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

Joseph A. Akinyele[*§] Matthew Green[*†] Susan Hohenberger[*‡]
Matthew W. Pagano[*§]

February 26, 2014

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. In particular, it found an algorithm that is 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.

In this work, we expand upon our paper on AutoBatch appearing in ACM CCS 2012 [2] in a number of ways. We add a new loop-unrolling technique and show that it helps cut the batch verification cost of one scheme by roughly half. We describe our pruning and search algorithms in greater detail, including pseudocode and diagrams. All experiments were also re-run using the RELIC pairing library. We compare those results to our earlier results using the MIRACL library, and discuss why RELIC outperforms MIRACL in all but two cases. Automated proofs of several new batching algorithms are also included.

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.

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

[*]Johns Hopkins University, {`akinyelj, mgreen, susan, mpagano`}`@cs.jhu.edu`

[†]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 under contract N00014-11-1-0470, NSF grant CNS 1010928 and HHS 90TR0003/01. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the HHS.

[‡]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 under contract N00014-11-1-0470, NSF CNS 1154035, and a Microsoft Faculty Fellowship. Applying to all authors, the views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

[§]Supported in part by NSF grant CNS 1010928 and HHS 90TR0003/01.

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 [23, 60], 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 [15] 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 [28] — a 95% saving!

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,[1] an automated tool that transforms a high-level description of a signature scheme[2] into an optimized batch verification program in either Python or C++. This high-level specification is written in a language called *Scheme Description Language* (SDL), which is designed specifically for automation. AutoBatch takes as input an SDL specification of a signature scheme and 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. Despite these heuristics, AutoBatch is not guaranteed to terminate but we conjecture that it does in practice. Our tool produces a modified SDL file and executable code, which 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 an argument for why security is preserved.

AutoBatch was tested on several pairing-based schemes. It produced the first batching algorithms, to our knowledge, for the Camenisch-Lysyanskaya [20] and Hohenberger-Waters [36] signatures.[3] It also discovered a faster algorithm for batching the proofs of the verifiable random functions (VRF) of Hohenberger and Waters [37]. Moreover, AutoBatch is able to handle batches with more than one type of signature. Indeed, we found that the Hess [35] and Cha-Cheon [24] 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 to 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]The AutoBatch source and test cases described herein are publicly available at `https://github.com/JHUISI/auto-tools`.

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

[3]It also produced a candidate batching scheme for the Waters dual-system [66] signatures, although this signature scheme does not have perfect correctness and therefore our techniques do not immediately apply to it. See Section 2.1.1 for more.

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. Our parser assumes that the scheme has been implemented in Python following a specific structure (see Section 3.5 for more details). Given such an implementation, the Parser obtains the signature verification equation and encodes it into an intermediate representation in SDL.
2. A Batcher, which takes as input an SDL file describing a signature verification equation. In addition to the signature verification equation, the Batcher requires details in SDL such as types, variable names of public parameters and signatures, and estimated batch size. It first consolidates the set of individual verification equations into a single equation, then derives a batch verification equation. The Batcher then searches through a series of rules, which may be applied repeatedly, to optimize the equation and thus derive a new equation of a batch verifier. The output of the Batcher is a second SDL file, which includes the individual and batch verifiers, along with an analysis of the batcher's estimated running time. For transparency, the Batcher optionally outputs a LaTeX file that can be compiled into a human-readable document describing the batching algorithm and arguing that it maintains the unforgeability of the original scheme.
3. A Code Generator, which takes the output of the Batcher and generates working source code to implement the batch verifier. The batch verifier implementation includes group 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. The user can choose either Python or C++ as the output language, either building on the MIRACL [59] or RELIC [4] library.

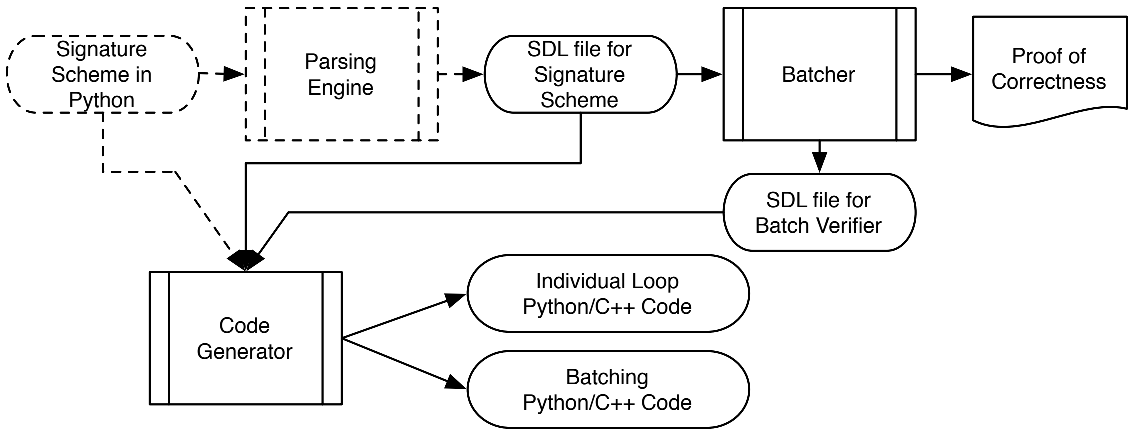
There are two usage scenarios for AutoBatch. The most common may be that a user begins with a hand-coded SDL file and feeds this directly into the Batcher. Since SDL files are human-readable ASCII-based files containing a mathematical representation of the scheme, some developers may prefer to implement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

As a second scenario, if the user has a working implementation of the scheme in Charm [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 is a Python and C++ based prototyping framework created by Akinyele et al. [1] 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 [10] for their invention of EasyCrypt, an automated tool for generating security proofs of cryptographic systems 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 [29] for a variant of RSA. In 1994, an interactive batch verifier for DSA presented in an early version of [55] was broken by Lim and Lee [44]. In 1995, Laih and Yen proposed a new method for batch verification of DSA and RSA signatures [41], but the RSA batch verifier was broken five years later by Boyd and Pavlovski [17]. In 1998, two batch verification techniques were presented for DSA and RSA [32, 33] but both were later broken [17, 38, 39]. The same year, Bellare, Garay and Rabin took the first systematic look at batch verification [11] and presented three generic methods for batching modular exponentiations, one of which is called the *small exponents test.* Unfortunately, in 2000, Boyd and Pavlovski [17] published attacks against various batching schemes which were using the small exponents test incorrectly. In 2003-2004, several batch verification schemes based on bilinear maps (a.k.a.,



Signature Scheme in Python
Parsing Engine
SDL file for Signature Scheme
Batcher
Proof of Correctness
SDL file for 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 correctness (as a PDF typeset using 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.

pairings) were proposed [24, 68, 70, 71] but all were later broken by Cao, Lin and Xue [22]. In 2006, a method was given for identifying invalid signatures in RSA-type batches [43], but it was also flawed [64].

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 correctness 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 [19] 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 [17] 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 correctness for every algorithm it outputs relative to this definition (in Section 3.3), 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 works on batch verification mentioned 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 [61]. Boneh, Lynn and Shacham provided a single-signer batch verifier for BLS signatures [15]. Camenisch, Hohenberger and Pedersen [19] gave multiple-signer batch verifiers for Waters identity-based signatures [65] 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 [28]. Blazy, Fuchsbauer, Izabachéne, Jambert, Sibert and Vergnaud [12] 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 [42, 50, 51].

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 [40,45,46,57,63], optimizations to software implementations involving elliptic-curve cryptography [9] and bilinear-map functions [56], secure two-party computation [34,48,49], and zero-knowledge proofs [3,5–7,21,30,52].

2 Background

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

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 typically said to be correct *(or perfectly correct) if for all* $\mathsf{Gen}(1^\ell) \rightarrow (pk, sk)$*,*

$$\mathsf{Verify}(pk, m, \mathsf{Sign}(sk, m)) = 1 \textit{ for all } m.$$

That is, a scheme is correct if all honestly generated signatures pass the verification test. Our focus will be on perfectly correct schemes, however, we discuss in Section 2.1.1 the implications for batch verification if some correctness error is allowed.

A scheme is defined to be *unforgeable* as follows [31]: 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 $\mathsf{Verify}(pk, m, \sigma) = 1$ must be negligible in ℓ.

In this work, we explore three variants:

1. **Identity-Based Signatures [62]**: 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 public key of the master authority and the public identity string of the purported signer.
2. **Privacy Signatures:** Group [26] and ring [58] 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 who.
3. **Verifiable Random Functions [53]:** 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 The Basics of Batch Verification for Signatures

Our security focus here is not directly on unforgeability [31]. 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 (except perhaps with a negligible probability).[4] *If an input scheme is unforgeable, then our batching algorithm will preserve this property in the output scheme.* If an insecure scheme is provided as input, then all bets are off on the output.

[4]We assume perfectly correct schemes here.

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 batch verification from Bellare, Garay and Rabin [11] as extended in [19] to deal with multiple signers. We note that this definition is well specified for perfectly correct schemes, but not for schemes that allow some correctness error. We discuss this further shortly.

Definition 2.2 (Batch Verification of Signatures) *Let ℓ be the security parameter. Suppose* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *is a signature scheme with perfect correctness, $k, n \in poly(\ell)$, 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 a probabilistic algorithm* 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 perfectly-correct 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 [8].

2.1.1 On Schemes with a Correctness Error

The standard definition for signature batch verification (as presented in Definition 2.2)[5] assumes that the basic signature scheme has perfect correctness. That is, the first part of the definition inherently assumes that all valid signatures will pass the individual verification test. This is the case for the majority of signature schemes as well as all signature schemes that we are aware of being actively used in practice.

However, one could imagine a signature scheme with a negligible or small constant correctness error. One example of a scheme with a negligible correctness error is the Waters09 scheme as derived from the Waters Dual-System IBE [66] using the technique described by Naor [14]. In this scheme, a signature on message m corresponds to the IBE private key on identity m. The verification test operates by choosing a random message m', encrypting it for identity m, running the decryption algorithm using the signature as the private key, and testing to see that decryption successfully recovers m'. Since the Dual-System IBE [66] has a negligible correctness error in the decryption algorithm, this signature scheme also has a negligible correctness error in verification. This leaves the question: what is the right batching definition for such a scheme?

For a scheme that allows an arbitrary amount of correctness error, the first requirement of Definition 2.2 no longer makes sense. Rather in this setting it seems to us that one could no longer base the batching security on the base signature security, but rather would have to create a new game-based definition that simulated the batching scenario and directly prove that the algorithm matches the definition. Direct proofs of this sort are currently beyond our ability to automate.

One might instead narrow the focus to schemes that allow at most a negligible correctness error. In this case, we suggest relaxing both of the batching requirements by a negligible probability taken over the randomness of the *individual and batch* verification algorithms. We leave as an open problem a formal treatment of batching for schemes in this class.

We tested AutoBatch on one scheme with a correctness error, Waters09 [66], because its complication made it a challenging test case. We report on the candidate batching algorithm we found in Section 4,

[5]We added the restriction to perfect correctness in Definition 2.2. It was assumed in prior works but not always made explicit.

although we note there and in Appendix D that our automated proofs were only written to handle schemes with perfect correctness. This is a correction over the conference version of this work [2] which did not make this distinction.

2.2 Algebraic Setting

Bilinear Groups. Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ be groups of prime order q. A map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is an admissible bilinear map (or pairing) if it satisfies the following three properties:

1. Bilinearity: for all $g \in \mathbb{G}_1$, $h \in \mathbb{G}_2$, and $a, b \in \mathbb{Z}_q$, it holds that $e(g^a, h^b) = e(g, h)^{ab}$.
2. Non-degeneracy: if g and h are generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively, then $e(g, h)$ is a generator of $\mathbb{G}_T$.
3. Efficiency: there exists an efficiently computable function that given any $g \in \mathbb{G}_1$ and $h \in \mathbb{G}_2$, computes $e(g, h)$.

An admissible bilinear map generator BSetup is an algorithm that on input a security parameter 1^ℓ, outputs the parameters for a bilinear group $(q, g, h, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ such that groups of prime order $q \in \Theta(2^\ell)$, $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ are groups of order q where g generates $\mathbb{G}_1$, h generates $\mathbb{G}_2$ and $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is an admissible bilinear map.

The above bilinear map is called *asymmetric* and our implementations use this highly efficient setting. We also consider *symmetric* maps where there is an efficient isomorphism $\psi : \mathbb{G}_1 \rightarrow \mathbb{G}_2$ (and vice versa) such that a symmetric map $\hat{e}$ is defined as $\hat{e} : \mathbb{G}_1 \times \psi(\mathbb{G}_1) \rightarrow \mathbb{G}_T$. We abstractly treat symmetric groups equally ($\mathbb{G}_1 = \mathbb{G}_2$) for simplicity.

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 [17] 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 $E(\mathbb{F}_p) \setminus \mathbb{G}_1$. 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 $y^q \mod q = 1$ [19]. 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. [28] and confirmed by our tests.

2.3 Batch Verification in Bilinear Groups

Let us recall from [28] the formal definition of a *bilinear-based* (or pairing-based) batch verifier. A pairing-based verification equation is represented by a *generic pairing-based claim* X corresponding to a boolean relation of the following form: $\prod_{i=1}^{k} e(f_i, h_i)^{c_i} \overset{?}{=} A$, for $k \in \text{poly}(\tau)$ and $f_i \in \mathbb{G}_1, h_i \in \mathbb{G}_2$ and $c_i \in \mathbb{Z}_q^*$, for each $i = 1, \ldots, k$. A pairing-based verifier Verify for a generic pairing-based claim is a probabilistic poly(τ)-time algorithm which on input the representation $\langle A, f_1, \ldots, f_k, h_1, \ldots, h_k, c_1, \ldots, c_k\rangle$ of a claim X, outputs *accept* if X holds and *reject* otherwise. We define a batch verifier for pairing-based claims.

Definition 2.3 (Bilinear-based Batch Verifier)
Let $\mathsf{BSetup}(1^\tau) \rightarrow (q, g_1, g_2, \mathbb{G}_a, \mathbb{G}_b, \mathbb{G}_T, e)$*. For each* $j \in [1, \eta]$*, where* $\eta \in poly(\tau)$*, let* $X^{(j)}$ *be a generic pairing-based claim and let* Verify *be a pairing-based verifier. We define a* pairing-based batch verifier *for* Verify *as a probabilistic poly*(τ)*-time algorithm which outputs:*

- accept *if* $X^{(j)}$ *holds for all* $j \in [1, \eta]$*;*
- reject *if* $X^{(j)}$ *does not hold for any* $j \in [1, \eta]$ *except with negligible probability.*

2.4 Small Exponents Test Applied to Bilinear Groups

Bellare, Garay and Rabin [11] proposed methods for verifying multiple equations of the form $y_i = g^{x_i}$ for $i = 1$ to n, where g is a generator for a group of prime order. One might be tempted to just multiply these equations together and check if $\prod_{i=1}^{n} y_i = g^{\sum_{i=1}^{n} x_i}$. However, it would be easy to produce two pairs (x_1, y_1) and (x_2, y_2) such that the product of them verifies correctly, but each individual verification does not, e.g. by submitting the pairs $(x_1 - \alpha, y_1)$ and $(x_2 + \alpha, y_2)$ for any α. Instead, Bellare et al. proposed the following method for batching the verification of these equations, which we will shortly apply to bilinear groups.

The Small Exponents Test of Bellare, Garay and Rabin: Choose exponents δ_i of (a small number of) ℓ_b bits and compute $\prod_{i=1}^{n} y_i^{\delta_i} = g^{\sum_{i=1}^{n} x_i \delta_i}$. Then the probability of accepting a bad pair is $2^{-\ell_b}$. The size of ℓ_b is a tradeoff between efficiency and security. (By default in AutoBatch, we set $\ell_b = 80$ bits and select random exponents from the range $[1, 2^{\lambda} - 1]$. Even though 0 is allowed for the test, we forbid it in our implementation.)

Subsequently, Ferrara, Green, Hohenberger and Pedersen [28] proved that the Small Exponents Test could be securely applied to bilinear groups as well. We recall the following theorem from their work which encapsulates the test as well.

Theorem 2.4 (Small Exponents Test Applied to Bilinear Groups [28]) *Let* $\mathsf{BSetup}(1^{\tau}) \rightarrow (q, g_1, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ *where q is prime. For each $j \in [1, \eta]$, where $\eta \in poly(\tau)$, let $X^{(j)}$ be a generic claim as in Definition 2.3. For simplicity, assume that $X^{(j)}$ is of the form $A \stackrel{?}{=} Y^{(j)}$ where A is fixed for all j and all the input values to the claim $X^{(j)}$ are in the correct groups. For any random vector $\Delta = (\delta_1, \ldots, \delta_\eta)$ of ℓ_b bit elements from $\mathbb{Z}_q$, an algorithm* Batch *which tests the following equation $\prod_{j=1}^{\eta} A^{\delta_j} \stackrel{?}{=} \prod_{j=1}^{\eta} {Y^{(j)}}^{\delta_j}$ is a pairing-based batch verifier that accepts an invalid batch with probability at most $2^{-\ell_b}$.*

In later sections, we will frequently make use of the small exponents tests and rely on the security guarantees of Theorem 2.4 as proven by Ferrara et al. [28].

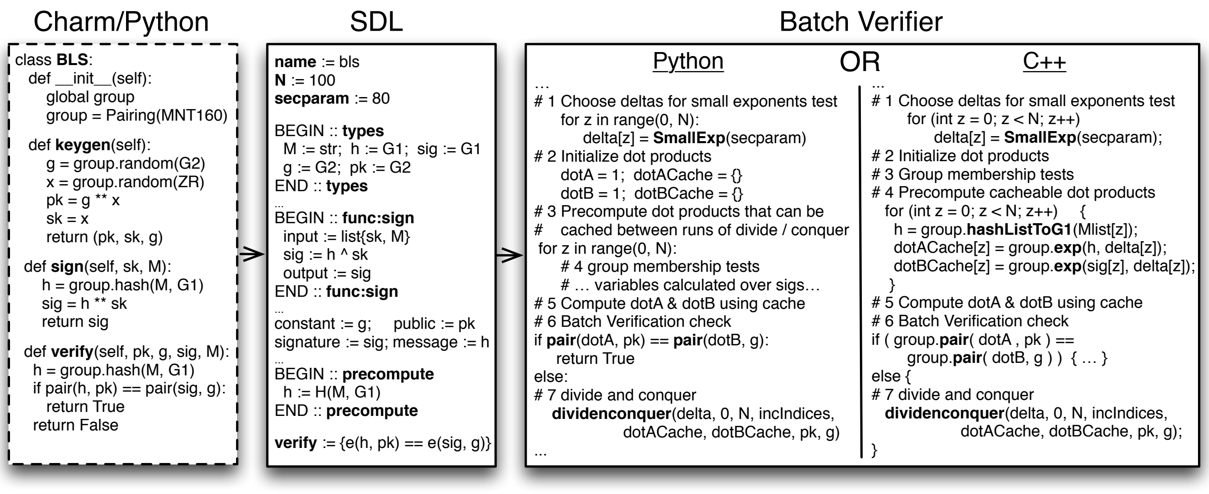
3 The AutoBatch Toolchain

In this section we summarize the techniques used by AutoBatch to programmatically generate batch verifiers from standard signature schemes. A high level abstraction is 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. We provide a description of the SDL grammar in Appendix E, as well as a description of the SDL semantics and several examples in Appendix F. 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.[6]

2. Apply techniques and optimize the batch 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 re-arranging the components to minimize the number of expensive operations. To prevent known attacks, we apply the small exponents test of Bellare, Garay and Rabin [11], and optimize the resulting equation to ensure that all signature elements are in the group with the smallest representation (typically, $\mathbb{G}_1$). Additionally, the Batcher embeds a recursive *divide-and-conquer* strategy to handle cases where batch verification fails due to invalid signatures. This binary search strategy is borrowed from Law and Matt [42] and could be extended to support other methods

[6]We developed this capability for two reasons. First, there is already a library of pairing-based signature schemes 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:
  def __init__(self):
     global group
     group = Pairing(MNT160)

  def keygen(self):
     g = group.random(G2)
     x = group.random(ZR)
     pk = g ** x
     sk = x
     return (pk, sk, g)

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

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

```
name := bls
N := 100
secparam := 80

BEGIN :: types
 M := str;  h := G1;  sig := G1
 g := G2;  pk := G2
END :: types
...
BEGIN :: func:sign
 input := list{sk, M}
 sig := h ^ sk
 output := sig
END :: func:sign
...
constant := g;     public := pk
signature := sig; message := h
...
BEGIN :: precompute
 h := H(M, G1)
END :: precompute

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

```
...
# 1 Choose deltas for small exponents test
     for z in range(0, N):
          delta[z] = SmallExp(secparam)
# 2 Initialize dot products
     dotA = 1;  dotACache = {}
     dotB = 1;  dotBCache = {}
# 3 Precompute dot products that can be
#    cached between runs of divide / conquer
 for z in range(0, N):
     # 4 group membership tests
     # … variables calculated over sigs…
# 5 Compute dotA & dotB using cache
# 6 Batch Verification check
if pair(dotA, pk) == pair(dotB, g):
    return True
else:
# 7 divide and conquer
   dividenconquer(delta, 0, N, incIndices,
                  dotACache, dotBCache, pk, g)
...
```

```
...
# 1 Choose deltas for small exponents test
      for (int z = 0; z < N; z++)
            delta[z] = SmallExp(secparam);
# 2 Initialize dot products
# 3 Group membership tests
# 4 Precompute cacheable dot products
   for (int z = 0; z < N; z++)     {
     h = group.hashListToG1(Mlist[z]);
     dotACache[z] = group.exp(h, delta[z]);
     dotBCache[z] = group.exp(sig[z], delta[z]);
    }
# 5 Compute dotA & dotB using cache
# 6 Batch Verification check
if ( group.pair( dotA , pk ) ==
      group.pair( dotB, g ) )  { … }
else {
# 7 divide and conquer
   dividenconquer(delta, 0, N, incIndices,
                dotACache, dotBCache, pk, g);
}
```

Figure 2: The Boneh-Lynn-Shacham (BLS) signature scheme [15] 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.

that outperform this approach. Finally, the output of this phase is a modified SDL file, and (optionally) a human-readable proof that the resulting equation is a secure 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 an estimate of the runtime 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.[7]

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 Python or C++ using 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). Two fragments of generated code (Python and C++) are shown in Figure 2.

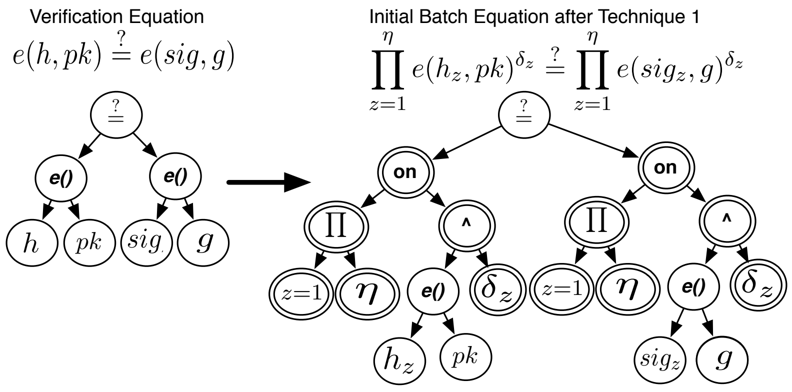
We will now describe each of the above steps in detail.

3.1 Batching and Optimization

Given an SDL file containing the verification equation and variable types, the Batcher first securely consolidates the individual verification equations into a single equation using the small exponents test. Then, the Batcher applies a series of optimizations to the batch verification equation in order to derive an efficient batch verifier. Many of these techniques were first explored in previous works [19,28]. 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.[8] Furthermore, unlike previous works we are

[7]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.

[8]For example: techniques 2 and 3 of [19] each combine a series of logical operations that are more widely applicable and easily managed by splitting them into finer-grained sub-techniques.



Verification Equation
$e(h, pk) \overset{?}{=} e(sig, g)$
Initial Batch Equation after Technique 1
$\prod_{z=1}^{\eta} e(h_z, pk)^{\delta_z} \overset{?}{=} \prod_{z=1}^{\eta} e(sig_z, g)^{\delta_z}$
$\overset{?}{=}$
e()
h
pk
sig
g
on
$\prod$
^
$z{=}1$
η
δ_z
h_z
sig_z

Figure 3: The Boneh-Lynn-Shacham (BLS) signature scheme [15] with same signer and η signatures in a batch. We show the abstract syntax tree (AST) of the unoptimized batch equation after Batcher has applied technique 1 by combining all instances of the verification equations (denoted by $\prod$ node) and applying the small exponents test (denoted by δ_z node).

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, the Batcher first parses and performs type checking on the SDL input file to extract an abstract syntax tree (AST) representing the verification equation. During the type checking, it informs users if there are type mismatches or if the typing information is incomplete in SDL. Next, the Batcher traverses the AST of the verification equation, applying various techniques at various nodes in the tree.

We now list those techniques and provide details on how some of these techniques are implemented on the AST.

Technique 0a: 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 [11] to combine these equations into a single verification equation.[9] A variation of this is *Technique 0b* which is applicable for schemes that utilize for loops in the verification equation (e.g., VRF [37]). If the bounds over the loop are known it might be useful to unroll the loop to allow application of other techniques.

$$\text{Replace for } i = 1 \text{ to } t:\ e(g, h_i) \overset{?}{=} e(c, d_i) \text{ with } e(g, h_1)^{\delta_1} \cdot \ldots \cdot e(g, h_t)^{-\delta_t} \overset{?}{=} e(c, d_1)^{\delta_1} \cdot \ldots \cdot e(c, d_t)^{-\delta_t}$$

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 [28], 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 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. See Figure 3 for an example.

Technique 2: Move exponents inside the pairing. When a term 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.

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

[9]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 $e(a, b)^{\delta_1} e(c, d)^{-\delta_1} e(a, c)^{\delta_2} e(g, h)^{-\delta_2} = 1$.

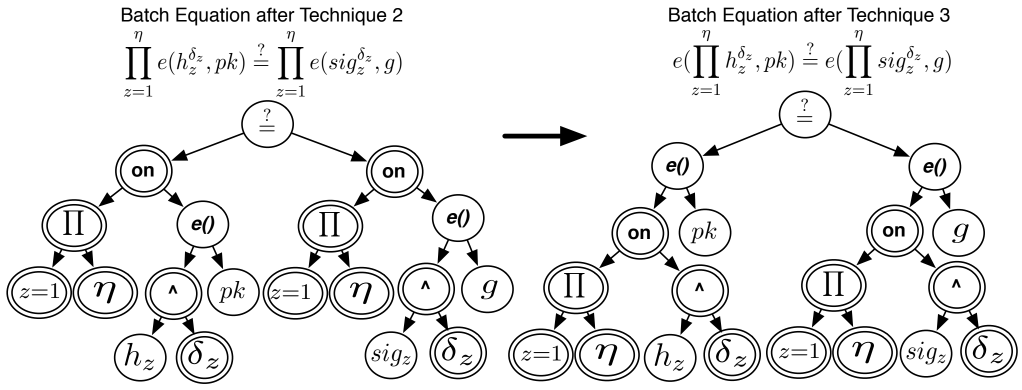

Batch Equation after Technique 2
$\prod_{z=1}^{\eta} e(h_z^{\delta_z}, pk) \stackrel{?}{=} \prod_{z=1}^{\eta} e(sig_z^{\delta_z}, g)$
Batch Equation after Technique 3
$e(\prod_{z=1}^{\eta} h_z^{\delta_z}, pk) \stackrel{?}{=} e(\prod_{z=1}^{\eta} sig_z^{\delta_z}, g)$
on
e()
z=1
pk
g
sig_z
h_z
δ_z
η

Figure 4: The Boneh-Lynn-Shacham (BLS) signature scheme [15] with same signer and η signatures in a batch. Upon applying technique 1 from Figure 3 to obtain the initial secure batch verifier, the goal is to optimize the equation. We first show the AST of the equation *after* the Batcher has applied technique 2 (move exponents inside the pairing). Then, we show the result of applying technique 3a (move products inside the pairing) to arrive at an optimized batch equation.

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.[10]

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

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

A special case of this technique is *Technique 3b* where $\eta = 2$. In this case, when two terms share a common first or second argument, they can also be combined. For example:

$$\text{Replace } e(a, g) \cdot e(b, g) \text{ with } e(a \cdot b, g) \text{ where } a \neq 1 \wedge b \neq 1.$$

For a concrete example, we show how techniques 2 and 3a are programmatically applied to the BLS scheme [15] in Figure 4.

Technique 4: Optimize the Waters Hash. A variety of identity-based signature schemes employ a hash function by Waters [65], which can be generalized [25, 54]. Verifying signatures generated by these schemes requires hashing identity strings of the form $V = v_1 v_2 \dots v_z$ where each v_i is a short string. The hash function is evaluated as $u' \prod_{i=1}^{z} u_i^{v_i}$ where u' and $u_1 u_2 \dots u_z$ are public generators in $\mathbb{G}_1$ or $\mathbb{G}_2$.

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. This is most useful when $\eta > z$.

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

In future versions, AutoBatch will output code to switch between the most efficient verifier when $\eta > z$ and $\eta \leq z$.

[10]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.

Technique 5: Distribute products. When a product is applied to two or more terms, distribute the product to each term 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: 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 pre-compute 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 7: Precompute constant pairings. When pairings have a constant first and second argument, 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).

Technique 8: Split pairings. In some rare cases it can be useful to apply technique 3b 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 [18].

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 7) *could* have been applied in published algorithm descriptions [20, 37], 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.

3.2 Technique Search Approach

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. Moreover, some techniques might actually worsen the performance of the verifier in the hope of applying other techniques to obtain a better solution. An automated search algorithm must balance all of these issues and must also identify the orderings in an efficient manner.

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. This limit can be vastly different as the complexity of the scheme increases. While 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 Search Approach: Our approach is a "pruned" breadth-first search (PBFS) which utilizes a finite state transition function to constrain the transitions between techniques. This transition function determines which techniques can be applied to the current state and was constructed with our experience of how the optimization techniques work together logically. For instance, if technique 5 is applied to the current state (i.e., distribute products to pairings), then techniques 2-4 most likely will apply given that these techniques move exponents or products inside pairings. From the current state, only the subset of techniques in which the conditions for the transformation are met are pursued further in the search.

Our search algorithm is broken down into three stages. The first stage of the search is to try technique 0a if there are multiple verification equations. After consolidating the verification equations, we try technique 3b since there may have been pairings with common elements from separate equations. Our intuition for attempting technique 3b in this stage is to combine as many pairings as possible before embarking on the search. The side effect is that it reduces the number of paths explored by the PBFS, thereby making the search more efficient. Moreover, it is useful to attempt technique 7 at this stage and precompute pairings that utilize generators. We then apply technique 1 to combine η instances of the equations to form an initial batch verifier. However, if the scheme specifies a single verification equation, then only technique 1 is applied in the first stage.

The second stage of the search employs the PBFS (starting with technique 2) and terminates when none of the techniques can be applied to the current state of a batch verifier. Each path from the set of ordering paths uncovered during the PBFS is evaluated in terms of total running time. The algorithm selects the path from the candidate paths that provides the highest cost savings. From the selected path, the final (or post-processing) stage of the search attempts to apply technique 0b (unroll loops) if the equation utilizes for loops. We delay testing for technique 0b until the post-processing stage to limit the search space for an efficient batch verifier. If technique 0b is applied, then we always attempt technique 3b given that there may now be pairings that can be further combined.

To prevent infinite loops during our PBFS, the state function disallows the application of certain techniques that might potentially *undo* optimizations. For example, technique 8 performs a reverse split on pairings to allow further optimizations; this might affect technique 3b, which combines pairings that have common elements. Certain combinations of techniques 8 and 3b lead to an infinite cycle that combines and splits the same pairings. Thus, the state function only allows a transition from Technique 8 to 3b to occur once on a given path. We provide the pseudocode of our search in Algorithm 1 & 2 and our state transition function in Table 1.

Our approach is effective and enables efficiently deriving batch verification algorithms. 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.

Current State	**Next States**
(2, _)	{3a, 3b, 4, 5, 6, 7, 8}
(3a, _)	{2, 3b, 4, 5, 6}
(3b, _)	{2, 3a, 3b, 5, 8}
(4, _)	{2, 3a, 3b, 5}
(5, _)	{2, 3a, 4}
(6, _)	{2, 3b, 5}
(7, _)	{2, 3a, 6}
(8, true)	{2, 4, 5, 6}
(8, false)	{2, 4, 5, 3b, 6}

Table 1: This represents the **transitionFun** of Algorithm 2 developed for pruning our breath-first search (PBFS) algorithm. The function accepts as input the current state which represents the technique that was previously applied to the batch equation and whether there exists a transition from technique 8 to 3b along the path. In an effort to ensure that all paths terminate, the function restricts the transition from technique 8 to 3b to occur once on a given path. Although we do not prove that our algorithm is guaranteed to terminate, we conjecture that it does in practice. In fact, it terminated promptly for all of our test cases.

Algorithm 1 Global search: Our overall search procedure takes as input an AST representation of the initial verification equation, then attempts technique 0a, 3b, 7 and 1 in the pre-processing step. The rest of the algorithm executes the **PBFSearch** algorithm (shown in Algorithm 2) to determine ordering. Then, it applies the post-processing step by attempting technique 0b and 3b if there are loops involved. The search returns the best batch verification algorithm and the order of techniques applied.

```
procedure GlobalSearch($eq$)
    $path1 \leftarrow \{\}$
    $pre\_techniques \leftarrow \{3b, 7\}$        $\rightarrow$ Pre-processing stage
    $applied,\ eq1 \leftarrow$ applyTechnique($technique = 0a,\ eq$)        $\rightarrow$ Try to consolidate equations
    if $applied = True$ then        $\rightarrow$ Technique 0a condition is satisfied
        for all $x \in pre\_techniques$ do
            $applied,\ new\_eq \leftarrow$ applyTechnique($technique = x,\ eq1$)
            if $applied = True$ then
                $path1 \leftarrow path1 + [x]$
                $eq1 \leftarrow new\_eq$
            end if
        end for
    end if
    $applied,\ eq1 \leftarrow$ applyTechnique($technique = 1,\ eq1$)        $\rightarrow$ Combine $\eta$ instances of equations
    assert(applied = True)
    $path1 \leftarrow path1 + [1]$

    $AllThePaths \leftarrow$ PBFSearch($eq1, path1, allPaths = \emptyset, start\_technique = 2$)
    $(bestEq, path2) \leftarrow$ findMin($AllThePaths$)
        $\rightarrow$ Finds path with lowest runtime estimate recorded during PBFS

    $post\_techniques \leftarrow \{0b, 3b\}$        $\rightarrow$ Post-processing stage
    for all $x \in post\_techniques$ do
        $applied,\ new\_eq \leftarrow$ applyTechnique($technique = x,\ bestEq$)
        if $applied = True$ then
            $path2 \leftarrow path2 + [x]$
            $bestEq \leftarrow new\_eq$
        end if
    end for

    return $bestEq,\ path2$
end procedure
```

Algorithm 2 Pruned Breadth-First Search: the PBFS algorithm takes as input an AST of the equation, sequence of applied techniques (called *path*), an empty set for storing all uncovered paths (called *allPaths*), and a start technique for the search. The *path* argument records the techniques being explored in the search execution. The algorithm returns a set of paths dictated by **transitionFunc** which is illustrated in Table 1 and an estimate of the batch verifier runtime that is associated with each path. Our algorithm selects whichever path yields the lowest runtime.

```
1:  procedure PBFSearch($eq$, $path$, $allPaths$, $technique$)
2:      $applied,\ new\_eq \leftarrow$ applyTechnique($technique,\ eq$)
3:
4:      if $applied = True$ then          → Technique condition is satisfied
5:          $path \leftarrow path + [technique]$          → Append technique to path
6:          $checkRes \leftarrow$ checkForEdge($8,\ 3b,\ path$)          → check if transition from 8 to 3b exists in path
7:          $tech\_set \leftarrow$ transitionFunc($technique,\ checkRes$)          → return pruned set
8:          for all $x \in tech\_set$ do
9:              $newAllPaths \leftarrow$ PBFSearch($new\_eq,\ path,\ allPaths,\ x$)
10:             $allPaths \leftarrow allPaths \cup newAllPaths$
11:         end for
12:     else          → Reached dead end with this path
13:         if $path \notin allPaths$ then
14:             $allPaths \leftarrow allPaths \cup path$          → Add $path$ to set of all paths
15:             $time \leftarrow$ estimateRuntime($eq,\ N,\ T$)          → $N$ for batch size & $T$ for group op. costs
16:             recordTime($time,\ path$)          → record in a global database
17:         end if
18:     end if
19:     return $allPaths$
20: end procedure
```

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. 0a), and apply the small exponents test as follows: For each of the $z = 1$ to η signatures, choose random $\delta_1, \delta_2 \in [1, 2^\lambda - 1]$ and compute for each equation:

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

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

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

Step 3: Merge pairings with common first or second argument (tech. 3b):

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

Step 4: Merge pairings with common first or second argument (tech. 3b):

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

Step 5: Move products inside pairings to reduce η pairings to 1 (tech. 3a):

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

Step 6: Distribute products (tech. 5):

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

Step 7: Move products inside pairings to reduce η pairings to 1 (tech. 3a):

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

Figure 5: A fragment of the machine-generated security proof of a single-signer batch verifier for the bilinear CL signature scheme [20]. An earlier portion of the proof asserted that a group membership test would be done prior to checking the final equation. Here the value g is a generator of a bilinear group, the values X, Y are part of the public key, a signature is a tuple (a, b, c) and the message signed is m.

3.3 Security and Machine-Aided Analysis

Efficiency Analysis. Efficiency of the batch verifiers is computed in two separate ways. During the PBFS algorithm, the Batcher uses the batch size specified by the user to compute an estimate of the runtime for all batch verifiers. The resulting estimates enable selection of an efficient batch verifier from many candidate verifiers. As indicated in Algorithm 2, the estimates are calculated using a database of average operation times measured at library initialization. Once the Batcher has selected the most efficient batch equation, it performs another analysis to determine a "crossover point", i.e., the batch size where batch verification becomes more efficient than individual verification. This analysis is done by counting the number of operations required as a function of the batch size. These operations also include group operations, pairings, hashes, as well as random element generation. It then combines this operation count with the database of average operation times to compute the crossover point.

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 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).[11]

Theorem 3.1 (Security of AutoBatch) *Let an AutoBatch algorithm be generalized as any algorithm that transforms an individual pairing-based signature verification test with perfect correctness into a pairing-based batch verification equation as follows:*

1. *Check the group membership of all input elements, and if no errors, apply Techniques 0a and 1 to the individual verification equation(s) using security parameter λ to obtain a single equation X.*
2. *Apply any of Techniques 2-8 to X to obtain equation X' and set $X := X'$.*
3. *Repeat previous step any number of times to X.*
4. *Check if there are loops in X and the bounds are known, then apply Technique 0b with security parameter λ to X and further attempt Technique 3b if applicable. Then, return X.*

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

Proof. We analyze this proof in three parts. First, after Step 1 (the application of Techniques 0a and 1), there will be one batch equation X (possibly with a loop over it) and it will satisfy the security requirements of Definition 2.2 with error probability $2^{-\lambda}$. These two techniques combine a set of equations (possibly with loops over them) into a single equation (possibly with loops over them) using the small exponents test with security parameter λ. Ferrara et al. [28, Theorem 3.2] prove that this equation will verify if and only if all individual equations from the set verify, except with probability at most $2^{-\lambda}$. By default in AutoBatch, we set $\lambda = 80$.

Second, given a single arbitrary, pairing-based equation X (possibly with a loop over it), we apply one of Techniques 2-8 (in Steps 2 and 3). For each Technique 2-8, 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 0a and 1 were applied, completing the theorem.

It remains to argue that for each Technique 2-8, it is indeed the case that the input and output equations are identical, up to algebraic manipulations. Techniques 2, 3, 4, 6 and 8 follow relatively straightforwardly from the bilinearity of the groups. As an example, consider Technique 3b which claims that $e(a, g) \cdot e(b, g) = e(a \cdot b, g)$, for all $a, b \in \mathbb{G}_1$ and $g \in G_2$ where $a \neq 1 \wedge b \neq 1$. 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 $e(a \cdot a^k, g) = e(a^{k+1}, g) = e(a, g)^{k+1}$. Technique 5 requires only associativity in $\mathbb{G}_T$. Technique 7 pre-computes and caches values instead of re-computing them on the fly.

Finally, we come to Step 4 with a single equation X, possibly having a loop over it. If bounds for the loop are known, then this step unrolls the loop using Technique 0b, whereby a loop representing $i = 1$ to t iterations over an equation X_i is replaced logically by the set of those equations $\{X_1, X_2, \ldots, X_t\}$. This set of

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

equations is then combined into a single equation using the small exponents test, as described above. Finally, Technique 3b is applied if applicable; as discussed above, this technique is a simple algebraic manipulation of the equation and does not change it.

An error of 2^{λ} is introduced with each small exponents test in Technique 0a, 1 and then 0b, thus the total batch verification error is at most $3(2^{-\lambda})$. This completes the theorem. □

To offer transparency on how AutoBatch derived any given batch verifier, the Batcher produces both an SDL file and, optionally, a human-readable proof that the resulting batch verifier is as secure as verifying the signatures individually. 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 [36] was produced automatically by AutoBatch.

We show a fragment of this human-readable proof for the Camenisch-Lysyanskaya (CL) scheme [20] in Figure 5. Full proofs for the Hohenberger-Waters (HW) scheme [36], the Camenisch-Lysyanskaya (CL) scheme [20], and the Verifiable Random Functions (VRF) scheme [37] are given in Appendices B, **??**, and C, respectively. In Appendix D, we detail the results of AutoBatch on the Waters09 scheme (derived from the Waters Dual-System IBE of [66]); because this scheme has a negligible correctness error our automated proof techniques do not directly apply, although we conjecture that the resulting scheme is secure up to an additional negligible error rate. In particular, there will be a negligible chance that the batcher will output reject on a set of valid signatures.

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.

3.4 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 usable Python or C++ source code that can operate on real batch inputs. The SDL representation consists of the individual *and* batch verification equations including logic for the following components:

1. **Group membership tests.** For each element in the signature (and optionally the public key, if the user requests)[12] the membership to 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 re-used within a verification equation. When this happens, the batch verifier can *pre-compute* certain results once, rather than needlessly compute them several times.

3. **Verification method.** 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 crossover point 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

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

e.g,. [28] 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 Charm/C++ representations of the batch verifiers. It would be relatively easy to extend this module to add support for additional languages and settings.

3.5 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 [47]) 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 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 types. 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.[13]

To simplify the parsing, we restrict the subset of Python converted to SDL. In particular, we do not support the use of functional constructs in Python such as lambda functions. Our database currently includes signatures for the following types:

1. All pairings and their parameters and types.
2. All hashes and their parameters and types.
3. All Python dictionaries, their key names, their value names, and their types. Charm makes extensive use of this data structure, so this is important.
4. All constant numbers and strings.

4 Implementation & Performance

Subsequent to our initial publication of the conference version of this work [2], we identified a software bug in the group membership function of Charm v0.42 that affected our results. The results in this paper include the corrections to the affected group membership test which reduces the efficiency gains of batch verification in all our test cases. In particular, there are noticeable reductions in performance for CL [20], Waters09 [66] and HW (with different signers) [36]. Although an optional feature, our membership tests include public

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

Scheme	Type	Model	Ind-Verify	By Hand Batch-Verify	Reference	By AutoBatch Batch-Verify	Techniques
1. Boyen-Lynn-Shacham (BLS) (ss)	S	RO	2η	2	[16]	2	1,2,3a
2. Camenisch et al. (CHP) (same period)	S	RO	3η	3	[19]	3	1,2,3a,5,3a
3. Camenisch-Lysyanskaya (CL) (ss)	S	P	5η	5η	none	**3**	0a,1,2,3b,3b,3a,5,3a
4. Hohenberger-Waters (HW) (ss)	S	P	2η	2η	none	**4**	1,2,3a,8,6,5,3a
5. Hohenberger-Waters (HW)	S	P	2η	2η	none	**4**	1,2,3a,5,3a
6. Waters09 (ss)	S	P	9η	9η	none	**13**	1,2,8,5,3a,6,3b
7. Hess	I	RO	2η	2	[28]	2	1,2,3a
8. Cha-Cheon (ChCh)	I	RO	2η	2	[42]	2	1,2,3a
9. Waters05	I	P	3η	$z + 3$	[19]	$z + 3$	1,2,3a,8,6,5,3a,4,3b
10. Chow-Yiu-Hui (CYH)	IR	RO	2η	2	[28]	2	1,2,3a,2
11. Boyen (same ring)	R	P	$\ell\eta + \ell$	$3\ell + 1$	[28]	$3\ell + 1$	1,2,8,4,3b,8,5,3a
12. Boneh-Boyen-Shacham (BBS)	G	RO	5η	2	[28]	2	1,2,3b,3b,5,3a
13. VRF equations 1,3,4 & 2 (ss)	V	P	$\mathbf{3\eta + 2\ell}$	$\mathbf{3\ell + 1}$	[37]	$\mathbf{\ell + 3}$	0a,3b,1,2,3a,1,2,3a,5,3a,3b,0b,3b
14. ChCh and Hess together	M	RO	2η	4	none	2	0a,1,2,3a,5,3a,3b

Table 2: Digital Signature Schemes used as test cases in AutoBatch. We show a comparison between naive batch verifiers designed by hand or discovered in the literature and ones found by AutoBatch. Scheme names followed by an "ss" were only batched for the same signers; otherwise, different signers were allowed. 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 for the Waters hash [65] and 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 applied to obtain the resulting batch verifier. The **rows in bold** are the schemes where AutoBatch discovered new or improved algorithms. Finally, the *italicized row* represents the ability of AutoBatch to construct batch verifiers for different signature types. This is an instance of cross-scheme batching and we compare it to batching naively per signature type.

keys to reflect the worst case performance of batch verification without invalid signatures in the batch. See Figure 6 for the new graphs.

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 [59].[14] 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.

[14]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.

	Approx. Signature Size		MIRACL w/ BN256		RELIC w/ BN256	
	MNT160	BN256	Individual	Batched*	Individual	Batched*
Signatures						
BLS [16] (same signer)	160 bits	256 bits	26.6 ms	2.2 ms	11.9 ms	1.5 ms
CHP [19] (same time period)	160 bits	256 bits	46.1 ms	7.2 ms	24.0 ms	7.8 ms
HW [36] (same signer)	320 bits	512 bits	40.5 ms	4.7 ms	22.4 ms	3.0 ms
HW [36] (diff signer)	320 bits	512 bits	40.5 ms	61.1 ms	22.4 ms	29.2 ms
Waters09 [66, §6.1] (same signer)	6240 bits	6912 bits	153.2 ms	33.1 ms	93.7 ms	44.2 ms
CL [20] (same signer)	480 bits	768 bits	72.0 ms	15.9 ms	34.6 ms	18.0 ms
ID-Based Signatures						
Hess [35]	1120 bits	3328 bits	32.7 ms	22.0 ms	17.1 ms	8.4 ms
ChCh [24]	320 bits	512 bits	27.5 ms	4.6 ms	12.6 ms	2.4 ms
Waters05 [65]	480 bits	768 bits	45.3 ms	11.8 ms	21.5 ms	11.0 ms
Group, Ring and ID-based Ring Signatures						
BBS [13] Group signature	2400 bits	5376 bits	99.9 ms	31.2 ms	63.9 ms	18.7 ms
Boyen [18] Ring signature, 3-member ring	960 bits	1536 bits	64.2 ms	15.0 ms	41.5 ms	9.8 ms
CYH [27] Ring signature, 10-member ring	1760 bits	2816 bits	34.2 ms	22.3 ms	20.7 ms	16.2 ms
VRFs						
HW VRF [Hohenberger-Waters 2010] (same signer, $\ell = 8$)	2240 bits	5120 bits	251.4 ms	36.1 ms	112.5 ms	18.3 ms
Combinations						
ChCh + Hess	1440 bits	3840 bits	55.6 ms	26.2 ms	25.7 ms	10.4 ms

*Verification time *per signature* when batching 100 signatures.

Table 3: Cryptographic overhead and verification time for all of the pairing-based signatures in an alternative implementation of AutoBatch. RELIC is faster on 12 of 14 schemes, but MIRACL is better on CL and Waters09. We speculate that this is because modular exponentiation in $\mathbb{G}_1$ and $\mathbb{G}_2$ is slightly slower in RELIC compared to MIRACL. Since RELIC is an actively developed library, we believe this issue can be addressed in future versions. In the case of HW (with different signers), individual verification outperforms batch verification in both libraries because batch time is dominated by group membership tests.

We instantiated all of our cryptographic implementations using a 160-bit MNT elliptic curve and a 256-bit Barreto-Naehrig (BN) curve provided with MIRACL. Results are shown in Table 3 and Figure 6.

A note on the library. We chose MIRACL because it is mature and well supported. However, some research libraries like RELIC [4] 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 BN256 curve, we observed a 6-to-1 differential between pairings and operations in $\mathbb{G}_1$. Our main results do hold in this setting, and in fact improve the overall performance in that we can process a higher number of signatures with batch verification. We provide the details of this alternative version of AutoBatch and a complete comparison against the BN256 curve MIRACL implementation in Table 3.

4.2 Signature Schemes Used as Test Cases and Summary of the Results

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 and group signatures, as well as verifiable random functions. 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. We summarize AutoBatch's performance on existing schemes in Table 2.

In eight cases, the batching algorithm output by AutoBatch matched the prior best known result. In the remaining cases, AutoBatch provided a faster algorithm. We now describe these cases in more detail.

We briefly recall the verification equations in VRF [37]. The public key is represented by $\hat{U}, U, g_1, g_2, h$, the signature is represented by $y, \pi = \pi_0 \pi_1, \ldots, \pi_\ell$, and the message is $x = x_1, \ldots, x_\ell$, where ℓ denotes the number of bits in the VRF input. The equations are as follows:

1. $e(\pi_1, g_2) \stackrel{?}{=} e(g_1^{(1-x_1)} \cdot U_1^{x_1}, \hat{U})$
2. for $t = 2$ to ℓ it holds: $e(\pi_t, g_2) \stackrel{?}{=} e(\pi_{t-1}^{(1-x_t)}, g_2) \cdot e(\pi_{t-1}^{x_t}, U_t)$
3. $e(\pi_0, g_2) \stackrel{?}{=} e(\pi_l, U_0)$
4. $e(\pi_0, h) \stackrel{?}{=} y$

AutoBatch first realized a batching algorithm for the VRF [37] that takes only two-thirds the time of the one provided in [37] (or $2\ell + 2$ total pairings). Then, after 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 the 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 implemented this loop unrolling technique, incorporated it into AutoBatch and automatically applied it to VRF to obtain $\ell + 3$ pairings. The VRF batching algorithm and proof appear in Appendix C.

In test case 14 shown in Table 2 (ChCh [24] and Hess [35] together), 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 [24] signatures and Hess [35] 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 [24] and Hess [35] batch, this cuts 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 [36], we assume that each public key includes the precomputed values as suggested in [36, 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 [66] using the technique described by Naor [14]. Because the decryption algorithm of this IBE scheme has a negligibly small correctness error, the resulting signature scheme also has a negligible correctness error. That is, there is a small chance that a valid signature will be rejected by the verification test. Although this means that our automated proof techniques do not immediately apply, we still wanted to run the program on this complicated test case to see how efficient of a candidate batching scheme it could produce. The details of these batching algorithms appear in Appendices B and D respectively.

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 and C++ 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 Table 4.

To obtain our microbenchmarks, we ran AutoBatch on several exemplary pairing-based schemes as listed in Table 2. 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,[15]

[15]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.

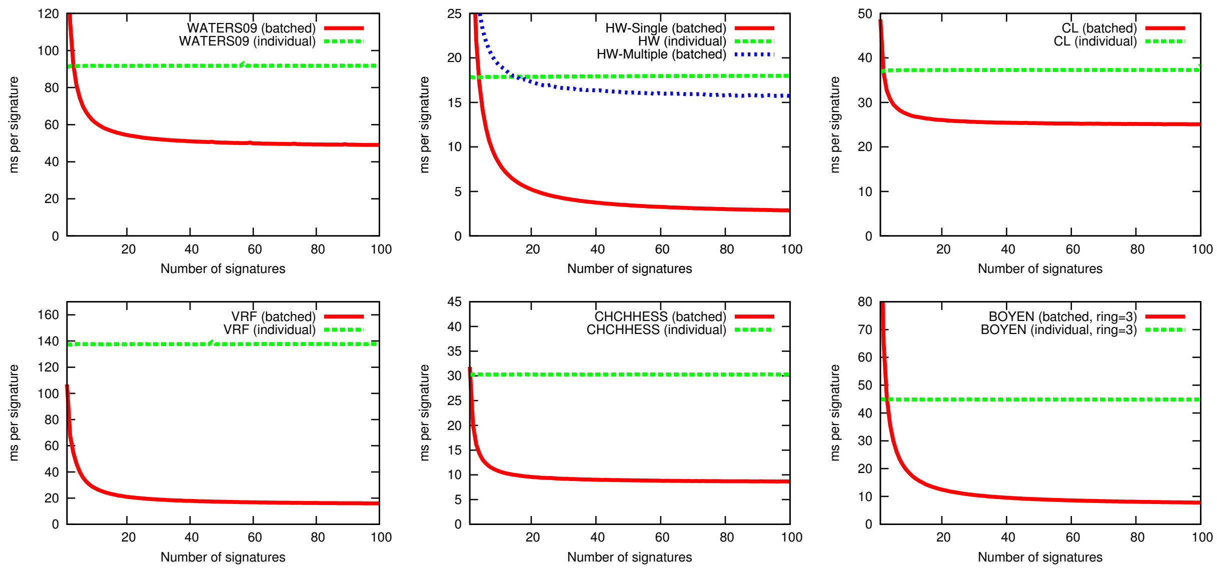


Figure 6: Signature scheme microbenchmarks for Waters09 [66], HW [36] and CL [20] public-key signatures (same signer), the VRF [37] (with block size of 8), combined verification of ChCh+Hess IBS [24, 35], and Boyen ring signature (3 signer ring) [18]. Per-signature times were computed by dividing total batch verification time by the number of signatures verified. All trials were conducted with 10 iterations and were instantiated using a 160-bit MNT elliptic curve. 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.

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., [28, 42, 50, 51, 69]. 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 car-to-car communications base station.

Our server is designed to process as many signatures as possible, and is limited only by its computational resources.[16] 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.[17] This approach can lead to some transient errors (batches that require significantly more or less than one

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

[17]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.

Process	BLS	CHP	CL	HW-diff	Waters09	Waters05	ChCh/Hess	CYH	Boyen	BBS	VRF
Batcher	103.1	90.1	295.2	126.1	578.9	1859.2	160.1	101.2	545.1	443.5	419.5
Partial-Codegen	124.3	171.7	152.2	242.3	361.6	291.2	162.0	242.8	321.2	315.1	251.2
Full-Codegen	491.7	757.8	785.9	1481.6	3405.8	1507.1	798.6	876.3	1233.5	1998.3	2748.3

Table 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 the crossover point between individual and batch verification. The Partial-Codegen time represents the generation of the batch verifier code from a partial SDL description and Charm implementation of the scheme in Python. The Full-Codegen time represents the generation of code from a full SDL description only. 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.

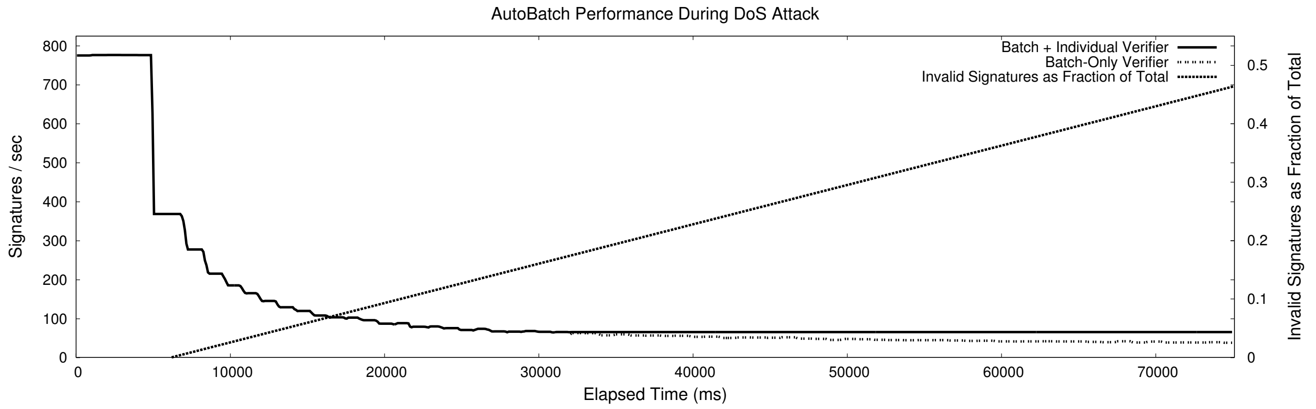


Figure 7: Simulated service denial attacks against a batch verifier (BLS signatures, single signer). The "Invalid Signatures as Fraction of Total" line (right scale) shows the fraction of invalid signatures in the stream. Batcher throughput is measured in signatures per second (left scale). The "Batch-Only Verifier" line depicts a standard batch verifier. The solid line is a batch verifier that automatically switches to *individual* verification when batching becomes suboptimal.

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 Section 3, each of our batch verifiers incorporates a recursive divide-and-conquer strategy for identifying these invalid signatures, which is borrowed from Law and Matt [42]. 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,[18] 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%.

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

A switch to individual verification. Our experiments indicate 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.[19] 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 AutoBatch Toolkit

The AutoBatch source code and test cases described in this paper are publicly available in the github repository at `https://github.com/JHUISI/auto-tools`.

6 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 [23, 60]. 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 [10] 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., [42, 50, 51, 69]. We are continuously on the lookout for more efficient means of computing in bilinear groups. Future versions of AutoBatch will support MIRACL's API for computing "multipairings" (efficient products of multiple bilinear pairings). It would be interesting to understand how this and future inclusions may impact performance.

References

[1] Akinyele, J. A., Garman, C., Miers, I., Pagano, M. W., Rushanan, M., Green, M., and Rubin, A. D. Charm: a framework for rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering 3*, 2 (2013), 111–128.

[2] Akinyele, J. A., Green, M., Hohenberger, S., and Pagano, M. W. Machine-generated algorithms, proofs and software for the batch verification of digital signature schemes. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 474–487.

[3] 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* (2010), ESORICS'10, Springer-Verlag, pp. 151–167.

[19] 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.

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

[5] Bacelar Almeida, J., Barbosa, M., Bangerter, E., Barthe, G., Krenn, S., and Zanella Béguelin, S. Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), CCS '12, ACM, pp. 488–500.

[6] 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* (2008), SP '08, IEEE Computer Society, pp. 202–215.

[7] 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* (2010), EuroPKI'09, Springer-Verlag, pp. 67–82.

[8] 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.

[9] 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* (2007), OTM'07, Springer-Verlag, pp. 1785–1802.

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

[11] 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.

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

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

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

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

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

[17] Boyd, C., and Pavlovski, C. Attacking and repairing batch verification schemes. In *Advances in Cryptology – ASIACRYPT '00* (2000), vol. 1976, pp. 58–71.

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

[19] 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.`

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

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

[27] 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.

[28] 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.

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

[30] Fournet, C., Kohlweiss, M., Danezis, G., and Luo, Z. ZQL: A compiler for privacy-preserving data processing. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (2004), SSYM'04, USENIX Association, pp. 20–20.

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

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

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

[34] 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* (2010), CCS '10, ACM, pp. 451–462.

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

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

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

[38] 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.

[39] 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.

[40] 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)* (2008), ISA '08, IEEE Computer Society, pp. 20–25.

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

[42] 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.

[43] 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.

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

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

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

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

[48] 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* (2003), CCS '03, ACM, pp. 210–219.

[49] 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* (2004), SSYM'04, USENIX Association, pp. 20–20.

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

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

[52] 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* (2010), USENIX Security'10, USENIX Association, pp. 13–13.

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

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

[55] 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.

[56] 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* (2010), Pairing'10, Springer-Verlag, pp. 207–224.

[57] 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* (2004), AINA '04, IEEE Computer Society, pp. 400–.

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

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

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

[61] 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.

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

[63] 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* (2001), CAV '01, Springer-Verlag, pp. 241–245.

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

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

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

[67] Waters, B. Dual system encryption: Realizing fully secure ibe and hibe under simple assumptions. Cryptology ePrint Archive, Report 2009/385, 2009. http://eprint.iacr.org/.

[68] 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.

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

[70] 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.

[71] 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.

A Machine-Generated Batch Verification Equations

In Figure 8, we provide the final batch verification equations output by AutoBatch for each of the signature schemes tested.

B A machine-generated proof for HW

The following proof was automatically generated by the Batcher while processing the HW *signature scheme [36]. This execution allows signatures on different signing keys.*

B.1 Definitions

This document contains a proof that HW.BatchVerify is a valid batch verifier for the signature scheme HW. Let U, V, D, g, w, z, h be values drawn from the key and/or parameters, and $M, \sigma_1, \sigma_2, r, i$ represent a message (or message hash) and signature. The individual verification equation HW.Verify is:

$$e(\sigma_1, g) \stackrel{?}{=} U^M \cdot V^r \cdot D \cdot e(\sigma_2, w^{\lceil \lg(i) \rceil} \cdot z^i \cdot h)$$

Let η be the number of signatures in a batch, and $\delta_1, \ldots \delta_\eta \in [1, 2^\lambda - 1]$ be a set of random exponents chosen by the verifier. The batch verification equation HW.BatchVerify is:

$$e(\prod_{z=1}^{\eta} \sigma_{z,1}^{\delta_z}, g) \stackrel{?}{=} \prod_{z=1}^{\eta} U_z{}^{M_z \cdot \delta_z} \cdot \prod_{z=1}^{\eta} V_z{}^{r_z \cdot \delta_z} \cdot \prod_{z=1}^{\eta} D_z{}^{\delta_z} \cdot e(\prod_{z=1}^{\eta} \sigma_{z,2}^{\delta_z \cdot \lceil \lg(i_z) \rceil}, w) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,2}^{\delta_z \cdot i_z}, z) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,2}^{\delta_z}, h)$$

We will now formally define a batch verifier and demonstrate that HW.BatchVerify is a secure batch verifier for the HW signature scheme.

Definition B.1 (Pairing-based Batch Verifier) *Let* BSetup$(1^\tau) \to (q, g, \mathbb{G}, \mathbb{G}_T, e)$. *For each* $j \in [1, \eta]$, *where* $\eta \in poly(\tau)$, *let* $X^{(j)}$ *be a generic pairing-based claim and let* Verify *be a pairing based verifier. We define* pairing-based batch verifier *for* Verify *a probabilistic poly(τ)-time algorithm which outputs* accept *if* $X^{(j)}$ *holds for all* $j \in [1, \eta]$ *whereas it outputs* reject *if* $X^{(j)}$ *does not hold for any* $j \in [1, \eta]$ *except with negligible probability.*

Theorem B.2 HW.BatchVerify *is a batch verifier for the* HW *signature scheme.*

Scheme	Batch Verification Equation output by AutoBatch
Signatures	
BLS [16] (same signer)	$e(\prod_{z=1}^{\eta} h_z^{\delta_z}, pk) \overset{?}{=} e(\prod_{z=1}^{\eta} sig_z^{\delta_z}, g)$
CHP [19] (same time period)	$e(\prod_{z=1}^{\eta} sig_z^{\delta_z}, g) \overset{?}{=} e(a, \prod_{z=1}^{\eta} pk_z^{\delta_z}) \cdot e(h, \prod_{z=1}^{\eta} pk_z^{b_z \cdot \delta_z})$
HW [36] (same signer)	$e(\prod_{z=1}^{\eta} {\sigma_1}_z^{\delta_z}, g) \overset{?}{=} U^{\sum_{z=1}^{\eta} M_z \cdot \delta_z} \cdot V^{\sum_{z=1}^{\eta} r_z \cdot \delta_z} \cdot D^{\sum_{z=1}^{\eta} \delta_z}$
	$\cdot e(\prod_{z=1}^{\eta} {\sigma_2}_z^{\lg(i_z) \cdot \delta_z}, w) \cdot e(\prod_{z=1}^{\eta} {\sigma_2}_z^{i_z \cdot \delta_z}, z) \cdot e(\prod_{z=1}^{\eta} {\sigma_2}_z^{\delta_z}, h)$
HW [36] (different signers)	$e(\prod_{z=1}^{\eta} \sigma_{z,1}^{\delta_z}, g) \overset{?}{=} \prod_{z=1}^{\eta} {U_z}^{M_z \cdot \delta_z} \cdot \prod_{z=1}^{\eta} {V_z}^{r_z \cdot \delta_z}$
	$\cdot \prod_{z=1}^{\eta} {D_z}^{\delta_z} \cdot e(\prod_{z=1}^{\eta} \sigma_{z,2}^{\delta_z \cdot \lceil \lg(i) \rceil_z}, w) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,2}^{\delta_z \cdot i_z}, z) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,2}^{\delta_z}, h)$
Waters09 [66] (same signer)	$e({g_1}^{b}, \prod_{z=1}^{\eta} \sigma_{z,1}^{s_z \cdot \delta_z}) \cdot e({g_1}^{b \cdot a_1}, \prod_{z=1}^{\eta} \sigma_{z,2}^{s_{z,1} \cdot \delta_z})$
	$\cdot e({g_1}^{a_1}, \prod_{z=1}^{\eta} \sigma_{z,3}^{s_{z,1} \cdot \delta_z}) \cdot e({g_1}^{b \cdot a_2}, \prod_{z=1}^{\eta} \sigma_{z,4}^{s_{z,2} \cdot \delta_z})$
	$\cdot e({g_1}^{a_2}, \prod_{z=1}^{\eta} \sigma_{z,5}^{s_{z,2} \cdot \delta_z}) \overset{?}{=} e(\prod_{z=1}^{\eta} \sigma_{z,6}^{\delta_z \cdot s_{z,1}}, \tau_1)$
	$\cdot e(\prod_{z=1}^{\eta} \sigma_{z,6}^{\delta_z \cdot s_{z,2}}, \tau_2) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot s_{z,1}}, {\tau_1}^{b})$
	$\cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot s_{z,2}}, {\tau_2}^{b}) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{(\delta_z \cdot -t_z + \theta_z \cdot \delta_z \cdot tag_{z,c} \cdot t_z)}, w)$
	$\cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot M_z \cdot t_z}, u) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot t_z}, h)$
	$\cdot e(g_1, \prod_{z=1}^{\eta} \sigma_{z,K}^{-t_z \cdot \theta_z \cdot \delta_z}) \cdot A^{\sum_{z=1}^{\eta} s_{z,2} \cdot \delta_z}$
CL [20] (same signer)	$e(g, \prod_{z=1}^{\eta} {b_z}^{\delta_{z,1}} \cdot {c_z}^{\delta_{z,2}}) \cdot e(Y, \prod_{z=1}^{\eta} {a_z}^{-\delta_{z,1}}) \overset{?}{=} e(X, \prod_{z=1}^{\eta} {a_z}^{\delta_{z,2}} \cdot {b_z}^{m_z \cdot \delta_{z,2}})$
ID-based Signatures	
Hess [35]	$e(\prod_{z=1}^{\eta} {S_2}_z^{\delta_z}, g_2) \overset{?}{=} e(\prod_{z=1}^{\eta} pk_z^{a_z \cdot \delta_z}, P_{pub}) \cdot \prod_{z=1}^{\eta} {S_1}_z^{\delta_z}$
ChCh [24]	$e(\prod_{z=1}^{\eta} {S_2}_z^{\delta_z}, g_2) \overset{?}{=} e(\prod_{z=1}^{\eta} ({S_1}_z \cdot pk^{a_z})^{\delta_z}, P_{pub})$
Waters05 [65]	$e(\prod_{z=1}^{\eta} {S_1}_z^{\delta_z}, g_2) \cdot e(\prod_{z=1}^{\eta} {S_2}_z^{\delta_z}, \hat{u_1\prime}) \cdot \prod_{i=1}^{l} e(\prod_{z=1}^{\eta} {S_2}_z^{\delta_z \cdot k_{i,z}} \cdot {S_3}_z^{\delta_z \cdot m_{i,z}}, \hat{u}_i)$
	$\cdot e(\prod_{z=1}^{\eta} {S_3}_z^{\delta_z}, \hat{u_2\prime}) \overset{?}{=} e(g_1, g_2)^{\sum_{z=1}^{\eta} \delta_z}$
Group, Ring, and ID-based Ring Signatures	
BBS [13]	$e(\prod_{z=1}^{\eta} T_{z,3}^{s_{z,x} \cdot \delta_z} \cdot h^{(-s_{z,\gamma 1} - s_{z,\gamma 2}) \cdot \delta_z} \cdot g_1^{-c_z \cdot \delta_z}, g_2)$
	$\cdot e(h^{\sum_{z=1}^{\eta} (-s_{z,\alpha} - s_{z,\beta}) \cdot \delta_z} \cdot \prod_{z=1}^{\eta} T_{z,3}^{c_z \cdot \delta_z}, w) \overset{?}{=} \prod_{z=1}^{\eta} R_{z,3}^{\delta_z}$
Boyen [18] (same ring)	$\prod_{y=1}^{l} e(\prod_{z=1}^{\eta} {S_{y,z}}^{\delta_z}, \hat{A}_y) \cdot e(\prod_{z=1}^{\eta} {S_{y,z}}^{m_{y,z} \cdot \delta_z}, \hat{B}_y) \cdot e(\prod_{z=1}^{\eta} {S_{y,z}}^{t_{y,z} \cdot \delta_z}, \hat{C}_y) \overset{?}{=} \prod_{z=1}^{\eta} D^{\delta_z}$
CYH [27]	$e(\prod_{z=1}^{\eta} \prod_{y=1}^{l} u_{y,z} \cdot pk_{y,z}^{h_{y,z}}{}^{\delta_z}, P) \overset{?}{=} e(\prod_{z=1}^{\eta} S_z^{\delta_z}, g)$
VRFs	
HW VRF [37] (same signer)	$e(\prod_{z=1}^{\eta} g_1^{(1-x_1) \cdot \delta_{z,2}} \cdot U_1^{x_1 \cdot \delta_{z,2}}, \hat{U}) \cdot e(\prod_{z=1}^{\eta} \pi_{z,1}^{-\delta_{z,2}} \cdot \pi_{z,2}^{\delta_{z,3}} \cdot \pi_{z,1}^{(1-x_{z,2}) \cdot -\delta_{z,3}}$
	$\cdot \pi_{z,3}^{-\delta_{z,4}} \cdot \pi_{z,2}^{(1-x_{z,3}) \cdot -\delta_{z,4} \cdot -1} \cdot \pi_{z,4}^{-\delta_{z,5}} \cdot \pi_{z,3}^{(1-x_{z,4}) \cdot -\delta_{z,5} \cdot -1}$
	$\cdot \pi_{z,5}^{-\delta_{z,6}} \cdot \pi_{z,4}^{(1-x_{z,5}) \cdot -\delta_{z,6} \cdot -1} \cdot \pi_{z,6}^{-\delta_{z,7}} \cdot \pi_{z,5}^{(1-x_{z,6}) \cdot -\delta_{z,7} \cdot -1} \cdot \pi_{z,7}^{-\delta_{z,8}} \cdot \pi_{z,6}^{(1-x_{z,7}) \cdot -\delta_{z,8} \cdot -1}$
	$\cdot \pi_{z,8}^{-\delta_{z,9}} \cdot \pi_{z,7}^{(1-x_{z,8}) \cdot -\delta_{z,9} \cdot -1}, g_2) \overset{?}{=}$
	$e(\prod_{z=1}^{\eta} \pi_{z,l}^{\delta_{z,1}}, U_0) \cdot \prod_{z=1}^{\eta} {y_z}^{\delta_{z,1}} \cdot e(\prod_{z=1}^{\eta} \pi_{z,0}^{-\delta_{z,1}}, g_2 \cdot h)$
	$\cdot e(\prod_{z=1}^{\eta} \pi_{z,1}^{x_{z,2} \cdot \delta_{z,3}}, U_2) \cdot e(\prod_{z=1}^{\eta} \pi_{z,2}^{x_{z,3} \cdot \delta_{z,4} \cdot -1}, U_3) \cdot e(\prod_{z=1}^{\eta} \pi_{z,3}^{x_{z,4} \cdot \delta_{z,5} \cdot -1}, U_4)$
	$\cdot e(\prod_{z=1}^{\eta} \pi_{z,4}^{x_{z,5} \cdot \delta_{z,6} \cdot -1}, U_5) \cdot e(\prod_{z=1}^{\eta} \pi_{z,5}^{x_{z,6} \cdot \delta_{z,7} \cdot -1}, U_6) \cdot e(\prod_{z=1}^{\eta} \pi_{z,6}^{x_{z,7} \cdot \delta_{z,8} \cdot -1}, U_7)$
	$\cdot e(\prod_{z=1}^{\eta} \pi_{z,7}^{x_{z,8} \cdot \delta_{z,9} \cdot -1}, U_8)$ for block size of 8
Combinations	
ChCh + Hess	$e(\prod_{z=1}^{\eta} {pk_z}^{ah_z \cdot \delta_{z,1}} \cdot Sc_{z,1}^{-\delta_{z,2}} \cdot {pk_z}^{ac_z \cdot -\delta_{z,2}}, P_{pub}) \cdot \prod_{z=1}^{\eta} Sh_{z,1}^{\delta_{z,1}} \cdot$
	$e(\prod_{z=1}^{\eta} Sh_{z,2}^{-\delta_{z,1}} \cdot Sc_{z,2}^{\delta_{z,2}}, g_2) \overset{?}{=} 1$

Figure 8: These are the final batch verification equations output by AutoBatch. Due to space, we do not include the full schemes or further describe the elements of the signature or our shorthand for them, such as setting $h = H(M)$ in BLS. However, a reader could retrace our steps by applying the techniques in Section 3 to the original verification equation in the order specified in Figure 2. "Combined signatures" refers to the combined batching of multiple signature verification equations that share algebraic structure.

B.2 Proof

Proof. Via a series of steps, we will show that if HW is a secure signature scheme, then BatchVerify is a secure batch verifier. Recall our batch verification software will perform a group membership test to ensure that each group element of the signature is a member of the proper subgroup, so here will we assume this fact. We begin with the original verification equation.

$$e(\sigma_1, g) \overset{?}{=} U^M \cdot V^r \cdot D \cdot e(\sigma_2, w^{\lceil \lg(i) \rceil} \cdot z^i \cdot h) \tag{1}$$

Step 1: Combine η signatures (technique 1):

$$\prod_{z=1}^{\eta} e(\sigma_{z,1}, g) \overset{?}{=} \prod_{z=1}^{\eta} U_z{}^{M_z} \cdot V_z{}^{r_z} \cdot D_z \cdot e(\sigma_{z,2}, w^{\lceil \lg(i_z) \rceil} \cdot z^{i_z} \cdot h) \tag{2}$$

Step 2: Apply the small exponents test, using exponents $\delta_1, \ldots \delta_\eta \in \left[1, 2^\lambda - 1\right]$:

$$\prod_{z=1}^{\eta} e(\sigma_{z,1}, g)^{\delta_z} \overset{?}{=} \prod_{z=1}^{\eta} U_z{}^{M_z \cdot \delta_z} \cdot \prod_{z=1}^{\eta} V_z{}^{r_z \cdot \delta_z} \cdot \prod_{z=1}^{\eta} D_z{}^{\delta_z} \cdot \prod_{z=1}^{\eta} e(\sigma_{z,2}, w^{\lceil \lg(i_z) \rceil} \cdot z^{i_z} \cdot h)^{\delta_z} \tag{3}$$

Step 3: Move exponent(s) inside the pairing (technique 2):

$$\prod_{z=1}^{\eta} e(\sigma_{z,1}^{\delta_z}, g) \overset{?}{=} \prod_{z=1}^{\eta} U_z{}^{M_z \cdot \delta_z} \cdot \prod_{z=1}^{\eta} V_z{}^{r_z \cdot \delta_z} \cdot \prod_{z=1}^{\eta} D_z{}^{\delta_z} \cdot \prod_{z=1}^{\eta} e(\sigma_{z,2}^{\delta_z}, w^{\lceil \lg(i_z) \rceil} \cdot z^{i_z} \cdot h) \tag{4}$$

Step 4: Move products inside pairings to reduce η pairings to 1 (technique 3a):

$$e(\prod_{z=1}^{\eta} \sigma_{z,1}^{\delta_z}, g) \overset{?}{=} \prod_{z=1}^{\eta} U_z{}^{M_z \cdot \delta_z} \cdot \prod_{z=1}^{\eta} V_z{}^{r_z \cdot \delta_z} \cdot \prod_{z=1}^{\eta} D_z{}^{\delta_z} \cdot \prod_{z=1}^{\eta} e(\sigma_{z,2}^{\delta_z}, w^{\lceil \lg(i_z) \rceil}) \cdot e(\sigma_{z,2}^{\delta_z}, z^{i_z}) \cdot e(\sigma_{z,2}^{\delta_z}, h) \tag{5}$$

Step 5: Distribute products (technique 5):

$$e(\prod_{z=1}^{\eta} \sigma_{z,1}^{\delta_z}, g) \overset{?}{=} \prod_{z=1}^{\eta} U_z{}^{M_z \cdot \delta_z} \cdot \prod_{z=1}^{\eta} V_z{}^{r_z \cdot \delta_z} \cdot \prod_{z=1}^{\eta} D_z{}^{\delta_z} \cdot \prod_{z=1}^{\eta} e(\sigma_{z,2}^{\delta_z}, w^{\lceil \lg(i_z) \rceil}) \cdot \prod_{z=1}^{\eta} e(\sigma_{z,2}^{\delta_z}, z^{i_z}) \cdot \prod_{z=1}^{\eta} e(\sigma_{z,2}^{\delta_z}, h) \tag{6}$$

Step 6: Move products inside pairings to reduce η pairings to 1 (technique 3a):

$$e(\prod_{z=1}^{\eta} \sigma_{z,1}^{\delta_z}, g) \overset{?}{=} \prod_{z=1}^{\eta} U_z{}^{M_z \cdot \delta_z} \cdot \prod_{z=1}^{\eta} V_z{}^{r_z \cdot \delta_z} \cdot \prod_{z=1}^{\eta} D_z{}^{\delta_z} \cdot e(\prod_{z=1}^{\eta} \sigma_{z,2}^{\delta_z \cdot \lceil \lg(i_z) \rceil}, w) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,2}^{\delta_z \cdot i_z}, z) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,2}^{\delta_z}, h) \tag{7}$$

Steps 1 and 2 form the Combination Step in [28], which was proven to result in a secure batch verifier in [28, Theorem 3.2]. We observe that the remaining steps are merely reorganizing terms within the same equation. Hence, the final verification equation (7) is also batch verifier for HW. □

C A Machine-Generated Proof for VRF

The following proof was automatically generated by the Batcher while processing the VRF *signature scheme [37]. This execution was restricted to signatures on a single signing key.*

C.1 Definitions

This document contains a proof that VRF.BatchVerify is a valid batch verifier for the signature scheme VRF. Let $\hat{U}, U, g_1, g_2, h$ be values drawn from the key and/or parameters, and x, π, y represent a message (or message hash) and signature. The ℓ parameter represents the ℓ-bit input size of VRF and varies in practice. We have shown an example of $\ell = 8$ to simplify the proof. The individual verification equation VRF.Verify is:

$$e(\pi_1, g_2) \stackrel{?}{=} e(g_1^{(1-x_1)} \cdot U_1^{x_1}, \hat{U}) \text{ and } e(\pi_0, g_2) \stackrel{?}{=} e(\pi_l, U_0) \text{ and } e(\pi_0, h) \stackrel{?}{=} y \text{ and}$$
$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } e(\pi_t, g_2) \stackrel{?}{=} e(\pi_{t-1}^{(1-x_t)}, g_2) \cdot e(\pi_{t-1}^{x_t}, U_t)$$

Let η be the number of signatures in a batch, and $\delta_{1,i}, \ldots \delta_{\eta,i} \in [1, 2^\lambda - 1]$ be a set of random exponents chosen by the verifier. Since the input size of $\ell = 8$, then $i = 9$. The batch verification equation for VRF is:

VRFBatchVerify:

$$e(\prod_{z=1}^{\eta} g_1^{(1-x_1)\cdot\delta_{z,2}} \cdot U_1^{x_1\cdot\delta_{z,2}}, \hat{U}) \cdot e(\prod_{z=1}^{\eta} \pi_{z,1}^{-\delta_{z,2}} \cdot \pi_{z,2}^{\delta_{z,3}} \cdot \pi_{z,1}^{(1-x_{z,2})\cdot-\delta_{z,3}} \cdot \pi_{z,3}^{-\delta_{z,4}} \cdot \pi_{z,2}^{(1-x_{z,3})\cdot-\delta_{z,4}\cdot-1} \cdot \pi_{z,4}^{-\delta_{z,5}} \cdot \pi_{z,3}^{(1-x_{z,4})\cdot-\delta_{z,5}\cdot-1}$$
$$\cdot \pi_{z,5}^{-\delta_{z,6}} \cdot \pi_{z,4}^{(1-x_{z,5})\cdot-\delta_{z,6}\cdot-1} \cdot \pi_{z,6}^{-\delta_{z,7}} \cdot \pi_{z,5}^{(1-x_{z,6})\cdot-\delta_{z,7}\cdot-1} \cdot \pi_{z,7}^{-\delta_{z,8}} \cdot \pi_{z,6}^{(1-x_{z,7})\cdot-\delta_{z,8}\cdot-1} \cdot \pi_{z,8}^{-\delta_{z,9}} \cdot \pi_{z,7}^{(1-x_{z,8})\cdot-\delta_{z,9}\cdot-1}, g_2) \stackrel{?}{=}$$
$$e(\prod_{z=1}^{\eta} \pi_{z,l}^{\delta_{z,1}}, U_0) \cdot \prod_{z=1}^{\eta} y_z^{\delta_{z,1}} \cdot e(\prod_{z=1}^{\eta} \pi_{z,0}^{-\delta_{z,1}}, g_2 \cdot h) \cdot e(\prod_{z=1}^{\eta} \pi_{z,1}^{x_{z,2}\cdot\delta_{z,3}}, U_2) \cdot e(\prod_{z=1}^{\eta} \pi_{z,2}^{x_{z,3}\cdot\delta_{z,4}\cdot-1}, U_3) \cdot e(\prod_{z=1}^{\eta} \pi_{z,3}^{x_{z,4}\cdot\delta_{z,5}\cdot-1}, U_4)$$
$$\cdot e(\prod_{z=1}^{\eta} \pi_{z,4}^{x_{z,5}\cdot\delta_{z,6}\cdot-1}, U_5) \cdot e(\prod_{z=1}^{\eta} \pi_{z,5}^{x_{z,6}\cdot\delta_{z,7}\cdot-1}, U_6) \cdot e(\prod_{z=1}^{\eta} \pi_{z,6}^{x_{z,7}\cdot\delta_{z,8}\cdot-1}, U_7) \cdot e(\prod_{z=1}^{\eta} \pi_{z,7}^{x_{z,8}\cdot\delta_{z,9}\cdot-1}, U_8)$$

We will now formally define a batch verifier and demonstrate that VRF.BatchVerify is a secure batch verifier for the VRF signature scheme.

Definition C.1 (Pairing-based Batch Verifier) *Let* BSetup$(1^\tau) \rightarrow (q, g, \mathbb{G}, \mathbb{G}_T, e)$. *For each* $j \in [1, \eta]$, *where* $\eta \in poly(\tau)$, *let* $X^{(j)}$ *be a generic pairing-based claim and let* Verify *be a pairing based verifier. We define* pairing-based batch verifier *for* Verify *a probabilistic poly(τ)-time algorithm which outputs* accept *if* $X^{(j)}$ *holds for all* $j \in [1, \eta]$ *whereas it outputs* reject *if* $X^{(j)}$ *does not hold for any* $j \in [1, \eta]$ *except with negligible probability.*

Theorem C.2 VRF BatchVerify *is a batch verifier for the* VRF *signature scheme.*

C.2 Proof

Proof. Via a series of steps, we will show that if VRF is a secure signature scheme, then BatchVerify is a secure batch verifier. Recall our batch verification software will perform a group membership test to ensure that each group element of the signature is a member of the proper subgroup, so here will we assume this fact. We begin with the original verification equation.

$$e(\pi_1, g_2) \stackrel{?}{=} e(g_1^{(1-x_1)} \cdot U_1^{x_1}, \hat{U}) \text{ and } e(\pi_0, g_2) \stackrel{?}{=} e(\pi_l, U_0) \text{ and } e(\pi_0, h) \stackrel{?}{=} y \text{ and}$$
$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } e(\pi_t, g_2) \stackrel{?}{=} e(\pi_{t-1}^{(1-x_t)}, g_2) \cdot e(\pi_{t-1}^{x_t}, U_t)$$

EQ1 Step 1: Consolidate the verification equations (technique 0a), merge pairings with common first or second argument (technique 3b), and apply the small exponents test as follows: For each of the $z = 1$ to η signatures, choose random $\delta_1, \delta_2 \in [1, 2^\lambda - 1]$ and compute for each equation:

$$e(g_1^{(1-x_1)} \cdot U_1^{x_1}, \hat{U})^{\delta_2} \cdot e(\pi_1, g_2)^{-\delta_2} \stackrel{?}{=} e(\pi_l, U_0)^{\delta_1} \cdot y^{\delta_1} \cdot e(\pi_0, g_2 \cdot h)^{-\delta_1} \quad (8)$$

EQ1 Step 2: Combine η signatures (technique 1), move exponent(s) inside pairing (technique 2):

$$\prod_{z=1}^{\eta} e(g_1^{(1-x_1)\cdot\delta_{z,2}} \cdot U_1^{x_1\cdot\delta_{z,2}}, \hat{U}) \cdot \prod_{z=1}^{\eta} e(\pi_{z,1}^{-\delta_{z,2}}, g_2) \overset{?}{=} \prod_{z=1}^{\eta} e(\pi_{z,l}^{\delta_{z,1}}, U_0) \cdot \prod_{z=1}^{\eta} {y_z}^{\delta_{z,1}} \cdot \prod_{z=1}^{\eta} e(\pi_{z,0}^{-\delta_{z,1}}, g_2 \cdot h) \tag{9}$$

EQ1 Step 3: Move products inside pairings to reduce η pairings to 1 (technique 3a):

$$e(\prod_{z=1}^{\eta} g_1^{(1-x_1)\cdot\delta_{z,2}} \cdot U_1^{x_1\cdot\delta_{z,2}}, \hat{U}) \cdot e(\prod_{z=1}^{\eta} \pi_{z,1}^{-\delta_{z,2}}, g_2) \overset{?}{=} e(\prod_{z=1}^{\eta} \pi_{z,l}^{\delta_{z,1}}, U_0) \cdot \prod_{z=1}^{\eta} {y_z}^{\delta_{z,1}} \cdot e(\prod_{z=1}^{\eta} \pi_{z,0}^{-\delta_{z,1}}, g_2 \cdot h) \tag{10}$$

EQ2 Step 4: Combine η signatures (technique 1):

$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } \prod_{z=1}^{\eta} e(\pi_{z,t}, g_2) \overset{?}{=} \prod_{z=1}^{\eta} e(\pi_{z,t-1}^{(1-x_{z,t})}, g_2) \cdot e(\pi_{z,t-1}^{x_{z,t}}, U_t) \tag{11}$$

EQ2 Step 5: Apply the small exponents test, using exponents $\delta_1, \ldots \delta_\eta \in \left[1, 2^\lambda\right]$:

$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } \prod_{z=1}^{\eta} e(\pi_{z,t}, g_2)^{\delta_z} \overset{?}{=} \prod_{z=1}^{\eta} \left(e(\pi_{z,t-1}^{(1-x_{z,t})}, g_2) \cdot e(\pi_{z,t-1}^{x_{z,t}}, U_t)\right)^{\delta_z} \tag{12}$$

EQ2 Step 6: Move exponent(s) inside the pairing (technique 2):

$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } \prod_{z=1}^{\eta} e(\pi_{z,t}^{\delta_z}, g_2) \overset{?}{=} \prod_{z=1}^{\eta} e(\pi_{z,t-1}^{(1-x_{z,t})\cdot\delta_z}, g_2) \cdot e(\pi_{z,t-1}^{x_{z,t}\cdot\delta_z}, U_t) \tag{13}$$

EQ2 Step 7: Move products inside pairings to reduce η pairings to 1 (technique 3a):

$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } e(\prod_{z=1}^{\eta} \pi_{z,t}^{\delta_z}, g_2) \overset{?}{=} \prod_{z=1}^{\eta} e(\pi_{z,t-1}^{(1-x_{z,t})\cdot\delta_z}, g_2) \cdot e(\pi_{z,t-1}^{x_{z,t}\cdot\delta_z}, U_t) \tag{14}$$

EQ2 Step 8: Distribute products (technique 5):

$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } e(\prod_{z=1}^{\eta} \pi_{z,t}^{\delta_z}, g_2) \overset{?}{=} \prod_{z=1}^{\eta} e(\pi_{z,t-1}^{(1-x_{z,t})\cdot\delta_z}, g_2) \cdot \prod_{z=1}^{\eta} e(\pi_{z,t-1}^{x_{z,t}\cdot\delta_z}, U_t) \tag{15}$$

EQ2 Step 9: Move products inside pairings to reduce η pairings to 1 (technique 3a):

$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } e(\prod_{z=1}^{\eta} \pi_{z,t}^{\delta_z}, g_2) \overset{?}{=} e(\prod_{z=1}^{\eta} \pi_{z,t-1}^{(1-x_{z,t})\cdot\delta_z}, g_2) \cdot e(\prod_{z=1}^{\eta} \pi_{z,t-1}^{x_{z,t}\cdot\delta_z}, U_t) \tag{16}$$

EQ2 Step 10: Merge pairings with common first or second argument (technique 3b):

$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } e(\prod_{z=1}^{\eta} \pi_{z,t}^{\delta_z} \cdot \pi_{z,t-1}^{(1-x_{z,t})\cdot -\delta_z}, g_2) \overset{?}{=} e(\prod_{z=1}^{\eta} \pi_{z,t-1}^{x_{z,t}\cdot\delta_z}, U_t) \tag{17}$$

EQ2 Step 11: Unroll for loop (technique 0b) and choose random $\delta_3, \ldots, \delta_9 \in [1, 2^\lambda - 1]$ for each z = 1 to

η equations:

$$e(\prod_{z=1}^{\eta} \pi_{z,2}^{\delta_{z,3}} \cdot \pi_{z,1}^{(1-x_{z,2})\cdot -\delta_{z,3}} \cdot \pi_{z,3}^{-\delta_{z,4}} \cdot \pi_{z,2}^{(1-x_{z,3})\cdot -\delta_{z,4}\cdot -1} \cdot \pi_{z,4}^{-\delta_{z,5}} \cdot \pi_{z,3}^{(1-x_{z,4})\cdot -\delta_{z,5}\cdot -1} \cdot \pi_{z,5}^{-\delta_{z,6}} \cdot \pi_{z,4}^{(1-x_{z,5})\cdot -\delta_{z,6}\cdot -1}$$

$$\cdot \pi_{z,6}^{-\delta_{z,7}} \cdot \pi_{z,5}^{(1-x_{z,6})\cdot -\delta_{z,7}\cdot -1} \cdot \pi_{z,7}^{-\delta_{z,8}} \cdot \pi_{z,6}^{(1-x_{z,7})\cdot -\delta_{z,8}\cdot -1} \cdot \pi_{z,8}^{-\delta_{z,9}} \cdot \pi_{z,7}^{(1-x_{z,8})\cdot -\delta_{z,9}\cdot -1}, g_2) \stackrel{?}{=}$$

$$e(\prod_{z=1}^{\eta} \pi_{z,1}^{x_{z,2}\cdot\delta_{z,3}}, U_2) \cdot e(\prod_{z=1}^{\eta} \pi_{z,2}^{x_{z,3}\cdot\delta_{z,4}\cdot -1}, U_3) \cdot e(\prod_{z=1}^{\eta} \pi_{z,3}^{x_{z,4}\cdot\delta_{z,5}\cdot -1}, U_4) \cdot e(\prod_{z=1}^{\eta} \pi_{z,4}^{x_{z,5}\cdot\delta_{z,6}\cdot -1}, U_5)$$

$$\cdot e(\prod_{z=1}^{\eta} \pi_{z,5}^{x_{z,6}\cdot\delta_{z,7}\cdot -1}, U_6) \cdot e(\prod_{z=1}^{\eta} \pi_{z,6}^{x_{z,7}\cdot\delta_{z,8}\cdot -1}, U_7) \cdot e(\prod_{z=1}^{\eta} \pi_{z,7}^{x_{z,8}\cdot\delta_{z,9}\cdot -1}, U_8) \quad (18)$$

Step 12: Combine equations 1 and 2, then pairings within final equation (technique 3b):

$$e(\prod_{z=1}^{\eta} g_1^{(1-x_1)\cdot\delta_{z,2}} \cdot U_1^{x_1\cdot\delta_{z,2}}, \hat{U}) \cdot e(\prod_{z=1}^{\eta} \pi_{z,1}^{-\delta_{z,2}} \cdot \pi_{z,2}^{\delta_{z,3}} \cdot \pi_{z,1}^{(1-x_{z,2})\cdot -\delta_{z,3}} \cdot \pi_{z,3}^{-\delta_{z,4}} \cdot \pi_{z,2}^{(1-x_{z,3})\cdot -\delta_{z,4}\cdot -1} \cdot \pi_{z,4}^{-\delta_{z,5}} \cdot \pi_{z,3}^{(1-x_{z,4})\cdot -\delta_{z,5}\cdot -1}$$

$$\cdot \pi_{z,5}^{-\delta_{z,6}} \cdot \pi_{z,4}^{(1-x_{z,5})\cdot -\delta_{z,6}\cdot -1} \cdot \pi_{z,6}^{-\delta_{z,7}} \cdot \pi_{z,5}^{(1-x_{z,6})\cdot -\delta_{z,7}\cdot -1} \cdot \pi_{z,7}^{-\delta_{z,8}} \cdot \pi_{z,6}^{(1-x_{z,7})\cdot -\delta_{z,8}\cdot -1} \cdot \pi_{z,8}^{-\delta_{z,9}} \cdot \pi_{z,7}^{(1-x_{z,8})\cdot -\delta_{z,9}\cdot -1}, g_2) \stackrel{?}{=}$$

$$e(\prod_{z=1}^{\eta} \pi_{z,l}^{\delta_{z,1}}, U_0) \cdot \prod_{z=1}^{\eta} {y_z}^{\delta_{z,1}} \cdot e(\prod_{z=1}^{\eta} \pi_{z,0}^{-\delta_{z,1}}, g_2 \cdot h) \cdot e(\prod_{z=1}^{\eta} \pi_{z,1}^{x_{z,2}\cdot\delta_{z,3}}, U_2) \cdot e(\prod_{z=1}^{\eta} \pi_{z,2}^{x_{z,3}\cdot\delta_{z,4}\cdot -1}, U_3) \cdot e(\prod_{z=1}^{\eta} \pi_{z,3}^{x_{z,4}\cdot\delta_{z,5}\cdot -1}, U_4)$$

$$\cdot e(\prod_{z=1}^{\eta} \pi_{z,4}^{x_{z,5}\cdot\delta_{z,6}\cdot -1}, U_5) \cdot e(\prod_{z=1}^{\eta} \pi_{z,5}^{x_{z,6}\cdot\delta_{z,7}\cdot -1}, U_6) \cdot e(\prod_{z=1}^{\eta} \pi_{z,6}^{x_{z,7}\cdot\delta_{z,8}\cdot -1}, U_7) \cdot e(\prod_{z=1}^{\eta} \pi_{z,7}^{x_{z,8}\cdot\delta_{z,9}\cdot -1}, U_8) \quad (19)$$

Steps 1 and 2 form the Combination Step in [28], which was proven to result in a secure batch verifier in [28, Theorem 3.2]. We observe that the remaining steps are merely reorganizing terms within the same equation except for the application of technique 0b, which applies the small exponents test again while unrolling the loop. Hence, the final verification equation (19) is also batch verifier for VRF. □

D A machine-generated candidate batch algorithm for WATERS09

The following candidate batching algorithm was automatically generated by the Batcher while processing the WATERS09 *signature scheme [66, 67]. This execution was restricted to signatures on a single signing key.*

D.1 Definitions

Let g_1, g_2 be values drawn from the key and/or parameters, and $M, \sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6, \sigma_7, \sigma_K, tag_k$ represent a message (or message hash) and signature. Select s_1, s_2, t, tag_c variables at random in $\mathbb{Z}_q$ and the variables θ, A are computed as follows: $\theta = 1/(tag_c - tag_k), A = e(g, g)^{\alpha\cdot a_1\cdot b}$. The individual verification equation WATERS09.Verify [§6.1][20] is:

$$e({g_1}^{bs}, \sigma_1) \cdot e({g_1}^{b\cdot a_1 s_1}, \sigma_2) \cdot e({g_1}^{a_1 s_1}, \sigma_3) \cdot e({g_1}^{b\cdot a_2 s_2}, \sigma_4) \cdot e({g_1}^{a_2 s_2}, \sigma_5) \stackrel{?}{=}$$
$$e(\sigma_6, \tau_1^{s_1} \cdot \tau_2^{s_2}) \cdot e(\sigma_7, {\tau_1}^{bs_1} \cdot {\tau_2}^{bs_2} \cdot w^{-t}) \cdot (e(\sigma_7, u^{M\cdot t} \cdot w^{tag_c\cdot t} \cdot h^t) \cdot e(g_1^{-t}, \sigma_K))^{\theta} \cdot A^{s_2}$$

[20]For simplicity, Waters [67] presents this verification equation as a series of calculations. We have merely combined these calculations, reorganized a few terms in the verification equation and turned division operations into multiplication.

Let η be the number of signatures in a batch, and $\delta_1, \ldots \delta_\eta \in \{1, 2^\lambda - 1\}$ be a set of random exponents chosen by the verifier. The batch verification equation WATERS09.BatchVerify is:

$$e({g_1}^b, \prod_{z=1}^{\eta} \sigma_{z,1}^{s_z \cdot \delta_z}) \cdot e({g_1}^{b \cdot a_1}, \prod_{z=1}^{\eta} \sigma_{z,2}^{s_{z,1} \cdot \delta_z}) \cdot e({g_1}^{a_1}, \prod_{z=1}^{\eta} \sigma_{z,3}^{s_{z,1} \cdot \delta_z}) \cdot e({g_1}^{b \cdot a_2}, \prod_{z=1}^{\eta} \sigma_{z,4}^{s_{z,2} \cdot \delta_z}) \cdot e({g_1}^{a_2}, \prod_{z=1}^{\eta} \sigma_{z,5}^{s_{z,2} \cdot \delta_z}) \stackrel{?}{=}$$
$$e(\prod_{z=1}^{\eta} \sigma_{z,6}^{\delta_z \cdot s_{z,1}}, \tau_1) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,6}^{\delta_z \cdot s_{z,2}}, \tau_2) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot s_{z,1}}, {\tau_1}^b) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot s_{z,2}}, {\tau_2}^b) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{(\delta_z \cdot -t_z + \theta_z \cdot \delta_z \cdot tag_{z,c} \cdot t_z)}, w)$$
$$\cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot M_z \cdot t_z}, u) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot t_z}, h) \cdot e(g_1, \prod_{z=1}^{\eta} \sigma_{z,K}^{-t_z \cdot \theta_z \cdot \delta_z}) \cdot A^{\sum_{z=0}^{\eta} s_{z,2} \cdot \delta_z}$$

We conjecture that this scheme satisfies a relaxation of Definition 2.2 to allow for two-sided negligible error; that is, where there is also a chance that a set of valid signatures will be rejected by the Batcher.

D.2 Details on How Candidate Construction was Derived

Via a series of steps, we show how the above batching algorithm was derived. We begin with the original verification equation.

$$e({g_1}^{bs}, \sigma_1) \cdot e({g_1}^{b \cdot a_1 s_1}, \sigma_2) \cdot e({g_1}^{a_1 s_1}, \sigma_3) \cdot e({g_1}^{b \cdot a_2 s_2}, \sigma_4) \cdot e({g_1}^{a_2 s_2}, \sigma_5) \stackrel{?}{=}$$
$$e(\sigma_6, \tau_1^{s_1} \cdot \tau_2^{s_2}) \cdot e(\sigma_7, {\tau_1}^{bs_1} \cdot {\tau_2}^{bs_2} \cdot w^{-t}) \cdot (e(\sigma_7, u^{M \cdot t} \cdot w^{tag_c \cdot t} \cdot h^t) \cdot e(g_1^{-t}, \sigma_K))^{\theta} \cdot A^{s_2} \quad (20)$$

Step 1: Combine η signatures (technique 1):

$$\prod_{z=1}^{\eta} e({g_1}^{bs_z}, \sigma_{z,1}) \cdot e({g_1}^{b \cdot a_1 s_{z,1}}, \sigma_{z,2}) \cdot e({g_1}^{a_1 s_{z,1}}, \sigma_{z,3}) \cdot e({g_1}^{b \cdot a_2 s_{z,2}}, \sigma_{z,4}) \cdot e({g_1}^{a_2 s_{z,2}}, \sigma_{z,5}) \stackrel{?}{=}$$
$$\prod_{z=1}^{\eta} e(\sigma_{z,6}, \tau_1^{s_{z,1}} \cdot \tau_2^{s_{z,2}}) \cdot e(\sigma_{z,7}, {\tau_1}^{bs_{z,1}} \cdot {\tau_2}^{bs_{z,2}} \cdot w^{-t_z})$$
$$\cdot (e(\sigma_{z,7}, u^{M_z \cdot t_z} \cdot w^{tag_{z,c} \cdot t_z} \cdot h^{t_z}) \cdot e(g_1^{-t_z}, \sigma_{z,K}))^{\theta_z} \cdot A^{s_{z,2}} \quad (21)$$

Step 2: Apply the small exponents test, using exponents $\delta_1, \ldots \delta_\eta \in [1, 2^\lambda - 1]$:

$$\prod_{z=1}^{\eta} (e({g_1}^{bs_z}, \sigma_{z,1}) \cdot e({g_1}^{b \cdot a_1 s_{z,1}}, \sigma_{z,2}) \cdot e({g_1}^{a_1 s_{z,1}}, \sigma_{z,3}) \cdot e({g_1}^{b \cdot a_2 s_{z,2}}, \sigma_{z,4}) \cdot e({g_1}^{a_2 s_{z,2}}, \sigma_{z,5}))^{\delta_z} \stackrel{?}{=}$$
$$\prod_{z=1}^{\eta} (e(\sigma_{z,6}, \tau_1^{s_{z,1}} \cdot \tau_2^{s_{z,2}}) \cdot e(\sigma_{z,7}, {\tau_1}^{bs_{z,1}} \cdot {\tau_2}^{bs_{z,2}} \cdot w^{-t_z})$$
$$\cdot (e(\sigma_{z,7}, u^{M_z \cdot t_z} \cdot w^{tag_{z,c} \cdot t_z} \cdot h^{t_z}) \cdot e(g_1^{-t_z}, \sigma_{z,K}))^{\theta_z} \cdot A^{s_{z,2}})^{\delta_z} \quad (22)$$

Step 3: Move exponent(s) inside the pairing (technique 2):

$$\prod_{z=1}^{\eta} e({g_1}^{bs_z \cdot \delta_z}, \sigma_{z,1}) \cdot e({g_1}^{b \cdot a_1 s_{z,1} \cdot \delta_z}, \sigma_{z,2}) \cdot e({g_1}^{a_1 s_{z,1} \cdot \delta_z}, \sigma_{z,3}) \cdot e({g_1}^{b \cdot a_2 s_{z,2} \cdot \delta_z}, \sigma_{z,4}) \cdot e({g_1}^{a_2 s_{z,2} \cdot \delta_z}, \sigma_{z,5}) \stackrel{?}{=}$$
$$\prod_{z=1}^{\eta} e(\sigma_{z,6}^{\delta_z}, \tau_1^{s_{z,1}} \cdot \tau_2^{s_{z,2}}) \cdot e(\sigma_{z,7}^{\delta_z}, {\tau_1}^{bs_{z,1}} \cdot {\tau_2}^{bs_{z,2}} \cdot w^{-t_z})$$
$$\cdot e(\sigma_{z,7}^{\theta_z \cdot \delta_z}, u^{M_z \cdot t_z} \cdot w^{tag_{z,c} \cdot t_z} \cdot h^{t_z}) \cdot e(g_1^{-t_z \cdot \theta_z \cdot \delta_z}, \sigma_{z,K}) \cdot A^{s_{z,2} \cdot \delta_z} \quad (23)$$

Step 4: Split pairings (technique 8):

$$\prod_{z=1}^{\eta} e({g_1}^{b s_z \cdot \delta_z}, \sigma_{z,1}) \cdot e({g_1}^{b \cdot a_1 s_{z,1} \cdot \delta_z}, \sigma_{z,2}) \cdot e({g_1}^{a_1 s_{z,1} \cdot \delta_z}, \sigma_{z,3}) \cdot e({g_1}^{b \cdot a_2 s_{z,2} \cdot \delta_z}, \sigma_{z,4}) \cdot e({g_1}^{a_2 s_{z,2} \cdot \delta_z}, \sigma_{z,5}) \stackrel{?}{=}$$
$$\prod_{z=1}^{\eta} e(\sigma_{z,6}^{\delta_z}, {\tau_1}^{s_{z,1}}) \cdot e(\sigma_{z,6}^{\delta_z}, {\tau_2}^{s_{z,2}}) \cdot e(\sigma_{z,7}^{\delta_z}, {\tau_1}^{b s_{z,1}}) \cdot e(\sigma_{z,7}^{\delta_z}, {\tau_2}^{b s_{z,2}}) \cdot e(\sigma_{z,7}^{\delta_z}, w^{-t_z}) \cdot e(\sigma_{z,7}^{\theta_z \cdot \delta_z}, u^{M_z \cdot t_z})$$
$$\cdot e(\sigma_{z,7}^{\theta_z \cdot \delta_z}, w^{tag_{z,c} \cdot t_z}) \cdot e(\sigma_{z,7}^{\theta_z \cdot \delta_z}, h^{t_z}) \cdot e(g_1^{-t_z \cdot \theta_z \cdot \delta_z}, \sigma_{z,K}) \cdot A^{s_{z,2} \cdot \delta_z} \quad (24)$$

Step 5: Distribute products (technique 5):

$$\prod_{z=1}^{\eta} e({g_1}^{b s_z \cdot \delta_z}, \sigma_{z,1}) \cdot \prod_{z=1}^{\eta} e({g_1}^{b \cdot a_1 s_{z,1} \cdot \delta_z}, \sigma_{z,2}) \cdot \prod_{z=1}^{\eta} e({g_1}^{a_1 s_{z,1} \cdot \delta_z}, \sigma_{z,3}) \cdot \prod_{z=1}^{\eta} e({g_1}^{b \cdot a_2 s_{z,2} \cdot \delta_z}, \sigma_{z,4}) \cdot \prod_{z=1}^{\eta} e({g_1}^{a_2 s_{z,2} \cdot \delta_z}, \sigma_{z,5}) \stackrel{?}{=}$$
$$\prod_{z=1}^{\eta} e(\sigma_{z,6}^{\delta_z}, {\tau_1}^{s_{z,1}}) \cdot \prod_{z=1}^{\eta} e(\sigma_{z,6}^{\delta_z}, {\tau_2}^{s_{z,2}}) \cdot \prod_{z=1}^{\eta} e(\sigma_{z,7}^{\delta_z}, {\tau_1}^{b s_{z,1}}) \cdot \prod_{z=1}^{\eta} e(\sigma_{z,7}^{\delta_z}, {\tau_2}^{b s_{z,2}}) \cdot \prod_{z=1}^{\eta} e(\sigma_{z,7}^{\delta_z}, w^{-t_z}) \cdot \prod_{z=1}^{\eta} e(\sigma_{z,7}^{\theta_z \cdot \delta_z}, u^{M_z \cdot t_z})$$
$$\cdot \prod_{z=1}^{\eta} e(\sigma_{z,7}^{\theta_z \cdot \delta_z}, w^{tag_{z,c} \cdot t_z}) \cdot \prod_{z=1}^{\eta} e(\sigma_{z,7}^{\theta_z \cdot \delta_z}, h^{t_z}) \cdot \prod_{z=1}^{\eta} e(g_1^{-t_z \cdot \theta_z \cdot \delta_z}, \sigma_{z,K}) \cdot \prod_{z=1}^{\eta} A^{s_{z,2} \cdot \delta_z} \quad (25)$$

Step 6: Move products inside pairings to reduce η pairings to 1 (technique 3a) and move product to summation on precomputed pairing (technique 6):

$$e({g_1}^{b}, \prod_{z=1}^{\eta} \sigma_{z,1}^{s_z \cdot \delta_z}) \cdot e({g_1}^{b \cdot a_1}, \prod_{z=1}^{\eta} \sigma_{z,2}^{s_{z,1} \cdot \delta_z}) \cdot e({g_1}^{a_1}, \prod_{z=1}^{\eta} \sigma_{z,3}^{s_{z,1} \cdot \delta_z}) \cdot e({g_1}^{b \cdot a_2}, \prod_{z=1}^{\eta} \sigma_{z,4}^{s_{z,2} \cdot \delta_z}) \cdot e({g_1}^{a_2}, \prod_{z=1}^{\eta} \sigma_{z,5}^{s_{z,2} \cdot \delta_z}) \stackrel{?}{=}$$
$$e(\prod_{z=1}^{\eta} \sigma_{z,6}^{\delta_z \cdot s_{z,1}}, \tau_1) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,6}^{\delta_z \cdot s_{z,2}}, \tau_2) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot s_{z,1}}, {\tau_1}^{b}) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot s_{z,2}}, {\tau_2}^{b}) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot -t_z}, w) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot M_z \cdot t_z}, u)$$
$$\cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot tag_{z,c} \cdot t_z}, w) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot t_z}, h) \cdot e(g_1, \prod_{z=1}^{\eta} \sigma_{z,K}^{-t_z \cdot \theta_z \cdot \delta_z}) \cdot A^{\sum_{z=0}^{\eta} s_{z,2} \cdot \delta_z} \quad (26)$$

Step 7: Merge pairings with common first or second argument (technique 3b):

$$e({g_1}^{b}, \prod_{z=1}^{\eta} \sigma_{z,1}^{s_z \cdot \delta_z}) \cdot e({g_1}^{b \cdot a_1}, \prod_{z=1}^{\eta} \sigma_{z,2}^{s_{z,1} \cdot \delta_z}) \cdot e({g_1}^{a_1}, \prod_{z=1}^{\eta} \sigma_{z,3}^{s_{z,1} \cdot \delta_z}) \cdot e({g_1}^{b \cdot a_2}, \prod_{z=1}^{\eta} \sigma_{z,4}^{s_{z,2} \cdot \delta_z}) \cdot e({g_1}^{a_2}, \prod_{z=1}^{\eta} \sigma_{z,5}^{s_{z,2} \cdot \delta_z}) \stackrel{?}{=}$$
$$e(\prod_{z=1}^{\eta} \sigma_{z,6}^{\delta_z \cdot s_{z,1}}, \tau_1) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,6}^{\delta_z \cdot s_{z,2}}, \tau_2) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot s_{z,1}}, {\tau_1}^{b}) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot s_{z,2}}, {\tau_2}^{b}) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{(\delta_z \cdot -t_z + \theta_z \cdot \delta_z \cdot tag_{z,c} \cdot t_z)}, w)$$
$$\cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot M_z \cdot t_z}, u) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot t_z}, h) \cdot e(g_1, \prod_{z=1}^{\eta} \sigma_{z,K}^{-t_z \cdot \theta_z \cdot \delta_z}) \cdot A^{\sum_{z=0}^{\eta} s_{z,2} \cdot \delta_z} \quad (27)$$

E SDL Grammar

We provide a full description of our SDL grammar below:

$\langle$*assign-statement*$\rangle$ ::= $\langle$*single-assignment*$\rangle$ | $\langle$*func-call-statement*$\rangle$
| $\langle$*group-assign-statement*$\rangle$ | $\langle$*dict-statement*$\rangle$ | $\langle$*random-statement*$\rangle$ | $\langle$*hash-statement*$\rangle$ | $\langle$*pair-statement*$\rangle$.

⟨*single-assignment*⟩ ::= ⟨*variable-target*⟩ ⟨*assign-op*⟩ ⟨*expr-statement*⟩ | ⟨*variable-target*⟩.

⟨*variable-target*⟩ ::= ⟨*keywords*⟩ | ⟨*variable-name*⟩.

⟨*group-assign-statement*⟩ ::= ⟨*variable-name*⟩ ⟨*assign-op*⟩ ⟨*type*⟩.

⟨*func-call-statement*⟩ ::= ⟨*variable-name*⟩ ⟨*assign-op*⟩ ⟨*variable-name*⟩ ‘`(`’ ⟨*arg-list*⟩ ‘`)`’.

⟨*arg-list*⟩ ::= ⟨*arg-name*⟩ | ⟨*arg-name*⟩ ‘`,`’ ⟨*arg-list*⟩.

⟨*arg-name*⟩ ::= ⟨*variable-name*⟩.

⟨*random-statement*⟩ ::= ⟨*variable-name*⟩ ⟨*assign-op*⟩ ⟨*random-func-name*⟩ ‘`(`’ ⟨*group-type*⟩ ‘`)`’.

⟨*hash-statement*⟩ ::= ⟨*variable-name*⟩ ⟨*assign-op*⟩ ⟨*hash-func-name*⟩ ‘`(`’ ⟨*arg-list*⟩ ‘`,`’ ⟨*group-type*⟩ ‘`)`’.

⟨*dict-statement*⟩ ::= ⟨*variable-name*⟩ ⟨*assign-op*⟩ ‘`list{`’ ⟨*arg-list*⟩ ‘`}`’ | ‘`expand{`’ ⟨*arg-list*⟩ ‘`}`’.

⟨*dot-prod-statement*⟩ ::= ‘`prod{`’ ⟨*single-assignment*⟩ ‘`,`’ ⟨*single-assignment*⟩ ‘`} on`’ ⟨*expr-statement*⟩.

⟨*sum-of-statement*⟩ ::= ‘`sum{`’ ⟨*single-assignment*⟩ ‘`,`’ ⟨*single-assignment*⟩ ‘`} of`’ ⟨*expr-statement*⟩

⟨*for-statement*⟩ ::= ⟨*proc-token*⟩ ⟨*block-sep*⟩ ‘`for`’
  | ‘`for{`’ ⟨*assign-statement*⟩ `,` ⟨*assign-statement*⟩ ‘`}`’ [⟨*new-line*⟩ ⟨*expr-statement*⟩]*
  | `forall{` ⟨*assign-statement*⟩ `,` ⟨*assign-statement*⟩ ‘`}`’ [⟨*new-line*⟩ ⟨*expr-statement*⟩]*.

⟨*conditional-statement*⟩ ::= ⟨*proc-token*⟩ ⟨*block-sep*⟩ ‘`if`’
  | ‘`if{`’ ⟨*cond-statement*⟩ ‘`}`’ [⟨*new-line*⟩ ⟨*expr-statement*⟩]+
  | ‘`else`’ [⟨*new-line*⟩⟨*expr-statement*⟩]+.

⟨*expr-statement*⟩ ::= ⟨*pair-statement*⟩ | ⟨*expr0-statement*⟩.

⟨*expr0-statement*⟩ ::= ⟨*expr0-statement*⟩ ⟨*group-op*⟩ ⟨*expr0-statement*⟩
  | ⟨*exp1-statement*⟩
  | ⟨*variable-name*⟩.

⟨*cond-statement*⟩ ::= ⟨*cond-statement*⟩ [⟨*bool-op*⟩ ⟨*cond-statement*⟩]* | ⟨*expr-statement*⟩.

⟨*pair-statement*⟩ ::= ‘`e(`’ ⟨*expr-statement*⟩ ‘`,`’ ⟨*expr-statement*⟩ ‘`)`’
  | ⟨*pair-statement*⟩ ⟨*group-op*⟩ ⟨*pair-statement*⟩
  | ⟨*pair-statement*⟩ ⟨*exp*⟩ ⟨*exp1-statement*⟩.

⟨*exp1-statement*⟩ ::= ⟨*exp1-statement*⟩ ⟨*exp*⟩ ⟨*exp2-statement*⟩ | ⟨*expr2-statement*⟩

⟨*exp2-statement*⟩ ::= ⟨*expr-statement*⟩ ⟨*exp-ops*⟩ ⟨*expr-statement*⟩
  | ⟨*expr-statement*⟩ ⟨*group-op*⟩ ⟨*expr-statement*⟩
  | ⟨*negate-op*⟩ [⟨*exp2-statement*⟩ | ⟨*integer*⟩].

⟨*element-type*⟩ ::= None | int | str | ZR | G1 | G2 | GT

⟨*type*⟩ ::= ⟨*element-type*⟩ | ‘`list{`’ ⟨*element-type*⟩, [⟨*element-type*⟩]* ‘`}`’
  | ‘`expand{`’ ⟨*element-type*⟩, [⟨*element-type*⟩]* ‘`}`’
  | ⟨*type-list*⟩.

⟨*type-list*⟩ ::= ⟨*type*⟩ ‘;’ ⟨*type-list*⟩ | ⟨*type*⟩.

⟨*procedure*⟩ ::= ⟨*proc-token*⟩ ⟨*block-sep*⟩ ⟨*procedure-name*⟩.

⟨*procedure-name*⟩ ::= ⟨*variable-name*⟩ | ‘func:’ ⟨*procedure-name*⟩.

⟨*variable-name*⟩ ::= [0-9, a-z, A-Z,⟨*symbols*⟩]*

⟨*symbols*⟩ ::= ‘_’ | ‘#’ | ‘?’ | ‘$’

⟨*proc-token*⟩ ::= ‘BEGIN’ | ‘END’

⟨*block-sep*⟩ ::= ‘::’

⟨*random-func-name*⟩ ::= ‘random’

⟨*hash-func-name*⟩ ::= ‘H’

⟨*keywords*⟩ ::= ‘N’ | ‘verify’ | ‘constant’ | ‘public’ | ‘signature’ | ‘message’ | ‘public_count’ | ‘signature_count’ | ‘message_count’ | ‘latex’ | ‘precompute’ | ‘types’ | ‘name’ | ‘setting’ | ‘symmetric’ | ‘asymmetric’

⟨*assign-op*⟩ ::= ‘:=’

⟨*exp*⟩ ::= ‘^’

⟨*exp-ops*⟩ ::= ‘+’ | ‘-’ | ⟨*group-op*⟩

⟨*group-op*⟩ ::= ‘*’ | ‘/’

⟨*bool-op*⟩ ::= ‘|’ | ‘and’ | ‘or’ | ‘==’ | ‘!=’ | ‘<’ | ‘<=’ | ‘>’ | ‘>=’

⟨*negate-op*⟩ ::= ‘-’

F Semantics of SDL

We provide a brief overview of our domain specific language and examples of how schemes are written in it. SDL can accommodate a full description of pairing schemes in situations where an existing implementation of a signature scheme does not exist or a developer prefers to code their scheme directly in SDL. This information is used to inform AutoBatch on details needed to generate the scheme implementation and the batch algorithm. The SDL file consists of two parts.

The first part is a full representation of the signature scheme which consists of the descriptions of each algorithm such as **keygen**, **sign**, **verify** and a **types** section. This information is used to generate executable code for the scheme either in Python or C++.

The second part is a broken down version of the verification algorithm in a form for AutoBatch to derive the desired batch verification algorithm. To this end, there are several keywords used to provide context for AutoBatch. **Public**, **signature** and **message** keywords are used to identify the public key variables and the signature and message variables. Additionally, the **public_count** keyword is used to determine whether public keys belong to the same or different signers. The **signature_count** and **message_count** keywords describe the number of signatures and messages expected per batch. The **constants** keyword describes variables in the scheme shared by signers such as the generators of a group. **Precompute** section represents computation steps necessary before each verification check. The **verify** keyword is used to describe the verification equation as a mathematical expression. Finally, we include a block for LaTeX to assist the proof generator map variables in SDL to equivalent LaTeX representation.

Our abstract language is capable of representing a variety of programming constructs such as dot products, for loops, summation, and boolean operators. Thus, very complex schemes can be described using our SDL and to reflect this we provide full SDL descriptions below for BLS [16] and CL04 [20]. See our github repository for other full SDL examples.

```
#########################################################
##                 BLS signature scheme                ##
#########################################################
name := bls
# expected batch size per some time period
N := 100
setting := asymmetric

# types for variables used in verification.
# all other variable types are inferred by SDL parser
BEGIN :: types
 M := Str
END :: types

# description of key generation, signing, and verification algorithms
BEGIN :: func:keygen
input := list{None}
 # choose random generator in a prime order group G2
 g := random(G2)
 # choose random integer modulo prime r
 x := random(ZR)
 pk := g^x
 sk := x
 # keygen returns a tuple consisting of three elements
output := list{pk, sk, g}
END :: func:keygen

BEGIN :: func:sign
input := list{sk, M}
 # H is a general purpose hash function that maps its inputs
 # (consisting of strings, group elements, etc)
 # to a particular target group (either ZR, G1 or G2)
 sig := (H(M, G1)^sk)
output := sig
END :: func:sign

BEGIN :: func:verify
input := list{pk, M, sig, g}
 h := H(M, G1)
 BEGIN :: if
 if {e(h, pk) == e(sig, g)}
      output := True
 else
      output := False
 END :: if
END :: func:verify
```

```
# Batcher SDL input
constant := g
public :=  pk
signature :=  sig
message :=  h

# same signer
BEGIN :: count
 message_count := N
 public_count := one
 signature_count := N
END :: count

# variables computed before each signature verification
BEGIN :: precompute
  h := H(M, G1)
END :: precompute

# verification equation
verify := {e(h, pk) == e(sig, g)}



##########################################################
##                CL signature scheme                  ##
##########################################################
name := cl04
N := 100
setting := asymmetric

BEGIN :: types
 M := Str
 # represents a list of elements in group G2
 sig := list{G2}
END :: types

BEGIN :: func:setup
input := list{None}
 g := random(G1)
output := g
END :: func:setup

BEGIN :: func:keygen
input := list{g}
 x := random(ZR)
 y := random(ZR)
 X := g^x
 Y := g^y
 sk := list{x, y}
 pk := list{X, Y}
output := list{pk, sk}
END :: func:keygen
```

```
BEGIN :: func:sign
input := list{sk, M}
 # expand macro is shorthand for extracting the variables contained
 # in the list or tuple to make them accessible within the function
 sk := expand{x, y}
 a := random(G2)
 m := H(M, ZR)
 b := a^y
 c := a^(x + (m * x * y))
 sig := list{a, b, c}
output := sig
END :: func:sign

BEGIN :: func:verify
input := list{pk, g, M, sig}
 pk := expand{X, Y}
 sig := expand{a, b, c}
 m := H(M, ZR)
 BEGIN :: if
 if {{ e(Y, a) == e(g, b) } and { (e(X, a) * (e(X, b)^m)) == e(g, c) }}
   output := True
 else
   output := False
 END :: if
END :: func:verify

# Batcher input
BEGIN :: precompute
 m := H(M, ZR)
END :: precompute

constant := g
public := pk
signature := sig
message := m

# same signer
BEGIN :: count
 message_count := N
 public_count := one
 signature_count := N
END :: count

verify := { e(Y, a) == e(g, b) } and { (e(X, a) * (e(X, b)^m)) == e(g, c) }
```