An abridged version of this paper appears in The 4th International Conference on Pairing Based Cryptography (Pairing 2010), M. Joyce, A. Miyaji and A. Otsuka (Eds.), LNCS 6487, pp. 78 - 95, 2010 [23]. This is the full version.

Identification of Multiple Invalid Pairing-based Signatures in Constrained Batches

Brian J. Matt

Johns Hopkins University
Applied Physics Laboratory
Laurel, MD 21102, USA
brian.matt@jhuapl.edu

Abstract. This paper describes a new method in pairing-based signature schemes for identifying the invalid digital signatures in a batch after batch verification has failed. The method more efficiently identifies non-trivial numbers, w, of invalid signatures in constrained sized, N, batches than previously published methods, and does not require that the verifier possess detailed knowledge of w. Our method uses "divide-and-conquer" search to identify the invalid signatures within a batch, pruning the search tree to reduce the number of pairing computations required. The method prunes the search tree more rapidly than previously published techniques and thereby provides performance gains for batch sizes of interest.

We are motivated by wireless systems where the verifier seeks to conserve computations or a related resource, such as energy, by using large batches. However, the batch size is constrained by how long the verifier can delay batch verification while accumulating signatures to verify.

We compare the expected performance of our method (for a number of different signature schemes at varying security levels) for varying batch sizes and numbers of invalid signatures against earlier methods. We find that our new method provides the best performance for constrained batches, whenever the number of invalid signatures is less than half the batch size. We include recently published methods based on techniques from the group-testing literature in our analysis. Our new method consistently outperforms these group-testing based methods, and substantially reduces the cost ($> 50\%$) when $w \leq N/4$.

Keywords Pairing-based signatures, Batch verification, Invalid Signature Identification, Identity-based signatures, Short signatures, Wireless networks

1 Introduction

In many network security and E-commerce systems that use batch signature verification, the verifier does not have the freedom to accumulate arbitrarily large batches of messages and signatures to maximize the efficiency of the batch verifier. Typically the batch size is constrained by how long the verifier can delay verification of early arriving messages while waiting to accumulate additional messages for the batch. In such applications, whenever a batch fails verification, the verifier then chooses the best method available to identify the invalid signatures. The choice is determined by the size of the batch, and perhaps based on some belief about the likely number of, or some estimate of the bound on the number of, invalid signatures.

When the system is part of a data rate limited wireless network, the signature scheme of choice is often a communication efficient bilinear pairing-based scheme.Some examples of

such systems are: some secure wireless routing protocols [25, 41, 33, 26]; secure accounting and charging schemes or schemes to provide incentives to nodes for exhibiting desirable behavior [31, 5, 17, 28, 44]; authenticated localization messages and safety messages (vehicular networks) [32, 30, 43]; and authenticating bundled data messages in delay (or disruption) tolerant networks [10, 37]. This choice is justifiable if the need for communication efficiency justifies the higher processing costs of these schemes compared to 1) conventional signature schemes such as ECDSA [13], or 2) signature schemes using implicit-certificates [1, 29, 4]. Such pairing-based schemes include short signature schemes [3, 6] and bandwidth efficient identity-based signature schemes [7, 6, 40, 6].

When batch verification fails, a number of methods have been proposed, primarily for large batches, to identify the invalid signatures in batch verifiable, pairing-based signature schemes. These proposals include "divide-and-conquer" (DC) methods such as Fast DC verifier [27] and Binary Quick Search [18], and methods that significantly augment DC with other techniques (i.e., hybrid methods) [21], and some specialized techniques that are practical for batches with only a very few invalid signatures [18]. Recently, methods based on group testing have been proposed [42]. However, no methods have been proposed specifically for constrained batches.

Our contribution.

In this paper, we present a new method for finding invalid signatures in pairing-based schemes based on hybrid divide-and-conquer searching. The method outperforms earlier hybrid divide-and-conquer methods when N is constrained (16 - 128) and $w < N/2$. We compare our method with earlier work for a number of pairing-based schemes and present the results using cost parameters drawn from a realization of the Cha-Cheon [7] signature scheme at the 80 bit and 192 bit security levels. Our analysis can be easily applied to other schemes and at other security levels. Our new hybrid method seeks to identify more invalid signatures in each (sub-) batch than earlier hybrid methods before resorting to sub-dividing the (sub-)batches. The new method reduces the number of computations required whenever $w < N/2$.

Recently group testing algorithms [9] have been proposed for use in identifying invalid signatures in batches [42]. However, many group algorithms assume that w (or an upper bound) is known. If the estimate d of w must be precise in order to obtain good performance, then such methods will not be useful in practice. We compare the expected performance of our method against the best methods in [42] for use in single processor systems. We find that our new method, as well as some earlier methods, always significantly outperform the proposed group testing methods, even when w is precisely known. We also examine the impact of an inaccurate estimate on the expected performance of the method in [42] which has the best worse case performance in our setting when w is known precisely. We find that even when the estimate is good ($d_w = 2w$), the impact on performance is severe when $2 \leq w < N/2$.

2 Notation

In this paper we assume that pairing-based schemes use bilinear pairings on an elliptic curve E, defined over $\mathbb{F}_q$, where q is a large prime. $\mathbb{G}_1$ and $\mathbb{G}_2$ are distinct subgroups of prime order r on this curve, where $\mathbb{G}_1$ is a subset of the points on E with coordinates in $\mathbb{F}_q$, and $\mathbb{G}_2$ is a subset of the points on E with coordinates in $\mathbb{F}_{q^d}$, for a small integer d (the embedding degree).

The pairing e is a map from $\mathbb{G}_1 \times \mathbb{G}_2$ into $\mathbb{G}_T$ where $\mathbb{G}_T$ is a multiplicative group of order r in the field $\mathbb{F}_{q^d}$.

Once the initial batch verification is performed, the costs of the methods for finding the invalid signatures in a batch are dominated by the cost of a product of pairings computations, $\mathsf{CstMultPair}$, and the cost of multiplying two elements of $\mathbb{G}_T$, $\mathsf{CstMult}\mathbb{G}_\mathrm{T}$. A cost that can be significant in large batches for some methods is the cost of additions in $\mathbb{G}_1$, $\mathsf{CstAdd}\mathbb{G}_1$, (or additions in $\mathbb{G}_2$). The other operations used in the methods discussed in this paper, such as exponentiation $\mathsf{CstExpt}\mathbb{G}_\mathrm{T}(\mathrm{t}_1)$ in $\mathbb{F}_{q^d}$ (for small t_1), computing an inverse in $\mathbb{G}_T$ ($\mathsf{CstInv}\mathbb{G}_\mathrm{T}$), multiplying an element of $\mathbb{G}_1$ or $\mathbb{G}_2$ by a modest sized scalar c, i.e.,$c \leq N$, has minimal impact.

3 Background

Fiat [12] introduced batch cryptography, and the first batch verification signature scheme was that of Naccache *et al.* [24] for a variant of DSA signatures. Bellare *et al.* [2] presented three generic methods for batching modular exponentiations: *the random subset test*, the *small exponents test* (SET), and the *bucket test*, which are related to techniques in [24, 38].

A number of pairing-based signature schemes have batch verifiers which use the small exponents test, many of which have the form

$$e\left(\sum_{i=1}^{N} B_i, S\right) = \prod_{h=1}^{\bar{\mathsf{n}}-1} e\left(\sum_{i=1}^{N} D_{i,h}, T_h\right) \tag{1}$$

where S and T_h are system parameters. Examples when $\bar{\mathsf{n}} = 2$ include Boneh, Lynn and Shacham short signatures [3], when the batch consists of messages signed by a single signer or a common message signed by different signers, the Cha-Cheon identity-based scheme [7], and the scheme of Xun Yi [39] as interpreted by Solinas [35]. Examples of schemes that have this form with $\bar{\mathsf{n}} = 3$ include the Camenisch, Hohenberger, and Pedersen (CHP) short signature scheme (for a common time period [6]) and a recent proposal of Zhang et al. [43] for signing and batch verifying location and safety messages in vehicular networks.[1]

3.1 Identifying Invalid Signatures

Methods for identifying invalid signatures fall into three categories: divide-and-conquer methods [27, 18], exponent testing methods [19, 20, 36, 18], and hybrid techniques [20, 21] which combine aspects of divide-and-conquer and other methods.

Divide-and-Conquer Methods Pastuszak *et al.* [27] first investigated methods for identifying invalid signatures within a batch. They explored divide-and-conquer methods for generic batch verifiers, methods that work with any of the three batch verifiers studied by Ballare *et al.* In these methods the set of signatures in an invalid batch is repeatedly divided into $d \geq 2$ smaller sub-batches to verify. The most efficient of their techniques, the Fast DC Verifier

[1] Some of these schemes are defined for pairings where $\mathbb{G}_1 = \mathbb{G}_2$. In CHP short signatures one of the pairings has the form $e\left(T_h, \sum_{i=1}^{N} D_{i,h}\right)$. For simplicity of presentation we ignore such distinctions in the remainder of this paper.

Method, exploits knowledge of the results of the first $d-1$ sub-batch verifications to determine whether the verification of the d^{th} sub-batch is necessary. Performance measurements of one of the methods of [27] for the Boneh, Lynn and Shacham (BLS) [3] signature scheme have been reported [11].

In [18] a more efficient divide-and-conquer method, called Binary Quick Search (BQS), for small exponents test based verifiers was presented. In this method a batch verifier that compares two quantities, X and Y, is replaced with an equivalent test $A = XY^{-1}$, and the batch is accepted if $A = 1$. The BQS algorithm is always as least as efficient as any $d = 2^n$ary DC Verifier [22]. The upper bound of the number of batch verifications required by BQS is half that of the Fast DC Verifier for $d = 2$ [18].

Exponent Testing Methods The first exponent testing method, developed by Lee *et al.* [19], was capable of finding a single invalid signature within a batch of "DSA-type" signatures. Law and Matt presented two exponent testing methods for pairing-based batch signatures, the Exponentiation Method and the Exponentiation with Sectors (EwS) Method, in [18]. Both methods use exhaustive search during batch verification, resulting in exponential cost.

The Exponentiation Method replaces (1) with $\alpha_0 = \prod_{h=0}^{\bar{\mathsf{n}}-1} e\left(\sum_{i=1}^{N} D_{i,h}, T_h\right)$ where $D_{i,0} = B_i$ and $T_0 = -S$. If α_0 is equal to the identity, the batch is valid. Otherwise compute α_j, for $1 \leq j \leq w$,

$$\alpha_j = \prod_{h=0}^{\bar{\mathsf{n}}-1} e\left(\sum_{i=1}^{N} i^j D_{i,h}, T_h\right) \tag{2}$$

and perform a test on the values $\alpha_j, \alpha_{j-1}, \ldots \alpha_0$. For $j = 1$, test whether $\alpha_1 = \alpha_0^{z_1}$ has a solution for $1 \leq z_1 \leq N$ using Shanks' giant-step baby-step algorithm [34]. If successful, $w = 1$ and z_1 is the position of the invalid signature. In general the method tests whether

$$\alpha_j = \prod_{t=1}^{j} (\alpha_{j-t})^{(-1)^{t-1} p_t} \tag{3}$$

has a solution where p_t is the tth elementary symmetric polynomial in $1 \leq z_1 < \ldots < z_j \leq N$. The authors show that the tests can be performed in $O(\sqrt{N})$ for $j = 1$ and $O(N^{j-1}/(j-1)!)$ for $j \geq 2$ multiplications in $\mathbb{F}_{q^d}$. If a test fails increment j, compute α_j, and test. When $j = w$ the test will succeed, and the values of $z_1, \ldots, z_w$ are the positions of the invalid signatures.

The Exponentiation with Sectors Method uses two stages. In the first stage, the batch is divided into approximately $\sqrt{N}$ sectors of approximately equal size and the Exponentiation Method is used, where each $D_{i,h}$ within a sector is multiplied by the same constant to identify the v invalid sectors. In the second stage, the Exponentiation Method is used to find the invalid signatures within a batch consisting of the signatures from the v invalid sectors.[2]

Hybrid DC Methods Lee et al. [20] applied their approach for DSA-type signatures to identifying a single invalid signature in batches of RSA signatures. They addressed the problem

[2] The EwS method is always outperformed by one or more of the other methods we discuss in this section in our setting; therefore we do not discuss the performance of this method in Section 6.

of identifying multiple invalid RSA signatures by using their RSA method in a DC method. Each (sub-)batch is tested using their RSA method. If the (sub-)batch has multiple invalid signatures, it is divided and its child sub-batches are tested. If a (sub-)batch has a single invalid signature, that signature is identified; if a (sub-)batch has no invalid signatures, that (sub-)batch is not tested further. Otherwise the (sub-)batch is divided and its child sub-batches are tested. However, Stanek showed in [36] that their approach for RSA signatures is not secure.

In [22] Matt presented two hybrid DC methods. The first, called Single Pruning Search (SPS), uses (2) and (3) for $0 \leq j \leq 1$ to identify single invalid signatures in (sub-)batches until the root of every maximal sub-tree of the search tree with a single invalid signature is identified. This method is somewhat similar to the Lee et al. method for RSA signature batches with multiple invalid signatures.

The second method, Paired Single Pruning Search (PSPS), extends SPS with an additional test. When a (sub-)batch B has two or more invalid signatures, $\alpha_{0,L}$ is computed for the left child sub-batch of B, and if both child sub-batches have invalid signatures, then $\alpha_{0,R} = \alpha_0 \cdot \alpha_{0,L}^{-1}$ is calculated for the right child sub-batch. Then $\alpha_1 = \alpha_{0,L}^{z_L} \cdot \alpha_{0,R}^{z_R}$ is tested for a solution where the exponents are restricted to the set of i's used in the child sub-batches. A solution will exist whenever both child sub-batches have a single invalid signature. The additional test determines if the two child nodes are both roots of maximal sub-trees of the search tree with a single invalid signature, without computing $\alpha_{1,L}$ and $\alpha_{1,R}$.

4 A Hybrid DC Method Exploiting $w = 2$ Maximal Sub-Trees

Hybrid divide-and-conquer methods operate on (for simplicity) a binary tree T with $w \geq 1$ invalid signatures whose root node is the batch, and each pair of child nodes represents the two nearly equal size sub-batches of their parent. The SPS and PSPS methods search down though T until the roots of the w maximal sub-trees ST_i, $i = 1, .., w$, of T, which represent sub-batches that have a single invalid signature are reached and tested. The Triple Pruning Search method we describe in this paper searches down through T until the roots of the maximal sub-trees $ST2_i$, $i = 1, .., v$, of T, which represent the sub-batches that have exactly two invalid signatures, and the maximal sub-trees $ST1_j$, $j = 1, .., w-2v$, which represent sub-batches that have a single invalid signature and which are not a sub-tree of any of the $ST2_i$ sub-trees, are reached and tested.

Let B be the batch. $|X|$ is the size the (sub-)batch X of B, $lowbnd(X)$ is the index in B of the lowest position signature in X, $w(X)$ is the number of invalid signatures in X, and $invalid(X)$ is the set of invalid signatures in X. If T is a binary tree and X is sub-batch, then $\hat{X}$ is the sibling of X.

4.1 Triple Pruning Search (TPS) Method.

The recursive algorithm below describes the Triple Pruning Search (TPS) method on a batch B, which is a list of $N = 2^h$, $h \geq 2$, randomly ordered message / signature pairs $((m_1, s_1), \ldots, (m_N, s_N))$, where the signature components are verified elements of the appropriate groups. On the initial call to $TPS(X)$, $X = B$.

$TPS(X)$ includes the initial batch verification (lines 2 through 4). When $X = B$, $Get_0(B)$ computes $\alpha_{0,B}$ following the SET algorithm, and then computes $\alpha_{0,B}^{-1}$. The test $\alpha_{0,B} = 1$ determines whether $w(B) = 0$.

Algorithm 4.1 $TPS\ (X)$ *(Triple Pruning Search)*

Input: X – a list of message / signature pairs.
Output: A list of the invalid pairs in the batch.

```
 1: if X = B then
 2:         α_{0,[B]} ← Get_0(B)
 3:         if α_{0,[B]} = 1 then
 4:                 return
 5:         α_{1,[B]} ← Get_1(B)
 6:         z ← Shanks(B)
 7:         if z ≠ 0 then
 8:                 print (m_z, s_z)
 9:                 return
10:         α_{2,[B]} ← Get_2(B)
11:         (z_1, z_2) ← FastFactor(B)
12:         if z_1 ≠ 0 then
13:                 print (m_{z_1}, s_{z_1}), (m_{z_2}, s_{z_2})
14:                 return
15: (SearchLeft, SearchRight) ← TPSQuadSolver(X, Left(X), Right(X)) .
16: if SearchLeft = true then
17:         TPS (Left(X))
18: if SearchRight = true then
19:         TPS (Right(X))
20: if X = B then
21:         PrintList() // Prints the sorted list of invalid message / signature pairs
22: return
```

Lines 5 through 9 determine whether $w(B) = 1$, and if so they locate the invalid signature. $Get_1(B)$ computes $\alpha_{1,B}$ in about $\bar{n} \cdot N \cdot \mathsf{CstAdd}\mathbb{G}_1 + \mathsf{CstMultPair}$ operations using the partial results from the computation of $\alpha_{0,B}$, and then computes $\alpha_{1,B}^{-1}$. $Shanks(B)$ is used to locate a single invalid signature. $Shanks(X)$ tests whether $\alpha_{1,X} \cdot (\alpha_{0,X}^{-1})^d = (\alpha_{0,X}^s)^c$ has a solution with $l \le d \le s + l$ and $0 \le c \le t$, where $s \approx \sqrt{|X|}, t \approx |X|/s$ and $l = lowbnd(X)$. If $w(X) = 1$, $Shanks(X)$ returns $d + c * s$, the position of the invalid signature. If $w(X) > 1$, then it returns 0. $Shanks(X)$ uses the giant-step baby-step algorithm [34].

Lines 10 through 14 determine whether $w(B) = 2$, and if so they locate the two invalid signatures. $Get_2(B)$ computes $\alpha_{2,B}$ in about $\bar{n} \cdot N \cdot \mathsf{CstAdd}\mathbb{G}_1 + \mathsf{CstMultPair}$ operations using the partial results from the computation of $\alpha_{1,B}$ and $\alpha_{0,B}$, and then computes $\alpha_{2,B}^{-1}$. $FastFactor(B)$ is used to locate the pair of invalid signatures. $FastFactor(X)$ tests whether $\alpha_{2,X}^4 \cdot (\alpha_{1,X}^{-4})^n \cdot \alpha_{0,X}^{n^2} = \alpha_{0,X}^{m^2}$ has a solution with $2l + 1 \le n \le 2(l + |X|) - 1$ and $1 \le m \le |X| - 1$, where $l = lowbnd(X)$; if so, then $z_2 = (n + m)/2$ and $z_1 = (n - m)/2$ with $z_2 > z_1$ are the positions of the two invalid signatures. If $w(X) = 2$, $FastFactor(X)$ returns $(z_1,\ z_2)$. If $w(X) > 2$, $FastFactor(X)$ returns $(0, 0)$. See Appendix A.3.

In line 15, the function $TPSQuadSolver(X,\ Left(X),\ Right(X))$ determines whether X has two or fewer invalid signatures in its left sub-batch $Left(X)$ and two or fewer invalid signatures in its right sub-batch $Right(X)$. $TPSQuadSolver$ places the locations of the invalid signature it identifies in a list which $PrintList()$ outputs.

4.2 $TPSQuadSolver(Parent, Left, Right)$

The algorithm on page 8 describes the $TPSQuadSolver(Parent,\ Left,\ Right)$ function on a (sub-)batch $Parent$ with $|Parent| = 2^h$, $h \geq 2$, and $w(Parent) \geq 3$. $Left$ and $Right$ represent the two equal size sub-batches of $Parent$.

$TPSQuadSolver(Parent,\ Left,\ Right)$ uses $Get_0(Left)$ to compute $\alpha_{0,Left}$ (and $\alpha_{0,Left}^{-1}$), which requires a $\mathsf{CstMultPair}$ computation as well as some comparatively minor cost computations in $\mathbb{G}_1$. Lines 3 through 8 determine whether all of the invalid signatures in $Parent$ are in either $Left$, $Right$, or are divided between the two. If both $Left$ and $Right$ have at least one invalid signature then $Get_0(Right)$ is used to compute $\alpha_{0,Right}$ ($\alpha_{0,Right}^{-1}$) with negligible cost.[3]

If $TScap \geq |Parent| \geq 4$, $TriFactor(Parent)$ is used (line 11) to determine whether case 1) $w(Left) = 2$ and $w(Right) = 1$, or case 2) $w(Left) = 1$ and $w(Right) = 2$; otherwise it fails. $TriFactor(Parent)$, see Appendix A.4, uses the function $TriSolver$ to test case 1 and if that fails, case 2.

Case 1) $TriSolver(P = Parent,\ L = Left,\ R = Right)$
If $\alpha_{2,P}^4 \cdot (\alpha_{0,R}^{-4})^{z_3^2} \cdot (\alpha_{1,P}^{-4})^{n_L} \cdot (\alpha_{0,R}^4)^{n_L \cdot z_3} \cdot \alpha_{0,L}^{{n_L}^2} = \alpha_{0,L}^{{m_L}^2}$ has a solution with $2l_L + 1 \leq n_L \leq 2(l_L + |L|) - 1$, $1 \leq m_L \leq |L| - 1$ and $l_R \leq z_3 < l_R + |R|$, where $l_L = lowbnd(L)$ and $l_R = lowbnd(R)$, then $z_2 = (n_L + m_L)/2$ and $z_1 = (n_L - m_L)/2$ where $z_2 > z_1$ are the positions of the two invalid signatures in L, and z_3 in R.

Case 2) $TriSolver(P = Parent,\ R = Right,\ L = Left)$
If $\alpha_{2,P}^4 \cdot (\alpha_{0,L}^{-4})^{z_1^2} \cdot (\alpha_{1,P}^{-4})^{n_R} \cdot (\alpha_{0,L}^4)^{n_R \cdot z_1} \cdot \alpha_{0,R}^{{n_R}^2} = \alpha_{0,R}^{{m_R}^2}$ has a solution with $2l_R + 1 \leq n_R \leq 2(l_R + |R|) - 1$, $1 \leq m_R \leq |R| - 1$ and $l_L \leq z_1 < l_L + |L|$, then $z_3 = (n_R + m_R)/2$ and $z_2 = (n_R - m_R)/2$ where $z_3 > z_2$ are the positions of the invalid signatures in R, and z_1 in L.

If $w(Parent) = 3$, $TriFactor(Parent)$ returns the positions of the three invalid signatures, which are added to the list of invalid signatures (line 13).

If $|Parent| = 4$ and $TriFactor(Parent)$ fails, then the positions of the four signatures in $Parent$ are added to the list of invalid signatures (line 16).

If $|Parent| > 4$ and $TriFactor(Parent)$ failed (or was not used), then Get_1 is used to compute $\alpha_{1,Left}$ (and $\alpha_{1,Left}^{-1}$) as well as $\alpha_{1,Right}$ (and $\alpha_{1,Right}^{-1}$) with approximate total cost $\mathsf{CstMultPair}$ (line 19).

If the following $Shanks(Left)$ test succeeds, then $Get_2(Right)$ can compute $\alpha_{2,Right} = \alpha_{2,Parent} \cdot \alpha_{1,Left}^{z_1}$ and its inverse efficiently with cost $2\,\mathsf{CstMult}\mathbb{G}_\mathrm{T} + 2\,\mathsf{CstInv}\mathbb{G}_\mathrm{T} + \mathsf{CstExpt}\mathbb{G}_\mathrm{T}(\mathrm{t}_1)$, where $t_1 < \lceil log_2(N) \rceil$. This cost is much less than $\mathsf{CstMultPair}$, we ignore this cost in Section 5. Next if $TriFactor(Parent)$ was not used, then $FastFactor(Right)$ is used (line 24) to test whether $w(Right) = 2$ and if so, identify the two invalid signatures in $Right$. If $TriFactor(Parent)$ was used, it must have failed, and so would $FastFactor(Right)$.

If the $Shanks(Left)$ test (line 20) fails, then $Shanks(Right)$ (line 30) is used to test the right sub-batch. If that test succeeds, then by exchanging $Left$ and $Right$, the preceding paragraph describes the function of lines 31 through 38.

If $w(Left) \geq 2$ and $w(Right) \geq 2$, then Get_2 is used to compute $\alpha_{2,Left}$ and $\alpha_{2,Right}$ and their inverses, with approximate total cost $\mathsf{CstMultPair}$ (line 40), followed by tests of $Left$ and $Right$ using $FastFactor$.

[3] $\alpha_{i,Right}$ where $i = 0, 1, 2$ can be computed inexpensively if $\alpha_{i,Left}$ is known by $\alpha_{i,Right} = \alpha_{i,Parent} \cdot \alpha_{i,Left}^{-1}$.

Algorithm 4.2 $TPSQuadSolver(Parent, Left, Right)$

Input: $Parent, Left, Right$ – the lists of message / signature pairs.
Return: ($SearchLeft$, $SearchRight$) – control behavior of TPS

1: $(z_1, z_2, z_3, z_4) \leftarrow (0, 0, 0, 0)$
2: $\alpha_{0,[Left]} \leftarrow Get_0(Left)$
3: **if** $\alpha_{0,[Left]} = 1$ **then**
4: copyAlphasAndInverses($Right$, $Parent$);
5: **return** (**false**, **true**)
6: **if** $\alpha_{0,[Left]} = \alpha_{0,[Parent]}$ **then**
7: copyAlphasAndInverses($Left$, $Parent$)
8: **return** (**true**, **false**)
9: $\alpha_{0,[Right]} \leftarrow Get_0(Right)$
10: **if** $TScap \geq |Parent| \geq 4$ **then** // $TScap = 8$, see Section 6
11: $(z_1, z_2, z_3) \leftarrow TriFactor(Parent)$
12: **if** $z_1 \neq 0$ **then**
13: AddToList(z_1, z_2, z_3)
14: **return** (**false**, **false**)
15: **if** $|Parent| = 4$ **then**
16: $i \leftarrow lowbnd(Parent)$; AddToList$(i, i+1, i+2, i+3)$
17: **return** (**false**, **false**)
18: $SearchLeft \leftarrow$ **false**; $SearchRight \leftarrow$ **false**
19: $\alpha_{1,[Left]} \leftarrow Get_1(Left)$; $\alpha_{1,[Right]} \leftarrow Get_1(Right)$;
20: $z_1 \leftarrow Shanks(Left)$
21: **if** $z_1 \neq 0$ **then** // $w(Left) = 1$
22: $\alpha_{2,[Right]} \leftarrow Get_2(Right)$
23: **if** $|Parent| > TScap$ **then**
24: $(z_3, z_4) \leftarrow FastFactor(Right)$
25: AddToList(z_1, z_3, z_4) // zeros are not added to the list
26: **if** $z_3 = 0$ **then**
27: $SearchRight \leftarrow$ **true**
28: **return** (**false**, $SearchRight$)
29: **else** // $w(Left) \geq 2$
30: $z_3 \leftarrow Shanks(Right)$
31: **if** $z_3 \neq 0$ **then** // $w(Right) = 1$
32: $\alpha_{2,[Left]} \leftarrow Get_2(Left)$
33: **if** $|Parent| > TScap$ **then**
34: $(z_1, z_2) \leftarrow FastFactor(Left)$
35: AddToList(z_1, z_2, z_3) // zeros are not added to the list
36: **if** $z_1 = 0$ **then**
37: $SearchLeft \leftarrow$ **true**
38: **return** ($SearchLeft$, **false**)
39: **else** // $w(Left) \geq 2$ and $w(Right) \geq 2$
40: $\alpha_{2,[Left]} \leftarrow Get_2(Left)$; $\alpha_{2,[Right]} \leftarrow Get_2(Right)$
41: $(z_1, z_2) \leftarrow FastFactor(Left)$; $(z_3, z_4) \leftarrow FastFactor(Right)$
42: AddToList(z_1, z_2, z_3, z_4) // zeros are not added to the list
43: **if** $z_1 = 0$ **then**
44: $SearchLeft \leftarrow$ **true**
45: **if** $z_3 = 0$ **then**
46: $SearchRight \leftarrow$ **true**
47: **return** ($SearchLeft, SearchRight$)

5 Expected Cost of the New Method

TPS requires that initial batch verification is performed using the Small Exponents Test. For simplicity, we assume that the batch verifier is of the form $\alpha_{0,B} = \prod_{h=0}^{\bar{\mathsf{n}}-1} e\left(\sum_{i=1}^{N} D_{i,h}, T_h\right)$. The cost of this process for Cha-Cheon signatures ($\bar{\mathsf{n}} = 2$) includes first checking that the signature components are in $\mathbb{G}_1$, (and that the authenticated system parameters T_h are in $\mathbb{G}_2$) then computing the terms $\sum_{i=1}^{N} D_{i,h}, h = 0, 1$ in $\mathbb{G}_1$, and finally computing $\alpha_{0,B}$ and testing whether $\alpha_{0,B} = 1$. If $\alpha_{0,B} \neq 1$ compute $\alpha_{0,B}^{-1}$.

If $\alpha_{0,B} \neq 1$ (and all $D_{i,h} \in \mathbb{G}_1$), then assuming that the intermediate values $D_{i,h}$ are retained, the cost of computing $\alpha_{1,B}$ and the cost of computing $\alpha_{2,B}$ are each $\bar{n} \cdot |B| \cdot \mathsf{CstAdd}\mathbb{G}_1 + \mathsf{CstInv}\mathbb{G}_\mathrm{T} + \mathsf{CstMultPair}$. Since $\mathsf{CstInv}\mathbb{G}_\mathrm{T} \ll \mathsf{CstMultPair}$, we ignore the cost of computing the inverses.

If $w = 1$, the cost of TPS, not including initial verification, is $\bar{n} \cdot |B| \cdot \mathsf{CstAdd}\mathbb{G}_1 + \mathsf{CstMultPair}$ plus the average cost of a successful $Shanks(B)$, which is $\frac{4}{3}\sqrt{|B|}\,\mathsf{CstMult}\mathbb{G}_\mathrm{T}$.

If $w = 2$, the cost is $2(\bar{n} \cdot |B| \cdot \mathsf{CstAdd}\mathbb{G}_1 + \mathsf{CstMultPair}) + 2\sqrt{|B|}\,\mathsf{CstMult}\mathbb{G}_\mathrm{T} + \frac{11}{4}|B|\,\mathsf{CstMult}\mathbb{G}_\mathrm{T})$, which is the cost of computing the two products of pairings, including their inputs, a failed $Shanks(B)$, and the average cost of a successful $FastFactor(B)$.

If $w > 2$, the cost includes the term $2(\bar{n} \cdot |B| \cdot \mathsf{CstAdd}\mathbb{G}_1 + \mathsf{CstMultPair}) + 2\sqrt{|B|}\,\mathsf{CstMult}\mathbb{G}_\mathrm{T} + \frac{9}{2}|B|\,\mathsf{CstMult}\mathbb{G}_\mathrm{T}$, which is two products of pairings computations, a failed $Shanks(B)$, and a failed $FastFactor(B)$. In addition the cost includes the cost generated by the recurrence relation $\mathbf{R}_{(\mathbf{TPS})}(w, M)$ below, with $|B| = 2^h$, where $h \geq 2$, and on initial call $M = |B|$ and $w(B) \geq 3$.

$\mathbf{R}_{(\mathbf{TPS})}(w, M) =$

$$\begin{cases} 0, & \begin{array}{l} w = 0, 1, 2, \\ w > M; \end{array} \\ \dfrac{\left[\begin{array}{c} \sum_{i=0}^{w} \binom{M/2}{w-i}\binom{M/2}{i}(\mathbf{R}_{(\mathbf{TPS})}(w-i, M/2) + \mathbf{R}_{(\mathbf{TPS})}(i, M/2) \\ + \mathrm{C}_{(\mathrm{TPS})}(w-i, i, M/2)) \end{array}\right]}{\binom{M}{w}}, & w \geq 3, \end{cases}$$

where the cost functions $\mathrm{C}_{(\mathrm{TPS})}(\cdot, \cdot, \cdot)$ are given in the following table. Note that $\mathsf{CstMult}\mathbb{G}_\mathrm{T} \ll \mathsf{CstMultPair}$, so we ignore small numbers of $\mathsf{CstMult}\mathbb{G}_\mathrm{T}$. For both $TScap \geq M > 4$ and for $M > TScap$, $\mathrm{C}_{(\mathrm{TPS})}(w, 0, M/2) = \mathrm{C}_{(\mathrm{TPS})}(0, w, M/2) = \mathsf{CstMultPair}$, $\mathrm{C}_{(\mathrm{TPS})}((w-2) > 2, 2, M/2) = \mathrm{C}_{(\mathrm{TPS})}(2, (w-2) > 2, M/2)$ and $\mathrm{C}_{(\mathrm{TPS})}((w-i) > 2, i > 2, M/2) = \mathrm{C}_{(\mathrm{TPS})}(i > 2, (w-i) > 2, M/2)$.

Argument		*Costs*	
		CstMultPair	CstMult$\mathbb{G}_{\mathrm{T}}$
$M = 4$	$\mathrm{C}_{(\mathrm{TPS})}(1, 2, M/2)$	1	$\frac{40+14\sqrt{2}}{32}M + \frac{9+3\sqrt{2}}{32}M^2$
	$\mathrm{C}_{(\mathrm{TPS})}(2, 1, M/2)$	1	$\frac{128+14\sqrt{2}}{32}M + \frac{33+3\sqrt{2}}{32}M^2$
	$\mathrm{C}_{(\mathrm{TPS})}(2, 2, M/2)$	1	$\frac{11}{2}M + \frac{3}{2}M^2$
$TScap \geq M > 4$	$\mathrm{C}_{(\mathrm{TPS})}(1, 2, M/2)$	1	$\frac{40+14\sqrt{2}}{32}M + \frac{9+3\sqrt{2}}{32}M^2$
	$\mathrm{C}_{(\mathrm{TPS})}(2, 1, M/2)$	1	$\frac{128+14\sqrt{2}}{32}M + \frac{33+3\sqrt{2}}{32}M^2$
	$\mathrm{C}_{(\mathrm{TPS})}(1, (w-1) > 2, M/2)$	2	$\frac{4}{3}\sqrt{M} + \frac{11}{2}M + \frac{3}{2}M^2$
	$\mathrm{C}_{(\mathrm{TPS})}((w-1) > 2, 1, M/2)$	2	$\frac{10}{3}\sqrt{M} + \frac{11}{2}M + \frac{3}{2}M^2$
	$\mathrm{C}_{(\mathrm{TPS})}(2, 2, M/2)$	3	$4\sqrt{M} + 11M + \frac{3}{2}M^2$
	$\mathrm{C}_{(\mathrm{TPS})}((w-2) > 2, 2, M/2)$	3	$4\sqrt{M} + 12\frac{3}{4}M + \frac{3}{2}M^2$
	$\mathrm{C}_{(\mathrm{TPS})}((w-i) > 2, i > 2, M/2)$	3	$4\sqrt{M} + 14\frac{1}{2}M + \frac{3}{2}M^2$
$M > TScap$	$\mathrm{C}_{(\mathrm{TPS})}(1, 2, M/2)$	2	$\frac{4}{3}\sqrt{M} + \frac{11}{4}M$
	$\mathrm{C}_{(\mathrm{TPS})}(2, 1, M/2)$	2	$\frac{10}{3}\sqrt{M} + \frac{11}{4}M$
	$\mathrm{C}_{(\mathrm{TPS})}(1, (w-1) > 2, M/2)$	2	$\frac{4}{3}\sqrt{M} + \frac{9}{2}M$
	$\mathrm{C}_{(\mathrm{TPS})}((w-1) > 2, 1, M/2)$	2	$\frac{10}{3}\sqrt{M} + \frac{9}{2}M$
	$\mathrm{C}_{(\mathrm{TPS})}(2, 2, M/2)$	3	$4\sqrt{M} + \frac{11}{2}M$
	$\mathrm{C}_{(\mathrm{TPS})}((w-2) > 2, 2, M/2)$	3	$4\sqrt{M} + 7\frac{1}{4}M$
	$\mathrm{C}_{(\mathrm{TPS})}((w-i) > 2, i > 2, M/2)$	3	$4\sqrt{M} + 9M$

6 Performance

All of the methods discussed in this section perform initial batch verification in a similar manner. For Cha-Cheon signatures, they all check that the signature components are in $\mathbb{G}_1$, then compute α_0 for the batch, and then test whether $\alpha_0 = 1$. There are some slight variations in how the terms $\sum_{i=1}^{N} D_{i,h}$ are summed, but the cost in each case is the same. In Sections 6.1 we compare the expected performance of TPS against the methods discussed in Section 3.1 and in Sections 7.1 against the group testing based methods, once the initial batch verification has failed. See Section 5 and Appendix A for the derivations of the costs presented below and additional discussion of the performance of the methods.

We use Cases A and E of [14] for Cha-Cheon signatures to give an indication of how our results change with variations in the relative cost of operations.[4] In Case A, the group order r is a 160-bit value, the elliptic curve E is defined over $\mathbb{F}_q$, where q is a 160-bit value, and the embedding degree $d = 6$. In Case E, the group order r is a 384-bit value, q is a 384-bit value, and the embedding degree $d = 12$. All costs are given in terms of the number of multiplications (m) in $\mathbb{F}_q$, assuming that squaring has the same cost as multiplication, using the following estimates from Granger, Page and Smart [14], Granger and Smart [15], and Devegili et al. [8].

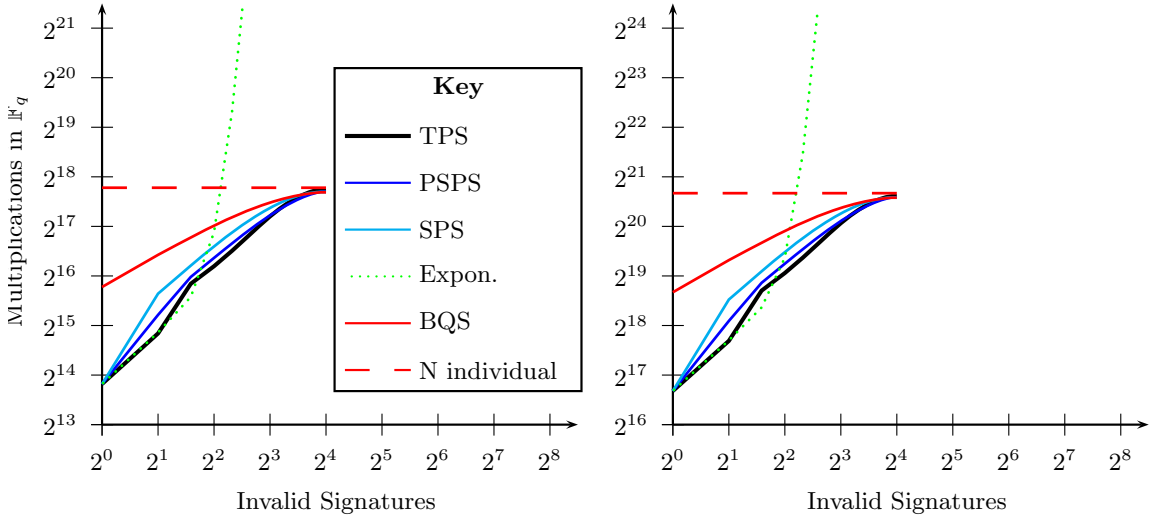
[4] The most important factor in the relative performance of all the methods is the ratio of CstMultPair to CstMult$\mathbb{G}_{\mathrm{T}}$. The ratio of CstMultPair to $\bar{n} \cdot |B| \cdot$ CstAdd$\mathbb{G}_1$ is much less significant in our setting.

– For Case A, 1 double product of pairings $= 14,027m$, 1 multiplication in $F_{q^6} = 15m$, 1 addition in $\mathbb{G}_1 = 11m$.

– For Case E, 1 double product of pairings $= 104,316m$, 1 multiplication in $F_{q^{12}} = 45m$, 1 addition in $\mathbb{G}_1 = 11m$.

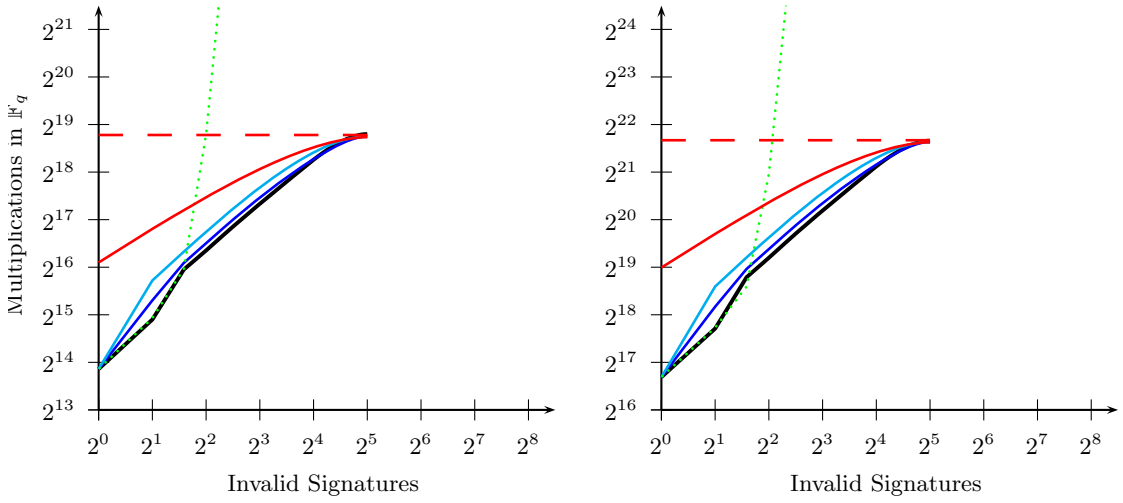
6.1 Peformance TPS vs Earlier Methods

Figures 1 through 4 show the expected cost of TPS, PSPS, SPS and BQS, the Exponential method, as well as testing the signatures individually, in units of multiplications in $\mathbb{F}_q$.



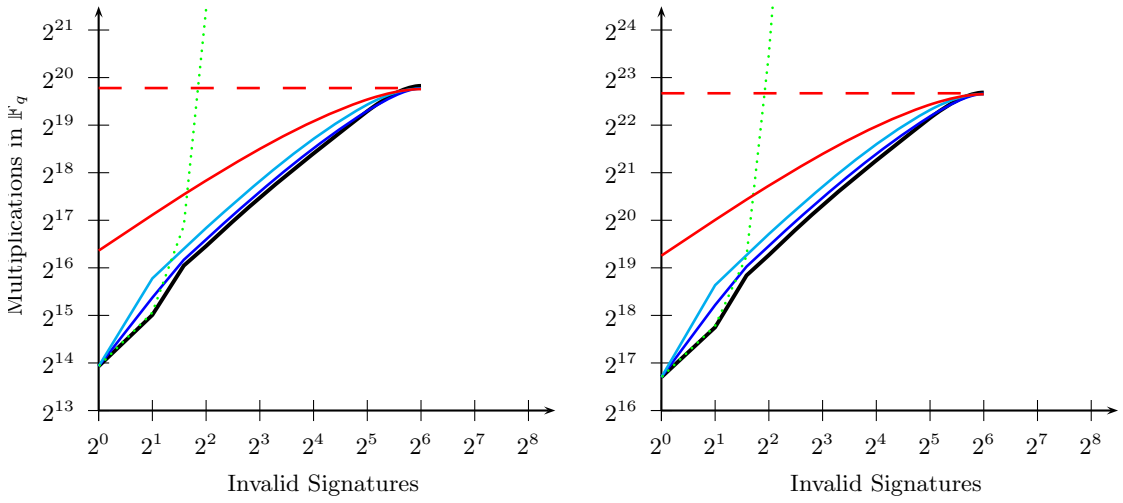
(a) Case A (b) Case E

Fig. 1: Number of multiplies in $\mathbb{F}_q$, $N = 16$.



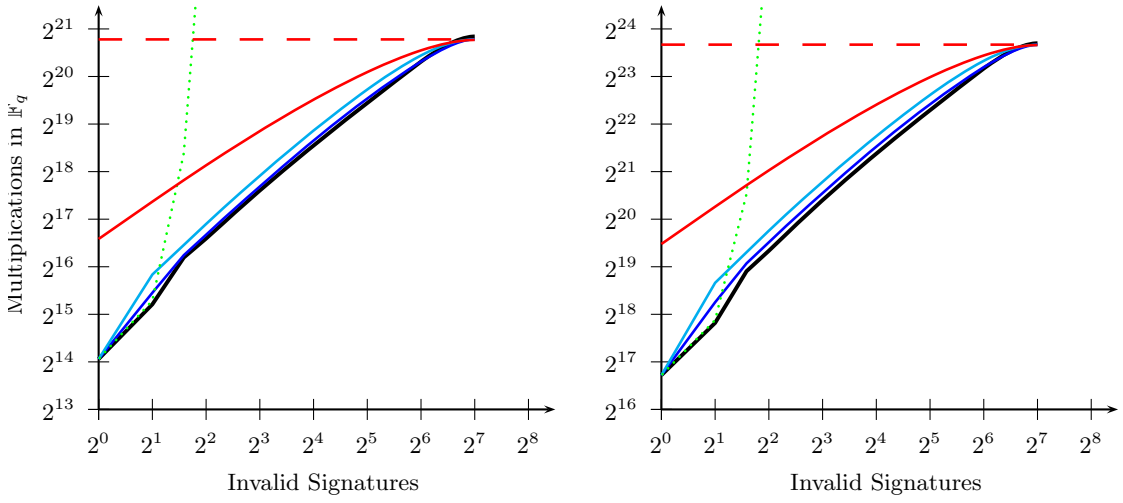
(a) Case A (b) Case E

Fig. 2: Number of multiplies in $\mathbb{F}_q$, $N = 32$.



(a) Case A (b) Case E

Fig. 3: Number of multiplies in $\mathbb{F}_q$, $N = 64$.

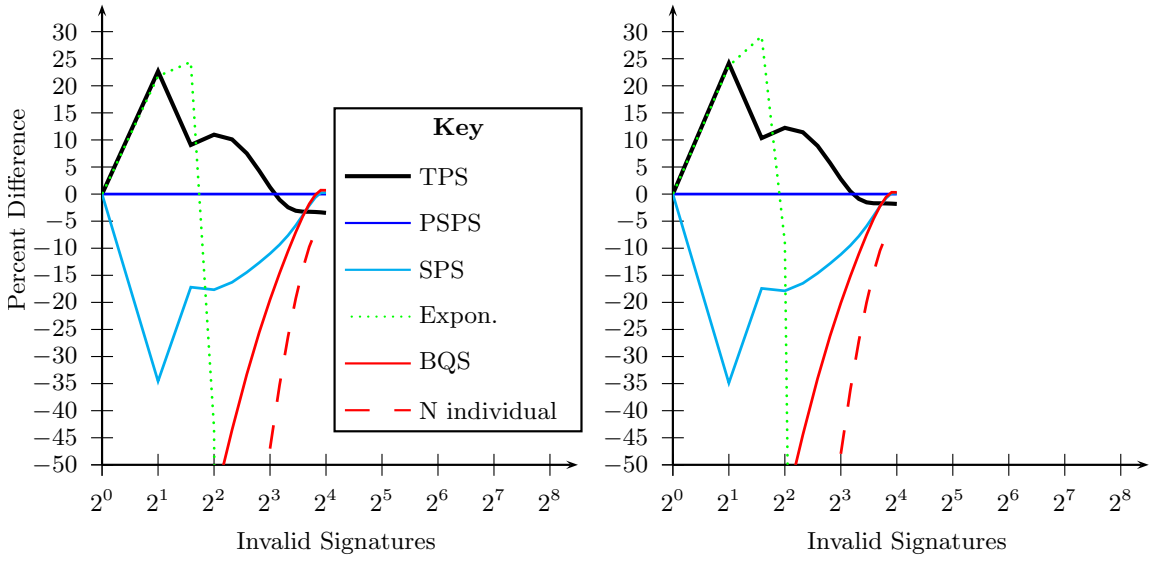


(a) Case A (b) Case E

Fig. 4: Number of multiplies in $\mathbb{F}_q$, $N = 128$.

6.2 Percent Difference Comparison of TPS with Earlier Methods

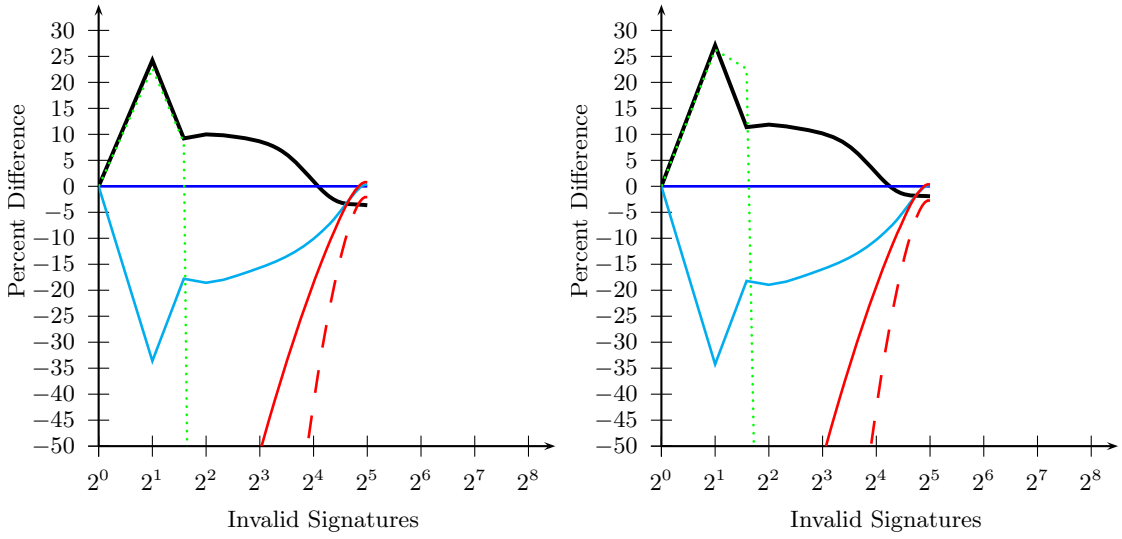
Figures 5 through 8 compare methods relative to PSPS, previously the best performing method for our setting. We also include two additional divide-and-conquer methods, SPS and BQS, the Exponential method, as well as testing the signatures individually.



(a) Case A

(b) Case E

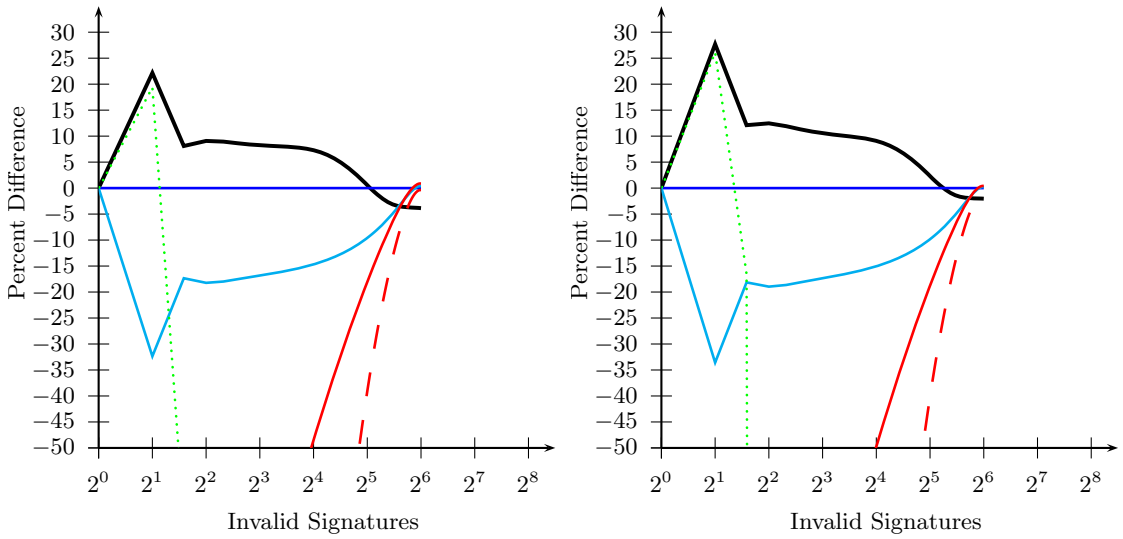
Fig. 5: Percent Difference Comparison with PSPS, $N = 16$.



(a) Case A

(b) Case E

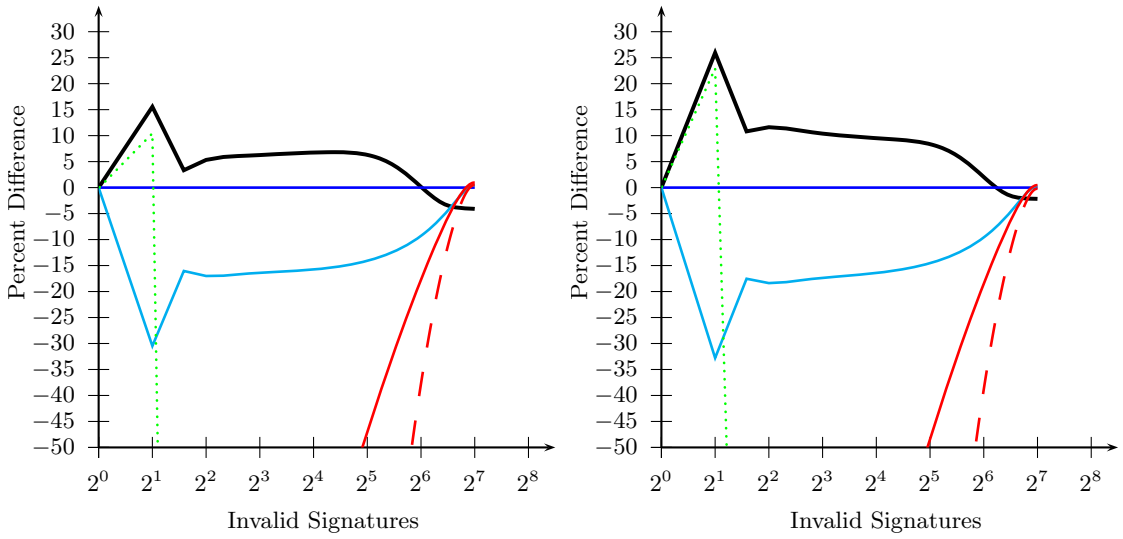
Fig. 6: Percent Difference Comparison with PSPS, $N = 32$.



(a) Case A

(b) Case E

Fig. 7: Percent Difference Comparison with PSPS, $N = 64$.



(a) Case A

(b) Case E

Fig. 8: Percent Difference Comparison with PSPS, $N = 128$.

TPS uses successful $TriSolver$ tests to avoid computing CstMultPair for α_1's (and perhaps) α_2's for child sub-batches. We observe that the $O(N^2)$ cost of these tests requires that we restrict the use of $TriFactor$ to parent batches of size less than or equal to $TScap$ (line 10 of Algorithm 4.2); otherwise for larger batches the cost of $TriFactor$, even when successful, would become greater than the cost of the α's and their associated $Shanks$ and $FastFactor$ tests. In Figures 1 through 8 above, and Figures 9 through 16 in Section 7, $TScap = 8$ which limits the cost of $TriFactor$ to no more than ≈ 140 CstMult$\mathbb{G}_{\mathrm{T}}$. Since the ratio of CstMultPair to CstMult$\mathbb{G}_{\mathrm{T}}$ for Case A is 1275 : 1 and Case E is 2318 : 1, the cost of $TriFactor$ does not significantly impact the overall cost of TPS in these figures. However, setting $TScap = 8$ rather than $TScap = |B|$ increases the number of product of pairing computations used by TPS.

7 Group Testing Based Methods

Zaverucha and Stinson [42] recently examined algorithms from the group testing literature for use in identifying invalid signatures in batches. Like Pastuszak et al., they work with generic batch verifiers. Zaverucha and Stinson state that for single processor systems, identifying invalid signatures using Binary Splitting (same method as the Fast DC-verifier of Pastuszak *et al.* [27]) and Hwang's Generalized Binary Splitting (HGBS) methods have the lowest bounds on the worse case number of verifications. Here we examine the expected performance of these algorithms.

Binary Splitting tests an invalid (sub-)batch of size M by first testing $\lfloor \frac{M}{2} \rfloor$ signatures from the (sub-)batch. If this test indicates an invalid signature, a test of the remaining $\lceil \frac{M}{2} \rceil$ signatures is required, and the method is applied to both sub-batchs; otherwise the test is not performed on the sub-batch with the $\lceil \frac{M}{2} \rceil$ signatures and the method is applied only to that sub-batch. Binary splitting is the same method as the Fast DC-verifier of Pastuszak *et al.* [27].

The HGBS method [16] requires an estimate d_w of the number of invalid signatures in a batch of size M. The descriptions of the HGBS method which have appeared in the literature differ slightly. Here we describe the version of HGBS which appears in [16].

G1:
If $M \leq 2(d_w - 1)$ verify the M signatures individually, otherwise compute $a \in \mathbb{N}$ s.t. $2^{a+1} > (M - d_w + 1)/d \geq 2^a$ and goto **G2**.

G2:
Test a sub-batch X of size 2^a. If all are valid $M \leftarrow M - 2^a$ and go to **G1** for the other signatures. If X is invalid find one invalid signature in X using a tests via binary search and dispose of the invalid signature and all valid sub-batches identified during the search. Create a new batch consisting of all the remaining sub-batches. Set M to the size of this batch and $d_w \leftarrow d_w - 1$ and go to **G1**.

Note that d_w must be greater than or equal to w, otherwise the method is undefined.[5]

7.1 Comparison of TPS with Group Testing based Methods

In Figures 9 through 16 we compare the performance of TPS against two group testing methods, Binary Splitting and HGBS, relative to BQS. We use BQS since it only requires that a batch

[5] In [16] d is an upper bound, in [9] it is the known number of invalid signatures.

verifier that compares two quantities, X and Y, can be replaced with the test $A = XY^{-1}$. BQS is intermediate between the more signature scheme specific TPS method and the general group testing methods. We also show the extent to which uncertainty in the estimated (or a bound of the) number of the invalid signatures in a batch degrades performance of HGBS. Binary Splitting does not use such an estimate.

8 Conclusion

We presented the TPS method for identifying invalid signatures in pairing-based batch signature schemes using SET, and have analyzed its expected performance. The new method provides improved performance for $1 < w \leq N/2$, for the range of batch sizes of interest. The new method is the best available for our setting, constrained sized batches verified by single processor systems, when the number of invalid signatures in a batch can vary considerably but does not exceed $N/2$. The method is applicable to a number of batch verified signature schemes, those presented in [11] and that of Zhang et al. [43].

In [42] the authors investigated using generic verifier methods derived from group testing algorithms for invalid signature identification. Of the five methods they discussed, two — Binary Splitting and HGBS — were identified as the best methods for single processor verifiers. A number of group testing algorithms such as HGBS rely on an estimate, d_w, of the number of invalid signatures. In [42] the authors state that when d_w differs from w "it is unclear to what extent this will hurt the performance of the algorithm." We investigated this issue for the expected performance of HGBS and showed that the impact can be severe.

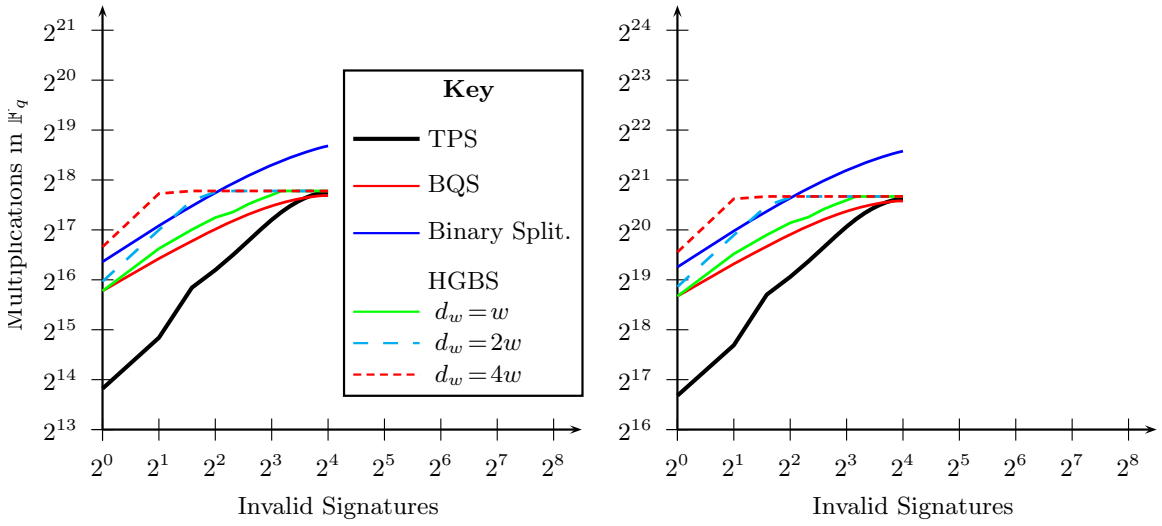
The authors also observed that more restrictive verifiers such as the Exponentiation and EwS methods of Law and Matt [18] and the hybrid methods of Matt [21] (and by extension ours) will outperform their generic verifiers for the class of signature schemes to which these methods apply. We observe that BQS assumes only a common feature of many batch verifiers, yet outperforms the generic group testing based verifiers, especially when the choice of value of d is uncertain.

9 Acknowledgment

Prepared in part through collaborative participation in the Communications and Networks Consortium sponsored by the U. S. Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD-19-01-2-0011. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

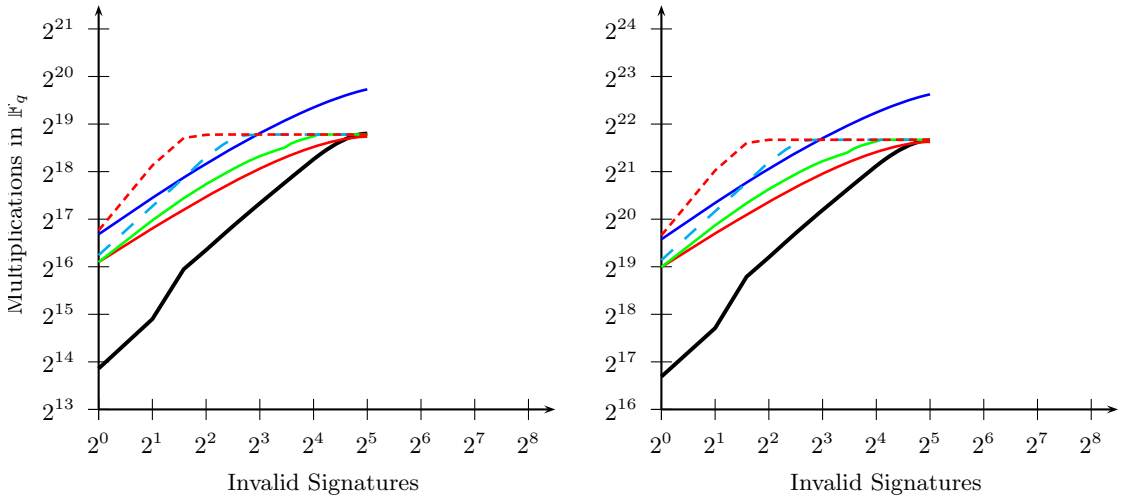
10 Disclaimer

The views and conclusions contained in this paper are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U. S. Government.



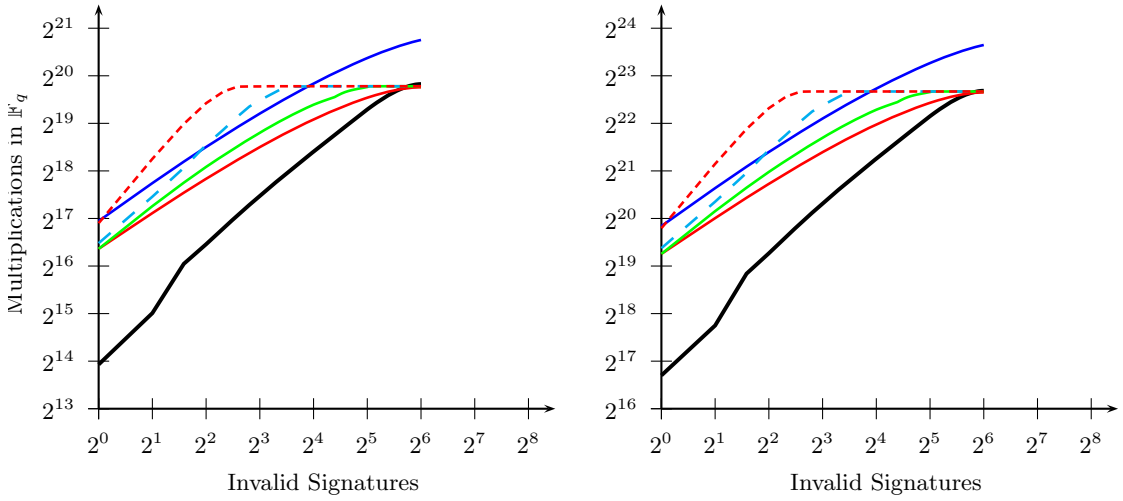
(a) Case A (b) Case E

Fig. 9: Number of multiplies in $\mathbb{F}_q$, $N = 16$.



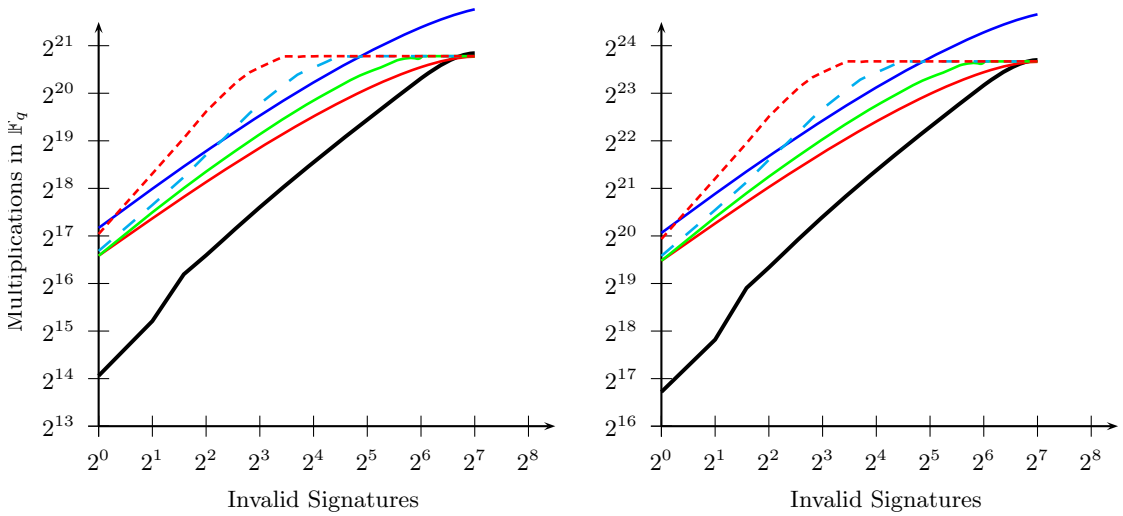
(a) Case A (b) Case E

Fig. 10: Number of multiplies in $\mathbb{F}_q$, $N = 32$.



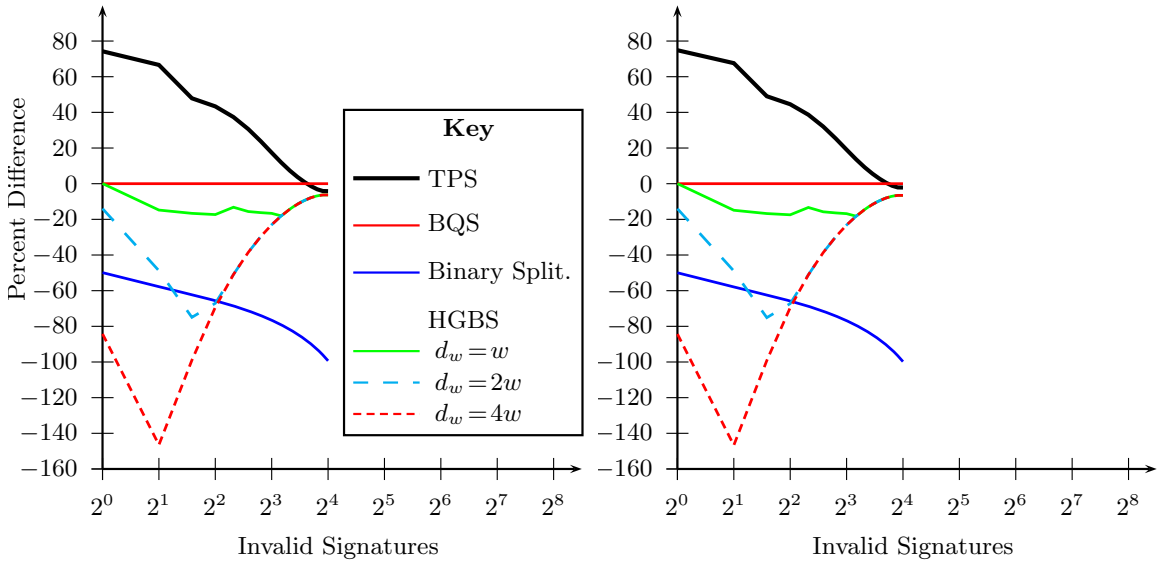
(a) Case A (b) Case E

Fig. 11: Number of multiplies in $\mathbb{F}_q$, $N = 64$.



(a) Case A (b) Case E

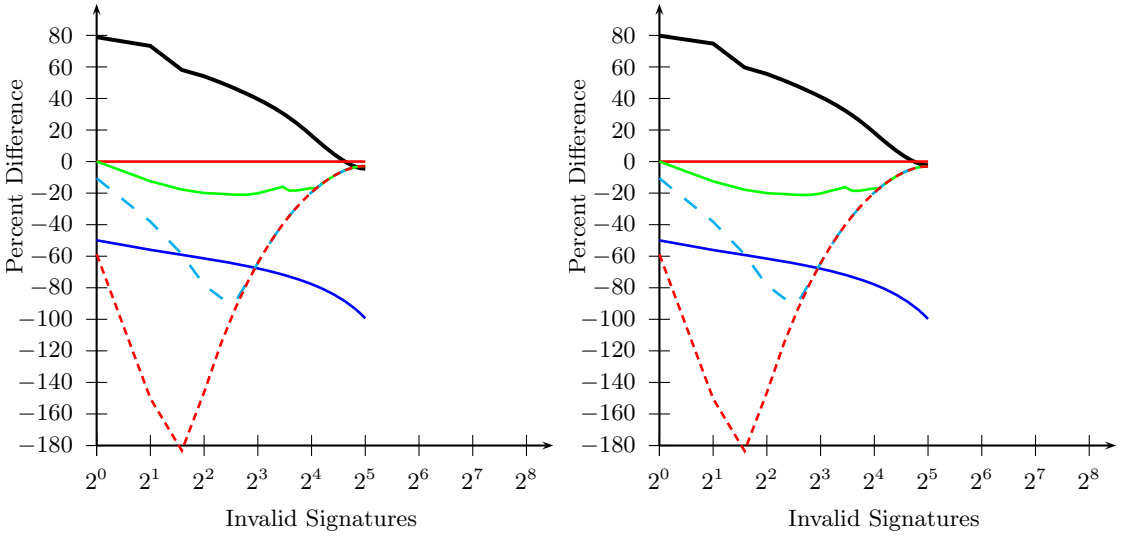
Fig. 12: Number of multiplies in $\mathbb{F}_q$, $N = 128$.



(a) Case A

(b) Case E

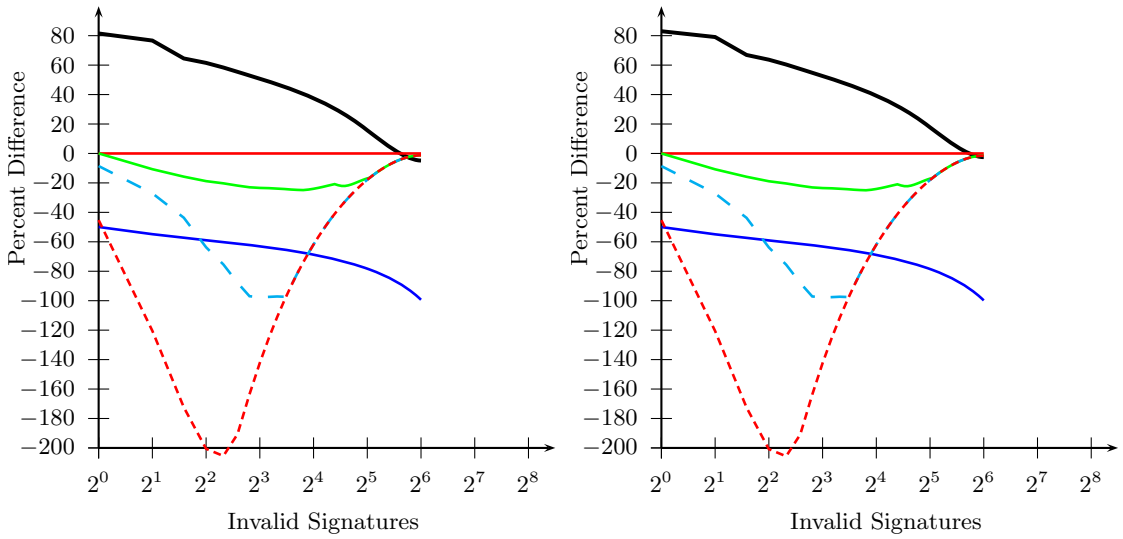
Fig. 13: Percent Difference Comparison with BQS, $N = 16$.



(a) Case A

(b) Case E

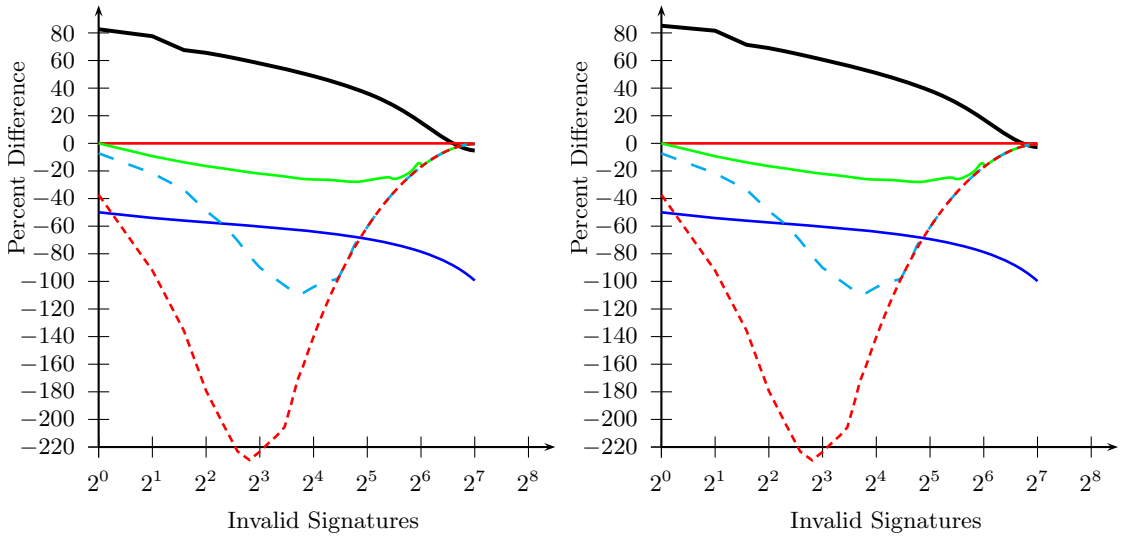
Fig. 14: Percent Difference Comparison with BQS, $N = 32$.



(a) Case A

(b) Case E

Fig. 15: Percent Difference Comparison with BQS, $N = 64$.



(a) Case A

(b) Case E

Fig. 16: Percent Difference Comparison with BQS, $N = 128$.

References

1. Arazi, B.: Certification of dl/ec keys. Submission to P1363 (August 1998) (updated May 1999), http://grouper.ieee.org/groups/1363/StudyGroup/Hybrid.html.
2. Bellare, M., Garay, J., Rabin, T.: Fast batch verification for modular exponentiation and digital signatures. In Nyberg, K., ed.: EUROCRYPT 1998. Volume 1403 of LNCS., Springer-Verlag (1998) 236–250
3. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. In Boyd, C., ed.: ASIACRYPT 2001. Volume 2248 of LNCS., Springer-Verlag (2001) 514–532
4. Brown, D., Gallant, R., Vanstone, S.: Provably secure implicit certificate schemes. In Syverson, P., ed.: Financial Cryptography 2001. Volume 2339 of LNCS., Springer-Verlag (2001) 105–120
5. Buchegger, S., Boudec, J.Y.L.: Performance analysis of the CONFIDANT protocol (Cooperation of Nodes: Fairness In Dynamic Ad-hoc NeTworks). In: ACM/SIGMOBILE Third International Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC), ACM Press (June 2002)
6. Camenisch, J., Hohenberger, S., Pedersen., M.: Batch verification of short signatures. In Naor, M., ed.: Advances in Cryptology - Eurocrypt 2007 Proceedings. Volume 4515 of LNCS., Springer-Verlag (2007) 246–263
7. Cha, J., Cheon, J.: An identity-based signature from gap diffie-hellman groups. In Desmedt, Y., ed.: Public Key Cryptography - PKC 2003. Volume 2567 of LNCS., Springer-Verlag (2003) 18–30
8. Devegili, A.J., hÉigeartaigh, C.O., Scott, M., Dahab, R.: Multiplication and squaring on pairing-friendly fields. Technical report, Cryptology ePrint Archive, Report 2006/471, 2006. http://eprint.iacr.org/.
9. Du, D., Hwang, F.K.: Combinatorial Group Testing And Its Applications (2nd Ed.). World Scientific (Dec. 1999)
10. Fall, K.: A delay-tolerant network architecture for challenged internets. In: SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, ACM Press (2003) 27–34
11. Ferrara, A.L., Green, M., Hohenberger, S., Pedersen, M.O.: Practical short signature batch verification. In Fischlin, M., ed.: Topics in Cryptography – CT-RSA 2009. Volume 5473 of LNCS., Springer-Verlag (2009) 309–324
12. Fiat, A.: Batch RSA. In Brassard, G., ed.: CRYPTO 1989. Volume 435 of LNCS., Springer-Verlag (1989) 175–185
13. FIPS 186-2: Digital Signature Standard (DSS). Federal Information Processing Standards Publication 186-2 (January 2000)
14. Granger, R., Page, D., Smart, N.P.: High security pairing-based cryptography revisited. In Hess, F., Pauli, S., Pohst, M., eds.: Algorithmic Number Theory Symposium - ANTS VII. Volume 4076 of LNCS., Springer-Verlag (2006) 480–494
15. Granger, R., Smart, N.P.: On computing products of pairings. Cryptology ePrint Archive, Report 2006/172, http://eprint.iacr.org/2006/172. (2006)
16. Hwang, F.K.: A method for detecting all defective members in a population by group testing. Journal of the American Statistical Association **67**(339) (1972)
17. Lamparter, B., Paul, K., Westhoff, D.: Charging support for ad hoc stub networks. Journal of Computer Communication **26**(13) (2003) 1504–1514
18. Law, L., Matt, B.J.: Finding invalid signatures in pairing-based batches. In Galbraith, S., ed.: In the proceedings of the Eleventh IMA International Conference on Cryptography and Coding. Volume 4887 of LNCS., Springer-Verlag (2007) 35–53
19. Lee, S., Cho, S., Choi, J., Cho, Y.: Batch verification with DSA-type digital signatures for ubiquitous computing. In Hao, Y., et al., eds.: Computational Intelligence and Security (CIS) 2005. LNCS, Springer-Verlag (2005) 125–130
20. Lee, S., Cho, S., Choi, J., Cho, Y.: Efficient identification of bad signatures in RSA-type batch signature. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences **E89-A**(1) (2006) 74–80
21. Matt, B.J.: Identification of multiple invalid signatures in pairing-based batched signatures. In Jarecki, S., Tsudik, G., eds.: Public Key Cryptography, PKC 2009. Volume 5443 of LNCS., Springer-Verlag (2009) 337–356

22. Matt, B.J.: Identification of multiple invalid signatures in pairing-based batched signatures. Cryptology ePrint Archive (2009) http://eprint.iacr.org/2009.
23. Matt, B.J.: Identification of multiple invalid pairing-based signatures in constrained batches. In Joyce, M., Miyaji, A., Otsuka, A., eds.: 4^{th} International Conference on Pairing Based Cryptography (Pairing 2010). Volume 6487 of LNCS., Springer-Verlag (2010) 78–95 http://www.springer.com/content/866670h782051976/.
24. Naccache, D., M'Raihi, D., Vaudenay, S., Raphaeli, D.: Can D.S.A. be improved? complexity trade-offs with the Digital Signature Standard. In Santis, A.D., ed.: EUROCRYPT 1994. Volume 950 of LNCS., Springer-Verlag (1994) 77–85
25. Papadimitratos, P., Haas, Z.: Secure routing for mobile ad hoc networks. In: SCS Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS) 2002. (January 2002) http://wnl.ece.cornell.edu/Publications/cnds02.pdf.
26. Papadimitratos, P., Haas, Z.: Secure link state routing for mobile ad hoc networks. In: Proc. IEEE Workshop on Security and Assurance in Ad Hoc Networks, IEEE (2003) 27–31
27. Pastuszak, J., Michalek, D., Pieprzyk, J., Seberry, J.: Identification of bad signatures in batches. In Santis, A.D., ed.: Public Key Cryptography - PKC 2000. Volume 1751 of LNCS., Springer-Verlag (2000) 28–45
28. Paul, K., Westhoff, D.: Context aware detection of selfish node in DSR based ad-hoc networks. In: Proceedings of IEEE GLOBECOM '02, IEEE (November 2002)
29. Pintsov, L., Vanstone, S.: Postal revenue collection in the digital age. In Frankel, Y., ed.: Financial Cryptography 2000. Volume 1962 of LNCS., Springer-Verlag (2000) 105–120
30. Raya, M., Hubaux, J.P.: Securing vehicular ad hoc networks. Journal of Computer Security, Special Issue on Security of Ad Hoc and Sensor Networks **15**(1) (2007) 39–68
31. Salem, N.B., Buttyan, L., Hubaux, J.P., Jakobsson, M.: A charging and rewarding scheme for packet forwarding in multi-hop cellular networks,. In: ACM/SIGMOBILE 4th International Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC), ACM Press (June 2003)
32. Sampigethaya, K., Mingyan, L., Leping, H., Poovendran, R.: Amoeba: Robust location privacy scheme for vanet. IEEE JSAC Special Issue on Vehicular Networks **25**(8) (October 2007) 1569–1589
33. Sanzgiri, K., Dahill, B., Levine, B., Shields, C., Belding-Royer, E.M.: A secure routing protocol for ad hoc networks. In: ICNP '02: Proceedings of the 10th IEEE International Conference on Network Protocols, IEEE (2002) 78–89
34. Shanks, D.: Class number, a theory of factorization and genera. In: Symposium on Pure Mathematics. Volume 20., AMS (1971) 415–440
35. Solinas, J.: Identity-based digital signature algorithms. 7th Workshop on Elliptic Curve Cryptography (ECC 2003) (August 2003) invited talk.
36. Stanek, M.: Attacking LCCC batch verification of RSA signatures. Cryptology ePrint Archive, Report 2006/111, http://eprint.iacr.org/2006/111. (2006)
37. Symington, S., Farrell, S., Weiss, H., Lovell, P.: Bundle security protocol specification. draft-irtf-dtnrg-bundle-security-04 (September 2007) work in progress.
38. Yen, S., Laih, C.: Improved digital signature suitable for batch verification. IEEE Transactions on Computers **44**(7) (1995) 957–959
39. Yi, X.: An identity-based signature scheme from the weil pairing. IEEE Communications Letters **7**(2) (Feburary 2003)
40. Yoon, H., Cheon, J.H., Kim, Y.: Batch verifications with ID-based signatures. In Park, C., Chee, S., eds.: ICISC 2004. Volume 3506 of LNCS., Springer-Verlag (2005) 223–248
41. Zapata, M.G., Asokan, N.: Securing ad hoc routing protocols. In: WiSE '02: Proceedings of the 1st ACM workshop on Wireless security, ACM Press (2002) 1–10
42. Zaverucha, G., Stinson, D.: Group testing and batch verification. In Kurosawa, K., ed.: The 4th International Conference on Information Theoretic Security (ITCS 2009). Volume 5973 of LNCS., Springer-Verlag (2009) 140–157
43. Zhang, C., Lu, R., Lin, X., Ho, P.H., Shen, X.: An efficient identity-based batch verification scheme for vehicular sensor networks. In: The 27th IEEE International Conference on Computer Communications (INFOCOM 2008), IEEE (2008)
44. Zhong, S., Yang, Y., Chen, J.: Sprite: a simple, cheat-proof, credit-based system for mobile ad-hoc networks. In: Proceedings of IEEE INFOCOM'03, IEEE (March 2003)

A Auxiliary Algorithms for TPS

This section describes the new tests used by the TPS method in Section 4, $FastFactor(X)$, and $TriFactor(Parent)$. and the functions $Get_2(X)$ and $Get_1(X)$. [6]

A.1 $Get_2(X)$

TPS uses $Get_2(X)$ to obtain $\alpha_{2,X}$ (and $\alpha_{2,X}^{-1}$) for the (sub-)batch X. Algorithm A.1 presents Get_2 for batch verifers of the form $\alpha_0 = \prod_{h=0}^{\bar{\mathsf{n}}-1} e\left(\sum_{i=1}^{N} D_{i,h}, T_h\right)$. In Algorithm A.1, $V0_{i,h} = \sum_{j=N}^{i} D_{j,h}$ computed by $Get_0(B)$, and $V1_{i,h} = i\sum_{j=N}^{i+1} D_{j,h} + \sum_{j=i}^{1} jD_{j,h}$ by $Get_1(B)$. The cost to compute $\alpha_{2,B}$ and its inverse is $\bar{n} \cdot |B| \cdot \mathsf{CstAdd}\mathbb{G}_1 + \mathsf{CstInv}\mathbb{G}_\mathrm{T} + \mathsf{CstMultPair}$. We use $\bar{n} \cdot |B| \cdot \mathsf{CstAdd}\mathbb{G}_1 + \mathsf{CstMultPair}$ for this case in Appendix 5.

When X is a left child the cost is no more than $3 \cdot \mathsf{CstAdd}\mathbb{G}_1 + 2 \cdot \mathsf{CstSub}\mathbb{G}_1 + \mathsf{CstInv}\mathbb{G}_\mathrm{T} + \mathsf{CstDlbMult}\mathbb{G}_1(\mathrm{t}_1, \mathrm{t}_2) + \mathsf{CstMultPair}$ with $t_1 < \lceil log_2(N) + 1 \rceil$ and $t_2 < \lceil 2log_2(N) \rceil$. We estimate the cost of $Get_2(X)$ in this case as $\mathsf{CstMultPair}$. If X is a right child and α_2 for its sibling has been computed, the cost is at most $2\,\mathsf{CstMult}\mathbb{G}_\mathrm{T}$ which we ignore. If the position of the single invalid signature in the sibling of X is known then the cost of computing $\alpha_{2,X}$ and its inverse is $2\,\mathsf{CstMult}\mathbb{G}_\mathrm{T} + \mathsf{CstInv}\mathbb{G}_\mathrm{T} + \mathsf{CstExpt}\mathbb{G}_\mathrm{T}(\mathrm{t}_1)$ where $t_1 < \lceil log_2(N) \rceil$. We use $\mathsf{CstExpt}\mathbb{G}_\mathrm{T}(\mathrm{t}_1)$ for this case in Appendix 5. Since $\mathsf{CstExpt}\mathbb{G}_\mathrm{T}(\mathrm{t}_1) \ll \mathsf{CstMultPair}$ we could ignore this cost as well.

A.2 $Get_1(X)$

TPS uses $Get_1(X)$ to obtain $\alpha_{1,X}$ (and $\alpha_{1,X}^{-1}$) for the (sub-)batch X. Algorithm A.2 presents Get_1 for batch verifers of the form $\alpha_0 = \prod_{h=0}^{\bar{\mathsf{n}}-1} e\left(\sum_{i=1}^{N} D_{i,h}, T_h\right)$. In Algorithm A.2, $V0_{i,h} = \sum_{j=N}^{i} D_{j,h}$ computed by $Get_0(B)$. The cost to compute $\alpha_{1,B}$ and its inverse is $\bar{n} \cdot |B| \cdot \mathsf{CstAdd}\mathbb{G}_1 + \mathsf{CstInv}\mathbb{G}_\mathrm{T} + \mathsf{CstMultPair}$. We use $\bar{n} \cdot |B| \cdot \mathsf{CstAdd}\mathbb{G}_1 + \mathsf{CstMultPair}$ for this case in Appendix 5.

When X is a left child the cost is no more than $3 \cdot \mathsf{CstAdd}\mathbb{G}_1 + 2 \cdot \mathsf{CstSub}\mathbb{G}_1 + \mathsf{CstInv}\mathbb{G}_\mathrm{T} + \mathsf{CstDlbMult}\mathbb{G}_1(\mathrm{t}_1, \mathrm{t}_2) + \mathsf{CstMultPair}$ with $t_1 < \lceil log_2(N) + 1 \rceil$ and $t_2 < \lceil 2log_2(N) \rceil$. We estimate the cost of $Get_2(X)$ in this case as $\mathsf{CstMultPair}$. If X is a right child and α_2 for its sibling has been computed, the cost is at most $2\,\mathsf{CstMult}\mathbb{G}_\mathrm{T}$ which we ignore.

[6] TPS uses a modified version of $Get_1(X)$ which orginally appeared in [22], as well as $Get_0(X)$ which also appeared in [22].

Algorithm A.1 $Get_2(X)$ *(Obtain α_2 (and α_2^{-1}))*

Input: X a list of message / signature pairs.
Output: None.
Return: The value $\alpha_{2,[X]}$ for X.

$P \leftarrow Parent(X)$; $L \leftarrow Left(P)$; $R \leftarrow Right(P)$; $\hat{X} \leftarrow Sibling(X)$
if $(\alpha_{2,[X]})$ **then** // True if $\alpha_{2,[X]}$ has been computed
    return $(\alpha_{2,[X]})$
else if $(\alpha_{0,[X]} = \alpha_{0,[P]})$ **then**
    $\alpha_{2,[X]} \leftarrow \alpha_{2,[P]}$
    $inv\alpha_{2,[X]} \leftarrow inv\alpha_{2,[P]}$
    return $(\alpha_{2,[X]})$
else if $(ses[\hat{X}] \neq 0)$ **then** // Shanks found $\alpha_{1,[\hat{X}]}$ and stored the position of the invalid signature in $ses[\hat{X}]$
    $\alpha_{2,[X]} \leftarrow \alpha_{2,[X]} \cdot inv\alpha_{1,[\hat{X}]}^{ses[\hat{X}]}$
    $inv\alpha_{2,[X]} \leftarrow \alpha_{2,[X]}^{-1}$
    return $(\alpha_{2,[X]})$
else if $(X = R)$ **then** // Right child
    $\alpha_{2,[R]} \leftarrow \alpha_{2,[P]} \cdot inv\alpha_{2,[L]}$
    $inv\alpha_{2,[R]} \leftarrow inv\alpha_{2,[P]} \cdot \alpha_{2,[L]}$
    return $(\alpha_{2,[R]})$
else if $(X = L)$ **then** // Left child
    $l \leftarrow lowbnd(L)$; $u \leftarrow upbnd(L)$
    for $h = 0$ **to** $\bar{n} - 1$ **do**
        $W2_{l,u,h} \leftarrow V2_{1,h} - (2 \cdot V2_{u+1,h} + (2u-1) \cdot V1_{u+1,h} + u^2 \cdot V0_{u+1,h})$
    if $(l \neq 1)$ **then**
        if $(W2_{1,l-1})$ **then** // Test whether $W2_{1,l-1,h=\{0,\bar{n}-1\}}$ has been computed
            for $h = 0$ **to** $\bar{n} - 1$ **do**
                $W2_{l,u,h} \leftarrow W2_{l,u,h} - W2_{1,l-1,h}$
        else
            for $h = 0$ **to** $\bar{n} - 1$ **do**
                $W2_{1,l-1,h} \leftarrow V2_{1,h} - (2 \cdot V2_{l,h} + (2l-3) \cdot V1_{l,h} + (l-1)^2 \cdot V0_{l,h})$
                $W2_{l,u,h} \leftarrow W2_{l,u,h} - W2_{1,l-1,h}$
    $\alpha_{2,[L]} \leftarrow \prod_{h=0}^{\bar{n}-1} e\,(W2_{l,u,h}, T_h)$
    if $(\ \alpha_{2,[L]} \neq \alpha_{2,[P]})$ **then**
        $inv\alpha_{2,[L]} \leftarrow \alpha_{2,[L]}^{-1}$
    else
        $inv\alpha_{2,[L]} \leftarrow inv\alpha_{2,[P]}$
    return $(\alpha_{2,[L]})$
else // Root node, compute $\alpha_{2,[B]}$
    for $h = 0$ **to** $\bar{n} - 1$ **do**
        $V2_{|X|,h} \leftarrow V1_{|X|,h}$
    for $i = |X| - 1$ **to** 1 **do**
        for $h = 0$ **to** $\bar{n} - 1$ **do**
            $V2_{i,h} \leftarrow V2_{i+1,h} + V1_{i,h}$
    for $h = 0$ **to** $\bar{n} - 1$ **do**
        $V2_{1,h} \leftarrow V2_{1,h} + V2_{2,h}$
    $\alpha_{2,[B]} \leftarrow \prod_{h=0}^{\bar{n}-1} e\,(V2_{1,h}, T_h)$
    $inv\alpha_{2,[B]} \leftarrow \alpha_{2,[B]}^{-1}$
    return $(\alpha_{2,[B]})$

A.3 *FastFactor*(X)

$FastFactor(X)$ shown in Algorithm A.3 below tests whether the equation

$$\alpha_{2,X}^4 \cdot (\alpha_{1,X}^{-4})^n \cdot \alpha_{0,X}^{n^2} = \alpha_{0,X}^{m^2} \tag{4}$$

has a solution when with $2l+1 \leq n \leq 2(l+|X|)-1$ and $1 \leq m \leq |X|-1$ where $l = lowbnd(X)$. If a solution exists then $z_2 = (n+m)/2$ and $z_1 = (n-m)/2$ where $z_2 > z_1$ are the positions of the two invalid signatures. The algorithm is an improved version of the Factor Method [18]. Since $2|n \iff 2|m$, a solution to (4) can be found by searching for a solution to (5) and (6).

$$\alpha_{2,X}^4 \cdot (\alpha_{1,X}^{-4})^{2\hat{n}} \cdot (\alpha_{0,X}^4)^{\hat{n}^2} = (\alpha_{0,X}^4)^{\hat{m}^2} \tag{5}$$

In (5) $n = 2\hat{n}$ and $m = 2\hat{m}$ and the algorithm tests whether a solution exist such that $l+1 \leq \hat{n} \leq (l+|X|)-2$ and $1 \leq \hat{m} \leq |X|/2-1$.

$$\alpha_{2,X}^4 \cdot (\alpha_{1,X}^{-4})^{2\hat{n}+1} \cdot (\alpha_{0,X}^4)^{\hat{n}^2+\hat{n}+1} = (\alpha_{0,X}^4)^{\hat{m}^2+\hat{m}+1} \tag{6}$$

In (6) $n = 2\hat{n}+1$ and $m = 2\hat{m}+1$ and the algorithm tests whether a solution exist such that $l \leq \hat{n} \leq (l+|X|)-2$ and $0 \leq \hat{m} \leq |X|/2-1$.

In the algorithm the function $findMatch(leftside, rightside, Parent, flag)$ searches for a match between the used locations in the lists $leftside$ and $rightside$ and returns the positions of the invalid signatures in B or $(0,0)$.

When $l \neq 1$ the worse case cost of the algorithm is $3\,\mathsf{CstExpt}\mathbb{G}_{\mathrm{T}}(\mathrm{t}_1)$ and $(4.5|X|)\,\mathsf{CstMult}\mathbb{G}_{\mathrm{T}}$. When $l = 1$ the cost is about $4.5|X|\,\mathsf{CstMult}\mathbb{G}_{\mathrm{T}}$ compared to the worse case cost $8|X|$ of the original Factor Method [18]. The average cost of the algorithm is $3\,\mathsf{CstExpt}\mathbb{G}_{\mathrm{T}}(\mathrm{t}_1)$ when $l \neq 1$, plus approximately $2\frac{3}{4}\,|X|\,\mathsf{CstMult}\mathbb{G}_{\mathrm{T}}$ compared to approximately $4.8\,|X|\,\mathsf{CstMult}\mathbb{G}_{\mathrm{T}}$ to $4.3\,|X|\,\mathsf{CstMult}\mathbb{G}_{\mathrm{T}}$ of the original [22]. Since $\mathsf{CstMultPair} >> \mathsf{CstExpt}\mathbb{G}_{\mathrm{T}}(\mathrm{t}_1)$, we ignore the $3\,\mathsf{CstExpt}\mathbb{G}_{\mathrm{T}}(\mathrm{t}_1)$ cost as well as the cost of the small constant number $\mathsf{CstMult}\mathbb{G}_{\mathrm{T}}$ in Appendix 5.

Algorithm A.2 *$Get_1(X)$ (Obtain α_1 (and α_1^{-1}))*

Input: X a list of message / signature pairs.
Output: None.
Return: The value $\alpha_{1,[X]}$ for X.

$P \leftarrow Parent(X)$; $L \leftarrow Left(P)$; $R \leftarrow Right(P)$; $\hat{X} \leftarrow Sibling(X)$
if $(\alpha_{1,[X]})$ **then** // True if $\alpha_{1,[X]}$ has been computed
    return $(\alpha_{1,[X]})$
else if $(\alpha_{0,[X]} = \alpha_{0,[P]})$ **then**
    $\alpha_{1,[X]} \leftarrow \alpha_{1,[P]}$
    $inv\alpha_{1,[X]} \leftarrow inv\alpha_{1,[P]}$
    return $(\alpha_{1,[X]})$
else if $(X = R)$ **then** // Right child
    $\alpha_{1,[R]} \leftarrow \alpha_{1,[P]} \cdot inv\alpha_{1,[L]}$
    $inv\alpha_{1,[R]} \leftarrow inv\alpha_{1,[P]} \cdot \alpha_{1,[L]}$
    return $(\alpha_{1,[R]})$
else if $(X = L)$ **then** // Left child
    $l \leftarrow lowbnd(L)$; $u \leftarrow upbnd(L)$
    for $h = 0$ **to** $\bar{\mathsf{n}} - 1$ **do**
        $W1_{l,u,h} \leftarrow V1_{1,h} - (V1_{u+1,h} + u \cdot V0_{u+1,h})$
    if $(l \neq 1)$ **then**
        if $(W1_{1,l-1})$ **then** // Test whether $W1_{1,l-1,h=\{0,\bar{\mathsf{n}}-1\}}$ has been computed
            for $h = 0$ **to** $\bar{\mathsf{n}} - 1$ **do**
                $W1_{l,u,h} \leftarrow W1_{l,u,h} - W1_{1,l-1,h}$
        else
            for $h = 0$ **to** $\bar{\mathsf{n}} - 1$ **do**
                $W1_{1,l-1,h} \leftarrow V1_{1,h} - (V1_{l,h} + (l-1) \cdot V0_{l,h})$
                $W1_{l,u,h} \leftarrow W1_{l,u,h} - W1_{1,l-1,h}$
    $\alpha_{1,[L]} \leftarrow \prod_{h=0}^{\bar{\mathsf{n}}-1} e\,(W1_{l,u,h}, T_h)$
    if $(\ \alpha_{1,[L]} \neq \alpha_{1,[P]})$ **then**
        $inv\alpha_{1,[L]} \leftarrow \alpha_{1,[L]}^{-1}$
    else
        $inv\alpha_{1,[L]} \leftarrow inv\alpha_{1,[P]}$
    return $(\alpha_{1,[L]})$
else // Root node, compute $\alpha_{1,[B]}$
    for $h = 0$ **to** $\bar{\mathsf{n}} - 1$ **do**
        $V1_{|X|,h} \leftarrow V0_{|X|,h}$
    for $i = |X| - 1$ **to** 1 **do**
        for $h = 0$ **to** $\bar{\mathsf{n}} - 1$ **do**
            $V1_{i,h} \leftarrow V1_{i+1,h} + V0_{i,h}$
    $\alpha_{1,[B]} \leftarrow \prod_{h=0}^{\bar{\mathsf{n}}-1} e\,(V1_{1,h}, T_h)$
    $inv\alpha_{1,[B]} \leftarrow \alpha_{1,[B]}^{-1}$
    return $(\alpha_{1,[B]})$

Algorithm A.3 *FastFactor*(X) *(Identify two invalid signatures in* X*)*

Input: X a list of message / signature pairs.
Output: None.
Return: The indexes in B of both invalid signatures in X or $(0, 0)$.

$l \leftarrow lowbnd(X)$
$rightsideODD[0] \leftarrow rightsideEVEN[1] \leftarrow \beta_1 \leftarrow \alpha^4_{0,[X]}$
$leftsideODD[l] \leftarrow \alpha^4_{2,[X]} * (inv\alpha^4_{1,[X]})^{2l+1} * (\beta_1)^{l^2+l+1}$
$\delta_l \leftarrow inv\alpha^4_{1,[X]} * (\beta_1)^l$
$leftsideEVEN[l+1] \leftarrow leftsideODD[l] * \delta_l$
$\beta_2 \leftarrow \beta_1 * \beta_1$
$rightsideODD[1] \leftarrow rightsideEVEN[1] * \beta_2$
$\delta_{l+2} \leftarrow \delta_l * \beta_2$
$leftsideODD[l+1] \leftarrow leftsideEVEN[l+1] * \delta_{l+2}$
if $(((z_1, z_2) \leftarrow findMatch(leftsideEVEN, rightsideEVEN, X, even)) \neq (0, 0))$ **then**
    return (z_1, z_2)
if $(((z_1, z_2) \leftarrow findMatch(leftsideODD, rightsideODD, X, odd)) \neq (0, 0)))$ **then**
    return (z_1, z_2)
$\delta_{l+1} \leftarrow \delta_l * \beta_1$
for i = 2 **to** —X—-1 **do**
    if $i <= (|X|/2) - 1$ **then**
        $rightsideEVEN[i] \leftarrow rightsideODD[i-1] * \beta_{i-1}$
    $leftsideEVEN[l+i] \leftarrow leftsideODD[l+i-1] * \delta_{l+i-1}$
    if $(((z_1, z_2) \leftarrow findMatch(leftsideEVEN, rightsideEVEN, X, even)) \neq (0, 0))$ **then**
        return (z_1, z_2)
    if $i <= (|X|/2) - 1$ **then**
        $\beta_{i+1} \leftarrow \beta_i * \beta_1$
        $rightsideODD[i] \leftarrow rightsideEVEN[i] * \beta_{i+1}$
    $\delta_{l+i+1} \leftarrow \delta_{l+i} * \beta_1$
    $leftsideODD[l+i] \leftarrow leftsideEVEN[l+i] * \delta_{l+i+1}$
    if $(((z_1, z_2) \leftarrow findMatch(leftsideODD, rightsideODD, X, odd)) \neq (0, 0)))$ **then**
        return (z_1, z_2)
return $(0, 0)$

A.4 *TriFactor(Parent)*

The algorithm for $TriFactor(Parent)$ shown on page 29 uses $TriSolver(Parent, Left, Right)$ to test whether equation

$$\alpha_{2,B}^4 \cdot (\alpha_{0,R}^{-4})^{z_3^2} \cdot (\alpha_{1,B}^{-4})^{n_L} \cdot (\alpha_{0,R}^4)^{n_l \cdot z_3} \cdot \alpha_{0,L}^{{n_L}^2} = \alpha_{0,L}^{{m_L}^2} \tag{7}$$

has a solution with $2l_L + 1 \le n_L \le 2(l_L + |L|) - 1$, $1 \le m_L \le |L| - 1$ and $l_R \le z_3 < l_R + |R|$, where $l_L = lowbnd(L)$ (L is $Left$ the left child of $Parent$) and similarly for l_R (and R). If the test succeeds then $z_2 = (n_L + m_L)/2$ and $z_1 = (n_L - m_L)/2$ where $z_2 > z_1$ are the positions of the two invalid signatures in L and z_3 is the position of the invalid signature in R. If the test fails then $TriSolver(Parent, Right, Left)$ is used to test whether

$$\alpha_{2,B}^4 \cdot (\alpha_{0,L}^{-4})^{z_1^2} \cdot (\alpha_{1,B}^{-4})^{n_R} \cdot (\alpha_{0,L}^4)^{n_R \cdot z_1} \cdot \alpha_{0,R}^{{n_R}^2} = \alpha_{0,R}^{{m_R}^2} \tag{8}$$

has a solution with $2l_R + 1 \le n_R \le 2(l_R + |R|) - 1$, $1 \le m_R \le |R| - 1$ and $l_L \le z_1 < l_L + |L|$. If the test succeeds then $z_3 = (n_R + m_R)/2$ and $z_2 = (n_R - m_R)/2$ where $z_3 > z_2$ are the positions of the two invalid signatures in R and z_1 is the position of the invalid signature in L.

Similar to $FastFactorMethod$, the $TriSolver$ algorithm exploits the observation that $2|n_X \iff 2|m_X$ where $X = L$ or $X = R$. If the test of (7) is successful the expected cost of $TriSolver(Left, Right)$ is approximately

$$\frac{1}{32}\left(40 + 14\sqrt{2} + 3\left(3 + \sqrt{2}\right)M\right) M \, \mathsf{CstMult}\mathbb{G}_{\mathrm{T}}$$

where $M = |P|$, otherwise the cost of the failed call to $TriSolver(Left, Right)$ is $\frac{1}{4}(11 + 3M)M \, \mathsf{CstMult}\mathbb{G}_{\mathrm{T}}$.

If the test of (8) is successful, then cost is the sum of the failed $TriSolver(Left, Right)$, plus a successful $TriSolver(Right, Left)$ which is $\frac{1}{64}\left(2\left(64 + 7\sqrt{2}\right) + 3\left(11 + \sqrt{2}\right)M\right) M \, \mathsf{CstMult}\mathbb{G}_{\mathrm{T}}$. Therefore the expected approximate cost of successful call to $TriFactor(Parent)$ is $\frac{1}{32}\left(14\left(6 + \sqrt{2}\right) + 3\left(7 + \sqrt{2}\right)M\right) M \, \mathsf{CstMult}\mathbb{G}_{\mathrm{T}}$ and a failed call is $\frac{1}{2}(11 + 3M)M \, \mathsf{CstMult}\mathbb{G}_{\mathrm{T}}$.

The function $findMatch(left, right, Parent, Child1, Child2, z, flag)$ searches for a match between the used locations in the lists $left$ and $right$ and returns the positions of the invalid signatures in B or $(0, 0)$.

Algorithm A.4 *TriFactor(Parent)* *(Identify three invalid signatures in Parent)*

Input: *Parent*, a list of message / signature pairs.
Return: The indexes (z_a, z_b, z_c) in B of three invalid signatures in *Parent*
s.t. $z_a < z_b < z_c$ and z_a is in $Left$ and z_c is in $Right$, or return $(0, 0)$.

$Left \leftarrow Left(Parent)$; $Right \leftarrow Right(Parent)$;
$(z_1, z_2, z_3) \leftarrow TriSolver(Parent, Left, Right)$
if $z_1 \neq 0$ **then**
    return (z_1, z_2, z_3)
$(z_1, z_2, z_3) \leftarrow TriSolver(Parent, Right, Left)$
return (z_3, z_1, z_2)

Function TriSolver(P, X, Y)
Input: X, Y sibling lists of message / signature pairs.
Return: The indexes in B of three invalid signatures, two in X and one in Y.

$l_X \leftarrow lowbnd(X)$; $l_Y \leftarrow lowbnd(Y)$
$b \leftarrow (inv\alpha_{1,[P]} * \alpha_{0,[Y]})^{l_Y}$; $\hat{b} \leftarrow (\alpha_{0,[Y]})$
$a \leftarrow (inv\alpha_{0,[Y]})^{l_Y^2} * b^{2l_X+1}$; $\hat{a} \leftarrow (inv\alpha_{0,[Y]})^{2l_Y+1} * (\alpha_{0,[Y]})^{2l_X+1}$; $\hat{\hat{a}} \leftarrow (inv\alpha_{0,[Y]})^2$
$d \leftarrow (\alpha_{0,[X]})^{l_X^2+l_X+1}$
$e \leftarrow (\alpha_{0,[X]})^{l_X}$
$\beta_1 \leftarrow \alpha_{0,[X]}$; $\beta_2 \leftarrow \beta_1 * \beta_1$
$rightsideEVEN[1] \leftarrow \beta_1$
$rightsideODD[1] \leftarrow rightsideEVEN[1] * \beta_2$
$j \leftarrow 2$
for $z = l_Y$ **to** $l_Y + |Y| - 1$ **do**
    if $j \leq (|Y|/2) - 1$ **then**
        $rightsideEVEN[j] \leftarrow rightsideODD[j-1] * \beta_{j-1}$
        if $(((z_a, z_b) \leftarrow findMatch(leftsideEVEN, rightsideEVEN, P, X, Y, z, even) \neq (0,0)))$ **then**
            return (z, z_a, z_b)
        $\beta_{j+1} \leftarrow \beta_j * \beta_1$
        $rightsideODD[j] \leftarrow rightsideEVEN[j] * \beta_{j+1}$
        $j \leftarrow j + 1$
        if $(((z_a, z_b) \leftarrow findMatch(leftsideODD, rightsideODD, P, X, Y, z, odd) \neq (0,0)))$ **then**
            return (z, z_a, z_b)
    $leftsideODD[l_X, z] \leftarrow \alpha_{2,[P]} * a * d$
    $\delta_l \leftarrow b * e$; $\delta_{l+1} \leftarrow \delta_l * \beta_1$; $\delta_{l+2} \leftarrow \delta_l * \beta_2$
    $leftsideEVEN[l_X + 1, z] \leftarrow leftsideODD[l_X, z] * \delta_l$
    $leftsideODD[l_X + 1, z] \leftarrow leftsideEVEN[l_X + 1, z] * \delta_{l+2}$
    if $(((z_a, z_b) \leftarrow findMatch(leftsideEVEN, rightsideEVEN, P, X, Y, z, even) \neq (0,0)))$ **then**
        return (z, z_a, z_b)
    if $(((z_a, z_b) \leftarrow findMatch(leftsideODD, rightsideODD, P, X, Y, z, odd) \neq (0,0)))$ **then**
        return (z, z_a, z_b)
    for $i = 2$ **to** $|X| - 1$ **do**
        $leftsideEVEN[l_X + i, z] \leftarrow leftsideODD[l_X + i - 1, z] * \delta_{l+i-1}$
        if $(((z_a, z_b) \leftarrow findMatch(leftsideEVEN, rightsideEVEN, P, X, Y, z, even) \neq (0,0)))$ **then**
            return (z, z_a, z_b)
        $\delta_{l+i+1} \leftarrow \delta_{l+i} * \beta_1$
        $leftsideODD[l_X + i, z] \leftarrow leftsideEVEN[l_X + i, z] * \delta_{l+i+1}$
        if $(((z_a, z_b) \leftarrow findMatch(leftsideODD, rightsideODD, P, X, Y, z, odd) \neq (0,0)))$ **then**
            return (z, z_a, z_b)
    $a \leftarrow a * \hat{a}$; $\hat{a} \leftarrow \hat{a} * \hat{\hat{a}}$
    $b \leftarrow b * \hat{b}$
return $(0, 0, 0)$