Machine-Generated Algorithms, Proofs and Software for the Batch Verification of Digital Signature Schemes

Joseph A. Akinyele
Johns Hopkins University
Baltimore, MD, USA
akinyelj@cs.jhu.edu

Matthew Green
Johns Hopkins University
Baltimore, MD, USA
mgreen@cs.jhu.edu

Susan Hohenberger
Johns Hopkins University
Baltimore, MD, USA
susan@cs.jhu.edu

Matthew W. Pagano
Johns Hopkins University
Baltimore, MD, USA
mpagano@cs.jhu.edu

ABSTRACT

As devices everywhere increasingly communicate with each other, many security applications will require low-bandwidth signatures that can be processed quickly. Pairing-based signatures can be very short, but are often costly to verify. Fortunately, they also tend to have efficient batch verification algorithms. Finding these batching algorithms by hand, however, can be tedious and error prone.

We address this by presenting AutoBatch, an automated tool for generating batch verification code in either Python or C++ from a high level representation of a signature scheme. AutoBatch outputs both software and, for transparency, a LaTeX file describing the batching algorithm and arguing that it preserves the unforgeability of the original scheme.

We tested AutoBatch on over a dozen pairing-based schemes to demonstrate that a computer could find competitive batching solutions in a reasonable amount of time. Indeed, it proved highly competitive. In particular, it found an algorithm that is significantly faster than a batching algorithm from Eurocrypt 2010. Another novel contribution is that it handles *cross-scheme* batching, where it searches for a common algebraic structure between two distinct schemes and attempts to batch them together. We describe other features and performance details herein.

AutoBatch is a useful tool for cryptographic designers and implementors, and to our knowledge, it is the first attempt to outsource to machines the design, proof writing and implementation of signature batch verification schemes.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Cryptographic controls, Authentication, Access controls, Verification

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1651-4/12/10 ...$15.00.

Keywords

Digital Signatures, Pairing-Based Cryptography, Batch Verification, Automation, Cryptographic Compilers

1. INTRODUCTION

We anticipate a future where computers are everywhere as an integrated part of our surroundings, continuously exchanging messages, e.g., sensor networks, smartphones, vehicular communications. For these systems to work properly, messages must carry some form of authentication, and yet the system requirements on this authentication are particularly demanding. Applications such as vehicular communications [21, 57], where cars communicate with each other and the highway infrastructure to report on road conditions, traffic congestion, etc., require both that signatures be short (due to the limited spectrum available) and that many messages from different sources can be processed quickly.

Pairing-based signatures are attractive due to their small size, but they often carry a costly verification procedure. Fortunately, these schemes also lend themselves well to *batch verification*, where valuable time is saved by processing many messages at once. E.g., Boneh, Lynn and Shacham [13] presented a 160-bit signature together with a batching algorithm over signatures by the same signer, where verification time could be reduced from 47.6ms to 2.28ms per signature in a batch of 200 [26] — a 95% savings!

To prepare for a future of ubiquitous messaging, we would like batching algorithms for as many pairing-based schemes as possible. Designing batch verification algorithms by hand, however, is challenging. First, it can be tedious. It requires knowledge of many batching rules and exploration of a potentially huge space of algebraic manipulations in the hunt for a good candidate algorithm. Second, it can be error prone. In Section 1.3, we discuss both the success and failure of the past fifteen years in batching digital signatures. The clear lesson is that mistakes are common and that even when generic methods for batching have been suggested, they have often been misapplied (e.g., a critical step is forgotten.) This paper demonstrates that it is feasible for humans to turn over some of the design, proof writing and implementation work in batch verification to machines.

1.1 Our Contributions

We present AutoBatch, an automated tool that transforms a high-level description of a signature scheme[1] into an optimized batch verification program in either Python or C++. The algorithm behind AutoBatch searches for a batching algorithm by repeatedly applying a combination of novel and existing batching techniques. Because some loops or other infinite paths could occur, AutoBatch prunes its search using a set of carefully designed heuristics. The final code also includes logic for altering the behavior of the batching algorithm based on its input size or past input.

To our knowledge, this is the first attempt to automatically identify when certain batching techniques are applicable and to apply them in a secure manner. Importantly, the way in which we combine these techniques and optimizations preserves the unforgeability of the original scheme. Specifically, with all but a negligible probability, the batch verifier will accept a batch S of signatures if and only if every $s \in S$ would have been accepted by the individual verification algorithm. AutoBatch also produces a machine-generated LaTeX file that specifies each technique applied and the argument for why security holds.

AutoBatch was tested on several pairing-based schemes. It produced the first batching algorithms, to our knowledge, for the Camenisch-Lysyanskaya [18], Hohenberger-Waters [33] and Waters dual-system [63] signatures. It also discovered a significantly faster algorithm for batching the proofs of the verifiable random functions in [34]. Moreover, AutoBatch is able to handle batches with more than one type of signature. Indeed, we found that the Hess [32] and Cha-Cheon [22] identity-based signatures can be processed twice as fast when batched together compared to sorting by type and batching within the type. The capability to do *cross-scheme* batching is a novel contribution of this paper, and we feel could be of great value for applications, such as mail servers, which may encounter many signature types at once.

AutoBatch is a tool with many applications for both existing and future signature schemes. It helps enable the secure, but rapid processing of authenticated messages, which we believe will be of increasing importance in a wide-variety of future security applications.

1.2 Overview of Our Approach

We present a detailed explanation of AutoBatch in §3. In this section and in Figure 1 we provide a brief overview of the techniques. At a high level, AutoBatch is designed to analyze a scheme, extract the signature verification equation, and derive working code for a batch verifier. This involves three distinct components:

1. (Optional) A Code Parser, which retrieves the verification equation and variable types from some existing scheme implementation. This process naturally assumes that the scheme has been implemented within certain constraints, which we discuss later in the paper. Given such an implementation, the Parser obtains the signature verification equation and encodes it into an intermediate representation called *Scheme Description Language* (SDL).
2. A Batcher, which takes as input an SDL file describing a signature verification equation. It searches through a series of rules, which may be applied repeatedly, to optimize the equation and thus derive a new equation for a batch verifier. The output of this equation is second SDL file containing the individual and batch equations, along with an analysis of the batcher's estimated running time. For transparency, the Batcher optionally outputs a human-readable file written in LaTeX describing the batching algorithm and containing a security proof that it maintains the unforgeability of the original scheme.
3. A Code Generator, which takes the output of Batcher and generates working source code to implement the batch verifier. The user can choose either Python or C++ as the output language; either building on the MIRACL library [56]. Beyond simply implementing the verification equation, the Generator adds a series of additional components, including group element membership checks, a recursive divide-and-conquer process to handle batches that contain *invalid* signatures, and additional logic to identify cases where individual verification is likely to outperform batching.

There are two usage scenarios for AutoBatch. The most common may be that a user begins with a hand-coded SDL file and feeds this directly into the Batcher. Since SDL files are human-readable ASCII-based files containing a mathematical representation of the scheme, some developers may prefer to implement new schemes directly in this language, which is agnostic to the programming language of the final implementation.

As a second scenario, if the user already has a working implementation of the scheme in Charm/Python [1], then she can save time. This program can be given to the Code Parser, which will extract the necessary information from the code to generate an SDL file. Charm [1] is a Python-based prototyping framework created by Akinyele, Green and Rubin that provides infrastructure for developing advanced cryptographic schemes. There is already a library of pairing-based signatures publicly available in Charm/Python, so we provide this as a second interface option to our tool.

1.3 Related Work

Computer-aided security is a goal of high importance. Recently, the best paper award at CRYPTO 2011 was given to Barthe, Grégoire, Heraud and Zanella Béguelin [8] for their invention of EasyCrypt, an automated tool for generating security proofs of cryptographic system from proof sketches. The reader is referred there for a summary of efforts to automate the verification of cryptographic security proofs.

In 1989, batch cryptography was introduced by Fiat [27] for a variant of RSA. In 1994, an interactive batch verifier for DSA presented in an early version of [52] was broken by Lim and Lee [41]. In 1995 Laih and Yen proposed a new method for batch verification of DSA and RSA signatures [38], but the RSA batch verifier was broken five years later by Boyd and Pavlovski [15]. In 1998, two batch verification techniques were presented for DSA and RSA [29, 30] but both were later broken [15, 35, 36]. The same year, Bellare, Garay and Rabin took the first systematic look at batch verification [9] and presented three generic methods for batching modular exponentiations, one of which is called the *small exponents test*. Unfortunately, in 2000, Boyd and Pavlovski [15] published attacks against various batching schemes which were using the small exponents test

[1] Optionally, one can start with an existing implementation, from which AutoBatch will extract a representation.

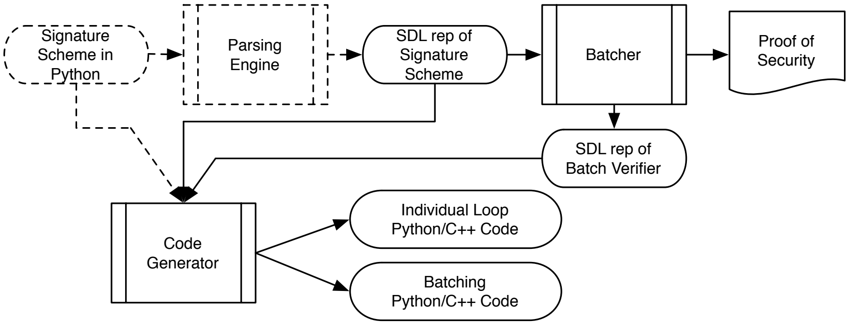

Signature Scheme in Python
Parsing Engine
SDL rep of Signature Scheme
Batcher
Proof of Security
SDL rep of Batch Verifier
Code Generator
Individual Loop Python/C++ Code
Batching Python/C++ Code

Figure 1: The flow of AutoBatch. The input is a signature scheme comprised of key generation, signing and verification algorithms, represented in the domain-specific SDL language. The scheme is processed by a Batcher, which applies the techniques and optimizations from Section 3 to produce a new SDL file containing a *batch verification* algorithm. Optionally, the Batcher outputs a proof of security (as a PDF written in LaTeX) that explains, line by line, each technique applied and its security justification. Finally, the Code Generator produces executable C++ or Python code implementing both the resulting batch verifier, and the original (unbatched) verification algorithm. An optional component, the Parsing Engine, allows for the automatic derivation of SDL inputs based on existing scheme implementations.

incorrectly. In 2003-2004, several batch verification schemes based on bilinear maps (a.k.a., pairings) were proposed [22, 64,66,67] but all were later broken by Cao, Lin and Xue [20]. In 2006, a method was given for identifying invalid signatures in RSA-type batches [40], but it was also flawed [61].

It is natural to ask what the source of the errors were in these papers. In several cases, the mathematics of the scheme were simply unsound and the proof of security was either missing or lacking in rigor. However, there were two other common problems. One was that the paper claimed *in English* to be doing batch verification, but the security definition provided in the paper was insufficient to establish this guarantee. Most commonly this matched the strictly weaker *screening* guarantee; see [17] for more. A second problem was more insidious: the security definition and proof were "correct", but the scheme was still subject to a practical attack because the authors started the proof by explicitly *assuming* that elements of the signature were members of certain algebraic groups and this was not a reasonable assumption to make in practice. Boyd and Pavlovski [15] provide numerous examples of this case.

AutoBatch addresses these common pitfalls. It uses one security definition (in Section 2.1) and provides a proof of security for every algorithm it outputs relative to this definition (in Section 3.2), where no assumptions about the algebraic structure of the input are made and therefore any necessary tests are explicitly performed by the algorithm.

In addition to the batching work above, we mention a few more. Shacham and Boneh presented a modified version of Fiat's batch verifier for RSA to improve the efficiency of SSL handshakes on a busy server [58]. Boneh, Lynn and Shacham provided a single-signer batch verifier for BLS signatures [13]. Camenisch, Hohenberger and Pedersen [17] gave multiple-signer batch verifiers for Waters identity-based signatures [62] and a novel construction. Ferrara, Green, Hohenberger, and Pedersen outlined techniques for batching pairing-based signatures and showed how to batch group and ring signatures [26]. Blazy, Fuchsbauer, Izabachéne, Jambert, Sibert and Vergnaud [10] applied batch verification techniques to the Groth-Sahai zero-knowledge proof system as well as group signatures and anonymous credential systems relying on them, obtaining significant savings.

Law and Matt describe methods for identifying invalid signatures in a batch [39,47,48].

Lastly, there have been several research efforts toward automatically generating cryptographic protocols and executable code. This compiler-like approach has been applied to cryptographic applications such as security protocols [37,42,43,54,60], optimizations to software implementations involving elliptic-curve cryptography [7] and bilinear-map functions [53], secure two-party computation [31,45,46], and zero-knowledge proofs [2,4,5,19,49].

2. BACKGROUND

A *digital signature scheme* is comprised of the usual probabilistic polynomial-time algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$. We recall the basics of signatures, as well as the identity-based, privacy-preserving and verifiable random function variants.

Definition 2.1 (A Digital Signature). *A* digital signature scheme *is a tuple of probabilistic polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *as:*

1. $\mathsf{Gen}(1^\lambda) \rightarrow (pk, sk)$*: the key generation algorithm takes as input the security parameter* 1^λ *and outputs a pair of keys* (pk, sk)*.*

2. $\mathsf{Sign}(sk, m) \rightarrow \sigma$*: the signing algorithm takes as input a secret key* sk *and a message* m *from the message space and outputs a signature* σ*.*

3. $\mathsf{Verify}(pk, m, \sigma) \rightarrow \{0, 1\}$*: the verification algorithm takes as input a public key* pk*, a message* m *and a purported signature* σ*, and outputs a bit indicating the validity of the signature.*

A scheme is correct *if for all* $\mathsf{Gen}(1^\ell) \rightarrow (pk, sk)$*, the algorithm* $\mathsf{Verify}(pk, m, \mathsf{Sign}(sk, m)) = 1$*.*

A scheme is defined to be *unforgeable* as follows [28]: Let $\mathsf{Gen}(1^\ell) \rightarrow (pk, sk)$. Suppose (m, σ) is output by a p.p.t. adversary with access to a signing oracle $\mathcal{O}_{sk}(\cdot)$ and input

pk. Then the probability that m was *not* queried to $\mathcal{O}_{sk}(\cdot)$ and yet $\mathsf{Verify}(pk, m, \sigma) = 1$ is negligible in ℓ.

In this work, we explore three variants:

1. **Identity-Based Signatures [59]**: Gen is executed by a master authority who publishes pk and uses sk to generate signing keys for users according to their public identity string, e.g., email address. To verify a signature on a given message, one only needs the pk of the master authority and the public identity string of the purported signer.

2. **Privacy Signatures:** Group [24] and ring [55] signatures are associated with a group of users, where verification shows that at least one member of the group signed the message, but it is difficult to tell whom.

3. **Verifiable Random Functions [50]:** A VRF is a pseudo-random function, where the computing party publishes a public key pk and then can offer a short non-interactive *proof* that the function was correctly evaluated for a given input. This proof can be viewed as a signature by the computing party on the input to the pseudo-random function.

2.1 Batch Verification

Our security focus here is not directly on unforgeability [28]. Rather we are interested in designing batch verification algorithms that accept a set of signatures *if and only if* each signature would have been accepted by its verification algorithm individually. *Thus, if a scheme is unforgeable, then our batching algorithm will preserve this property.*

Specifically, we consider the case where we want to quickly verify a set of signatures on possibly different messages by possibly different signers. The input is $\{(t_1, m_1, \sigma_1), \ldots, (t_n, m_n, \sigma_n)\}$, where t_i specifies the verification key against which σ_i is purported to be a signature on message m_i. It is important to understand that here one or more *signers* may be maliciously colluding against the batch verifier.

We recall the definition of Bellare, Garay and Rabin [9] as extended in [17] to deal with multiple signers.

Definition 2.2 (Batch Verification of Signatures). *Let ℓ be the security parameter. Suppose* (Gen, Sign, Verify) *is a signature scheme, $k, n \in poly(\ell)$, and $(pk_1, sk_1), \ldots, (pk_k, sk_k)$ are generated independently according to* $\mathsf{Gen}(1^\ell)$. *Let $PK = \{pk_1, \ldots, pk_k\}$. We call probabilistic* Batch *a batch verification algorithm when the following conditions hold:*

- *If $pk_{t_i} \in PK$ and $\mathsf{Verify}(pk_{t_i}, m_i, \sigma_i) = 1$ for all $i \in [1, n]$, then $\mathsf{Batch}((pk_{t_1}, m_1, \sigma_1), \ldots, (pk_{t_n}, m_n, \sigma_n)) = 1$.*
- *If $pk_{t_i} \in PK$ for all $i \in [1, n]$ and $\mathsf{Verify}(pk_{t_j}, m_j, \sigma_j) = 0$ for some $j \in [1, n]$, then $\mathsf{Batch}((pk_{t_1}, m_1, \sigma_1), \ldots, (pk_{t_n}, m_n, \sigma_n)) = 0$ except with probability negligible in ℓ, taken over the randomness of* Batch.

The above definition can be generalized beyond signatures to apply to any keyed scheme with a verification algorithm. This includes zero-knowledge proofs, verifiable random functions, and variants of regular signatures, such as identity-based, attribute-based, ring, group, aggregate, etc. The above definition requires that signing keys be generated honestly. In practice, users could register their keys and prove some necessary properties of the keys at registration time [6].

2.2 Algebraic Setting

Bilinear (a.k.a., Pairing) Groups. Let BSetup be an algorithm that, on input the security parameter 1^ℓ, outputs the parameters for a bilinear map (also called a pairing) as $(q, g, h, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, where $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ are groups of prime order $q \in \Theta(2^\ell)$. The efficient mapping $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is both: (*bilinear*) for all $g \in \mathbb{G}_1$, $h \in \mathbb{G}_2$ and $a, b \leftarrow \mathbb{Z}_q$, $e(g^a, h^b) = e(g, h)^{ab}$; and (*non-degenerate*) if g generates $\mathbb{G}_1$ and h generates $\mathbb{G}_2$, then $e(g, h) \neq 1$. See [17] or Section 4 for more. The above bilinear map is called *asymmetric* and our implementations use this highly efficient setting. We optionally could consider *symmetric* maps where $\mathbb{G}_1 = \mathbb{G}_2$.

Testing Membership in Bilinear Groups. When batching, it is critical to test that the elements of each signature are members of the appropriate algebraic group. Boyd and Pavlovski [15] demonstrated efficient attacks on batching algorithms for DSA signature verification which omitted a subgroup membership test.

In this paper, we must test membership in bilinear groups. We require that elements of purported signatures are members of $\mathbb{G}_1$ and *not*, say, members of $E(\mathbb{F}_p)\backslash\mathbb{G}_1$. Determining whether some data represents a point on a curve is easy. The question is whether it is in the correct subgroup. If the order of $\mathbb{G}_1$ is a prime q, one option is to verify that an element y is in $\mathbb{G}_1$ by checking that $y^q \mod q = 1$ [17]. Although this costs an extra modular exponentiation per group element, this will largely be dwarfed by the savings from reducing the total pairings, as experimentally verified first by Ferrara et al. [26] and confirmed by our tests.

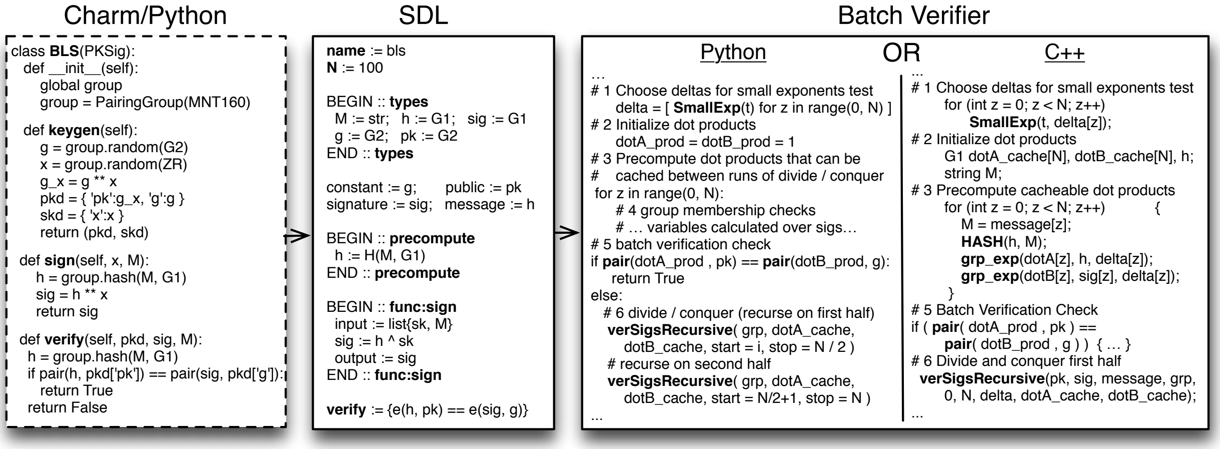
3. THE AUTOBATCH TOOLCHAIN

In this section we summarize the techniques used by AutoBatch to programmatically generate batch verifiers from standard signature schemes. A high level abstraction was provided in Figure 1. The main stages are as follows.

1. Derive the scheme's SDL representation. The AutoBatch toolchain begins with an SDL representation of a signature scheme. While SDL is not a full programming language, it provides sufficient flexibility to represent most pairing-based signature schemes. A detailed description of SDL and some examples can be found in the full version of this work. For developers who already have an existing Charm/Python implementation, we also provide a Parsing Engine that can optionally *derive* an SDL representation directly from this Python code.[2]

2. Apply batching techniques and optimize the verification equation. We first apply a set of techniques designed to convert the SDL signature verification equation into a batch verifier. These techniques optimize the verification equation by combining pairing equations and rearranging the components to minimize the number of expensive operations. To prevent known attacks, we apply the small exponents test of Bellare, Garay and Rabin [9], and

[2] We developed this capability for two reasons. First, there is already a library of pairing-based signatures available in Charm/Python (in fact, the number of Charm implementations is greater than all other settings combined). Secondly, we believe that there is value in providing multiple interfaces to our tools, particularly interfaces that work with real implementations.



```
class BLS(PKSig):
  def __init__(self):
     global group
     group = PairingGroup(MNT160)

  def keygen(self):
     g = group.random(G2)
     x = group.random(ZR)
     g_x = g ** x
     pkd = { 'pk':g_x, 'g':g }
     skd = { 'x':x }
     return (pkd, skd)

 def sign(self, x, M):
    h = group.hash(M, G1)
    sig = h ** x
    return sig

 def verify(self, pkd, sig, M):
   h = group.hash(M, G1)
   if pair(h, pkd['pk']) == pair(sig, pkd['g']):
     return True
   return False
```

```
name := bls
N := 100

BEGIN :: types
 M := str;   h := G1;   sig := G1
 g := G2;   pk := G2
END :: types

constant := g;      public := pk
signature := sig;   message := h

BEGIN :: precompute
 h := H(M, G1)
END :: precompute

BEGIN :: func:sign
 input := list{sk, M}
 sig := h ^ sk
 output := sig
END :: func:sign

verify := {e(h, pk) == e(sig, g)}
```

```
…
# 1 Choose deltas for small exponents test
     delta = [ SmallExp(t) for z in range(0, N) ]
# 2 Initialize dot products
     dotA_prod = dotB_prod = 1
# 3 Precompute dot products that can be
#    cached between runs of divide / conquer
 for z in range(0, N):
     # 4 group membership checks
     # … variables calculated over sigs…
# 5 batch verification check
if pair(dotA_prod , pk) == pair(dotB_prod, g):
    return True
else:
   # 6 divide / conquer (recurse on first half)
   verSigsRecursive( grp, dotA_cache,
      dotB_cache, start = i, stop = N / 2 )
   # recurse on second half
   verSigsRecursive( grp, dotA_cache,
      dotB_cache, start = N/2+1, stop = N )
...
```

```
...
# 1 Choose deltas for small exponents test
       for (int z = 0; z < N; z++)
            SmallExp(t, delta[z]);
# 2 Initialize dot products
       G1 dotA_cache[N], dotB_cache[N], h;
       string M;
# 3 Precompute cacheable dot products
       for (int z = 0; z < N; z++)          {
          M = message[z];
          HASH(h, M);
          grp_exp(dotA[z], h, delta[z]);
          grp_exp(dotB[z], sig[z], delta[z]);
        }
# 5 Batch Verification Check
if ( pair( dotA_prod , pk ) ==
       pair( dotB_prod , g ) )  { … }
# 6 Divide and conquer first half
 verSigsRecursive(pk, sig, message, grp,
       0, N, delta, dotA_cache, dotB_cache);
...
```

Figure 2: The Boneh-Lynn-Shacham (BLS) signature scheme [13] at various stages in the AutoBatch toolchain. At the left, an initial Charm-Python implementation of the scheme. In the center, an SDL representation of the same scheme, programmatically extracted by the Parsing Engine. At right, a fragment of the resulting batch verifier generated after applying the Batcher and Code Generator.

optimize the resulting equation to ensure that all signature elements are in the group with the smallest representation (typically, $\mathbb{G}_1$). The output of this phase is a modified SDL file, and (optionally) a human-readable proof that the resulting equation is a batch verifier.

3. Evaluate the capabilities of the batch verifier. Given the optimized batching equation produced in the previous step, we estimate the performance of the verifier under various conditions. This is done by counting the operations in the verifier, and deriving a runtime estimate based on the expected cost of each mathematical operation (e.g., pairing, exponentiation, multiplication). The cost of each operation is determined via a set of diagnostic tests conducted when the library is initialized.[3]

4. Generate code for the resulting batch verifier. Finally, we translate the resulting SDL file into a working batch verifier. This verifier can be implemented in either C++ (using the MIRACL library [56] for curve operations) or in Python based on the Charm framework. It implements the SDL-specified batch verification equation as well as the individual verification equation. Based on the calculations of the previous step, the generated code embeds logic to automatically determine *which* verifier is most appropriate for a given dataset (individual or batch). Additionally, the generated code embeds a recursive *divide-and-conquer* strategy to handle cases where batch verification fails due to invalid signatures. Two fragments of generated code (Python and C++) are shown in Figure 2.

We will now describe each of the above steps in detail.

3.1 Batching and Optimization

Given an SDL file containing the verification equation and variable types, the Batcher applies a series of optimizations to the verification equation in order to derive an efficient batch verifier. Many of these techniques were first explored in previous works [17, 26]. However, the intended audience of those works is *humans* performing manual batching of signatures. Hence, they are in many cases somewhat less 'general' than the techniques we describe here.[4] Furthermore, unlike previous works we are able to programmatically identify when these techniques are applicable, and apply them to the verification equation in a consistent way.

The Batcher assumes that the input will be a collection of η signatures, possibly on different messages and public keys (or identities). To construct a batch verifier, Batcher first validates the type information in SDL input file and converts the verification equation into a tree structure. During this phase, it informs users if there are type mismatches or if the typing information is incomplete in SDL. Next, the batcher traverses the tree, applying various operations at various nodes in the verification equation. We now list those techniques.

Technique 0: Consolidate the verification equation. Many pairing-based signature schemes actually require the verifier to check more than one pairing equation. During the first phase of the batching process, the batcher applies the small exponents test from [9] to combine these equations into a single verification equation.[5]

Technique 1: Combine equations. Assume we are given η signature instances that can be verified using the consolidated equation from the previous step. We now combine all instances into one equation by applying the Combination Step of [26], which employs as a subroutine the small exponents test. This results in a single verification equation. The correctness of the resulting equation requires that all

[3] Obviously these experiments are very specific to the machine and curve parameters on which they are run. Our implementation re-runs these experiments whenever the library is initialized with a given set of parameters.

[4] For example: techniques 2 and 3 of [17] each combine a series of logical operations that are more widely applicable and easily managed by splitting them into more granular sub-techniques.

[5] For example, consider two verification conditions $e(a, b) = e(c, d)$ and $e(a, c) = e(g, h)$. These can be verified simultaneously by selecting random δ_1, δ_2 and evaluating the single equation $e(a, b)^{\delta_1} e(c, d)^{-\delta_1} e(a, c)^{\delta_2} e(g, h)^{-\delta_2} = 1$.

elements be in the correct subgroup, i.e., that group membership has already been checked. AutoBatch ensures that this check will be explicitly conducted in the final batch verifier program.

Technique 2: Move exponents into the pairing. When a pairing of the form $e(g_i, h_i)^{\delta_i}$ appears, move the exponent δ_i into $e()$. Since elements of $\mathbb{G}_1$ and $\mathbb{G}_2$ are usually smaller than elements of $\mathbb{G}_T$, this gives a noticeable speedup when computing the exponentiation.

$$\text{Replace } e(g_i, h_i)^{\delta_i} \text{ with } e(g_i^{\delta_i}, h_i)$$

Wherever possible, we move the exponent into the group with the lowest exponentiation cost. We identify this group based on a series of operation microbenchmarks that run automatically at code initialization.[6]

Technique 3: Move dot products into the pairing. When a pairing of the form $\prod_{i=1}^{\eta} e(a_i, g)$ with a constant first or second element appears, move the dot product inside to reduce the number of pairings from η to 1.

$$\text{Replace } \prod_{i=1}^{\eta} e(a_i, g) \text{ with } e(\prod_{i=1}^{\eta} a_i, g)$$

Technique 4: Optimize the Waters Hash. A variety of bilinear signature schemes employ a hash function by Waters [62], which can be generalized [23, 51]. Assume the identity is a k-bit string $V = v_1 v_2 \ldots v_z$ where each v_i is a short string. The hash function is evaluated as $u' \prod_{i=1}^{m} u_i^{v_i}$.

When batching η equations containing the Waters hash, one often encounters terms of the form $\prod_{j=1}^{\eta} e(g_j, \prod_{i=1}^{z} u_i^{v_{ij}})$. This can be rewritten to make the number of pairings independent of the number of equations one wants to batch.

$$\text{Replace } \prod_{j=1}^{\eta} e(g_j, \prod_{i=1}^{z} u_i^{v_{ij}}) \text{ with } \prod_{i=1}^{z} e(\prod_{j=1}^{\eta} g_j{}^{v_{ij}}, u_i)$$

Technique 5: Distribute dot products. When a dot product is applied to two or more pairings, distribute the dot product to each pairing to allow application of other techniques such as techniques 3 or 4. For example:

$$\text{Replace } \prod_{i=1}^{\eta} (e(a_i, g_i) \cdot e(b_i, h_i)) \text{ with } \prod_{i=1}^{\eta} e(a_i, g_i) \cdot \prod_{i=1}^{\eta} e(b_i, h_i)$$

Technique 6: Combine pairings with common elements. When two or more pairings share a common first or second element, they can be combined. For example:

$$\text{Replace } e(a, g) \cdot e(b, g) \text{ with } e(ab, g)$$

Technique 7: Move known exponents outside pairing and precompute pairings. In some cases it may be necessary to move exponents outside of a pairing. For example, when $\prod_{i=1}^{\eta} e(g^{a_i}, h^{b_i})$ appears, move the exponents outside of pairing. When multiple such exponents appear, we can precompute the sum of $a_i \cdot b_i$ for all η and exponentiate once in $\mathbb{G}_T$.

$$\text{Replace } \prod_{i=1}^{\eta} e(g^{a_i}, h^{b_i}) \text{ with } e(g, h)^{\sum_i (a_i \cdot b_i)}$$

Technique 8: Precompute constant pairings. When pairings have a constant first and second element, we can simply remove these from the equation and pre-compute them once at the beginning of verification (equivalent to making them a public parameter). We refer to this as Technique 8.

Technique 9: Split pairings. In some rare cases it can be useful to apply Technique 3 in reverse: splitting a single pairing into two or more pairings. This temporarily increases the number of pairings in the verification equation, but may be necessary in order to apply subsequent techniques. For example, this optimization is necessary so that we can apply the Waters hash optimization (Technique 4) to the ring signature of Boyen [16].

Discussion: Several of the above techniques are quite simple, in that they perform optimizations that would seem "obvious" to an experienced cryptographer. However, many optimizations (*e.g.,* Technique 8) *could* have been applied in published algorithm descriptions, and yet were not. Moreover, it is a computer and not a human that is performing the search for us, so an important contribution of this work is providing a detailed list of which optimizations we tell the computer to try out and in which order, and verifying that such an approach can find competitive solutions in a reasonable amount of time. This is nontrivial: we discovered that many orderings lead to "dead ends", where the optimal solution is *not* discovered. We now describe our approach to finding the order of techniques.

Technique Search Algorithm: The challenge in automating the batching process is to identify the *order* in which techniques should be applied to a given verifier. This is surprisingly difficult, as there are many possible orderings, many of which require several (possibly repeated) invocations of specific techniques. Due to space considerations, we leave a complete discussion to the full version, and provide only a brief description of our approach.

The naive approach to this problem is simply to try all possible combinations up to a certain limit, then identify the best resulting verifier based on an estimate of total running time. Although this approach is feasible for simple schemes, it is quite inefficient for schemes that require the application of several techniques. Moreover, there is the separate difficulty of determining when the algorithm should halt, as the application of one technique will sometimes produce a new equation that is amenable to further optimization, and this process can continue for several operations.

Our approach is to "prune the tree" by utilizing a finite state transition function that constrains the transitions between techniques. This function examines the history of techniques already applied and determines which techniques can be applied to the current state. Our search algorithm begins with techniques 0, 1 or 2, then employs a breadth-first search to identify multiple candidate paths that form a batch verifier.

To prevent infinite loops in the search, the state function

[6] For many common elliptic curves, this is the $\mathbb{G}_1$ base group. However, in some curves the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ have similar operation costs; this may give us some flexibility in modifying the equation.

We begin with the original verification equation.

$$e(Y,a) \overset{?}{=} e(g,b) \text{ and } e(X,a) \cdot e(X,b)^m \overset{?}{=} e(g,c)$$

Step 1: Consolidate the verification equations (tech. 0), merge pairings with common first or second element (tech. 6), and apply the small exponents test, using exponents $\delta_1, \dots \delta_\eta \in \left[1, 2^\lambda\right]$ for each equation:

$$e(g, b \cdot c^{-1})^{\delta_{1,2}} \cdot e(Y,a)^{-\delta_1} \overset{?}{=} e(X,a)^{\delta_2} \cdot e(X,b)^{m \cdot \delta_2}$$

Step 2: Combine η signatures (tech. 1), move the exponent(s) in pairing (tech. 2):

$$\prod_{z=1}^{\eta} e(g, (b_z \cdot {c_z}^{-1})^{\delta_{1,2,z}}) \cdot e(Y, {a_z}^{-\delta_{1,z}}) \overset{?}{=} \prod_{z=1}^{\eta} e(X, {a_z}^{\delta_{2,z}}) \cdot e(X, {b_z}^{m_z \cdot \delta_{2,z}})$$

Step 3: Merge pairings with common first or second element (tech. 6):

$$\prod_{z=1}^{\eta} e(g, (b_z \cdot {c_z}^{-1})^{\delta_{1,2,z}}) \cdot e(Y, {a_z}^{-\delta_{1,z}}) \overset{?}{=} \prod_{z=1}^{\eta} e(X, {a_z}^{\delta_{2,z}} \cdot {b_z}^{m_z \cdot \delta_{2,z}})$$

Step 4: Move dot products inside pairings to reduce from η to 1 (tech. 3):

$$\prod_{z=1}^{\eta} e(g, (b_z \cdot {c_z}^{-1})^{\delta_{1,2,z}}) \cdot e(Y, {a_z}^{-\delta_{1,z}}) \overset{?}{=} e(X, \prod_{z=1}^{\eta} {a_z}^{\delta_{2,z}} \cdot {b_z}^{m_z \cdot \delta_{2,z}})$$

Step 5: Distribute dot products (tech. 5): :

$$\prod_{z=1}^{\eta} e(g, (b_z \cdot {c_z}^{-1})^{\delta_{1,2,z}}) \cdot \prod_{z=1}^{\eta} e(Y, {a_z}^{-\delta_{1,z}}) \overset{?}{=} e(X, \prod_{z=1}^{\eta} {a_z}^{\delta_{2,z}} \cdot {b_z}^{m_z \cdot \delta_{2,z}})$$

Step 6: Move dot products inside pairings to reduce from η to 1 (tech. 3):

$$e(g, \prod_{z=1}^{\eta} (b_z \cdot {c_z}^{-1})^{\delta_{1,2,z}}) \cdot e(Y, \prod_{z=1}^{\eta} {a_z}^{-\delta_{1,z}}) \overset{?}{=} e(X, \prod_{z=1}^{\eta} {a_z}^{\delta_{2,z}} \cdot {b_z}^{m_z \cdot \delta_{2,z}})$$

Figure 3: A fragment of the machine-generated security proof of a single-signer batch verifier for the Camenisch-Lysyanskaya (CL) signature scheme [18]. An earlier portion of the proof asserted that a group membership test would be done prior to checking the final equation.

disallows the application of certain techniques that might potentially *undo* optimizations. For example, Technique 9 performs a reverse split on pairings to allow further optimizations; this might affect technique 6, which combines pairings that have common elements. Certain combinations of techniques 9 and 6 lead to an infinite cycle that combines and splits the same pairings. Thus, the state function only allows a transition from Technique 6 to 9 to occur once on a given path.

The search algorithm terminates when none of the techniques can be applied to the current state. It then selects the path from the candidate paths that provides the highest cost savings. While our approach does not guarantee the optimal batch equation, in practice we rediscover all existing lower bounds on batch verification performance, and in some cases we improve on results developed by humans.

3.2 Security and Machine-Aided Analysis

Efficiency Analysis. Once the Batcher has produced a final equation for the batch verifier, it counts the number of operations required as a function of the batch size. These operations include point operations, pairings, hashes, as well as random element generation. It then combines this operation count with a database of average operation times that were measured at library initialization. The resulting calculation allows it to determine the "crossover point", i.e., the batch size where batch verification becomes more efficient than individual verification.

Security Analysis. We have two points to make regarding the security of AutoBatch. First, we argue that the algorithm used by AutoBatch to produce a batch verification equation *unconditionally* satisfies Security Definition 2.2. That is, the batch verification equation will hold if and only if each of the individual signatures would have passed the individual verification test (up to a negligible error probability).[7]

THEOREM 3.1 (SECURITY OF AUTOBATCH). *Let an AutoBatch algorithm be generalized as any algorithm that transforms an individual pairing-based signature verification test into a pairing-based batch verification equation as follows:*

1. *Check the group membership of all input elements, and if no errors, apply Techniques 0 and 1 to the individual verification equation(s) using security parameter λ to obtain a single equation X.*

2. *Loops until done on: Apply any of Techniques 2-9 to X to obtain equation X' and set $X := X'$.*

Then all AutoBatch algorithms unconditionally satisfy Definition 2.2, where the probability of accepting an invalid batch is at most $2^{-\lambda}$.

Proof. We analyze this proof in two parts. First, after Step 1 (the application of Techniques 0 and 1), there will be one batch equation X and it will satisfy the security requirements of Definition 2.2 with error probability $2^{-\lambda}$. These two techniques combine a set of equations into a single equation using the Small Exponents Test with security parameter λ. Ferrara et al. [26, Theorem 3.2] prove that this equation will verify if and only if all individual equations verify, except with probability at most $2^{-\lambda}$. By default in AutoBatch, we set $\lambda = 80$.

Next, given a single arbitrary, pairing-based equation X, we apply one of Techniques 2-9. For each Technique 2-9, we argue that the output equation X' holds if and only if the input equation X holds; that is, the equations are identical up to algebraic manipulations. If this is true, the final batch equation output by AutoBatch satisfies Definition 2.2 with the same error probability as the equation output after Techniques 0 and 1 were applied, completing the theorem.

It remains to argue that for each Technique 2-9, it is indeed the case that the input and output equations are identical, up to algebraic manipulations. Techniques 2, 3, 4, 6, 7 and 9 follow relatively straightforwardly from the bilinearity of the groups. As an example, consider Technique 6 which claims that $e(a,g) \cdot e(b,g) = e(ab,g)$, for all $a, b \in \mathbb{G}_1$ and $g \in G_2$. Let $b = a^k$ for some $k \in \mathbb{Z}_p$. Then we have $e(a,g) \cdot e(a^k, g)$ as the LHS, which is $e(a,g) \cdot e(a,g)^k$ by the bilinearity, which is $e(a,g)^{k+1}$ by multiplication in $\mathbb{G}_T$. The RHS is similarly $e(aa^k, g) = e(a^{k+1}, g) = e(a,g)^{k+1}$. Technique 5 requires only associativity in $\mathbb{G}_T$. Technique 8 pre-computes and caches some values instead of repeatedly computing them on the fly. □

[7] The security of the underlying signature scheme depends on a computational assumption, but the batcher unconditionally maintains whatever security is offered by the scheme.

Process	BLS	CHP	CL	HW-diff	Waters09	Waters05	ChCh/Hess	CYH	Boyen	BBS	VRF
Batcher	42.0	70.1	127.4	122.7	529.4	1169.1	186.9	76.7	230.3	302.0	209.4
Codegen	124.3	171.7	152.2	242.3	361.6	291.2	162.0	242.8	321.2	315.1	251.2
Total	166.3	241.8	279.6	365.0	891.0	1460.3	348.9	319.5	551.5	617.1	460.6

Figure 4: Time in milliseconds required by the Batcher and Code Generator to process a variety of signature schemes (averaged over 100 test runs). Batcher time includes search time for the technique ordering, generating the proof and estimating cross over point between individual and batch verification. The running times are a product of the complexity of each scheme as well as the number of unique paths uncovered by our search algorithm. In all cases, the standard deviation in the results were within ±3% of the average.

To offer transparency on how AutoBatch derived any given batch verifier, Batcher produces both an SDL file and, optionally, a human-readable proof of security for the resulting batch verifier. This proof is a LaTeX file that includes the individual and batch verification equations, with an enumeration of the various steps used to convert the former into the latter. Thus, while *Theorem 3.1 already argues that this proof is valid*, this provides a means for independently verifying the security of any given batching equation. Interestingly, the first proof for the batch verification of the HW signatures [33] was produced automatically by AutoBatch.

We show a fragment of this human-readable proof of security for the Boneh-Lynn-Shacham (BLS) signature scheme [13] in Figure 3. Human-readable proofs of security for some of the other schemes against which we evaluated AutoBatch are given in the full version of this work.

The security analysis provided in this section applies to the mathematics only. AutoBatch goes on to convert this mathematical batching equation into code, which could potentially introduce *software* errors. However, our hope is that the deliberate process by which AutoBatch generates code would actually help reduce software errors by systematically including steps, such as the group membership test, which could easily be accidentally omitted by a human implementor.

3.3 Code Generation

The output of the Batcher is a batch verification equation encoded in SDL. This file defines all of the datatypes for the signature, message and public key (or identity and public parameters in the case of an identity-based signature). The Code Generator converts this SDL representation into useable Python or C++ source code that can operate on real batch inputs.

The Code Generator translates the individual *and* batch verification equations into C++ or Python code, and wraps them with the following additional logic components:

1. **Group membership tests.** For each element in the signature (and optionally the public key, if the user requests)[8] the membership of the group is tested using an exponentiation. Section 2.2 discusses the importance and details of this test.

2. **Pre-computation.** Several values often will be reused within a verification equation. When this happens, the batch verifier can *pre-compute* certain results once, rather than needlessly compute them several times.

3. **Technique selection.** For relatively small batch sizes, it may be *more* efficient to bypass the batch verifier and simply verify the signatures using the individual verification function. For this reason, our Code Generator generates this function as well (the output of the Batcher contains both functions), and adds logic to programmatically choose between batch and individual verification when the batch size is below a certain threshold automatically determined in the Analysis phase.

4. **Invalid signature detection.** To handle the presence of invalid signatures in a batch, our batch verifier code includes a recursive *divide-and-conquer* strategy to recover from a batching failure (see e.g,. [26] for a discussion of this). On failure, this verifier divides the signature collection into two halves and recurses by repeating verification on each half until all of the invalid signatures have been identified.

The Code Generator consists of two "back-end" modules, which produce Charm/Python and C++/MIRACL representations of the batch verifiers. It would be relatively easy to extend this module to add support for additional languages and settings.

3.4 Code Parsing

While SDL is the primary input language for our batcher, we also support batching from a pre-existing implementation of a signature scheme. To facilitate this, we provide a Code Parsing engine that interprets signature schemes written in a high level language, derives their verification equation and data types, and produces a resulting SDL file. While our techniques should work with various languages (provided that the signature implementation is somewhat constrained), our prototype implementation is based on Charm/Python. This means we can take advantage of a relatively large library of pre-existing Charm implementations. Additionally, in this setting we are assisted by the Python interpreter, which grants programatic access to the Python Abstract Syntax Tree via the `compiler.ast` module.

While Charm implementations are relatively constrained in terms of their structure, a challenging aspect of code parsing is identifying the type of each variable. We stress that this problem is not unique to Python: indeed, many standard libraries (such as the the C-based Stanford Pairing-Based Crypto library [44]) employ abstract data types to represent group elements. Interpreting code written using these languages will also require techniques similar to the ones we use.

Code parsing consists of the following stages. First, we parse the entire signature scheme file to identify the AST node of the signature `verify()` method, and then identify the

[8] In many applications we can assume that the public keys are trusted, thus we can omit group membership testing on these values.

equality comparisons in this function that are fundamentally responsible for the signature verification process. We next build a map of variable names, types, structure, and operations. For each assignment, we check the properties of that assignment using a further set of heuristics. If we determine that a given assignment is relevant, we extract certain information about it, such as the *type* of the variables. We obtain this information by applying known rules to infer type. For example, we know that certain hash calls indicate an element of $\mathbb{G}_1$, a pairing indicates an element in $\mathbb{G}_T$, random element generation calls typically indicate the type of element being generated, and so on.[9]

4. IMPLEMENTATION & PERFORMANCE

4.1 Experimental Setup

To evaluate the performance of our techniques we implemented them as part of the Charm prototyping framework [1]. Charm is a Python-based cryptographic prototyping framework, and provides native support for bilinear-map based cryptography and other useful primitives, *e.g.,* hashing and serialization. We used a version of Charm that implements all bilinear group operations using the C-based MIRACL library [56].[10] The necessary MIRACL calls are accessed from within our Python code via the C module interface.

To determine the performance of our system in isolation, we first conducted a number of experiments on various components of our code. First, we used the code parsing component to convert several Python signature implementations into our intermediate "SDL" representation. Next, we applied our batcher to the SDL result in order to obtain an optimized equation for a *batch verifier.* We then applied our code generator to convert this representation into a functioning batch verifier program, which we applied to various test data sets.

Hardware configuration. For consistent results we ran all of our experiments on a single hardware platform: a 2 x 2.66 GHz 6-Core Intel Xeon Macintosh Pro running MacOS version 10.7.3 with 12GB of RAM. We ran all of our tests within a single thread, and thus used resources from only a single core of the Intel processor. We instantiated all of our cryptographic implementations using a 160-bit MNT elliptic curve provided with MIRACL.

A note on the library. We chose MIRACL because it is mature and well supported. However, some research libraries like RELIC [3] provide alternative pairing implementations that may outperform MIRACL in specific settings. We note that our results will apply to any implementation where there is a substantial difference between group operation and pairing times. In our experiments with RELIC using a provided 256-bit Barreto-Naehrig curve, we observed a 6-to-1 differential between pairings and operations in $\mathbb{G}_1$. This indicates that our main results should hold in this setting, and will in fact improve (in that we can process a higher number of signatures). We will provide details on this alternative implementation in the full version of this work.

4.2 Signature Schemes used as Test Cases

We ran our experiments using two sets of test cases. The first set was comprised of a variety of existing schemes, including regular, identity-based, ring, group signatures and verifiable random functions.

The results are summarized in Figure 5. In all cases, the batching algorithm output by AutoBatch either matched the prior best known result or outperformed it. In particular, AutoBatch realized a batching algorithm for the VRF in [34] that takes only two-thirds the time of the one provided in [34]. Actually, when we double-checked this result by hand, we realized that the verification of equation 2 could be further optimized to only $\ell - 1$ pairings by unrolling a constant-size loop and combining the individual verification equations checked at each iteration. Moreover, a portion of the unrolled loop with the g_2 term could be combined with the corresponding term in the combined equations 1,3,4 for a total pairing count of only $\ell+3$ pairings to batch an arbitrary number of VRF proofs for ℓ-bit inputs. We deem automatic loop unrolling as a future technique that we will incorporate into AutoBatch as it searches for optimal solutions.

Also of note, in test case 10, we simulated a scenario where a batch contains a mix of two different types of signatures. In this case, the batch consisted of both ChCh [22] signatures and Hess [32] signatures in a randomized order. Instead of sorting the signatures into two groups and batching them individually, AutoBatch automatically looked for the common algebraic structure between the two distinct schemes and applied the batching techniques described in Section 3.1. As a generalized example, if two signature schemes both use the same generator g, where the first signature scheme uses $e(A, g)$ in its verification equation and the second signature scheme uses $e(B, g)$ in its verification equation, then AutoBatch will apply Technique 6 to obtain $e(A \cdot B, g)$ in the combined verification equation (as well as apply the small exponents test). In the case of the ChCh [22] and Hess [32] batch, this cut the total number of pairings in half. To the best of our knowledge, this is the first documented result for *cross-scheme* signature batch verification.

For the Hohenberger-Waters signatures [33], we assume that each public key includes the precomputed values as suggested in [33, Section 4.2]. For the case of different signers, we assume that the base group elements g, u, v, d, w, z, h are chosen by a trusted third party and shared by all users. The Waters09 scheme is derived from the Waters Dual-System IBE of [63] using the technique described by Naor [12].

To make AutoBatch as robust as possible, we also tested it on a second set of fabricated pairing-product equations that we designed by hand to trigger many different orderings on the techniques.

4.3 Microbenchmarks

To evaluate the efficiency of AutoBatch, we implemented several pairing-based signature schemes in Charm. We ran AutoBatch to extract an SDL-based intermediate representation of the scheme's verification equation, an optimized batch verifier for the scheme and Python code for implementing the batch verifier. We measured the processing time for each of the above steps. Our timings, averaged over 100 runs, are presented in Figure 4.

[9] We believe that this approach may also be useful in the future for static checking and formal verification of dynamically-typed cryptographic implementations.

[10] The version of Charm we used (0.42) can be found in the Charm github repository at `www.charm-crypto.com`. It uses MIRACL 5.5.4 for bilinear group operations.

Scheme	Type	Model	Ind-Verify	By Hand		By AutoBatch	
				Batch-Verify	Ref	Batch-Verify	Techniques
1. Boyen-Lynn-Shacham (BLS) [14] (same signer)	S	RO	2η	2	[14]	2	1,2,3
2. Camenisch-Hohenberger-Pedersen (CHP) [17] (same period)	S	RO	3η	3	[17]	3	1,2,3,5,3
3. Camenisch-Lysyanskaya (CL) [18] (same signer)	S	P	5η	5η	none	**3**	0,6,1,2,6,3,5,3
4. Hohenberger-Waters (HW) [33] (same signer)	S	P	2η	2η	none	**4**	1,2,3,7
5. Hohenberger-Waters (HW) [33] (diff signers)	S	P	2η	2η	none	**4**	1,2,3,9,5,3
6. Waters09 [63] (same signer)	S	P	9η	9η	none	**13**	1,2,9,5,3,7,6
7. Hess [32]	I	RO	2η	2	[26]	2	1,2,3
8. Cha-Cheon (ChCh) [22]	I	RO	2η	2	[39]	2	1,2,3,2
9. Waters05 [62]	I	P	3η	$z+3$	[17]	$z+3$	1,2,3,9,7,5,3,4,6
10. ChCh [22] and Hess [32] together	M	RO	2η	4	[26, 39]	**2**	0,1,2,3,5,3,2,6
11. Chow-Yiu-Hui (CYH) [25]	IR	RO	2η	2	[26]	2	1,2,3,2
12. Boyen [16] (same ring)	R	P	$\ell\eta+\ell$	$3\ell+1$	[26]	$3\ell+1$	1,2,9,4,6,9,5,3
13. Boneh-Boyen-Shacham (BBS) [11]	G	RO	5η	2	[26]	2	1,2,6,6,5,3
14. VRF eq. 1,3,4 [34] (same signer)	V	P	5η	4	[34]	4	0,6,1,2,3,2
15. VRF eq. 2 [34] (same signer)	V	P	$2(\ell-1)\eta$	$3\ell-3$	[34]	$\mathbf{2\ell-2}$	1,2,3,5,3,6

Figure 5: Digital Signature Schemes used as test cases in AutoBatch. For types, S stands for regular signature, I stands for identity-based, M stands for a batch that contains a mix of two different types of signatures, R stands for ring, G stands for group and V stands for verifiable random function. For models, RO stands for random oracle and P stands for plain. Let ℓ be either the size of the ring or the number of bits in the VRF input. Let z be a security parameter that can be set to 5 in practice. To approximate verification performance, we count the total number of pairings needed to process η valid signatures. Unless otherwise noted, the inputs are from different signers. The final column indicates the order of the techniques from Section 3 that AutoBatch recognized as applicable and applied to obtain the resulting batch verifier. The rows in bold are the schemes where AutoBatch discovered new or improved algorithms.

To obtain our microbenchmarks, we ran AutoBatch on several exemplary pairing-based schemes as listed in Figure 5. We then experimented with these schemes at different batch sizes, in order to evaluate their raw performance. The results are presented in Figure 6.

Each graph shows the average per-signature verification time for a batch of η signatures, for η ranging from 1 to 100. We conducted these tests by first generating a collection of η keypairs and random messages,[11] then computing a valid signature over each message. We fed each collection to the batch verifier. ID-based signatures were handled in a similar manner, although we substitute random identities in place of keys. For the Boyen ring signature, we generated a group of three signing keys to construct our ring. In each case, we averaged our results over 100 experimental runs and computed verification time per signature by dividing the total batching time by the number of signatures batched.

4.4 Batch Verification in Practice

Prior works considered the implication of *invalid* signatures in a batch, *e.g.,* [26,39,47,48,65]. Mainly, these works estimated raw signature verification times under various conditions. To evaluate how signature batching might work in real life, we constructed a simulation to determine the resilience of our techniques to various denial of service attacks launched by an adversary.

Basic Model. For this experiment, we simulated a server that verifies incoming signed messages read from a network connection. This might be a reasonable model for a busy server-side TLS endpoint using client authentication or for a vehicle-to-vehicle communications base station.

Our server is designed to process as many signatures as possible, and is limited only by its computational resources.[12] Signatures are drawn off of the "wire" and grouped into batches, with each batch size representing the expected number of signatures that can be verified in one second. Initially this number is simply a guess, which is adjusted upwards or downwards based on the time required to verify each batch.[13] This approach can lead to some transient errors (batches that require significantly more or less than one second to evaluate) when the initial guess is wrong, or when conditions change. In normal usage, however, this approach converges on an appropriate batch size within 1-2 seconds.

4.4.1 Basic DoS Attacks

A major concern when using a batch verifier is the possibility of *service denial* or degradation, resulting from the presence of some invalid signatures in the batch. As described in §3, each of our batch verifiers incorporates a recursive divide-and-conquer strategy for identifying these invalid signatures. This recursion comes at a price; the presence of even a small number of invalid signatures can seriously degrade the performance of a batch verifier.

To measure this, we simulated an adversary who injects invalid signatures into the input stream. Under the assumption that these signatures are well-mixed with the remaining valid signatures,[14] we measured the verifier's throughput. Our adversary injects no invalid signatures for the first several seconds of the experiment, then gradually ramps up its output until the number of invalid signatures received by the verifier approaches 50%.

A switch to individual verification. Our experiments indi-

[11] We used 100-byte random strings for each message. In the case of the stateful HW signature, we batched only signatures with the same counter value.

[12] This models a server that delays, drops or redirects the signatures that it cannot handle (*e.g.,* via load balancing).

[13] The adjustment is handled in a relatively naive way: the server simply computes the next batch size by extrapolating based on its time to compute the previous batch.

[14] In practice, this is not a strong assumption, as a server can simply randomize the order of the signatures it receives.

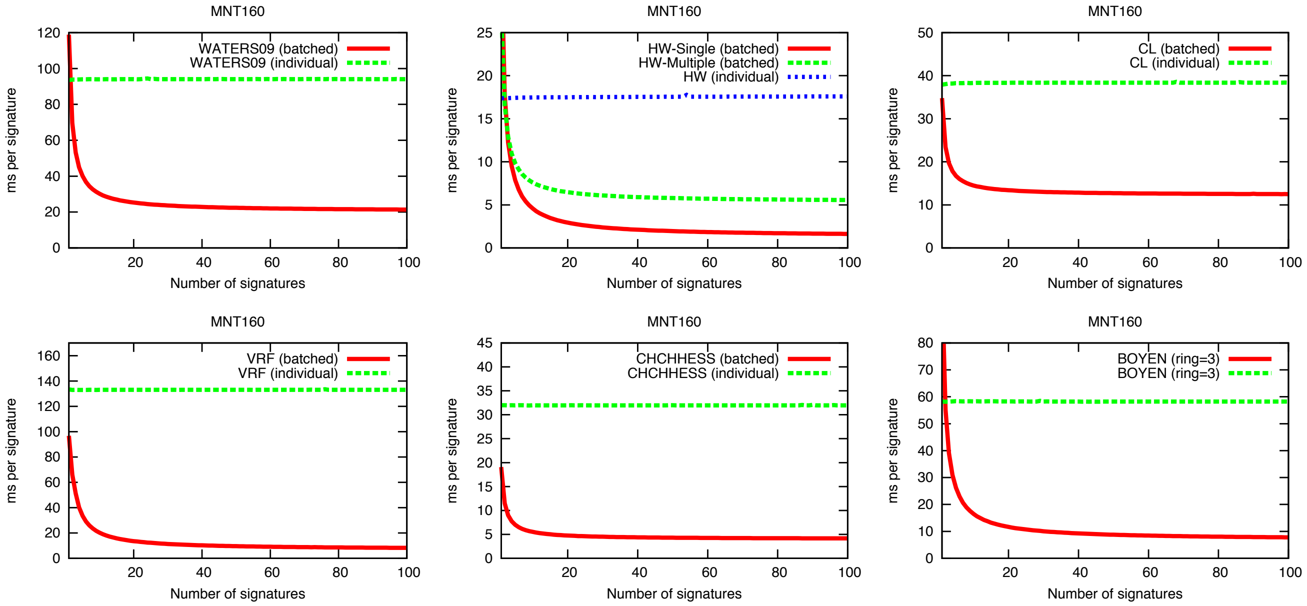


Figure 6: Signature scheme microbenchmarks for Waters09 [63], HW [33] and CL [18] public-key signatures (same signer), the VRF [34] (with block size of 8), combined verification of ChCh+Hess IBS [22,32], and Boyen ring signature (3 signer ring) [16]. Per-signature times were computed by dividing total batch verification time by the number of signatures verified. All trials were conducted with 100 iterations. Variation in running time between trials of the same signature size were minimal for each scheme. Note that in one HW case, all signatures are formulated by the same signer (as for certificate generation). All other schemes are without such restrictions. Individual verification times are included for comparison.

cate that batch verification performance exceeds that of individual verification even in the presence of a relatively large fraction of invalid signatures. However, at a certain point the batch verifier inevitably begins to underperform individual verification.[15] To address this, we implemented a "countermeasure" in our batch verifier to automatically switch to individual verification whenever it detects the presence of a significant fraction of invalid signatures.

Analysis of results. We tested the batch verifier on the single-signer BLS scheme with and without the individual-verification countermeasure. See Figure 7. Throughput is quite sensitive to even small numbers of invalid signatures in the input stream. Yet, when comparing batch verification to *individual* verification throughput, *even under a significant attack* batch verification dramatically outperforms individual verification (up to approximately 15% ratio of invalid signatures). Similarly, the switch to individual verification is a useful countermeasure for attacks that exceed approximately 20% invalid signatures. While these threshold switches do not thwart DoS attacks, they do provide some mitigation of the potential damage.

5. CONCLUSION

The batch verification of pairing-based signatures is a great fit for applications where short signatures are a design requirement and yet high verification throughput is required, such as car-to-car communications [21,57]. This work demonstrates for the first time that the design of these batching algorithms can be efficiently and securely automated.

The next step is to tackle the automated design of more complex functionalities, where it may be infeasible to replicate a theorem like Theorem 3.1 arguing that automated design process unconditionally preserves security. In this case, one might instead focus on having the design tool also output a proof sketch that could be fed into and verified by EasyCrypt [8] or a similar proof checking tool. Indeed, what are the natural settings where the creativity of the design process can be feasibly replaced by an extensive computerized search (perhaps with smart pruning)? Can the "proof sketches" needed for verification by EasyCrypt be generated automatically for these designs? These are exciting questions which could fundamentally change cryptography.

On the implementation of AutoBatch, future work could be more resilient to DoS and related attacks by implementing alternative techniques for recognizing invalid signatures in a batch, e.g., [39, 47, 48, 65]. We are continuously on the lookout for more efficient means of computing in bilinear groups. Future versions of AutoBatch will support the RELIC toolkit [3] and MIRACL's API for computing "multipairings" (efficient products of multiple bilinear pairings). It would be interesting to understand how these and future inclusions may impact performance.

Acknowledgments

Joseph A. Akinyele and Matthew W. Pagano are supported in part by NSF CNS-1010928 and HHS 90TR0003/01.

[15] The reason for this is easy to explain: since our batch verifier handles invalid signatures via a divide-and-conquer approach (cutting the signature batch into halves, and recursing on each half), at a certain point the number of "extra" operations exceeds those required for individual verification.

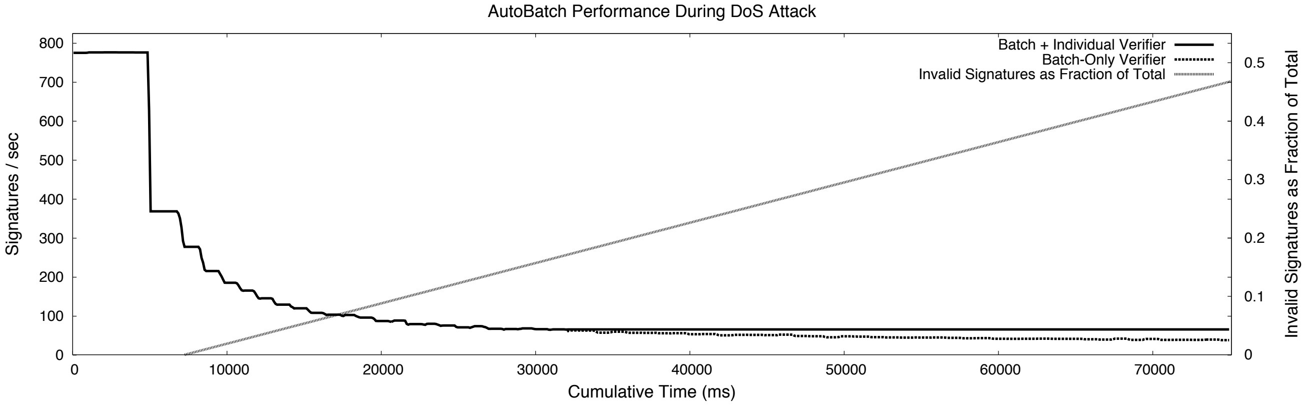


Figure 7: Simulated service denial attacks against a batch verifier (BLS signatures, single signer). The grey line (right scale) shows the fraction of invalid signatures in the stream. Batcher throughput is measured in signatures per second (left scale). The broken line depicts a standard batch verifier. The black line is a batch verifier that automatically switches to *individual* verification when batching becomes suboptimal.

Matthew Green is supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contract FA8750-11-2-0211, the Office of Naval Research contract N00014-11-1-0470, NSF grant CNS-1010928 and HHS 90TR0003/01.

Susan Hohenberger is supported in part by NSF CNS-1154035, the Office of Naval Research under contract N00014-11-1-0470, DARPA and the Air Force Research Laboratory (AFRL) under contract FA8750-11-2-0211, DARPA contract N11AP20006 and a Microsoft Faculty Fellowship.

Applying to all authors, the contents and views expressed are solely those of the authors and do not reflect the official policy, position or views of the Department of Defense, the HHS or the U.S. Government.

6. REFERENCES

[1] Akinyele, J. A., Green, M., and Rubin, A. Charm: A framework for rapidly prototyping cryptosystems. Cryptology ePrint Archive, Report 2011/617, 2011. `http://eprint.iacr.org/`.

[2] Almeida, J. B., Bangerter, E., Barbosa, M., Krenn, S., Sadeghi, A.-R., and Schneider, T. A certifying compiler for zero-knowledge proofs of knowledge based on σ-protocols. In *Proceedings of the 15th European conference on Research in computer security* (Berlin, Heidelberg, 2010), ESORICS'10, Springer-Verlag, pp. 151–167.

[3] Aranha, D. F., and Gouvêa, C. P. L. RELIC is an Efficient LIbrary for Cryptography. `http://code.google.com/p/relic-toolkit/`.

[4] Backes, M., Maffei, M., and Unruh, D. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), SP '08, IEEE Computer Society, pp. 202–215.

[5] Bangerter, E., Briner, T., Henecka, W., Krenn, S., Sadeghi, A.-R., and Schneider, T. Automatic generation of sigma-protocols. In *Proceedings of the 6th European conference on Public key infrastructures, services and applications* (Berlin, Heidelberg, 2010), EuroPKI'09, Springer-Verlag, pp. 67–82.

[6] Barak, B., Canetti, R., Nielsen, J. B., and Pass, R. Universally composable protocols with relaxed set-up assumptions. In *FOCS* (2004), IEEE Computer Society, pp. 186–195.

[7] Barbosa, M., Moss, A., and Page, D. Compiler assisted elliptic curve cryptography. In *Proceedings of the 2007 OTM confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part II* (Berlin, Heidelberg, 2007), OTM'07, Springer-Verlag, pp. 1785–1802.

[8] Barthe, G., Grégoire, B., Heraud, S., and Béguelin, S. Z. Computer-aided security proofs for the working cryptographer. In *CRYPTO* (2011), pp. 71–90.

[9] Bellare, M., Garay, J. A., and Rabin, T. Fast batch verification for modular exponentiation and digital signatures. In *EUROCRYPT '98* (1998), vol. 1403 of LNCS, Springer, pp. 236–250.

[10] Blazy, O., Fuchsbauer, G., Izabachène, M., Jambert, A., Sibert, H., and Vergnaud, D. Batch groth-sahai. In *ACNS '10* (2010), Springer, pp. 218–235.

[11] Boneh, D., Boyen, X., and Shacham, H. Short group signatures. In *CRYPTO '04* (2004), vol. 3152 of LNCS, pp. 45–55.

[12] Boneh, D., and Franklin, M. K. Identity-based encryption from the Weil pairing. In *CRYPTO* (2001), pp. 213–229.

[13] Boneh, D., Lynn, B., and Shacham, H. Short signatures from the Weil pairing. In *ASIACRYPT '01* (2001), vol. 2248 of LNCS, pp. 514–532.

[14] Boneh, D., Lynn, B., and Shacham, H. Short signatures from the Weil pairing. *Journal of Cryptology 17(4)* (2004), 297–319.

[15] Boyd, C., and Pavlovski, C. Attacking and repairing batch verification schemes. In *Advances in*

Cryptology – ASIACRYPT '00 (2000), vol. 1976, pp. 58–71.

[16] Boyen, X. Mesh signatures: How to leak a secret with unwitting and unwilling participants. In *EUROCRYPT* (2007), vol. 4515, pp. 210–227.

[17] Camenisch, J., Hohenberger, S., and Pedersen, M. Ø. Batch verification of short signatures. In *EUROCRYPT '07* (2007), vol. 4515 of LNCS, Springer, pp. 246–263. Full version at `http://eprint.iacr.org/2007/172.`

[18] Camenisch, J., and Lysyanskaya, A. Signature schemes and anonymous credentials from bilinear maps. In *CRYPTO '04* (2004), vol. 3152 of LNCS, Springer, pp. 56–72.

[19] Camenisch, J., Rohe, M., and Sadeghi, A. Sokrates - a compiler framework for zero- knowledge protocols. In *Proceedings of the Western European Workshop on Research in Cryptology* (2005), WEWoRC 2005.

[20] Cao, T., Lin, D., and Xue, R. Security analysis of some batch verifying signatures from pairings. *International Journal of Network Security 3*, 2 (2006), 138–143.

[21] Car 2 Car. Communication consortium. `http://car-to-car.org`.

[22] Cha, J. C., and Cheon, J. H. An identity-based signature from gap Diffie-Hellman groups. In *PKC '03* (2003), vol. 2567 of LNCS, Springer, pp. 18–30.

[23] Chatterjee, S., and Sarkar, P. HIBE with short public parameters without random oracle. In *ASIACRYPT '06* (2006), vol. 4284 of LNCS, pp. 145–160.

[24] Chaum, D., and van Heyst, E. Group signatures. In *EUROCRYPT* (1991), pp. 257–265.

[25] Chow, S. S. M., Yiu, S.-M., and Hui, L. C. Efficient identity based ring signature. In *ACNS* (2005), vol. 3531 of LNCS, pp. 499–512.

[26] Ferrara, A. L., Green, M., Hohenberger, S., and Pedersen, M. Ø. Practical short signature batch verification. In *CT-RSA* (2009), vol. 5473 of LNCS, pp. 309–324.

[27] Fiat, A. Batch RSA. In *Advances in Cryptology – CRYPTO '89* (1989), vol. 435, pp. 175–185.

[28] Goldwasser, S., Micali, S., and Rivest, R. L. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Computing 17(2)* (1988).

[29] Harn, L. Batch verifying multiple DSA digital signatures. *Electronics Letters 34(9)* (1998), 870–871.

[30] Harn, L. Batch verifying multiple RSA digital signatures. *Electronics Letters 34(12)* (1998), 1219–1220.

[31] Henecka, W., K ögl, S., Sadeghi, A.-R., Schneider, T., and Wehrenberg, I. Tasty: tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 451–462.

[32] Hess, F. Efficient identity based signature schemes based on pairings. In *Selected Areas in Cryptography* (2002), vol. 2595 of LNCS, Springer, pp. 310–324.

[33] Hohenberger, S., and Waters, B. Realizing hash-and-sign signatures under standard assumptions. In *EUROCRYPT* (2009), pp. 333–350.

[34] Hohenberger, S., and Waters, B. Constructing verifiable random functions with large input spaces. In *EUROCRYPT* (2010), pp. 656–672.

[35] Hwang, M.-S., Lee, C.-C., and Tang, Y.-L. Two simple batch verifying multiple digital signatures. In *3rd Information and Communications Security (ICICS)* (2001), pp. 233–237.

[36] Hwang, M.-S., Lin, I.-C., and Hwang, K.-F. Cryptanalysis of the batch verifying multiple RSA digital signatures. *Informatica, Lithuanian Academy of Sciences 11*, 1 (2000), 15–19.

[37] Kiyomoto, S., Ota, H., and Tanaka, T. A security protocol compiler generating c source codes. In *Proceedings of the 2008 International Conference on Information Security and Assurance (isa 2008)* (Washington, DC, USA, 2008), ISA '08, IEEE Computer Society, pp. 20–25.

[38] Laih, C.-S., and Yen, S.-M. Improved digital signature suitable for batch verification. *IEEE Transactions on Computers 44*, 7 (1995), 957–959.

[39] Law, L., and Matt, B. J. Finding invalid signatures in pairing-based batches. In *Cryptography and Coding* (2007), vol. 4887 of LNCS, pp. 34–53.

[40] Lee, S., Cho, S., Choi, J., and 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.

[41] Lim, C., and Lee, P. Security of interactive DSA batch verification. In *Electronics Letters* (1994), vol. 30(19), pp. 1592–1593.

[42] Lowe, G. Casper: a compiler for the analysis of security protocols. *J. Comput. Secur. 6*, 1-2 (Jan. 1998), 53–84.

[43] Lucks, S., Schmoigl, N., and Tatli, E. I. Issues on designing a cryptographic compiler. In *WEWoRC* (2005), pp. 109–122.

[44] Lynn, B. The Stanford Pairing Based Crypto Library. Available from `http://crypto.stanford.edu/pbc`.

[45] MacKenzie, P., Oprea, A., and Reiter, M. K. Automatic generation of two-party computations. In *Proceedings of the 10th ACM conference on Computer and communications security* (New York, NY, USA, 2003), CCS '03, ACM, pp. 210–219.

[46] Malkhi, D., Nisan, N., Pinkas, B., and Sella, Y. Fairplay – a secure two-party computation system. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 20–20.

[47] Matt, B. J. Identification of multiple invalid signatures in pairing-based batched signatures. In *Public Key Cryptography* (2009), pp. 337–356.

[48] Matt, B. J. Identification of multiple invalid pairing-based signatures in constrained batches. In *Pairing* (2010), pp. 78–95.

[49] Meiklejohn, S., Erway, C. C., Küpçü, A., Hinkle, T., and Lysyanskaya, A. Zkpdl: a language-based system for efficient zero-knowledge proofs and

electronic cash. In *Proceedings of the 19th USENIX conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 13–13.

[50] Micali, S., Rabin, M. O., and Vadhan, S. P. Verifiable random functions. In *FOCS* (1999), pp. 120–130.

[51] Naccache, D. Secure and *practical* identity-based encryption, 2005. Cryptology ePrint Archive: Report 2005/369.

[52] Naccache, D., M'Raïhi, D., Vaudenay, S., and Raphaeli, D. Can DSA be improved? complexity trade-offs with the digital signature standard. In *Advances in Cryptology – EUROCRYPT '94* (1994), vol. 950, pp. 77–85.

[53] Perez, L. J. D., and Scott, M. Designing a code generator for pairing based cryptographic functions. In *Proceedings of the 4th international conference on Pairing-based cryptography* (Berlin, Heidelberg, 2010), Pairing'10, Springer-Verlag, pp. 207–224.

[54] Pozza, D., Sisto, R., and Durante, L. Spi2java: Automatic cryptographic protocol java code generation from spi calculus. In *Proceedings of the 18th International Conference on Advanced Information Networking and Applications - Volume 2* (Washington, DC, USA, 2004), AINA '04, IEEE Computer Society, pp. 400–.

[55] Rivest, R. L., Shamir, A., and Tauman, Y. How to leak a secret. In *ASIACRYPT* (2001), pp. 552–565.

[56] Scott, M. Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL), Oct. 2007. Published by Shamus Software Ltd., `http://www.shamus.ie/`.

[57] SeVeCom. Security on the road. `http://www.sevecom.org`.

[58] Shacham, H., and Boneh, D. Improving SSL handshake performance via batching. In *Cryptographer's Track at RSA Conference '01* (2001), vol. 2020, pp. 28–43.

[59] Shamir, A. Identity-based cryptosystems and signature schemes. In *CRYPTO* (1984), pp. 47–53.

[60] Song, D. X., Perrig, A., and Phan, D. Agvi - automatic generation, verification, and implementation of security protocols. In *Proceedings of the 13th International Conference on Computer Aided Verification* (London, UK, UK, 2001), CAV '01, Springer-Verlag, pp. 241–245.

[61] Stanek, M. Attacking LCCC batch verification of RSA signatures, 2006. Cryptology ePrint Archive: Report 2006/111.

[62] Waters, B. Efficient identity-based encryption without random oracles. In *EUROCRYPT '05* (2005), vol. 3494 of LNCS, Springer, pp. 320–329.

[63] Waters, B. Dual System Encryption: Realizing Fully Secure IBE and HIBE under Simple Assumptions. In *CRYPTO* (2009), pp. 619–636.

[64] Yoon, H., Cheon, J. H., and Kim, Y. Batch verifications with ID-based signatures. In *ICISC* (2004), Lecture Notes in Computer Science, pp. 233–248.

[65] Zaverucha, G. M., and Stinson, D. R. Group testing and batch verification. In *Proceedings of the 4th international conference on Information theoretic security* (Berlin, Heidelberg, 2010), ICITS'09, Springer-Verlag, pp. 140–157.

[66] Zhang, F., and Kim, K. Efficient ID-based blind signature and proxy signature from bilinear pairings. In *8th Information Security and Privacy, Australasian Conference (ACISP)* (2003), vol. 2727, pp. 312–323.

[67] Zhang, F., Safavi-Naini, R., and Susilo, W. Efficient verifiably encrypted signature and partially blind signature from bilinear pairings. In *Progress in Cryptology – INDOCRYPT '03* (2003), vol. 2904, pp. 191–204.