\}$ at random, let $x = \sum_{i=1}^n s_i x_i$ and $y = \prod_{i=1}^n y_i^{s_i}$, and check that $g^x = y$. (This corresponds to choosing the set $S = \{ i : s_i = 1 \}$.) We know this test has error $1/2$. The idea to get lower error is to choose $s_1, \ldots, s_n$ from a larger domain, say t bit strings for some $t > 1$. There are now two things to ask: whether this does help lower the error faster, and, if so, at what rate as

```
GIVEN: $g$ a generator of $G$, and $(x_1,y_1),\ldots,(x_n,y_n)$ with $x_i \in Z_p$ and $y_i \in G$.
       Also a security parameter $l$.

CHECK: That $\forall i \in \{1,\ldots,n\} : y_i = g^{x_i}$.

- Random Subset (RS) Test: Repeat the following atomic test, independently
  $l$ times, and accept iff all sub-tests accept:
      ATOMIC RANDOM SUBSET TEST:
      (1) For each $i = 1,\ldots,n$ pick $b_i \in \{0,1\}$ at random
      (2) Let $S = \{ i : b_i = 1 \}$
      (3) Compute $x = \sum_{i \in S} x_i \bmod q$, and $y = \prod_{i \in S} y_i$
      (4) If $g^x = y$ then accept, else reject.

- Small Exponents (SE) Test:
  (1) Pick $s_1,\ldots,s_n \in \{0,1\}^l$ at random
  (2) Compute $x = \sum_{i=1}^{n} x_i s_i \bmod q$, and $y = \prod_{i=1}^{n} y_i^{s_i}$
  (3) If $g^x = y$ then accept, else reject.

- Bucket Test: Takes an additional parameter $m \geq 2$. Set $M = 2^m$. Repeat
  the following atomic test, independently $\lceil l/(m-1) \rceil$ times, and accept iff all
  sub-tests accept:
      ATOMIC BUCKET TEST:
      (1) For each $i = 1,\ldots,n$ pick $t_i \in \{1,\ldots,M\}$ at random
      (2) For each $j = 1,\ldots,M$ let $B_j = \{ i : t_i = j \}$
      (3) For each $j = 1,\ldots,M$ let $c_j = \sum_{i \in B_j} x_i \bmod q$, and $d_j = \prod_{i \in B_j} y_i$
      (4) Run the Small Exponent Test on the instance $(c_1,d_1),\ldots,(c_M,d_M)$
          with security parameter set to $m$.
```

Fig. 2. *Batch verification algorithms for exponentiation with a common base.*

a function of t; and then as we increase t, how performance is impacted. Let's look at the latter first.

If we can keep t small, then we have only a single exponentiation to a large (ie. k_1-bit) exponent, as compared to l of them in the random subset test. That's where we expect the main performance gain. But now we have added n new exponentiations. However, to a smaller exponent. Thus, the question is how large t has to be to get the desired error of 2^{-l}.

We use some group theory to show that the tradeoff between the length t of the s_i's and the error is about as good as we could hope as long as the order q of the group is prime, namely setting $t = l$ yields the desired error 2^{-l}. (See Section 3.5 for discussion of what happens when q is not prime.) The corresponding test is the SMALL EXPONENTS (SE) TEST and is depicted in Figure 2. The proof of the following is in [2].

Theorem 2. *Given a group G of prime order q and a generator g of G. Then* Small Exponents Test *is a batch verifier for the relation* $\mathrm{EXP}_{G,g}(\cdot,\cdot)$ *with cost $l + n(1 + l/2) + \mathit{ExpCost}_G(k_1)$ multiplications, where $k_1 = |q|$.*

3.4 The Bucket Test

We saw that the Small Exponents Test was quite efficient, especially for an n that was not too large. We now present another test that does even better for large n. Our Bucket Test, shown in Figure 2, repeats m times an Atomic Bucket Test for some parameter m to be determined. In its first stage, which is steps (1)–(3) of the description, the atomic test forms M "buckets" $B_1, \ldots, B_M$. For each i it picks at random one of the M buckets, and "puts" the pair (x_i, y_i) in this bucket. (The value t_i in the test description chooses the bucket for i.) The x_i values of pairs falling in a particular bucket are added while the corresponding y_i values are multiplied; this yields the values c_j, d_j for $j = 1, \ldots, M$ specified in the description. The first part of the analysis below shows that if there had been some i for which $g^{x_i} \neq y_i$ then except with quite small probability (2^{-m}) there is a "bad bucket," namely one for which $g^{c_j} \neq d_j$.

Thus we are reduced to another instance of the same batch verification problem with a smaller instance size M. Namely, given $(c_1, d_1), \ldots, (c_M, d_M)$ we need to check that $g^{c_j} = d_j$ for all $j = 1, \ldots, M$. The desired error is 2^{-m}.

We can use the Small Exponents Test to solve the smaller problem. (Alternatively, we could recursively apply the bucket test, bottoming out the recursion with a use of the SE test after a while. This seems to help, yet for n so large that it doesn't really matter in practice. Thus, we shall continue our analysis under the assumption that the smaller sized problem is solved using the Small Exponents Test.) This yields a test depending on a parameter m. Finally, we would optimize to choose the best value of m. Note that until these choices are made we don't have a concrete test but rather a framework which can yield many possible tests. To enable us to make the best choices we now provide the analysis of the Atomic Bucket Test and Bucket Test with a given value of the parameter m, and evaluate the performance as a function of the performance of the inner test, which is SE. Later we can optimize. Since we use Small Exponents Test, we require the order of the group to be prime. The proof of the following is in [2].

Lemma 2. *Suppose G is a group of prime order q, and g is a generator of G. Suppose $(x_1, y_1), \ldots, (x_n, y_n)$ is an incorrect batch instance of the batch verification problem for* $\mathrm{EXP}_g(\cdot,\cdot)$. *Then the* Atomic Bucket Test *with parameter m accepts $(x_1, y_1), \ldots, (x_n, y_n)$ with probability at most $2^{-(m-1)}$.*

Regarding performance, it takes n multiplications to generate the buckets and the smaller instance. To evaluate the smaller instance using SE with parameters $2^m, m, |q|, k_2$ takes $m + 2^m m/2 + 2^m + \mathit{ExpCost}_G(|q|)$ multiplications by Theorem 2. This process is repeated $\lceil l/(m-1) \rceil$ times. When we run the test, we choose the optimal value of m, meaning that which minimizes the cost. Thus we have the following.

Theorem 3. *Given a group G of prime order q, and a generator g of G. Then the* Bucket Test *(with m set to the optimal value) is a batch verifier for the relation* $\mathrm{EXP}_{G,g}(\cdot,\cdot)$ *with cost*

$$\min_{m\geq 2}\left\{\left\lceil\frac{l}{m-1}\right\rceil\cdot\left(n+m+2^{m-1}(m+2)+ExpCost_G(k_1)\right)\right\}$$

multiplications, where $k_1 = |q|$.

To minimize analytically we would set $m \approx \log(n+k_1) - \log\log(n+k_1)$, but in practice it is better to work with the above formula and find the best value of m by search. This is what is done to compute the numbers in Table 3.

3.5 Prime versus non-prime order

The analysis of the Small Exponents Test as given by Theorem 2 (and hence of the Bucket Test as given by Theorem 3) is for groups of *prime order*. We are not working in Z_q^* (which has order $q-1$, not a prime) but in a group G which has order q a prime. In practice this is not really a restriction. As is standard in many schemes, we can work in an appropriate subgroup of Z_p^* where p is a prime such that q divides $p-1$. In fact, prime order groups seem superior to plain integers modulo a prime in many ways. The discrete logarithm problem seems harder there, and they also have nice algebraic properties which many schemes exploit to their advantage.

When the order is not prime, the Small Exponents Test (and hence the Bucket Test) do not work; it is easy to find counter-examples. For example, let p be a prime, and consider $G = Z_p^*$, which has non-prime order $p-1$. Let $g \in G$ be a generator of G and consider the batch instance $(x, -y \bmod p-1), (x, y)$ where $y = g^x \bmod p$. The Small Exponents Test will accept this instance whenever s_1 is even, which happens half the time, so its error will not be 2^{-l}, but only $1/2$. (Obvious fixes like using only odd values of s_i don't work.)

3.6 Performance

Table 3 looks at the concrete performance of the tests as we vary the size n of the batch instance. We have set $k_1 = 1024$, and $l = 60$. (Meaning the exponentiation is for 1024 bit moduli, and the error probability will be 2^{-60}.) We count the number of multiplications. We compare with the naive batch test, but this test is not naively implemented, in the sense that to be fair we use fast exponentiation as per [9, 13] to get the numbers in the first column. (Our tests use the same fast exponentiation methods as subroutines.) We assume a single exponentiation requires 200 multiplications [13]. (Using other storage to time tradeoffs as per [13] doesn't change the results, namely that our tests consistently perform better.)

Observe that which test is better depends on the value of n. As we expected, the RS test is actually *worse* than naive for small n. Until n about 200, the Small Exponents Test test is the best. From then on, the Bucket Test performs better. But at least one of our tests always beats the naive one. Furthermore, observe that benefits come in *even for small values of n*: at $n = 5$ the SE test is a factor of 2 better than naive. The factor of improvement increases

n	No. of multiplications used by different tests			
	Naive	RANDOM SUBSET	SMALL EXPONENTS	BUCKET
5	1 K	12 K	<u>0.4 K</u>	4.3 K
10	2 K	12.5 K	<u>0.6 K</u>	4.4 K
50	10 K	13.5 K	<u>1.8 K</u>	5 K
100	20 K	15 K	<u>3.2 K</u>	5.7 K
200	40 K	18 K	<u>6.2 K</u>	7.1 K
500	100 K	27 K	15.2 K	<u>10.7 K</u>
1,000	200 K	42 K	30.2 K	<u>16.5 K</u>
5,000	1000 K	162 K	150 K	<u>56 K</u>

Fig. 3. *Example:* **For increasing values of n, we list the number of 1024-bit multiplications (in thousands, rounded up), for each method to verify n exponentiations with error probability 2^{-60}. The lowest number for each n is underlined.**

with n: at $n = 200$ we can do about 6 times better than naive (using SE); at $n = 5000$, about 17 times better (using BUCKET).

4 Fast Screening for RSA

Batch verification for digital signature verification is a particular case of the general batch verification problem in which the relation is the signature verification relation. In particular, the above results help to get faster batch verification for discrete logarithm-based signatures like DSS (cf. [2]). However, we can do even better if we focus specifically on signatures, via the notion of screening presented in Section 2.2.

This is particularly interesting for RSA signatures. Here the verification relation is modular exponentiation, but with a common exponent, namely the relation $R_{N,e}(x, y) = 1$ iff $x^e \equiv y \bmod N$, and thus the above batch verifiers, which are for modular exponentiation in a common base, don't address this problem. (The tests are easily adapted to the common exponent case, but since the group is not of prime order, they don't work.) However, we present screening algorithms for the standard "hash-the-sign" type RSA signatures that are much faster than any of the above batch verifiers.

Note that RSA signature verification may be relatively fast anyway if one chooses a small public exponent, like three. Yet, there are various reasons one might want to use a bigger verification exponent (for example, to play with the signing exponent and speed up the signing). Actually our screening tests improve over the standard verification method even for small exponents, but obviously the gains are larger for large exponents.

HASH-THEN-DECRYPT RSA SCHEMES. The user has public key N, e and secret key N, d where N is an RSA modulus, $e \in Z^*_{\varphi(N)}$ an encryption exponent, and

```
GIVEN: $N, e$ and $(M_1, x_1), \ldots, (M_n, x_n)$ with $x_i \in Z_N^*$, and oracle access to hash function $H$

FDH-RSA Signature Screening Test
  If $(\prod_{i=1}^{n} x_i)^e = \prod_{i=1}^{n} H(M_i) \bmod N$ then return 1 else return 0
```

Fig. 4. *RSA signature screening test.*

d the corresponding decryption exponent. Define functions f, f^{-1}: $Z_N^* \to Z_N^*$ by $f(x) = x^e \bmod N$ and $f^{-1}(y) = y^d \bmod N$. The standard paradigm for signing with RSA in practice is to let $Sign_{N,d}(M) = H(M)^d \bmod N$ for some hash function H. A pair (M, x) is verified by checking that $x^e = H(M) \bmod N$. This was named the "hash-then-decrypt" paradigm and studied recently in [5] who point out that collision-freeness of H is not a strong enough requierement to guarantee security of this scheme based on the one-wayness of RSA. To get a better security guarantee without sacrificing performance, [5] appeals to the random oracle paradigm [4] and considers a couple of schemes in this setting. The simplest is the Full Domain Hash (FDH-RSA) scheme, which assumes H is a random oracle mapping $\{0,1\}^*$ to Z_N^*, and they show that FDH-RSA scheme is secure assuming RSA is a one-way function.

SCREENING FOR FDH-RSA. Our screening algorithm, called FDH-RSA SIGNATURE SCREENING TEST, is presented in Figure 4. It is very simple. Note there is no security parameter l in it: the failure probability of the test is related only to the difficulty of inverting RSA as Theorem 4 indicates.

This test is very efficient. There are n hashings (cheap), $2n$ multiplications, and then a single exponentiation, so that the total number of multiplication is $2n + ExpCost_{Z_N^*}(|e|)$ multiplications. This compares very favorably with our batch verifiers.

Note this test does not provide a batch verifier in the sense of Definition 1. For example, let x be a valid signature of message M and α some value in $Z_N^* - \{1\}$. Then the batch instance $(M, x\alpha), (M, x/\alpha)$ is incorrect, but passes the above test. This is not a problem from the screening perspective, because the property we want here is only that one cannot create such incorrect batch instances without knowing the signatures of the messages in the instance. Indeed, above, we had to know x to create the incorrect instance, meaning M is valid, even if the given signature is not. Thus, this example is not a counter-example to the screening property.

However it may not be a priori clear that our test really has the screening property: maybe there is a clever attack. Below, we show there is not, unless inverting RSA is easy.

CORRECTNESS OF THE SCREEN TEST. Since this is based on the hardness of RSA we first recall the latter, following the concrete treatment of [5]. Fix some prime number e. The RSA generator, $\text{RSA}_{(e)}$, on input 1^k, picks a pair of random distinct $(k/2)$-bit primes p, q such that neither $p-1$ nor $q-1$ are multiplies of e, lets $N = pq$, and computes d so that $ed \equiv 1 \bmod \varphi(N)$. It returns N, e, d. The success probability of an *inverting algorithm* I is the probability that it outputs $y^d \bmod N$ on input N, e, y when N, e, d are obtained by running $\mathrm{RSA}_{(e)}(1^k)$ and $y = x^e \bmod N$ for an x chosen at random from Z_N^*. We say that I (t, ϵ)-breaks $\mathrm{RSA}_{(e)}$, where t: $\mathsf{N} \to \mathsf{N}$ and ϵ: $\mathsf{N} \to [0,1]$, if, in the above experiment, I runs for at most $t(k)$ steps and has success probability at least $\epsilon(k)$. We say that $\mathrm{RSA}_{(e)}$ is (t, ϵ)-secure collection of one-way functions if there is no inverter which (t, ϵ)-breaks $\mathrm{RSA}_{(e)}$.

The following theorem says that if RSA is one-way then an adversary can't produce a batch instance for FDH-RSA SIGNATURE SCREENING TEST that contains a message that was never signed by the signer but still passes the test. Furthermore we indicate the "concrete security" of the reduction. Refer to Section 2.2 for definitions. Note that in our case the treatment there is "lifted" to the random oracle model and we need to consider an additional parameter, namely the number of hash queries by the adversary.

Theorem 4. *Suppose $RSA_{(e)}$ is a (t', ϵ')-secure collection of one-way functions. Let A be an adversary who after a chosen message attack on the FDH-RSA signature scheme outputs a batch instance, for the FDH-RSA signature verification relation, in which at least one message was never legally signed. Suppose this batch instance is of size n; suppose that in the chosen message attack A makes q_{sig} FDH signature queries and q_{hash} hash queries; and suppose the total running time of A is at most $t(k) = t'(k) - \Omega(k^3) \cdot (n + q_{\mathrm{sig}} + q_{\mathrm{hash}})$. Then the probability that* FDH-RSA SIGNATURE SCREENING TEST *accepts the batch instance is at most $\epsilon(k) = \epsilon'(k) \cdot (n + q_{\mathrm{sig}} + q_{\mathrm{hash}})$.*

5 Open Problems

Devise fast batch verification algorithms for modular exponentiation in groups of non-prime order. Perhaps begin by looking at important special cases like Z_p^* where p is prime or Z_N^* where N is an RSA modulus. Also devise such algorithms for the case of modular exponentiation with a fixed exponent rather than a fixed base. Find fast screening algorithms for other signature schemes like DSS. Extend our screen test for FDH-RSA to other RSA based signature schemes like PSS [5] which have tighter security, and try to get tighter reductions of the security of the screen test to that of RSA as a one-way function.

Acknowledgments

The first author was supported by a 1996 Packard Foundation Fellowship in Science and Engineering, and by NSF CAREER Award CCR-9624439.

We thank the (anonymous) referees of Eurocrypt 98 for their comments.

References

1. M. BELLARE, J. GARAY AND T. RABIN. Distributed pseudo-random bit generators— a new way to speed-up shared coin tossing. *Proceedings Fifteenth Annual Symposium on Principles of Distributed Computing*, ACM, 1996.

2. M. Bellare, J. Garay and T. Rabin. Fast batch verification for modular exponentiation and digital signatures. Full version of this paper, available via `http://www-cse.ucsd.edu/users/mihir`, 1998.
3. M. Bellare, J. Garay and T. Rabin. Batch verification with applications to cryptography and checking (Invited Paper), *Latin American Theoretical INformatics 98 (LATIN '98) Proceedings*, LNCS Vol. 1830, C. Lucchesi and A. Moura eds., Springer-Verlag, 1998.
4. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. *First ACM Conference on Computer and Communications Security*, ACM, 1994.
5. M. Bellare and P. Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. *Advances in Cryptology – Eurocrypt* 96 *Proceedings*, LNCS Vol. 1070, U. Maurer ed., Springer-Verlag, 1996.
6. M. Beller and Y. Yacobi. Batch Diffie-Hellman key agreement systems and their application to portable communications. *Advances in Cryptology – Eurocrypt* 92 *Proceedings*, LNCS Vol. 658, R. Rueppel ed., Springer-Verlag, 1992.
7. M. Blum and S. Kannan. Designing programs that check their work. *Proceedings of the* 21st *Annual Symposium on the Theory of Computing*, ACM, 1989.
8. J. Bos and M. Coster. Addition chain heuristics. *Advances in Cryptology – Crypto* 89 *Proceedings*, LNCS Vol. 435, G. Brassard ed., Springer-Verlag, 1989.
9. E. Brickell, D. Gordon, K. McCurley and D. Wilson. Fast exponentiation with precomputation. *Advances in Cryptology – Eurocrypt* 92 *Proceedings*, LNCS Vol. 658, R. Rueppel ed., Springer-Verlag, 1992.
10. E. Brickell, P. Lee and Y. Yacobi. Secure audio teleconference. *Advances in Cryptology – Crypto* 87 *Proceedings*, LNCS Vol. 293, C. Pomerance ed., Springer-Verlag, 1987.
11. A. Fiat. Batch RSA. *Journal of Cryptology*, Vol. 10, No. 2, 1997, pp. 75–88.
12. National Institute for Standards and Technology. Digital Signature Standard (DSS). *Federal Register*, Vol. 56, No. 169, August 30, 1991.
13. C. Lim and P. Lee. More flexible exponentiation with precomputation. *Advances in Cryptology – Crypto* 94 *Proceedings*, LNCS Vol. 839, Y. Desmedt ed., Springer-Verlag, 1994.
14. D. M'Raïhi and D. Naccache. Batch exponentiation - A fast DLP based signature generation strategy. *3rd ACM Conference on Computer and Communications Security*, ACM, 1996.
15. D. Naccache, D. M'Raïhi, S. Vaudenay and D. Raphaeli. Can D.S.A be improved? Complexity trade-offs with the digital signature standard. *Advances in Cryptology – Eurocrypt* 94 *Proceedings*, LNCS Vol. 950, A. De Santis ed., Springer-Verlag, 1994.
16. P. de Rooij. Efficient exponentiation using precomputation and vector addition chains. *Advances in Cryptology – Eurocrypt* 94 *Proceedings*, LNCS Vol. 950, A. De Santis ed., Springer-Verlag, 1994.
17. R. Rubinfeld. Batch Checking with applications to linear functions. *Information Processing Letters*, Vol 42, 1992, pp. 77–80.
18. J. Sauerbrey and A. Dietel. Resource requirements for the application of addition chains modulo exponentiation. *Advances in Cryptology – Eurocrypt* 92 *Proceedings*, LNCS Vol. 658, R. Rueppel ed., Springer-Verlag, 1992.