Batch Verification with Applications to Cryptography and Checking

Garay, Juan A.
Bellare, Mihir
Rabin, Tal

(Invited Paper)

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

[1] Department of Computer Science & Engineering, Mail Code 0114,
University of California at San Diego, 9500 Gilman Drive, La Jolla, CA 92093, USA.
mihir@cs.ucsd.edu http://www-cse.ucsd.edu/users/mihir
[2] IBM T.J. Watson Research Center,
PO Box 704, Yorktown Heights, New York 10598, USA.
{garay,talr}@watson.ibm.com http://www.research.ibm.com/security

Abstract. Let $R(\cdot)$ be a polynomial time-computable boolean relation. Suppose we are given a sequence $inst_1, \ldots, inst_n$ of instances and asked whether it is the case that $R(inst_i) = 1$ for *all* $i = 1, \ldots, n$. The naive way to figure out the answer is to compute $R(inst_i)$ for each i and check that we get 1 each time. But this takes n computations of R. Can one do any better?

The above is the "batch verification" problem. We initiate a broad investigation of it. We look at the possibility of designing probabilistic batch verifiers, or tests, for basic mathematical relations R. Our main results are for modular exponentiation, an expensive operation in terms of number of multiplications: here g is some fixed element of a group G and $R(x, y) = 1$ iff $g^x = y$. We find surprisingly fast batch verifiers for this relation. We also find efficient batch verifiers for the degrees of polynomials.

The first application is to cryptography, where modular exponentiation is a common component of a large number of protocols, including digital signatures, bit commitment, and zero knowledge. Similarly, the problem of verifying the degrees of polynomials underlies (verifiable) secret sharing, which in turn underlies many secure distributed protocols.

The second application is to program checking. We can use batch verification to provide faster batch checkers, in the sense of [20], for modular exponentiation. These checkers also have stronger properties than standard ones, and illustrate how batch verification can not only speed up how we do old things, but also enable us to do new things.

* Work supported in part by NSF CAREER Award CCR-9624439 and a 1996 Packard Foundation Fellowship in Science and Engineering.

1 Introduction

We suggest the notion of batch verification. Based on this we suggest and implement a new paradigm for program checking [7]. We also suggest applications in cryptography. Motivated by this we design batch verifiers for some particular functions of interest in these domains.

1.1 Batch Verification

Let R be a (polynomial time-computable, boolean) relation. 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 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 Application Domains

Batch verification will be useful in any algorithmic setting where there are repetitive tasks. Before presenting our results and the particular applications that ensue, let us briefly discuss two concrete application domains that have motivated our work.

Cryptography. 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. A setting where batch verification is useful is in the verification of digital signatures. For example, the validity of a sequence of *electronic coins* needs to be verified by checking the bank's signature on each coin. When there are lots of coins, batch verification will help. Similarly one may receive many *certificates*, containing public keys signed by a certification authority, and one can check all the signatures simultaneously.

Beyond this, batch verification is useful for a large number of standard cryptographic protocols. These protocols typically involve repetition of some operation, such as a committal, done for example via the discrete exponentiation function $x \mapsto g^x$ in a group with generator g, so that a party commits to x by providing $y = g^x$, and later de-commits by revealing x. At this point, someone must check that indeed $y = g^x$. In a zero-knowledge protocol, thousands or more committals are being performed simultaneously, and batch verification will be useful. The same is true for other standard "cut-and-choose" type protocols, for example for key escrow.

We also provide fast batch verification methods for degrees of polynomials, which have applications in verifiable secret sharing and other robust distributed tasks. We elaborate on more specific applications of our results in this domain in Sect. 1.5.

Program checking. The notion of batch verification has on the face of it nothing to do with program checking: as Sect. 1.1 indicates, 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, has the following benefits: it permits fast checking; and it permits instance checking, not just program checking, in the sense that a correct result is not rejected just because the program might be wrong on some other instance. (The last is in contrast to standard program checking.) 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. We do not elaborate here, but Sect. 3 presents in more detail both the approach and the background, including explanations of how this differs from other notions like batch program checking [20]. Now we move on to the design of batch verifiers.

1.3 Batch Verifiers for Modular Exponentiation

We have been able to design some surprisingly efficient batch verifiers for modular exponentiation. By the approach of Sect. 3, these translate into fast batch program instance checkers. In particular the amortized (per instance) cost of our checkers is significantly lower than that of [1].

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; multiplication (the group operation) will be our basic operation by which we shall compute costs.

Folklore techniques yield a first, basic test that we include for completeness, calling it the Random SubsetTest. Our main results are two better tests, the SmallExponentsTest and the BucketTest. They are presented, with analysis of correctness, in Sect. 5. Their performance is summarized in Fig. 1, with the naive test listed for comparison. We explain the notation used in the Fig.: $k_1 = \lg(|G|)$; $\mathit{ExpCost}_G(k_1)$ 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 $\mathit{ExpCost}^s_G(k_1)$ is the cost of computing s different such exponentiations. (Obviously $\mathit{ExpCost}^s_G(k_1) \leq s \cdot \mathit{ExpCost}_G(k_1)$, but there are ways to make it

Test	**No. of multiplications**
Naive	$ExpCost_G^n(k_1)$
Random Subset (RS)	$nl/2 + ExpCost_G^l(k_1)$
Small Exponents (SE)	$l + nl/2 + ExpCost_G(k_1)$
Bucket	$\min_{m\geq 2} \left\lceil \frac{l}{m-1} \right\rceil \cdot (n + m + 2^{m-1}m + ExpCost_G(k_1))$

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}. Here n is the number of instances to be checked, $k_1 = \lg(|G|)$, l is the security parameter, $ExpCost_G(k_1)$ is the number of multiplications required to perform a single exponentiation with a k_1-bit exponent, and $ExpCost_G^s(k_1)$ is the number of multiplications to perform s such exponentiations. See text for explanations.

strictly less [10, 17, 9], which is why it is separate parameter. Under the normal square-and-multiply method, $ExpCost_G(k_1) \approx 1.5k_1$ multiplications in the group, but again, it could be less [10, 17, 9]. See Sect. 4 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.

Fig. 4 in Sect. 5.4 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 SmallExponentsTest is better, while for larger values, BucketTest 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.

Exponentiation with common exponent. Above, we consider exponentiation to a fixed base g. Another version of the problem is when the exponent is fixed, and the relation becomes $\mathrm{BASE}_{G,v}(x, y) = 1$ iff $x^v = y$ in G, where G is some appropriate underlying group. (This is the kind of verification that is needed for the RSA function [19], which has the form $x \mapsto x^v$ with $G = Z_N^*$.) The results discussed above do not apply to this version. (Actually the tests are easily adapted, but the correctness is a different story. It turns out the natural adaptations don't work.) In the full paper we suggest a different notion of batch verification for this case which we call "screening."

1.4 Batch Verification of Degrees 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 $\mathcal{F}$.

A single verification of the degree of one polynomial requires one polynomial interpolation. Hence, the naive verifier for the batch instance would be very expensive. The batch verifier which we present allows for the verification of multiple (exponentially many in k, for a field of size 2^k) polynomials at the same cost of a single polynomial interpolation. The general idea underlying the batch verifier is to compute a random linear combination of the shares corresponding to the various polynomials. This in turn generates a new single instance of the problem. The correlation is such that, with high probability, if the single instance is correct then so is the batch instance.

1.5 Applications

Our results for modular exponentiation immediately apply to any discrete log-based protocol in which discrete exponentiation needs to be verified. In some cases, we need to tweak the techniques.

DSS signatures [15] are a particularly attractive target for batch verification because signing is fast and verification is slow. Naccache *et al.* [18] were able to give some batch verification algorithms for a slight variant of DSS. In the full paper we show how to adapt our tests to apply to this variant, and get faster batch verification algorithms than the ones in [18].

Many popular zero knowledge or witness-hiding proofs are based on discrete logarithms. For example, discrete exponentiation may be used to implement bit commitment, and such protocols typically involve a lot of bit commitments. Verifying the de-commitments corresponds to verification of modular exponentiation, and the use of our batch verifiers can speed up this process.

We also improve the discrete log-based n-party signature/identification protocols of Brickel *et al.* [11]. 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.

The problem of checking the degrees of polynomials has wide applications in the fields of fault-tolerant and secure distributed computation, where some of the participants may be (maliciously) faulty. Roughly, the ability of the good players to verify the existence of a valid interpolating polynomial through points that are distributed among the participants, is a basic building block for Verifiable Secret Sharing (VSS) [12]. VSS, in turn, enables fundamental distributed primitives such as shared coins, Byzantine agreement, broadcast channel, and secret balloting and voting. In [3] we use the techniques for the batch verification of this relation to construct a very efficient shared coin tossing scheme.

1.6 Related Work

There has been a lot of previous work on speeding up the modular exponentiation operation itself, for example by pre-processing (Brickell *et al.* [10], Lim and Lee [17] and others) or addition chain heuristics (Bos and Coster [9], Saerbrey and Dietel [24]). These works provide faster ways to do modular exponentiations. 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 Fiat's batch RSA [14], Naccache *et al.*'s batch verification for a variant of DSS [18], and Beller and Yacobi's batch Diffie-Hellman key agreement [4]. 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 indicate 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.

In the context of program checking, batch program checking was introduced by Rubinfeld [20]. 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.* [6] to check programs that handle data structures. In this paper we introduce the notion of batch instance checking and show how to achieve it using batch verification.

1.7 Organization of the Paper

The remainder of the paper is organized as follows. In Sect. 2 we formalize the notion of batch verification. Sect. 3 is devoted to our approach to program checking; this section is somewhat independent from the rest of the paper, so a reader only interested in the algorithmic techniques can directly proceed to Sect. 4, where we discuss the costs of multiplication and exponentiation. In Sect. 5 we present our batch verifiers for modular exponentiation, while in Sect. 6 we treat the batch verification of degrees of polynomials.

2 The Notion of Batch Verification

Here we provide a formal definition of the notion, extending the discussion in Sect. 1.1. 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 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 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}.

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. But this takes n computations of R. We want to do better. The goal is to design batch verifiers for various relations which are efficient compared to the naive verifier. We will always seek to have a low error $\epsilon = 2^{-l}$, controlled by a security parameter l. In practice, setting l to be about 60 will suffice.

The above is a worst-case notion. Sometimes we might be interested in a more "average case" version. For example, say $R = R_f$ is the graph of some function f, meaning $R_f(x, y) = 1$ iff $f(x) = y$. We might be in a setting where in each instance $inst_i = (x_i, y_i)$ we know that x_i is uniformly distributed. We still want to check that indeed $y_i = f(x_i)$. The batch verifier need only work for instances drawn from a distribution where each x_i is chosen independently and uniformly. This can happen in a cryptographic protocol where one party chooses $x_1, \ldots, x_n$ at random, another party computes $y_1, \ldots, y_n$, and the first party must check that $f(x_i) = y_i$ for all $i = 1, \ldots, n$. For example, say $f(x) = g^x$ is exponentiation in some group of which g is a generator; then this kind of thing does arise in zero-knowledge protocols.

In the above it is impossible to fool the batch verifier except with low probability. We are also interested in a weaker notion under which it is possible, in principle, to fool the batch verifier, but computationally infeasible to find instances that do so. This notion, called computational batch verification, is again useful in cryptographic settings where we might not be able to design full-fledged batch verifiers but are able to do so under the assumption that the underlying cryptosystem can't be broken.

Let $R = \{R_d\}_{d \in D}$ be a family of relations over an index set D. Associated to D is some probability distribution. The batch verifier $V(\cdot, \cdot)$ gets input d, a batch instance X for R_d, and a security parameter l. It outputs a bit $V(d, X, l)$. We consider an algorithm A that given d, l tries to produce batch instances that fool V. Let

$$\mathsf{Pass}(A, V, R, l) = \Pr[\, d \leftarrow D \; ; \; X \leftarrow A(d, l) \; : \; X \text{ is incorrect but } V(d, X, l) = 1]$$

be the probability that V accepts even though the instance is incorrect. The probability is over the coins of both A and the test V. We want to say this probability is small as long as A is not allowed too much computing time.

Definition 2. A *computational batch verifier* for relation family $\{R_d\}_{d \in D}$ is a probabilistic algorithm V that takes as input d, a batch instance $X = (inst_1, \ldots, inst_n)$ for R_d, and a security parameter l provided in unary. V is said to be (t, m, ϵ)-reliable if the following are true:

(1) If X is correct then V outputs 1.
(2) $\mathsf{Pass}(A, V, R, l) \leq \epsilon$ for any algorithm A running in time at most t.

Here t, m, ϵ may be functions of $|d|, l$.

3 Batch Program Instance Checking

In this section 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.

3.1 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 [7], 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 [7, 1, 6, 16, 8, 21, 22, 13]. Checking has also proven very useful in the design of probabilistic proofs [23, 2].

Batch program checking was introduced by Rubinfeld [20]. 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.* [6] 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 [8], 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 [20] 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.

3.2 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 $P_{\mathrm{slow}}(x)$ 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 [20]. 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 Sect. 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 Sect. 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 [6]. 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, one can figure out the bad instance off line; this may be acceptable if it doesn't have to be done too often.

3.3 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 3. 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 concentrate on getting efficient batch verifiers.

4 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.

Cost of one exponentiation. 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. However, there are methods to do better. For example, using the windowing method based on addition chains [9, 24], 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 [10, 17] (a range of values can be obtained here; we give some numerical examples in Sect. 5.4). Accordingly it is best to treat the cost of exponentiation as a parameter. We let $ExpCost_G(k_1)$ 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.

Multiple exponentiations. 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 $n \cdot ExpCost_G(t)$ multiplications. However, it is possible to do better, via the techniques of [10, 17], 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 $ExpCost_G^n(t)$.

Computing the product of powers. We now present a general algorithm we will use in Sect. 5 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 some 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 $ExpCost_G^n(t) + n - 1$ 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 [10], we can do better, as shown in Fig. 2. This algorithm performs 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.

GIVEN: $a_1, \ldots, a_n \in G$; $b_1, \ldots, b_n$ integers in the range $0, \ldots, 2^t - 1 < |G|$.
COMPUTE: $a = \prod_{i=1}^n a_i^{b_i}$.

```
Algorithm FastMult((a_1, b_2), ..., (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 · a_i;
            a := a^2
return a
```

Fig. 2. *Fast algorithm for computing the product of powers.*

5 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.

We are interested in groups arising in cryptography for which the discrete log problem (computing $\log_g(y)$ given y) is hard. This is not an assumption needed for our results (in particular we do not use any hardness assumptions), it is rather the motivation. In this category what is important is that k_2 is quite large, about $k_2 = 1024$. In comparing complexities we think of k_2 as about this much.

With G, g fixed we want to construct fast batch verifiers for the relation $\mathrm{EXP}_{G,g}$. We begin with a simple test which, although better than the naive method, is not so efficient.

5.1 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.

This test seems simple enough that it might be viewed as folklore. Its analysis is quite simple, and we skip the proofs but state the results, for comparison with our later better methods.

Lemma 4. *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 $1/2$.* □

This lemma tells us that the test does work, but not too well, in the sense that the error is not small, but a constant, namely $1/2$. (Moreover, one can show that this analysis is best possible.) So to lower the error to the desired 2^{-l} we must repeat the atomic test independently l times. We call this the Random SubsetTest. See Fig. 3. 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 5. *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* $k_1 = \lceil \lg(|G|) \rceil$. □

5.2 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: setting $t = l$ yields the desired error 2^{-l}. The corresponding test is the Small Exponents (SE) Test and is depicted in Fig. 3. The following theorem summarizes its properties and provides the analysis proving our claim about the error.

Theorem 6. *Given a group G of prime order q and a generator g of G, then* SmallExponentsTest *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* $k_1 = |q|$.

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 $y = \mathrm{FastMult}((y_1, s_1), \ldots, (y_n, s_n))$, the algorithm being that of Sect. 4. 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

$$g^{s_1 x_1 + \cdots + s_n x_n} = y_1^{s_1} \cdots y_n^{s_n} . \tag{1}$$

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, on the instance $(c_1, d_1), \ldots, (c_M, d_M)$, the Small Exponent Test with security parameter set to m.

Fig. 3. *Batch verification algorithms for exponentiation with a common base.*

But the right hand side is also equal to $g^{s_1 x'_1 + \cdots + s_n x'_n}$. Therefore, 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}. □

Remark 7. We stress that this result holds in a group of *prime order*. We are not working in Z_q^* (which has order $q-1$) but in a group G which has order q a prime. (If the group does not have prime order, it is easy to find examples to show that our tests don't work.) 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.

5.3 The Bucket Test

We saw that the SmallExponentsTest 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 BucketTest, shown in Fig. 3, 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 SE test to solve the smaller problem as has been described in Fig. 3. (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 SE.) 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 BucketTest 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.

Lemma 8. *Suppose G is a group of prime order p, 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 Thm. 6, 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 Thm. 6. 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 9. *Given a group G of prime order q, and a generator g of G, the* BucketTest *(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 in the next section.

5.4 Performance Analysis

We look at the actual performance of the batch verification tests of Fig. 3. For a given value of n (the number of instances we are simultaneously verifying), exactly how much work does each test need, and which is the best? In particular we don't want to end up with results that are purely asymptotic, ie. the improvement is only for very large n. For $n = 5$ or $n = 10$, what happens? And how does it grow?

To measure this, we count exactly the number of (k_2-bit) multiplications used by each test. These numbers are also tabulated in Fig. 1. Let us fix some reasonable values for k_1 and the security parameter l: set $k_1 = 1024$, and $l = 60$.

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. 4. *Example:* For increasing values of n, we list the number of multiplications (in thousands, rounded up) for 1024-bit exponents for each method to verify n exponentiations with error probability 2^{-60}. We assume that a single exponentiation requires 200 multiplications [17]. The lowest number for each n is underlined: notice how it is not always via the same test!

(Meaning the exponentiation is for 1024 bit moduli, and the error probability will be 2^{-60}.) For various values of the number n of terms in the batch instance, we compare the number of multiplications each test takes. We compare it to the results of [10, 17] as they seem harder to beat.[1] These results are tabulated in Fig. 4. We stress that these savings occur also if other methods for computing exponentiations are used.

We find that the speedups provided by our tests are real. First, observe that even for small values of n, we can do much better than naive: at $n = 5$ the SE test is a factor of 2 better than naive. Also observe that which test is better depends on the value of n. (In the figure, we underline the best for each value of n.) As we expected, the RS test is actually *worse* than naive for small n. Until n about 200, the SmallExponentsTest test is the best. From then on, the BucketTest performs better. Note that the factor of improvement increases: at $n = 200$ we can do about 6 times better than naive (using SE); at $n = 5000$, about 17 times better (using Bucket).

Another relevant value for k_1 is 160. Suppose $n = 40$, and one would be happy with $l = 40$. Using the methods of [17] would require about 1700 multiplications.

[1] Lim and Lee [17] present different configurations to perform exponentiation with precomputation that trade-off the number of multiplications with storage. The estimate of 200 multiplications corresponds to an intermediate configuration, with an acceptable storage requirement (300 pre-computed values). Their fastest configuration—with a considerable storage blow-up—uses ≈ 100 multiplications. Still in this case our tests perform consistently better.

On the other hand, SmallExponentsTest uses 1080, and using plain square-and-multiply. Combining SmallExponentsTest with pre-processing with reasonable storage brings the number of multiplications below 900.

In other words, using these tests can bring sizeable speedups in any setting where we need to perform over five modular exponentiations simultaneously, and as n increases the savings get even larger.

6 Batch Verification of Degree of Polynomials

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 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). [2] 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 CombinationTest, appears in Fig. 5.

[2] 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.

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 $\gamma_i \stackrel{\text{def}}{=} r^n \alpha_{i,n} + \ldots + r\alpha_{i,1}$.
(* 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 $\text{DEG}_{\mathcal{F},t,(\beta_1,\ldots,\beta_m)}(\gamma_1, ..., \gamma_m) = 1$, then output "correct," else output "incorrect."

Fig. 5. *Batch verification algorithm for checking the degree of polynomials.*

Theorem 10. *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 LINEARCOMBINATIONTEST *is a batch verifier for the relation $\text{DEG}_{\mathcal{F},t,(\beta_1,\ldots,\beta_m)}(\cdot)$ 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 $a_m \neq 0$,

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

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

Proof. In order for RANDOM LINEARCOMBINATIONTEST to output "correct," it must be the case that $\text{DEG}_{\mathcal{F},t,(\beta_1,\ldots,\beta_m)}(\gamma_1,, ..., \gamma_m) = 1$. 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 LINEARCOMBINATIONTEST 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 $\frac{n}{p}$.

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. Consider the following variant of the $\mathrm{DEG}_{\mathcal{F},t,(\beta_1,\ldots,\beta_m)}$ problem: 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 [5]. 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 LINEARCOMBINATIONTEST 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.

References

1. L. Adleman and K. Kompella. Fast Checkers for Cryptography. In A. J. Menezes and S. Vanstone, editors, *Advances in Cryptology — Crypto '90*, pages 515–529, Berlin, 1990. Springer-Verlag. Lecture Notes in Computer Science No. 537.
2. S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proc. 33rd Annual Symposium on Foundations of Computer Science*, pages 14–23. IEEE, 1992.
3. M. Bellare, J. Garay, and T. Rabin. Distributed Pseudo-Random Bit Generators—A New Way to Speed-Up Shared Coin Tossing. In *Proceedings Fifteenth Annual Symposium on Principles of Distributed Computing*, pages 191–200. ACM, 1996.
4. M. Beller and Y. Yacobi. Batch Diffie-Hellman Key Agreement Systems and their Application to Portable Communications. In R. Rueppel, editor, *Advances in Cryptology — Eurocrypt '92*, pages 208–220, Berlin, 1992. Springer-Verlag. Lecture Notes in Computer Science No. 658.
5. E. Berlekamp and L. Welch. Error correction of algebraic block codes. US Patent 4,633,470.
6. M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the Correctness of Memories. In *Proceeding 32nd Annual Symposium on the Foundations of Computer Science*, pages 90–99. IEEE, 1991.
7. M. Blum and S. Kannan. Designing Programs that Check their Work. In *Proceedings 21st Annual Symposium on the Theory of Computing*, pages 86–97. ACM, 1989.

8. 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.
9. J. Bos and M. Coster. Addition Chain Heuristics. In *Advances in Cryptology– Proceedings of Crypto 89, Lecture Notes in Computer Science Vol. 658*, pages 400–407. Springer-Verlag, 1989.
10. E. Brickell, D. Gordon, K. McCurley, and D. Wilson. Fast Exponentiation with Precomputation. In R. Rueppel, editor, *Advances in Cryptology — Eurocrypt '92*, pages 200–207, Berlin, 1992. Springer-Verlag. Lecture Notes in Computer Science No. 658.
11. E. Brickell, P. Lee, and Y. Yacobi. Secure Audio Teleconference. In *Advances in Cryptology– Proceedings of Crypto 87, Lecture Notes in Computer Science Vol. 293, C. Pomerance editor*, pages 418–426. Springer-Verlag, 1987.
12. B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults. In *Proceeding 26th Annual Symposium on the Foundations of Computer Science*, pages 383–395. IEEE, 1985.
13. F. Ergun, S. Ravi Kumar, and R. Rubinfeld. Approximate Checking of Polynomials and Functional Equations. In *Proc. 37th Annual Symposium on Foundations of Computer Science*, pages 592–601. IEEE, 1996.
14. A. Fiat. Batch RSA. *Journal of Cryptology*, 10(2):75–88, 1997.
15. National Institute for Standards and Technology. Digital Signature Standard (DSS). Technical Report 169, August 30 1991.
16. P. Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Wigderson. Self-testing/correcting for polynomials and for approximate functions. In *Proc. Twenty Third Annual ACM Symposium on Theory of Computing*, pages 32–42. ACM, 1991.
17. C.H. Lim and P.J. Lee. More Flexible Exponentiation with Precomputation. In Y. Desmedt, editor, *Advances in Cryptology — Crypto '94*, pages 95–107, Berlin, 1994. Springer-Verlag. Lecture Notes in Computer Science No. 839.
18. D. Naccache, D. M'Rahi, S. Vaudenay, and D. Raphaeli. Can D.S.A be improved? Complexity trade-offs with the digital signature standard. In A. De Santis, editor, *Advances in Cryptology — Eurocrypt '94*, pages 77–85, Berlin, 1994. Springer-Verlag. Lecture Notes in Computer Science No. 950.
19. R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21:120–126, 1978.
20. R. Rubinfeld. Batch Checking with Applications to Linear Functions. *Information Processing Letters*, 42:77–80, 1992.
21. R. Rubinfeld. On the Robustness of Functional Equations. In *Proc. 35th Annual Symposium on Foundations of Computer Science*, pages 2–13. IEEE, 1994.
22. R. Rubinfeld. Designing Checkers for Programs that Run in Parallel. *Algorithmica*, 15(4):287–301, 1996.
23. R. Rubinfeld and M. Sudan. Robust Characterizations of Polynomials with Applications to Program Testing. *SIAM Journal on Computing*, 25(2):252–271, 1996.
24. J. Sauerbrey and A. Dietel. Resource requirements for the application of addition chains modulo exponentiation. In *Advances in Cryptology– Eurorypt '92, Lecture Notes in Computer Science Vol. 658.* Springer-Verlag, 1992.