ENABLING MACHINE-AIDED CRYPTOGRAPHIC DESIGN

by

Joseph Ayo Akinyele

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

November, 2013

© Joseph Ayo Akinyele 2013

All rights reserved

Abstract

The design of cryptographic primitives such as digital signatures and public-key encryption is very often a manual process conducted by expert cryptographers. This persists despite the fact that many new generic or semi-generic methods have been proposed to construct new primitives by transforming existing ones in interesting ways. However, manually applying transformations to existing primitives can be error-prone, ad-hoc and tedious. A natural question is whether automating the process of applying cryptographic transformations would yield competitive or better results?

In this thesis, we explore a compiler-based approach for automatically performing certain cryptographic designs. Similar approaches have been applied to various types of cryptographic protocol design with compelling results [1–10]. We extend this same approach and show that it also can be effective towards automatically applying cryptographic transformations.

We first present our extensible architecture that automates a class of cryptographic transformations on primitives. We then propose several techniques that address the aforementioned question including the Charm [11] cryptographic framework, which enables

rapid prototyping of cryptographic primitives from abstract descriptions. We build on this work and show the extent to which transformations can be performed automatically given these descriptions. To illustrate this automation, we present a series of cryptographic tools that demonstrate the effectiveness of our automated approach. Our contributions are listed as follows:

- AutoBatch: Batch verification is a transformation that improves signature verification time by efficiently processing many signatures at once. Historically, this manual process has been prone to error and tedious for practitioners. We describe the design of an automated tool that finds efficient batch verification algorithms from abstract descriptions of signature schemes.

- AutoGroup: Cryptographers often prefer to describe their pairing-based constructions using symmetric group notation for simplicity, while they prefer asymmetric groups for implementation due to the efficiency gains. The symmetric-to-asymmetric translation is usually performed through manual analysis of a scheme and finding an efficient translation that suits applications can be quite challenging. We present an automated tool that uses SMT solvers to find efficient asymmetric translations from abstract descriptions of cryptographic schemes.

- AutoStrong: Strongly unforgeable signatures are desired in practice for a variety of cryptographic protocols. Several transformations exist in the literature that show how to obtain strongly unforgeable signatures from existentially unforgeable ones.

ABSTRACT

> We focus on a particular highly-efficient transformation due to Boneh, Shen and Waters [12] that is applicable if the signature satisfies a notion of partitioning. Checking for this property can be challenging and has been less explored in the literature. We present an automated tool that also utilizes SMT solvers to determine when this property is applicable for constructing efficient strongly unforgeable signatures from abstract descriptions.

We anticipate that these proof-of-concept tools embody the notion that certain cryptographic transformations can be safely and effectively outsourced to machines.

Primary Reader: Aviel D. Rubin

Secondary Readers: Jonathan Katz, Matthew D. Green and Susan Hohenberger

Acknowledgments

I am thankful to many great people for their support in completing this thesis. I would like to start by thanking my advisors who have made my Ph.D. experience at Johns Hopkins so intellectually rewarding. First, I would like to thank Avi Rubin for bringing me to Hopkins and for providing me timely, invaluable advice and encouragement. I would also like to thank Matthew Green for his guidance and support. I learned a great deal from him and will always be thankful to him for introducing me to cryptography and his willingness to share his knowledge. Moreover, I would like to thank Susan Hohenberger for engaging and challenging me in the area of automated cryptographic design. I very much appreciate her enthusiasm and her vast knowledge has helped me tremendously in my research. I thank Jonathan Katz for serving on my thesis committee and reviewing my thesis on such short notice.

There are many others that have made my time as a Ph.D student enjoyable. In particular, I thank my research partner, Matthew Pagano for his willingness to share his expertise from the moment I arrived at Hopkins. We have spent countless hours working together and I have learned a lot from him. I also thank Ryan Gardner for always willing to answer my

questions and sharing his experiences with me; Sam Small for his continuous encouragement during the tough times as a student; Steve Checkoway; Zachary Peterson; Michael Rushanan; Christina Garman; Ian Miers; Paul Martin and Gary Truslow. I am sincerely grateful that I had an opportunity to work with such great people.

I would also like to extend thanks to the administrative staff in the CS department for consistently putting up with my requests. Also, I am grateful to the National Science Foundation (under grant CNS-1154035), the Office of Naval Research (under contract N00014-11-1-0470) and SHARPS for kindly funding my research. Lastly, I am very grateful to my family for believing in me even when I doubted myself. I am thankful for their prayers and unwavering support.

I especially thank my wonderful wife, Mojola, for her encouragement during the discouraging times of my Ph.D pursuit. She provided perspective when I lacked it and support when I needed it. I will always be grateful for the happiness she has given me.

Finally, I express my sincere gratitude to everyone that has helped me along my journey. I am forever grateful. Thanks!

Dedication

This thesis is dedicated to my daughter, Tori and my wife, Mojola for their love and endless support.

Contents

Abstract **ii**

Acknowledgments **v**

List of Tables **xiv**

List of Figures **xv**

1 Introduction **1**

1.1 Our Approach 3

1.2 Summary of Our Contributions 4

1.3 Outline of This Work 6

2 Cryptographic Preliminaries **8**

2.1 Notation 8

2.2 Bilinear Groups 9

2.3 Standard Definitions for Digital Signatures 10

2.4 Standard Definitions for Public Key Encryption 13

3 Extensible Architecture for Automation 15

3.1 Overview 15

3.2 Background 16

3.3 Overview of Transformation Tasks 17

3.4 Our Architecture 19

3.5 Our Implementation 21

3.5.1 Scheme Description Language 21

3.5.2 SDL Parser 25

3.5.3 Cryptographic Transformations 27

3.5.3.1 Batching Digital Signatures 27

3.5.3.2 Optimizing Cryptographic Schemes 31

3.5.3.3 Constructing Strongly Unforgeable Signatures 35

3.5.4 Code Generator 39

3.6 Literature Review 40

4 Charm: A framework for Rapidly Prototyping Cryptosystems 45

4.1 Overview 46

4.2 Introduction 46

4.3 Background 51

4.4 Approach 52

4.5 Implementation 59
4.5.1 Schemes 62
4.5.2 Protocol Engine 64
4.5.3 ZKP Compiler 66
4.5.4 Meta-information and Adapters 68
4.5.5 Type checking and conversion 71
4.5.6 Using Charm in C applications 72
4.6 Performance 73
4.6.1 Comparison with C Implementations 74
4.7 Related Work 76
4.8 Charm-Crypto Toolkit 80
4.9 Challenges and Open Problems 80

5 Machine-Generated Algorithms, Proofs and Software for the Batch Verification of Digital Signature Schemes 83

5.1 Overview 84
5.2 Introduction 85
5.2.1 Our Contributions 86
5.2.2 Overview of Our Approach 88
5.2.3 Related Work 90
5.3 Batch Verification for Signatures 93
5.3.1 On Schemes with a Correctness Error 95

5.3.2 Algebraic Setting 96

5.3.3 Batch Verification in Bilinear Groups 97

5.3.4 Small Exponents Test Applied to Bilinear Groups 98

5.4 The AutoBatch Toolchain 99

5.4.1 Batching and Optimization 102

5.4.2 Technique Search Approach 107

5.4.3 Security and Machine-Aided Analysis 111

5.4.4 Code Generation 115

5.5 Implementation & Performance 116

5.5.1 Experimental Setup 118

5.5.2 Test Cases and Summary of the Results 119

5.5.3 Microbenchmarks 122

5.5.4 Batch Verification in Practice 124

5.5.4.1 Basic DoS Attacks 125

5.6 AutoBatch Toolkit 127

5.7 Challenges and Open Problems 127

6 Using SMT solvers to Automate Design Tasks for Encryption and Signature Schemes 129

6.1 Overview 130

6.2 Introduction 131

6.2.1 Our Contributions 132

6.2.2 Related Work 135
6.3 Tools Used 136
6.4 AutoGroup 137
6.4.1 Background on Pairing Groups 137
6.4.2 How AutoGroup Works 139
6.4.3 Security Analysis of AutoGroup 146
6.4.4 Experimental Evaluation of AutoGroup 147
6.5 AutoStrong 148
6.5.1 Background on Digital Signatures 149
6.5.2 How AutoStrong Works 156
6.5.3 Security Analysis of AutoStrong 163
6.5.4 Experimental Evaluation of AutoStrong 164
6.6 Challenges and Open Problems 165

7 Summary **166**

A Additional Material **167**
A.1 Scheme Examples In Charm 168
A.2 Semantics of SDL 173
A.3 Machine-Generated Batch Verification 182
A.4 Proof for Batch Verification of HW Signatures 184
A.4.1 Definitions 184

A.4.2 Proof 185

A.5 Proof for Batch Verification of CL04 Signatures 186

A.5.1 Definitions 187

A.5.2 Proof 187

A.6 Proof for Batch Verification of VRF 189

A.6.1 Definitions 189

A.6.2 Proof 191

A.7 Candidate Batch Verification for WATERS09 Signatures 194

A.7.1 Definitions 194

A.7.2 How Candidate Construction was Derived 195

Bibliography **199**

Vita **226**

List of Tables

4.1 A partial listing of the cryptographic schemes we implemented. "Code Lines" indicates the number of lines of Python code used to implement the scheme (excluding comments and whitespace), and does not include the framework itself. ROM indicates that a scheme is secure in the Random Oracle Model. CRS indicates that a scheme is secure in the Common Reference String Model. A "-" indicates a generic transform (adapter). * indicates a choice made for efficiency reasons. See the rest of the listing in Appendix A.1. 63

A.1 Another listing of the cryptographic schemes we implemented. "Code Lines" indicates the number of lines of Python code used to implement the scheme (excluding comments and whitespace), and does not include the framework itself. ROM indicates that a scheme is secure in the Random Oracle Model. CRS indicates that a scheme is secure in the Common Reference String Model. A "-" indicates a generic transform (adapter). * indicates a choice made for efficiency reasons. 168

List of Figures

3.1 At a high-level, the SDL parser takes as input a SDL file description of a cryptographic scheme along with some metadata. The parser converts this input file into an intermediate representation (IR). From this IR, the parser performs type checking utilizing external tools such as an SMT solver. A user-selected cryptographic transformation is applied to the IR, which may also employ external tools to assist with the transformation. The transformation produces a modified SDL file and optionally, a human-readable proof that the transformation preserves the security of the input scheme. The code generator produces a working implementation of the modified SDL in Python and/or C++ using a cryptographic library (*e.g.*, Charm) . . 19

3.2 A high-level presentation of the AutoBatch tool, which automates finding efficient batch verification algorithms. 30

3.3 A high-level presentation of the AutoGroup tool, which optimizes cryptographic schemes specified in the symmetric setting. 33

3.4 A high-level presentation of the AutoStrong tool, which automates the construction of strongly unforgeable signatures. 37

4.1 Overview of the Charm architecture. 52

4.2 Listing of scheme types defined in Charm. Subtypes are indicated with dotted lines. 54

4.3 Example of an adapter chain converting the Boneh-Boyen selective-ID secure IBE [66] into a signature scheme using Naor's technique [95]. The scheme carries meta-information including the complexity assumptions and computational model used in its security proof. 55

4.4 Adapter chain converting the Boneh-Boyen selective-ID secure IBE [66] into a CCA-secure public-key hybrid encryption scheme via the CHK transform [14]. $\{sig\}$ stands for the complexity assumptions added by the signature scheme. 56

4.5 Encryption and Decryption in the Cramer-Shoup scheme [106]. The top box shows the description of the algorithm in the published paper while the bottom box reflects the Charm code. Charm is designed to enable cryptographers to implement their schemes using mathematical notation that mirrors the paper description. 60
4.6 A partial listing of the generated protocol produced by our Zero-Knowledge compiler for the honest-verifier proof $\mathsf{ZKPoK}\{(x) : h = g^x\}$. 68
4.7 For EC-DSA, we select the NIST P-192 elliptic curve and for CP-ABE [29], we measure 50 attributes for keygen and 50 leaves in the policy tree for encrypt and decrypt. 76

5.1 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 5.4 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. 90
5.2 The Boneh-Lynn-Shacham (BLS) signature scheme [120] 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. 100
5.3 The Boneh-Lynn-Shacham (BLS) signature scheme [120] 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). 104
5.4 The Boneh-Lynn-Shacham (BLS) signature scheme [120] with same signer and η signatures in a batch. Upon applying technique 1 from Figure 5.3 to obtain the initial secure batch verifier, the goal is to optimize the equation. We first show the AST of the equation *after* Batcher has applied technique 2 (move exponents inside the pairing). Then, we show the result of applying technique 3 (move products inside the pairing) to arrive at an optimized batch equation. 105

5.5 The state transition table represents the transition function we developed for pruning our breath-first search (PBFS) algorithm. The function accepts as input the current state which represents the technique that was applied to the batch equation. The PBFS always starts in state 2 (where it tries to apply Technique 2). Then from there, the search attempts to follow any suggested states and applies the corresponding techniques. If the technique does not apply, the path is terminated. Otherwise, we check whether that path is already a subset of the paths we have covered so far. We continue with the search until all open paths are terminated. In an effort to ensure that all paths terminate, the state function restricts the transition from Technique 9 to 6 to occur once on a given path (indicated by *). 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. Once all paths are terminated, we attempt to apply Technique 10 to each path in a post-processing phase. 111
5.6 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. 117
5.7 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 that can be set to 5 in practice. To approximate verification performance, we count the total number of pairings needed to process η valid signatures. Unless otherwise noted, the inputs are from different signers. The final column indicates the order of the techniques from Section 5.4 that AutoBatch applied to obtain the resulting batch verifier. The **rows in bold** are the schemes where AutoBatch discovered new or improved algorithms. 120

5.8 Signature scheme microbenchmarks for Waters09 [57], HW [135] and CL [18] public-key signatures (same signer), the VRF [19] (with block size of 8), combined verification of ChCh+Hess IBS [136, 137], and Boyen ring signature (3 signer ring) [70]. 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. 123
5.9 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 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. 125
5.10 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. 126

6.1 A high-level presentation of the new automated tools, AutoGroup and AutoStrong. They take as input a Scheme Description Language (SDL) representation of a cryptographic scheme and output an SDL representation of a transformation of the scheme, which can possibly be further transformed by another tool. These tools are compatible with the existing AutoBatch tool and Code Generator (shaded). An SDL input to the Code Generator produces a software implementation of the scheme in either C++ or Python. 132
6.2 A high-level presentation of the AutoGroup tool, which uses external tools Z3 and SDL Parser. 137

6.3 AutoGroup on encryption schemes under various optimization options. We show running times and sizes for several schemes generated in C++ and compare symmetric to automatically generated asymmetric implementations at the same security levels (roughly equivalent with 3072 bit RSA). For IBE schemes, we measured with the identity string length at 100 bytes. For BGW, n denotes the number of users in the system. 145
6.4 A high-level presentation of the AutoStrong tool, which uses external tools Z3, Mathematica and SDL Parser. 148
6.5 We show the result of AutoGroup and AutoStrong on signature schemes. For CL, BB, and Waters (with length of identities, $\ell = 128$), we first apply AutoStrong to determine that the signature scheme is partitioned, then apply the BSW transform to obtain a strongly unforgeable signature in the symmetric setting. We then feed this as input to AutoGroup to realize an asymmetric variant under a given optimization. We also tested AutoStrong on the DSE signature and ACDK structure-preserving signature, even though these are not known to be existentially unforgeable. A partition was found for ACDK, but not DSE. 155
6.6 Running time required by the AutoGroup and AutoStrong routines to process the schemes discussed in this work (averaged over 10 test runs). The running time for AutoGroup includes the execution time of the Z3 SMT solver. The running time for AutoStrong also includes Z3 and Mathematica and the application of the BSW transformation. In all cases, the standard deviation in the results were within ±3% of the average. For AutoGroup, running times are correlated with the number of unique solutions found and the minimization of the weighted function using Z3. AutoStrong running times are highly correlated with the complexity of the verification equations. 161

A.1 A working example of how the API is utilized in a C application to embed a hybrid encryption adapter (see Figure A.2b) for any CP-ABE scheme such as the BSW07 [29] scheme. We provide several high-level functions that simplify using Charm schemes. In particular, the CallMethod() encapsulates several types of arguments to Python such as: %O for Charm objects, %s for ASCII strings, %A to convert into a Python list, and %b to a binary object. 169
A.2 Adapters in Charm. (a). The entire IBE to signature adapter scheme [28]. (b) A hybrid encryptor for ABE schemes in Charm. 170
A.3 Keygen in the Cramer-Shoup scheme [106]. We exclude group parameter generation. 171
A.4 CL signatures [73] are a useful building block for anonymous credential systems. We provide a full scheme description and Charm code, but exclude group parameter generation. 172

A.5 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 5.4 to the original verification equation in the order specified in Figure 5.7. 'Combined signatures' refers to the combined batching of multiple signature verification equations that share algebraic structure. . . . 183

Chapter 1

Introduction

Since public-key cryptography was introduced in the 1970s, the cryptographic research community has made impressive progress in developing new cryptographic primitives and protocols. Our understanding of basic technologies such as public key encryption and digital signatures has advanced considerably. These advances have given us entirely new paradigms for securing data and techniques for searching and computing on encrypted data. However, many of these advances exist mostly in research papers and have often gone unimplemented due to the lack of adequate tools support to implement them. This is a loss for users and we believe that addressing this problem should be a priority for the cryptographic community.

To change this trend, we set out to investigate new tools for developing and deploying cryptographic schemes. One of these tool is a framework called Charm [11] which facilitates rapid prototyping of cryptosystems by providing an extensible and modular ar-

chitecture while promoting the reuse of components. The hope is that Charm will bridge the gap between theory and practice to lower the barriers for practitioners to implement schemes. As such, Charm emphasizes implementing cryptography using abstract, mathematical notation similar to how cryptosystems are natively described in research papers. It effectively places focus on the cryptographic algorithms rather than on low-level implementation details.

While Charm has been used to implement over forty primitives in the literature, many more variants are possible by applying cryptographic transformations. Cryptographers have made a number of useful observations with respect to designing cryptographic schemes by transforming existing constructions in novel ways. These general transformations either obtain entirely new primitives, strengthen security or improve efficiency. For example, constructing batch verification algorithms is a general transformation in which the aim is to improve the efficiency of signature verification. In practice, transformed cryptographic schemes are useful and form core building blocks in a variety of larger cryptographic protocols [13–16].

Unfortunately, applying cryptographic transformations by hand to document all possible variations of a given primitive has been quite challenging and ad-hoc in nature. The existing manual approach has largely been tedious, error-prone, and in some cases, has lead to many insecure constructions. Generally speaking, history has taught us that these transformations must be applied to existing schemes in a careful, deliberate manner in order to preserve the security of the original scheme. To that end, an automated approach

for applying cryptographic transformations to digital signatures and encryption schemes has the potential to yield better results than what is currently done today. While an automated approach is ideal, a unified framework to accomplish such automation targeting cryptographic transformations has been largely unexplored in the research literature.

In this thesis, we explore an automated approach to show that we can expand on existing schemes in many ways – increase their efficiency, optimize their algebraic setting and even strengthen their security. These variations, numbering in the thousands for some constructions, can be generated on demand or stored in Charm for practical use. We believe that future cryptographic libraries will include not only one implementation of a certain algorithm, but also automations for deriving optimal variants. This thesis presents the first such comprehensive library of its kind.

1.1 Our Approach

In this work, we will investigate the answers to four important questions with respect to cryptographic transformations: Which tasks are naturally amenable to automation? What type of paradigm, tools and specification language are necessary for automating cryptographic transformations? To what extent can cryptographic design tasks be performed in an automated fashion and what are the limitations? How do we prove or verify that the machine-designed schemes are correct and secure?

We believe that a domain-specific language and compiler-based architecture possess

the necessary ingredients for enabling machine-aided cryptographic transformations. We show how such an architecture can aid in automating three general transformations in the literature. As evidence, we present a series of cryptographic compiler tools that realize the implementation of these transformations. More specifically, our tools utilize three main ideas to effectively automate these transformations: 1) a high-level description language to represent and implement cryptographic schemes, 2) an encoding of a cryptographic transformation as a series of simple transformations guided by a concrete set of rules, and 3) a compiler that programmatically applies the encoded rules to abstract specifications and derives executable code from the given specifications.

1.2 Summary of Our Contributions

We present the design of an extensible framework for automating cryptographic transformations and evaluate the framework on three interesting case studies. Our implementation of this framework is embodied in four tools that we describe herein and forms the technical contribution of this thesis.

1. **Charm.** We first introduce Charm [11], a cryptographic library for designing, implementing and evaluating cryptographic primitives and protocols. Our library provides several reusable and modular components that facilitate the rapid prototyping of advanced cryptographic constructions. Using Charm, we implemented over forty cryptographic schemes in the research literature, including new ones that to our

knowledge have never been built in practice. Charm serves as a building block in our architecture for experimentally measuring the effectiveness of the transformations on cryptographic primitives.

2. **AutoBatch.** We present AutoBatch [17], a tool for automatically finding efficient batch verification algorithms from high-level descriptions of digital signature schemes. The tool searches for a batching algorithm by repeatedly applying a combination of novel and existing batching techniques. 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. AutoBatch is an instance of our general architecture described herein and to our knowledge, our tool produced the first batching algorithm for the Camenisch-Lysyanskaya [18] and Hohenberger-Waters [19] signatures in an automated fashion.

3. **AutoGroup.** We present AutoGroup [20], a tool for automatically optimizing the efficiency and bandwidth of pairing-based signature and encryption schemes from high-level descriptions. Traditionally, cryptographers prefer simple, symmetric-based group notation for describing pairing-based schemes, whereas for implementation, they prefer asymmetric-based groups which is more efficient. In practice, converting schemes represented in the former to the latter is non-trivial and can be quite challenging. AutoGroup implements this conversion using our general architecture and leverages SMT solvers to automatically perform the transformation. To our knowl-

edge, this is the first attempt to automatically identify optimal translations for a given cryptographic scheme.

4. **AutoStrong.** Finally, we present AutoStrong [20], a tool for automatically converting an existentially unforgeable signature into one that is strongly unforgeable. Strongly unforgeable signatures are desired in practice for larger cryptographic protocols such as signcryption and chosen-ciphertext secure encryption, just to name a few. Several transformations exist in the literature for achieving strong unforgeability with different requirements for when each transformation is applicable. AutoStrong was also implemented using our architecture and to our knowledge, this is the first attempt to leverage tools such as SMT solvers to determine when certain highly-efficient transformations [12] can be safely applied to a given signature scheme.

1.3 Outline of This Work

Let us now give a layout of the remaining sections of this thesis:

Chapter 2 describe the cryptographic schemes and security definitions we employ in this thesis.

Chapter 3 details our extensible architecture for automating certain cryptographic transformations described herein.

Chapter 4 describes the Charm cryptographic library and its building blocks for facili-

tating rapid prototyping of advanced cryptosystems. Charm provides the necessary components to implement and measure the performance of cryptographic schemes.

Chapter 5 describes the first case study on constructing batch verification algorithms. In this chapter, we present AutoBatch which automates the process of designing batch verification algorithms.

Chapter 6 describes the second case study on symmetric-to-asymmetric translations and a third case study on constructing strongly unforgeable signatures. In this chapter, we describe the implementation of the automated tool called AutoGroup for the second study and AutoStrong for the third study.

Chapter 7 concludes by enumerating the main points of this thesis.

In the appendix, we provide additional material including cryptographic scheme descriptions that serve as test cases for our automated transformations, machine-generated proofs and details on our high-level description language that we utilize in this work.

Previous Publications. A majority of this work has previously appeared in the proceedings of other venues. Notably, Chapter 4 was originally published in the Journal of Cryptographic Engineering (JCEN) 2013 [11]. An extended abstract of Chapter 5 was originally published in the proceedings of the Association for Computing Machinery (ACM) Conference on Computer and Communications Security, (CCS) 2012 [17]. Similarly, Chapter 6 is based on work that appeared in the proceedings of ACM CCS 2013 [20].

Chapter 2

Cryptographic Preliminaries

Before we present the design of our extensible architecture, we must first describe several concepts that are integral to our approach. We focus our discussion on bilinear (or pairing) groups, digital signatures and public-key encryption schemes. We provide security definitions and various types of signatures that we will employ in later chapters. Finally, we describe encryption schemes that we optionally consider for some cryptographic transformations.

2.1 Notation

We begin by describing the notation we use throughout this thesis:

- **Running Time.** By a *p.p.t* algorithm or adversary we are typically referring to a probabilistic, polynomial-time Turing machine.

- **Security parameter.** The cryptographic schemes that we discuss make use of a security parameter 1^{λ} where λ is an integer. For example, this parameter is used to initialize schemes typically during key generation and defines the level of security for the scheme. A large security parameter λ determines the difficulty for a *p.p.t* adversary to break the scheme.

- **Negligible Function.** A negligible function $\varepsilon(\cdot)$ is defined such that for all polynomial functions $p(\cdot)$ and a sufficiently large security parameter ℓ, it holds that $\varepsilon(\ell) < 1/p(\ell)$.

2.2 Bilinear Groups

Let BMsetup be an algorithm that, on input the security parameter 1^{ℓ}, outputs the parameters for a bilinear group $(q, g, h, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$. A bilinear map (or pairing) is an efficiently computable mapping $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, where $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ are multiplicative cyclic groups of prime order $q \in \Theta(2^{\ell})$. A pairing has two important properties: **bilinearity** and **non-degenerate maps**. The bilinear property is that for all generators $g \in \mathbb{G}_1$, $h \in \mathbb{G}_2$, and $a, b \leftarrow \mathbb{Z}_q$, it holds that $e(g^a, h^b) = e(g, h)^{ab}$; Non-degenerate maps ensure that $e(g, h) \neq 1$. 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 pairing $\hat{e}$ 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 simplic-

ity and compare performance between symmetric and asymmetric pairings in Chapter 6.

2.3 Standard Definitions for Digital Signatures

Definition 2.3.1 (A Digital Signature)**.** A *digital signature scheme* is a tuple of 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. The output 1 denotes a valid signature while 0 denotes an invalid one.

A scheme is typically said to be *correct* (or perfectly correct) if for all $\mathsf{Gen}(1^\ell) \rightarrow (pk, sk)$ and for all m in the message space,

$$\mathsf{Verify}(pk, m, \mathsf{Sign}(sk, m)) = 1$$

That is, a scheme is correct if all honestly generated signatures pass the verification test. Our focus will be on correct schemes, however, we discuss in Section 5.3.1 the implications for batch verification if some correctness error is allowed.

We present two security definitions for signature schemes that we refer to in this work:

Definition 2.3.2 (Existentially Unforgeable)**.** A scheme is defined to be existentially unforgeable under an adaptive chosen-message attack if for all p.p.t adversaries A the success probability of A is negligible in the following game [21]: Let $\mathsf{Gen}(1^\ell) \rightarrow (pk, sk)$. Suppose the pair (m, σ) is output by A who has access to the input pk and makes queries to a signing oracle $O_{sk}(\cdot)$, obtaining signatures on as many messages as it wants. Let M denote the set of messages m queried to $O_{sk}(\cdot)$ by A. Then the probability that m was not queried to $O_{sk}(\cdot)$ (i.e., $m \notin M$) and yet $\mathsf{Verify}(pk, m, \sigma) = 1$ must be negligible in ℓ.

Definition 2.3.3 (Strongly Unforgeable)**.** A scheme is defined to be strongly unforgeable under an adaptive chosen-message attack if for all p.p.t adversaries A the success probability of A is negligible in the following game [13]: Let $\mathsf{Gen}(1^\ell) \rightarrow (pk, sk)$. Suppose the pair (m, σ) is output by A who has access to the input pk and makes queries to a signing oracle $O_{sk}(\cdot)$, obtaining signatures on as many messages as it wants. Let $Q = \{(m_i, \sigma_i)\}$ be the set of pairs where m_i denotes the i-th message query by A to $O_{sk}(\cdot)$ and the σ_i denotes the resulting signature. Then the probability that the pair (m, σ) was not among the pairs queried to $O_{sk}(\cdot)$ (i.e., $(m, \sigma) \notin Q$) and yet $\mathsf{Verify}(pk, m, \sigma) = 1$ must be negligible in ℓ.

We apply cryptographic transformations to regular signatures as described in Definition 2.3.1 and also some variants which we now describe below:

- **Identity-Based Signatures (IBS)** [22]: IBS was originally conceived by Adi Shamir in 1984. It consists of a key generation algorithm that is executed by a master author-

ity who publishes the public key and uses the master secret key to generate signing keys for users according to their public identity string. 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.

- **Privacy Signatures**: Group signatures were first introduced by Chaum and Van Heyst [23] in 1991 with two important properties: traceability and anonymity. For anonymity, a signature is 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 signed the message. For traceability, it allows a designated group manager with his secret key to extract the identity of the member that signed a given message. Ring signatures [24] are similar to group signatures in that they provide anonymity but not traceability.

- **Verifiable Random Functions (VRF)** [25]: A VRF is a pseudo-random function, where the computing party publishes a public key 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.

- **Structure-preserving Signatures (SPS)** [26]: SPS is a signature in which the public key, messages and signatures are all elements of pairing groups. The verification consists of checking conjunctions of pairing-product equations against the public

key, and messages/signature [26]. This structure-preserving property of the messages enable these signatures to be combined with the Groth-Sahai non-interactive zero-knowledge (NIZK) proof system [27].

2.4 Standard Definitions for Public Key Encryption

Definition 2.4.1 (Public-Key Encryption)**.** A *public-key encryption scheme* is a tuple of p.p.t algorithms ($\mathsf{Gen}, \mathsf{Encrypt}, \mathsf{Decrypt}$):

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

2. $\mathsf{Encrypt}(pk, m) \to C$: the randomized encryption algorithm takes as input a public key pk and a message m from the message space and outputs a ciphertext C.

3. $\mathsf{Decrypt}(sk, C) \to m$: the deterministic (or possibly randomized) decryption algorithm takes as input a secret key sk and a ciphertext C, and outputs a message m or $\perp$ if a decryption error occurred.

A scheme is typically said to be *correct* (or perfectly correct) if for all $\mathsf{Gen}(1^\ell) \to (pk, sk)$ and for all m in the message space,

$$\mathsf{Decrypt}(sk, \mathsf{Encrypt}(pk, m)) = m$$

That is, a scheme is correct if all honestly generated ciphertexts pass this test.

We optionally consider some variants of public-key encryption schemes:

- **Identity-Based Encryption (IBE)** [22, 28]: IBE was first conceived by Adi Shamir in 1984 to address the shortcomings of public-key distribution. Boneh and Franklin [28] realized the first practical and efficient construction using pairing groups in 2001. IBE is a form of public-key encryption where the public key is an identity (*e.g.*, email address in the form of a string). Users obtain a decryption key for their public identity from a master authority and can decrypt messages that are encrypted to their identity.

- **Attribute-Based Encryption (ABE)** [29–31]: ABE was first proposed by Sahai and Waters [32] in 2004. ABE is a generalization of IBE where the public identity is a set of attributes. Users can only decrypt if the attributes associated with their private key matches certain attributes specified in the ciphertext.

- **Broadcast Encryption (BE)** [33]: BE was first introduced by Fiat and Naor in 1994. It enables encrypting a message to only a subset of qualified users. Only qualified users listening to the broadcast channel can decrypt using their private key. Revoked users are not able to decrypt and even if these users collude, they cannot obtain any information about the contents of the broadcast. [33]

Chapter 3

Extensible Architecture for Automation

3.1 Overview

In this chapter, we present an extensible architecture that can automate the design of certain cryptographic transformations. Our approach demonstrates how to safely and effectively outsource a class of general transformations to machines. We describe a novel high-level description language geared for abstractly representing cryptographic primitives. Furthermore, we show how cryptographic compilers can be designed around this language to automate transformations successfully. In particular, we describe three case studies of general transformations in the literature that we automate in this work. We present them in an increasing order of complexity.

3.2 Background

Before we describe our architecture, we must first provide some background on the building blocks we employ in our automation. Our architecture utilizes external tools such as the Z3 Satisfiability Modulo Theories (SMT) solver to assist in automating some aspects of cryptographic design. Z3 [34, 35] is a freely-available, state-of-the-art and highly efficient SMT solver developed by Microsoft Research. SMT is a generalization of boolean satisfiability (SAT) solving, which determines whether assignments exist for boolean variables in a given logical formula that evaluates the formula to *true*. SMT solvers builds on SAT to support many rich first-order theories such as equality reasoning, arithmetic, and arrays. In practice, SMT solvers have been used to solve a number of constraint-satisfaction problems and are receiving increased attention in applications such as software verification, program analysis, and testing. Z3 in particular has been used as a core building block in API design tools such as Spec#/Boogie [36, 37] and in verifying C compilers such as VCC.

Additionally, we utilize the development platform provided by Wolfram Research's Mathematica [38] (version 9), which allows us to simplify equations for several of our analytical techniques. We leverage Mathematica in our automation to validate that given cryptographic algorithms have certain mathematical properties.

3.3 Overview of Transformation Tasks

We now describe the three cryptographic transformations currently done by hand that we believe can be securely automated. The following is an overview of each transformation:

1. **Construct batch verification schemes.** A batch verification scheme is a probabilistic algorithm that accepts a set of signatures if and only if each signature would have been accepted by its verification algorithm individually. The idea is that valuable verification time will be saved by processing many messages and signatures together than separately. Given these advantages, batch verification techniques are utilized in practice for many applications. In addition, batching algorithms employ techniques that reduce the probability of accepting invalid signatures within a batch. The goal is to automate the design of secure and efficient batch verification schemes for pairing-based signatures.

2. **Optimize the efficiency and bandwidth of cryptographic schemes.** Pairing-based encryption and signature schemes are usually written using a simple symmetric group notation ($\mathbb{G}_1 = \mathbb{G}_2$), but practitioners often prefer implementation in an asymmetric group ($\mathbb{G}_1 \neq \mathbb{G}_2$) due to its efficiency gains. For example, in asymmetric groups, group operations in $\mathbb{G}_1$ are significantly more efficient than operations in $\mathbb{G}_2$. Converting from symmetric to asymmetric settings requires altering the cryptographic scheme such that group elements are given either $\mathbb{G}_1$ or $\mathbb{G}_2$ assignments, but not both.

The goal is to automate the conversion from symmetric to asymmetric schemes and find optimal solutions based on users' efficiency preferences.

3. **Convert an existentially unforgeable signature into a strongly unforgeable signature.** Many signatures are defined under the standard existential unforgeability definition which guarantees that an adversary cannot produce a signature on a *new* message. Whereas, the strong unforgeability definition provides a more powerful guarantee that an adversary cannot produce a *new* signature even on a previously signed message. Several transformations exist in the literature for transforming an existentially unforgeable signature into one that is strongly unforgeable. Some transformations (*e.g.*, Boneh, Shen, and Waters [12]) produce more efficient strongly unforgeable signatures than others (*e.g.*, Bellare-Shoup [39]). The goal is to automatically determine when such efficient transformations are applicable then apply them.

3.4 Our Architecture

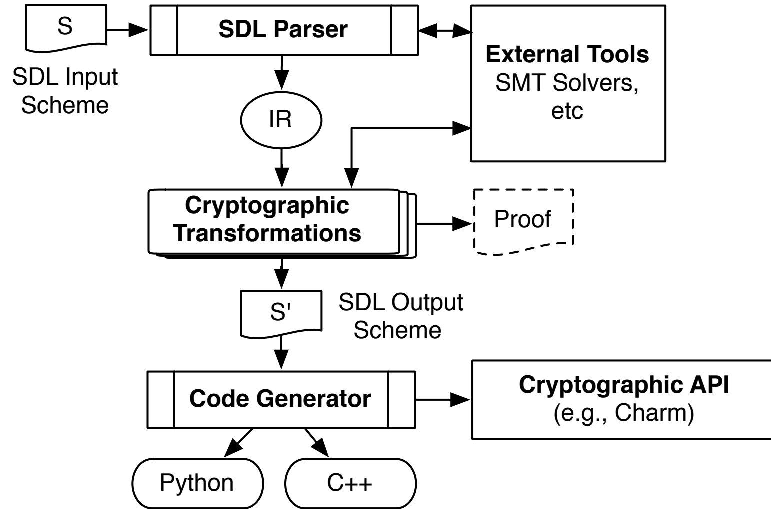


Figure 3.1: At a high-level, the SDL parser takes as input a SDL file description of a cryptographic scheme along with some metadata. The parser converts this input file into an intermediate representation (IR). From this IR, the parser performs type checking utilizing external tools such as an SMT solver. A user-selected cryptographic transformation is applied to the IR, which may also employ external tools to assist with the transformation. The transformation produces a modified SDL file and optionally, a human-readable proof that the transformation preserves the security of the input scheme. The code generator produces a working implementation of the modified SDL in Python and/or C++ using a cryptographic library (*e.g.*, Charm)

We present in Figure 3.1 our approach for automatically applying the aforementioned cryptographic transformations. Our architecture comprises four major components listed as follows:

1. **Scheme Description Language (SDL)**: SDL is a domain-specific language for abstractly representing pairing-based cryptographic schemes; the purpose of SDL is to capture the essence of a cryptographic algorithm using mathematical notation. Our language relieves practitioners of specifying low-level details and instead focuses on

the high-level aspects of the cryptographic algorithm. SDL is the language around which our architecture is built and provides the necessary foundation for effectively implementing cryptographic transformations.

2. **SDL Parser**: parses encryption or signature schemes written in SDL, and translates the SDL file into an intermediate representation (IR) as a series of abstract syntax tree (AST) structures. From this IR, the parser performs type checking and inference using external tools. In addition, during SDL processing, the parser records relationships between variables in SDL, which includes dependencies between variables and how each variable is computed.

3. **Cryptographic Transformations**: represents a series of algorithms and rules that perform transformations on SDL IRs to achieve a design objective. Each encoded algorithm may employ external tools such as SMT solvers to assist in implementing known transformations in the research literature. For transparency, a human-readable proof of security may be *optionally* provided to show that a given transformation preserves the security of the input scheme. Alternatively, verification tools such as EasyCrypt [40] or CryptoVerif [41] may be utilized to provide machine-checkable evidence that the security of the transformed scheme preserves the security of the input scheme. However, achieving such evidence in an automated fashion is currently an open problem and beyond the scope of our current architecture.

4. **Code Generator**: converts original and/or modified SDL of a cryptographic scheme into Python and/or C++ source code. We opted to provide support for statically-typed languages like C++ for environments in which Python (a dynamically-typed interpreted language) is impractical (*e.g.*, embedded devices). As such, the typing information collected by the SDL parser is used here to generate C++ code. Finally, the code generator utilizes a high-level backend cryptographic API to realize working implementations of the SDL descriptions for various public-key encryption and signature types.

3.5 Our Implementation

In this section, we provide details on our approach for implementing each of these components.

3.5.1 Scheme Description Language

Although SDL is a restricted subset of a programming language, it is expressive enough to abstractly describe a variety of cryptographic algorithms. We emphasize that our current focus is on pairing-based digital signatures and public-key encryption schemes. To implement such primitives, SDL includes several basic programming language concepts often used by cryptographic constructs such as functions, conditionals, loops and products. We demonstrate the general syntax and semantics of SDL through scheme examples (see Ap-

pendix A.2). Finally, SDL is a typed language and our toolchain provides extensive type checking and inferencing to simplify the language for users.

An SDL description of a cryptographic scheme consists of several elements to be considered well-formed. First, it must define a few global variables such as the name of the scheme and the pairing setting (*e.g.*, symmetric vs. asymmetric). Because the SDL is parsed in order, the user must declare a type section as the first block in the SDL. Secondly, the remaining blocks consists of the mathematical description of the cryptographic algorithms.

We will now describe the contents of SDL including types, operators, data structures and other basic constructs:

Variable Assignment. Our language supports the basic notion of an assignment statement using the := operator. This is used to set the value of a variable.

Types and Operators. SDL provides five abstract types for describing elements within pairing-based schemes. These basic types consist of Str, Int, ZR, G1, G2, and GT. A Str type typically refers to a bitstring of arbitrary size, $\{0, 1\}^*$, and an Int type is an integer in $\mathbb{Z}$. A ZR type represents an integer in $\mathbb{Z}_r$ where r is the prime order of the group. Finally, G1, G2 and GT refer to the pairing groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$, respectively. Operators +, −, *,/,^ are group operations and in particular, ^ denotes exponentiation.

Data Structures. SDL supports data structures that are commonly used in schemes such as one-dimensional, two-dimensional arrays and tuples. One-dimensional arrays are declared as list{type} where type is one of the basic types. For instance, a list{G1} type

annotation denotes an array of $\mathbb{G}_1$ elements. Two-dimensional arrays are declared similarly. SDL also provides support for tuple data structures. These are useful for describing public keys or ciphertexts which may contain many different element types. Tuples are represented as list{x, y} where x and y are variables with any one of the supported types (including array types).

Built-in Functions. We provide several built-in functions in SDL to simplify and abstract away certain implementation details. For instance, the `random()` function represents the selection of generators in $\mathbb{G}_1, \mathbb{G}_2$ or $\mathbb{G}_T$ and selection of exponents in $\mathbb{Z}_r$. Moreover, we provide a general purpose cryptographic hash function, `H()`, that is often used in schemes.[1] Because SDL is a restricted language, we can support additional abstract functions that are commonly used in a variety of cryptographic constructions, thereby placing more focus on the algorithm.

User-defined Functions. To support representation of cryptographic algorithms, SDL allows for user-defined functions. The syntax is simple and very straightforward:

```
BEGIN :: func:keygen
input := None
 g := random(G2)
 x := random(ZR)
 pk := g^x
 sk := x
output := list{pk, sk, g}
END :: func:keygen
```

We show the key generation algorithm for the Boneh, Lynn, and Shacham (BLS) [42]

[1] `H()` takes two arguments: first is the input variables and the second is the target group for the output group element.

signature scheme. Intuitively, the `BEGIN::func:keygen` denotes the beginning of a function block while the `END::func:keygen` denotes the conclusion of the function. Furthermore, the `input` and `output` keywords capture the inputs/outputs of the function.

Conditionals and Loops. SDL provides support for conditional statements which is often implicitly required in scheme descriptions. As an example, we show the BLS verification algorithm:

```
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
```

Furthermore, we provide similar support for loops including products and summations. We show a for loop as an example:

```
j := 0
BEGIN :: for
for{i := 0, N}
j := j + i
END :: for
```

Additionally, products are typically represented as `prod{i:=0,N} on (x*y)` while summation is represented as `sum{i:=1,L} of x`.

In summary, we took a minimalistic approach with the design of SDL. It provides a minimal set of features that are necessary to describe cryptographic algorithms, but it is expressive enough to represent very complex schemes. Extending the language is rather

trivial in the sense that additional built-in functions can be added to support new tools utilized by cryptographic schemes.

3.5.2 SDL Parser

The role of the parser is to process SDL descriptions and transform them into intermediate representations to enable our automation. More specifically, the parser extracts an abstract syntax tree (AST) from the SDL representation of a cryptographic scheme. During extraction of the AST, we record various metadata about the SDL's contents such as variable types, dependencies and how certain variables are computed (*e.g.*, via generators and exponents). For example, in order to record typing information, the parser relies on an SMT solver (*e.g.*, Z3) to assist with type checking and inferring types. In general, the recorded metadata is not only used to support our automated cryptographic transformations but also for code generation.

To ensure correctness, the parser validates that the variables in the SDL are used correctly with respect to their specified or inferred types. Using SMT solvers, we validate that SDL statements are indeed correctly formed in terms of a set of rules that describe our type system. For instance, a simple rule expressed in the SMT solver for group operations (*e.g.*, multiplication) is that the variables must have the same type. To encode such rules, we rely on features such as uninterpreted functions, abstract data types and quantifiers to model the SDL type system. Moreover, we utilize the theory of arrays in the SMT solver to model the use of data structures within SDL. Our approach here will enable extending SDL to new

cryptographic settings without onerous effort.

Our main technique for inferring types is to convert SDL statements into logical formulas, then evaluates them against a model of the SDL type system. Our parser translates the SDL into an equivalent formulation in Z3 replacing variable names with known types. For usability, the parser notifies the user when variable types cannot be inferred and in such situations, the user is required to provide type annotations. To illustrate our type system, consider the following SDL statement: `a := b * (c ^ d)`. If $b, c \in \mathbb{G}_1$, and $d \in Z_r$, then the generated Z3 input is: `a = mul(G1, exp(G1, ZR))`. Once we feed this formula into the Z3 model of our type system to resolve the types, the output is $a = \mathbb{G}_1$. Therefore, if an assignment statement is well-formed and variables have the correct types, then the solver will also produce a correct type for that assignment. Otherwise, a type violation (`nil`) is reported. Indeed, our approach to validating the types is fairly straightforward and has been tested on several encryption and signature schemes from the literature that we have encoded in SDL.

As indicated earlier, the parser records how certain variables are computed in terms of their base generators and exponents. For example, this information can provide a complete picture on how secret-keys or signatures are constructed. This feature is particularly useful as a building block in some of the cryptographic transformations we discuss in this work and is considered a limited form of *term rewriting*. Moreover, the parser records the variables that a given variable influences and depends on. The influence metadata is considered a forward analysis while the dependency metadata is a backward analysis on each defined

variable. For instance, consider two simplistic statements $a = b + c$ and $e = a + d$, the outcome of variable a is dependent on b and c. Therefore, the parser would record the b and c in the dependency list of a. Similarly, the influence list for b would include a and e. These relationships are helpful in situations where a *program slice* is required of a variable of interest in either direction (influence vs. dependent). In fact, one of the automated transformations we discuss rely on this particular feature to achieve the design objective (*e.g.*, AutoGroup).

3.5.3 Cryptographic Transformations

In this section, we describe the existing design challenges with respect to three case studies of general transformations in the literature. Then, we discuss our approach to automating these transformation using external tools such as SMT solvers to assist with portions of the design. We summarize our approach for each case study by comparing the existing manual process to our automated transformation. We provide a security analysis and discuss our automated implementations of the transformations in more detail in Chapters 5 and 6.

3.5.3.1 Batching Digital Signatures

Pairing-based signature schemes are attractive due to their small size and privacy-friendly nature for several applications (*e.g.*, vehicle-to-vehicle communication, embedded sensor networks). However, the verification of these signatures are expensive due to the

cost of computing pairings. Fortunately, these schemes are conducive to batch verification, where valuable time is saved by processing many messages and signatures together in a batch. Given these advantages, batch verification algorithms are desired for many signature schemes in practice.

Batch verification was first introduced by Fiat [43] for a variant of RSA signatures [44] in 1989. Since then, many research efforts have explored the security and efficiency aspects of batch verification with mixed results. In particular, several batching algorithms have been proposed for well-known signature schemes (*e.g.*, RSA, DSA [45] and etc), but many of them have been shown to be insecure [46–50]. Although the process of deriving batch verification algorithms is relatively straightforward, mistakes are common and generic methods for batching securely have often been misapplied.

Despite these issues, a few positive results have demonstrated that batch verification can be done in a secure and consistent manner. In 1998, Bellare, Garay and Rabin introduced generic methods for securely batching modular exponentiations using randomness. One proposed technique is called the small exponents test and is described in more detail in Section 5.3.4. More recently, Ferrera, Green, Hohenberger and Pedersen [51] in 2009 adapt techniques introduced by Bellare *et al.* [52] to securely and efficiently batch pairings. However, leveraging these techniques manually can be tedious given the complexity of pairing-based verification procedures (*e.g.*, Waters09 [53]). Using our architecture, we believe it is possible to systematically transform an individual verification scheme into a secure and efficient batch verification scheme.

We first recall the high-level process for securely deriving batch verification algorithms then describe the automation of this transformation using our architecture.

Manual Process. Batch verification is one of the more natural general transformations in the research literature. At a high-level, the process begins with a secure and correct verification equation of a signature scheme and proceeds with applying two general steps to derive a batch verification algorithm.

Step 1: Combine Instances and Randomize Verification. This consists of combining η instances of the verification equations where η denotes the size of a batch. This single computation symbolizes the verification of all signatures at once. As indicated by Bellare *et al.* [52], randomizing the verification to reduce the probability of accepting an invalid signature in the batch is crucial for securely batching signatures. After combining the instances of the equation and randomizing the verification using the small exponents, this forms an initial batch verification equation for the signature scheme.

Step 2: Optimize Batch Equation and Generate Complete Algorithm. The next step is to optimize the batch verification equation by applying the techniques described in the work of Ferrera *et al.* [51] in any order. To derive a complete batch algorithm, it remains to perform group membership tests on elements of the signature and to apply a suitable method for detecting invalid signatures in a batch. A straightforward approach is the divide and conquer method introduced by Law and Matt [54] where a batch is divided into two halves, then recursively perform batch verification on each half and repeat until all invalid signatures have been identified.

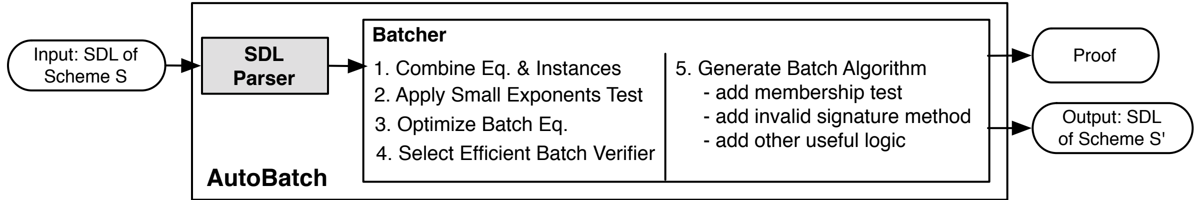


Figure 3.2: A high-level presentation of the AutoBatch tool, which automates finding efficient batch verification algorithms.

Automated Process. We begin with an SDL description of a signature scheme and extract the verification equations to form an abstract syntax tree (AST) of the equations. Each step described above is implemented as a series of simple transformations on the AST representation. A high-level of the automated tool is shown in Figure 3.2 and we describe the automated steps below.

Step 1: Combine Instances and Randomize Verification. We implement a transformation on the AST that introduces small random exponents and products to the AST representation. This denotes verification over η instances of the signatures and forms a secure batch verification algorithm. Furthermore, this phase of the automation also handles cases where there are multiple verification equations in one signature scheme. In this case, our logic would consolidate these equations also using the small exponents before combining the η instances. We discuss these cases and other variations in more detail in Chapter 5.

Step 2: Optimize Batch Equation and Generate Complete Algorithm. Optimizing the batch equation is the most technically challenging portion of this step. Practitioners are able to intuitively discern when the optimization rules are applicable for simple schemes, but this

becomes tedious as the complexity of the signature scheme increases. Determining the order in which optimization techniques should be applied in an automated sense is non-trivial. We select the best batch verifier automatically through a pruned breadth-first search algorithm. The pruning is achieved by a heuristic function which enables us to uncover the best order to derive an optimized batch equation. We provide more details on our automated search and heuristic function in Chapter 5.

Upon identifying the optimized batch equation, it remains to generate the rest of the batch algorithm by adding explicit logic for performing group membership tests on elements of the signature and public key. Additionally, we cache certain computations in the batch equation in preparation for detecting invalid signatures using the divide and conquer method. Finally, we output a modified SDL that contains the complete batch verification algorithm.

3.5.3.2 Optimizing Cryptographic Schemes

Often, pairing-based cryptographic schemes are presented in the literature using symmetric-group notation. In symmetric groups, $\mathbb{G}_1 = \mathbb{G}_2$ or there exists an efficient isomorphism from $\mathbb{G}_1$ to $\mathbb{G}_2$ and vice versa. While symmetric notation simplifies the description of new cryptographic schemes, the corresponding groups are rarely the most efficient setting for implementation [55]. Asymmetric groups represent the state of the art in terms of efficiency where $\mathbb{G}_1 \neq \mathbb{G}_2$ and no efficient isomorphism exists between the two groups.

Translating from symmetric to asymmetric groups is a non-trivial exercise and is cur-

rently done through manual analysis of a cryptographic scheme. As an example, the work of Ramanna, Chatterjee, and Sarkar [56] translates the Waters [57] dual system encryption scheme from symmetric to several asymmetric schemes. These conversions are made difficult by restrictions to certain types of asymmetric groups (*e.g.*, hashing operation only supported in $\mathbb{G}_1$). For some schemes, there are hundreds of possible asymmetric translations and identifying the optimal translation for a given application is quite challenging in practice. We believe that this translation can be efficiently automated with the aid of an SMT solver.

We first recall the process for converting a scheme from the symmetric to asymmetric setting. We then describe our automation of this translation using Z3 and how we identify the optimal translation for a given set of application-specific requirements.

Manual Process. A conversion of a signature or encryption scheme in the symmetric setting to an asymmetric one is typically broken down into three general steps:

Step 1: Identify Asymmetric Assumptions. The first objective of a practitioner is to identify the asymmetric assumptions to base the scheme. For common hardness assumptions such as Computational Diffie-Hellman (CDH) or Bilinear Diffie-Hellman (BDH), there exists analogous asymmetric assumptions: co-CDH and co-BDH. However, such analogies may not exist for arbitrary symmetric assumptions. For cases where a scheme is based on a symmetric assumption without an asymmetric counterpart, one could attempt to find a related or perhaps stronger asymmetric assumption to reconstruct the scheme. This route requires re-imagining the scheme under the new asymmetric assumption which can be non-

trivial.

Step 2: Select New Generators and Determine Group Assignments. Once the target asymmetric assumptions have been identified, proceed with selecting the new generators for the scheme for both groups $\mathbb{G}_1$ and $\mathbb{G}_2$. Then, determine the group assignments for elements of the secret-key and ciphertext based on the user's optimization objectives. Similarly, for signatures, one would determine group assignments for public-key and signature elements. An optimization objective could be a user who desires an asymmetric solution in which the signature has a short representation. After group assignments have been determined, the next step is to compute each element using the appropriate generators of the assigned group.

Step 3: Verify Correctness and Prove Security. The last step is to verify that the inputs to the pairing are either assigned to $\mathbb{G}_1$ or $\mathbb{G}_2$. If this verification is successful, the practitioner can now proceed with proving that the new variant is secure with respect to the appropriate asymmetric assumptions.

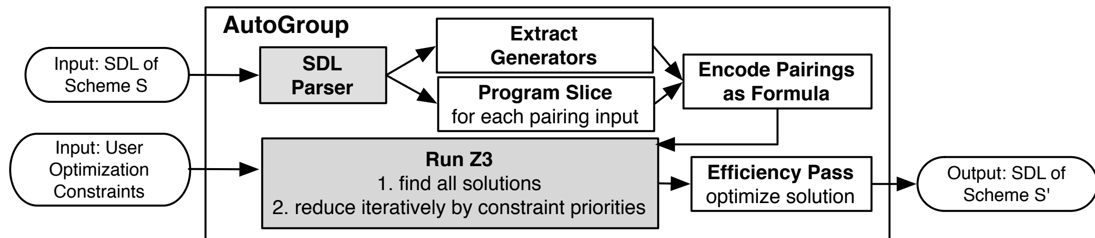


Figure 3.3: A high-level presentation of the AutoGroup tool, which optimizes cryptographic schemes specified in the symmetric setting.

Automated Process. We recall that asymmetric pairings have a single restriction on their inputs: $\mathbb{G}_1 \neq \mathbb{G}_2$. Consequently, one technical challenge is how to automatically find suit-

able solutions to the group assignment problem. Once possible group assignments have been determined, a separate challenge is finding the optimal translation for given user requirements. We make the observation that one can view the group assignment problem as an instance of boolean satisfiability. As such, an SMT solver can be employed to assist in identifying the optimal translation. Figure 3.3 shows a high-level view of our implementation.

Step 1: Identify Asymmetric Assumptions. In general, our automated translation does not attempt to reconstruct a scheme using stronger complexity assumptions. Rather, we assume that there exists an equivalent asymmetric assumption to base the variant scheme. In the event that the input scheme requires both an efficient isomorphism and hashing to $\mathbb{G}_2$, then it might not be realizable in the asymmetric setting. We discuss the issues further in Chapter 6.

Step 2: Select New Generators and Determine Group Assignments. Using the information recorded by the parser, we determine which algorithm is responsible for parameter generation in SDL and extract the generators used by the scheme. The idea is to recreate these generators in the asymmetric setting for both $\mathbb{G}_1$ and $\mathbb{G}_2$. The group assignment decisions made by the SMT solver will dictate which generators to use, so we might not use all of them.

We first encode constraints over asymmetric pairings in terms of inequality operations separated by conjunctions (*e.g.*, $A \neq B \wedge C \neq D$, etc). We then feed this logical formula (and general constraints over the target asymmetric group) into the solver to obtain possible

solutions. These solutions are the set of all possible variable mappings of assignments that satisfy user constraints. To select the optimal one, we utilize the solver to minimize an objective function over the solutions.

The purpose of the objective function is to identify a minimal solution that corresponds to the constraint priorities. Once a minimal solution is obtained, we convert this solution into an asymmetric solution for the input scheme. In order to do this, we also extract all the pairing inputs in the scheme and compute a program slice on each input variable. Each slice helps in navigating which variables are affected as we rewrite the scheme in the asymmetric setting. We discuss more details and features of our translation in Chapter 6.

Step 3: Verify Correctness and Prove Security. Using the solver, we are able to correctly identify candidate solutions that satisfy the constraints on the asymmetric scheme. We output the optimal translation in SDL and it remains for the practitioner to manually prove the security of the variant against the appropriate asymmetric assumptions.

3.5.3.3 Constructing Strongly Unforgeable Signatures

As indicated before, many signature schemes in the literature are presented under the existential unforgeability definition wherein an adversary cannot produce a signature on a *new* message. This is a traditional definition introduced by Goldwasser, Micali and Rivest [21] for signatures and provides a minimum level of security in the face of an adaptive chosen message attack. However, strong unforgeability guarantees more – that the adversary cannot produce a *new* signature even on a previously signed message. In prac-

tice, strongly-unforgeable signatures are crucial for a variety of applications and often used as a building block in signcryption [13], chosen-ciphertext secure encryption [14, 58], and group signatures [15, 59].

There are several general transformations in the literature for obtaining strongly unforgeable signatures from existentially unforgeable signatures. We focus specifically on the highly-efficient transformation due to Boneh, Shen and Waters (BSW) [12] that only applies if a signature satisfies a notion of partitioning (defined below). If the signature is not partitioned, then a less-efficient transformation due to Bellare-Shoup (BS) [39] can be applied. Given that both transformations achieve strong-unforgeability, the main challenge is deciding when the highly-efficient transformation is applicable. That is, automatically identifying when a signature satisfies the definition of the partitioning property. We believe that this partitioning check is amenable to our automated techniques using tools like SMT solvers as a core building block.

We first recall the process for determining whether a signature is partitioned according to the BSW transformation then we describe the automation of detecting this property.

Manual Process. In the BSW [12] transform, a partitioned signature is defined as having the following two properties:

Property 1. Break down the signing algorithm into two deterministic functions, F_1 and F_2 so that a signature on a message, m, using secret, sk, is computed as follows:

1. Set $\sigma_1 \leftarrow F_1(m, r, sk)$ and $\sigma_2 \leftarrow F_2(r, sk)$, where r is randomly selected in R.

2. Output the signature $\sigma \leftarrow (\sigma_1, \sigma_2)$

Property 2. Given m and σ_2 there is at most one σ_1 so that (σ_1, σ_2) verifies as a valid signature on m under the public-key, pk.

The main idea is that if half of the signature, σ_2, does not depend on the message, m, and property 2 holds, then the signature is considered partitioned and the BSW transformation can be applied. Otherwise, apply the general BS transform which converts *any* unforgeable signature to a strongly unforgeable one. We discuss the details of the BSW and BS transformations in Chapter 6.

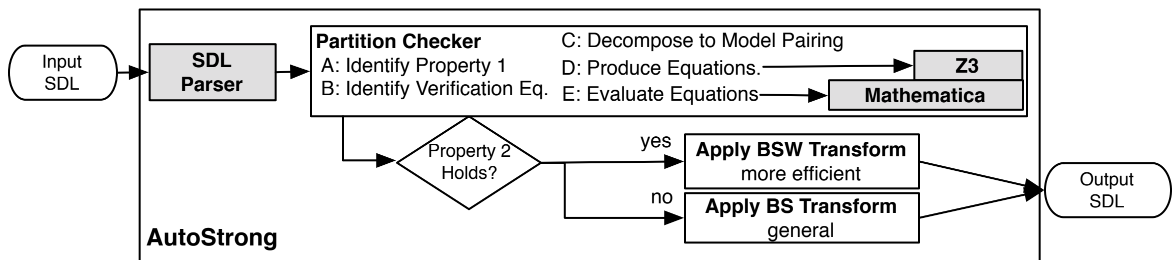


Figure 3.4: A high-level presentation of the AutoStrong tool, which automates the construction of strongly unforgeable signatures.

Automated Process. We begin with an SDL description of the signature and assume it is existentially unforgeable, then proceed with checking both properties. We extract an AST representation of the Sign and Verify algorithms as a whole. During extraction, we analyze the Sign algorithm to determine how the signature is computed and proceed to check for each property as follows:

Property 1. We divide the signature by categorizing components of the signature into either

σ_1 or σ_2. The differentiating factor between the two is that σ_1 variables depend on the message and σ_2 ones do not. Specifically, we obtain the program slice of each element of the signature and determine if m is used in any part of that element's computation. If so, then we add to the list of σ_1 variables and otherwise, add to the list of σ_2 variables.

Property 2. We observe that property 2 can be restated as such: does there exist a σ'_1 such that $\sigma_1 \neq \sigma'_1$ and signature pairs, (σ_1, σ_2) and (σ'_1, σ_2) both *verify* under m and pk. If such a pair exists, then property 2 does not hold and the signature may not be partitionable. To prove such a property in an automated fashion, our partition checker needs to somehow mathematically evaluate the verification equation on the inputs to determine if a contradiction can be found. Specifically, our checker attempts to prove that this logical statement does not hold: $\sigma_1 \neq \sigma'_1 \wedge \mathsf{Verify}(pk, m, (\sigma_1, \sigma_2)) = 1 \wedge \mathsf{Verify}(pk, m, (\sigma'_1, \sigma_2)) = 1$.

Since we restrict ourselves to pairing-based verification procedures, our main challenge is figuring out how to model the behavior of pairings to establish the validity of property 2 for a given signature scheme. One crucial observation is that the bilinearity property of pairings can be modeled in the exponent using Z3. In particular, we utilize Z3 to reduce the pairing-based verification equation into a simple integer equation. Once an integer equation is obtained, Z3 is less suited to mathematically evaluate such equations and instead, we leverage the Mathematica equation reasoning techniques. If zero or one solution exists to the system of integer equations, only then is the signature scheme considered partitioned. Otherwise, the signature may not be partitionable. We provide further details on our implementation in Chapter 6.

3.5.4 Code Generator

The purpose of the code generator is to translate abstract SDL descriptions into concrete cryptographic implementations. It is our experience that several cryptographic schemes in the literature have never been built. Absent such implementations, it can be difficult to measure the effectiveness of certain transformations on abstract descriptions of cryptographic primitives. Our architecture requires a backend cryptographic framework that provides a sufficient level of abstraction for implementing primitives. Such a framework enables us to concretely evaluate SDL descriptions and can inform how implementations of transformed primitives will impact potential applications.

Our code generator produces concrete cryptographic implementations using the Charm framework [11]. As mentioned before, Charm is a framework we developed to address the lack of implementations for cryptographic schemes and provides the necessary building blocks to realize a variety of cryptographic primitives and protocols. Charm was designed to use mathematical notation familiar to cryptographers and closely resembles our SDL as well. Charm supports both dynamically interpreted languages such as Python and statically-typed languages such as C++ with the same programming API for consistency.

Charm is suitable for our purposes and facilitates automatically generating concrete implementations of abstract SDL descriptions. We provide more details on the design and implementation of Charm in Chapter 4. Our code generator produces working implementations in both Python and C++; it utilizes the typing information recorded by the SDL parser to support C++. We remark that our code generator can be trivially extended to

support additional languages.

3.6 Literature Review

There are several research efforts towards automating the design of various aspects of cryptography. In these efforts, a cryptographic compiler-like approach has been applied in the design of various security protocols, secure multi-party computation, and zero-knowledge proofs. We discuss in detail each area and how it relates to our efforts to automate the design of cryptographic transformations for digital signatures and public-key encryption schemes.

Security Protocols. Several researchers have tackled the automation of security protocol design using a compiler-like technique. Here we refer to security protocols that deal with authentication, key exchange and etc. Lowe [60] investigated how to automate the analysis of security protocols. More specifically, they propose a tool, called Casper, to simplify the process of deriving process algebra Communicating Sequential Processes (CSP) from abstract descriptions of security protocols (*e.g.*, key exchange, authentication, etc). This requires domain expertise and the tool automatically converts abstract protocol descriptions into CSP code. In addition, the tool leverages a model checker to analyze the protocol against a security specification and finds concrete attacks against protocols. Song, Perrig and Phan [1] proposed a toolkit for generating secure implementations of security protocols. The automatic generation, verification and implementation (AVGI) toolkit takes as

input a protocol specification, application requirements and then attempts to find an optimal protocol design for the given application. Finally, the tool translates the design into a Java implementation. Pozza, Sisto, and Durante [2] proposed a Spi2Java tool that generates Java implementations from a formal spi calculus and detects protocol flaws as well. Lucks, Schmoigl, and Tatli [3] delve into the design issues with respect to cryptographic compilers for security protocols. They introduce an experimental language for an ideal abstract specification for representing protocols and generating source code. Kiyomoto, Ota and Tanaka [61] also proposed a compiler that takes as input a high-level specification of a security protocol in eXtensible Markup Language (XML) and a security definition file and automatically generates C modules of that specification. The use case is for dynamically generating protocol implementations for web services to maintain protocol agility.

Zero-Knowledge Proofs. Zero-Knowledge (ZK) proofs are an essential component of privacy-preserving cryptography and have inspired many research efforts to automate various aspects of their design and implementation. Camenisch, Rohe, and Sadeghi [4] proposed a design and implementation of a compiler called Sokrates for designing efficient zero-knowledge proofs of knowledge on one-way homomorphisms. Backes, Maffei, and Unruh [5] explore abstractions of non-interactive zero-knowledge (NIZK) proofs using applied pi-calculus. They leverage and devise an equational theory for abstractly characterizing semantics of NIZK proofs. The authors' approach transforms the abstractions into formalisms that can be verified using ProVerif and apply the theory to mechanize the verification of a direct anonymous protocol that utilizes these proofs.

Bangerter, Briner, Henecka, Krenn, Sadeghi, and Schneider [6] introduce a specification language for describing zero-knowledge proof of knowledge (ZK-PoK) protocol specifications. Moreover, they propose a compiler that translates the Σ-protocol specifications into Java implementations. Almeida, Bangerter, Barbosa, Krenn, Sadeghi, and Schneider [7] extends the work of Bangerter *et al.* and proposes a certifying compiler that transforms abstract descriptions of ZK-PoK goals into provably sound interactive protocol implementations in C. Additionally, their compiler is comprehensive in that it supports a number of proof composition techniques in the literature (*e.g.*, AND, OR, and others) and produces a formal proof that the protocol generated fulfills its specification (*i.e.*, proof goal). The proof is formally verified using the Isabelle/HOL formal theorem prover. Meiklejohn, Erway, Küpçü, Hinkle, and Lysyanskaya [8] introduce a similar ZK-PoK compiler for applications such as e-cash and provides precomputation optimizations which is not supported in Almeida *et al.* [7]. However, the ZKPDL compiler is not as comprehensive as the CACE compiler [7].

Almeida, Barbosa, Bangerter, Barthe, Krenn, and Zanella Béguelin [62] present an optimizing and certifying compiler called ZKCrypt for ZK-PoK protocols. In particular, ZKCrypt integrates verified and verifying compilers to produce formal proofs in CertiCrypt. ZKCrypt is fully automated and provides strong assurances that the implementation is secure with respect to the specified abstract proof goal. The authors demonstrate their compiler on anonymous credential protocols. Fournet, Kohlweiss, Danezis, and Luo [63] introduce a query language (called ZQL) for expressing computations on private

data. They design a compiler that transforms ZQL queries into interactive ZK-PoK over the private data and automatically generates F# or C++ of the protocol interactions between client/server. Furthermore, the authors evaluate queries using the compiler for applications that require such privacy guarantees such as smart-meter billing.

Secure two-party computation. A secure two-party computation comprises mutually distrusting parties that want to jointly compute an arbitrary function on private inputs without revealing any information other than the results of the computation on the shared secrets. MacKenzie, Oprea, and Reiter [64] design a compiler that automatically generates efficient, provably secure two-party protocols. The compiler takes as input a high-level description of a cryptographic function such as computing signatures or decrypting ciphertexts and produces as output, the source code implementing each side of the two-party protocol. This work focuses on a specific class of two-party computations that use arithmetic operations over groups and fields and that are efficient enough for practical applications. Malkhi, Nisan, Pinkas, and Sella [9] proposed a generic two-party computation engine called Fairplay. Fairplay takes a high-level description language (SFDL) of a secure computation and compiles it into a boolean circuit. The tool also produces modules that securely evaluate the circuits that represent the desired computation. Fairplay has been extended to multi-party computations in recent work by Ben-David, Nisan, and Pinkas [65]. Henecka, Kögl, Sadeghi, Schneider, and Wehrenberg [10] propose TASTY, a tool for automatically generating, optimizing, implementing and benchmarking two-party protocols based on homomorphic encryption and garbled circuits. Unlike previous works, TASTY automatically

transforms a high-level description of a computation on encrypted data and generates the interactive protocol and corresponding implementation.

Cryptographic Primitives. Our approach also introduces a high-level description language and presents the design and implementation of a compiler for automating the construction of cryptographic schemes. The use of SMT solvers in our architecture and observations of how they can be useful in automating cryptographic transformations is both novel and unique. Our results indicate that this compiler-like approach to designing cryptographic primitives can outperform manual approaches in an efficient and secure manner while producing competitive results.

Chapter 4

Charm: A framework for Rapidly Prototyping Cryptosystems

In the previous chapter, we described our extensible architecture for automating certain cryptographic transformations. As indicated before, the code generator component of the architecture requires a suitable, high-level cryptographic framework to evaluate the effectiveness of the cryptographic schemes that are transformed. The goal of the framework is to provide a usable, extensible, and modular architecture to facilitate rapid prototyping of a variety of cryptographic primitives and protocols from abstract descriptions. The framework described in this chapter serves as the backbone of our architecture and is crucial for validating the results of our cryptographic transformations.

4.1 Overview

In this chapter, we describe Charm, an extensible framework for rapidly prototyping cryptographic systems. Charm provides a number of features that explicitly support the development of new protocols, including: support for modular composition of cryptographic building blocks, infrastructure for developing interactive protocols, and an extensive library of re-usable code. Our framework also provides a series of specialized tools that enable different cryptosystems to interoperate.

We implemented over forty cryptographic schemes using Charm, including some new ones that to our knowledge have never been built in practice. This chapter describes our modular architecture, which includes a built-in benchmarking module to compare the performance of Charm primitives to existing C implementations. We show that in many cases our techniques result in an order of magnitude decrease in code size, while inducing an acceptable performance impact.

Lastly, the Charm framework is freely available to the research community and to date, we have developed a large, active user base.

4.2 Introduction

Recent developments in cryptography have the potential to greatly impact real world systems. Advances in lattices and pairings have driven new paradigms for securely processing and protecting sensitive information such as identity-based encryption [28, 66–69]

and attribute-based encryption [29–32], and privacy-preserving schemes such as ring signatures [70, 71], group signatures [59, 72] and anonymous credentials [16, 73]. Without these kind of advances, a number of results in top security conferences would not be possible [74–76].

Unfortunately, many potentially useful and novel schemes exist only in research papers and have not actually been implemented. A few of these schemes find their way into isolated C libraries that are maintained purely by their creator, executed only as proof of concept and are operated solely in their own limited domain. While elliptic curves and lattices enabled some of these advances, they also substantially increased the complexity: writing software for cryptosystems no longer involves only number theory and modular arithmetic. This is doubly problematic because the size of typical C implementations makes bugs likely and audits hard. The barrier to usage, consequently, remains very high.

There have been a handful of elegant implementations of a small number of new primitives [77–79] as well as some tools for protocol development [80–85]. These systems serve their special purposes well, but are not interoperable, and so developers wishing to build a system using multiple primitives must write non-cohesive *glue* code to piece their implementations together.

In practice, libraries such as Sage [86], the Stanford Pairing-Based Crypto (PBC) [78] and MIRACL [87] fulfill an important role of providing implementations of advanced mathematics for algebra, number theory, and elliptic curves just to name a few. While these libraries provide a solid foundation for developing advanced cryptography, they were

not designed with *usability* or *interoperability* in mind in terms of composing, structuring, and reusing cryptographic primitives. Although this may seem like an engineering detail, serious theoretical issues can arise from the improper combination of cryptographic primitives. Therefore, great care must be taken to accommodate the theoretical foundations of underlying primitives when designing a system that provides robust, composable, and modular cryptography.

Our Contribution. We present Charm[1] [88], a new, extensible and unified framework for *rapidly prototyping* experimental cryptographic schemes and leveraging them in system applications. Charm is built around the concepts of extensibility, composability, and modularity. The framework is implemented in Python, a well-supported high-level language, designed to reduce development time and code complexity while promoting component reuse. Computationally-intensive mathematical operations are implemented as native modules, enabling performant schemes and protocols while preserving the advantages of high-level languages for scheme implementations. Although Charm is written in a dynamically typed interpreted language, the concepts and abstractions developed in this chapter can be realized in a variety of programming languages.

The design goals of Charm are:

Enabling Efficient, Extensible Numeric Computation. New primitives are invented and existing implementations of primitives are optimized on a regular basis. For example, the PBC library [78], one of the original libraries providing pairings, has been sup-

[1] Project webpage: `http://charm-crypto.com`.

planted in terms of performance by alternative libraries such as MIRACL [87] and RELIC [89]. Similarly, lattice-based cryptographic operations are an increasingly desirable feature in scheme development. In practice, the math libraries supporting any given cryptographic operation are subject to change. The challenge is how to enable these changes without disrupting the higher-level scheme.

Supporting Succinct Cryptographic Protocols. Although cryptographic protocols only capture the mathematical formulas on paper, in practice network protocols must embed the necessary logic required for message serialization, data transmission, state transitions, error handling, and the execution of subprotocols. Protocols involving zero-knowledge proof statements are particularly problematic: concrete implementations require explicit information not usually present in an algorithmic sketch. The challenge is to provide an interface for wire protocols roughly equivalent to the way the protocols are specified in research papers.

Supporting Scheme Composition. Composing cryptographic algorithms allows for the rapid creation of new schemes, protocols and facilitates code reuse. Not only does this make implementers more efficient, it improves the security of the system by ensuring there is one canonical version of a given scheme or technique. However, composability creates its own set of hurdles: schemes may use different plaintext and ciphertext spaces, security assumptions and security models. The challenge is abstracting away these differences while preserving the schemes' underlying security

and functionality.

Providing Measurement Capability. Benchmarking and profiling are particularly important, both from a theoretical perspective and an implementation standpoint for complex schemes (*e.g.,* homomorphic encryption). Simple benchmarking allows quick prototyping and comparison of novel variations of naïve implementations of schemes. Profiling enables in-depth optimization of full-fledged schemes with fine-grained performance data. The difficulty is providing both seamless benchmarking and in-depth profiling while maintaining component modularity.

Allowing Application Embedding. Rapid prototyping and ease of use require that the framework be written in a user-friendly, high-level language. If developers outside of the cryptographic community are to build applications with advanced cryptographic constructs, the choice of language is critical. The dilemma is how to provide a level of abstraction (or embedding API) to outside systems without unduly limiting the expressiveness of the framework.

Allowing Cryptographic Algorithm Agility. As noted by Acar *et al* [90], cryptographic algorithms have a limited shelf life. For example, once exhaustive search rendered DES keys insecure, DES was replaced by AES. Similarly, MD5 and SHA1 were discovered to contain vulnerabilities [91, 92]. A system must be designed such that algorithms can be replaced when necessary [93]. Cipher algorithm replacement must be done without compromising security, without breaking functionality, and if possi-

ble, without requiring keys to change.

4.3 Background

We note that practical implementations of advanced forms of encryption such as identity-based encryption (IBE) [28, 66] and attribute-based encryption (ABE) [29–31] typically involve the use of pairings. Recall that a pairing is an efficient mapping $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ over three multiplicative cyclic groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order p. Moreover, a pairing has two properties: bilinearity and non-degenerate maps. Bilinearity is that given generators $g \in \mathbb{G}_1, h \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$ it holds that $e(g^a, h^b) = e(g, h)^{ab}$. Non-degenerate maps ensures that $e(g, h) \neq 1$. Lastly, cryptographic primitives that utilize lattices are an exciting area of research that hold promise for post-quantum cryptography. We briefly mention lattices in this chapter, but defer to Regev's work [94] for an in-depth introduction.

We also discuss techniques for performing transformations over cryptographic primitives to achieve desired security properties. For example, Naor [95] proposed a technique for converting an IBE scheme into a public-key signature scheme. Canetti et al [14] proposed a technique for transforming any IBE scheme into one that is secure against adaptive chosen-ciphertext attacks. In general, we refer to these types of cryptographic transformations as *adapters* in this chapter.

Finally, we refer to Zero-knowledge Proofs of Knowledge(ZK-PoK) [96], which allow one party to prove knowledge of a secret to another party without revealing the secret.

4.4 Approach

Charm realizes the aforementioned goals at the architectural level through various components and levels of modularization as depicted in Figure 4.1.

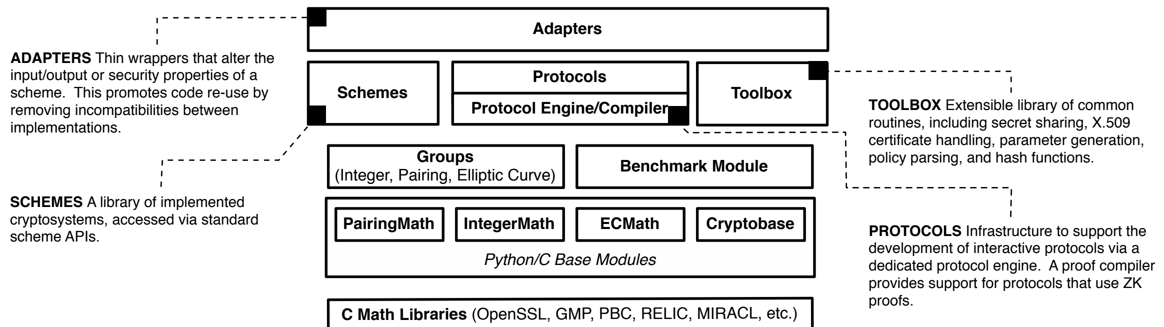


Figure 4.1: Overview of the Charm architecture.

We now describe the building blocks of the Charm framework. The lower-level components, at the bottom of Figure 4.1, are optimized for efficiency, while the ones at the top focus on ease of use and interoperability. One of the primary drivers of our approach is our objective to simplify the code written by cryptographers who utilize the framework. Our modular component architecture reflects this.

Scheme Annotation and Adapters. In practice, implementations of different cryptosystems may be incompatible even if their APIs are the same. For example, two systems might have different input and output requirements. Consider that many public key encryption schemes require plaintexts to be pre-encoded as elements of a cyclic group $\mathbb{G}$, or as strings of some fixed size. These requirements frequently depend on how the scheme is configured, *e.g.,* depending on parameters used. Different developers are unlikely to make all of the same

choices in their implementations, so even if they build their code with a standard API template, their systems are unlikely to interoperate cleanly.

More subtle incompatibilities may arise when schemes of a given class provide differing security guarantees: for example, public-key encryption schemes can provide either IND-CPA [97] or IND-CCA2 [98] security. These properties become more relevant whenever the scheme is used as a building block for a more complex protocol.

Meta-Information. To address these issues, Charm must provide some mechanism to identify the pertinent information inherent in each scheme, including (but not limited to) input/output space, security definition, complexity assumptions, computational model, and performance characteristics. We defer the discussion of whether this should be done automatically or by the programmer to Section 5.5.

Capability Matching. Once this meta-information is collected, Charm uses it to facilitate compatibility among schemes. First, it provides tools to programmatically interrogate a scheme to determine whether the scheme satisfies certain criteria. This makes it easy to substitute schemes into a protocol at runtime, since the protocol can simply specify its requirements (*e.g.,* EU-CMA [21] signature scheme) and Charm will ensure that they are met. To make this workable, Charm includes a dictionary of security definitions and complexity assumptions, as well as the implications between them. Thus, a protocol that requires only an EU-CMA signature scheme will be satisfied if instantiated with an SU-CMA [13] signature, but not vice versa. However, the implication can be bypassed in some cases, for example, if EU-CMA is required and SU-CMA is not suitable for a given composition

where re-randomizable signatures are required.

Structured Interfaces. To facilitate scheme composition and reuse, Charm provides a set of APIs for common cryptographic primitives such as digital signatures, bit commitment, encryption, and related functions. Schemes with identical APIs are identified and are interchangeable in our framework. For example, DSA [45] can be used instead of RSA-PSS [99] within a larger protocol with a simple, almost trivial change to the code.

Scheme interfaces are implemented using standard object-oriented programming techniques. The current Charm interface hierarchy appears in Figure 4.2. This list is sufficient for the schemes we have currently implemented (see Figure 5.7), but we expect it to expand with the addition of new cryptosystems.

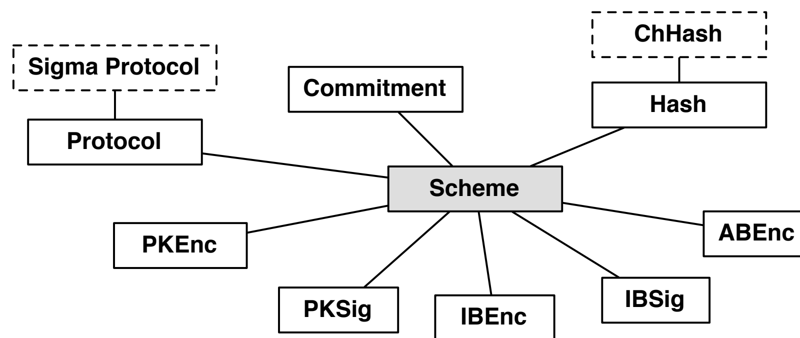


Figure 4.2: Listing of scheme types defined in Charm. Subtypes are indicated with dotted lines.

Adapters. Since we now have enough information to safely and securely compose schemes, Charm includes *adapters* for this purpose and for handling mismatches between schemes. Adapters are code wrappers implemented as thin classes. For example, they permit developers to bridge the gap between primitives with disparate message/output spaces or security

requirements. In our experience so far, the most common use of adapters is to convert an input type so that a scheme can be used for a specific application. For example, we use adapters to encode messages or in the case of hybrid encryption, to expand the message space of a public key encryption scheme.

Adapters can perform even more sophisticated functions, such as modifying a scheme's security properties. In Figure 4.3 we illustrate an adapter using a hash function to perform a conversion from a selectively-secure IBE scheme into one that is adaptively secure (note here that the hash function is modeled as a random oracle).

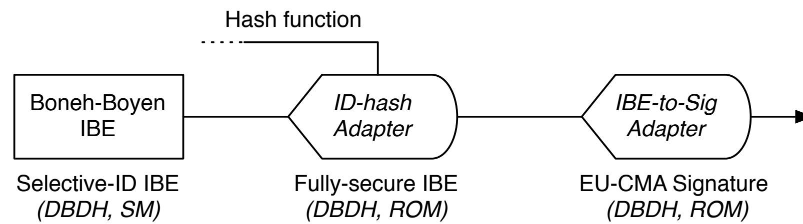


Figure 4.3: Example of an adapter chain converting the Boneh-Boyen selective-ID secure IBE [66] into a signature scheme using Naor's technique [95]. The scheme carries meta-information including the complexity assumptions and computational model used in its security proof.

Adapters can also combine schemes to produce entirely different cryptosystems. This means that there are *implicit* schemes in Charm that do not physically appear in the scheme library, demonstrating Charm's success at the goal of composability. Figure 4.4 provides another example of such a conversion.

Extensible Numeric Computation. The mathematics underlying modern cryptography has changed considerably, driven by advances in lattices and pairings, and is sure to continue in this trend. It is fundamentally important that any system that wishes to maintain rel-

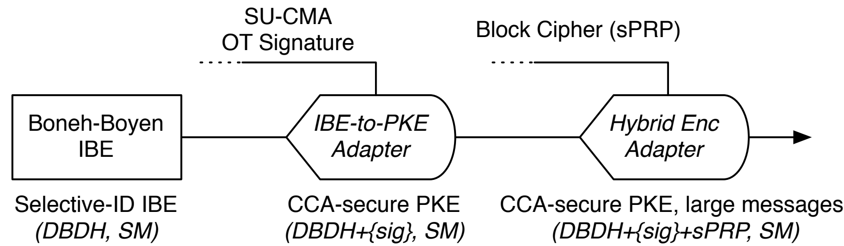


Figure 4.4: Adapter chain converting the Boneh-Boyen selective-ID secure IBE [66] into a CCA-secure public-key hybrid encryption scheme via the CHK transform [14]. $\{sig\}$ stands for the complexity assumptions added by the signature scheme.

evancy be able to incorporate these advances. By necessity, these libraries are implemented in C and require a certain specialty and expertise to implement correctly (*e.g.,* elliptic curves). Charm provides domain separation by incorporating four base modules that implement the core cryptographic routines. This shelters developers from having to deal with very domain-specific concepts like elliptic curves. For performance reasons these base modules are written in C/C++ and include integermath, ecmath (elliptic curve subgroups), and pairingmath.[2] The cryptobase module provides efficient implementations of basic cryptographic primitives such as hash functions and block ciphers. These modules include code from standard C libraries including libgmp, OpenSSL, libpbc, and PyCrypto [77, 78, 100, 101]. To maximize code readability, the module interfaces employ language features such as operator overloading. Finally, Charm provides high-level Python interfaces for constructs such as algebraic groups and fields.

The base modules implement only those lower-level routines where implementation in C is crucial for performance. Charm also provides an extensive toolbox of useful Python

[2]A dedicated module to support lattice-based cryptography is in preparation for a future release.

routines including secret sharing, encryption padding, group parameter generation, message encoding, and ciphertext parsing. We are continuously adding routines to the toolbox, and future releases will include contributions from external developers.

Protocol Engine. Interactive protocols often seem simple on paper but in reality require a variety of different considerations. Zero-knowledge proofs are especially tricky as they often utilize information that is not specified in the documentation. General protocol implementations must include network communications, data serialization, error handling, and state machine transition. Charm simplifies development by providing all of these features as part of a reusable *protocol engine*. An implementation in our framework consists of a list of parties, a description of states and transitions, and the core logic for each state. Serialization, transmission and error handling are handled at the lower levels and are available freely to the developer.

Our protocol engine provides native support for the execution of sub-protocols and supports recursion. We have found subprotocols to be particularly useful in constructions that use interactive proofs of knowledge.

Given a protocol implementation, an application executes it by selecting a party type and optional initial state, and by providing a collection of socket connections to the remote parties. Sockets in Python are an abstract interface and can be extended to support various communication mechanisms.

ZKP Compiler. Zero-knowledge proofs of knowledge allow a Prover to demonstrate knowledge of a secret without revealing it to a Verifier. Such proofs are common in privacy-

preserving protocols such as the idemix anonymous credential system and Direct Anonymous Attestation [102, 103]. These proofs may be interactive or non-interactive (via the Fiat-Shamir heuristic, or using new bilinear-map based techniques [27, 104]). Regardless of the underlying mechanism, it has become common in the literature to describe such proofs using the notation of Camenisch and Stadler [105]. For instance,

$$\mathsf{ZKPoK}\{(x, y) : h = g^x \ \wedge \ j = g^y\}$$

denotes a proof of knowledge of two integers x, y that satisfy both $h = g^x$ and $j = g^y$. All values not enclosed in parentheses are assumed to be known to the verifier.

Converting these statements into working protocols is challenging, even for expert developers. To assist implementation, Charm borrows from the techniques of ZKPDL and CACE [80, 81], providing native support for honest verifier Schnorr-type proofs via an automated protocol compiler.

Benchmarking System. Performance is often critical when designing and implementing real-world cryptosystems. Therefore developers are frequently interested in the efficiency of their schemes, both from a timing and computational perspective. They also might wonder how changes they make can affect these important aspects and how their schemes compare to others. In order to help developers measure the performance of a prototype implementation, Charm incorporates a native benchmark module to collect information on a scheme's performance. This module collects and aggregates statistics on a set of operations defined by the user. All of the operations in the core modules are instrumented separately, allowing for detailed profiling including total operation counts, average operation time for

various critical operations, and network bandwidth (for interactive protocols). Users can define their own measurements within a given implementation (*e.g.,* a scheme or subroutine). When these measurements involve timing, the benchmarking module automatically performs and collects timing information. Many of our experiments in Section 4.6 were performed using the benchmarking system. The benchmarking system is easy to switch on or off and has minimal impact on the system when it is not in use. An example of using the benchmarking system is provided in Section 5.5.

4.5 Implementation

In this section, we describe our implementation and provide further details on components of our architecture. In Section 4.5.1 below, we reference an example comparing a protocol description from the literature to one implemented in our system. The code fragment shown in Figure 4.5 is a good overall example of using Charm and is worth studying at this point to understand our approach.

Language Features. Python provides many useful features that simplify development for programmers using Charm. Benefits include support for object-oriented programming, dynamic typing, overloading of mathematical operators, automatic memory allocation and garbage collection.

The language also provides useful built-in data structures such as tuples and dictionaries (essentially, key-value stores) useful for common tasks such as storing ciphertexts and

CS98 Encryption

Encryption. Given a message $m \in G$, the encryption algorithm runs as follows. First it chooses $r \in Z_q$ at random. Then it computes

$$u_1 = g_1{}^r, u_2 = g_2{}^r, e = h^r m, \alpha = H(u_1, u_2, e), v = c^r d^{r\alpha}$$

The ciphertext is (u_1, u_2, e, v)

```
def encrypt(self, pk, M):
    r     = group.random(ZR)
    u1    = (pk['g1'] ** r)
    u2    = (pk['g2'] ** r)
    e     = group.encode(M) * (pk['h'] ** r)
    alpha = group.hash((u1, u2, e))
    v     = (pk['c'] ** r) * (pk['d'] ** (r*alpha))

    return { 'u1' : u1, 'u2' : u2, 'e' : e, 'v' : v }
```

CS98 Decryption

Decryption. Given a ciphertext (u_1, u_2, e, v), the decryption algorithm runs as follows. It first computes $\alpha = H(u_1, u_2, e)$, and tests if

$$u_1{}^{x_1 + y_1 \alpha} u_2{}^{x_2 + y_2 \alpha} = v$$

If this condition does not hold, the decryption algorithm outputs ``reject"; otherwise, it outputs $m = e/u_1{}^z$

```
def decrypt(self, pk, sk, c):
    alpha = group.hash((c['u1'], c['u2'], c['e']))
    v_pr  = (c['u1'] ** (sk['x1']+(sk['y1']*alpha)))*
            (c['u2'] ** (sk['x2']+(sk['y2']*alpha)))
    if (c['v'] != v_pr):
        return False
    return group.decode(c['e'] / (c['u1'] ** sk['z']))
```

Figure 4.5: Encryption and Decryption in the Cramer-Shoup scheme [106]. The top box shows the description of the algorithm in the published paper while the bottom box reflects the Charm code. Charm is designed to enable cryptographers to implement their schemes using mathematical notation that mirrors the paper description.

public keys. These values can be automatically serialized and deserialized, eliminating the need for custom parsing code. To read legacy files with a specific binary format we use the python `struct` module, which performs packing and unpacking of binary data. Our decision to use Python is supported by the fact that much of the effort in a typical C implementation relates to laboriously defining and serializing data structures.

Python also supports dynamic generation of code. This feature is particularly useful in constructing a Zero-Knowledge proof compiler (see Section 4.5.3). The features discussed here are not unique to Python and can be found in other high-level languages.[3] However Python has a large and devoted user base and provides a good balance between usability, stability, and performance.[4]

Low-level Python/C Modules. As discussed in Section 4.4, for performance reasons, our

[3] Nor are we the first to import cryptographic operations into Python. See for example [86, 107].

[4] It is also well supported. Our experiments show that there have been significant performance improvements between Python 2.x and 3.x. Charm supports both versions for backwards compatibility with legacy applications.

implementation of Charm supports a variety of C math libraries including GMP [100], OpenSSL [77], RELIC [89], MIRACL [87] and the PBC library [78]. We provide Python/C extensions for these libraries.

Our base modules expose arithmetic operations using standard mathematical operators such as $*$, + and $**$ (exponentiation).[5] Besides group operations, our base modules also perform essential functions such as element serialization and encoding.

In addition to the base modules, we provide a cryptobase module that includes fast routines for bitstring manipulation, evaluation of block ciphers, MACs, and hash functions. Supported ciphers include AES, DES, and 3DES. Moreover, this module implements several standard modes of operation such as CBC and CTR (drawn from PyCrypto [101] and libTomCrypt [108]) that facilitate encryption of arbitrary amounts of data.

Benchmark Module. As described in Section 4.4, we provide a benchmark module for measuring computation time and counting operations, such as exponentiations and multiplications, in a given snippet of code at runtime. Our benchmark module provides a consistent interface that developers can use to perform these measurements. Each base module inherits the *benchmark* interface and is incorporated into a cryptographic scheme as follows:

```
assert InitBenchmark(), "failed to initialize benchmark"
# select benchmark options
StartBenchmark(["RealTime", "Exp", "Mul", "Add", "Sub"])
... code ...
EndBenchmark()
```

[5] For consistency, group operations are always specified in multiplicative notation, thus $*$ is used for EC point addition and $**$ for point multiplication. This makes it easy to switch between group settings.

```
# obtain results
msmtDict = GetGeneralBenchmarks()
print(msmtDict["Exp"])
```

As stated earlier, benchmarking can be easily removed or disabled after measurements are complete and introduces negligible overhead.

Algebraic Groups and Fields. While our base modules provide low-level numerical functions, there are still differences in how each module handles serializing elements, encoding messages, and generating group parameters. For instance, for the ecmath module we employ subgroups of elliptic curves over a finite field, whereas the integermath module implements integer groups, rings, and fields. To reconcile these differences, we provide a thin Python interface to encapsulate differences in group/field parameter generation, serialization, message encoding, and hashing. This interface allows us to standardize calls to the underlying base modules from a developer's perspective.

With this approach, cryptographers are able to adjust the algebraic setting (standard EC, integer or pairing groups) on the fly without having to re-implement the scheme. For instance, our implementations of DSA [45], ElGamal [109] and Cramer-Shoup [106] can be instantiated in any group with an appropriate structure.

4.5.1 Schemes

To demonstrate the potential of our framework, we implemented a number of standard and experimental cryptosystems. We provide a collection of implemented schemes that

Scheme	Type	Setting	Comp. Model	Lines
Encryption				
RSA-OAEP [110]	Public-Key	Integer	ROM	22
CS98 [106]	Public-Key	EC/Integer	Standard	40
ElGamal [111]	Public-Key	EC/Integer	Standard	34
Paillier99 [112]	Public-Key	Integer	Standard	31
BF01 [28]	Identity-Based	Pairing	ROM	51
BB04 [66]	Identity-Based	Pairing	Standard	45
Waters05 [67]	Identity-Based	Pairing	Standard	49
CKRS09 [68]	Identity-Based	Pairing	Standard	55
LSW08 [69]	Identity-Based	Pairing	ROM*	69
SW05 [32]	Fuzzy Identity-Based	Pairing	Standard	68
BSW07 [29]	Attribute-Based	Pairing	ROM*	62
Waters08 [30]	Attribute-Based	Pairing	ROM*	61
LW10 [31]	MA Attribute-Based	Pairing	ROM*	67
FE12 [113]	DFA-based Functional	Pairing	Standard	71
HVE08 [114]	Hidden Vector	Pairing	Standard	104
Digital Signatures				
Schnorr [115]	Regular	Integer	ROM	33
RSA-PSS [99]	Regular	Integer	ROM	32
EC-DSA/DSA [45]	Regular	EC/Integer	*n/a*	32
HW09 [116]	Regular	Integer	Standard	113
CHP [117]	Regular	Pairing	Standard	30
CL03 [16]	Regular	Integer	Standard	58
CL04 [73]	Regular	Pairing	ROM	25
HW [116]	Regular	Pairing	Standard	48
Hess [118]	Identity-Based	Pairing	ROM	31
CHCH [119]	Identity-Based	Pairing	ROM	31
Waters05 [67]	Identity-Based	Pairing	Standard	43
Boyen [70]	Ring-based	Pairing	CRS	65
CYH [71]	Ring-based	Pairing	ROM	58
BLS03 [120]	Regular/Short Signature	Pairing	ROM	23
BBS04 [59]	Group-based	Pairing	ROM	60

Table 4.1: A partial listing of the cryptographic schemes we implemented. "Code Lines" indicates the number of lines of Python code used to implement the scheme (excluding comments and whitespace), and does not include the framework itself. ROM indicates that a scheme is secure in the Random Oracle Model. CRS indicates that a scheme is secure in the Common Reference String Model. A "-" indicates a generic transform (adapter). * indicates a choice made for efficiency reasons. See the rest of the listing in Appendix A.1.

includes a variety of encryption schemes, signatures, commitments, and interactive protocols.[6] Most of the implementations consist of fewer than 100 lines of code (see Table 4.1 for a listing).

We provide several examples to illustrate code in Charm. Figure 4.5 shows the encryption and decryption algorithms for the Cramer-Shoup [106] scheme, and the corresponding Charm code. We provide the remaining algorithms, along with some additional examples, in Appendix A.1. We note that our framework was designed to minimize the differences between published algorithms and code (as shown in Figure 4.5), in the hope of lowering the barriers to implementation.

4.5.2 Protocol Engine

Every protocol implementation in Charm is a subclass of the `Protocol` base class. This interface provides all of the core protocol functionality, including functions to support protocol implementations, a database for maintaining state, serialization, network I/O, and a state machine for driving the protocol progression.

Creating a new interactive protocol is straightforward. The implementation must provide a description of the parties, protocol states and transitions (including error transitions for caught exceptions), as well as the core functionality for each state. State functions accept and return Python dictionaries containing the passed parameters. Socket I/O and data serialization is handled transparently before and after each state function runs. Developers

[6] For more scheme implementations, see `http://jhuisi.github.com/charm/schemes.html`.

have the option to implement their own serialization functionality for protocols with a custom message format. Public parameters may either be passed into the protocol or defined in the `init` function. Finally, we provide templates for some common protocol types (such as Σ-protocols). Figure 4.6 contains an example of a machine-generated `Protocol` subclass.

Executing protocols and subprotocols. Executing a protocol consists of two calls to the `Protocol` interface. First, the application calls `Setup()` to configure the protocol with an identifier of one of the parties in the protocol, optional initial state, public parameters, a list of remote parties, and a collection of open sockets. It then calls `Execute()` to initiate communication.

We also provide support for the execution of *subprotocols*. Launching a subprotocol is simpler than an initial execution, since the protocol engine already has information on the remote parties. The caller simply identifies for the server the role played by each of the parties in the subprotocol (*e.g.,* the Server party may be remapped to be the Prover for the subprotocol), and instructs the protocol engine to run the subprotocol via the `Execute()` method.

Our engine currently supports only synchronous operation. Asynchronous protocol runs must be handled by the application itself using Python's threading capabilities. Callback functions may be supplied by passing function references as part of the public parameters. We plan to provide more complete support for asynchronous execution in future releases.

4.5.3 ZKP Compiler

Many advanced cryptographic protocols (*e.g.,* [59, 121, 122]) employ zero-knowledge or witness-indistinguishable proofs as part of their protocol structure. The notation of Camenisch and Stadler [105] has become the de facto standard in the cryptography literature. This notation, while elegant, stands in for a complex interactive or non-interactive subprotocol that must be derived before the base protocol can be implemented.

To handle such complex protocols, Charm includes an automated compiler for common ZK proof statements. Such compilers have been implemented in the past by Meiklejohn *et al.* (ZKPDL) [80] and Bangerter *et al.* (CACE) [123]. Our compiler interprets Camenisch-Stadler style proof descriptions at runtime and derives an executable honest-verifier protocol. At present our compiler handles a limited set of discrete-log statements, and is not currently as rich as ZKPDL or CACE. However, it offers some advantages over those systems.

First, as Python is an interpreted language, we do not require a custom interpreter for the compiled proofs, as ZKPDL does. Instead, we exploit Python's ability to dynamically generate and execute code at runtime. We employ this feature to convert Camenisch-Stadler proof statements into Charm code, which we feed directly to the interpreter and protocol engine.[7] Second, since our compiler has access to the public and secret[8] variables at compile time, Charm can use introspection to determine the variable types, settings and parameter

[7]In practice, we first compile to bytecode, then execute. This reduces overhead for proofs that will be conducted multiple times.

[8]Clearly the verifier does *not* have access to the secret variables. We address this later in this section.

sizes. This information forms the bulk of what is provided in a ZKPDL or CACE Protocol Specification Language (PSL) program. Thus, from a developer's perspective, executing a ZK proof is nearly as simple as writing a Camenisch-Stadler statement.

Our compiler, implemented in Python itself, outputs Python code. The interface to the compiler closely resembles a Camenisch-Stadler proof statement. The caller provides two Python dictionaries containing the public and secret parameters, as well as a string describing the proof goal. In some cases, such as when configuring the Verifier portion of an interactive proof, the secret values are not available. We currently deal with this by providing "dummy" variables of the appropriate type. Our runtime compiler can examine the variables and automatically generate appropriate code on the fly. The compiler produces one of two possible outputs: a routine for computing a non-interactive protocol via the Fiat-Shamir heuristic, or a subclass of `Protocol` describing the Prover and Verifier interactions, in the case of interactive protocols.

In the interactive case, we provide support routines to generate the class definition, compile the generated code into Python bytecode, initialize communication with sockets provided by the caller, and execute the proof of knowledge. The code below illustrates a typical interactive proof execution from the Prover:

```
# prover
public = {'h':g ** x, 'g':g, 'j':g ** y}
secret = {'x':x, 'y':y}
result = executeIntZKProof(public, secret,
       "(h = g^x) and (j = g^y)", party_info)
```

Figure 4.6 shows a generated `Protocol` subclass for the proof goal $h = g^x$.

The runtime technique is useful for developers who require compact, readable code. However, we note that since our protocol produces Python code, it can also be used to compile static protocol code which may be added to a project.

At present our compiler is intended as a proof of concept because it lacks support for many types of statements (e.g. Boolean-OR) and proof settings. Our compiler is less sophisticated than CACE and ZKPDL. For example, in addition to supporting more complex conjunctions and statement types, CACE includes formal verification of proofs. We believe that our approach is complementary to these projects, and we hope to establish collaborations to extend Charm's capabilities in future versions.

```
class ZKProof(Protocol):
    def __init__(self, groupObj, common_input=None):
        Protocol.__init__(self)
        # ... init of party, states and transitions ...
        # ... setup group object ...
        # ... init of base class db ...

    def prover_state1(self):
        pk = Protocol.get(self, ['h','j','g'], dict)
        (x,) = Protocol.get(self, ['x'])
        k0 = self.group.random(ZR)
        val_k0 = pk['g'] ** k0
        Protocol.store(self, ('k0',k0),('x',x))
        Protocol.setState(self, 3)
        return {'val_k0':val_k0, 'pk':pk }

    def verifier_state2(self, input):
        c = self.group.random(ZR)
        Protocol.store(self, ('c',c),
                   ('pk',input['pk']),
                   ('val_k0', input['val_k0']) )
        Protocol.setState(self, 4)
        return {'c':c}                                 ...
```

```
...
  def prover_state3(self, input):
      c = input['c']
      val = Protocol.get(self, ['x','k0',], dict)
      z0 = val['x'] * c + val['k0']
      Protocol.setState(self, 5)
      return {'z0':z0,}

  def verifier_state4(self, input):
      z0 = input['z0'];
      val = Protocol.get(self, ['pk','val_k0','c'], dict)
      if (val['pk']['g'] ** z0) ==
         ((val['pk']['h'] ** val['c']) * val['val_k0'] ):
          result = 'OK'
      else:
          result = 'FAIL'
      Protocol.setState(self, 6)
      Protocol.setErrorCode(self, result)
      return result
```

Figure 4.6: A partial listing of the generated protocol produced by our Zero-Knowledge compiler for the honest-verifier proof $\mathsf{ZKPoK}\{(x) : h = g^x\}$.

4.5.4 Meta-information and Adapters

Charm provides the ability to label schemes so that they carry meta-information about their input/output space and security definitions. Wherever possible this information is

derived automatically, *e.g.,* from the scheme type or function definitions. Optionally, developers can provide other details such as the complexity assumption and computational models used in the scheme's security proof via a standard annotation interface. This information allows developers to compare and check compatibility between schemes.

All schemes descend from the `Scheme` class, which provides tools to record and evaluate meta-information. Developers use the `setProperty()` method to specify important properties. For example, the `init` function of an Identity-Based Encryption scheme might include a call of this form:

```
# Set the scheme's security definition,
# ID space, and message space.
setProperty(self, secdef=IND_ID_CPA,
     id=str, messageSpace=str)
```

Schemes with more restrictive parameters, *e.g.,* group elements and/or strings of limited length, can specify these requirements as well.[9] Once each scheme is labeled with the appropriate metadata, we can programmatically extract this information at run-time to verify a given set of criteria.

Adapter example. To illustrate how this functionality works in practice, we consider the process of constructing *adapters* between different schemes. In Section 4.4 we proposed an adapter chain to convert the Boneh-Boyen IND-sID-CPA-secure signature scheme [66] into an EU-CMA signature (see Figure 4.3). This transformation requires two adapters: one to convert the selectively-secure IBE scheme into an adaptively-secure IBE scheme (in the

[9] In some cases, evaluation of a scheme depends on the scheme's public key.

random oracle model), and another to transform the resulting IBE into a signature using the technique of Naor [95].

The Hash Identity adapter has an explicit and implicit function. Explicitly, it applies a hash function to the Boneh-Boyen IBE, which accepts identities in the group $\mathbb{Z}_r$,[10] thus altering the identity-space to $\{0, 1\}^*$. Implicitly, it converts the security definition of the resulting IBE scheme from IND-sID-CPA to the stronger IND-ID-CPA definition and updates the meta-information to note that the security analysis is in the random oracle model.[11] The adapter itself is implemented as a subclass of `IBEnc` (see Figure A.2a in Appendix A). It accepts the Boneh-Boyen IBE (also an `IBEnc` class) as input to its constructor. At construction time, the adapter must verify the properties of the given scheme using the `checkProperty()` call. It then advertises its own identity space and security information. This code is contained within the adapter's `init` routine and appears as follows:

```
...
if IBEnc.checkProperty(self, scheme,
    [('scheme','IBEnc'),('secDef',IND_sID_CPA),
      ('id',ZR)]):
   self.ibe = scheme
   IBEnc.updateProperty(self, scheme,
         secDef=IND_ID_CPA, id=str,
         secModel=ROM)
...
```

The IBE-to-Sig adapter converts any adaptively-secure IBE scheme into an EU-CMA signature.[12] This adapter is implemented as a subclass of `PKSig`. It accepts an object

[10] The value r is typically a large prime.

[11] On a call to `encrypt` or `keygen` the adapter simply hashes an arbitrary string into an element of $\mathbb{Z}_r$, then passes the result to the underlying IBE scheme. This technique and its security implications are described in [66].

[12] Naor [95] observed that adaptively-secure IBE can be converted into a signature scheme by using the

derived from IBEnc and verifies that it advertises at least IND-ID-CPA security (IND-sID-CPA is not sufficient, hence our use of the previous adapter) and possesses an appropriate message space. With this check satisfied, this adapter inherits the security model of the underlying IBE, adopts the IBE's identity space as the message space for the signature, and advertises the EU-CMA security definition.

In future versions of the library, we hope to significantly extend the usefulness of this meta-data, and to include detailed information on performance (gathered through automatic testing). We also intend to provide tools for *automatically* constructing useful adapter chains based on specific requirements.

4.5.5 Type checking and conversion

Python programs are dynamically typed. In general, we believe that this is a benefit for a rapid prototyping system: dynamic typing makes it possible to assemble and modify complex data structures (*e.g.,* ciphertexts) "on the fly" without the need for detailed structure definitions.

Of course, the lack of static typing has disadvantages. For example, type errors may not be detected until runtime. Furthermore, it can limit the utility of adapters that depend on having *a priori* knowledge about a scheme's input or output characteristics.

To address these issues, Charm provides optional support for static typing using the Python annotation interface. When it is provided, Charm uses this type information to val-

IBE key extraction algorithm for signing.

idate the inputs provided to a cryptographic algorithm and, in cases where the inputs are of the wrong type, to automatically convert them. For the latter purpose, Charm provides a standard library designed to encode values to and from a variety of standard types, including bit strings and various types of group elements. An example of the Charm typing syntax is provided below:

```
pk_t = {'g1':G, 'g2':G, 'c':G, 'd':G, 'h':G}
c_t = {'u1':G, 'u2':G, 'e':G, 'v':G}

@Input(pk_t, str)
@Output(c_t)
def encrypt(self, pk, M):
     ...
```

We believe that support for explicit typing also provides a foundation for adding formal verification techniques to Charm, though we leave such verification to future work.

4.5.6 Using Charm in C applications

To enable the use of Charm schemes in existing C applications, we provide an embed API for integrating Charm schemes without burdening developers. Our approach achieves two important goals. First, the embed API is easy-to-use, intuitive, and straightforward for developers to use a scheme based on its scheme type API (*e.g.,* keygen, encrypt/decrypt). Second, the API allows C applications to interchange primitives of the same type with minimal modifications.

To embed a scheme, the application first calls the InitializeCharm() function to setup the Charm environment. Once Charm is setup, the application creates a group object for

instantiating a scheme. This is accomplished by calling the group initialization function for a given setting such as InitPairingGroup(), InitIntegerGroup(), etc. Next, the application calls InitScheme() and includes the scheme file name, class name, and the group object handle returned from the previous call. To call any function within the scheme, the application uses the CallMethod() and supplies the arguments for the target function. Finally, we provide serialization methods (objectToBytes() and bytesToObject()) for converting Charm objects to/from base-64 encoded binary strings. We believe our simple embed API enables Charm to be seamlessly integrated into a variety of applications that require advanced cryptographic constructs. For a detailed example, see Figure A.1 in Appendix A.

4.6 Performance

Charm is primarily intended for rapid prototyping, with an emphasis on compactness of source code and similarity between standard protocol notation and code. These properties all favor the developer and are qualities designed to facilitate more semantically correct, robust, and secure code. However, we recognize that achieving these properties is likely to come at a tradeoff in performance.

As such, in this section we report representative performance metrics collected through the use of Charm's built-in benchmarking system. These metrics are quantitatively compared against detailed timing experiments of two existing C cryptographic system implementations. We observe that the performance cost of using Charm is variable, and it is

directly dependent on the nature of the scheme implementation.

4.6.1 Comparison with C Implementations

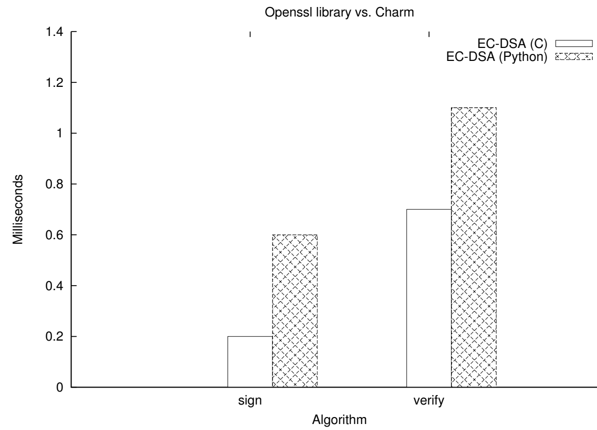
We conducted detailed timing experiments on two of the cryptosystems we implemented: EC-DSA [45] and a CP-ABE scheme due to Bethencourt, Sahai, and Waters [29]. We chose these two because of their available C implementations, thus realistic choices against which to compare. Our experiments comprise two different points on a spectrum: our EC-DSA experiment considers Charm's performance in an algorithm with very fast operation times, and our CP-ABE experiment considered a scheme with a high computational burden (to stress this, we instantiated the scheme with a 50-element policy).

Experimental setup. We used the benchmark module to collect timings for our Charm implementation of the EC-DSA Sign and Verify algorithms. This provided us with total operation time for both algorithms. We then collected total operation times for OpenSSL's implementation of the same algorithms using the built-in `speed` command.

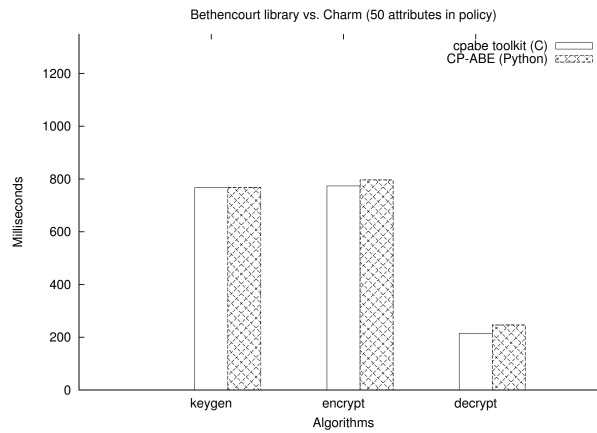
For CP-ABE we used benchmark again to collect measurements for our ABE key generation, encryption and decryption implementations (omitting the setup routine). For key generation, we extracted a key containing 50 attributes $(1, \ldots, 50)$. We next encrypted a random message (in the group $\mathbb{G}_T$) under a policy consisting solely of AND gates: (1 AND 2 AND ... AND 50). Finally, we decrypted the message using the extracted key. For each experiment, we measured total time and repeated these experiments using John Bethencourt's library (available from [124]) to obtain the C time.

We conducted our experiments on a Macbook Pro with a 2.4Ghz Intel i5 with 8GB of RAM running Mac OS 10.7 and Python v3.2.3. All of our experiments were performed on a single core of the processor. For all experiments (Charm and C), we used either OpenSSL v1.0.1c library or libpbc 0.5.12 to perform the underlying mathematical operations. Our EC-DSA experiments used the standard NIST P-192 elliptic curve. For CP-ABE, we used a 512-bit supersingular curve (with embedding degree $k = 2$) from libpbc. All of our timing results are the average of 10 experimental runs.

Experimental results. The results of our experiments are presented in Figure 4.7. Unsurprisingly, our Charm implementation of EC-DSA suffered a substantial performance penalty when compared to the OpenSSL version. This is unavoidable given the relatively low overall time required for EC-DSA operations—even small interpretation inefficiencies add up to a large percentage of the total cost. Our results with CP-ABE (and 50 attributes) are encouraging. For the CP-ABE algorithms, Charm is competitive with the C implementation. As a result, we believe Charm can be a primary tool for cryptographers wishing to approximate the performance of their schemes or protocols in practice [125]. For additional performance measurements, see our technical report [88].



(a) Comparison to OpenSSL



(b) Comparison to Beth-cpabe toolkit

Figure 4.7: For EC-DSA, we select the NIST P-192 elliptic curve and for CP-ABE [29], we measure 50 attributes for keygen and 50 leaves in the policy tree for encrypt and decrypt.

4.7 Related Work

Our work builds upon previous efforts to provide software libraries for developers who use cryptography. We describe four different types of libraries below.

Cryptographic (primitive) libraries. The first widely available general purpose library for commonly used cryptographic functions was Jack Lacey's CryptoLib [126]. Following CryptoLib, many other packages were developed, including Peter Guttman's similarly named CryptLib[13], RSA's Bsafe Crypto-C[14], and more recently JAVA libraries such as Cryptix[15], BouncyCastle[16]. While these libraries have been useful for application developers, they were designed for specific and mostly isolated purposes. Moreover, they only implement commonly used and standardized cryptographic functions.

There have not been as many implementations of cryptosystems such as IBE, ABE, and related advanced primitives. Of note is the implementation by Bethencourt, Sahai and Waters [29], which provides an API for ciphertext policy ABE. This package is part of the Advanced Crypto Software collection (ACSC) [124], which in addition to this ABE library, includes separate packages for other advanced application-based primitives such as forward-secure signatures and broadcast encryption. Our Charm architecture provides a comprehensive and unified framework that is both usable and developer friendly for rapid prototyping of advanced primitives.

Math libraries. The GNU Multiple Precision Arithmetic Library (GMP) [100] is a free, high-precision mathematics library, specifically optimized for speed of cryptographic algorithms. The Stanford Pairing-Based Cryptography (PBC) library [78] is free, written in C, and built on top of GMP to provide abstractions for developing pairing algorithms. PBC

[13] http://www.cs.auckland.ac.nz/~pgut001/cryptlib/
[14] http://www.rsa.com/rsalabs/node.asp?id=2301
[15] http://www.cryptix.org/
[16] http://www.bouncycastle.org/

was built for expressiveness, but not designed for usability or performance. RELIC [89], also an open source library which relies on GMP, was built for speed and portability with support for big number arithmetic, traditional elliptic curves and pairings. While RELIC is highly configurable and supports a variety of cryptographic optimizations, it was not primarily built for usability.

The Multiprecision Integer and Rational Arithmetic Library (MIRACL) [87] is written in C/C++ and provides APIs for big number arithmetic, elliptic curve cryptography, block ciphers and hash functions. Similar to RELIC, MIRACL is a highly optimized library that is compatible with a variety of architectures and is quite expressive in terms of functionality. However, MIRACL places a secondary focus on usability. Using the library effectively requires knowledge of its inner workings. Our Charm framework shields developers from dealing with these libraries directly via layers of abstractions. Instead, cryptographers can utilize our abstractions to implement their schemes or protocols using standard notation and evaluate them against any of the math libraries supported in Charm.

Cryptographic compilers and frameworks. Ben Laurie's *Stupid* programming language [84] compiles into C and Haskell and is intended for constructs like ciphers and hash functions. Cryptol [85] compiles to a VHDL circuit for use with an FPGA. More recently, Dan Bernstein's *NaCl* (or “salt") [127] software library in C/C++ provides an easy-to-use interface (*e.g.,* encryption, decryption, signatures, etc.) to build higher-level cryptographic tools.

Protocol and Secure Function Evaluation compilers. The authors of the Zero Knowledge Proof Descriptive Language (ZKPDL) [80] offer a language and an interpreter for

implementing privacy-preserving protocols. Their example application is electronic cash, but their descriptive language is more general. A similar approach is provided by FairPlay [82], which provides a language-based system for secure multi-party computations. The authors of FairPlay provide a Secure Function Definition Language (SFDL), which can be used by programmers to specify code for multi-party computations. Charm takes a similar approach but with a focus on providing a simple language in the Camenisch and Stadler [105] notation for specifying high-level proof statements. From this proof statement, our compiler automatically generates the interactive protocol details.

A software package called Tool for Automating Secure Two-Party Computations (TASTY) [83] allows protocol designers to specify a high-level description of a computation that is to be performed on encrypted data. TASTY then generates protocols based on the specification, and compares the efficiency of different protocols. Similarly, the Computer Aided Cryptography Engineering (CACE) project has also developed a system that specifies a language for zero knowledge proofs [123, 128]. In this system, a compiler translates zero-knowledge protocol specifications into Java code or LaTeX statements. The CACE ZK compiler has many features, optimizations, and performance benefits. Our framework is certainly compatible with the CACE design and we intend on leveraging CACE as a building block in Charm.

4.8 Charm-Crypto Toolkit

The Charm framework is freely available at `http://charm-crypto.com/Download.html` with extensive documentation[17] for how to use it. To make Charm easy-to-use, we provide automated installers for various platforms such as Windows, Mac OS X and Linux. Additionally, to support embedded environments, we have ported the framework to mobile platforms such as Android. Our end goal is to enable Charm on as many platforms as possible.

4.9 Challenges and Open Problems

To provide extensibility and modularity in Charm, we require some building blocks to meet such challenges. For example, at the lowest level, we provide abstract C/C++ interfaces around the C math libraries to make them interchangeable at build time. This allows cryptographers to evaluate their scheme implementations against different libraries by only changing the Charm install configuration. With the pairingmath module, for instance, we can evaluate the performance of schemes against the PBC, MIRACL, and RELIC libraries without changing the scheme itself. It is relatively easy to extend our framework with new math libraries that adhere to our C/C++ abstract interface. Moreover, we are able to extend our platform to diverse environments with relatively low effort and without affecting the higher level components in Charm. Thus, all of these features enable Charm to pro-

[17] `http://charm-crypto.com/Documentation.html`

vide a test bed for rapidly prototyping and evaluating advanced cryptosystems against any appropriate underlying C library.

While the Charm architecture addresses a number of issues to facilitate rapid implementations of modern cryptography, it did not come without technical challenges. Our first challenge was determining the interface that should be exposed in Python for building schemes and protocols in a way that is standard and comprehensive. The second challenge was conforming the math libraries to this interface. This was not a significant issue for well established math libraries such as GMP, OpenSSL, PBC, and MIRACL. However, for more recent research libraries such as RELIC, this presented challenges due to missing functionality (*e.g.*, serialization) and the alpha software quality of the pairings interface. But given the optimizations available in RELIC for pairings, it has the potential to become the standard for pairing-based cryptography in the near future.

An open area is to develop automated compilers for performing various operations on cryptographic schemes. One such example is the translation of schemes between various settings, *e.g.,* composite-order to prime-order bilinear groups. Both David Freeman [129] and Alison Lewko [130] have recently proposed tools for this type of translation; however, all of these tools currently require human intervention. We believe that Charm provides an excellent platform for implementing techniques that *automatically* translate such schemes (represented in a domain-specific language) to working implementations.

On the engineering side, there are a number of issues related to improving Charm for applications that require extremely high performance. For example, the current Python

threading model is not ideal for applications that would benefit from substantial parallel processing (*e.g.,* lattice-based fully-homomorphic encryption schemes [131]). One of our major open problems is to find ways to take full advantage of multi-core systems. Finally, we understand that there may be instances where development requirements cannot support a high-level interpreted language such as Python. To address this we plan to examine the possibility of compiling Charm code directly to languages such as Haskell and C, using tools such as Shedskin [132].

Chapter 5

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

In the previous chapter, we introduced the Charm framework and how it fits into our architecture for automation. In this chapter, we will describe a tool that finds efficient batch verification algorithms in an automated fashion. We will explore how our approach can produce competitive results to previous manual approaches in a secure manner. Additionally, we will analyze the security of our batching techniques and show how it preserves the security of its inputs.

5.1 Overview

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

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

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

In this work, we expand upon an extended abstract on AutoBatch appearing in ACM CCS 2012 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.

5.2 Introduction

We anticipate a future where computers are everywhere as an integrated part of our surroundings, continuously exchanging messages, e.g., sensor networks, smartphones, vehicular communications. For these systems to work properly, messages must carry some form of authentication, and yet the system requirements on this authentication are particularly demanding. Applications such as vehicular communications [133, 134], 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 [120] 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 [51] — 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 4.7, 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 chapter demonstrates that it is feasible for humans to turn over some of the design, proof writing and implementation work in batch verification to machines.

5.2.1 Our Contributions

We present AutoBatch, an automated tool that transforms a high-level description of a signature scheme[1] into an optimized batch verification program in either Python or C++. AutoBatch takes as input a Scheme Description Language (SDL) representation of a signature scheme (see Section 3.5.1 for details on SDL) 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

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

of carefully designed heuristics. Our tool produces a modified SDL 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 the argument for why security holds.

AutoBatch was tested on several pairing-based schemes. It produced the first batching algorithms, to our knowledge, for the Camenisch-Lysyanskaya [18] and Hohenberger-Waters [135] signatures.[2] It also discovered a significantly faster algorithm for batching the proofs of the verifiable random functions (VRF) [19]. Moreover, AutoBatch is able to handle batches with more than one type of signature. Indeed, we found that the Hess [136] and Cha-Cheon [137] 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 chapter, and we feel could be of great value for applications, such as mail servers, which may encounter many signature types at once.

[2]It also produced a candidate batching scheme for the Waters dual-system [57] signatures, although this signature scheme does not have perfect correctness and therefore our automated proof techniques do not immediately apply to it. See Section 5.3.1 for more.

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

5.2.2 Overview of Our Approach

We present a detailed explanation of AutoBatch in §5.4. In this section and in Figure 5.1 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:

- (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 our technical report [138] for more details). Given such an implementation, the Parser obtains the signature verification equation and encodes it into SDL.
- A Batcher, which takes as input an SDL file describing a signature verification equation. In addition to the signature verification equation, 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 that it maintains the unforgeability of the original scheme.

- 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 [87] or RELIC [139] 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 [11], then she can save time. This program can be given to the Code Parser, which

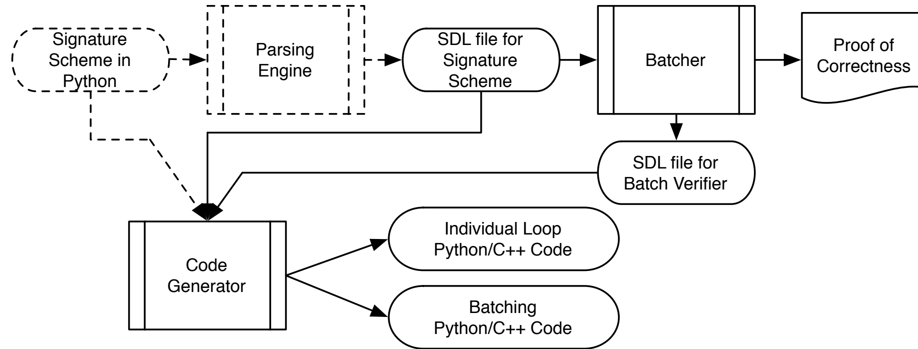


Figure 5.1: The flow of AutoBatch. The input is a signature scheme comprised of key generation, signing and verification algorithms, represented in the domain-specific SDL language. The scheme is processed by a Batcher, which applies the techniques and optimizations from Section 5.4 to produce a new 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.

will extract the necessary information from the code to generate a SDL file. 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.

5.2.3 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 [140] for their invention of EasyCrypt, an automated tool for generating security proof 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 [43] for a variant of RSA. In 1994, an interactive batch verifier for DSA presented in an early version of [141] was broken by Lim and Lee [142]. In 1995 Laih and Yen proposed a new method for batch verification of DSA and RSA signatures [143], but the RSA batch verifier was broken five years later by Boyd and Pavlovski [46]. In 1998, two batch verification techniques were presented for DSA and RSA [144, 145] but both were later broken [46–48]. The same year, Bellare, Garay and Rabin took the first systematic look at batch verification [52] and presented three generic methods for batching modular exponentiations, one of which is called the *small exponents test*. Unfortunately, in 2000, Boyd and Pavlovski [46] 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., pairings) were proposed [137, 146–148] but all were later broken by Cao, Lin and Xue [49]. In 2006, a method was given for identifying invalid signatures in RSA-type batches [149], but it was also flawed [50].

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 [150] 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 [46] provide numerous examples of this case.

AutoBatch addresses these common pitfalls. It uses one security definition (in Section 5.3) and provides a proof of correctness for every algorithm it outputs relative to this definition (in Section 5.4.3), 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 [151]. Boneh, Lynn and Shacham provided a single-signer batch verifier for BLS signatures [120]. Camenisch, Hohenberger and Pedersen [150] gave multiple-signer batch verifiers for Waters identity-based signatures [67] 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 [51]. Blazy, Fuchsbauer, Izabachéne, Jambert, Sibert and Vergnaud [152] 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 [54, 153, 154].

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 [1–3, 60, 61], optimizations to software implementations involving elliptic-curve cryptography [155] and bilinear-map functions [156], secure two-party computation [9, 10, 64], and zero-knowledge proofs [4–8, 62, 63].

5.3 Batch Verification for Signatures

Our security focus here is not directly on unforgeability [21]. 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.[3] *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.

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 [52] as extended in [150] to deal with multiple signers. We note that this definition is well specified

[3] We assume perfectly correct schemes here.

for perfectly correct schemes, but not for schemes that allow some correctness error. We discuss this further shortly.

Definition 5.3.1 (Batch Verification of Signatures)**.** *Let ℓ be the security parameter. Suppose* (Gen, Sign, Verify) *is a signature scheme with perfect correctness, $k, n \in \text{poly}(\ell)$, 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 [157].

5.3.1 On Schemes with a Correctness Error

The standard definition for signature batch verification (as presented in Definition 5.3.1)[4] 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 [57] using the technique described by Naor [28]. 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 decrypt algorithm using the signature as the private key, and testing to see that decryption successfully recovers m'. Since the Dual-System IBE [57] 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 5.3.1 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

[4] We added the restriction to perfect correctness in Definition 5.3.1. It was assumed in prior works but not always made explicit.

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 [57], because its complication made it a challenging test case. We report on the candidate batching algorithm we found in Section 5.5, although we note there and in Appendix A.7 that our automated proofs were only written to handle schemes with perfect correctness. This is a correction over the conference version of this work which did not make this distinction.

5.3.2 Algebraic Setting

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 [46] demonstrated efficient attacks on batching algorithms for DSA signature verification which omitted a subgroup membership test.

In this chapter, 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 $y^q \mod q = 1$ [150]. 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. [51] and confirmed by our tests.

5.3.3 Batch Verification in Bilinear Groups

Let us recall [51] 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: $\prod_{i=1}^{k} e(f_i, h_i)^{c_i} \stackrel{?}{=} A$, 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 5.3.2 (Bilinear-based Batch Verifier)**.**

Let $\mathsf{BMsetup}(1^\tau) \rightarrow (q, g_1, g_2, \mathbb{G}_a, \mathbb{G}_b, \mathbb{G}_T, e)$. *For each* $j \in [1, \eta]$, *where* $\eta \in \text{poly}(\tau)$, *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.*

5.3.4 Small Exponents Test Applied to Bilinear Groups

Bellare, Garay and Rabin [52] 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 [51] 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 5.3.3 (Small Exponents Test Applied to Bilinear Groups [51])**.** *Let* $\mathsf{BMsetup}(1^\tau) \rightarrow (q, g_1, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ *where q is prime. For each $j \in [1, \eta]$, where $\eta \in \text{poly}(\tau)$, let $X^{(j)}$ corresponds to a generic claim as in Definition 5.3.2. 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 5.3.3 as proven by Ferrara et al. [51].

5.4 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 5.1. 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 required by AutoBatch and provide several examples in Appendix A.2. For developers who already have an existing Charm/Python implementation, we also provide a Parsing Engine that can optionally *derive* an SDL representation

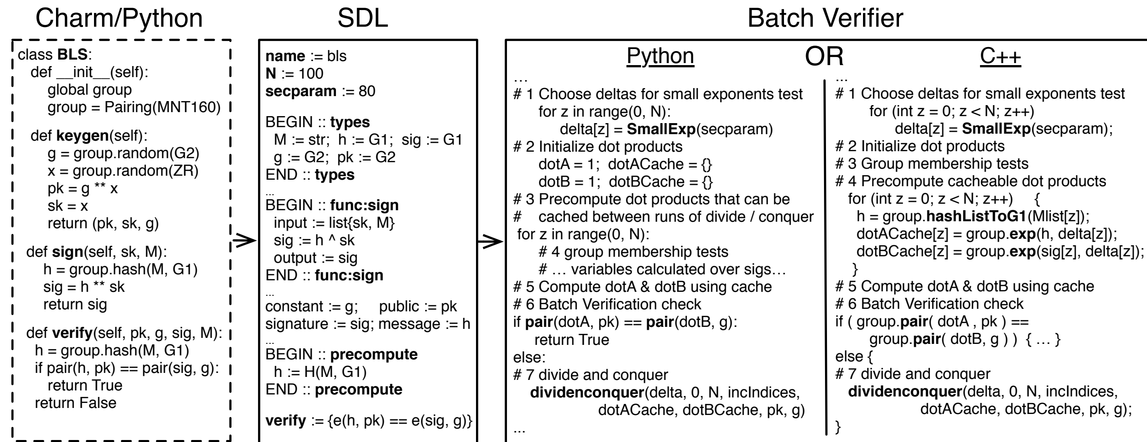


Figure 5.2: The Boneh-Lynn-Shacham (BLS) signature scheme [120] at various stages in the AutoBatch toolchain. At the left, an initial Charm-Python implementation of the scheme. In the center, an SDL representation of the same scheme, programmatically extracted by the Parsing Engine. At right, a fragment of the resulting batch verifier generated after applying the Batcher and Code Generator.

directly from this Python code.[5]

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 [52], 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

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

signatures. This binary search strategy is borrowed from Law and Matt [54] and could be extended to support other methods 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.[6]

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 5.2.

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

[6]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.

5.4.1 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, 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 [51, 150]. 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.[7] Furthermore, unlike previous works we are able to programmatically identify when these techniques are applicable, and apply them to the verification equation in a consistent way.

The Batcher assumes that the input will be a collection of η signatures, possibly on different messages and public keys (or identities). To construct a batch verifier, 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. For consistency, the techniques are presented as implemented in AutoBatch and the technique numbers do not indicate any particular order.

[7] For example: techniques 2 and 3 of [150] each combine a series of logical operations that are more widely applicable and easily managed by splitting them into finer-grained sub-techniques.

Technique 0: Consolidate the verification equation. Many pairing-based signature schemes actually require the verifier to check more than one pairing equation. During the first phase of the batching process, the batcher applies the small exponents test from [52] to combine these equations into a single verification equation.[8] A variation of this is *Technique 10* which is applicable for schemes that utilize for loops in the verification equation (*e.g.*, VRF [19]). If the bounds over the loop are known it might be useful to unroll the loop to allow application of other techniques.

Replace for $i = 1$ to $t : e(g, h_i) \stackrel{?}{=} e(c, d_i)$

$$\text{with } e(g, h_1)^{\delta_1} \cdot ... \cdot e(g, h_t)^{-\delta_t} \stackrel{?}{=} e(c, d_1)^{\delta_1} \cdot ... \cdot e(c, d_t)^{-\delta_t}$$

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 [51], 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 5.3 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

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

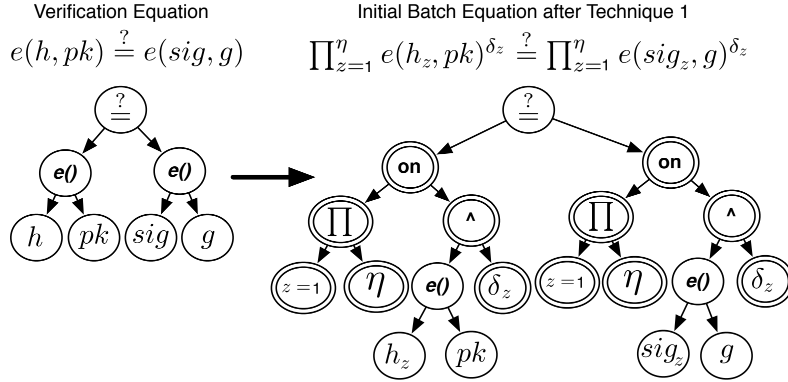


Figure 5.3: The Boneh-Lynn-Shacham (BLS) signature scheme [120] with same signer and η signatures in a batch. We show the abstract syntax tree (AST) of the unoptimized batch equation after Batcher has applied technique 1 by combining all instances of the verification equations (denoted by $\prod$ node) and applying the small exponents test (denoted by δ_z node).

elements of $\mathbb{G}_T$, this gives a noticeable speedup when computing the exponentiation.

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

Wherever possible, we move the exponent into the group with the lowest exponentiation cost. We identify this group based on a series of operation microbenchmarks that run automatically at code initialization.[9]

Technique 3: 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.

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

[9]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.

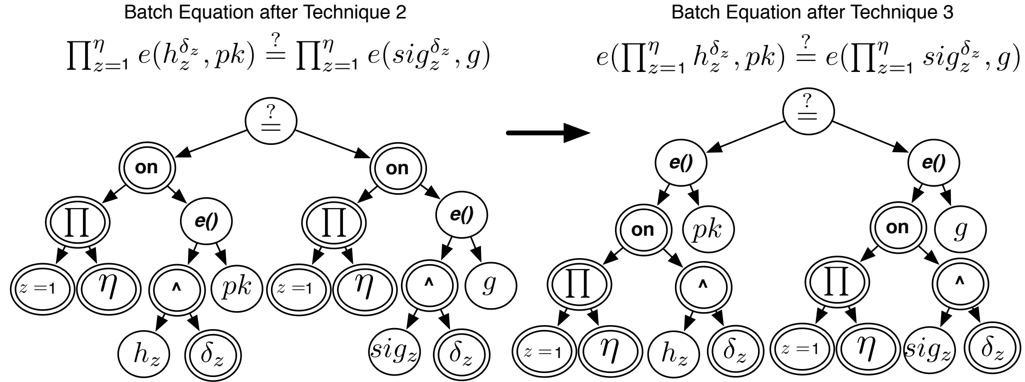


Figure 5.4: The Boneh-Lynn-Shacham (BLS) signature scheme [120] with same signer and η signatures in a batch. Upon applying technique 1 from Figure 5.3 to obtain the initial secure batch verifier, the goal is to optimize the equation. We first show the AST of the equation *after* Batcher has applied technique 2 (move exponents inside the pairing). Then, we show the result of applying technique 3 (move products inside the pairing) to arrive at an optimized batch equation.

A special case of this technique is *Technique 6* where $\eta = 2$. In this case, when two terms share a common first or second argument, they can also be combined. For example:

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

For a concrete example, we show how techniques 2 and 3 are programmatically applied to the BLS scheme [120] in Figure 5.4.

Technique 4: Optimize the Waters Hash. A variety of identity-based signature schemes employ a hash function by Waters [67], which can be generalized [158, 159]. 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 $\eta > z$.

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

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 7: Move known exponents outside pairing and precompute pairings. In some cases it may be necessary to move exponents outside of a pairing. For example, when $\prod_{i=1}^{\eta} e(g^{a_i}, h^{b_i})$ appears, move the exponents outside of pairing. When multiple such exponents appear, we can 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 8: 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 9: Split pairings. In some rare cases it can be useful to apply Technique 3 in reverse: splitting a single pairing into two or more pairings. This temporarily increases the number of pairings in the verification equation, but may be necessary in order to apply

subsequent techniques. For example, this optimization is necessary so that we can apply the Waters hash optimization (Technique 4) to the ring signature of Boyen [70].

Discussion: Several of the above techniques are quite simple, in that they perform optimizations that would seem “obvious" to an experienced cryptographer. However, many optimizations (*e.g.,* Technique 8) *could* have been applied in published algorithm descriptions [18, 19, 59], 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.

5.4.2 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.

Search Algorithm: 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 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 0 if there are multiple verification equations. After consolidating the verification equations, we try technique 6 since there may have been pairings with common elements from separate equations. Our intuition for attempting technique 6 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 8 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 10 (unroll loops) if the equation utilizes for loops. We delay testing for technique 10 until the post-processing stage to limit the search space for an efficient batch verifier. If technique 10 is applied, then we always attempt technique 6 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 9 performs a reverse split on pairings to allow further optimizations; this might affect technique 6, which combines pairings that have common elements. Certain combinations of techniques 9 and 6 lead to an infinite cycle that combines and splits the same pairings. Thus, the state function only allows a transition from Technique 9 to 6 to occur once on a given path. We provide the pseudocode of our search in Algorithm 1 and a table of our

finite state transition function in Figure 5.5.

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.

Algorithm 1 Pruned Breadth-First Search: the search algorithm takes as input the equation, sequences of techniques (called *path*) 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 Figure 5.5 and an estimate of the batch verifier runtime that is associated with each path. The user selects whichever path that yields the lowest runtime.

```
 1: procedure PBFSearch(eq, path, allPaths, technique)
 2:     applied, new_eq ← applyTechnique(technique, eq)
 3:     if applied = True then            → Technique condition is satisfied
 4:         path ← path + [technique]           → Append technique to path
 5:         checkRes ← checkForEdge(9, 6, path)
 6:         tech_list ← transitionFunc(technique, checkRes)
 7:         for all x ∈ tech_list do
 8:             newAllPaths ← PBFSearch(new_eq, path, allPaths, x)
 9:             allPaths ← allPaths ∪ newAllPaths
10:         end for
11:     else            → Reached dead end with this path
12:         if path ∉ allPaths then
13:             allPaths ← allPaths ∪ path            → Add path to set of all paths
14:             time ← estimateRuntime(eq, N, T)
15:             recordTime(time, path)            → record in a global database
16:         end if
17:     end if
18:     return allPaths
19: end procedure
```

Current State	Next States
2	3-9
3	2, 4, 5-7
4	2-3, 5-6
5	2-4
6	2-3, 5-6, 9
7	2, 5-6
8	2-3, 7
9	2, 4-6*, 7

Figure 5.5: The state transition table represents the transition function we developed for pruning our breath-first search (PBFS) algorithm. The function accepts as input the current state which represents the technique that was applied to the batch equation. The PBFS always starts in state 2 (where it tries to apply Technique 2). Then from there, the search attempts to follow any suggested states and applies the corresponding techniques. If the technique does not apply, the path is terminated. Otherwise, we check whether that path is already a subset of the paths we have covered so far. We continue with the search until all open paths are terminated. In an effort to ensure that all paths terminate, the state function restricts the transition from Technique 9 to 6 to occur once on a given path (indicated by *). 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. Once all paths are terminated, we attempt to apply Technique 10 to each path in a post-processing phase.

5.4.3 Security and Machine-Aided Analysis

Efficiency Analysis. Efficiency of the batch verifiers are computed in two separate ways. During the PBFS algorithm, 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 1, 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 verifica-

tion 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 5.3.1. 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).[10]

Theorem 5.4.1 (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 0 and 1 to the individual verification equation(s) using security parameter λ to obtain a single equation X.*

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

3. *Repeat previous step until none of the techniques apply and then return X.*

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

Then all AutoBatch algorithms unconditionally satisfy Definition 5.3.1, where the probability of accepting an invalid batch is at most $2^{-\lambda}$.

Proof. We analyze this proof in two parts. First, after Step 1 (the application of Techniques 0 and 1), there will be one batch equation X and it will satisfy the security requirements of Definition 5.3.1 with error probability $2^{-\lambda}$. These two techniques combine a set of equations into a single equation using the Small Exponents Test with security parameter λ. Ferrara et al. [51, Theorem 3.2] prove that this equation will verify if and only if all individual equations verify, except with probability at most $2^{-\lambda}$. By default in AutoBatch, we set $\lambda = 80$.

Next, given a single arbitrary, pairing-based equation X, we apply one of Techniques 2-9. For each Technique 2-9, we argue that the output equation X' holds if and only if the input equation X holds; that is, the equations are identical up to algebraic manipulations. If this is true, the final batch equation output by AutoBatch satisfies Definition 5.3.1 with the same error probability as the equation output after Techniques 0 and 1 were applied, completing the theorem.

It remains to argue that for each Technique 2-9, it is indeed the case that the input and output equations are identical, up to algebraic manipulations. Techniques 2, 3, 4, 6, 7 and 9 follow relatively straightforwardly from the bilinearity of the groups. As an example, consider Technique 6 which claims that $e(a, g) \cdot e(b, g) = e(a \cdot b, g)$, 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 8 pre-computes and caches values instead of re-computing them on the fly. □

To offer transparency on how AutoBatch derived any given batch verifier, 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 5.4.1 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 [135] was produced automatically by AutoBatch.

Full proofs for the Hohenberger-Waters (HW) scheme [135], the Camenisch-Lysyanskaya (CL) scheme [18], and the Verifiable Random Functions (VRF) scheme [19] are given in Appendices A.4, A.5, and A.6, respectively. In Appendix A.7, we detail the results of AutoBatch on the Waters09 scheme (derived from the Waters Dual-System IBE of [57]); 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.

5.4.4 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 useable 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)[11] the membership to the group is tested using an exponentiation. Section 5.3.2 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.

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

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 e.g,. [51] 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.

5.5 Implementation & Performance

Subsequent to our initial publication of the conference version of this work, we identified a software bug in the group membership function of Charm v0.42 that affected our results. The results in this chapter include the corrections to the affected group membership test which reduces the efficiency gains of batch verification in all our test cases. In

	Approx. Signature Size		MIRACL w/ BN256		RELIC w/ BN256	
	MNT160	BN256	Individual	Batched*	Individual	Batched*
Signatures						
BLS [42] (same signer)	160 bits	256 bits	26.6 ms	2.2 ms	11.9 ms	1.5 ms
CHP [150] (same time period)	160 bits	256 bits	46.1 ms	7.2 ms	24.0 ms	7.8 ms
HW [135] (same signer)	320 bits	512 bits	40.5 ms	4.7 ms	22.4 ms	3.0 ms
HW [135] (diff signer)	320 bits	512 bits	40.5 ms	61.1 ms	22.4 ms	29.2 ms
Waters09 [57, §6.1] (same signer)	6240 bits	6912 bits	153.2 ms	33.1 ms	93.7 ms	44.2 ms
CL [18] (same signer)	480 bits	768 bits	72.0 ms	15.9 ms	34.6 ms	18.0 ms
ID-Based Signatures						
Hess [136]	1120 bits	3328 bits	32.7 ms	22.0 ms	17.1 ms	8.4 ms
ChCh [137]	320 bits	512 bits	27.5 ms	4.6 ms	12.6 ms	2.4 ms
Waters05 [67]	480 bits	768 bits	45.3 ms	11.8 ms	21.5 ms	11.0 ms
Group, Ring and ID-based Ring Signatures						
BBS [59] Group signature	2400 bits	5376 bits	99.9 ms	31.2 ms	63.9 ms	18.7 ms
Boyen [70] Ring signature, 3-member ring	960 bits	1536 bits	64.2 ms	15.0 ms	41.5 ms	9.8 ms
CYH [160] 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.

Figure 5.6: 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.

particular, there are noticeable reductions in performance for CL [18], Waters09 [57] and HW (with different signers) [135]. Although an optional feature, our membership tests include public keys to reflect the worst case performance of batch verification without invalid signatures in the batch. See Figure 5.8 for the new graphs.

5.5.1 Experimental Setup

To evaluate the performance of our techniques we implemented them as part of the Charm prototyping framework [11]. 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 [87].[12] The necessary MIRACL calls are accessed from within our Python code via the C module interface.

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

Hardware configuration. For consistent results we ran all of our experiments on a single hardware platform: a 2 x 2.66 GHz 6-Core Intel Xeon Macintosh Pro running MacOS version 10.7.3 with 12GB of RAM. We ran all of our tests within a single thread, and thus used resources from only a single core of the Intel processor. We instantiated all of our cryptographic implementations using a 160-bit MNT elliptic curve and 256-bit Barreto-Naehrig (BN) curve provided with MIRACL shown in Figures 5.6 and 5.8.

[12]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.

A note on the library. We chose MIRACL because it is mature and well supported. However, some research libraries like RELIC [139] 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 Figure 5.6.

5.5.2 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, group signatures and 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 Figure 5.7.

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

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	[42]	2	1,2,3
2. Camenisch et al. (CHP) (same period)	S	RO	3η	3	[150]	3	1,2,3,5,3
3. Camenisch-Lysyanskaya (CL) (ss)	S	P	5η	5η	none	**3**	0,1,2,6,6,3,5,3
4. Hohenberger-Waters (HW) (ss)	S	P	2η	2η	none	**4**	1,2,3,9,7,5,3
5. Hohenberger-Waters (HW)	S	P	2η	2η	none	**4**	1,2,3,9,5,3
6. Waters09 (ss)	S	P	9η	9η	none	**13**	1,2,9,5,3,7,6
7. Hess	I	RO	2η	2	[51]	2	1,2,3
8. Cha-Cheon (ChCh)	I	RO	2η	2	[54]	2	1,2,3
9. Waters05	I	P	3η	$z+3$	[150]	$z+3$	1,2,3,9,7,5,3,4,6
10. ChCh and Hess together	M	RO	2η	4	[51, 54]	**2**	0,1,2,3,5,3,6
11. Chow-Yiu-Hui (CYH)	IR	RO	2η	2	[51]	2	1,2,3,2
12. Boyen (same ring)	R	P	$\ell\eta+\ell$	$3\ell+1$	[51]	$3\ell+1$	1,2,9,4,6,9,5,3
13. Boneh-Boyen-Shacham (BBS)	G	RO	5η	2	[51]	2	1,2,6,6,5,3
14. VRF equations 1,3,4 & 2 (ss)	V	P	$\mathbf{3\eta+2\ell}$	$\mathbf{3\ell+1}$	[19]	$\mathbf{\ell+3}$	0,6,1,2,3,1,2,3,5,3,6,10,6

Figure 5.7: 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 that can be set to 5 in practice. To approximate verification performance, we count the total number of pairings needed to process η valid signatures. Unless otherwise noted, the inputs are from different signers. The final column indicates the order of the techniques from Section 5.4 that AutoBatch applied to obtain the resulting batch verifier. The **rows in bold** are the schemes where AutoBatch discovered new or improved algorithms.

We briefly recall the verification equations in VRF [19]. 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 [19] that takes only two-thirds the time of the one provided in [19] (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 A.6.

In test case 10 shown in Figure 5.7 (ChCh [137] and Hess [136] 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 [137] signatures and Hess [136] 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 5.4.1. 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 expo-

nents test). In the case of the ChCh [137] and Hess [136] 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 [135], we assume that each public key includes the precomputed values as suggested in [135, 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 [57] using the technique described by Naor [28]. 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 A.4 and A.7 respectively. Finally, the details of the batching of CL signatures by the same signer appear in Appendix A.5.

5.5.3 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, Python and C++ code for implementing the batch verifier. We measured the processing

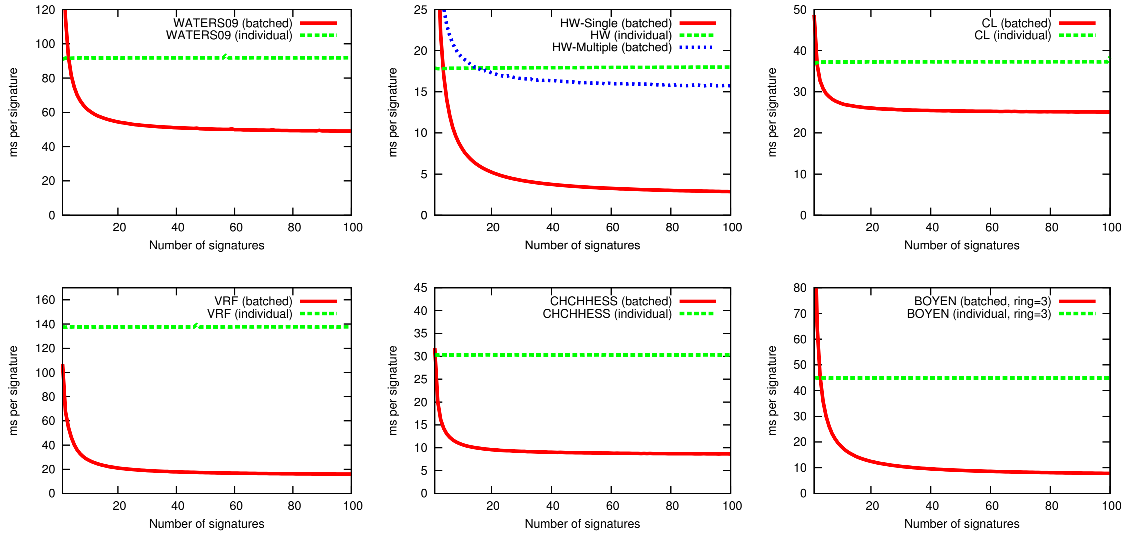


Figure 5.8: Signature scheme microbenchmarks for Waters09 [57], HW [135] and CL [18] public-key signatures (same signer), the VRF [19] (with block size of 8), combined verification of ChCh+Hess IBS [136, 137], and Boyen ring signature (3 signer ring) [70]. Per-signature times were computed by dividing 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.

time for each of the above steps. Our timings, averaged over 100 runs, are presented in Figure 5.9.

To obtain our microbenchmarks, we ran AutoBatch on several exemplary pairing-based schemes as listed in Figure 5.7. We then experimented with these schemes at different batch sizes, in order to evaluate their raw performance. The results are presented in Figure 5.8.

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,[13] then computing a valid signature over each message.

[13] We used 100-byte random strings for each message. In the case of the stateful HW signature, we batched

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.

5.5.4 Batch Verification in Practice

Prior works considered the implication of *invalid* signatures in a batch, *e.g.,* [51,54,153, 154, 161]. Mainly, these works estimated raw signature verification times under various conditions. To evaluate how signature batching might work in real life, we constructed a simulation to determine the resilience of our techniques to various denial of service attacks launched by an adversary.

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

Our server is designed to process as many signatures as possible, and is limited only by its computational resources.[14] Signatures are drawn off of the "wire" and grouped into batches, with each batch size representing the expected number of signatures that can be

only signatures with the same counter value.

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

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

Figure 5.9: 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 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.

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.[15] This approach can lead to some transient errors (batches that require significantly more or less than one second to evaluate) when the initial guess is wrong, or when conditions change. In normal usage, however, this approach converges on an appropriate batch size within 1-2 seconds.

5.5.4.1 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 §5.4, each of our batch verifiers incorporates a recursive divide-and-conquer strategy for identifying these invalid signatures, which is borrowed from Law and Matt [54]. 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.

[15]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.

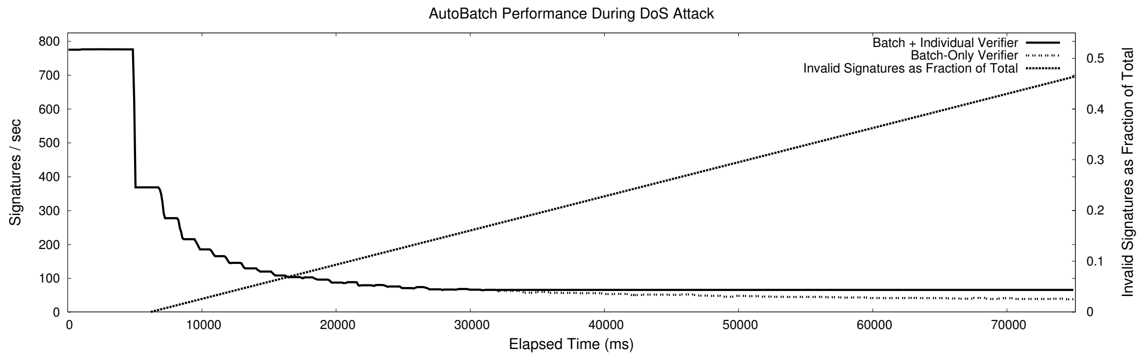


Figure 5.10: Simulated service denial attacks against a batch verifier (BLS signatures, single signer). The "Invalid Signatures as Fraction of Total" line (right scale) shows the fraction of invalid signatures in the stream. Batcher throughput is measured in signatures per second (left scale). The "Batch-Only Verifier" line depicts a standard batch verifier. The solid line is a batch verifier that automatically switches to *individual* verification when batching becomes suboptimal.

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,[16] we measured the verifier's throughput. Our adversary injects no invalid signatures for the first several seconds of the experiment, then gradually ramps up its output until the number of invalid signatures received by the verifier approaches 50%.

A switch to individual verification. Our experiments 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.[17] To address this, we implemented a "countermeasure" in our batch verifier to automatically switch to individual verification whenever it

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

[17]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.

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 5.10. 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.

5.6 AutoBatch Toolkit

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

5.7 Challenges and Open Problems

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 [133, 134]. 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 5.4.1 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 [140] 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., [54, 153, 154, 161]. 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.

Chapter 6

Using SMT solvers to Automate Design Tasks for Encryption and Signature Schemes

In the previous chapter, we presented an implementation of a tool that automated batch verification design. In this chapter, we will explore the automation of two additional types of general transformations that are common in the literature. One transformation deals with the optimizing the efficiency and bandwidth of signatures and we show that our techniques extend to encryption schemes as well. The second transformation addresses strengthening the security of signature schemes. In both cases, we demonstrate that our architecture is effective in implementing such transformations and we discuss the security limitations in applying transformations to certain cryptographic schemes.

6.1 Overview

Cryptographic design tasks are primarily performed by hand today. Shifting more of this burden to computers could make the design process faster, more accurate and less expensive. In this work, we investigate tools for programmatically altering existing cryptographic constructions to reflect particular design goals. Our techniques enhance both security and efficiency with the assistance of advanced tools including Satisfiability Modulo Theories (SMT) solvers.

Specifically, we propose two complementary tools, AutoGroup and AutoStrong. AutoGroup converts a pairing-based encryption or signature scheme written in (simple) symmetric group notation into a specific instantiation in the more efficient, asymmetric setting. Some existing symmetric schemes have hundreds of possible asymmetric translations, and this tool allows the user to optimize the construction according to a variety of metrics, such as ciphertext size, key size or computation time. The AutoStrong tool focuses on the security of digital signature schemes by automatically converting an existentially unforgeable signature scheme into a *strongly* unforgeable one. The main technical challenge here is to automate the "partitioned" check, which allows a highly-efficient transformation.

These tools integrate with and complement the AutoBatch tool (ACM CCS 2012), but also push forward on the complexity of the automation tasks by harnessing the power of SMT solvers. Our experiments demonstrate that the two design tasks studied can be performed automatically in a matter of seconds.

6.2 Introduction

Cryptographic design is challenging, time consuming and mostly performed by hand. A natural question to ask is: to what extent can computers ease this burden? Which common design tasks can computers execute faster, more accurately or less expensively?

In particular, this work investigates tools for programmatically altering existing cryptographic constructions in order to enhance efficiency or security design goals. For instance, digital signatures, which are critical for authenticating data in a variety of settings, ranging from sensor networks to software updates, come in many possible variations based on efficiency, functionality or security. Unfortunately, it is often infeasible or tedious for humans to document each possible optimal variation for each application. It would be enormously valuable if there could be a small number of simple ways to present a scheme – as simple as possible to avoid human-error in the design and/or verification process – and then computers could securely provide any variation that may be required by practitioners.

A simple, motivating example (which we explore in this work) is the design of pairing-based signature schemes, which are often presented in a simple "symmetric" group setting that aids in exposition, but does not map to the specific pairing-based groups that maximize efficiency. Addressing this disconnect is ripe for an automated tool.

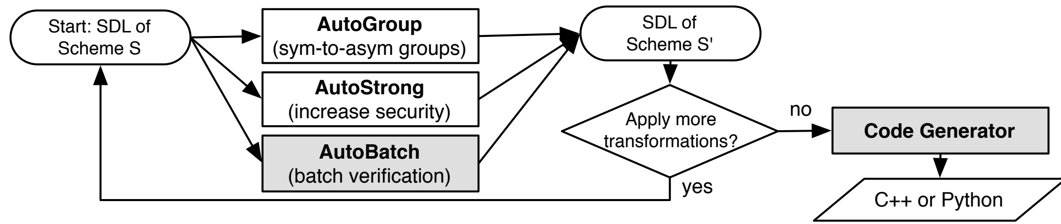


Figure 6.1: A high-level presentation of the new automated tools, AutoGroup and AutoStrong. They take as input a Scheme Description Language (SDL) representation of a cryptographic scheme and output an SDL representation of a transformation of the scheme, which can possibly be further transformed by another tool. These tools are compatible with the existing AutoBatch tool and Code Generator (shaded). An SDL input to the Code Generator produces a software implementation of the scheme in either C++ or Python.

6.2.1 Our Contributions

In this work, we explore two novel types of design problems for pairing-based cryptographic schemes. The first tool (AutoGroup) deals with efficiency, while the second (AutoStrong) deals with security. We illustrate how they interact in Figure 6.1. The tools take a Scheme Description Language (SDL) representation of a scheme (and optionally some user optimization constraints) and output an SDL representation of the altered scheme. This SDL output can be run through another tool or a Code Generator to produce C++ or Python software. We provide more details on our SDL in Section 3.5.1.

A contribution of this work is that we integrated our tools with the publicly-available source code for AutoBatch [17, 162], a tool that automatically identifies a batch verification algorithm for a given signature scheme, therein weaving together a larger automation system. For instance, a practitioner could take any symmetric-pairing signature scheme from the literature, use AutoGroup to reduce its bandwidth in the asymmetric setting, use AutoBatch to reduce its verification time, and then automatically obtain a C++ implemen-

tation of the optimized construction. Our work appears unique in that we apply advanced tools, such as SMT solvers and Mathematica, to perform complex design tasks related to pairing-based schemes.

Automated Task 1: Optimize Efficiency of an Encryption or Signature Scheme via User Constraints. Pairings are often studied because they can realize new functionalities, e.g., [28, 66], or offer low-bandwidth solutions, e.g., [66, 120]. Pairing (a.k.a., bilinear) groups consist of three groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ with an efficient bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. Many protocols are presented in a *symmetric* setting where $\mathbb{G}_1 = \mathbb{G}_2$ (or equivalently, there exists an efficient isomorphism from $\mathbb{G}_1$ to $\mathbb{G}_2$ or vice versa).

While symmetric groups simplify the description of new cryptographic schemes, the corresponding groups are rarely the most efficient setting for implementation [163]. The state of the art is to use *asymmetric* groups where $\mathbb{G}_1 \neq \mathbb{G}_2$ and no efficient isomorphism exists between the two. See for instance the work of Ramanna, Chatterjee and Sarkar [56] (PKC 2012) which translates the dual system encryption scheme of Waters [53] from the symmetric to a handful of asymmetric settings.

Such conversions currently require manual analysis (of all steps) – made difficult by the fact that certain operations such as group hash functions only operate in a single group. Moreover, in some cases, there are hundreds of possible symmetric to asymmetric translations, making it tedious to identify the optimal translation for a particular application.

We propose a tool called AutoGroup that automatically provides a “basic” translation

from symmetric to asymmetric groups.[1] It employs an SMT solver to identify valid group assignments for all group elements and also accepts user constraints to optimize the efficiency of the scheme according to a variety of metrics, including signature/ciphertext size, signing/encryption time, and public parameter size. The tool is able to enumerate the full set of possible solutions (which may run to the hundreds), and can rapidly identify the most efficient solution.

Automated Task 2: Strengthen the Security of a Digital Signature Scheme. Most signature schemes are presented under the classic, existential unforgeability definition [21], wherein an adversary cannot produce a signature on a “new” message. However, *strong* unforgeability guarantees more – that the adversary cannot produce a “new” signature even on a previously signed message. Strongly-unforgeable signatures are often used as a building block in signcryption [13], chosen-ciphertext secure encryption [14, 58] and group signatures [15, 59].

There are a number of general transformations from classic to strong security [39, 164–168], but also a highly-efficient transformation due to Boneh, Shen and Waters [12] that only applies to “partitioned” schemes. We propose a tool called AutoStrong that automatically decides whether a scheme is “partitioned” and then applies BSW if it is and a general transformation otherwise. The partitioned test is non-trivial, and our tool harnesses the power of both an SMT solver and Mathematica to make this determination. We are careful

[1] By "basic", we mean that it translates the scheme as written into the asymmetric setting, with minor optimizations performed, but does not attempt a re-imagining of the construction based on a stronger asymmetric complexity assumption. While the latter is sometimes possible, e.g., [56], it may not be required in some applications and the novel security analysis required places it beyond the current ability of our automation tools. See Section 6.4.3 for more.

to err only on false negatives (which impact efficiency), but not false positives (which could compromise security.) Earlier works [39, 168] claimed that there were "very few" examples of partitioned schemes; however, our tool proved this was not the case by identifying valid partitions for most schemes we tested.

6.2.2 Related Work

Many exciting works have studied how to automate various cryptographic tasks. Automation has been introduced into the design process for various security protocols [1–3, 61], optimizations to software implementations involving elliptic-curves [155] and bilinear-map functions [156], the batch verification of digital signature schemes [17], secure two-party computation [9, 10, 64], and zero-knowledge proofs [4–8].

Our current work is most closely related to the AutoBatch tool of Akinyele et al. [17] and we designed our tools so that they can integrate with the publicly-available source code of AutoBatch [162] to form a larger, more comprehensive solution. This work is different from AutoBatch in that it attacks new, more complicated design tasks and integrates external SMT solvers and Mathematica to find its solutions.

Prior work on automating the writing and verification of cryptographic proofs, such as the EasyCrypt work of Barthe et al. [140], are complimentary to but distinct from our effort. Their goal was automating the construction and verification of (game-based) cryptographic *proofs*. Our goal is automating the construction of cryptographic *schemes*. A system that combines both to automate the design of a scheme and then automate its security analysis

would be optimal.

6.3 Tools Used

Our automations make use of three external tools. First, Z3 [34,35] is a freely-available, state-of-the-art and highly efficient Satisfiability Modulo Theories (SMT) solver produced by Microsoft Research. SMT is a generalization of boolean satisfiability (SAT) solving, which determines whether assignments exist for boolean variables in a given logical formula that evaluates the formula to *true*. SMT solvers builds on SAT to support many rich first-order theories such as equality reasoning, arithmetic, and arrays. In practice, SMT solvers have been used to solve a number of constraint-satisfaction problems and are receiving increased attention in applications such as software verification, program analysis, and testing. Z3 in particular has been used as a core building block in API design tools such as Spec#/Boogie [36, 37] and in verifying C compilers such as VCC.

We leverage Z3 v4.3.1 to perform reasoning over statements involving arithmetic, quantifiers, and uninterpreted functions. We use Z3's theories for equality reasoning combined with the decision procedures for linear arithmetic expressions and elimination of universal quantifiers (*e.g.,* $\forall x$) over linear arithmetic. Z3 includes support for uninterpreted (or *free*) functions which allow any interpretation consistent with the constraints over free functions and variables.

Second, we utilize the development platform provided by Wolfram Research's Mathe-

matica [38] (version 9), which allows us to simplify equations for several of our analytical techniques. We leverage Mathematica in our automation to validate that given cryptographic algorithms have certain mathematical properties. Finally, we utilize some of the publicly-available source code of the AutoBatch tool [162], including its Scheme Description Language (SDL) parser and its Code Generator, which translates an SDL representation to C++ or Python.

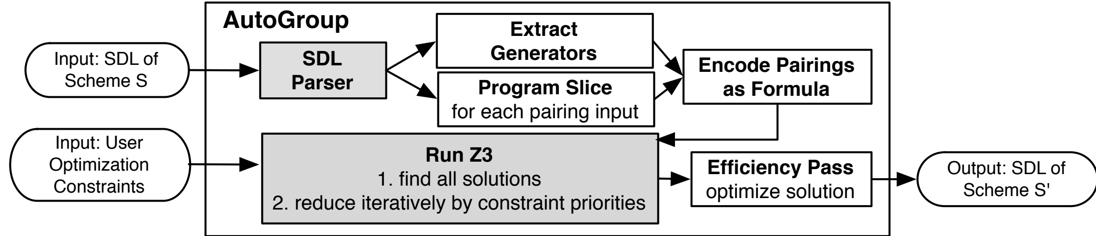


Figure 6.2: A high-level presentation of the AutoGroup tool, which uses external tools Z3 and SDL Parser.

6.4 AutoGroup

In this section, we present and evaluate a tool, called AutoGroup, for automatically altering a cryptographic scheme's algebraic setting to optimize for efficiency.

6.4.1 Background on Pairing Groups

Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be algebraic groups of prime order p.[2] We recall that $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is a pairing (a.k.a., bilinear map) if it is: efficiently-computable, (*bilinear*) for all $g \in \mathbb{G}_1$,

[2]Pairing groups may also have composite order, but we will be focusing on the more efficient prime order setting here.

$h \in \mathbb{G}_2$ and $a, b \leftarrow \mathbb{Z}_p$, $e(g^a, h^b) = e(g, h)^{ab}$; and (*non-degenerate*) if g generates $\mathbb{G}_1$ and h generates $\mathbb{G}_2$, then $e(g, h) \neq 1$. This is called the *asymmetric* setting. A specialized case is the *symmetric* setting, where $\mathbb{G}_1 = \mathbb{G}_2$.[3]

In practice, all efficient candidate constructions for pairing groups are constructed such that $\mathbb{G}_1$ and $\mathbb{G}_2$ are groups of points on some elliptic curve E, and $\mathbb{G}_T$ is a subgroup of a multiplicative group over a related finite field. The group of points on E defined over $\mathbb{F}_p$ is written as $E(\mathbb{F}_p)$. Usually $\mathbb{G}_1$ is a subgroup of $E(\mathbb{F}_p)$, $\mathbb{G}_2$ is a subgroup of $E(\mathbb{F}_{p^k})$ where k is the embedding degree, and $\mathbb{G}_T$ is a subgroup of $\mathbb{F}^*_{p^k}$. In the symmetric case $\mathbb{G}_1 = \mathbb{G}_2$ is usually a subgroup of $E(\mathbb{F}_p)$.

The challenge in selecting pairing groups is to identify parameters such that the size of $\mathbb{G}_T$ provides acceptable security against the MOV attack [169] by Menezes, Vanstone and Okamoto. Hence the size of p^k must be comparable to that of an RSA modulus to provide the same level of security – hence elements of $\mathbb{F}_{p^k}$ must be of size approximately 3,072 bits to provide security at the 128-bit symmetric equivalent level. The group order q must also be large enough to resist the Pollard-ρ attack on discrete logarithms, which means in this example $q \geq 256$.

Two common candidates for implementing pairing-based constructions are supersingular curves [170, 171] in which the embedding degree k is ≤ 6 and typically smaller (an example is $|p| = 1536$ for the 128-bit security level at $k = 2$), or ordinary curves such as MNT or Barreto-Naehrig (BN) [172]. In BN curves in particular, the embedding degree

[3] An alternative instantiation of the symmetric setting has $\mathbb{G}_1 \neq \mathbb{G}_2$ but admits an efficiently-computable isomorphism between the groups.

$k = 12$, thus $|p| = |q|$ can be as small as 256 bits at the 128-bit security level, with a corresponding speedup in field operations.

A challenge is that the recommended BN subgroups do not possess an efficiently-computable isomorphism from $\mathbb{G}_1$ to $\mathbb{G}_2$ or vice versa, which necessitates re-design of some symmetric cryptographic protocols. A related issue is that BN curves permit efficient hashing only into the group $\mathbb{G}_1$. This places restrictions on the set of valid group assignments we can use.

6.4.2 How AutoGroup Works

AutoGroup is a new tool for automatically translating a pairing-based encryption or signature scheme from the symmetric-pairing setting to the asymmetric-pairing setting. At a high-level, AutoGroup takes as input a representation of a cryptographic protocol (*e.g.,* signature or encryption scheme) written in a Domain-Specific Language called Scheme Description Language (SDL), along with a description of the optimizations desired by the user. These optimizations may describe a variety of factors, e.g., requests to minimize computational cost, key size, or ciphertext / signature size. The tool outputs a new SDL representation of the scheme, one that comprises the optimal assignment of groups for the given constraints. The assignment of groups is non-trivial, as many schemes are additionally constrained by features of common asymmetric bilinear groups settings, most notably, restrictions on which groups admit efficient hashing. At a high level, AutoGroup works by reducing this constrained group assignment problem to a boolean satisfiability prob-

lem, applying an SMT solver, and processing the results. We next describe the steps of AutoGroup, as illustrated in Figure 6.2.

1. Extract Generator Representation. The first stage of the AutoGroup process involves parsing SDL to identify all base generators of $\mathbb{G}$ that are used in the scheme. For each generator $g \in \mathbb{G}$, AutoGroup creates a pair of generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$. This causes an increase in the parameter size of the scheme, something that we must address in later steps.

We assume the Parser knows the basic structure of the scheme, and can identify the algorithm responsible for parameter generation. This allows us to parse the algorithm to observe which generators that are created. When AutoGroup detects the first generator, it marks this as the "base" generator of $\mathbb{G}$ and splits g into a pair $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$. Every subsequent group element sampled by the scheme is defined in terms of the base generators. For example, if the setup algorithm next calls for "choosing a random generator h in $\mathbb{G}$", then AutoGroup will select a random $t' \in \mathbb{Z}_p$ and compute new elements $h_1 = g_1^{t'}$ and $h_2 = g_2^{t'}$.

2. Traceback Inputs to the Pairing Function. Recall that the pairing function $e(A, B)$ takes two inputs. We extract all the pairings required in the scheme; these might come from the setup algorithm, encryption/signing, or decryption/verification. Prior to tracing the pairing inputs, we split pairings of the form $e(g, A \cdot B)$ as $e(g, A) \cdot e(g, B)$ to prepare for encoding pairings as logical formulas in the SMT solver. In the final step of AutoGroup we recombine the pairings to preserve efficiency. We reuse techniques introduced in [17, 51]

to split and combine pairings in AutoGroup.

After splitting applicable pairings, we obtain a program slice for each variable input to determine which (symmetric) generators were involved in computing it. This also helps us later track which variables are affected when an assignment for a given variable is made in $\mathbb{G}_1$ or $\mathbb{G}_2$. Consider the example $A = X \cdot Y$. Clearly, the group assignment of A affects variables X and Y, and capturing the slice for each pairing input variable is crucial for AutoGroup to perform correct re-assignment for the subset of affected variables.

3. Convert Pairings to Logical Formulas. Asymmetric pairings require that one input to the function be in $\mathbb{G}_1$, and the other be in $\mathbb{G}_2$. Conversion from a symmetric to an asymmetric pairing can be reduced to a constraint satisfiability problem; we model the asymmetric pairing as an inequality operator over binary variables. This is analogous because an inequality constraint enforces that the binary variables either have a 0 or 1 value, but not both for the equation to be satisfiable. Therefore, we express symmetric pairings as a logical formula of inequality operators over binary variables separated by conjunctive connectors (*e.g.,* $A \neq B \wedge C \neq D$). We then employ an SMT solver to find a satisfiable solution and apply the solver's solution to produce an equivalent scheme in the asymmetric setting.

4. Convert Pairing Limitations into Constraints. When translating from the symmetric to the asymmetric pairing setting, we encounter several limitations that must be incorporated into our model. Chief among these are limitations on hashing: in some asymmetric groups, hashing to $\mathbb{G}_2$ is not possible. In other groups, there is no such isomorphism, but it is possible to hash into $\mathbb{G}_1$. Depending on the groups that the user selects, we must identify

an asymmetric solution that respects these constraints. Fortunately these constraints can easily be expressed in our formulae, by simply assigning the output of hash functions to a specific group, e.g., $\mathbb{G}_1$.

5. Execute SMT Solver. We run the logical formula plus constraints through an SMT solver to identify a satisfying assignment of variables. The solver checks for a satisfiable solution and produces a model of 0 (or $\mathbb{G}_1$) and 1 (or $\mathbb{G}_2$) values for the pairing input variables that satisfies the specified constraints. We can go one step further and enumerate all the unique solutions (or models) found by the solver for a given formula and constraints. After obtaining all the possible models, we utilize the solver to evaluate each model and determine the solutions that satisfies the user's application-specific requirements.

6. Satisfy Application-specific Requirements. To facilitate optimizations in the asymmetric setting that suit user applications, we allow users to specify additional constraints on the chosen solution. There are two possible ways of tuning AutoGroup: one set of options focus on reducing the size of certain scheme outputs. For public key encryption, the user can choose to minimize the representation of the secret keys, ciphertext or both. Similarly, for signatures schemes, the user can optimize for minimal-sized public keys, signatures or both. The second set of options focus on reducing algorithm execution times. This is possible due to the fact that for many candidate asymmetric groups, group operations in $\mathbb{G}_1$ are dramatically more efficient than those that take place in $\mathbb{G}_2$. Users may also combine various operations, in order to find an optimal solution based on a combination of size and operation time.

We find application-specific solutions by minimizing an objective function over all the possible models obtained from the solver. Our objective function is straightforward and calculated as follows:

$$F(A, C, w_1, w_2) = \sum_{i=1}^{n}((1 - a_i) \cdot w_1 + a_i \cdot w_2) \cdot c_i$$

where $A = a_i, \ldots, a_n$ and represents the pairing input variables, w_1 and w_2 denote *weights* over groups $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively, $C = c_i, \ldots, c_n$ and each c_i corresponds to the *cost* for each a_i. Each input variable a_i can have a value of $0 = \mathbb{G}_1$ or $1 = \mathbb{G}_2$. We now describe how the above options are converted into parameters of F and discuss how the SMT solver is used to obtain a minimal solution.

For each parameter that we intend to optimize, we define a *weight function* that evaluates each candidate solution according to some metric. For each assigned variable, the weight function calculates the total "cost" of the construction as a function of some cost value for the specific variable, as well as an overall cost for an assignment of $\mathbb{G}_1$ and $\mathbb{G}_2$. In the case of ciphertext size we assign the cost value to 1 for each group element that appears in the ciphertext, and 0 for all others. For encryption time, we assign a cost that corresponds to the number of group operations applied to this variable during the encryption operation. The overall cost value then determines the cost of placing a value in one of the two groups – for size-related calculations, this roughly corresponds to the length of a group element's representation, and for operation time it corresponds to the cost of a single group operation. By assigning these costs correctly, we are able to create a series of different weight functions that represent all of the different values that we would like to minimize

(e.g., ciphertext size, parameter size, time).

If the user chooses to optimize for multiple criteria simultaneously, we must find a model that balances between all of these at the same time. This is not always possible. For example, some schemes admit solutions that favor a minimized secret key size or ciphertext size, but not both. In this case, we allow the user to determine which constraint to relax and thereby select the next best solution that satisfies their requirements.

7. Evaluate and Process the Solution. Once the application-specific solution is obtained from the solver, the next step is to apply the solution to produce an asymmetric scheme. As indicated earlier, we interpret the solution for each variable as $0 = \mathbb{G}_1$ and $1 = \mathbb{G}_2$. To apply the solution, we first pre-process each algorithm in SDL to determine how the pairing inputs are affected by each assignment. Consider a simplistic example: $e(A, B)$ where $A = g^a$ and $B = h^b$. Let us assume that the satisfying solution is that $A \in \mathbb{G}_1$ and $B \in \mathbb{G}_2$. Therefore, we would rewrite these two variables as $A = g_1^a$ and $B = h_2^b$ where $g_1 \in \mathbb{G}_1$ and $h_2 \in \mathbb{G}_2$. The program slice recorded for each pairing input in step (2) provides the necessary information to correctly rewrite the scheme in the asymmetric setting.

In addition to rewriting the scheme, AutoGroup performs several final optimizations. First, it removes any unused parameter values in the public and secret keys. For signature schemes, we try to optimize further by reducing the public parameters used per algorithm. In particular, we trace which variables in the public key are actually used during signing and verification. For elements that appear only in the signing (resp. decryption) algorithms, we split the public key into two: one is kept just for computing signatures (resp. decryption),

Encryption	Time			Approx. Size		Num.
	Keygen•	Encrypt•	Decrypt•	Secret Key	Ciphertext	Solutions
ID-Based Enc.						
BB04 [173, §4] Symmetric (SS1536)	59.9 ms	64.8 ms	125.4 ms	3072 bits	6144 bits	
Asymmetric (BN256) [Min. CT]	4.8 ms	7.8 ms	27.6 ms	2048 bits	3584 bits	4
Gentry06 [174, §3.1] Symmetric (SS1536)	39.9 ms	176.2 ms	67.8 ms	3072 bits	7680 bits	
Asymmetric (BN256) [Min. SK]	1.4 ms	41.0 ms	19.1 ms	512 bits	7168 bits	4
WATERS09 [53, §3.1] Symmetric (SS1536)	294.6 ms	286.8 ms	612.8 ms	13824 bits	18432 bits	
Asymmetric (BN256) [Min. SK/CT/Exp]	12.6 ms	19.2 ms	128.0 ms	5376 bits	8704 bits	256
Broadcast Encryption						
BGW05 [33, §3.1] Symmetric (SS1536) (n = 100)	1992.2 ms	119.6 ms	136.9 ms	19200 bytes	6144 bits	
Asymmetric (BN256) [Min. SK]	70.4 ms	25.7 ms	28.5 ms	3200 bytes	5120 bits	4

•Average time measured over 100 test runs and the standard deviation in all test runs were within ±1% of the average.

Figure 6.3: AutoGroup on encryption schemes under various optimization options. We show running times and sizes for several schemes generated in C++ and compare symmetric to automatically generated asymmetric implementations at the same security levels (roughly equivalent with 3072 bit RSA). For IBE schemes, we measured with the identity string length at 100 bytes. For BGW, n denotes the number of users in the system.

and the other is given out for use in encryption/verification. Second, AutoGroup performs an additional efficiency check and attempts to optimize pairing product equations to use as few pairings as possible. This is due to the decoupling of pairings in earlier phases of translating the scheme to the asymmetric setting or perhaps, just a loose design by the original SDL designer. In either case, we apply pairing optimization techniques from previous work [17,51] to provide this automatic efficiency check. Finally, AutoGroup outputs a new SDL of the modified scheme.

We do not offer the efficiency check of AutoGroup as a standalone tool for symmetric groups at present, because our experience inclines us to believe that most practitioners concerned with efficiency will want to work in asymmetric groups. However, our results herein also demonstrate that a simple tool of this sort is efficient and feasible.

6.4.3 Security Analysis of AutoGroup

Whether a scheme is translated by hand (as is done today [56]) or automatically (as in this work), a completely separate question applying to both is: is the resulting asymmetric scheme secure? The answer is not immediately clear. Unlike the signature transformation that we automate in Section 6.5 that already has an established security proofs showing that the transformations preserve security, the theoretical underpinnings of symmetric-to-asymmetric translations are less explored. Here are some things we can say.

First, the original proof of security is under a symmetric pairing assumption, and thus can no longer immediately apply since the construction and assumption are changing their algebraic settings. This would seem to require the identification of a new complexity assumption together with a new proof of security. In many examples, e.g., [120], the new assumption and proof are only minor deviations from the original ones, e.g., where the CDH assumption in $\mathbb{G}$ (given $[g, g^a, g^b]$, compute g^{ab}) is converted in a straight-forward manner to the co-CDH assumption in $(\mathbb{G}_1, \mathbb{G}_2)$ (given $[g_1, g_2,\ g_2^a]$, compute g_1^a). However, there could be cases where a major change is required to the proof of security. For instance, in some asymmetric groups it is not possible to hash into $\mathbb{G}_2$, but in these groups there exists an isomorphism from $\mathbb{G}_2$ to $\mathbb{G}_1$. In other groups there is no such isomorphism, but it is possible to hash into $\mathbb{G}_2$. So if a scheme requires both for the security proof, that scheme may not be realizable in the asymmetric setting (see [163] for more).

In best practices today, a human first devises the new construction (based on their desired optimizations) and then the human works to identify the new assumption and proof.

Our current work automates the first step in this process, and hopefully gives the human more time to spend on the second step. In this sense, our automation is arguably faster, and no less secure than what is done by hand today.

However, a more satisfactory solution requires a deeper theoretical study of symmetric-to-asymmetric pairing translations, which we feel is an important open problem, but which falls outside the scope of the current work. What can one prove about the preservation of security in symmetric-to-asymmetric translations? Is it necessary to dig into the proof of security? Or could one prove security of the asymmetric scheme solely on the assumption of security of the symmetric one? Will this work the same for encryption, signatures and other protocols? Do the rules by which translations are done (by hand or AutoGroup) need to change based on these findings? These questions remain open.

6.4.4 Experimental Evaluation of AutoGroup

To determine the effectiveness of our automation, we evaluate several encryption and signature schemes on a variety of optimization combinations supported by our tool. We summarize the results of our experiments on encryption schemes in Figure 6.3 and signature schemes in Figure 6.5.

System Configuration. All of our benchmarks were executed on a 2.66GHz 6-core Intel Xeon Mac Pro with 10GB RAM running Mac OS X 10.8.3 using only a single core of the Intel processor. Our implementation utilizes the MIRACL library (v5.5.4), Charm v0.43 [11] in C++ due to the efficiency gains over Python code, and Z3 SMT solver

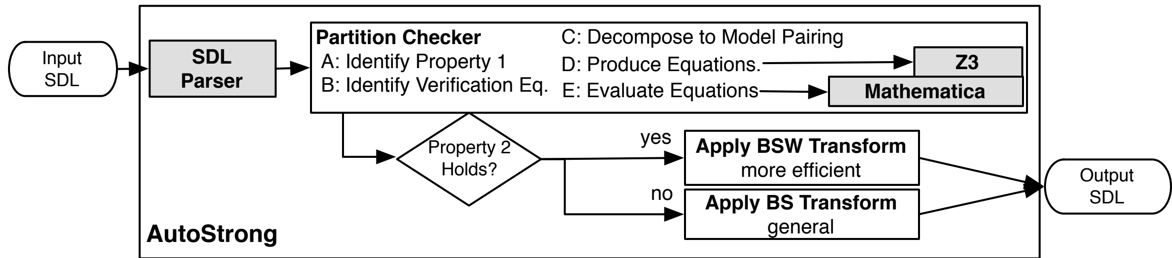


Figure 6.4: A high-level presentation of the AutoStrong tool, which uses external tools Z3, Mathematica and SDL Parser.

(v4.3.1). We based our implementations on the MIRACL library to fully compare each scheme's performance using symmetric and asymmetric curves at equivalent security levels.

Results. To demonstrate the soundness of AutoGroup on encryption and signature schemes, we compare algorithm running times, key and ciphertext/signature sizes between symmetric and asymmetric solutions. We tested AutoGroup on a variety of optimization combinations to extract different asymmetric solutions. In each test case, AutoGroup reports all the unique solutions, obtains the best solution for given user-specified constraints, and generates the executable code of the solution in a reasonable amount of time. AutoGroup execution time on each test case is reported in Figure 6.6, but does not include time for generating the C++ of the SDL output.

6.5 AutoStrong

In this section, we present and evaluate a tool, called AutoStrong, for automatically generating a strongly-unforgeable signature from an unforgeable signature scheme.

6.5.1 Background on Digital Signatures

A digital signature scheme is comprised of three algorithms: key generation, signing and verification. The classic (or "regular") security definition for signatures, as formulated by Goldwasser, Micali and Rivest [21], is called *existential unforgeability with respect to chosen message attacks*, wherein any p.p.t. adversary, given a public key and the ability to adaptively ask for a signature on any message of its choosing, should not be able to output a signature/message pair that passes the verification equation and yet where the message is "new" (was not queried for a signature), with non-negligible probability.

An, Dodis and Rabin [13] formulated *strong unforgeability* where the adversary should not only be unable to generate a signature on a "new" message, but also be unable to generate a different signature for an already signed message. Strongly-unforgeable signatures have many applications including building signcryption [13], chosen-ciphertext secure encryption systems [14, 58] and group signatures [15, 59].

Partitioned Signatures In 2006, Boneh, Shen and Waters [12] connected these two security notions, by providing a general transformation that converts any *partitioned* (defined below) existentially unforgeable signature into a strongly unforgeable one.

Definition 6.5.1 (Partitioned Signature [12])**.** *A signature scheme is* partitioned *if it satisfies two properties for all key pairs* (pk, sk)*:*

- **Property 1:** *The signing algorithm can be broken into two deterministic algorithms* F_1 *and* F_2 *so that a signature on a message* m *using secret key* sk *is computed as follows:*

  1. *Select a random* r *from a suitable randomness space.*
  2. *Set* $\sigma_1 = F_1(m, r, sk)$ *and* $\sigma_2 = F_2(r, sk)$*.*
  3. *Output the signature* (σ_1, σ_2)*.*

- **Property 2:** *Given* m *and* σ_2*, there is at most one* σ_1 *such that* (σ_1, σ_2) *verifies as a valid signature on* m *under* pk*.*

As one example of a partitioned scheme, Boneh et al. partition DSS [175] as follows, where x is the secret key:

$$F_1(m, r, x) = r^{-1}(m + xF_2(r, x)) \mod q$$

$$F_2(r, x) = (g^r \mod p) \mod q$$

Our empirical evidence shows that many discrete-log and pairing-based signatures in the literature are partitioned. Interestingly, some prominent prior works [39, 168] claimed that there were "few" examples of partitioned schemes "beyond Waters [67]", even though our automation discovered several examples existing prior to the publication of these works. We conjecture that it is not always easy for a human to detect a partition.

Chameleon Hashes The BSW transform uses a *chameleon hash* [176] function, which is characterized by the nonstandard property of being collision-resistant for the signer but collision tractable for the recipient. The chameleon hash is created by establishing public parameters and a secret trapdoor. The hash itself takes as input a message m and an auxiliary value s. There is an efficient algorithm that on input the trapdoor, any pair (m_1, s_1) and any additional message m_2, finds a value s_2 such that $\text{ChamHash}(m_1, s_1) = \text{ChamHash}(m_2, s_2)$.

Boneh et al. [12] employ a specific hash function based on the hardness of finding discrete logarithms.[4] Since pairing groups also require the DL problem to be hard, this chameleon hash does not add any new complexity assumptions. It works as follows in $\mathbb{G}$, where g generates $\mathbb{G}$ of order p. To setup, choose a random trapdoor $t \in \mathbb{Z}_p{}^*$ and compute $h = g^t$. The public parameters include the description of $\mathbb{G}$ together with g and h. The trapdoor t is kept secret. To hash on input $(m, s) \in \mathbb{Z}_p{}^2$, compute

$$\text{ChamHash}(m, s) = g^m h^s.$$

Later, given any pair m, s and any message m', anyone with the trapdoor can compute a consistent value $s' \in \mathbb{Z}_p$ as

$$s' = (m - m')/t + s$$

such that $\text{ChamHash}(m, s) = \text{ChamHash}(m', s')$.

The BSW Transformation The transformation [12] is efficient and works as follows. Let $\Pi_p = (\mathsf{Gen}_p, \mathsf{Sign}_p, \mathsf{Verify}_p)$ be a partitioned signature, where the signing algorithm is

[4] Indeed, we observe that substituting an arbitrary chameleon hash could break the transformation. Suppose $H(m, s)$ ignores the last bit of s (it is easy to construct such a hash assuming chameleon hashes exist.) Then the BSW transformation using this hash would result in a signature of the form (σ_1, σ_2, s), which is clearly not strongly unforgeable, since the last bit can be flipped.

partitioned using functions F_1 and F_2. Suppose the randomness for Sign_p is picked from some set R. Let $||$ denote concatenation. BSW constructs a new scheme Π as:

$\mathsf{Gen}(1^\lambda)$**:** Select a group $\mathbb{G}$ with generator g of prime order p (with λ bits). Select a random $t \in \mathbb{Z}_p$ and compute $h = g^t$. Select a collision-resistant hash function $H_{cr} : \{0,1\}^* \rightarrow \mathbb{Z}_p$. Run $\mathsf{Gen}_p(1^\lambda)$ to obtain a key pair (pk_p, sk_p). Set the keys for the new system as $pk = (pk_p, H_{cr}, \mathbb{G}, g, h, p)$ and $sk = (pk, sk_p, t)$.

$\mathsf{Sign}(sk, m)$**:** A signature on m is generated as follows:

1. Select a random $s \in \mathbb{Z}_p$ and a random $r \in R$.
2. Set $\sigma_2 = F_2(r, sk_p)$.
3. Compute $v = H_{cr}(m||\sigma_2)$.
4. Compute the chameleon hash $m' = g^v h^s$.
5. Compute $\sigma_1 = F_1(m', r, sk_p)$ and output the signature $\sigma = (\sigma_1, \sigma_2, s)$.

$\mathsf{Verify}(pk, m, \sigma)$**:** A signature $\sigma = (\sigma_1, \sigma_2, s)$ on a message m is verified as follows:

1. Compute $v = H_{cr}(m||\sigma_2)$.
2. Compute the chameleon hash $m' = g^v h^s$.
3. Output the result of $\mathsf{Verify}_p(pk_p, m', (\sigma_1, \sigma_2))$.

Theorem 6.5.2 (Security of BSW Transform [12])**.** *The signature scheme* $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *is strongly existentially unforgeable assuming the underlying scheme* $\Pi_p = (\mathsf{Gen}_p, \mathsf{Sign}_p, \mathsf{Verify}_p)$

is existentially unforgeable, H_{cr} is a collision-resistant hash function and the discrete logarithm assumption holds in $\mathbb{G}$.

The Bellare-Shoup Transformation The BSW transformation [12], which only works for partitioned signatures, sparked significant research interest into finding a general transformation for *any* existentially unforgeable signature scheme. Various solutions were presented in [39, 164–168], as well as an observation in [39] that an *inefficient* transformation was implicit in [177].

We follow the work of Bellare and Shoup [39, 168], which is less efficient than BSW and, for our case, requires a stronger complexity assumption, but works on any signature. Their approach uses *two-tier* signatures, which are "weaker" than regular signatures as hybrids of regular and one-time schemes. In a two-tier scheme, a signer has a primary key pair and, each time it wants to sign, it generates a fresh secondary key pair and produces a signature as a function of the both secret keys and the message. Both public keys are required to verify the signature. Bellare and Shoup transform any regular signature scheme by signing the signature from this scheme with a strongly unforgeable two-tier scheme. They also show how to realize a strongly unforgeable two-tier signature scheme by applying the Fiat-Shamir [178] transformation to the Schnorr identification protocol [179], which requires a one-more discrete logarithm-type assumption.

The BS transformation works as follows. Let $\Pi_r = (\mathsf{Gen}_r, \mathsf{Sign}_r, \mathsf{Verify}_r)$ be a regular signature scheme and let $\Pi_t = (\mathsf{PGen}_t, \mathsf{SGen}_t, \mathsf{Sign}_t, \mathsf{Verify}_t)$ be a two-tiered strongly unforgeable scheme. A new signature scheme Π is constructed as:

$\mathsf{Gen}(1^\lambda)$**:** Run $\mathsf{Gen}_r(1^\lambda) \rightarrow (pk_r, sk_r)$ and $\mathsf{PGen}_t(1^\lambda) \rightarrow (ppk, psk)$. Output the pair PK = (pk_r, ppk) and SK = (sk_r, psk).

$\mathsf{Sign}(\mathrm{SK}, m)$**:** A signature on m is generated as follows:

1. Parse SK as (sk_r, psk).
2. Run $\mathsf{SGen}_t(1^\lambda) \rightarrow (spk, ssk)$.
3. Sign the message and secondary key as $\sigma_1 \leftarrow \mathsf{Sign}_r(sk_r, (spk \| m))$.
4. Sign the first signature as $\sigma_2 \leftarrow \mathsf{Sign}_t(psk, ssk, \sigma_1)$.
5. Output the signature $\sigma = (\sigma_1, \sigma_2, spk)$.

$\mathsf{Verify}(\mathrm{PK}, m, \sigma)$**:** A signature $\sigma = (\sigma_1, \sigma_2, spk)$ on a message m is verified as follows:

1. Parse PK as (pk_r, ppk).
2. If $\mathsf{Verify}_r(pk_r, (spk \| m), \sigma_1) = 0$, then return 0.
3. If $\mathsf{Verify}_t(ppk, spk, \sigma_1, \sigma_2)$, then return 0.
4. Otherwise, return 1.

Theorem 6.5.3 (Security of BS Transformation [168])**.** *If the input scheme is existentially unforgeable, then the output signature is strongly existentially unforgeable assuming the strong unforgeability of the two-tier scheme.*

The Transformation used in AutoStrong For our purposes, we employ the following hybrid transformation combining BSW and Bellare-Shoup. On input a signature scheme, we automate the following procedure:

1. Identify a natural partition satisfying property 1 and test if it has property 2. (We allow false negatives, but not false positives. See Section 6.5.3.)

2. If a valid partition is found, apply the BSW transformation [12] (using SHA-256 and the DL-based chameleon hash above).

3. If a valid partition is not found, apply the Bellare-Shoup transformation [39, 168] (using the Schnorr Fiat-Shamir based two-tier scheme suggested in [168].)

4. Output the result.

The security of this transformation follows directly from the results of [12,168] as stated in Theorems 6.5.2 and 6.5.3. *The most challenging technical part is step one: determining if a scheme is partitioned.*

6.5.2 How AutoStrong Works

AutoStrong takes as input the SDL description of a digital signature scheme along with some metadata.[5] At a high-level, it runs the transformation described at the end of the last section, where the most challenging step is testing whether a scheme is *partitioned* according to Definition 6.5.1.

We now describe each step involved in testing that Properties 1 and 2 are satisfied and how we utilize Z3 and Mathematica to prove such properties, as illustrated in Figure 6.4.

[5] The user must specify the variables that denote message, signature, key material in a configuration file.

Signature		Time		Approx. Size		Num.
	Security	Sign•	Verify•	Public Key*	Signature	Solutions
CL04 [18, §3.1] Symmetric (SS1536)	EU-CMA	169.8 ms	316.6 ms	3072 bits	4608 bits	
Symmetric (SS1536)	SU-CMA	192.0 ms	387.8 ms	4608 bits	6144 bits	
Asymmetric (BN256) [Min. SIG]	SU-CMA	3.4 ms	56.8 ms	2048 bits	1024 bits	2
BB Short [66, §3] Symmetric (SS1536)	EU-CMA	21.5 ms	102.1 ms	7680 bits	3072 bits	
Symmetric (SS1536)	SU-CMA	62.8 ms	142.8 ms	9216 bits	4608 bits	
Asymmetric (BN256) [Min. PK]	SU-CMA	5.0 ms	18.3 ms	3840 bits	1536 bits	2
WATERS05 [67, §4] Symmetric (SS1536)	EU-CMA	47.9 ms	195.2 ms	4608 bits†	3072 bits	
Symmetric (SS1536)	SU-CMA	88.7 ms	236.4 ms	6144 bits†	4608 bits	
Asymmetric (BN256) [Min. SIG]	SU-CMA	6.5 ms	62.9 ms	2560 bits†	768 bits	8
WATERS09 [180, §6.1] Symmetric (SS1536)	WU-CMA	258.5 ms	896.8 ms	23040 bits	13824 bits	
Asymmetric (BN256) [Min. PK/SIG]	WU-CMA	13.6 ms	129.2 ms	12544 bits	5376 bits	256
ACDKNO12 [26, §5.3] Symmetric (SS1536)	RMA	346.4 ms	1307 ms	23040 bits	12288 bits	
Asymmetric (BN256) [Min. PK/SIG/Exp]	RMA	23.3 ms	279.9 ms	3840 bits	8192 bits	1024

•Average time measured over 100 test runs and the standard deviation in all test runs were within ±1% of the average.
*Refers to the approximate size of public parameters used in verification.
†Estimates do not include the public parameters for the Water's hash.

Figure 6.5: We show the result of AutoGroup and AutoStrong on signature schemes. For CL, BB, and Waters (with length of identities, $\ell = 128$), we first apply AutoStrong to determine that the signature scheme is partitioned, then apply the BSW transform to obtain a strongly unforgeable signature in the symmetric setting. We then feed this as input to AutoGroup to realize an asymmetric variant under a given optimization. We also tested AutoStrong on the DSE signature and ACDK structure-preserving signature, even though these are not known to be existentially unforgeable. A partition was found for ACDK, but not DSE.

Identify Property 1. The first goal is to identify the variables in the signature that should be mapped to σ_1 or σ_2 according to Definition 6.5.1. We assume that the input signature scheme is existentially unforgeable.[6] Given this assumption, our objective is to identify the portions of the signature that are computed based on the message and designate that component as σ_1. All other variables in the signature that do not meet this criteria are designated as σ_2. We determine that we have designated the correct variables for property 1 if and only if the variable mapping satisfy property 2. We test only the most "natural" division for property 1, which could result in a false negative, but this won't impact the security, so our system allows it.

[6]We remark that we tested the partition checker for AutoStrong on schemes that are not existentially unforgeable to fully vet the checker (see Figure 6.5), but the resulting output in these cases may not be strongly unforgeable.

To illustrate each step, we will show how our tool identifies the partition in the CL signature scheme [18].

CL signatures [18]: Key generation consists of selecting a generator, $g \in \mathbb{G}$, then randomly sampling $x \in \mathbb{Z}_q$ and $y \in \mathbb{Z}_q$. It sets $sk = (x, y)$ and $pk = (g, X = g^x, Y = g^y)$. To sign a message $m \in \mathbb{Z}_q$, the signer samples a uniformly from $\mathbb{G}$ and computes the signature as:

$$\sigma = (a, b = a^y, c = a^{x+m \cdot x \cdot y}).$$

The verifier can check σ by ensuring that $e(a, Y) = e(g, b)$ and $e(X, a) \cdot e(X, b)^m = e(g, c)$.

Intuitively, our logic would identify that c is dependent on the message, therefore, identifying that $\sigma_1 = c$ and $\sigma_2 = (a, b)$ which satisfies the definition of property 1. The next challenge is to determine whether property 2 holds given our identified mapping for σ_1 and σ_2.

Prove Property 2. Proving that a scheme satisfies this property requires the ability to abstractly evaluate the verification equations on the input variables. We require this ability to automatically prove that there exists at most one σ_1 which verifies under a fixed σ_2, m and pk for all possible inputs. To this end, the partition checker determines whether a σ_1' exists such that $\sigma_1' \neq \sigma_1$ and is a valid signature over the fixed variables. Finding such a σ_1' means the signature is not partitioned. The checker determines whether it can find a solution or if it can determine that *no* such solution exists. If no solutions exist, then the signature is indeed partitioned. Stated more precisely, does there exist a $\sigma_1' \neq \sigma_1$ such that the following condition holds:

$\mathsf{Verify}(pk, m, (\sigma_1, \sigma_2)) = 1 \wedge \mathsf{Verify}(pk, m, (\sigma_1', \sigma_2)) = 1$

At a high-level, our goal is to evaluate the pairing-based verification algorithms in a way that allows us to find a contradiction to the aforementioned condition. Recall that the bilinearity property of pairings states that $e(g^a, g^b) = e(g, g)^{ab}$ holds for all $a, b \in \mathbb{Z}_q$ where $g \in \mathbb{G}$. We observe that pairings can be modeled as an abstract function that performs multiplication in the exponent. Because the rules of multiplication and addition hold in the exponent, we can abstractly reduce pairings to basic integer arithmetic.

To accomplish this, we leverage Z3 to model the bilinearity of pairings so that it is possible to automatically evaluate them. Our partition checker relies on Z3's uninterpreted functions and universal quantifiers to reduce pairing product equations to simpler equations over the exponents. However, this reduction alone is not sufficient to completely evaluate the verification equations as required for detecting a partitioned signature. To satisfy the property 2 condition, we also need a way to evaluate these equations on all possible inputs. Z3 was less suited for this task and instead, we employ the Mathematica scripting framework to evaluate such equations. Our solution consists of five steps:

Step 1: Decompose Verification Equations. To model pairings using an SMT solver, we encode the verification equations into a form that the solver can interpret. The first phase extracts the verification equations in SDL, then decomposes the equations in terms of the generators and exponents used. We leverage recent term rewriting extensions introduced in the SDL Parser by Akinyele et al. [17]. Their techniques allow us to keep track of how variables are computed in terms of the generators and exponents. With knowledge of how

each variable is computed, we are able to fully decompose each equation in an automated fashion.

Our technique for modeling pairings in Z3 requires that decomposition of verification equations be guided by a few rules. First, generators must be rewritten in terms of some base generator, g, if the scheme is specified in the symmetric setting.[7] For example, the random generator $a \in \mathbb{G}$ chosen in CL would be represented as $g^{a'}$ for $a' \in \mathbb{Z}_q$. Second, hashing statements of the form $v = H(m)$ where $v \in \mathbb{G}$ are rewritten as $g^{v'}$ for some $v' \in \mathbb{Z}_q$.[8] Third, we do not decompose any variable designated as σ_1 for the purposes of determining whether a signature is partitioned. The intuition is that since σ_1' variables are adversarially controlled we also treat σ_1 as a black box. Finally, whenever we encounter signatures that compute a product over a list of elements – as in the case of the Waters hash, for example [67] – we require the user to provide an upper bound on the number of elements in this list (if known) so that we can "unroll" the product calculation and further apply our rules. When all the above reduction rules are automatically applied to the CL signature, we obtain the following equations:

$e(a, Y) = e(g, b)$ becomes $e(g^{a'}, g^y) = e(g, (g^{a'})^y)$

$e(X, a) \cdot e(X, b)^m = e(g, c)$ becomes

$$e(g^x, g^{a'}) \cdot e(g^x, (g^{a'})^y)^m = e(g, g^{c'})$$

Note that c' denotes the σ_1 for CL and is a free variable. All other variables that comprise

[7] The same would apply for asymmetric pairings except that we would specify $\mathbb{G}_1$ generators in terms of a base generator g_1 and $\mathbb{G}_2$ in terms of g_2.

[8] Note that this term re-writing is used only to determine whether a solution exists. The actual variables a' and v' would not (necessarily) be known in the real protocol.

m, pk, and σ_2 are fixed.

Step 2: Encode Rules for Evaluating Pairings. Once we have decomposed the verification equation as shown above, the next step is to encode the equations in terms that Z3 can understand. After the pairing equations are rewritten entirely using the base generator, we can model the behavior of pairings by simply focusing on the exponents. To capture the bilinearity of pairings, we rely on two features in Z3: uninterpreted functions and universal quantifiers. As mentioned earlier, uninterpreted functions enable one to abstractly model a function's behavior. Our model of a pairing is an uninterpreted function, E, that takes two integer variables and has a few mathematical properties. First, we define the multiplication rule as $\forall s,t : E(s,t) = s \cdot t$. Second, we define the addition rule as $\forall s,t,u : E(s+t,u) = s \cdot u + t \cdot u$.[9] Third, we adhere to the multiplicative notation in SDL and convert pairing products defined in terms of multiplication to addition and division to subtraction.

These rules are straightforward and sufficient for evaluating pairings. Moreover, by defining exponents in terms of integers, Z3 can apply all the built-in simplification rules for multiplication and addition. As a result, the solver uses these rules to reduce any pairing-based verification equation into a simpler integer equation.

To automatically encode the equations, we first simplify the decomposed pairing equation as much as possible using previous techniques [17]. Then, we convert each pairing to the modeled pairing function, E and remove the base generators. Upon simplifying and

[9] Similarly, $E(s,t+u) = s \cdot t + s \cdot u$

encoding the decomposed CL equations, we obtain the following:

$e(g^{a'}, g^y) = e(g, (g^{a'})^y)$ becomes $E(a', y) = E(1, a' \cdot y)$

$e(g^x, g^{a'}) \cdot e(g^x, (g^{a'})^y)^m = e(g, g^{c'})$ becomes

$$E(x, a') + E(x \cdot m, a' \cdot y) = E(1, c')$$

Step 3: Execute SMT Solver. After encoding the pairing functions in terms of E, the next step is to employ the solver to evaluate it. We first specify our rules in the SMT solver then evaluate these rules on each input equation. The result is a simplified integer equation representation of the verification algorithm. For the above CL formulas, the solver determines that the first equation is *true* for all possible inputs because a' and y are fixed variables. For the second equation, the solver produces: $a' \cdot x + a' \cdot x \cdot m \cdot y = c'$.

Step 4: Evaluate equations. At this point, we have obtained the integer equation version of the verification equation; we can now concretely express the conditions for property 2. That is,

$$c' \neq c'' \wedge a' \cdot x + a' \cdot x \cdot m \cdot y = c' \wedge a' \cdot x + a' \cdot x \cdot m \cdot y = c''$$

Process	BB-IBE	Gentry	Waters09-Enc	BGW	CL	BB Short Sig	Waters05	Waters09-Sig	ACDKNO
AutoGroup	0.33s	0.34s	4.30s	0.55s	0.34s	0.31s	0.54s	4.16s	17.65s
AutoStrong	-	-	-	-	0.28s	0.27s	0.37s	3.99s	1.23s

Figure 6.6: Running time required by the AutoGroup and AutoStrong routines to process the schemes discussed in this work (averaged over 10 test runs). The running time for AutoGroup includes the execution time of the Z3 SMT solver. The running time for AutoStrong also includes Z3 and Mathematica and the application of the BSW transformation. In all cases, the standard deviation in the results were within ±3% of the average. For AutoGroup, running times are correlated with the number of unique solutions found and the minimization of the weighted function using Z3. AutoStrong running times are highly correlated with the complexity of the verification equations.

We use Mathematica to prove that no such c'' exists assuming the verification condition is correct via the Mathematica Script API. In particular, we utilize the *FindInstance* function to mathematically find proof over non-zero real numbers then subsequently try finding a solution over integers. If *no* such solution exists, the *FindInstance* will return such a statement and the result is interpreted as an indicator that the signature is partitionable. Otherwise, the signature may not be partitionable.

During this step, we make an explicit assumption that the verification condition is mathematically correct. Suppose that this was not the case. In this scenario, our technique would also determine that it is not possible to find a σ_1' such that $\sigma_1' \neq \sigma_1$ and verifies over fixed variables. In reality, however, no σ_1 and σ_2 pair can produce a valid signature because the verification equation does not hold for *any* input. To limit the possibility of such scenarios, our partition checker offers a sanity check on the correctness of the input verification equations.

By relaxing the rule for decomposing the variables that are designated as σ_1 in Step 1, we can evaluate the verification equation over all inputs using Mathematica. For the CL signature, a full decomposition would produce the following equation in the exponent:

$$a' \cdot x + a' \cdot x \cdot m \cdot y = a' \cdot (x + x \cdot m \cdot y)$$

It is sufficient to leverage the *Simplify* function within Mathematica to evaluate that this holds for all possible inputs. Since Mathematica has built-in techniques for solving equations of this sort, it becomes trivial to show that the above equation is correct in all cases (due to the law of distribution). We subsequently inform the user on the output of this sanity

check, which is useful for determining the correctness of SDL signature descriptions.

Step 5: Apply Transformation. Once the partition checker determines whether the signature is partitioned or not, we apply the efficient BSW transform if deemed partitioned or the less-efficient BS transform if not as described in Section 6.5.1.

6.5.3 Security Analysis of AutoStrong

The theoretical security of the unforgeable-to-strongly-unforgeable transformations that we use in AutoStrong were previously established in [12, 39, 168], as discussed in Section 6.5.1.[10] The security of the BSW transform only holds, however, if the input scheme is partitioned. Our partition test allows false negatives, but not false positives. That is, our algorithm may fail to identify a scheme as partitioned even though it is, which results in a less efficient final scheme, but it will not falsely identify a scheme as partitioned when it is not, which would result in a security failure. To see why this claim holds, consider that the partition tester guesses a partition, Z3 interprets the verification equation as a system of equations, and then Mathematica fixes the variables on one partition side and asks how many solutions there are for the free variables on the other side. If 0 or 1 are found, then the scheme meets the partitioned definition. If more than 1 is found, then it is not partitioned. If there is no answer (program crash or times out), then we consider it not partitioned. Thus, false negatives can occur, but not false positives (in theory). Proving that there are

[10]Perfect correctness is assumed in these transformations. All schemes tested have perfect correctness, except the Waters DSE signatures [53]. With a negligible probability, the verification algorithm of this scheme will reject an honestly-generated signature. After applying the BS transformation to the DSE scheme, this negligible error probability is carried over in the verification of the strongly-secure scheme.

no software or hardware errors in AutoStrong, Z3, Mathematica or the underlying software and hardware on which they run is outside the scope of this work. We did experimentally verify AutoStrong's outputs and no errors were found.

6.5.4 Experimental Evaluation of AutoStrong

In 2008 [168], Bellare and Shoup remarked that "unfortunately, there seem to be hardly any [partitioned signature] schemes". Interestingly, our experimental results show that there are in fact many partitioned schemes, including a substantial number invented prior to 2008. We evaluated AutoStrong by testing it on a collection of signatures, including Camenisch-Lysyanskaya [18], short Boneh-Boyen [66], Waters 2005 [67], Waters Dual-System (DSE) signature [53], and a structure-preserving scheme of Abe *et al.* [26].

Of the above signatures, all but one – the Waters DSE signature – were successfully partitioned. We do not know whether the Waters DSE signature can be partitioned, although we suspect that the "randomness freedom" in the dual-system structure may inherently be at odds with the uniqueness property of the partitioned test. Although the Abe *et al.* scheme is partitioned, applying either the BSW or BS transformations destroys its structure-preserving property. An interesting open problem would be to refine the BSW or BS transformations to preserve the structured property. Figure 6.6 shows the time that it took our tool to identify the partitioning and output the revised signature equations. Figure 6.5 illustrates the performance and size of the resulting signatures, when evaluated on two different types of curve (using AutoGroup to calculate the group assignments).

6.6 Challenges and Open Problems

We explored two challenging new tasks in cryptographic automation. First, we presented a tool, AutoGroup, for automatically translating a symmetric pairing scheme into an asymmetric pairing scheme. The tool allows the user to choose from a variety of different optimization options. Second, we presented a tool, AutoStrong, for automatically altering a digital signature scheme to achieve *strong unforgeability* [13]. The tool automatically tests whether a scheme is "partitioned" according to a notion of Boneh et al. [12] and then applies a highly-efficient transformation if it is partitioned or a more general transformation otherwise. To perform some of these complex tasks, we integrated Microsoft's Z3 SMT Solver and Mathematica into our tools. Our performance measurements indicated that these standard cryptographic design tasks can be quickly, accurately and cost-effectively performed in an automated fashion.

We look onward to exciting problems left open by this work. Which other design tasks are naturally well suited for SMT solvers? Furthermore, can verification tools such as EasyCrypt [40] or CryptoVerif [41] be integrated into our automations to provide mechanized verification of the transformations? What techniques are required to automate the verification of such designs? Can they be generalized?

Chapter 7

Summary

We have presented the design and implementation of an extensible architecture to demonstrate the automation of certain general transformations from the literature. We showed how cryptographic primitives can be represented in our domain-specific language to facilitate automation and how tools can be developed to carry out transformation tasks. To illustrate this, we showed how this abstract language can be turned into working implementations using Charm. With this in place, we presented three case studies of transformations and how they can be safely, accurately and efficiently automated using SMT solvers to aid in some crucial aspects of the design. Finally, we discussed limitations in automating certain transformations and provided a security analysis for each of the tools that implement the transformations.

Appendix A

Additional Material

A.1 Scheme Examples In Charm

Scheme	Type	Setting	Comp. Model	Lines
Adapters				
CHK04 [14], BCHK05 [181]	IBE-to-PKE	-	-	23, 63
IBE-to-Signature [28]	Signature	-	-	24
Hybrid ABE	Hybrid ABE	-	-	27
Hybrid DABE	Hybrid DABE	-	-	28
Hybrid KPABE	Hybrid KPABE	-	-	26
Hybrid IBE [14]	Hybrid IBE	-	-	27
IBE Identity Hash	IBE	-	-	35
Hybrid PKE	Hybrid PKE	-	-	30
Miscellaneous				
GS07 [27]	Commitment	Pairing	CRS	28
Pedersen [182]	Commitment	EC/Integer	Standard	16
AdM05 [183]	Cham Hash	Integer	ROM	24
RSA HW09 [116]	Cham Hash	Integer	Standard	29
VRF [19]	VRF	Pairing	Standard	47
Protocols				
Schnorr91 [115]	Zero-Knowledge proof	EC/Integer	Standard	53
CNS07 [122]	Oblivious Transfer	Pairing	Standard	147

Table A.1: Another listing of the cryptographic schemes we implemented. “Code Lines” indicates the number of lines of Python code used to implement the scheme (excluding comments and whitespace), and does not include the framework itself. ROM indicates that a scheme is secure in the Random Oracle Model. CRS indicates that a scheme is secure in the Common Reference String Model. A “-” indicates a generic transform (adapter). * indicates a choice made for efficiency reasons.

Using BSW07 Scheme in C

```
# variable declarations
Charm_t *group,*cpabe,*hyabe,*keyTupl,*recmsg;
Charm_t *pkDict,*mskDict,*skDict,*ctDict,*ctBlob;
char *msg,*policy,*attrlist;

# setup Charm environment
InitializeCharm();
#initialize group with super singular curve
# and 512-bits for base field
group = InitPairingGroup(module, "SS512");
# initialize the scheme
cpabe = InitScheme("abenc_bsw07",
        "CPabe_BSW07",group);
# call to initialize adapters
hybae = InitAdapter("abenc_adapt_hybrid",
        "HybridABEnc",cpabe,group);
# no arguments to setup
keyTupl = CallMethod(hyabe, "setup", "");
#extract master public & private keys
pkDict = GetIndex(keyTupl, 0);
mskDict = GetIndex(keyTupl, 1);
# call keygen
attrlist = "[SALES, IT]";
skDict = CallMethod(hyabe,"keygen","%O%O%A",
         pkDict,mskDict,attrlist);
# call encrypt
msg = "this is a test message";
policy = "(CORPORATE and (SALES or IT))";
ctDict = CallMethod(hyabe,"encrypt",%O%b%s",
         pkDict,msg,policy);
# serialize object into base-64 string
ctBlob = objectToBytes(ctDict, group);
# call decrypt
recmsg = CallMethod(hyabe,"decrypt","%O%O%O",
         pkDict,skDict,ctDict);
# . . . free Charm_t variables . . .
# tear down the Charm environment
CleanupCharm();
```

Figure A.1: A working example of how the API is utilized in a C application to embed a hybrid encryption adapter (see Figure A.2b) for any CP-ABE scheme such as the BSW07 [29] scheme. We provide several high-level functions that simplify using Charm schemes. In particular, the CallMethod() encapsulates several types of arguments to Python such as: %O for Charm objects, %s for ASCII strings, %A to convert into a Python list, and %b to a binary object.

IBE-to-Sig Adapter

```
def __init__(self, scheme, groupObj):
    PKSig.__init__(self)
    global ibe, group
    condition = [('secDef',IND_ID_CPA),('scheme','IBenc'),
                ('messageSpace',GT)]
    if PKSig.checkProperty(self, scheme, condition):
       # inherit properties of scheme & update definitions
       PKSig.updateProperty(self, scheme, secDef=EU_CMA,
          id=str, secModel=ROM)
       ibe = scheme; group = groupObj
```

```
def keygen(self, secparam=None):
    (mpk, msk) = ibe.setup(secparam)
    return (mpk, msk)

def sign(self, sk, m):
    return ibe.extract(sk, str(m))

def verify(self, pk, m, sig):
    if hasattr(ibe, 'verify'):
       result = ibe.verify(pk, m sig)
       if result == False: return False
    new_m = group.random(GT)
    C = ibe.encrypt(pk, sig['IDstr'], new_m)
    if ibe.decrypt(sig, C) == new_m:
       return True
    else:
       return False
```

(a) IBE-to-Sig Adapter

Hybrid-Enc-ABE Adapter

```
class HybridABEnc(ABEnc):
    def __init__(self, scheme, groupObj):
        ABEnc.__init__(self)
        global abenc, group
        # ... verify scheme properties ...
        abenc = scheme
        group = groupObj

    def setup(self):
        return abenc.setup()

    def keygen(self, pk, mk, object):
        return abenc.keygen(pk, mk, object)
```

```
    def encrypt(self, pk, M, object):
        key = group.random(GT)
        c1 = abenc.encrypt(pk, key, object)
        # init a symmetric enc scheme from this key
        cipher = AuthCryptoAbstraction(sha1(key))
        c2 = cipher.encrypt(M)
        return { 'c1':c1, 'c2':c2 }

    def decrypt(self, pk, sk, ct):
        c1, c2 = ct['c1'], ct['c2']
        key = abenc.decrypt(pk, sk, c1)
        cipher = AuthCryptoAbstraction(sha1(key))
        return cipher.decrypt(c2)
```

(b) Hybrid Enc Adapter

Figure A.2: Adapters in Charm. (a). The entire IBE to signature adapter scheme [28]. (b) A hybrid encryptor for ABE schemes in Charm.

CS98 Keygen Description	Charm Implementation
Keygen. The key generation algorithm runs as follows. Random elements $g_1, g_2 \in G$ are chosen, and random elements $x_1, x_2, y_1, y_2, z \in Z_q$ are also chosen. Next, group elements $c = g_1^{x_1} g_2^{x_2}, d = g_1^{y_1} g_2^{y_2}, h = g_1^{z}$ are computed. Next, a hash function H is chosen from the family of universal one-way hash functions. The public key is (g_1, g_2, c, d, h, H), and the private key is (x_1, x_2, y_1, y_2, z)	see code below

```
def keygen(self, secparam):
    # code for checking group setting
    g1, g2 = group.random(G, 2)
    x1, x2, y1, y2, z = group.random(ZR, 5)
    c = (g1 ** x1) * (g2 ** x2)
    d = (g1 ** y1) * (g2 ** y2)
    h = (g1 ** z)
    pk = { 'g1':g1, 'g2':g2, 'c':c, 'd':d, 'h':h }
    sk = { 'x1':x1, 'x2':x2, 'y1':y1, 'y2':y2, 'z':z }
    return (pk, sk)
```

Figure A.3: Keygen in the Cramer-Shoup scheme [106]. We exclude group parameter generation.

CL04 Scheme Description

Keygen. The key generation algorithm runs the $Setup$ algorithm in order to generate (q, G, g, e) It then chooses $x \leftarrow Z_q$ and $y \leftarrow Z_q$ and sets $sk = (x, y)$, $pk = (q, G, g, e, X = g^x, Y = g^y)$

Sign. On input message m, secret key $sk = (x, y)$, and public key $pk = (q, G, g, e, X, Y)$, choose a random $a \in G$, and output the signature $\sigma = (a, a^y, a^{x+mxy})$

Verify. On input $pk = (q, G, g, e, X, Y)$, message m, and purported signature $\sigma = (a, b, c)$, check that the following holds:

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

```
def keygen(self):
    g = group.random(G1)
    x, y = group.random(ZR, 2)
    sk = { 'x':x, 'y':y }
    pk = { 'X':g ** x, 'Y':g ** y, 'g':g }
    return (pk, sk)

def sign(self, sk, M):
    (x, y) = sk['x'], sk['y']
    a = group.random(G2)
    m = group.hash(M, ZR)
    sig = { 'a':a, 'b':a ** y, 'c':a ** (x + (m * x * y)) }
    return sig

def verify(self, pk, M, sig):
    (a, b, c) = sig['a'], sig['b'], sig['c']
    m = group.hash(M, ZR)
    if pair(a,pk['Y']) == pair(pk['g'],b) and
        (pair(pk['X'],a) * (pair(pk['X'],b) ** m)) == pair(pk['g'],c):
       return True
    return False
```

Figure A.4: CL signatures [73] are a useful building block for anonymous credential systems. We provide a full scheme description and Charm code, but exclude group parameter generation.

A.2 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 the 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 describe 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 [42], CL04 [18], HW [135], and Waters09 [57]:

```
name := bls
# expected batch size per time
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 := None
 g := random(G2)
 x := random(ZR)
 pk := g^x
 sk := x
output := list{pk, sk, g}
END :: func:keygen

BEGIN :: func:sign
input := list{sk, M}
 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)}
```

The CL04 full SDL description:

```
name := cl04
N := 100
setting := asymmetric

BEGIN :: types
 M := Str
 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 := expand{x, y}
 pk := expand{X, Y}
output := list{pk, sk}
END :: func:keygen

BEGIN :: func:sign
input := list{sk, M}
 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)}
```

The HW full SDL description:

```
name := hw
N := 100
setting := asymmetric

BEGIN :: types
 m := Str
 n := ZR
 i := Int
END :: types

BEGIN :: func:setup
input := list{None}
 g1 := random(G1)
 g2 := random(G2)
output := list{g1, g2}
END :: func:setup

BEGIN :: func:keygen
input := list{g1, g2}
 a := random(ZR)
 A := g2^a
 u := random(G1)
 v := random(G1)
 d := random(G1)
```

```
 U := e(u, A)
 V := e(v, A)
 D := e(d, A)
 w := random(ZR)
 z := random(ZR)
 h := random(ZR)
 w1 := g1 ^ w
 w2 := g2 ^ w
 z1 := g1 ^ z
 z2 := g2 ^ z
 h1 := g1 ^ h
 h2 := g2 ^ h
 i := 0
 pk := list{U, V, D}
 spk := list{g1, w1, z1, h1, u, v, d}
 vpk := list{g2, w2, z2, h2}
 sk := a
output := list{i, pk, sk}
END :: func:keygen

BEGIN :: func:sign
input := list{spk, sk, i, m}
 spk := expand{g1, w1, z1, h1, u, v, d}
 i := i + 1
 M := H(m, ZR)
 r := random(ZR)
 t := random(ZR)
 n := ceillog(2, i)
 sig1:= (((u^M)*(v^r)*d)^sk)*((w1^n)*(z1^i)*h1)^t
 sig2 := g1 ^ t
 sig := list{sig1, sig2, r, i}
output := sig
END :: func:sign

BEGIN :: func:verify
input := list{pk, g2, w2, z2, h2, m, sig}
 pk := expand{U, V, D}
 sig := expand{sig1, sig2, r, i}
 M := H(m, ZR)
 n := ceillog(2, i)
BEGIN :: if
 if {e(sig1,g2) == ((U^M) * (V^r) * D * e(sig2,((w2^n)*((z2^i)*h2))))}
```

```
    output := True
 else
    output := False
END :: if
END :: func:verify

# Batcher input
constant := list{g2, w2, z2, h2}
public := pk
signature := sig
message := M

BEGIN :: precompute
 M := H(m, ZR)
 n := ceillog(2, i)
END :: precompute

# different signer
BEGIN :: count
message_count := N
public_count := N
signature_count := N
END :: count

verify := {e(sig1,g2) == ((U^M)*(V^r)*D*e(sig2,((w2^n)*((z2^i)*h2))))}
```

The Waters09 full SDL description:

```
name := waters09
N := 100
setting := asymmetric

BEGIN :: types
 m := Str
END :: types

BEGIN :: func:keygen
input := None
 g1 := random(G1)
 g2 := random(G2)
 a1 := random(ZR)
```

```
 a2 := random(ZR)
 b := random(ZR)
 alpha := random(ZR)
 wExp := random(ZR)
 hExp := random(ZR)
 vExp := random(ZR)
 v1Exp := random(ZR)
 v2Exp := random(ZR)
 uExp := random(ZR)
 vG2 := g2 ^ vExp
 v1G2 := g2 ^ v1Exp
 v2G2 := g2 ^ v2Exp
 wG1 := g1 ^ wExp
 hG1 := g1 ^ hExp
 w := g2 ^ wExp
 h := g2 ^ hExp
 uG1 := g1 ^ uExp
 u := g2 ^ uExp
 tau1 := vG2 * (v1G2 ^ a1)
 tau2 := vG2 * (v2G2 ^ a2)
 g1b := g1 ^ b
 g1a1 := g1 ^ a1
 g1a2 := g1 ^ a2
 g1ba1 := g1 ^ (b * a1)
 g1ba2 := g1 ^ (b * a2)
 tau1b := tau1 ^ b
 tau2b := tau2 ^ b
 A := (e(g1, g2)) ^ (alpha * a1 * b)
 g2AlphaA1 := g2 ^ (alpha * a1)
 g2b := g2 ^ b

 pk := list{g1, g2, g1b, g1a1, g1a2, g1ba1, g1ba2, tau1, tau2,
  tau1b, tau2b, uG1, u, wG1, hG1, w, h, A}
 sk := list{g2AlphaA1, g2b, vG2, v1G2, v2G2, alpha}
output := list{pk, sk}
END :: func:keygen

BEGIN :: func:sign
input := list{pk, sk, m}
 pk := expand{g1, g2, g1b, g1a1, g1a2, g1ba1, g1ba2, tau1, tau2,
  tau1b, tau2b, uG1, u, wG1, hG1, w, h, A}
 sk := expand{g2AlphaA1, g2b, vG2, v1G2, v2G2, alpha}
```

```
 r1 := random(ZR)
 r2 := random(ZR)
 z1 := random(ZR)
 z2 := random(ZR)
 tagk := random(ZR)
 r := r1 + r2
 M := H(m, ZR)
 S1 := g2AlphaA1 * (vG2 ^ r)
 S2 := (g2 ^ -alpha) * (v1G2 ^ r) * (g2 ^ z1)
 S3 := g2b ^ -z1
 S4 := (v2G2 ^ r) * (g2 ^ z2)
 S5 := g2b ^ -z2
 S6 := g1b ^ r2
 S7 := g1 ^ r1
 SK := (((u ^ M) * (w ^ tagk)) * h)^ r1
output := list{S1, S2, S3, S4, S5, S6, S7, SK, tagk}
END :: func:sign

BEGIN :: func:verify
input := list{pk, m, sig}
 pk := expand{g1, g2, g1b, g1a1, g1a2, g1ba1, g1ba2, tau1, tau2,
  tau1b, tau2b, uG1, u, wG1, hG1, w, h, A}
 sig := expand{S1, S2, S3, S4, S5, S6, S7, SK, tagk}
 s1 := random(ZR)
 s2 := random(ZR)
 t := random(ZR)
 tagc := random(ZR)
 s := s1 + s2
 M := H(m, ZR)
 theta := ((tagc - tagk)^-1)
BEGIN :: if
if { (e(g1b^s,S1) * (e(g1ba1^s1,S2) * (e(g1a1^s1,S3) *
      (e(g1ba2^s2,S4) * e(g1a2^s2,S5))))) == (e(S6,(tau1^s1)*(tau2^s2)) *
      (e(S7,((tau1b^s1)*((tau2b^s2)*w^-t))) *
      (((e(S7,((u^(M*t))*(w^(tagc*t)))*h^t) * (e(g1^-t,SK)))^theta) *
      (A^s2)))) }
   output := True
else
   output := False
END :: if
END :: func:verify
```

```
# Batcher input
BEGIN :: precompute
 s1 := random(ZR)
 s2 := random(ZR)
 t := random(ZR)
 tagc := random(ZR)
 s := s1 + s2
 M := H(m, ZR)
 theta := ((tagc - tagk)^-1)
END :: precompute

constant := list{g1, g2}
public := pk
signature := sig
message := M

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

verify := {(e(g1b^s,S1) * (e(g1ba1^s1,S2) * (e(g1a1^s1,S3) *
(e(g1ba2^s2,S4)  * e(g1a2^s2,S5))))) == (e(S6,(tau1^s1)*(tau2^s2)) *
(e(S7,((tau1b^s1)*((tau2b^s2)*w^-t))) *
(((e(S7,((u^(M*t))*(w^(tagc*t)))*h^t) * (e(g1^-t,SK)))^theta) * (A^s2))))}
```

A.3 Machine-Generated Batch Verification

In Figure A.5, we provide the final batch verification equations output by AutoBatch for each of the signature schemes tested.

Scheme	Batch Verification Equation output by AutoBatch
Signatures	
BLS [42] (same signer)	$e(\prod_{z=1}^{\eta} h_z^{\delta_z}, pk) \stackrel{?}{=} e(\prod_{z=1}^{\eta} sig_z^{\delta_z}, g)$
CHP [150] (same time period)	$e(\prod_{z=1}^{\eta} sig_z^{\delta_z}, g) \stackrel{?}{=} e(a, \prod_{z=1}^{\eta} pk_z^{\delta_z}) \cdot e(h, \prod_{z=1}^{\eta} pk_z^{b_z \cdot \delta_z})$
HW [135] (same signer)	$e(\prod_{z=1}^{\eta} \sigma_{1z}^{\delta_z}, g) \stackrel{?}{=} U^{\sum_{z=1}^{\eta} M_z \cdot \delta_z} \cdot V^{\sum_{z=1}^{\eta} r_z \cdot \delta_z} \cdot D^{\sum_{z=1}^{\eta} \delta_z}$ $\cdot e(\prod_{z=1}^{\eta} \sigma_{2z}^{\lg(i_z) \cdot \delta_z}, w) \cdot e(\prod_{z=1}^{\eta} \sigma_{2z}^{i_z \cdot \delta_z}, z) \cdot e(\prod_{z=1}^{\eta} \sigma_{2z}^{\delta_z}, h)$
HW [135] (different signers)	$e(\prod_{z=1}^{\eta} \sigma_{z,1}^{\delta_z}, g) \stackrel{?}{=} \prod_{z=1}^{\eta} U_z^{M_z \cdot \delta_z} \cdot \prod_{z=1}^{\eta} V_z^{r_z \cdot \delta_z}$ $\cdot \prod_{z=1}^{\eta} D_z^{\delta_z} \cdot e(\prod_{z=1}^{\eta} \sigma_{z,2}^{\delta_z \cdot \lceil \lg(i) \rceil_z}, w) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,2}^{\delta_z \cdot i_z}, z) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,2}^{\delta_z}, h)$
Waters09 [57] (same signer)	$e(g_1{}^{b}, \prod_{z=1}^{\eta} \sigma_{z,1}^{s_z \cdot \delta_z}) \cdot e(g_1{}^{b \cdot a_1}, \prod_{z=1}^{\eta} \sigma_{z,2}^{s_{z,1} \cdot \delta_z})$ $\cdot e(g_1{}^{a_1}, \prod_{z=1}^{\eta} \sigma_{z,3}^{s_{z,1} \cdot \delta_z}) \cdot e(g_1{}^{b \cdot a_2}, \prod_{z=1}^{\eta} \sigma_{z,4}^{s_{z,2} \cdot \delta_z})$ $\cdot e(g_1{}^{a_2}, \prod_{z=1}^{\eta} \sigma_{z,5}^{s_{z,2} \cdot \delta_z}) \stackrel{?}{=} e(\prod_{z=1}^{\eta} \sigma_{z,6}^{\delta_z \cdot s_{z,1}}, \tau_1)$ $\cdot e(\prod_{z=1}^{\eta} \sigma_{z,6}^{\delta_z \cdot s_{z,2}}, \tau_2) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot s_{z,1}}, \tau_1{}^{b})$ $\cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot s_{z,2}}, \tau_2{}^{b}) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{(\delta_z \cdot -t_z + \theta_z \cdot \delta_z \cdot tag_{z,c} \cdot t_z)}, w)$ $\cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot M_z \cdot t_z}, u) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot t_z}, h)$ $\cdot e(g_1, \prod_{z=1}^{\eta} \sigma_{z,K}^{-t_z \cdot \theta_z \cdot \delta_z}) \cdot A^{\sum_{z=1}^{\eta} s_{z,2} \cdot \delta_z}$
CL [18] (same signer)	$e(g, \prod_{z=1}^{\eta} b_z{}^{\delta_{z,1}} \cdot c_z{}^{\delta_{z,2}}) \cdot e(Y, \prod_{z=1}^{\eta} a_z{}^{-\delta_{z,1}}) \stackrel{?}{=} e(X, \prod_{z=1}^{\eta} a_z{}^{\delta_{z,2}} \cdot b_z{}^{m_z \cdot \delta_{z,2}})$
ID-based Signatures	
Hess [136]	$e(\prod_{z=1}^{\eta} S_{2z}^{\delta_z}, g_2) \stackrel{?}{=} e(\prod_{z=1}^{\eta} pk_z^{a_z \cdot \delta_z}, P_{pub}) \cdot \prod_{z=1}^{\eta} S_{1z}^{\delta_z}$
ChCh [137]	$e(\prod_{z=1}^{\eta} S_{2z}^{\delta_z}, g_2) \stackrel{?}{=} e(\prod_{z=1}^{\eta} (S_{1z} \cdot pk^{a_z})^{\delta_z}, P_{pub})$
Waters05 [67]	$e(\prod_{z=1}^{\eta} S_{1z}^{\delta_z}, g_2) \cdot e(\prod_{z=1}^{\eta} S_{2z}^{\delta_z}, \hat{u_1'}) \cdot \prod_{i=1}^{l} e(\prod_{z=1}^{\eta} S_{2z}^{\delta_z \cdot k_{i,z}} \cdot S_{3z}^{\delta_z \cdot m_{i,z}}, \hat{u}_i)$ $\cdot e(\prod_{z=1}^{\eta} S_{3z}^{\delta_z}, \hat{u_2'}) \stackrel{?}{=} e(g_1, g_2)^{\sum_{z=1}^{\eta} \delta_z}$
Group, Ring, and ID-based Ring Signatures	
BBS [59]	$e(\prod_{z=1}^{\eta} T_{z,3}^{s_{z,x} \cdot \delta_z} \cdot h^{(-s_{z,\gamma 1} - s_{z,\gamma 2}) \cdot \delta_z} \cdot g_1^{-c_z \cdot \delta_z}, g_2)$ $\cdot e(h^{\sum_{z=1}^{\eta} (-s_{z,\alpha} - s_{z,\beta}) \cdot \delta_z} \cdot \prod_{z=1}^{\eta} T_{z,3}^{c_z \cdot \delta_z}, w) \stackrel{?}{=} \prod_{z=1}^{\eta} R_{z,3}^{\delta_z}$
Boyen [70] (same ring)	$\prod_{y=1}^{l} e(\prod_{z=1}^{\eta} S_{y,z}{}^{\delta_z}, \hat{A}_y) \cdot e(\prod_{z=1}^{\eta} S_{y,z}{}^{m_{y,z} \cdot \delta_z}, \hat{B}_y) \cdot e(\prod_{z=1}^{\eta} S_{y,z}{}^{t_{y,z} \cdot \delta_z}, \hat{C}_y) \stackrel{?}{=} \prod_{z=1}^{\eta} D^{\delta_z}$
CYH [160]	$e(\prod_{z=1}^{\eta} \prod_{y=1}^{l} u_{y,z} \cdot pk_{y,z}^{h_{y,z}{}^{\delta_z}}, P) \stackrel{?}{=} e(\prod_{z=1}^{\eta} S_z^{\delta_z}, g)$
VRFs	
HW VRF [19] (same signer)	$e(\prod_{z=1}^{\eta} g_1^{(1-x_1) \cdot \delta_{z,2}} \cdot U_1^{x_1 \cdot \delta_{z,2}}, \hat{U}) \cdot e(\prod_{z=1}^{\eta} \pi_{z,1}^{-\delta_{z,2}} \cdot \pi_{z,2}^{\delta_{z,3}} \cdot \pi_{z,1}^{(1-x_{z,2}) \cdot -\delta_{z,3}}$ $\cdot \pi_{z,3}^{-\delta_{z,4}} \cdot \pi_{z,2}^{(1-x_{z,3}) \cdot -\delta_{z,4} \cdot -1} \cdot \pi_{z,4}^{-\delta_{z,5}} \cdot \pi_{z,3}^{(1-x_{z,4}) \cdot -\delta_{z,5} \cdot -1}$ $\cdot \pi_{z,5}^{-\delta_{z,6}} \cdot \pi_{z,4}^{(1-x_{z,5}) \cdot -\delta_{z,6} \cdot -1} \cdot \pi_{z,6}^{-\delta_{z,7}} \cdot \pi_{z,5}^{(1-x_{z,6}) \cdot -\delta_{z,7} \cdot -1} \cdot \pi_{z,7}^{-\delta_{z,8}} \cdot \pi_{z,6}^{(1-x_{z,7}) \cdot -\delta_{z,8} \cdot -1}$ $\cdot \pi_{z,8}^{-\delta_{z,9}} \cdot \pi_{z,7}^{(1-x_{z,8}) \cdot -\delta_{z,9} \cdot -1}, g_2) \stackrel{?}{=}$ $e(\prod_{z=1}^{\eta} \pi_{z,l}^{\delta_{z,1}}, U_0) \cdot \prod_{z=1}^{\eta} y_z{}^{\delta_{z,1}} \cdot e(\prod_{z=1}^{\eta} \pi_{z,0}^{-\delta_{z,1}}, g_2 \cdot h)$ $\cdot e(\prod_{z=1}^{\eta} \pi_{z,1}^{x_{z,2} \cdot \delta_{z,3}}, U_2) \cdot e(\prod_{z=1}^{\eta} \pi_{z,2}^{x_{z,3} \cdot \delta_{z,4} \cdot -1}, U_3) \cdot e(\prod_{z=1}^{\eta} \pi_{z,3}^{x_{z,4} \cdot \delta_{z,5} \cdot -1}, U_4)$ $\cdot e(\prod_{z=1}^{\eta} \pi_{z,4}^{x_{z,5} \cdot \delta_{z,6} \cdot -1}, U_5) \cdot e(\prod_{z=1}^{\eta} \pi_{z,5}^{x_{z,6} \cdot \delta_{z,7} \cdot -1}, U_6) \cdot e(\prod_{z=1}^{\eta} \pi_{z,6}^{x_{z,7} \cdot \delta_{z,8} \cdot -1}, U_7)$ $\cdot e(\prod_{z=1}^{\eta} \pi_{z,7}^{x_{z,8} \cdot \delta_{z,9} \cdot -1}, U_8)$ for block size of 8
Combinations	
ChCh + Hess	$e(\prod_{z=1}^{\eta} pk_z{}^{ah_z \cdot \delta_{z,1}} \cdot Sc_{z,1}^{-\delta_{z,2}} \cdot pk_z{}^{ac_z \cdot -\delta_{z,2}}, P_{pub}) \cdot \prod_{z=1}^{\eta} Sh_{z,1}^{\delta_{z,1}} \cdot$ $e(\prod_{z=1}^{\eta} Sh_{z,2}^{-\delta_{z,1}} \cdot Sc_{z,2}^{\delta_{z,2}}, g_2) \stackrel{?}{=} 1$

Figure A.5: 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 5.4 to the original verification equation in the order specified in Figure 5.7. ‘Combined signatures’ refers to the combined batching of multiple signature verification equations that share algebraic structure.

A.4 Proof for Batch Verification of HW Signatures

The following proof was automatically generated by the Batcher while processing the HW *signature scheme [135]. This execution allows signatures on different signing keys.*

A.4.1 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 $\delta_1, \ldots \delta_\eta \in \left[1, 2^\lambda - 1\right]$ 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.

Theorem A.4.1. HW.BatchVerify *is a batch verifier for the* HW *signature scheme.*

A.4.2 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.

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

Step 1: Combine η signatures (tech 1):

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

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

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

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

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

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

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

Step 5: Distribute products (tech 5):

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

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

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

Steps 1 and 2 form the Combination Step in [51], which was proven to result in a secure batch verifier in [51, Theorem 3.2]. We observe that the remaining steps are merely reorganizing terms within the same equation. Hence, the final verification equation (A.7) is also batch verifier for HW. □

A.5 Proof for Batch Verification of CL04 Signatures

The following proof was automatically generated by the Batcher while processing the CL04 *signature scheme [18]. This execution was restricted to signatures on a single signing key.*

A.5.1 Definitions

This document contains a proof that CL04.BatchVerify is a valid batch verifier for the signature scheme CL04. Let g be values drawn from the key and/or parameters, and a, b, c represent a message (or message hash) and signature. The individual verification equation CL04.Verify is:

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

Let η be the number of signatures in a batch, and $\delta_{1,i}, \ldots \delta_{\eta,i} \in \left[1, 2^\lambda - 1\right]$ where $i = 2$ be a set of random exponents chosen by the verifier. The batch verification equation for CL04 is:

CL04.BatchVerify:

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

We will now formally define a batch verifier and demonstrate that CL04.BatchVerify is a secure batch verifier for the CL04 signature scheme.

Theorem A.5.1. CL04.BatchVerify *is a batch verifier for the* CL04 *signature scheme.*

A.5.2 Proof

Proof. Via a series of steps, we will show that if CL04 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.

$$e(Y, a) \stackrel{?}{=} e(g, b) \text{ and } e(X, a) \cdot e(X, b)^m \stackrel{?}{=} e(g, c) \tag{A.8}$$

Step 1: Consolidate the verification equations (technique 0), and apply the small exponents test as follows: For each of the $z = 1$ to η signatures, choose random $\delta_{z,1}, \delta_{z,2} \in [1, 2^\lambda - 1]$ and compute the equation:

$$e(g, b)^{\delta_1} \cdot e(Y, a)^{-\delta_1} \stackrel{?}{=} e(X, a)^{\delta_2} \cdot e(X, b)^{m \cdot \delta_2} \cdot e(g, c)^{-\delta_2} \tag{A.9}$$

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

$$\prod_{z=1}^{\eta} e(g, {b_z}^{\delta_{z,1}}) \cdot e(Y, {a_z}^{-\delta_{z,1}}) \stackrel{?}{=} \prod_{z=1}^{\eta} e(X, {a_z}^{\delta_{z,2}}) \cdot e(X, {b_z}^{m_z \cdot \delta_{z,2}}) \cdot e(g, {c_z}^{-\delta_{z,2}}) \tag{A.10}$$

Step 3: Merge pairings with common first or second argument (technique 6):

$$\prod_{z=1}^{\eta} e(g, {b_z}^{\delta_{z,1}} \cdot {c_z}^{\delta_{z,2}}) \cdot e(Y, {a_z}^{-\delta_{z,1}}) \stackrel{?}{=} \prod_{z=1}^{\eta} e(X, {a_z}^{\delta_{z,2}}) \cdot e(X, {b_z}^{m_z \cdot \delta_{z,2}}) \tag{A.11}$$

Step 4: Merge pairings with common first or second argument (technique 6):

$$\prod_{z=1}^{\eta} e(g, {b_z}^{\delta_{z,1}} \cdot {c_z}^{\delta_{z,2}}) \cdot e(Y, {a_z}^{-\delta_{z,1}}) \stackrel{?}{=} \prod_{z=1}^{\eta} e(X, {a_z}^{\delta_{z,2}} \cdot {b_z}^{m_z \cdot \delta_{z,2}}) \tag{A.12}$$

Step 5: Move products inside pairings to reduce η pairings to 1 (technique 3):

$$\prod_{z=1}^{\eta} e(g, {b_z}^{\delta_{z,1}} \cdot {c_z}^{\delta_{z,2}}) \cdot e(Y, {a_z}^{-\delta_{z,1}}) \stackrel{?}{=} e(X, \prod_{z=1}^{\eta} {a_z}^{\delta_{z,2}} \cdot {b_z}^{m_z \cdot \delta_{z,2}}) \tag{A.13}$$

Step 6: Distribute products (technique 5):

$$\prod_{z=1}^{\eta} e(g, {b_z}^{\delta_{z,1}} \cdot {c_z}^{\delta_{z,2}}) \cdot \prod_{z=1}^{\eta} e(Y, {a_z}^{-\delta_{z,1}}) \stackrel{?}{=} e(X, \prod_{z=1}^{\eta} {a_z}^{\delta_{z,2}} \cdot {b_z}^{m_z \cdot \delta_{z,2}}) \tag{A.14}$$

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

$$e(g, \prod_{z=1}^{\eta} {b_z}^{\delta_{z,1}} \cdot {c_z}^{\delta_{z,2}}) \cdot e(Y, \prod_{z=1}^{\eta} {a_z}^{-\delta_{z,1}}) \stackrel{?}{=} e(X, \prod_{z=1}^{\eta} {a_z}^{\delta_{z,2}} \cdot {b_z}^{m_z \cdot \delta_{z,2}}) \tag{A.15}$$

Steps 1 and 2 form the Combination Step in [51], which was proven to result in a secure batch verifier in [51, Theorem 3.2]. We observe that the remaining steps are merely reorganizing terms within the same equation. Hence, the final verification equation (A.15) is also batch verifier for CL04. □

A.6 Proof for Batch Verification of VRF

The following proof was automatically generated by the Batcher while processing the VRF *signature scheme [19]. This execution was restricted to signatures on a single signing key.*

A.6.1 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 $\mathsf{VRF.Verify}$ 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 $\delta_{1,i}, \ldots \delta_{\eta,i} \in \left[1, 2^{\lambda} - 1\right]$ 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:

$\mathsf{VRFBatchVerify}$:

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

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

Theorem A.6.1. $\mathsf{VRF\ BatchVerify}$ *is a batch verifier for the* VRF *signature scheme.*

A.6.2 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 (tech 0), merge pairings with common first or second argument (tech 6), and apply the small exponents test as follows: For each of the $z = 1$ to η signatures, choose random $\delta_{z,1}, \delta_{z,2} \in [1, 2^\lambda - 1]$ and compute the equation:

$$e(g_1^{(1-x_1)} \cdot U_1^{x_1}, \hat{U})^{\delta_2} \cdot e(\pi_1, g_2)^{-\delta_2} \stackrel{?}{=} e(\pi_l, U_0)^{\delta_1} \cdot y^{\delta_1} \cdot e(\pi_0, g_2 \cdot h)^{-\delta_1} \tag{A.16}$$

EQ1 Step 2: Combine η signatures (tech 1), move exponents inside pairing (tech 2):

$$\prod_{z=1}^{\eta} e(g_1^{(1-x_1)\cdot\delta_{z,2}} \cdot U_1^{x_1\cdot\delta_{z,2}}, \hat{U}) \cdot \prod_{z=1}^{\eta} e(\pi_{z,1}^{-\delta_{z,2}}, g_2) \stackrel{?}{=} \prod_{z=1}^{\eta} e(\pi_{z,l}^{\delta_{z,1}}, U_0) \cdot \prod_{z=1}^{\eta} y_z^{\delta_{z,1}} \cdot \prod_{z=1}^{\eta} e(\pi_{z,0}^{-\delta_{z,1}}, g_2 \cdot h) \tag{A.17}$$

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

$$e(\prod_{z=1}^{\eta} g_1^{(1-x_1)\cdot\delta_{z,2}} \cdot U_1^{x_1\cdot\delta_{z,2}}, \hat{U}) \cdot e(\prod_{z=1}^{\eta} \pi_{z,1}^{-\delta_{z,2}}, g_2) \stackrel{?}{=} e(\prod_{z=1}^{\eta} \pi_{z,l}^{\delta_{z,1}}, U_0) \cdot \prod_{z=1}^{\eta} y_z^{\delta_{z,1}} \cdot e(\prod_{z=1}^{\eta} \pi_{z,0}^{-\delta_{z,1}}, g_2 \cdot h) \tag{A.18}$$

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

$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } \prod_{z=1}^{\eta} e(\pi_{z,t}, g_2) \stackrel{?}{=} \prod_{z=1}^{\eta} e(\pi_{z,t-1}^{(1-x_{z,t})}, g_2) \cdot e(\pi_{z,t-1}^{x_{z,t}}, U_t) \tag{A.19}$$

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

$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } \prod_{z=1}^{\eta} e(\pi_{z,t}, g_2)^{\delta_z} \stackrel{?}{=} \prod_{z=1}^{\eta} (e(\pi_{z,t-1}^{(1-x_{z,t})}, g_2) \cdot e(\pi_{z,t-1}^{x_{z,t}}, U_t))^{\delta_z} \tag{A.20}$$

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

$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } \prod_{z=1}^{\eta} e(\pi_{z,t}^{\delta_z}, g_2) \stackrel{?}{=} \prod_{z=1}^{\eta} e(\pi_{z,t-1}^{(1-x_{z,t})\cdot\delta_z}, g_2) \cdot e(\pi_{z,t-1}^{x_{z,t}\cdot\delta_z}, U_t) \tag{A.21}$$

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

$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } e(\prod_{z=1}^{\eta} \pi_{z,t}^{\delta_z}, g_2) \stackrel{?}{=} \prod_{z=1}^{\eta} e(\pi_{z,t-1}^{(1-x_{z,t})\cdot\delta_z}, g_2) \cdot e(\pi_{z,t-1}^{x_{z,t}\cdot\delta_z}, U_t) \tag{A.22}$$

EQ2 Step 8: Distribute products (tech 5):

$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } e(\prod_{z=1}^{\eta} \pi_{z,t}^{\delta_z}, g_2) \stackrel{?}{=} \prod_{z=1}^{\eta} e(\pi_{z,t-1}^{(1-x_{z,t})\cdot\delta_z}, g_2) \cdot \prod_{z=1}^{\eta} e(\pi_{z,t-1}^{x_{z,t}\cdot\delta_z}, U_t) \tag{A.23}$$

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

$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } e(\prod_{z=1}^{\eta} \pi_{z,t}^{\delta_z}, g_2) \stackrel{?}{=} e(\prod_{z=1}^{\eta} \pi_{z,t-1}^{(1-x_{z,t})\cdot\delta_z}, g_2) \cdot e(\prod_{z=1}^{\eta} \pi_{z,t-1}^{x_{z,t}\cdot\delta_z}, U_t) \tag{A.24}$$

EQ2 Step 10: Merge pairings with common first or second argument (tech 6):

$$\text{for } t = 2 \text{ to } \ell \text{ it holds: } e(\prod_{z=1}^{\eta} \pi_{z,t}^{\delta_z} \cdot \pi_{z,t-1}^{(1-x_{z,t})\cdot-\delta_z}, g_2) \stackrel{?}{=} e(\prod_{z=1}^{\eta} \pi_{z,t-1}^{x_{z,t}\cdot\delta_z}, U_t) \tag{A.25}$$

EQ2 Step 11: Unrolling for loop (tech 10) and choose random $\delta_{z,3}, \delta_{z,9} \in [1, 2^\lambda - 1]$ for

each unrolled equation:

$$e(\prod_{z=1}^{\eta} \pi_{z,2}^{\delta_{z,3}} \cdot \pi_{z,1}^{(1-x_{z,2})\cdot -\delta_{z,3}} \cdot \pi_{z,3}^{-\delta_{z,4}} \cdot \pi_{z,2}^{(1-x_{z,3})\cdot \delta_{z,4}} \cdot \pi_{z,4}^{-\delta_{z,5}} \cdot \pi_{z,3}^{(1-x_{z,4})\cdot \delta_{z,5}} \cdot \pi_{z,5}^{-\delta_{z,6}} \cdot \pi_{z,4}^{(1-x_{z,5})\cdot \delta_{z,6}}$$
$$\cdot \pi_{z,6}^{-\delta_{z,7}} \cdot \pi_{z,5}^{(1-x_{z,6})\cdot \delta_{z,7}} \cdot \pi_{z,7}^{-\delta_{z,8}} \cdot \pi_{z,6}^{(1-x_{z,7})\cdot \delta_{z,8}} \cdot \pi_{z,8}^{-\delta_{z,9}} \cdot \pi_{z,7}^{(1-x_{z,8})\cdot \delta_{z,9}}, g_2) \stackrel{?}{=}$$
$$e(\prod_{z=1}^{\eta} \pi_{z,1}^{x_{z,2}\cdot \delta_{z,3}}, U_2) \cdot e(\prod_{z=1}^{\eta} \pi_{z,2}^{x_{z,3}\cdot -\delta_{z,4}}, U_3) \cdot e(\prod_{z=1}^{\eta} \pi_{z,3}^{x_{z,4}\cdot -\delta_{z,5}}, U_4) \cdot e(\prod_{z=1}^{\eta} \pi_{z,4}^{x_{z,5}\cdot -\delta_{z,6}}, U_5)$$
$$\cdot e(\prod_{z=1}^{\eta} \pi_{z,5}^{x_{z,6}\cdot -\delta_{z,7}}, U_6) \cdot e(\prod_{z=1}^{\eta} \pi_{z,6}^{x_{z,7}\cdot -\delta_{z,8}}, U_7) \cdot e(\prod_{z=1}^{\eta} \pi_{z,7}^{x_{z,8}\cdot -\delta_{z,9}}, U_8) \quad \text{(A.26)}$$

Step 12: Combine equations 1 and 2, then pairings within final equation (tech 6):

$$e(\prod_{z=1}^{\eta} g_1^{(1-x_1)\cdot \delta_{z,2}} \cdot U_1^{x_1\cdot \delta_{z,2}}, \hat{U}) \cdot e(\prod_{z=1}^{\eta} \pi_{z,1}^{-\delta_{z,2}} \cdot \pi_{z,2}^{\delta_{z,3}} \cdot \pi_{z,1}^{(1-x_{z,2})\cdot -\delta_{z,3}} \cdot \pi_{z,3}^{-\delta_{z,4}} \cdot \pi_{z,2}^{(1-x_{z,3})\cdot \delta_{z,4}}$$
$$\cdot \pi_{z,4}^{-\delta_{z,5}} \cdot \pi_{z,3}^{(1-x_{z,4})\cdot \delta_{z,5}} \cdot \pi_{z,5}^{-\delta_{z,6}} \cdot \pi_{z,4}^{(1-x_{z,5})\cdot \delta_{z,6}} \cdot \pi_{z,6}^{-\delta_{z,7}} \cdot \pi_{z,5}^{(1-x_{z,6})\cdot \delta_{z,7}} \cdot \pi_{z,7}^{-\delta_{z,8}} \cdot \pi_{z,6}^{(1-x_{z,7})\cdot \delta_{z,8}} \cdot \pi_{z,8}^{-\delta_{z,9}}$$
$$\cdot \pi_{z,7}^{(1-x_{z,8})\cdot \delta_{z,9}}, g_2) \stackrel{?}{=} e(\prod_{z=1}^{\eta} \pi_{z,l}^{\delta_{z,1}}, U_0) \cdot \prod_{z=1}^{\eta} y_z^{\delta_{z,1}} \cdot e(\prod_{z=1}^{\eta} \pi_{z,0}^{-\delta_{z,1}}, g_2 \cdot h) \cdot e(\prod_{z=1}^{\eta} \pi_{z,1}^{x_{z,2}\cdot \delta_{z,3}}, U_2)$$
$$\cdot e(\prod_{z=1}^{\eta} \pi_{z,2}^{x_{z,3}\cdot -\delta_{z,4}}, U_3) \cdot e(\prod_{z=1}^{\eta} \pi_{z,3}^{x_{z,4}\cdot -\delta_{z,5}}, U_4) \cdot e(\prod_{z=1}^{\eta} \pi_{z,4}^{x_{z,5}\cdot -\delta_{z,6}}, U_5) \cdot e(\prod_{z=1}^{\eta} \pi_{z,5}^{x_{z,6}\cdot -\delta_{z,7}}, U_6)$$
$$\cdot e(\prod_{z=1}^{\eta} \pi_{z,6}^{x_{z,7}\cdot -\delta_{z,8}}, U_7) \cdot e(\prod_{z=1}^{\eta} \pi_{z,7}^{x_{z,8}\cdot -\delta_{z,9}}, U_8) \quad \text{(A.27)}$$

Steps 1 and 2 form the Combination Step in [51], which was proven to result in a secure batch verifier in [51, Theorem 3.2]. We observe that the remaining steps are merely reorganizing terms within the same equation except for the application of technique 10, which applies the small exponents test again while unrolling the loop. Hence, the final verification equation (A.27) is also batch verifier for VRF. □

A.7 Candidate Batch Verification for WATERS09 Signatures

The following candidate batching algorithm was automatically generated by the Batcher while processing the WATERS09 *signature scheme [57,180]. This execution was restricted to signatures on a single signing key.*

A.7.1 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][1] is:

$$e({g_1}^{bs}, \sigma_1) \cdot e({g_1}^{b \cdot a_1 s_1}, \sigma_2) \cdot e({g_1}^{a_1 s_1}, \sigma_3) \cdot e({g_1}^{b \cdot a_2 s_2}, \sigma_4) \cdot e({g_1}^{a_2 s_2}, \sigma_5) \stackrel{?}{=}$$

$$e(\sigma_6, \tau_1^{s_1} \cdot \tau_2^{s_2}) \cdot e(\sigma_7, {\tau_1}^{b s_1} \cdot {\tau_2}^{b s_2} \cdot w^{-t}) \cdot (e(\sigma_7, u^{M \cdot t} \cdot w^{tag_c \cdot t} \cdot h^t) \cdot e(g_1^{-t}, \sigma_K))^{\theta} \cdot A^{s_2}$$

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

[1] For simplicity, Waters [180] 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.

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 5.3.1 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.

A.7.2 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.

$$e(g_1{}^{bs}, \sigma_1) \cdot e(g_1{}^{b \cdot a_1 s_1}, \sigma_2) \cdot e(g_1{}^{a_1 s_1}, \sigma_3) \cdot e(g_1{}^{b \cdot a_2 s_2}, \sigma_4) \cdot e(g_1{}^{a_2 s_2}, \sigma_5) \stackrel{?}{=}$$
$$e(\sigma_6, \tau_1^{s_1} \cdot \tau_2^{s_2}) \cdot e(\sigma_7, \tau_1{}^{b s_1} \cdot \tau_2{}^{b s_2} \cdot w^{-t}) \cdot (e(\sigma_7, u^{M \cdot t} \cdot w^{tag_c \cdot t} \cdot h^{t}) \cdot e(g_1^{-t}, \sigma_K))^{\theta} \cdot A^{s_2} \quad \text{(A.28)}$$

Step 1: Combine η signatures (tech 1):

$$\prod_{z=1}^{\eta} e({g_1}^{bs_z}, \sigma_{z,1}) \cdot e({g_1}^{b \cdot a_1 s_{z,1}}, \sigma_{z,2}) \cdot e({g_1}^{a_1 s_{z,1}}, \sigma_{z,3}) \cdot e({g_1}^{b \cdot a_2 s_{z,2}}, \sigma_{z,4})$$
$$\cdot e({g_1}^{a_2 s_{z,2}}, \sigma_{z,5}) \stackrel{?}{=} \prod_{z=1}^{\eta} e(\sigma_{z,6}, \tau_1^{s_{z,1}} \cdot \tau_2^{s_{z,2}}) \cdot e(\sigma_{z,7}, {\tau_1}^{bs_{z,1}} \cdot {\tau_2}^{bs_{z,2}} \cdot w^{-t_z})$$
$$\cdot (e(\sigma_{z,7}, u^{M_z \cdot t_z} \cdot w^{tag_{z,c} \cdot t_z} \cdot h^{t_z}) \cdot e(g_1^{-t_z}, \sigma_{z,K}))^{\theta_z} \cdot A^{s_{z,2}} \quad \text{(A.29)}$$

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

$$\prod_{z=1}^{\eta} (e({g_1}^{bs_z}, \sigma_{z,1}) \cdot e({g_1}^{b \cdot a_1 s_{z,1}}, \sigma_{z,2}) \cdot e({g_1}^{a_1 s_{z,1}}, \sigma_{z,3}) \cdot e({g_1}^{b \cdot a_2 s_{z,2}}, \sigma_{z,4})$$
$$\cdot e({g_1}^{a_2 s_{z,2}}, \sigma_{z,5}))^{\delta_z} \stackrel{?}{=} \prod_{z=1}^{\eta} (e(\sigma_{z,6}, \tau_1^{s_{z,1}} \cdot \tau_2^{s_{z,2}}) \cdot e(\sigma_{z,7}, {\tau_1}^{bs_{z,1}} \cdot {\tau_2}^{bs_{z,2}} \cdot w^{-t_z})$$
$$\cdot (e(\sigma_{z,7}, u^{M_z \cdot t_z} \cdot w^{tag_{z,c} \cdot t_z} \cdot h^{t_z}) \cdot e(g_1^{-t_z}, \sigma_{z,K}))^{\theta_z} \cdot A^{s_{z,2}})^{\delta_z} \quad \text{(A.30)}$$

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

$$\prod_{z=1}^{\eta} e({g_1}^{bs_z \cdot \delta_z}, \sigma_{z,1}) \cdot e({g_1}^{b \cdot a_1 s_{z,1} \cdot \delta_z}, \sigma_{z,2}) \cdot e({g_1}^{a_1 s_{z,1} \cdot \delta_z}, \sigma_{z,3}) \cdot e({g_1}^{b \cdot a_2 s_{z,2} \cdot \delta_z}, \sigma_{z,4})$$
$$\cdot e({g_1}^{a_2 s_{z,2} \cdot \delta_z}, \sigma_{z,5}) \stackrel{?}{=} \prod_{z=1}^{\eta} e(\sigma_{z,6}^{\delta_z}, \tau_1^{s_{z,1}} \cdot \tau_2^{s_{z,2}}) \cdot e(\sigma_{z,7}^{\delta_z}, {\tau_1}^{bs_{z,1}} \cdot {\tau_2}^{bs_{z,2}} \cdot w^{-t_z})$$
$$\cdot e(\sigma_{z,7}^{\theta_z \cdot \delta_z}, u^{M_z \cdot t_z} \cdot w^{tag_{z,c} \cdot t_z} \cdot h^{t_z}) \cdot e(g_1^{-t_z \cdot \theta_z \cdot \delta_z}, \sigma_{z,K}) \cdot A^{s_{z,2} \cdot \delta_z} \quad \text{(A.31)}$$

Step 4: Split pairings (tech 9):

$$\prod_{z=1}^{\eta} e({g_1}^{b s_z \cdot \delta_z}, \sigma_{z,1}) \cdot e({g_1}^{b \cdot a_1 s_{z,1} \cdot \delta_z}, \sigma_{z,2}) \cdot e({g_1}^{a_1 s_{z,1} \cdot \delta_z}, \sigma_{z,3}) \cdot e({g_1}^{b \cdot a_2 s_{z,2} \cdot \delta_z}, \sigma_{z,4})$$
$$\cdot e({g_1}^{a_2 s_{z,2} \cdot \delta_z}, \sigma_{z,5}) \stackrel{?}{=} \prod_{z=1}^{\eta} e(\sigma_{z,6}^{\delta_z}, \tau_1^{s_{z,1}}) \cdot e(\sigma_{z,6}^{\delta_z}, \tau_2^{s_{z,2}}) \cdot e(\sigma_{z,7}^{\delta_z}, {\tau_1}^{b s_{z,1}}) \cdot e(\sigma_{z,7}^{\delta_z}, {\tau_2}^{b s_{z,2}})$$
$$\cdot e(\sigma_{z,7}^{\delta_z}, w^{-t_z}) \cdot e(\sigma_{z,7}^{\theta_z \cdot \delta_z}, u^{M_z \cdot t_z}) \cdot e(\sigma_{z,7}^{\theta_z \cdot \delta_z}, w^{tag_{z,c} \cdot t_z}) \cdot e(\sigma_{z,7}^{\theta_z \cdot \delta_z}, h^{t_z}) \cdot e(g_1^{-t_z \cdot \theta_z \cdot \delta_z}, \sigma_{z,K}) \cdot A^{s_{z,2} \cdot \delta_z} \tag{A.32}$$

Step 5: Distribute products (tech 5):

$$\prod_{z=1}^{\eta} e({g_1}^{b s_z \cdot \delta_z}, \sigma_{z,1}) \cdot \prod_{z=1}^{\eta} e({g_1}^{b \cdot a_1 s_{z,1} \cdot \delta_z}, \sigma_{z,2}) \cdot \prod_{z=1}^{\eta} e({g_1}^{a_1 s_{z,1} \cdot \delta_z}, \sigma_{z,3}) \cdot \prod_{z=1}^{\eta} e({g_1}^{b \cdot a_2 s_{z,2} \cdot \delta_z}, \sigma_{z,4})$$
$$\cdot \prod_{z=1}^{\eta} e({g_1}^{a_2 s_{z,2} \cdot \delta_z}, \sigma_{z,5}) \stackrel{?}{=} \prod_{z=1}^{\eta} e(\sigma_{z,6}^{\delta_z}, \tau_1^{s_{z,1}}) \cdot \prod_{z=1}^{\eta} e(\sigma_{z,6}^{\delta_z}, \tau_2^{s_{z,2}}) \cdot \prod_{z=1}^{\eta} e(\sigma_{z,7}^{\delta_z}, {\tau_1}^{b s_{z,1}})$$
$$\cdot \prod_{z=1}^{\eta} e(\sigma_{z,7}^{\delta_z}, {\tau_2}^{b s_{z,2}}) \cdot \prod_{z=1}^{\eta} e(\sigma_{z,7}^{\delta_z}, w^{-t_z}) \cdot \prod_{z=1}^{\eta} e(\sigma_{z,7}^{\theta_z \cdot \delta_z}, u^{M_z \cdot t_z}) \cdot \prod_{z=1}^{\eta} e(\sigma_{z,7}^{\theta_z \cdot \delta_z}, w^{tag_{z,c} \cdot t_z})$$
$$\cdot \prod_{z=1}^{\eta} e(\sigma_{z,7}^{\theta_z \cdot \delta_z}, h^{t_z}) \cdot \prod_{z=1}^{\eta} e(g_1^{-t_z \cdot \theta_z \cdot \delta_z}, \sigma_{z,K}) \cdot \prod_{z=1}^{\eta} A^{s_{z,2} \cdot \delta_z} \tag{A.33}$$

Step 6: Move products inside pairings to reduce η pairings to 1 (tech 3) and move product to summation on precomputed pairing (tech 7):

$$e({g_1}^{b}, \prod_{z=1}^{\eta} \sigma_{z,1}^{s_z \cdot \delta_z}) \cdot e({g_1}^{b \cdot a_1}, \prod_{z=1}^{\eta} \sigma_{z,2}^{s_{z,1} \cdot \delta_z}) \cdot e({g_1}^{a_1}, \prod_{z=1}^{\eta} \sigma_{z,3}^{s_{z,1} \cdot \delta_z}) \cdot e({g_1}^{b \cdot a_2}, \prod_{z=1}^{\eta} \sigma_{z,4}^{s_{z,2} \cdot \delta_z})$$
$$\cdot e({g_1}^{a_2}, \prod_{z=1}^{\eta} \sigma_{z,5}^{s_{z,2} \cdot \delta_z}) \stackrel{?}{=} e(\prod_{z=1}^{\eta} \sigma_{z,6}^{\delta_z \cdot s_{z,1}}, \tau_1) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,6}^{\delta_z \cdot s_{z,2}}, \tau_2) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot s_{z,1}}, {\tau_1}^{b})$$
$$\cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot s_{z,2}}, {\tau_2}^{b}) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot -t_z}, w) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot M_z \cdot t_z}, u) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot tag_{z,c} \cdot t_z}, w)$$
$$\cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot t_z}, h) \cdot e(g_1, \prod_{z=1}^{\eta} \sigma_{z,K}^{-t_z \cdot \theta_z \cdot \delta_z}) \cdot A^{\sum_{z=0}^{\eta} s_{z,2} \cdot \delta_z} \tag{A.34}$$

Step 7: Merge pairings with common first or second argument (tech 6):

$$e(g_1{}^{b}, \prod_{z=1}^{\eta} \sigma_{z,1}^{s_z \cdot \delta_z}) \cdot e(g_1{}^{b \cdot a_1}, \prod_{z=1}^{\eta} \sigma_{z,2}^{s_{z,1} \cdot \delta_z}) \cdot e(g_1{}^{a_1}, \prod_{z=1}^{\eta} \sigma_{z,3}^{s_{z,1} \cdot \delta_z}) \cdot e(g_1{}^{b \cdot a_2}, \prod_{z=1}^{\eta} \sigma_{z,4}^{s_{z,2} \cdot \delta_z})$$
$$\cdot e(g_1{}^{a_2}, \prod_{z=1}^{\eta} \sigma_{z,5}^{s_{z,2} \cdot \delta_z}) \stackrel{?}{=} e(\prod_{z=1}^{\eta} \sigma_{z,6}^{\delta_z \cdot s_{z,1}}, \tau_1) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,6}^{\delta_z \cdot s_{z,2}}, \tau_2) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot s_{z,1}}, \tau_1{}^{b})$$
$$\cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\delta_z \cdot s_{z,2}}, \tau_2{}^{b}) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{(\delta_z \cdot -t_z + \theta_z \cdot \delta_z \cdot tag_{z,c} \cdot t_z)}, w) \cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot M_z \cdot t_z}, u)$$
$$\cdot e(\prod_{z=1}^{\eta} \sigma_{z,7}^{\theta_z \cdot \delta_z \cdot t_z}, h) \cdot e(g_1, \prod_{z=1}^{\eta} \sigma_{z,K}^{-t_z \cdot \theta_z \cdot \delta_z}) \cdot A^{\sum_{z=0}^{\eta} s_{z,2} \cdot \delta_z} \quad \text{(A.35)}$$

Bibliography

[1] D. X. Song, A. Perrig, and D. Phan, "AGVI - automatic generation, verification, and implementation of security protocols," in *Proceedings of the 13th International Conference on Computer Aided Verification*, ser. CAV '01. Springer-Verlag, 2001, pp. 241–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=647770.734267

[2] D. Pozza, R. Sisto, and L. Durante, "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*, ser. AINA '04. IEEE Computer Society, 2004, pp. 400–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977394.977464

[3] S. Lucks, N. Schmoigl, and E. I. Tatli, "Issues on designing a cryptographic compiler," in *WEWoRC*, 2005, pp. 109–122.

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

[5] M. Backes, M. Maffei, and D. Unruh, “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*, ser. SP ’08. IEEE Computer Society, 2008, pp. 202–215. [Online]. Available: http://dx.doi.org/10.1109/SP.2008.23

[6] E. Bangerter, T. Briner, W. Henecka, S. Krenn, A.-R. Sadeghi, and T. Schneider, “Automatic generation of sigma-protocols,” in *Proceedings of the 6th European conference on Public key infrastructures, services and applications*, ser. EuroPKI’09. Springer-Verlag, 2010, pp. 67–82. [Online]. Available: http://dl.acm.org/citation.cfm?id=1927830.1927838

[7] J. B. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider, “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*, ser. ESORICS’10. Springer-Verlag, 2010, pp. 151–167. [Online]. Available: http://dl.acm.org/citation.cfm?id=1888881.1888894

[8] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya, “ZKPDL: a language-based system for efficient zero-knowledge proofs and electronic cash,” in *Proceedings of the 19th USENIX conference on Security*, ser. USENIX Security’10. USENIX Association, 2010, pp. 13–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929820.1929838

[9] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, “Fairplay – a secure two-party computation system,” in *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, ser. SSYM’04. USENIX Association, 2004, pp. 20–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251375.1251395

[10] W. Henecka, S. K ögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, “TASTY: tool for automating secure two-party computations,” in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS ’10. ACM, 2010, pp. 451–462. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866358

[11] J. A. Akinyele, C. Garman, I. Miers, M. W. Pagano, M. Rushanan, M. Green, and A. D. Rubin, “Charm: a framework for rapidly prototyping cryptosystems,” *Journal of Cryptographic Engineering*, vol. 3, no. 2, pp. 111–128, 2013. [Online]. Available: http://dx.doi.org/10.1007/s13389-013-0057-3

[12] D. Boneh, E. Shen, and B. Waters, “Strongly unforgeable signatures based on computational Diffie-Hellman,” in *PKC*, 2006, pp. 229–240.

[13] J. H. An, Y. Dodis, and T. Rabin, “On the security of joint signature and encryption,” in *Advances in Cryptology – EUROCRYPT ’02*, ser. Lecture Notes in Computer Science, L. R. Knudsen, Ed., vol. 2332. Springer, 2002, pp. 83–107.

[14] R. Canetti, S. Halevi, and J. Katz, “Chosen-ciphertext security from Identity Based Encryption,” in *EUROCRYPT*, vol. 3027 of LNCS, 2004, pp. 207–222.

[15] G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik, “A practical and provably secure coalition-resistant group signature scheme,” in *CRYPTO ’00*, vol. 1880 of LNCS, 2000, pp. 255–270.

[16] J. Camenisch and A. Lysyanskaya, “A signature scheme with efficient protocols,” in *Proceedings of the 3rd international conference on Security in communication networks*, ser. SCN. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 268–289. [Online]. Available: http://dl.acm.org/citation.cfm?id=1766811.1766838

[17] J. A. Akinyele, M. Green, S. Hohenberger, and M. W. Pagano, “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*, ser. CCS ’12. New York, NY, USA: ACM, 2012, pp. 474–487. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382248

[18] J. Camenisch and A. Lysyanskaya, “Signature schemes and anonymous credentials from bilinear maps,” in *CRYPTO*, vol. 3152 of LNCS. Springer, 2004, pp. 56–72.

[19] S. Hohenberger and B. Waters, “Constructing verifiable random functions with large input spaces,” in *Proceedings of the 29th Annual international conference on Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 656–672. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13190-5_33

[20] J. A. Akinyele, M. Green, and S. Hohenberger, “Using SMT solvers to

automate design tasks for encryption and signature schemes," in *Proceedings of the 2013 ACM conference on Computer and communications security*, ser. CCS '13. Berlin, Germany: ACM, 2013, pp. 399–410. [Online]. Available: http://dx.doi.org/10.1145/2508859.2516718

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

[22] A. Shamir, "Identity-based cryptosystems and signature schemes," in *CRYPTO*, 1984, pp. 47–53.

[23] D. Chaum and E. van Heyst, "Group signatures," in *EUROCRYPT*, 1991, pp. 257–265.

[24] R. L. Rivest, A. Shamir, and Y. Tauman, "How to leak a secret," in *ASIACRYPT*, 2001, pp. 552–565.

[25] S. Micali, M. O. Rabin, and S. P. Vadhan, "Verifiable random functions," in *FOCS*, 1999, pp. 120–130.

[26] M. Abe, M. Chase, B. David, M. Kohlweiss, R. Nishimaki, and M. Ohkubo, "Constant-size structure-preserving signatures: Generic constructions and simple assumptions," Cryptology ePrint Archive, Report 2012/285, 2012, http://eprint.iacr.org/.

[27] J. Groth and A. Sahai, "Efficient non-interactive proof systems for bilinear groups," in *EUROCRYPT*, vol. 4965 of LNCS. Springer, 2008, pp. 415–432.

[28] D. Boneh and M. K. Franklin, "Identity-based encryption from the Weil Pairing," in *CRYPTO*, vol. 2139 of LNCS, 2001, pp. 213–229.

[29] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy Attribute-Based Encryption," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2007, pp. 321–334.

[30] B. Waters, "Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization," Cryptology ePrint Archive, Report 2008/290, 2008, http://eprint.iacr.org/.

[31] A. Lewko and B. Waters, "Decentralizing attribute-based encryption," in *EUROCRYPT*, K. G. Patterson, Ed., vol. 6632 of LNCS. Springer, 2011, pp. 568–588, http://eprint.iacr.org/.

[32] A. Sahai and B. Waters, "Fuzzy identity-based encryption," in *EUROCRYPT*, 2005, pp. 457–473.

[33] D. Boneh, C. Gentry, and B. Waters, "Collusion resistant broadcast encryption with short ciphertexts and private keys," in *CRYPTO'05*, 2005, pp. 258–275.

[34] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," in *Proceedings of the Theory and practice of Software*, ser. TACAS'08/ETAPS'08, 2008, pp. 337–340.

[35] L. Moura and G. O. Passmore, “The strategy challenge in SMT solving,” in *Automated Reasoning and Mathematics*, 2013, vol. 7788, pp. 15–44.

[36] M. Barnett, K. R. M. Leino, and W. Schulte, “The spec# programming system: An overview.” Springer, 2004, pp. 49–69.

[37] R. DeLine, K. Rustan, and M. Leino, “Boogie pl: A typed procedural language for checking object-oriented programs,” Technical Report MSR-TR-2005-70.

[38] Wolfram, “Mathematica, version 9,” http://www.wolfram.com/mathematica/.

[39] M. Bellare and S. Shoup, “Two-tier signatures, strongly unforgeable signatures, and fiat-shamir without random oracles,” in *PKC*, 2007, pp. 201–216.

[40] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella-Béguelin, “Computer-aided security proofs for the working cryptographer,” in *Advances in Cryptology – CRYPTO 2011*, ser. Lecture Notes in Computer Science, vol. 6841. Springer, 2011, pp. 71–90. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22792-9_5

[41] B. Blanchet, “CryptoVerif: A computationally sound mechanized prover for cryptographic protocols,” in *Dagstuhl seminar "Formal Protocol Verification Applied"*, Oct. 2007.

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

[43] A. Fiat, "Batch RSA," in *Advances in Cryptology – CRYPTO '89*, vol. 435, 1989, pp. 175–185.

[44] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342

[45] NIST, "Digital Signature Standard (DSS)," Federal Information Processing Standards Publication 186, May 1994.

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

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

[48] M.-S. Hwang, I.-C. Lin, and K.-F. Hwang, "Cryptanalysis of the batch verifying multiple RSA digital signatures," *Informatica, Lithuanian Academy of Sciences*, vol. 11, no. 1, pp. 15–19, 2000.

[49] T. Cao, D. Lin, and R. Xue, "Security analysis of some batch verifying signatures from pairings," *International Journal of Network Security*, vol. 3, no. 2, pp. 138–143, 2006.

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

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

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

[53] B. Waters, "Dual system encryption: Realizing fully secure ibe and hibe under simple assumptions," in *CRYPTO*, 2009, pp. 619–636.

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

[55] S. Galbraith, K. Paterson, and N. Smart, "Pairings for cryptographers," Cryptology ePrint Archive, Report 2006/165, 2006, http://eprint.iacr.org/2006/165.

[56] S. C. Ramanna, S. Chatterjee, and P. Sarkar, "Variants of Waters' dual system primitives using asymmetric pairings - (extended abstract)," in *Public Key Cryptography*, 2012, pp. 298–315.

[57] B. Waters, "Dual System Encryption: Realizing Fully Secure IBE and HIBE under Simple Assumptions," in *CRYPTO*, 2009, pp. 619–636.

[58] D. Dolev, C. Dwork, and M. Naor, “Nonmalleable cryptography,” *SIAM J. Comput.*, vol. 30, no. 2, pp. 391–437, 2000.

[59] D. Boneh, X. Boyen, and H. Shacham, “Short group signatures,” in *CRYPTO*, vol. 3152 of LNCS, 2004, pp. 45–55.

[60] G. Lowe, “Casper: a compiler for the analysis of security protocols,” *J. Comput. Secur.*, vol. 6, no. 1-2, pp. 53–84, Jan. 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=353677.353680

[61] S. Kiyomoto, H. Ota, and T. Tanaka, “A security protocol compiler generating C source codes,” in *Proceedings of the 2008 International Conference on Information Security and Assurance (isa 2008)*, ser. ISA ’08. IEEE Computer Society, 2008, pp. 20–25. [Online]. Available: http://dx.doi.org/10.1109/ISA.2008.13

[62] J. Bacelar Almeida, M. Barbosa, E. Bangerter, G. Barthe, S. Krenn, and S. Zanella Béguelin, “Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS ’12. ACM, 2012, pp. 488–500. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382249

[63] C. Fournet, M. Kohlweiss, G. Danezis, and Z. Luo, “ZQL: A compiler for privacy-preserving data processing,” in *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, ser. SSYM’04. USENIX Association, 2004, pp. 20–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251375.1251395

[64] P. MacKenzie, A. Oprea, and M. K. Reiter, "Automatic generation of two-party computations," in *Proceedings of the 10th ACM conference on Computer and communications security*, ser. CCS '03. ACM, 2003, pp. 210–219. [Online]. Available: http://doi.acm.org/10.1145/948109.948139

[65] A. Ben-David, N. Nisan, and B. Pinkas, "Fairplaymp: a system for secure multi-party computation," in *Proceedings of the 15th ACM conference on Computer and communications security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 257–266. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455804

[66] D. Boneh and X. Boyen, "Efficient selective-ID secure Identity-Based Encryption without random oracles." in *EUROCRYPT*, vol. 3027 of LNCS, 2004, pp. 223–238.

[67] B. Waters, "Efficient Identity-Based Encryption without random oracles," in *EUROCRYPT*, vol. 3494 of LNCS, 2005, pp. 114–127.

[68] J. Camenisch, M. Kohlweiss, A. Rial, and C. Sheedy, "Blind and anonymous identity-based encryption and authorised private searches on public key encrypted data," in *PKC*, ser. Irvine. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 196–214. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00468-1_12

[69] A. Lewko, A. Sahai, and B. Waters, "Revocation systems with very small private keys," in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. SP. Washington, DC, USA: IEEE Computer Society, 2010, pp. 273–285. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.23

[70] X. Boyen, "Mesh signatures: How to leak a secret with unwitting and unwilling participants," in *EUROCRYPT, volume 4515 of LNCS*. Springer, 2007, pp. 210–227.

[71] S. S. M. Chow, S. M. Yiu, and L. C. K. Hui, "Efficient identity based ring signature," in *Applied Crypto And Network Security - ACNS, LNCS 3531*. Springer, 2005, pp. 499–512.

[72] J. Camenisch and J. Groth, "Group signatures: Better efficiency and new theoretical aspects," in *Security in Communication Networks*, ser. Lecture Notes in Computer Science, C. Blundo and S. Cimato, Eds., vol. 3352. Springer Berlin Heidelberg, 2005, pp. 120–133. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30598-9_9

[73] J. Camenisch and A. Lysyanskaya, "Signature schemes and anonymous credentials from bilinear maps." Springer-Verlag, 2004, pp. 56–72.

[74] S. Meiklejohn, K. Mowery, S. Checkoway, and H. Shacham, "The phantom tollbooth: privacy-preserving electronic toll collection in the presence of driver collusion," in *Proceedings of the 20th USENIX conference on Security*, ser. SEC. Berkeley, CA, USA: USENIX Association, 2011, pp. 32–32. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028099

[75] E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman, "Telex: Anticensorship

in the network infrastructure," in *Proceedings of the 20th USENIX Security Symposium*, Aug. 2011.

[76] J. Bethencourt, D. Song, and B. Waters, "Analysis-resistant malware," in *NDSS*, 2008.

[77] The OpenSSL Project, "OpenSSL: The open source toolkit for SSL/TLS," April 2010, www.openssl.org.

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

[79] J. Bethencourt, "Libpaillier," July 2006.

[80] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya, "ZKPDL: a language-based system for efficient zero-knowledge proofs and electronic cash," in *Proceedings of the 19th USENIX conference on Security*, ser. USENIX Security. Berkeley, CA, USA: USENIX Association, 2010, pp. 13–13. [Online]. Available: http://portal.acm.org/citation.cfm?id=1929820.1929838

[81] J. B. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider, "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*, ser. ESORICS. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 151–167. [Online]. Available: http://portal.acm.org/citation.cfm?id=1888881.1888894

[82] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay - a secure two-party computation system," in *Proceedings of the 13th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2004, pp. 287–302.

[83] W. Henecka, S. K ögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "Tasty: tool for automating secure two-party computations," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS. New York, NY, USA: ACM, 2010, pp. 451–462. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866358

[84] B. Laurie and B. Clifford, "The Stupid programming language," Source code available at http://code.google.com/p/stupid-crypto/.

[85] J. R. Lewis and B. Martin, "CRYPTOL: High Assurance, Retargetable Crypto Development and Validation," Available from http://www.galois.com/files/Cryptol_Whitepaper.pdf, October 2003.

[86] W. Stein *et al.*, *Sage Mathematics Software (Version 5.0.1)*, The Sage Development Team, YYYY, `http://www.sagemath.org`.

[87] M. Scott, "MIRACL library," indigo Software. http://indigo.ie/~mscott/#download.

[88] J. A. Akinyele, M. Green, and A. Rubin, "Charm-crypto framework," http://eprint.iacr.org/2011/617.

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

[90] T. Acar, M. Belenkiy, M. Bellare, and D. Cash, "Cryptographic agility and its relation to circular encryption," in *EUROCRYPT*, 2010.

[91] X. Wang and H. Yu, "How to break md5 and other hash functions," in *In EUROCRYPT*. Springer-Verlag, 2005.

[92] X. Wang, Y. L. Yin, and H. Yu, "Finding collisions in the full sha-1," in *Proceedings of Crypto*. Springer, 2005, pp. 17–36.

[93] T. Acar, C. Fournet, and D. Shumow, "Design and veriïňĄcation of a crypto-agile distributed key manager," 2011. [Online]. Available: http://research.microsoft.com/en-us/um/people/fournet/dkm/dkm-design-and-verification-draft.pdf

[94] O. Regev, "Lattice-based cryptography," in *Advances in Cryptology - CRYPTO 2006*, ser. Lecture Notes in Computer Science, C. Dwork, Ed., vol. 4117. Springer Berlin Heidelberg, 2006, pp. 131–141. [Online]. Available: http://dx.doi.org/10.1007/11818175_8

[95] D. Dolev, C. Dwork, and M. Naor, "Non-malleable cryptography," in *SIAM Journal on Computing*, 2000, pp. 542–552.

[96] O. Goldreich, S. Micali, and A. Wigderson, "Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems,"

J. ACM, vol. 38, no. 3, pp. 690–728, Jul. 1991. [Online]. Available: http://doi.acm.org/10.1145/116825.116852

[97] S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of computer and system sciences*, vol. 28, no. 2, pp. 270–299, 1984.

[98] M. Naor and M. Yung, "Public-key cryptosystems provably secure against chosen ciphertext attacks," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. ACM, 1990, pp. 427–437.

[99] M. Bellare and P. Rogaway, "The Exact Security of Digital Signatures - How to Sign with RSA and Rabin," in *EUROCRYPT '96*, vol. 1070 of LNCS. Springer, 1996, pp. 399–416.

[100] GNU, "The GNU Multiple Precision Arithmetic Library," Available from http://www.gmplib.org.

[101] D. C. Litzenberger, "PyCrypto - The Python Cryptography Toolkit," Available at http://www.dlitz.net/software/pycrypto/.

[102] J. Camenisch and E. Van Herreweghen, "Design and implementation of the idemix anonymous credential system," in *Proceedings of the 9th ACM conference on Computer and communications security*, ser. CCS. New York, NY, USA: ACM, 2002, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/586110.586114

[103] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in

Proceedings of the 11th ACM conference on Computer and communications security, ser. CCS. New York, NY, USA: ACM, 2004, pp. 132–145. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030103

[104] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *CRYPTO*, vol. 263 of LNCS, 1986, pp. 186–194.

[105] J. Camenisch and M. Stadler, "Efficient group signature schemes for large groups," in *CRYPTO*, vol. 1296 of LNCS, 1997, pp. 410–424.

[106] R. Cramer and V. Shoup, "A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack," in *CRYPTO*. London, UK: Springer, 1998, pp. 13–25.

[107] G. Condra, "pypbc," Available from http://www.gitorious.org/pypbc.

[108] T. S. Denis, "LibTomCrypt Project," Available at http://libtom.org.

[109] T. El Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Proceedings of Crypto*, 1984, pp. 10–18.

[110] M. Bellare and P. Rogaway, "Optimal asymmetric encryption padding — how to encrypt with rsa," in *EUROCRYPT*, 1994, pp. 92–111.

[111] G. Blakley, D. Chaum, and T. ElGamal, *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*. Springer Berlin / Heidelberg, 1985, vol. 196, pp. 10–18. [Online]. Available: http://dx.doi.org/10.1007/3-540-39568-7_2

[112] J. Stern and P. Paillier, *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes*. Springer Berlin / Heidelberg, 1999, vol. 1592, pp. 223–238. [Online]. Available: http://dx.doi.org/10.1007/3-540-48910-X_16

[113] B. Waters, "Functional encryption for regular languages," in *Advances in Cryptology âĂŞ CRYPTO 2012*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer Berlin Heidelberg, 2012, pp. 218–235. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32009-5_14

[114] V. Iovino and G. Persiano, "Hidden-vector encryption with groups of prime order," in *Proceedings of the 2nd international conference on Pairing-Based Cryptography*, ser. Pairing '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 75–88. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85538-5_5

[115] G. Brassard and C. Schnorr, *Efficient Identification and Signatures for Smart Cards*. Springer Berlin / Heidelberg, 1990, vol. 435, pp. 239–252. [Online]. Available: http://dx.doi.org/10.1007/0-387-34805-0_22

[116] S. Hohenberger and B. Waters, "Realizing hash-and-sign signatures under standard assumptions," in *Advances in Cryptology – EUROCRYPT*, 2009.

[117] J. Camenisch, S. Hohenberger, and M. ÃŸstergaard Pedersen, "Batch verification of short signatures," in *EUROCRYPT, volume 4515 of LNCS*. Springer, 2007, pp. 246–263.

[118] F. Hess, “Efficient identity based signature schemes based on pairings,” in *SAC, LNCS 2595*. Springer-Verlag, 2002, pp. 310–324.

[119] J. C. Cha and J. H. Cheon, “An identity-based signature from gap diffie-hellman groups,” in *PKC*. Springer-Verlag, LNCS 2139, 2003, pp. 18–30.

[120] D. Boneh, B. Lynn, and H. Shacham, “Short signatures from the Weil Pairing,” in *ASIACRYPT*, vol. 2248 of LNCS, 2001, pp. 514–532.

[121] J. Camenisch and A. Lysyanskaya, “An efficient system for non-transferable anonymous credentials with optional anonymity revocation,” in *EUROCRYPT*, vol. 2045 of LNCS. Springer, 2001, pp. 93–118.

[122] J. Camenisch, G. Neven, and abhi shelat, “Simulatable adaptive oblivious transfer,” in *EUROCRYPT*, vol. 4515 of LNCS, 2007, pp. 573–590.

[123] E. Bangerter, J. Camenisch, S. Krenn, A.-R. Sadeghi, and T. Schneider, “Automatic generation of sound zero-knowledge protocols,” Cryptology ePrint Archive, Report 2008/471, 2008, http://eprint.iacr.org/.

[124] “The Advanced Crypto Software Collection,” http://acsc.cs.utexas.edu/.

[125] Y. Rouselakis and B. Waters, “Practical constructions and new proof methods for large universe attribute-based encryption,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ser. CCS

'13. New York, NY, USA: ACM, 2013, pp. 463–474. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516672

[126] J. B. Lacy, "CryptoLib: Cryptography in software," *USENIX Security Conference IV*, pp. 1–18, 1993.

[127] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *Progress in Cryptology – LATINCRYPT*, ser. Lecture Notes in Computer Science, A. Hevia and G. Neven, Eds., vol. to appear. Springer-Verlag Berlin Heidelberg, 2012, document ID: 5f6fc69cc5a319aecba43760c56fab04, http://cryptojedi.org/papers/#coolnacl.

[128] E. Bangerter, S. Barzan, A. Sadeghi, T. Schneider, and J. Tsay, "Bringing zero-knowledge proofs of knowledge to practice," *17th International Workshop on Security Protocols*, 2009.

[129] D. Freeman, "Converting pairing-based cryptosystems from composite-order groups to prime-order groups." in *EUROCRYPT, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2010, pp. 44–61.

[130] A. B. Lewko, "Tools for simulating features of composite order bilinear groups in the prime order setting," *IACR Cryptology ePrint Archive*, vol. 2011, p. 490, 2011.

[131] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st annual ACM symposium on Theory of computing*, ser. STOC.

New York, NY, USA: ACM, 2009, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/1536414.1536440

[132] M. Dufour, “Shedskin,” Available from http://code.google.com/p/shedskin, July 2009.

[133] Car 2 Car, “Communication consortium,” http://car-to-car.org.

[134] SeVeCom, “Security on the road,” http://www.sevecom.org.

[135] S. Hohenberger and B. Waters, “Realizing hash-and-sign signatures under standard assumptions,” in *EUROCRYPT*, 2009, pp. 333–350.

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

[137] J. C. Cha and J. H. Cheon, “An identity-based signature from gap Diffie-Hellman groups,” in *PKC ’03*, vol. 2567 of LNCS. Springer, 2003, pp. 18–30.

[138] J. A. Akinyele, M. Green, S. Hohenberger, and M. W. Pagano, “Machine-generated algorithms, proofs and software for the batch verification of digital signature schemes,” Cryptology ePrint Archive, Report 2013/175, 2013, http://eprint.iacr.org/.

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

[140] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, “Computer-aided security proofs for the working cryptographer,” in *CRYPTO*, 2011, pp. 71–90.

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

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

[143] C.-S. Laih and S.-M. Yen, "Improved digital signature suitable for batch verification," *IEEE Transactions on Computers*, vol. 44, no. 7, pp. 957–959, 1995.

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

[145] ——, "Batch verifying multiple RSA digital signatures," *Electronics Letters*, vol. 34(12), pp. 1219–1220, 1998.

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

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

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

[149] S. Lee, S. Cho, J. Choi, and Y. Cho, “Efficient identification of bad signatures in RSA-type batch signature,” *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E89-A, no. 1, pp. 74–80, 2006.

[150] J. Camenisch, S. Hohenberger, and M. Ø. Pedersen, “Batch verification of short signatures,” in *EUROCRYPT ’07*, vol. 4515 of LNCS. Springer, 2007, pp. 246–263, full version at http://eprint.iacr.org/2007/172.

[151] H. Shacham and D. Boneh, “Improving SSL handshake performance via batching,” in *Cryptographer’s Track at RSA Conference ’01*, vol. 2020, 2001, pp. 28–43.

[152] O. Blazy, G. Fuchsbauer, M. Izabachène, A. Jambert, H. Sibert, and D. Vergnaud, “Batch groth-sahai,” in *ACNS ’10*. Springer, 2010, pp. 218–235.

[153] B. J. Matt, “Identification of multiple invalid signatures in pairing-based batched signatures,” in *Public Key Cryptography*, 2009, pp. 337–356.

[154] ——, “Identification of multiple invalid pairing-based signatures in constrained batches,” in *Pairing*, 2010, pp. 78–95.

[155] M. Barbosa, A. Moss, and D. Page, “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*, ser. OTM’07. Springer-Verlag, 2007, pp. 1785–1802. [Online]. Available: http://dl.acm.org/citation.cfm?id=1784707.1784769

[156] L. J. D. Perez and M. Scott, “Designing a code generator for pairing based cryptographic functions,” in *Proceedings of the 4th international conference on Pairing-based cryptography*, ser. Pairing’10. Springer-Verlag, 2010, pp. 207–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1948966.1948987

[157] B. Barak, R. Canetti, J. B. Nielsen, and R. Pass, “Universally composable protocols with relaxed set-up assumptions,” in *FOCS*. IEEE Computer Society, 2004, pp. 186–195.

[158] D. Naccache, “Secure and *practical* identity-based encryption,” 2005, cryptology ePrint Archive: Report 2005/369.

[159] S. Chatterjee and P. Sarkar, “HIBE with short public parameters without random oracle,” in *ASIACRYPT ’06*, vol. 4284 of LNCS, 2006, pp. 145–160.

[160] S. S. M. Chow, S.-M. Yiu, and L. C. Hui, “Efficient identity based ring signature,” in *ACNS*, vol. 3531 of LNCS, 2005, pp. 499–512.

[161] G. M. Zaverucha and D. R. Stinson, “Group testing and batch verification,” in *Proceedings of the 4th international conference on Information theoretic security*, ser. ICITS’09. Springer-Verlag, 2010, pp. 140–157. [Online]. Available: http://dl.acm.org/citation.cfm?id=1880513.1880531

[162] J. A. Akinyele, M. Green, S. Hohenberger, and M. W. Pagano, “AutoBatch Toolkit,” https://github.com/jhuisi/auto-tools.

[163] S. D. Galbraith, K. G. Paterson, and N. P. Smart, "Pairings for cryptographers," 2006, cryptology ePrint Archive: Report 2006/165.

[164] I. Teranishi, T. Oyama, and W. Ogata, "General conversion for obtaining strongly existentially unforgeable signatures," in *INDOCRYPT*, 2006, pp. 191–205.

[165] Q. Huang, D. S. Wong, and Y. Zhao, "Generic transformation to strongly unforgeable signatures," in *ACNS*, 2007, pp. 1–17.

[166] R. Steinfeld, J. Pieprzyk, and H. Wang, "How to strengthen any weakly unforgeable signature into a strongly unforgeable signature," in *CT-RSA*, 2007, pp. 357–371.

[167] I. Teranishi, T. Oyama, and W. Ogata, "General conversion for obtaining strongly existentially unforgeable signatures," *IEICE Transactions*, vol. 91-A, no. 1, pp. 94–106, 2008.

[168] M. Bellare and S. Shoup, "Two-tier signatures from the fiat-shamir transform, with applications to strongly unforgeable and one-time signatures," *IET Information Security*, vol. 2, no. 2, pp. 47–63, 2008.

[169] A. Menezes, S. Vanstone, and T. Okamoto, "Reducing elliptic curve logarithms to logarithms in a finite field," in *STOC*, 1991, pp. 80–89.

[170] S. D. Galbraith, "Supersingular curves in cryptography," in *ASIACRYPT*, 2001, pp. 495–513.

[171] D. Page, N. Smart, and F. Vercauteren, "A comparison of MNT curves and supersingular curves," *Applicable Algebra in Eng,Com and Comp*, vol. 17, no. 5, pp. 379–392, 2006.

[172] P. S. L. M. Barreto and M. Naehrig, "Pairing-friendly elliptic curves of prime order," in *SAC*, vol. 3897, 2006, pp. 319–331, http://cryptojedi.org/papers/#pfcpo.

[173] D. Boneh and X. Boyen, "Efficient selective-id secure identity-based encryption without random oracles," in *Advances in Cryptology - EUROCRYPT 2004*, ser. Lecture Notes in Computer Science, C. Cachin and J. Camenisch, Eds. Springer Berlin Heidelberg, 2004, vol. 3027, pp. 223–238. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24676-3_14

[174] C. Gentry, "Practical identity-based encryption without random oracles," in *EUROCRYPT*, vol. 4004 of LNCS, 2006, pp. 445–464.

[175] A. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.

[176] H. Krawczyk and T. Rabin, "Chameleon signatures," in *NDSS*, 2000.

[177] O. Goldreich, *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[178] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *CRYPTO*, 1986, pp. 186–194.

[179] C.-P. Schnorr, "Efficient signature generation by smart cards," *J. Cryptology*, vol. 4, no. 3, pp. 161–174, 1991.

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

[181] D. Boneh and J. Katz, "Improved efficiency for cca-secure cryptosystems built using identity based encryption," in *CT-RSA*, vol. 3376 of LNCS. Springer, 2005.

[182] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *CRYPTO*, vol. 576 of LNCS, 1992, pp. 129–140.

[183] G. Ateniese and B. de Medeiros, "On the key exposure problem in chameleon hashes," in *SCN*, vol. 3352 of LNCS. Springer, 2004, pp. 165–179.

Vita

Joseph Ayo Akinyele graduated from Bowie State University *summa cum laude* with a Bachelor of Science in Computer Science in 2006. As an undergraduate, he received a Model Institutions for Excellence (MIE) fellowship to conduct research in computer security. As a result, Joseph secured internships and worked with researchers at Fermi National Accelerator Laboratory (Fermilab) and Johns Hopkins University Applied Physics Laboratory (JHU-APL).

Upon graduating from Bowie, he pursued a Master of Science in Software Engineering at Carnegie Mellon University. He graduated in 2007 with *magna cum laude* distinction and was supported by a Graduate Degree for Minorities in Engineering and Science (GEM) fellowship. Joseph then worked at JHU-APL as a software engineer from 2007 to 2010.

In 2009, Joseph began the Ph.D. program in Computer Science as a member of the Information Security Institute (ISI). His primary research includes developing cryptographic frameworks to assist in the design of cryptography such as the open source Charm cryp-

tographic library. Furthermore, his research builds on this work to automate the design of certain aspects of cryptography including batch verification of digital signatures, construction of strongly unforgeable signatures, and optimizing the efficiency and bandwidth of cryptographic schemes.