An extended abstract of this paper appears in *Advances in Cryptology – Eurocrypt 98 Proceedings*, Lecture Notes in Computer Science Vol. 1403, K. Nyberg ed., Springer-Verlag, 1998. This is the full version.

Fast Batch Verification for Modular Exponentiation and Digital Signatures

MIHIR BELLARE[*] JUAN A. GARAY[†] TAL RABIN[‡]

June 1998

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.

---

[*] Department of Computer Science & Engineering, Mail Code 0114, University of California at San Diego, 9500 Gilman Drive, La Jolla, CA 92093, USA. E-mail: `mihir@cs.ucsd.edu`. URL: `http://d-ngnut-p--tp--m`

Contents

1 Introduction **3**
- 1.1 Batch verification . . . 3
- 1.2 Batch verifiers for modular exponentiation . . . 4
- 1.3 Screening: Fast but weak verification for signatures . . . 5
- 1.4 Batch program instance checking and other results . . . 6

2 Definitions **6**
- 2.1 Batch verification . . . 6
- 2.2 Signature screening: weak verification . . . 7
- 2.3 Costs of Multiplication and Exponentiation . . . 7

3 Batch Verification for Modular Exponentiation **8**
- 3.1 Random subset test . . . 8
- 3.2 Computing a product of powers . . . 9
- 3.3 The Small Exponents Test . . . 9
- 3.4 The Bucket Test . . . 11
- 3.5 Prime versus non-prime order . . . 12
- 3.6 Performance . . . 12

4 Fast screening for RSA **13**

5 Open problems **17**

References **17**

A Some Applications **19**
- A.1 Batch verification for DSS signatures . . . 20
- A.2 Discrete-log n-party signature protocols . . . 21

B Batch Verification of Degree of Polynomials **22**

C Batch program instance checking **24**
- C.1 Program checking: Background and issues . . . 24
- C.2 Checking fast programs with slow ones . . . 25
- C.3 Definition . . . 26

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.

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 preprocessing [14, 23] or addition chain heuristics [13, 32, 27]. 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 [18, 25, 8, 24, 34]. 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 $R(\mathit{inst}) \in \{0, 1\}$ 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

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

Figure 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.

algorithm with respect to some fixed public key.) The verification problem for R is: given an instance *inst*, check whether $R(\mathit{inst}) = 1$. In the batch verification problem we are given a sequence $\mathit{inst}_1, \ldots, \mathit{inst}_n$ of instances and asked to verify that for *all* $i = 1, \ldots, n$ we have $R(\mathit{inst}_i) = 1$. The naive way is to compute $R(\mathit{inst}_i)$, 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 V which takes $\mathit{inst}_1, \ldots, \mathit{inst}_n$ and produces a bit as output. We ask that when $R(\mathit{inst}_i) = 1$ for all $i = 1, \ldots, n$, this output be 1. On the other hand, if there is even a single i for which $R(\mathit{inst}_i) = 0$ then we want that $V(\mathit{inst}_1, \ldots, \mathit{inst}_n) = 1$ 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 [14, 23, 13]. Obviously $\mathit{ExpCost}_G^s(k_1) \leq s \cdot \mathit{ExpCost}_G(k_1)$, but there are ways to make it strictly less [14, 23, 13], 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 [20] are a particularly attractive target for batch verification because signing is fast and verification is slow. Naccache *et al.* [25] 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 Appendix A.

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.* [15]. See Appendix A.

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 [11]), 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 Section C for more information including explanations of how this differs from other notions like batch program checking [28].

In Appendix B we provide batch verification algorithms for degrees of polynomials, which has applications in verifiable secret sharing.

The idea of batch verification introduced here was applied in [3] 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 extended abstract of this paper appeared as [4]. An invited talk on batch verification including the material presented in this paper was given at *LATIN '98* [5].

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 $\mathit{inst}_1, \ldots, \mathit{inst}_n$ 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 $R(\mathit{inst}_i) = 1$ for all $i = 1, \ldots, n$, and *incorrect* if there is some $i \in \{1, \ldots, n\}$ for which $R(\mathit{inst}_i) = 0$.

Definition 2.1 A *batch verifier* for relation R is a probabilistic algorithm V that takes as input (possibly a description of R), a batch instance $X = (\mathit{inst}_1, \ldots, \mathit{inst}_n)$ 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 $R(\mathit{inst}_i) \neq 1$, the verifier must reject, except with probability 2^{-l}.

The *naive batch verifier*, or *naive test*, consists of computing $R(\mathit{inst}_i)$ 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.

Variants. Several variants are easily derived, but since we won't use them in this paper we only discuss them briefly. One is an "average case" version of the notion in which the instances are drawn from a distribution. Another is computational batch verification in which it is possible, in principle, to fool the batch verifier, but computationally infeasible to find instances that do so. These notions might be useful in cryptographic settings.

2.2 Signature screening: weak verification

SIGNATURES. A *digital signature scheme*, $(\mathit{Gen}, \mathit{Sign}, \mathit{Verify})$, 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 $(pk, sk) \overset{R}{\leftarrow} \mathit{Gen}(1^k)$ where k is the security parameter. A message is signed via $x \overset{R}{\leftarrow} \mathit{Sign}_{sk}(M)$. A candidate message-signature pair (M, x) is verified by making sure $\mathit{Verify}_{pk}(M, x) = 1$.

SCREENING. The notion was discussed in Section 1.3. We now provide the formalization. Fix a signature scheme $(\mathit{Gen}, \mathit{Sign}, \mathit{Verify})$. 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 $\mathit{ScreenTest}_{pk}((M_1, x_1), \ldots, (M_n, x_n))$ outputs a bit. We want to say what it means for this algorithm to be a good screening algorithm for the signature scheme.

The first requierement is the natural "validity," meaning correct signatures are accepted. That is, if $\mathit{Verify}_{pk}(M_i, x_i) = 1$ for all $i = 1, \ldots, n$ then $\mathit{ScreenTest}_{pk}((M_1, x_1), \ldots, (M_n, x_n))$ outputs 1.

The second requirement is the security. 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.

In more detail, the game is like this. A has oracle access to $\mathit{Sign}_{sk}(\cdot)$. 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 $\mathit{Sign}_{sk}(\cdot)$ 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 $\mathit{ScreenTest}_{pk}((M_1, x_1), \ldots, (M_n, x_n)) = 1$. We let $\mathit{Succ}(A)$ 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 $\mathit{Succ}(A)$ is small for any A whose computation time is not extraordinarily high. In the polynomial-time security framework, we would say that the screen test is secure if $\mathit{Succ}(A)$ is a negligible function of the security parameter k for every probabilistic, polynomial time adversary A. In the theorems (cf. Theorem 4.1) 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 [13, 32], 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 [14, 23] (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 [14, 23], 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 $\mathit{ExpCost}^n_G(t)$.

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.

Lemma 3.1 *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* $1/2$.

Proof: Let $p = |G|$. Since g is a generator of G there exist unique values $x'_1, \ldots, x'_n \in Z_p$ such that $g^{x'_i} = y_i$ for all $i = 1, \ldots, n$. Let $\alpha_i = x_i - x'_i \bmod p$. By assumption there exists an i such that $\alpha_i \neq 0$. For notational simplicity we may assume (wlog) that this is true for $i = 1$. (Note: This does *not* mean we are assuming $\alpha_j = 0$ for $j > 1$. There may be many $j > 1$ for which $\alpha_j \neq 0$.) Now, suppose the test accepts on a particular subset S. Then it must be that $\sum_{i \in S} x_i = \sum_{i \in S} x'_i$, both sums being mod$p$. Thus $\sum_{i \in S} \alpha_i = 0$. Now suppose $T \subseteq \{2, \ldots, n\}$. Then note that

$$\textstyle\sum_{i \in T} \alpha_i = 0 \implies \sum_{i \in T \cup \{1\}} \alpha_i \neq 0 \; .$$

So if the test succeeds on $S = T$ then it must fail on $S = T \cup \{1\}$. This means the test must fail on at least half the sets S. ▌

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 $nl/2 + \mathit{ExpCost}^l_G(k_1)$ 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 3.2 *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 $nl/2 + ExpCost^l_G(k_1)$ 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 $ExpCost^n_G(t) + n - 1$ 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 [14], 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 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.

Theorem 3.3 *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 $l + n(1 + l/2) + ExpCost_G(k_1)$ multiplications, where* $k_1 = |q|$.

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.

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

Proof: First let us see how to get the claim about the performance. Instead of computing $y_i^{s_i}$ individually for each value of i and then multiplying these values, we compute the product $y = \prod_{i=1}^{n} y_i^{s_i}$ directly and more efficiently as $y = \text{FastMult}((y_1, s_1), \ldots, (y_n, s_n))$, the algorithm being that of Section 2.3. Since $s_1, \ldots, s_n$ were random l-bit strings the cost is $l + nl/2$ multiplications on the average. Computing x takes n multiplications. Finally, there is a single exponentiation to the power x, giving the total number of multiplications stated in the theorem.

That the test always accepts when the input is correct is clear. Now we prove the soundness. Let the input $(x_1, y_1), \ldots, (x_n, y_n)$ be incorrect. Let $x'_i = \log_g(y_i)$ for $i = 1, \ldots, n$. For $i = 1, \ldots, n$ let $\alpha_i = x_i - x'_i$. Since the input is incorrect there is an i such that $\alpha_i \neq 0$. For notational simplicity we may assume (wlog) that this is true for $i = 1$. (Note: This does *not* mean we are assuming $\alpha_j = 0$ for $j > 1$. There may be many $j > 1$ for which $\alpha_j \neq 0$.) Now suppose the test accepts on a particular choice of $s_1, \ldots, s_n$. Then

$$g^{s_1 x_1 + \cdots + s_n x_n} \; = \; y_1^{s_1} \cdots y_n^{s_n} \; . \qquad (1)$$

But the right hand side is also equal to $g^{s_1 x'_1 + \cdots + s_n x'_n}$. Hence, we get $g^{s_1 x_1 + \cdots + s_n x_n} = g^{s_1 x'_1 + \cdots + s_n x'_n}$, or $g^{s_1 \alpha_1 + \cdots + s_n \alpha_n} = 1$. Since g is a primitive element of the group, it must be that $s_1 \alpha_1 + \cdots + s_n \alpha_n \equiv$

$0 \bmod q$. But $\alpha_1 \neq 0$. Since q is prime, α_1 has an inverse β_1 satisfying $\alpha_1\beta_1 \equiv 1 \bmod q$. Thus, we can write

$$s_1 \equiv -\beta_1 \cdot (s_2\alpha_2 + \cdots + s_n\alpha_n) \bmod q \ . \tag{2}$$

This means that for any fixed $s_2, \ldots, s_n$, there is exactly one (and hence at most one) choice of $s_1 \in \{0,1\}^l$ (namely that of Equation 2) for which Equation 1 is true. So for fixed $s_2, \ldots, s_n$, if we draw s_1 at random the probability that Equation 1 is true is at most 2^{-l}. Hence the same is true if we draw all of $s_1, \ldots, s_n$ independently at random. So the probability that the test accepts is at most 2^{-l}. ▌

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.

Lemma 3.4 *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 $2^{-(m-1)}$.*

Proof: As in the proof of Theorem 3.3, let $x'_i = \log_g(y_i)$ and $\alpha_i = x_i - x'_i$ for $i = 1, \ldots, n$. We may assume $\alpha_1 \neq 0$. Say that a bucket B_j is *good* ($1 \leq j \leq M$) if $g^{c_j} = d_j$. Let r be the probability, over the choice of $t_1, \ldots, t_n$, that all buckets $B_1, \ldots, B_M$ are good. We claim that $r \leq 1/M = 2^{-m}$.

To see this, first note that if a bucket B_j is good then $\sum_{i \in B_j} \alpha_i \equiv 0 \bmod q$. Now assume $t_2, \ldots, t_n$ have been chosen, so that $(x_2, y_2), \ldots, (x_n, y_n)$ have been allotted their buckets. Let $B'_j = \{\, i > 1 : t_i = j \,\}$— these are the current buckets. Say B'_j is good if $\sum_{i \in B'_j} \alpha_i \equiv 0 \bmod q$. If all of $B'_1, \ldots, B'_M$

are good, then after x_1 is assigned, there is at least one bad bucket, because $\alpha_1 \neq 0$. This means that there exists a j such that B'_j is bad. (This doesn't mean it's the only one, but if there are more bad buckets the test will fail. Thus we can assume that there is a single j.) The probability that $B_1, \ldots, B_M$ are good after x_1 is thrown in is at most the probability that x_1 falls in bucket j, which is $1/M$. So $r \leq 1/M$.

By assumption the test in Step (4) has error at most 2^{-m} so the total error of the atomic bucket test is $2 \cdot 2^{-m} = 2^{-(m-1)}$. ▮

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 3.3. 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 3.5 *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*

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

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 3.3 (and hence of the Bucket Test as given by Theorem 3.5) 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 [14, 23] 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

n	**No. of multiplications used by different tests**			
	Naive	Random Subset	Small Exponents	Bucket
5	1 K	12 K	$\underline{\text{0.4 K}}$	4.3 K
10	2 K	12.5 K	$\underline{\text{0.6 K}}$	4.4 K
50	10 K	13.5 K	$\underline{\text{1.8 K}}$	5 K
100	20 K	15 K	$\underline{\text{3.2 K}}$	5.7 K
200	40 K	18 K	$\underline{\text{6.2 K}}$	7.1 K
500	100 K	27 K	15.2 K	$\underline{\text{10.7 K}}$
1,000	200 K	42 K	30.2 K	$\underline{\text{16.5 K}}$
5,000	1000 K	162 K	150 K	$\underline{\text{56 K}}$

Figure 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.

multiplications [23]. (Using other storage to time tradeoffs as per [23] 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 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. Section A). 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

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

(1) PRUNING: (Remove duplicates) This step returns a sublist $(\overline{M}_1, \bar{x}_1), \ldots, (\overline{M}_{\bar{n}}, \bar{x}_{\bar{n}})$ of the original input list $(M_1, x_1), \ldots, (M_n, x_n)$ with two properties:
  - No Duplicates: $\overline{M}_1, \ldots, \overline{M}_{\bar{n}}$ are distinct
  - Representative: For every $i \in \{1, \ldots, n\}$ there is a $j \in \{1, \ldots, \bar{n}\}$ such that $M_i = \overline{M}_j$.

  See the text for ways to implement this step.

(2) MAIN TEST:

  If $(\prod_{i=1}^{\bar{n}} \bar{x}_i)^e = \prod_{i=1}^{\bar{n}} H(\overline{M}_i) \bmod N$ then return 1 else return 0

Figure 4: *FDH-RSA signature screening test.*

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 d the corresponding decryption exponent. Define functions $f, f^{-1}\colon Z_N^* \to Z_N^*$ 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 [7] 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, [7] appeals to the random oracle paradigm [6] 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. The main test is very simple: it simply multiplies all signatures, then raises the product to the encrpytion exponent, and checks whether or not this equals the product of the hashes of the messages. This main test needs to be preceded by a pruning stage, whose only purpose is to make sure that the messages going into the main test are all distinct.[1]

Note there is no security parameter l in our test: the failure probability of the test is related only to the difficulty of inverting RSA as Theorem 4.1 indicates.

This test is very efficient. In the main test 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.

The pruning problem is essentially that of eliminating duplicates in a given list: for any message $M \in \{M_1, \ldots, M_n\}$ we want to keep exactly one pair of the form (M, x), discarding any other pairs of the form (M, x') (regardless of whether or not $x = x'$). This can be done in a variety of ways via standard data structures and algorithms. Our suggestion is to work with the hashes so that one does not need to process a message (which may be long) more than once. The simplest thing to do is compute the hashes of all the messages, and then sort these values. If there are duplicates in

---

[1] In the preliminary version of this paper [4] we had omitted the pruning step, implicitly assuming (both in the test and in the analysis) that all messages $M_1, \ldots, M_n$ going into the main test were distinct. A fully specified test should not make such an assumption, so we have now added the explicit pre-processing step that guarantees the message distinctness. At Eurocrypt 98, David Naccache has given an example that shows that the main test can fail when the messages are not distinct, indicating that the pruning step is necessary.

this hash list, then keep one representative pair and discard any other pairs of which the message is the same. (The collision intractability of the hash function means that collisions in the hash values correspond to equal messages.) Using such an approach it should be possible to do the pruning with only a logarithmic overhead in time. Note once computed the hashes can be used in the main test; they do not need to be re-computed.

Note that the test is valid, meaning correct signatures are accepted. That is, if $x_i = H(M_i)^d \bmod N$ for all $i = 1, \ldots, n$ then our test accepts with probability one. Our concern is the security, namely what happens when some signatures are invalid.

Also note this test does not provide a batch verifier in the sense of Definition 2.1. 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 [7]. Fix some prime number e. The RSA generator, $\mathrm{RSA}_{(e)}$, 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 $t\colon \mathsf{N} \to \mathsf{N}$ and $\epsilon\colon \mathsf{N} \to [0, 1]$, 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 4.1 *Suppose $\mathrm{RSA}_{(e)}$ 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 $t(k) = t'(k) - \Omega(nk \log(nk)) - \Omega(k^3) \cdot (n + q_{\mathrm{sig}} + q_{\mathrm{hash}})$. Then the probability that* FDH-RSA SIGNATURE SCREENING TEST *accepts the batch instance is at most $\epsilon(k) = \epsilon'(k) \cdot (n + q_{\mathrm{sig}} + q_{\mathrm{hash}})$.*

Proof: Given A we construct an inverter I for the $\mathrm{RSA}_{(e)}$ family and then relate the parameters. I gets input N, e, y and is trying to find $x = y^d \bmod N$. It creates the public key $pk = (N, e)$ and runs $A(pk)$. The latter will make signature queries and hash queries, which I will answer itself, in a manner we will indicate below. Finally A will output some batch instance, from which I will find the desired x. The full description of I follows.

Let $q = n + q_{\mathrm{sig}} + q_{\mathrm{hash}}$. I begins by picking at random $l \in \{1, \ldots, q\}$. It initializes a counter

$c \leftarrow 0$. In the process of answering oracle queries, it builds a table, storing for each message M that is queried a pair (x_M, y_M) of points in Z_N^*, with one special entry: if this is the l-th hash query then x_M is undefined and $y_M = y$. Specifically, a hash oracle query of M is answered by running subroutine $H(M)$ and a sign oracle query is answered by running subroutine $\mathrm{Sign}(M)$, where–

```
Subroutine $H(M)$
$c \leftarrow c+1$
If $c = l$
    then $y(M) \leftarrow y$ ; $M^* \leftarrow M$
    else $x(M) \xleftarrow{R} Z_N^*$ ; $y(M) \leftarrow x(M)^e \bmod N$
return $y_M$
```

```
Subroutine $\mathrm{Sign}(M)$
If $M$ was not previously a hash query
    then $y(M) \leftarrow H(M)$
If $c = l$
    then abort
    else return $x(M)$
```

After all queries have been made and replies obtained, A outputs a batch instance

$$(M_1, x_1), \ldots, (M_n, x_n) \ .$$

I runs the pruning step of the test on this instance to get the pruned sublist $(\overline{M}_1, \bar{x}_1), \ldots, (\overline{M}_{\bar{n}}, \bar{x}_{\bar{n}})$. We know that the messages $\overline{M}_1, \ldots, \overline{M}_{\bar{n}}$ in this list are all distinct but also representative in the sense that any message of the original list is also a message in the new list. This means that if the original list contained a message that was never legally signed, so does the new one, and in addition this message appears only once in the new list. This will be used below.

For any i for which $\overline{M}_i$ was not previously a hash query, I goes ahead and makes a hash query $H(\overline{M}_i)$, so that we may assume hash queries corresponding to $\overline{M}_1, \ldots, \overline{M}_n$ have been made. By assumption there is some $m \in \{1, \ldots, \bar{n}\}$ such that $\overline{M}_m$ was not legally signed, meaning not a sign query, and we fix one such m for the analysis. If $M^* \neq M$ (meaning I did not correctly guess a message that would be in the output batch instance but not legally signed) then I aborts. (We can assume wlog that all hash queries are distinct, so this is not ambiguous. This assumption is made also below.) Else, meaning if $M^* = M$, it sets

$$x \ = \ \frac{\prod_{i=1}^{\bar{n}} \bar{x}_i}{\left[\prod_{i=1}^{m-1} x(\overline{M}_i)\right] \cdot \left[\prod_{i=m+1}^{\bar{n}} x(\overline{M}_i)\right]} \mod N \ . \tag{3}$$

It then outputs x and halts.

Claim. $x^e = y \bmod N$ with probability at least $\mathit{Succ}(A)/q$.

Proof. If the batch instance output by A passes the FDH-RSA signature batch verification test then we know that

$$\left(\prod_{i=1}^{\bar{n}} \bar{x}_i\right)^e \ = \ \prod_{i=1}^{\bar{n}} H(\overline{M}_i) \mod N \ .$$

We know that $H(\overline{M}_i) = y(\overline{M}_i)$ for all $i = 1, \ldots, \bar{n}$. Plug this in and then solve for $y(\overline{M}_m)$ to get

$$y(\overline{M}_m) \ = \ \frac{\prod_{i=1}^{\bar{n}} \bar{x}_i^e}{\left[\prod_{i=1}^{m-1} y(\overline{M}_i)\right] \cdot \left[\prod_{i=m+1}^{\bar{n}} y(\overline{M}_i)\right]} \mod N \ .$$

Let's assume $M^* = \overline{M}_m$, meaning the guess l made by I was correct. Then we know that $x(\overline{M}_i)^e = y(\overline{M}_i)$ for all $i \neq m$. (Notice we use here the fact that $\overline{M}_m$ is not equal to $\overline{M}_i$ for all $i \neq m$, which is guaranteed by the pruning.) From the above we have

$$y(\overline{M}_m) \ = \ \frac{\prod_{i=1}^{\bar{n}} \bar{x}_i^e}{\left[\prod_{i=1}^{m-1} x(\overline{M}_i)^e\right] \cdot \left[\prod_{i=m+1}^{\bar{n}} x(\overline{M}_i)^e\right]} \mod N \ .$$

Now look at Equation 3. With the above it implies $y(\overline{M}_m) = x^e \bmod N$. But if $M^* = \overline{M}_m$ we have $y = y(\overline{M}_m)$ so this means $x^e = y \bmod N$ as desired.

It remains to check that the probability of this event is as claimed. With probability at least $1/q$ the choice of l gives us $M^* = \overline{M}_m$, in which case I does not abort (while answering sign queries or later) and outputs x. Conditional on this event, the view of A while interacting with I is exactly the one it has in the real experiment defining its success, in which it interacts with the real oracles, and we know that its success probability in that experiment is $Succ(A)$. So I succeeds with probability at least $Succ(A)/q$ as claimed. □

To conclude the proof, we notice that the running time of I is at most $t(k)+\Omega(nk\log(nk))+\Omega(k^3)\cdot q$ where $t(k)$ is the running time of A. (The factor of $nk\log(nk)$ comes from the cost of the pruning, which is akin to sorting the hashed values of the messages.) The choice of t makes this at most $t'(k)$. The assumption that $\mathrm{RSA}_{(e)}$ is (t', ϵ')-secure then implies that I is successful with probability at most $\epsilon'(k)$. So by the Claim we have $Succ(A)/q \leq \epsilon'(k)$, meaning $Succ(A) \leq q\epsilon'(k)$. This is exactly what the theorem claims. ▌

5 Open problems

There are a number of good issues for further investigation:

- Devise fast batch verification algorithms for modular exponentiation in groups of non-prime order, and also devise such algorithms for the case of modular exponentiation with a fixed exponent rather than a fixed base. Perhaps begin by looking at important special cases like Z_p^* where p is prime or Z_N^* where N is an RSA modulus.
- 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 [7] 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

We thank the (anonymous) referees of Eurocrypt 98 for their comments. We also thank David Naccache for pointing out that the presentation of our RSA screen test in the preliminary version of this work [4] made the assumption that the messages are distinct. Accordingly we have added the pruning step in Figure 4.

References

[1] L. Adleman and K. Kompella. Fast Checkers for Cryptography. *Advances in Cryptology – Crypto* 90 *Proceedings*, Lecture Notes in Computer Science Vol. 537, A. J. Menezes and S. Vanstone ed., Springer-Verlag, 1990.

[2] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. *Proceedings of the* 33rd *Symposium on Foundations of Computer Science*, IEEE, 1992.

[3] 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.

[4] M. Bellare, J. Garay and T. Rabin. Fast batch verification for modular exponentiation and digital signatures. *Advances in Cryptology – Eurocrypt* 98 *Proceedings*, Lecture Notes in Computer Science Vol. 1403, K. Nyberg ed., Springer-Verlag, 1998.

[5] 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.

[6] 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.

[7] M. Bellare and P. Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. *Advances in Cryptology – Eurocrypt* 96 *Proceedings*, Lecture Notes in Computer Science Vol. 1070, U. Maurer ed., Springer-Verlag, 1996.

[8] M. Beller and Y. Yacobi. Batch Diffie-Hellman key agreement systems and their application to portable communications. *Advances in Cryptology – Eurocrypt* 92 *Proceedings*, Lecture Notes in Computer Science Vol. 658, R. Rueppel ed., Springer-Verlag, 1992.

[9] E. Berlekamp and L. Welch. Error correction of algebraic block codes. US Patent 4,633,470.

[10] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the Correctness of Memories. *Proceedings of the* 32nd *Symposium on Foundations of Computer Science*, IEEE, 1991.

[11] M. Blum and S. Kannan. Designing programs that check their work. *Proceedings of the* 21st *Annual Symposium on the Theory of Computing*, ACM, 1989.

[12] M. Blum, M. Luby, and R. Rubinfeld. Self-Testing/Correcting with Applications to Numerical Problems. *Journal of Computer and System Sciences*, 47:549–595, 1993.

[13] J. Bos and M. Coster. Addition chain heuristics. *Advances in Cryptology – Crypto* 89 *Proceedings*, Lecture Notes in Computer Science Vol. 435, G. Brassard ed., Springer-Verlag, 1989.

[14] E. Brickell, D. Gordon, K. McCurley and D. Wilson. Fast exponentiation with precomputation. *Advances in Cryptology – Eurocrypt* 92 *Proceedings*, Lecture Notes in Computer Science Vol. 658, R. Rueppel ed., Springer-Verlag, 1992.

[15] E. Brickell, P. Lee and Y. Yacobi. Secure audio teleconference. *Advances in Cryptology – Crypto* 87 *Proceedings*, Lecture Notes in Computer Science Vol. 293, C. Pomerance ed., Springer-Verlag, 1987.

[16] D. Chaum and J. van de Graaf. An Improved Protocol for Demonstrating Possession of a Discrete Logarithm and Some Generalizations. *Advances in Cryptology – Eurocrypt* 87 *Proceedings*, Lecture Notes in Computer Science Vol. 304, D. Chaum ed., Springer-Verlag, 1987.

[17] F. Ergun, S. Ravi Kumar, and R. Rubinfeld. Approximate Checking of Polynomials and Functional Equations. *Proceedings of the* 37th *Symposium on Foundations of Computer Science*, IEEE, 1996.

[18] A. Fiat. Batch RSA. *Journal of Cryptology*, Vol. 10, No. 2, 1997, pp. 75–88.

[19] A. Fiat and A. Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. *Advances in Cryptology – Crypto* 86 *Proceedings*, Lecture Notes in Computer Science Vol. 263, A. Odlyzko ed., Springer-Verlag, 1986.

[20] National Institute for Standards and Technology. Digital Signature Standard (DSS). *Federal Register*, Vol. 56, No. 169, August 30, 1991.

[21] P. Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Wigderson. Self-testing/correcting for polynomials and for approximate functions. *Proceedings of the* 23rd *Annual Symposium on the Theory of Computing*, ACM, 1991.

[22] R. Heiman. Secure Audio Teleconference: A Practical Solution. *Advances in Cryptology – Eurocrypt* 92 *Proceedings*, Lecture Notes in Computer Science Vol. 658, R. Rueppel ed., Springer-Verlag, 1992.

[23] C. Lim and P. Lee. More flexible exponentiation with precomputation. *Advances in Cryptology – Crypto 94 Proceedings*, Lecture Notes in Computer Science Vol. 839, Y. Desmedt ed., Springer-Verlag, 1994.

[24] 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.

[25] 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*, Lecture Notes in Computer Science Vol. 950, A. De Santis ed., Springer-Verlag, 1994.

[26] K. Ohta and T. Okamoto. A Modification of the Fiat-Shamir Scheme. *Advances in Cryptology – Crypto 88 Proceedings*, Lecture Notes in Computer Science Vol. 403, S. Goldwasser ed., Springer-Verlag, 1988.

[27] P. de Rooij. Efficient exponentiation using precomputation and vector addition chains. *Advances in Cryptology – Eurocrypt 94 Proceedings*, Lecture Notes in Computer Science Vol. 950, A. De Santis ed., Springer-Verlag, 1994.

[28] R. Rubinfeld. Batch Checking with applications to linear functions. *Information Processing Letters*, 42:77–80, 1992.

[29] R. Rubinfeld. On the Robustness of Functional Equations. *Proceedings of the 35th Symposium on Foundations of Computer Science*, IEEE, 1994.

[30] R. Rubinfeld. Designing Checkers for Programs that Run in Parallel. *Algorithmica*, 15(4):287–301, 1996.

[31] R. Rubinfeld and M. Sudan. Robust Characterizations of Polynomials with Applications to Program Testing. *SIAM Journal on Computing*, 25(2):252–271, 1996.

[32] J. Sauerbrey and A. Dietel. Resource requirements for the application of addition chains modulo exponentiation. *Advances in Cryptology – Eurocrypt 92 Proceedings*, Lecture Notes in Computer Science Vol. 658, R. Rueppel ed., Springer-Verlag, 1992.

[33] D. Stee, L. Strawczynski, W. Diffie, and M. Wiener. A Secure Audio Teleconference System. *Advances in Cryptology – Crypto 88 Proceedings*, Lecture Notes in Computer Science Vol. 403, S. Goldwasser ed., Springer-Verlag, 1988.

[34] S. Yen and C. Laih. Improved Digital Signature suitable for Batch Verification. Technical Report, Dept. of Electrical Engineering, Laboratory of Cryptography and Information Security, December 1994.

A Some Applications

Since verification of modular exponentiations is such a prevalent operation, there are many places where our tests help. In the domain of protocols, for zero-knowledge, bit commitment, verifiable secret sharing, and fancy kinds of digital signatures. In the latter, quite large value of n arise.

We can imagine many settings (e.g., Internet applications, electronic commerce, etc.) in which one has to simultaneously check a number of signatures. For example, one may receive many certificates, containing public keys signed by a certification authority, and one can check all the signatures simultaneously. Or a bank may be signing coins and we have to check a lot of them. So batching is a natural idea in signature verification.

Two things we want to discuss in more depth are batch verification for DSS signatures [20] and improvements to multi-party identification protocols [16] via batching.

A.1 Batch verification for DSS signatures

DSS signatures [20] are a particularly attractive target for batch verification because here signing is fast and verification is slow (it involves two modular exponentiations). Batch verification for DSS itself appears difficult. But Naccache *et al.* [25] introduced a slight variant of DSS for which they were able to give some batch verification algorithms. (This variant has slightly longer signatures than DSS.) We can adapt our tests to apply to this variant, and get faster batch verification algorithms than [25]. (Batch verification of modular exponentiation does not apply directly, because in the verification equation, certain numbers appear both in the base and the exponent, and are intertwined in strange ways, so we have to adapt.)

The DSS scheme. Recall that in DSS, the following are fixed and public, common to all users: a prime p (say 512 bits); a prime q (160 bits) such that q divides $p-1$; and a generator h of Z_p^*. We let $g = h^{(p-1)/q} \bmod p$. This g generates a subgroup of Z_p^* that we call G. Note $G = \{ g^i : i \in Z_q \}$ has prime order q. So the exponents are in a field, namely Z_q.

A particular user has secret key $x \in Z_q$ and public key $y = g^x \in G$. Let $m \in Z_q$ be the message to be signed. (This is actually the hash of the real message.) The signer picks $k \in Z_q^*$ at random, sets $\lambda = g^k \in G$, and sets $r = \lambda \bmod q$. It then lets $s = k^{-1}(m + xr)$, these computations being in the field Z_q. The signature is (r, s), which is 320 bits long. To verify this signature, the verifier, who has m, r, s, computes $w = s^{-1} \in Z_q$ and then checks that $r = g^{mw} y^{rw} \bmod q$. This last is called the verification equation.

Let's look at verification more closely. The computation in the exponents is in Z_q, and then we do the two exponentiations. These are performed in Z_p^*, the big underlying group in which the arithmetic takes place, so they are 512 bit exponentiations, and hence expensive.

DSS*. Naccache *et al.* [25] consider DSS modified as follows. The signature is (λ, s), not (r, s). That is, they forbear from "hashing" λ down to 160 bits. The verification equation now drops the mod q: we check $\lambda = g^{mw} y^{rw}$, operations in Z_p^*. The signature is now $512 + 160 = 672$ bits long. But the "essence" of DSS is preserved. Lets call this DSS*.

For DSS*, Naccache *et al.* [25] give some very nice batch verification algorithms. We can improve them by applying ideas of the Bucket Test.

Batch verification for DSS*. Given a batch instance $(\lambda_1, s_1, m_1), ..., (\lambda_n, s_n, m_n)$, we would like to verify that for all i the pair (λ_i, s_i) is a valid DSS* signature for m_i. We shall slightly reorganize the batch instance in order to simplify the exposition of the test. The batch instance will be $(\lambda_1, a_1, b_1), \ldots, (\lambda_n, a_n, b_n)$ where $a_i = s_i^{-1} m_i$ and $b_i = s_i^{-1} \lambda_i$. Now, the verification of the signature consists of checking that $\lambda_i = g^{a_i} y^{b_i}$. This in fact is not just a notational difference as there is computation involved in changing the batch instance, but these multiplications will be dominated by the computation carried out in the verification and hence do not change the analysis of efficiency for the batch.

We describe in Figure 5 the small exponents test and the bucket test for DSS*. The first is extremely similar to the test of [25], the difference only in the domains from which the exponents are chosen, and the analysis: unlike them, we do not need to assume DSS is unforgeable to prove the test works. The main reason to describe it is only that we need it in the better bucket test that follows. (We drop RS because the test of [25] was better than this anyway.)

Performance. Given that the size of p is 512 bits and q is 160 bits, we have that the Small Exponents Test carried out $480 + 60n$ multiplications mod$p$. The Bucket Test carries out $\frac{60(480+60\cdot 2^m)}{m-1}$ multiplication mod$p$. The analysis of when it is advantageous to use the Bucket Test as opposed to the Small Exponents Test is fairly similar to analysis in the general

GIVEN: DSS public parameters p, q, g, a public key y, and $(\lambda_1, a_1, b_1), \ldots, (\lambda_n, a_n, b_n)$
Also a security parameter $l \leq 159$.

CHECK: That $\forall i \in \{1, \ldots, n\} \, : \, \lambda_i = g^{a_i} y^{b_i}$.

- **Small Exponents (SE) Test:**
  (1) Pick $l_1, \ldots, l_n \in \{0,1\}^l$ at random
  (2) Compute $A = \sum_{i=1}^{n} a_i l_i \bmod q$, $B = \sum_{i=1}^{n} b_i l_i \bmod q$ and $R = \prod_{i=1}^{n} \lambda_i^{l_i}$
  (3) If $R = g^A y^B$ 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 $c_j = \prod_{i \in B_j} \lambda_i$, $d_j = \sum_{i \in B_j} a_i \bmod q$, and $e_j = \sum_{i \in B_j} b_i \bmod q$
  (4) Run the Small Exponent Test on the instance $(c_1, d_1, e_1), \ldots, (c_M, d_M, e_M)$ with security parameter set to m.

Figure 5: *Batch verification algorithms for DSS**

discrete log based batching as the bulk of the computation in both instances is the same, and the fact that in DSS* there is a need for two small exponentiations just gives a factor of two on the small factor of the computation. Thus, we refer to Table 3 for the advantages of the BUCKET TEST over the SMALL EXPONENTS TEST.

CORRECTNESS. For the soundness of these tests, it is important to observe that the group G is of order q which is a prime, and all the computations are being done in G. (Even though the arithmetic must be performed in the bigger group, the operands always stay in the subgroup!) So we can use the same techniques as before to prove that the tests are sound. Details omitted due to page limits.

A.2 Discrete-log n-party signature protocols

Multi-party identification protocols based on the discrete log problem were first considered by Chaum and van de Graaf [16]. One of the applications of these protocols is teleconferencing, where all the participants are connected to a central facility called a *bridge*. The bridge receives signals from the participants, operates on these signal in an appropriate way, and then broadcasts the result back to the participants.[2] In [15], Brickell, Lee and Yacobi present an efficient n-party signature protocol based on discrete log. The protocol, however, requires that each of the participants have k secrets (private keys) in order to achieve an error probability of 2^{-k}. We can apply the batch verification ideas of Section 3 to achieve the same error probability and computation and

---

[2] In the application we consider, the bridge or conference unit does not need to know any security information from the participants. When confidentiality of the teleconference is required, solutions have been proposed [33, 22] that avoid sharing the participants' secrets with the bridge. These solutions are also amenable to batching.

GIVEN: g a generator of G, n users, each with secret x_i; g^{-x_i} is public.
h a collision-resistant hash function. m the message to be signed.
Γ the list of the n users. Also a security parameter l.

CHECK: That all n parties sign message m (participate in the session),
with probability at least $1 - 2^{-l}$.

n-Party Signature Protocol:

(1) Party i, $1 \leq i \leq n$, picks $r_i \in_R Z_p$, computes $a_i = g^{r_i}$, and sends it to the bridge.
(2) The bridge computes $A = \prod_1^n a_i$, and broadcasts it.
(3) Each party i computes $s_1, \ldots, s_n \in \{0,1\}^l$ by computing $h(m, A, \Gamma, j)$, $j = 1, 2, \ldots$
i then computes $y_i = r_i + s_i x_i \bmod p$, and sends it to the bridge.
(4) The bridge computes $Y = \sum_1^n y_i$, and broadcasts it.
(5) Each party i computes $W = \prod_i^n w_i^{s_i}$, and then $Z = g^Y \cdot W$.
(6) If $Z = A$, then OK.

Figure 6: *The n-party signature protocol with small exponents.*

communication costs, while requiring the participants to have only one secret. (We note that n-party identification and signature protocols based on modifications of the Fiat-Shamir identification scheme [19] also require one secret integer [26]. However, as pointed out in [15], shift register technology makes discrete log-based schemes almost an order of magnitude faster in computation time than systems based on the difficulty of extracting L-th roots.)

For simplicity, we sketch the protocol based on the small exponents test. The protocol, shown in Figure 6, has the same general structure as those of [19, 15]. The proof is omitted from this extended abstract. In the proof, we think of h as a random oracle; in practice, it could be instantiated by, say, SHA-1. Depending on the actual values of n and l, h is applied in step (3) as many times as needed in order to obtain the n small exponents. (Alternatively, the exponents could be computed by the bridge and broadcast to everybody.)

B Batch Verification of Degree of Polynomials

Roughly, the problem of checking the degree of a polynomial is as follows: Given a set of points, determine whether there exists a polynomial of a certain degree, which passes through all these points. More formally, let $S \stackrel{\text{def}}{=} (\alpha_1, ..., \alpha_m)$ denote a set of points. We define the relation $\text{DEG}_{\mathcal{F},t,(\beta_1,...,\beta_m)}(S) = 1$ iff there exists a polynomial $f(x)$ such that the degree of $f(x)$ is at most t, and $\forall i \in \{1, .., m\}$, $f(\beta_i) = \alpha_i$, assuming that all the computations are carried out in the finite field $\mathcal{F}$.

Let the batch instance of this problem be $S_1, ..., S_n$, where $S_i = (\alpha_{i,1}, , ..., \alpha_{i,m})$. The batch instance is correct if $\text{DEG}_{\mathcal{F},t,(\beta_1,...,\beta_m)}(S_i) = 1$ for all $i = 1, ..., n$; incorrect otherwise.

The relation DEG can be evaluated by taking $t + 1$ values from the set and interpolating a polynomial $f(x)$ through them. This defines a polynomial of degree at most t. Then verify that all the remaining points are on the graph of this polynomial. Thus, a single verification of the degree requires a polynomial interpolation. Hence, the naive verifier for the batch instance would be highly expensive. The batch verifier which we present here carries out a single interpolation in a field of

Given: $S_1, ..., S_n$ where $S_i = (\alpha_{i,1}, ..., \alpha_{i,m})$; $\beta_1, ..., \beta_n$;
security parameter l; value t.

Check: That $\forall i \in \{1, \ldots, n\}$: $\exists f_i(x)$ such that $deg(f_i) \leq t$ and $f_i(\beta_1) = \alpha_{i,1}, ... f_i(\beta_m) = \alpha_{i,m}$.

Random Linear Combination Test:

1. Pick $r \in_R \mathcal{F}$
2. Compute $\gamma_i \overset{\text{def}}{=} r^n \alpha_{i,n} + \ldots + r\alpha_{i,1}$. (This can be efficiently computed as $(\cdots((r\alpha_{i,n} + \alpha_{i,(n-1)})r + \alpha_{i,(n-2)})\cdots)r + \alpha_{i,1})r$.)
3. If $\text{DEG}_{\mathcal{F},t,(\beta_1,...,\beta_m)}(\gamma_1, ..., \gamma_m) = 1$, then output "correct," else output "incorrect."

Figure 7: *Batch verification algorithm for checking the degree of polynomials.*

size $|\mathcal{F}|$, and achieves a probability of error less than $\frac{n}{|\mathcal{F}|}$. The general idea is that a random linear combination of the shares will be computed. This in return will generate a new single instance of DEG. The correlation will be such that, with high probability, if the single instance is correct then so is the batch instance. Hence, we can solve the batch instance computing a *single* polynomial interpolation, contrasting $O(m^2 n)$ multiplications with $O(mn)$ multiplications.

We will be working over a finite field $\mathcal{F}$ whose size will be denoted by p (not necessarily a prime). [3] We will be measuring the computational effort of the players executing a protocol by the number of multiplications that they are required to perform. Note that the size of the field is of relevance, as the naive multiplication in a field of size 2^k takes $O(k^2)$ steps. We note that the fields in which the computations are carried out can be specially constructed in order to multiply faster. The test (protocol), which we call Random Linear Combination Test, appears in Figure 7.

Theorem B.1 *Assume $\exists j$ such that for all polynomials $f_j(x)$ which satisfy that $\forall i \in \{1, ..., m\}$, $f_j(\beta_i) = \alpha_i$, it holds that the degree of $f_j(x)$ is greater than t. Then* Random Linear Combination Test *is a batch verifier for the relation* $\text{DEG}_{\mathcal{F},t,(\beta_1,...,\beta_m)}(\cdot)$ *which runs in time $O(mn)$ and has an error probability of at most $\frac{n}{p}$.*

Notation: Given a polynomial $f_i(x) = a_m x^m + ... + a_1 x + a_0$, where $a_m \neq 0$, denote by

$$f_i(x)|^{t+1} \overset{\text{def}}{=} a_m x^m + ... + a_{t+1} x^{t+1}.$$

If $m \leq t$, then $f_j(x)|^{t+1} = 0$.

Proof: In order for Random Linear Combination Test to output "correct," it must be the case that $\text{DEG}_{\mathcal{F},t,(\beta_1,...,\beta_m)}(\gamma_1, , ..., \gamma_m) = 1$. Namely, there exists a polynomial $F(x)$ of degree at most t which satisfies all the values in S. Let $f_i(x)$ be the polynomial interpolated by the set S_i; it might be that $deg(f_i) > t$. By definition, the polynomial $F(x) = \sum_{i=1}^{n} r^i f_i(x)$. As $deg(F) \leq t$, it holds that $\sum_{i=1}^{n} r^i f_i(x)|^{t+1}$ must be equal to 0. This is an equation of degree n and hence has at most n roots. In order for Random Linear Combination Test to fail, namely, to output "correct" when in fact the instance is incorrect, r must be one of the roots of the equation. However, this can happen with probability at most $\frac{n}{p}$.

---

[3] At this point we shall assume that the instances are computed in the same field $\mathcal{F}$ as the new instance that we generate. Later we shall show how to dispense with this assumption.

Each linear combination of the shares requires $O(mn)$ multiplications, and the final interpolation requires $O(m^2)$ multiplications. ▌

Batch verification of partial definition of polynomials. A variant of the $\mathrm{DEG}_{\mathcal{F},t,(\beta_1,\ldots,\beta_m)}$ problem is the following: Given the set S as above and a value t, there is an additional value s, and the requirement is that there exists a polynomial $f(x)$ of degree at most t such that for all but s of the values $f(\beta_i) = \alpha_i$. As this is in essence an error correcting scheme, some limitations exist on the value of r. The best known practical solution to this variation is given by Berlekamp and Welch [9]. It requires solving a linear equation system of size m. Hence, again, using a naive batch verifier to check a batch instance would be highly inefficient. RANDOM LINEAR COMBINATION TEST can be modified to solve this variant efficiently as well.

Different fields. It might be the case that the original instances were all computed in a field $\mathcal{F}$ of size p. Yet, $\frac{1}{p}$ is not deemed a small-enough probability of error. Therefore, we create an extension field $\mathcal{F}'$ of the original field, containing $\mathcal{F}$ as a subfield. For example, view $\mathcal{F}$ as the base field and let $\mathcal{F}' = \mathcal{F}[x]/ < r(x) >$ for some irreducible polynomial of the right degree (namely, of a degree big enough to make $\mathcal{F}'$ of the size we want). Thus, if $\mathcal{F} = GF(2^k)$ we will get $\mathcal{F}' = GF(2^{k'})$, for some $k' > k$, and the former is a subfield of the latter. It must be noted that if the extension field is considerably larger than the original field, then the computations in the extension field are more expensive. Thus, in this case there is a trade-off between using the sophisticated batch verifier and using the naive verifier.

C Batch program instance checking

We introduce the notion of batch instance checking and show how to achieve it using batch verification. We begin with some background and motivation, present the approach, and conclude with the formal definition of the notion.

C.1 Program checking: Background and issues

Let f be a function and P a program that supposedly computes it. A program checker, as introduced by Blum and Kannan [11], is a machine C which takes input x and has oracle access to P. It calls the program not just on x but also on other points. If P is correct, meaning it correctly computes f at all points, then C must accept x, but if $P(x) \neq f(x)$ then C must reject x with high probability.

Program checking has been extensively investigated, and checkers are now known for many problems [11, 1, 10, 21, 12, 29, 30, 17]. Checking has also proven very useful in the design of probabilistic proofs [31, 2].

Batch program checking was introduced by Rubinfeld [28]. Here the checker gets many instances $x_1, \ldots, x_n$. Again if P is entirely correct the checker must accept. And if $P(x_i) \neq f(x_i)$ for some i the checker must reject with high probability. Rubinfeld provides batch verifiers for linear functions. (Specifically, the mod function.) A similar notion is used by Blum *et al.* [10] to check programs that handle data structures.

THE LITTLE-OH CONSTRAINT. To make checking meaningful, it is required that the checker be "different" from the program. Blum captured this by asking that the checker run faster than any algorithm to compute f, formally in time little-oh of the time of any algorithm for f.

We will see that with our approach, we will use a slow program as a tool to check a fast one. Nonetheless, the checker *will* run faster than any program for f, so that Blum's constraint will be met.

PROBLEMS WITH CHECKING. Program checking is a very attractive notion, and some very elegant and useful checkers have been designed. Still the notion, or some current implementations, have some drawbacks that we would like to address:

- *Good results can be rejected:* Suppose P is correct on some instances and wrong on others. In such a case, even if $P(x)$ is correct, the checker is allowed to (and might) reject on input x. This is not a desirable property. It appears quite plausible, even likely, that we have some heuristic program that is correct on some but not all of the instances. We would like that whenever $P(x)$ is correct the checker accepts, else it doesn't. (As usual it is to be understood that in such statements we mean with high probability in both cases.) This is to some extent addressed by self-correction [12], but that only works for problems which have a nice algebraic structure, and needs assumptions about the fraction of correct instances for a program.
- *Checking is slow:* Even the best known checkers are relatively costly. For example, just calling the program twice to check one instance is costly in any real application, yet checkers typically call it a constant number of times to just get a constant error probability, meaning that to get error probability 2^{-l} the program might be invoked $\Omega(l)$ times. Batch checking improves on this to some extent, but, even here, to get error 2^{-l}, the mod function checker of [28] calls the program $\Omega(nl)$ times for n instances, so that the amortized cost per instance is $\Omega(l)$ calls to the program, plus overhead.

WHAT TO CHECK? We remain interested in designing checkers for the kinds of functions for which checkers have been designed in the past. For example, linear functions. The approach discussed below applies to any function, but to be concrete we think of f as the modular exponentiation function. This is a particularly interesting function because of the wide usage in cryptography, so that fast checkers would be particularly welcome.

C.2 Checking fast programs with slow ones

OUR APPROACH. To introduce our approach let us go back to the basic question. Let f be the function we want to check, say modular exponentiation. Why do we want to check a program P for f? Why can't we just put the burden on the programmer to get it right? After all modular exponentiation is not *that* complicated to code if you use the usual (simple, cubic time) algorithm. It should not be too hard to get it right.

The issue is that we probably do NOT want to use the usual algorithm. We want to design a program P that is faster. To achieve this speed it will try to optimize and cut corners in many ways. For example, it would try heuristics. These might be complex. Alternatively, it might be implemented in hardware. Now, we are well justified in being doubtful that the program is right, and asking about checking.

Thus, we conclude that it is reasonable to assume that it is not hard to design a reliable but slow program P_{slow} that correctly computes f on all instances. Our problem is that we have a fast but possibly unreliable program P that claims to compute f, and we want to check it.

Thus, a natural thought is to use P_{slow} to check P. That is, if $P(x)$ returns y, check that $P_{\text{slow}}(x)$ also returns y. Of course this makes no sense. If we were willing to invest the time to run P_{slow} on each instance, we don't need P anyway. Formally, we have violated the little-oh property: our checker is not faster than all programs for f, since it is not faster than P_{slow}.

However, what we want is to essentially do the above in a meaningful way. The answer is batching. However we will not do batch program checking in the sense of [28]. Instead we will be batch-verifying the outputs of P, using P_{slow}, and without invoking P at all.

More precisely, define the relation R, for any *inst* $= (x, y)$, by $R(x, y) = 1$ iff $f(x) = y$. Let's assume we could design a batch verifier V for R, in the sense of Section 1.1. (Typically, as in our later designs, V will make some number of calls to P_{slow}. But MUCH fewer than n calls, since its running time is less than n times the time to compute R.) Our program checker is for a batch instance $x_1, \ldots, x_n$. Say we have the outputs $y_1 = P(x_1), \ldots, y_n = P(x_n)$ of the program, and want to know if they are correct. We simply run V on the batch instance $(x_1, y_1), \ldots, (x_n, y_n)$ and accept if V returns one. The properties of a batch verifier as defined in Section 1.1 tells us the following. If P is correct on all the instances $x_1, \ldots, x_n$, then we accept. If P is wrong on any one of these instances then we reject. Thus, we have a guarantee similar to that of batch program checking (but a little stronger as we will explain) and at lower cost.

Since V makes some use of P_{slow} we view this as using a slow program to check a fast one.

FEATURES OF OUR APPROACH. We highlight the following benefits of our batch program checking approach:

- *Instance correctness:* In our approach, as long as P is correct on the specific instances $x_1, \ldots, x_n$ on which we want results, we accept, even if P is wrong on other instances. (Recall from the above that usual checkers can reject even when the program is correct on the instance in question, because it is wrong somewhere else, and this is a drawback.) In this sense we have more a notion of "program instance checking."
- *Speed:* In our approach, the program is called only on the original instances, so the number of program calls, amortized, is just one! Thus, we only need to worry about the overhead. However, with good batch verifiers (such as we will later design), this can be significantly smaller than the total running time of the program on the n calls. Thus the amortized additional cost of our checker is like $o(1)$ program calls, and this is to achieve *low* error, not just constant error. This is very fast.
- *Off-line checking:* Our checking can be done off-line as in [10]. Thus, for example, we can use (slow) software to check (fast) hardware.

Of course batching carries with it some issues too. When an error is detected in a batch instance $(x_1, y_1), \ldots (x_n, y_n)$ we know that some (x_i, y_i) is incorrect but we don't know which. There are several ways to compensate for this. First, we expect to be in settings where errors are rare. (As bugs are discovered they are fixed, so we expect the quality of P to keep improving.) In some cases it is reasonable to discard the entire batch instance. (In cryptographic settings, we are often just trying to exponentiate random numbers, and can throw away one batch and try another.) Alternatively, figure out the bad instance off line; if you don't have to do it too often, it can be OK.

C.3 Definition

We conclude by summarizing the formal definition of our notion of batch program instance checking. Similarly to relations, a batch instance for a (not necessarily boolean) function f is simply a sequence $X = x_1, \ldots, x_n$ of points in its domain. A program P is correct on X if $P(x_i) = f(x_i)$ for all $i = 1, \ldots, n$, and incorrect if there is some $i \in \{1, \ldots, n\}$ such that $P(x_i) \neq f(x_i)$. If f is a function we let R_f be its graph, namely the relation $R_f(x, y) = 1$ if $f(x) = y$, and 0 otherwise. Notice that P is correct on X iff $(x_1, P(x_1)), \ldots, (x_n, P(x_n))$ is a correct instance of the batch verification problem for R_f.

Definition C.1 A *batch program instance checker* for f is a probabilistic oracle algorithm C^P that takes as input (possibly a description of f), a batch instance $X = (x_1, \ldots, x_n)$ for f, and a security parameter l provided in unary. It satisfies:
(1) If P is correct on X then C^P outputs 1.
(2) If P is incorrect on X then the probability that C^P outputs 1 is at most 2^{-l}.

We wish to design such batch program instance checkers which have a very low complexity and make only marginally more than n oracle calls to the program. As indicated above, this is easily done for a function f if we have available batch verifiers for R_f, so we have such checkers for modular exponentiation as considered in Section 3.