Notes on Mathematics and Cryptography*

Christopher H. Gorman

February 26, 2023†

*Version 0.2
†Initial Public Release: June 28, 2022

About the Author The author is a mathematician by training. He earned his bachelor's, master's, and doctorate in mathematics. He did not learn cryptography in a classroom but rather learned it from reading textbooks and papers. The present work is meant to help others learn cryptography and the mathematics behind it in an easier manner.

Permanent ID of this document: `b3d5f96fbf2562f1da3c3f3f8d3a95b7`. Date: 2023.02.26.

Contents

List of Algorithms x

List of Figures xi

List of Listings xiii

List of Tables xiv

1 Introduction 1
- 1.1 Primary Goal . . . 1
- 1.2 Intended Audience . . . 1
- 1.3 Examples . . . 2
- 1.4 Overview . . . 2
- 1.5 Cryptographic Participants . . . 3
- 1.6 Mathematical Precision . . . 3
- 1.7 Additional Resources . . . 3

2 What Not to Do 4
- 2.1 General Advice . . . 4
- 2.2 Symmetric Key Encryption . . . 6
- 2.3 Cryptographic Hash Functions . . . 6
- 2.4 Cryptographically-Secure Pseudorandom Number Generators . . . 6
- 2.5 Public Key Encryption . . . 7
- 2.6 Digital Signatures . . . 7

3 Mathematical Review: Set Theory, Number Theory, and Complexity 8
- 3.1 Mathematical Notation, Set Theory, and Functions . . . 8
  - 3.1.1 Introduction to Set Theory . . . 8
  - 3.1.2 Standard Mathematical Sets . . . 10
  - 3.1.3 Functions . . . 12
  - 3.1.4 Function Properties . . . 12
  - 3.1.5 Permutations . . . 15
- 3.2 Bit Operations . . . 16
  - 3.2.1 Definitions . . . 16
  - 3.2.2 Bit Operations on Integers . . . 17
- 3.3 Size of Numbers . . . 18

3.4 Number Theory . . . 20
3.4.1 Divisibility and Prime Numbers . . . 20
3.4.2 Greatest Common Divisor . . . 21
3.4.3 Congruent Numbers . . . 21
3.4.4 Modular Arithmetic . . . 22
3.5 Computational Complexity . . . 24
3.5.1 Big-O Notation . . . 24
3.5.2 Standard Complexity Classes . . . 25
3.5.3 Complexity of Various Operations . . . 26

4 Mathematical Review: Groups, Rings, and Fields 27
4.1 Groups . . . 27
4.1.1 Why do we care about Groups? . . . 27
4.1.2 Intuition and Examples . . . 27
4.1.3 Formal Definition . . . 30
4.1.4 Continued Discussion . . . 30
4.1.5 More Definitions . . . 31
4.1.6 Encoding Groups . . . 32
4.2 Rings . . . 32
4.2.1 Why do we care about Rings? . . . 32
4.2.2 Intuition and Examples . . . 33
4.2.3 Formal Definition . . . 33
4.2.4 Continued Discussion . . . 34
4.2.5 Encoding Rings . . . 35
4.3 Fields . . . 35
4.3.1 Why do we care about Fields? . . . 35
4.3.2 Intuition and Examples . . . 35
4.3.3 Formal Definition . . . 36
4.3.4 Continued Discussion . . . 36
4.3.5 More Examples . . . 36
4.3.6 Encoding Fields . . . 38
4.4 Concluding Discussion . . . 38

5 Mathematical Review: Elliptic Curves, Pairings, and Interpolation 39
5.1 Important Information . . . 39
5.2 Elliptic Curves . . . 39
5.2.1 Why do we care about Elliptic Curves? . . . 39
5.2.2 Elliptic Curves are *not* Ellipses . . . 40
5.2.3 Elliptic Curves over the Reals . . . 40
5.2.4 Elliptic Curve Addition . . . 43
5.2.5 Elliptic Curves over General Fields . . . 47
5.2.6 Elliptic Curves over Finite Fields . . . 47
5.2.7 Elliptic Curve Scalar Multiplication . . . 51
5.2.8 Subgroups of Elliptic Curves over Finite Fields . . . 56
5.2.9 Elliptic Curve Point Compression . . . 58

5.2.10 Examples of Specific Curves . . . 60
5.2.11 Encoding Elliptic Curves over Finite Fields . . . 61
5.3 Bilinear Pairings . . . 61
5.3.1 Why do we care about Bilinear Pairings? . . . 61
5.3.2 Formal Definition . . . 62
5.3.3 Discussion . . . 62
5.4 Lagrange Interpolation . . . 62
5.4.1 Why do we care about interpolation? . . . 62
5.4.2 Lagrange Interpolation over the Reals . . . 62
5.4.3 Examples of Lagrange Interpolation over the Reals . . . 63
5.4.4 Problems with Lagrange Interpolation over the Reals . . . 65
5.4.5 Lagrange Interpolation over Finite Fields . . . 65
5.4.6 Examples of Lagrange Interpolation over Finite Fields . . . 65
5.4.7 Further Generalizations of Lagrange Interpolation . . . 67
5.5 Conclusion of Mathematical Review . . . 67

6 Symmetric Key Cryptography 69
6.1 The Need for Symmetric Key Cryptography . . . 69
6.2 Unbreakable Encryption: the One-Time Pad . . . 69
6.2.1 Definition . . . 69
6.2.2 Examples . . . 70
6.2.3 Discussion . . . 72
6.3 Encryption Schemes . . . 72
6.3.1 Stream Ciphers . . . 72
6.3.2 Block Ciphers . . . 74
6.4 Cryptographic Hash Functions . . . 76
6.5 Key Derivation Functions . . . 76
6.6 Cryptographically-Secure Pseudorandom Number Generators . . . 76
6.6.1 Discussion . . . 76
6.6.2 Examples . . . 77
6.7 Message Authentication Codes . . . 78
6.7.1 Discussion . . . 78
6.7.2 Examples . . . 79
6.8 Authenticated Encryption . . . 79

7 Cryptographic Hash Functions 80
7.1 The Need for Cryptographic Hash Functions . . . 80
7.2 Desired Properties of Cryptographic Hash Functions . . . 80
7.3 Random Oracle: The Ideal Cryptographic Hash Function . . . 82
7.4 Additional Properties . . . 82
7.5 Examples . . . 82
7.6 Domain Separation . . . 83
7.7 Known Challenges with Certain Hash Functions . . . 83
7.7.1 Length Extension Attacks . . . 88
7.7.2 Other Attacks . . . 88

7.8 Properties of a Secure Cryptographic Hash Function 88

8 Applications of Cryptographic Hash Functions 92
8.1 Ethereum Addresses 92
8.1.1 Address Definition 92
8.1.2 Colliding Addresses 92
8.1.3 Example 93
8.2 Commitment Schemes 93
8.3 Hash Chains 95
8.4 Merkle Trees I 97
8.4.1 Intuition 97
8.4.2 Definition 97
8.4.3 Continued Discussion 97
8.4.4 Examples 97
8.5 Merkle Trees II 112
8.5.1 Merkle Tree Security 112
8.5.2 Merkle Trees with Arbitrary Leaves 112
8.6 Sparse Merkle Trees 117
8.6.1 Discussion 117
8.6.2 Conventions 117
8.7 Hash-based Message Authentication Code 117
8.8 HMAC-based Key Derivation Function 118
8.9 Mask Generation Functions and Extendable Output Functions 119
8.10 Password Hashing 119
8.11 Concluding Discussion of Hash Functions 120

9 Public Key Cryptography 124
9.1 The Need for Public Key Cryptography 124
9.2 Brief Discussion of Public Key Infrastructure 124
9.3 Diffie-Hellman Key Exchange 124
9.3.1 Intuition and Discussion 125
9.3.2 Formal Definition 127
9.3.3 Potential Confusion between Public and Private Keys 127
9.3.4 Security of the Diffie-Hellman Key Exchange 127
9.4 Elgamal Encryption 127
9.4.1 Formal Definition 127
9.4.2 Discussion and Examples 128
9.5 Digital Signatures 129
9.6 Public Key Encryption in Practice 129
9.7 Ciphertext Malleability 130

10 Digital Signatures 131
10.1 The Need for Digital Signatures 131
10.2 Clearing Up Potential Confusion about Digital Signatures 131
10.3 Elgamal Signatures 132

10.3.1 Formal Definition . . . 132
10.3.2 Verification . . . 132
10.3.3 Security Considerations . . . 133
10.3.4 Example . . . 133
10.4 Schnorr Signatures . . . 134
10.4.1 Formal Definition . . . 134
10.4.2 Verification . . . 135
10.4.3 Group Realization . . . 135
10.4.4 Security Considerations . . . 135
10.4.5 Example . . . 136
10.4.6 Schnorr Signature Patent . . . 138
10.5 Digital Signature Algorithm (DSA) . . . 138
10.5.1 Formal Definition . . . 138
10.5.2 Verification . . . 138
10.5.3 Security Considerations . . . 139
10.5.4 Example . . . 139
10.6 Problems with DSA . . . 141
10.7 Comparison of Digital Signature Schemes . . . 141
10.7.1 Public Key Length . . . 141
10.7.2 Signature Length . . . 142
10.7.3 Computational Complexity . . . 142
10.7.4 Conclusion . . . 142
10.8 Malleable Signatures . . . 142

11 Elliptic Curve Cryptography 143
11.1 Problems with Subgroups of Finite Fields . . . 143
11.2 Elliptic Curve Diffie-Hellman (ECDH) . . . 144
11.2.1 Discussion . . . 144
11.2.2 Examples . . . 144
11.3 Elliptic Curve DSA (ECDSA) . . . 147
11.3.1 Formal Definition . . . 147
11.3.2 Verification . . . 148
11.3.3 ECDSA Malleability . . . 148
11.4 Problems with ECDSA . . . 149
11.5 Edwards Curve DSA (EdDSA) . . . 149
11.6 Recoverable ECDSA . . . 150
11.6.1 Deriving the Public Key from the ECDSA Signature . . . 150
11.6.2 Recoverable ECDSA in Ethereum . . . 152
11.7 Mnemonic Seed Phrases and BIP-39 . . . 153
11.7.1 The Challenge of Long Passwords . . . 153
11.7.2 Mnemonic Seed Phrases . . . 153
11.7.3 Examples of Mnemonic Seed Phrases with BIP-39 . . . 154

12 Pairing-Based Cryptography **159**
12.1 Cryptography in Ethereum . . . 159
12.2 BLS Signatures . . . 160
12.3 Hash-to-Curve Functions . . . 161
12.3.1 One Insecure Hash-to-Curve Function . . . 161
12.3.2 Nondeterministic Hash-to-Curve Algorithm . . . 161
12.3.3 Initial Hash-to-Curve Discussion . . . 161
12.3.4 Hashing to Finite Fields . . . 162
12.3.5 Deterministic Mapping from Finite Field to Elliptic Curve . . . 162
12.3.6 Ensuring Hash-to-Curve Uniformity . . . 162
12.4 Multi-Signatures . . . 163
12.5 Concluding Discussion . . . 164

13 Zero-Knowledge Proofs **165**
13.1 The Need for Zero-Knowledge Proofs . . . 165
13.2 Group for Examples . . . 165
13.3 Proving knowledge of Discrete Logarithms . . . 166
13.3.1 Knowledge of Discrete Logarithm . . . 166
13.3.2 Knowledge of Two Discrete Logarithms . . . 168
13.3.3 Knowledge of Multiple Discrete Logarithms . . . 172
13.3.4 Knowledge of Linear Combination of Two Discrete Logarithms . . . 172
13.3.5 Knowledge of Linear Combination of Multiple Discrete Logarithms . . . 176
13.4 Concluding Discussion . . . 178

14 Secret Sharing Protocols **179**
14.1 The Need for Secret Sharing Protocols . . . 179
14.2 Shamir's Secret Sharing . . . 180
14.2.1 Setup . . . 180
14.2.2 Secret Reconstruction . . . 180
14.2.3 Examples . . . 181
14.2.4 Discussion . . . 186
14.3 Verifiable Secret Sharing . . . 186
14.4 Distributed Key Generation Overview and Setup . . . 187
14.4.1 DKG Overview . . . 187
14.4.2 DKG Setup . . . 188
14.5 Distributed Key Generation Protocol . . . 188
14.5.1 Registration Phase . . . 189
14.5.2 Share Submission Phase . . . 189
14.5.3 Share Dispute Phase . . . 190
14.5.4 Key Share Submission Phase . . . 192
14.5.5 MPK Submission Phase . . . 193
14.5.6 GPK Submission Phase . . . 193
14.5.7 GPK Dispute Phase . . . 194
14.5.8 Completion Phase . . . 194
14.6 Threshold Signatures . . . 194

14.6.1 Deriving the Master Secret Key from the Group Secret Keys 194
14.6.2 Constructing Valid Group Signatures 196
14.7 Distributed Key Generation Example 197

15 Computational Hardness Assumptions **201**
15.1 Discrete Logarithm Problem 201
15.1.1 Formal Definition 201
15.1.2 General Methods for Solving DLP 201
15.1.3 Discussion 202
15.2 Diffie-Hellman Problems 202
15.2.1 Formal Definitions 202
15.2.2 General Methods for Solving CDH and DDH 202
15.2.3 Additional Variations 203

16 Conclusion **204**
16.1 Suggestions for Further Study 204
16.1.1 General STEM and Non-Mathematicians 204
16.1.2 Mathematicians 205
16.2 Online Resources 205
16.2.1 Online Cryptography Resources 205
16.2.2 Online Mathematics Resources 206
16.3 Other Good Books 206
16.4 Useful Areas of Mathematics and Books 207

A Additional Cryptography **209**
A.1 Construction of Cryptographic Hash Functions 209
A.1.1 Merkle-Damgård Construction 209
A.1.2 Sponge Construction 213
A.2 Applications of Cryptographic Hash Functions 216
A.2.1 Hash-based Message Authentication Code (HMAC) 216
A.2.2 Keccak Message Authentication Code (KMAC) 218
A.2.3 HMAC-based Key Derivation Function (HKDF) 220
A.2.4 Mask Generation Function and Extendable Output Function 224
A.2.5 PBKDF2 227
A.2.6 Hash to Finite Field 229

B Additional Mathematics **231**
B.1 The Real Numbers 231
B.2 Set Theory 231
B.2.1 Additional Set Operations 232
B.2.2 Relationships Between Set Operations 233
B.3 Number Theory 235
B.3.1 Euclidean Division 235
B.3.2 Euclidean Algorithm 237
B.3.3 Extended Euclidean Algorithm 241

B.4 Finite Fields 244
B.4.1 Generating Primitive Elements 244
B.4.2 Quadratic Residues and the Legendre Symbol 250
B.4.3 Square Roots 252

Bibliography **254**

Glossary **266**

List of Algorithms

5.1 Double-and-add formula for elliptic curve scalar multiplication 54

8.1 Compute Merkle Tree root hash 98
8.2 Validate Merkle Proof of Inclusion 98
8.3 Compute Merkle Tree root hash using recursion from [81] 117
8.4 Compute Merkle Tree root hash from [96] 118

14.1 Encryption scheme in the distributed key generation protocol 191
14.2 Discrete logarithm equality proof and validation 192

A.1 Merkle-Damgård construction of hash functions 211
A.2 Sponge construction of hash functions; based on [22, Alg. 1] 214
A.3 Hash-based Message Authentication Code 217
A.4 Keccak Message Authentication Code 218
A.5 Traditional Key Derivation Function 222
A.6 NIST Key Derivation Function (One-Step Key Derivation [4, Section 4]) . . 222
A.7 HMAC-based Key Derivation Function 223
A.8 Mask Generation Function 1 227
A.9 Password-Based Key Derivation Function Version 2 with HMAC 228

B.1 Euclidean Division 237
B.2 Euclidean Algorithm 237
B.3 Extended Euclidean Algorithm 242
B.4 Compute a primitive element of a finite field 245

List of Figures

1.1 xkcd Security 2

2.1 xkcd Security Holes 5

3.1 Set operations 10
3.2 Plot of a parabola 13

5.1 Plot of an ellipse over the reals 40
5.2 Plots of elliptic curves over the reals 1 41
5.3 Plots of elliptic curves over the reals 2 42
5.4 Plots of elliptic curve addition over the reals 45
5.5 Plots of elliptic curves over finite fields 1 48
5.6 Plots of elliptic curves over finite fields 2 49
5.7 Plots showing symmetry of elliptic curves over finite fields 50
5.8 Plots of elliptic curves over various finite fields 52
5.9 Plots of elliptic curve addition over finite fields 53
5.10 Plots of subgroups of elliptic curves over finite fields 57
5.11 Data points and Lagrange Interpolation over the reals 1 64
5.12 Data points and Lagrange Interpolation over the reals 2 65
5.13 Plot of Runge's Phenomenon 66
5.14 Data points and Lagrange Interpolation over finite fields 1 67
5.15 Data points and Lagrange Interpolation over finite fields 2 68

6.1 xkcd Cryptography 75
6.2 xkcd RNG 77
6.3 xkcd D65536 78

7.1 Example Files Producing SHA-1 Collision 91

8.1 2-Layer Merkle Tree and Merkle Proof 99
8.2 3-Layer Merkle Tree and Merkle Proof 104
8.3 3-Layer Merkle Tree and Merkle Multi-Proof 107
8.4 3-Layer Merkle Tree and Security 110
8.5 Merkle Tree with 5 leaves using recursive definition from [81]. 113
8.6 Merkle Tree with 6 leaves using recursive definition from [81]. 113
8.7 Merkle Tree with 7 leaves using recursive definition from [81]. 114

8.8 Merkle Tree with 8 leaves using recursive definition from [81]. 114
8.9 Merkle Tree with 5 leaves using definition from [96]. 115
8.10 Merkle Tree with 6 leaves using definition from [96]. 115
8.11 Merkle Tree with 7 leaves using definition from [96]. 116
8.12 Merkle Tree with 8 leaves using definition from [96]. 116
8.13 `xkcd` Encryptic 120
8.14 `xkcd` Password Strength 121
8.15 `xkcd` Password Reuse 122
8.16 `xkcd` Geohashing 123

9.1 `xkcd` Public Key 125

11.1 Plots of public keys and shared secrets for ECDH 145

A.1 Cryptographic hash function built with Merkle-Damgård construction 211
A.2 Cryptographic hash function built with Sponge construction 214

List of Listings

6.1 Encryption with the one-time pad 1 . . . 71
6.2 Encryption with the one-time pad 2 . . . 71

7.1 Output from the MD5 hash function . . . 84
7.2 Output from the SHA-1 hash function . . . 85
7.3 Output from the SHA-2-256 hash function . . . 86
7.4 Output from the SHA-3-256 hash function . . . 87
7.5 Example MD5 Collision from Marc Stevens [126]. Each message is 64 bytes. 89
7.6 Example SHA-1 Collision from [127]. Each file is a valid PDF. . . . 90

8.1 Commitment scheme using MD5 . . . 94
8.2 Hash chain using MD5 . . . 96
8.3 2-Layer Merkle Tree using MD5 . . . 101
8.4 2-Layer Merkle Proof using MD5 . . . 102
8.5 3-Layer Merkle Tree using MD5 . . . 105
8.6 3-Layer Merkle Proof using MD5 . . . 106
8.7 3-Layer Merkle Multi-Proof using MD5 . . . 108
8.8 Security of 3-Layer Merkle Proof using MD5 . . . 111

13.1 Zero-knowledge proof of knowledge of discrete logarithm . . . 169
13.2 Zero-knowledge proof of knowledge of two discrete logarithms . . . 173
13.3 Zero knowledge proof of knowledge of linear combination of two discrete logarithms . . . 177

A.1 HMAC using MD5 . . . 219
A.2 KMAC using SHA-3 . . . 221
A.3 HKDF using HMAC-SHA-2 . . . 225
A.4 HKDF using KMAC-SHA-3 . . . 226

List of Tables

14.1 Ethereum Functions 187

15.1 Key Length Estimates 202

A.1 Comparison of Cryptographic Hash Functions 213

Chapter 1

Introduction

A version of this text may be found online[1].

1.1 Primary Goal

These notes introduce cryptography and the necessary mathematics; the primary focus is Public Key Cryptography with particular emphasis on Elliptic Curve Cryptography and digital signatures. The required mathematical maturity is not too great provided the end goal is to gain a deeper understanding of cryptography. Ideally, after working through this material, the reader will understand the pseudocode of cryptographic algorithms and the ideas behind them. We note, however, that the discussion here is *woefully insufficient* to *implement* cryptographic algorithms; even so, we make an effort to point out problems that arise in practice. References are provided throughout for the inquisitive reader.

We note that intra-document links are blue while external links are magenta. For instance, see Figure 1.1 for an `xkcd` webcomic discussing security that may be found online at https://xkcd.com/538/.

These notes may also be helpful to understand the cryptography in the AliceNet Whitepaper[2] [58]. *Full disclosure:* the author of the present work helped write [58].

1.2 Intended Audience

This document is written for individuals without a significant background in mathematics. The necessary mathematics for Public Key Cryptography involves number theory and algebra (group theory, field theory, and elliptic curves). The main requirement for learning this material is dedication.

Although this is primarily aimed at software engineers interested in cryptography, other technically-inclined individuals may find these notes helpful; that is, the material presented here is meant to be accessible to those who studied mathematics, science, or engineering at the collegiate level. Individuals who do not meet this criterion are still encouraged to continue reading if the material is deemed sufficiently interesting.

[1] https://github.com/chgorman/notes-math-crypto

[2] https://github.com/alicenet/whitepaper

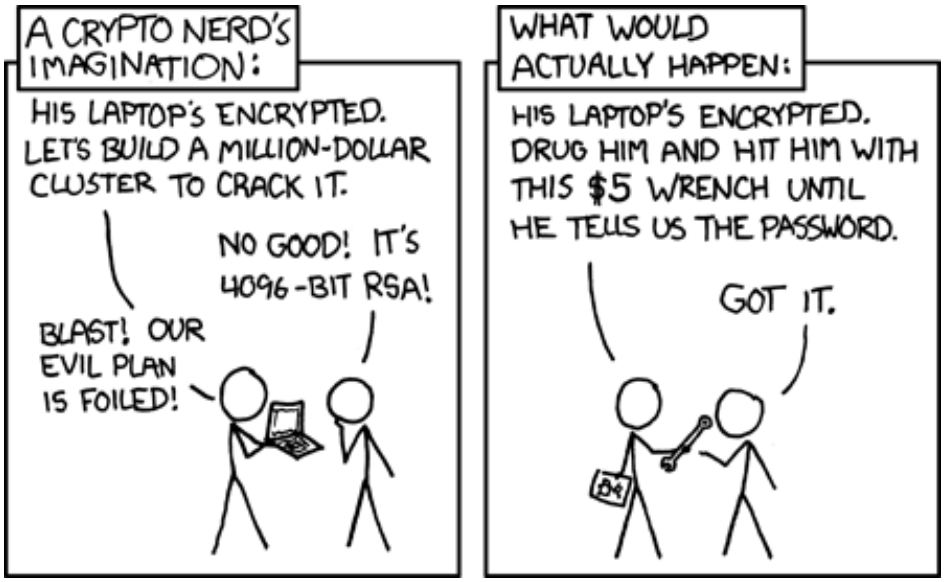


Figure 1.1: Here we have an example of cryptography in the wild. To defend against rubber-hose cryptanalysis, see [26]. Created by Randall Munroe on `xkcd`; posted online at https://xkcd.com/538/.

1.3 Examples

Concrete examples will be used throughout to assist comprehension. `Python` scripts are included that may easily be modified; see `examples/` and its subdirectories.

We note that `code/` stores code for printed examples throughout the text and is not meant to be modified; this is separate from `examples/`, which is designed to be modified.

1.4 Overview

In Chapter 2, we focus on "what not to do". This includes many bad ideas. The main takeaway is this: **do not write your own cryptographic algorithms or protocols**.

We start by introducing the mathematics used throughout these notes in Chapters 3, 4, and 5; the material becomes progressively more abstract. In Chapter 5, we discuss elliptic curves, bilinear pairings, and Lagrange Interpolation; this may be a bit advanced, so on the initial reading this chapter may be skipped and read at a future point in time.

After the mathematical review, we spend some time discussing Symmetric Key Cryptography in Chapter 6. This material is probably more familiar: Alice and Bob share a secret key and want to communicate. We spend significant time talking about cryptographic hash functions; hash functions are first discussed in Chapter 7 while their applications are discussed in Chapter 8.

At this point, we focus on Public Key Cryptography in Chapter 9. In this case, Alice and Bob use 2 keys to communicate: one public key and one private key. We spend Chapter 10 discussing an important aspect of Public Key Cryptography: digital signatures. After this, we look at Elliptic Curve Cryptography in Chapter 11; this involves reworking material from

Chapters 9 and 10 in terms of elliptic curves.

Starting with Chapter 12 on Pairing-Based Cryptography, the material becomes more advanced. After this, we discuss zero-knowledge proofs in Chapter 13. To wrap up, we end with a discussion of secret sharing protocols and distributed key generation in Chapter 14; the discussion here draws on material from essentially *all* of the previous chapters.

In Chapter 15, we discuss some of the hardness assumptions in Public Key Cryptography. If certain mathematical problems are hard to solve, then the cryptographic protocols discussed here are secure.

1.5 Cryptographic Participants

We will frequently encounter Alice and Bob. Alice and Bob will try to communicate without Eve (an adversarial eavesdropper) determining what is being sent. Other characters such as Charlie and Dave may pop up occasionally as well. More characters may be found online[3].

1.6 Mathematical Precision

In general, the author will try to be precise. With that said, this is meant to be an *introduction* to cryptography, so definitions will not always be as precise as possible. Care will be taken to refrain from using "impossible" unless it actually is. For instance, if a one-time pad is used and the secret key is uniformly random, then it is *impossible* to decrypt the message without the secret key. This is perfect security: no amount of computational power will enable the one-time pad to be broken. We will generally use "impractical" to describe situations where a significant amount of computational effort is required; for instance, breaking a system may require performing 2^{128} operations, which is thought to be impractical.

In most cases, only the material *required* is mentioned along with some related discussion. At times, additional material is included in the appendices: additional cryptography may be found in Appendix A while additional mathematics may be found in Appendix B. This material was deemed interesting or useful enough to be included for the curious reader even if it is not *strictly necessary*.

1.7 Additional Resources

All of the material here may easily be gathered from other books and sources. The author used many books to learn cryptography himself. Specific paths for additional learning are discussed in Chapter 16.

[3] https://en.wikipedia.org/wiki/Alice_and_Bob

Chapter 2

What Not to Do

This chapter is devoted to listing a bunch of *horrible* things to do; we list them here precisely because you should *never* do any of these things.

2.1 General Advice

- **Do not "roll your own crypto."** It is always a good idea to use software that professional cryptographers have written. It is a bad idea to write your own encryption scheme, digital signature scheme, hash function, ... Use something written by experts. That library should have been written by someone with many years of experience in cryptography and it should have been vetted by other people as well.

  Along those lines is a quote[1] from Bruce Schneier in 1998:

  > Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break. It's not even hard. What is hard is creating an algorithm that no one else can break, even after years of analysis.

  Bad things can happen when you do not know what you are doing; see Figure 2.1 for an example.

- **Do not randomly combine cryptographic primitives to arrive at something you think solves the problem.** Take the time to think about *exactly* what you are trying to do. If you are wanting to store passwords, then use a using a key derivation function designed for password storage and not a standard cryptographic hash function; both are useful but serve different purposes.

- **Whenever possible, do not use randomized algorithms.** There are many problems which may arise when using random number generators. Thus, whenever possible, use deterministic algorithms. There may be times when algorithms requiring randomness can be reworked to use "deterministic randomness" based on cryptographic hash functions. This would be the preferred method; see [102] for a reference to "deterministic randomness" in digital signatures.

[1] https://www.schneier.com/crypto-gram/archives/1998/1015.html#cipherdesign

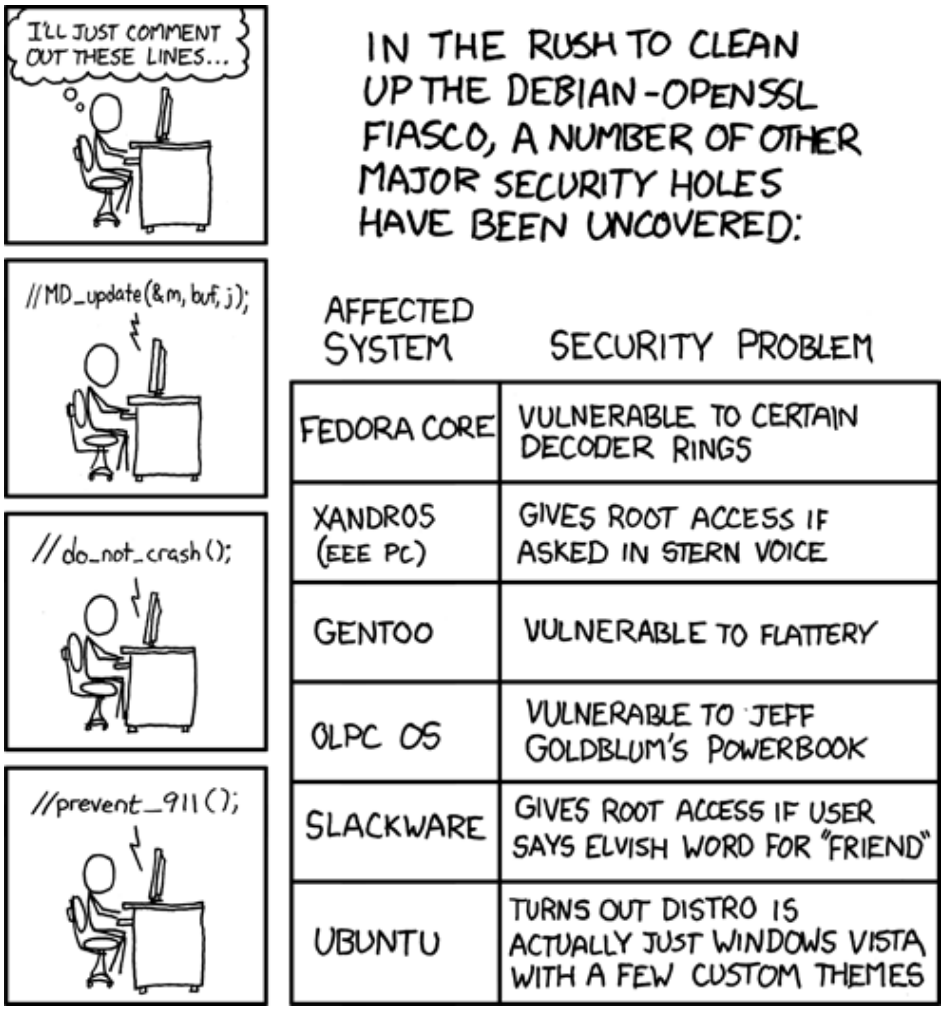


Figure 2.1: Here is an example of the fragility of cryptography in the real world. Created by Randall Munroe on `xkcd`; posted online at https://xkcd.com/424/.

- **Do not use classical algorithms.** If any real security is required, do not *ever* use classical algorithms like substitution or transposition ciphers. All classical ciphers can easily be brute-forced and may be broken given enough ciphertext.
- **The only encryption scheme with perfect security is the one-time pad.** Any perfectly secure encryption scheme is equivalent to the one-time pad [69, Theorems 2.10 and 2.11]. Anyone who claims to have a perfectly secure algorithm that is better than the one-time pad is *lying*.
- **Do not use "security through obscurity" or assume a secret algorithm will be sufficient for security.** This goes against *Kerchoff's principle* [69, Page 5]:

  > The cipher method must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience.

2.2 Symmetric Key Encryption

- **Do not use ECB-mode when encrypting with a block cipher.** ECB stands for *Electronic Codebook*. This method should *never* be used when performing encryption with a block cipher. It leaks too much information; see Chapter 6.3.2 for more information.
- **Do not use DES.** DES stands for *Data Encryption Standard* and is a block cipher. The small key size allows it to be easily broken [70]. The *only* reason to ever use it is because its use is required for compatibility with legacy systems.
- **Do not use RC4.** RC4 is a cryptographically-broken stream cipher [101]; see Chapter 6.3.1 for more information.
- **Do not use Blowfish.** Blowfish is an old block cipher, and in 2007 its author recommended[2] switching from Blowfish [116] to Twofish [117].

2.3 Cryptographic Hash Functions

- **Do not use MD5.** MD5 is a cryptographically-broken hash function [128]. The *only* reason to ever use it is because its use is required for compatibility with legacy systems.
- **Do not use SHA-1.** See the reasons above about why MD5 should not be used [100].
- **Do not use PBKDF2 as a key derivation function.** It has been shown to provide insufficient protection when protecting passwords [25].
- **Do not use `bcrypt` as a key derivation function.** It has also been shown to provide insufficient protection when protecting passwords [25].

2.4 Cryptographically-Secure Pseudorandom Number Generators

- **Do not use Dual_EC_DRBG.** The standard algorithm [7, Section 10.3.1] is believed to have a backdoor [21] and was removed in a later version [8].
- **Do not use non-cryptographic PRNGs for cryptographic protocols.** Cryptographic protocols *require* cryptographically-strong pseudorandom numbers; these numbers come from a cryptographically-secure pseudorandom number generator. The numbers provided by non-cryptographic PRNGs *are not sufficiently random* [31, 84].

  Here is a non-exhaustive list of non-cryptographic PRNGs: Linear Congruential Generator (LCG), Permuted Congruential Generator (PCG), Mersenne Twister. *None* of these PRNGs should *ever be used* in cryptographic situations.

[2] https://www.schneier.com/news/archives/2007/12/bruce_almighty_schne.html

2.5 Public Key Encryption

- **Do not use publish the private key.** This should go without saying.

2.6 Digital Signatures

- **Do not use publish the signing key.** This should go without saying.

Chapter 3

Mathematical Review: Set Theory, Number Theory, and Complexity

In this chapter we begin our mathematical review. Depending on one's background, it may be useful to skip these chapters and return to them as needed.

While the information here and in the following chapters will continually get more abstract, we make an effort to include numerous *concrete examples*. Furthermore, we will attempt to point out *why* we need the various mathematical objects as they arise.

3.1 Mathematical Notation, Set Theory, and Functions

We begin by reviewing standard mathematical notation, the basics of set theory, and functions.

3.1.1 Introduction to Set Theory

We use "$:=$ " when making definitions. Thus,

$$a := b \tag{3.1}$$

means that we *define* a as b and may be read as "a is (defined to be) b".

For our purposes, a *set* is a collection of elements or objects. We suppose that A is a set. If a is an element of A, we write $a \in A$; this may be read as "a is an element of A" or "a is in A". If a is not an element of A, we write $a \notin A$; this may be read as "a is not an element of A" or "a is not in A". We will always have that $a \in A$ or $a \notin A$ for all sets A and elements a. The set which contains no elements $\{\}$ is called the *empty set* and is denoted by $\emptyset$.

If the sets A and B contain the same elements, then they denote the same set and we write $A = B$; this may be read as "A equals B". Otherwise, A and B do not contain the same elements, and we write $A \neq B$; this may be read as "A does not equal B".

If B is a set and for every $b \in B$ we have $b \in A$, then B is a subset of A and we write $B \subseteq A$. We may read $B \subseteq A$ as "B is a subset of A". If B is a subset of A and B is not equal to A, then B is a proper subset of A; in this case, $B \subseteq A$ and there is some $a \in A$

such that $a \notin B$. We may write this as $B \subset A$; this may be read as "B is a proper subset of A". Given any set A, we always have that the empty set is a subset of A: $\emptyset \subseteq A$.

Starting with one set, we would like to make other sets. Let X be a set and P is some property that elements of X may or may not have. If we want to define a new set A as the set of all elements in X which have property P, we would write

$$A \coloneqq \{x \in X \mid P\}\,. \tag{3.2}$$

This is "set builder" notation. We should read $A \coloneqq \{x \in X \mid P\}$ as "A is (defined to be) the set of all x in X such that (property) P is true". Here, we use "$\mid$" as a separator that should be read as "such that." When using set builder notation, the overall set X may not always be explicitly stated.

Within set builder notation, we may use "$\exists$" as shorthand for "there exists" or "there is"; we may also use "$\forall$" as shorthand for "for all". Outside of set builder notation, we will try to not use this shorthand.

We will be interested in operations between sets, and we will want to combine them in various ways. Let A and B be sets. We have the following definitions:

$$\begin{aligned} A \cup B &\coloneqq \{x \mid x \in A \text{ or } x \in B\} \\ A \cap B &\coloneqq \{x \mid x \in A \text{ and } x \in B\} \\ A \setminus B &\coloneqq \{a \in A \mid a \notin B\}\,. \end{aligned} \tag{3.3}$$

Thus, $A \cup B$ contains elements that are in A or B; this should be read "A union B". $A \cap B$ is the set containing elements that are in both A and B; this should be read "A intersect B". $A \setminus B$ is the set containing elements in A that are not in B; this could be read "A set minus B" or "A minus B". See Figure 3.1 for a graphical representation of these set operations.

Example 3.1 (Examples of Sets and Set Operations)
We have the following definitions:

$$\begin{aligned} A &\coloneqq \{0, 1, 2, 3\} \\ B &\coloneqq \{1, 2, 3, 4\} \\ C &\coloneqq \{1, 2, 3\} \\ D &\coloneqq \{1, 3, 5, 7\}\,. \end{aligned} \tag{3.4}$$

Then we see

$$\begin{aligned} A \cup B &= \{0, 1, 2, 3, 4\} \\ A \cap B &= \{1, 2, 3\} \\ &= C \\ A \setminus B &= \{0\}\,. \end{aligned} \tag{3.5}$$

We also note that all of the elements in C are also in A and B; this implies that

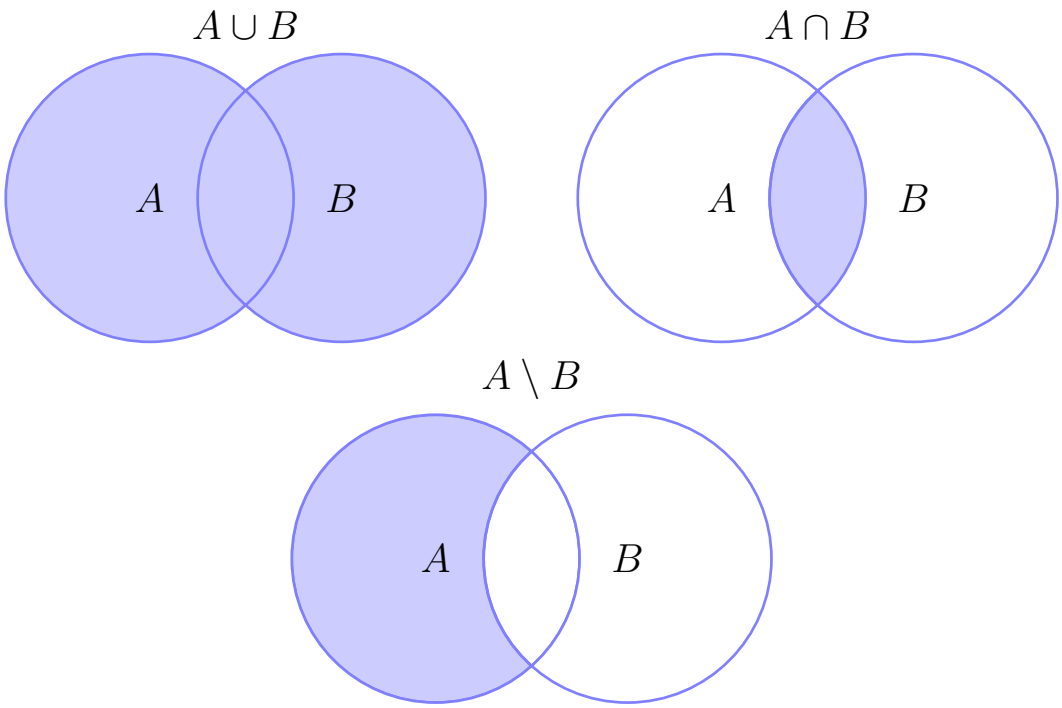


Figure 3.1: Here is a figure of various set operations: set union, set intersection, and set minus. Modified from an example online [137].

$$\begin{aligned} C &\subseteq A \\ C &\subseteq B. \end{aligned} \tag{3.6}$$

Because $C \neq A$ and $C \neq B$, we have

$$\begin{aligned} C &\subset A \\ C &\subset B. \end{aligned} \tag{3.7}$$

Finally, we have

$$\begin{aligned} (A \cup C) \cup D &= \{0, 1, 2, 3, 5, 7\} \\ (A \cap C) \cap D &= \{1, 3\} \\ C \setminus D &= \{2\} \\ C \setminus B &= \emptyset. \end{aligned} \tag{3.8}$$

A further discussion of set theory may be found in Appendix B.2.

3.1.2 Standard Mathematical Sets

We have the following standard definitions for common collections of numbers:

$$\begin{aligned}
\mathbb{N} &\coloneqq \{0, 1, 2, \ldots\} \\
\mathbb{Z} &\coloneqq \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
\mathbb{Q} &\coloneqq \{a/b \mid a, b \in \mathbb{Z}, b \neq 0\} \\
\mathbb{R} &\coloneqq \{\text{the set of all real numbers}\} \\
\mathbb{C} &\coloneqq \{a + bi \mid a, b \in \mathbb{R}\}\,.
\end{aligned} \tag{3.9}$$

Naturally, we have the naturals $\mathbb{N}$, the integers $\mathbb{Z}$, the rational numbers $\mathbb{Q}$, the real numbers $\mathbb{R}$, and the complex numbers $\mathbb{C}$.

We also have the following standard definitions. Not all of these will make sense at this time because not everything has been defined. We will discuss them in more detail in later chapters.

$$\begin{aligned}
n\mathbb{Z} &\coloneqq \{\ldots, -2n, -n, 0, n, 2n, \ldots\}\,, \quad n \geq 1 \\
\mathbb{Z}_n &\coloneqq \{0, 1, 2, \ldots, n-1\}\,, \quad n > 1 \\
\mathbb{Z}_n^* &\coloneqq \{a \in \mathbb{Z}_n \mid \gcd(a, n) = 1\} \\
\mathbb{F}_p &\coloneqq \{0, 1, 2, \cdots, p-1\}\,, \quad p \text{ prime} \\
\mathbb{F}_p^* &\coloneqq \mathbb{F}_p \setminus \{0\} \\
\{0,1\}^n &\coloneqq \{\text{The set of all } n\text{-bit strings}\} \\
\{0,1\}^* &\coloneqq \{\text{The set of all finite bit strings}\}\,.
\end{aligned} \tag{3.10}$$

Example 3.2 (More Examples of Sets and Set Operations)
From the definitions, we see

$$\begin{aligned}
\mathbb{Z} \cup \mathbb{N} &= \mathbb{Z} \\
\mathbb{Z} \cap \mathbb{N} &= \mathbb{N} \\
\mathbb{Z} \setminus \mathbb{N} &= \{\ldots, -3, -2, -1\} \\
\mathbb{Z} \setminus \mathbb{Q} &= \emptyset.
\end{aligned} \tag{3.11}$$

Additionally, for all sets A, we always have

$$\begin{aligned}
A \cup \emptyset &= A \\
A \cap \emptyset &= \emptyset.
\end{aligned} \tag{3.12}$$

We write $a \xleftarrow{\$} A$ to denote that a is chosen uniformly at random from the set A; in this case, we assume that A has a finite number of elements.

3.1.3 Functions

We will look at functions between sets. Let A and B be sets. A *function* $f : A \to B$ uniquely assigns every element in $a \in A$ to some element in B. More specifically, for all $a \in A$ we have $f(a) \in B$. We should read $f : A \to B$ as "(a function) f from A to B". We say that A is the *domain* of f and B is the *codomain* of f. The *range* of f is defined to be $f(A)$:

$$f(A) \coloneqq \{b \in B \mid \exists a \in A, f(a) = b\}. \tag{3.13}$$

This definition can be read as "$f(A)$ is the set of all $b \in B$ such that there is an $a \in A$ so that $f(a) = b$". In this way, the range is everything that f "hits": if $\bar{b} \in f(A)$, then there is some $\bar{a} \in A$ with $f(\bar{a}) = \bar{b}$.

Example 3.3 (Example of a Function)
We let $A = \{0, 1, 2, 3\}$ and $B = \{1, 2, 3, 4\}$. We define a function $f : A \to B$ as follows:

$$\begin{aligned} f(0) &\coloneqq 1 \\ f(1) &\coloneqq 2 \\ f(2) &\coloneqq 3 \\ f(3) &\coloneqq 2. \end{aligned} \tag{3.14}$$

While we may sometimes think of functions in terms of their graphs like in Figure 3.2, in this case we have fully specified f by assigning every element in A to some element in B; thus, f is a function.

We see that

$$f(A) = \{1, 2, 3\} \subset B. \tag{3.15}$$

We have $1 \in f(A)$ because $f(0) = 1$; $2 \in f(A)$ because $f(3) = 2$; and $3 \in f(A)$ because $f(2) = 3$. Thus, the range $f(A)$ is a proper subset of the codomain B.

Example 3.4 (Another Example of a Function)
We now have a more standard example of a function. We recall that $\mathbb{R}$ denotes the set of real numbers.

Let $f : \mathbb{R} \to \mathbb{R}$ and set $f(x) = x^2$. In this case, the domain of the function is $\mathbb{R}$ and the codomain of the function is $\mathbb{R}$. In this case, the range $f(\mathbb{R}) = [0, \infty)$. We know the graph of f is the standard parabola; see Figure 3.2.

3.1.4 Function Properties

We now list some standard properties that functions may have:

- A function $f : A \to B$ is *injective* or *one-to-one* if $f(x) = f(y)$ implies that $x = y$. Equivalently, f is injective if $x \neq y$ implies that $f(x) \neq f(y)$.
- A function $f : A \to B$ is *surjective* or *onto* if for every $b \in B$ there is an $a \in A$ such that $f(a) = b$. This means that the range is equal to the codomain: $f(A) = B$.

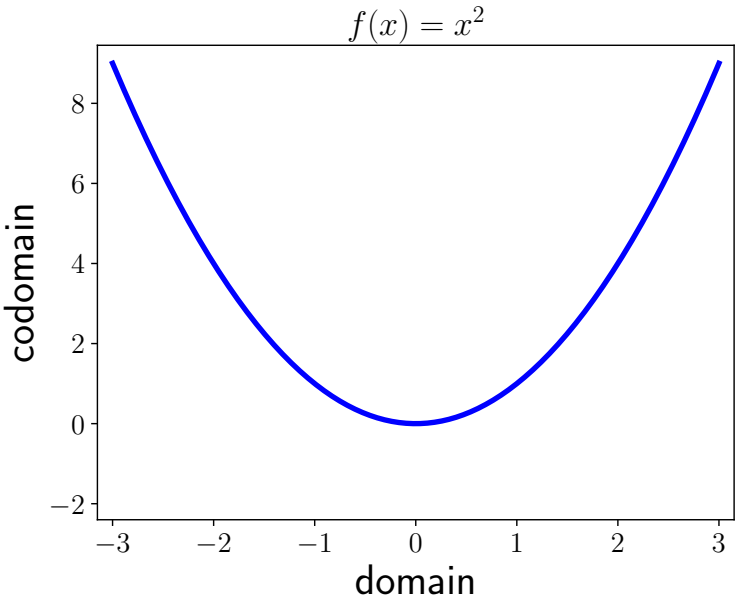


Figure 3.2: Here we have a plot of the function $f : \mathbb{R} \to \mathbb{R}$ with $f(x) = x^2$: the standard parabola. The domain of f is the real numbers $\mathbb{R}$; the codomain of f is also $\mathbb{R}$. We have that the range of f is the set of nonnegative real numbers: $f(\mathbb{R}) = [0, \infty)$.

- A function $f : A \to B$ is *bijective* (or f is a *bijection*) if f is both injective and surjective. In this case, f has an *inverse function* $g : B \to A$ which satisfies

$$g(f(a)) = a \tag{3.16}$$

and

$$f(g(b)) = b \tag{3.17}$$

for all $a \in A$ and $b \in B$.

Example 3.5
The function f in Example 3.3 is not injective, surjective, or bijective.

- f is not injective because $f(1) = f(3) = 2$ but $1 \neq 3$.
- f is not surjective because no element is mapped to $4 \in B$.
- f is not bijective because it is not injective and surjective.

In preparation for more examples, we define the following functions:

$$\begin{aligned} f &: \mathbb{N} \to \mathbb{N}, \quad f(n) \coloneqq n + 1 \\ g &: \mathbb{Z} \to \mathbb{Z}, \quad g(n) \coloneqq n + 1 \\ h &: \mathbb{R} \to \mathbb{R}, \quad h(x) \coloneqq x^2. \end{aligned} \tag{3.18}$$

Example 3.6 (Injective Functions 1: Example)
We use the function f as defined in Eq. (3.18).

If we have $n, m \in \mathbb{N}$ with

$$f(n) = f(m), \tag{3.19}$$

this implies that

$$n + 1 = m + 1, \tag{3.20}$$

so $n = m$. Thus, f is injective.

Example 3.7 (Injective Functions 2: Example)
We use the function g as defined in Eq. (3.18).

Using the same argument from Example 3.6, we see that g is injective.

Example 3.8 (Injective Functions 3: Non-example)
We use the function h as defined in Eq. (3.18).

For $x > 0$, we see that $f(x) = f(-x)$ with $x \neq -x$. In particular, we have $f(1) = f(-1) = 1$ but $1 \neq -1$. Thus, h is not injective.

Example 3.9 (Surjective Functions 1: Non-example)
We see that f is not surjective, because $0 \in \mathbb{N}$ and there is no $n \in \mathbb{N}$ such that $f(n) = 0$.

Example 3.10 (Surjective Functions 2: Example)
Given $n \in \mathbb{Z}$, we see that $n - 1 \in \mathbb{Z}$ and

$$g(n - 1) = n. \tag{3.21}$$

Thus, g is surjective.

Example 3.11 (Surjective Functions 3: Non-example)
We see that h is not surjective. This is because we know that for $x \in \mathbb{R}$, we have

$$h(x) = x^2 \geq 0. \tag{3.22}$$

In particular, $-1 \in \mathbb{R}$ yet there is no $x \in \mathbb{R}$ such that $h(x) = -1$.

Example 3.12 (Bijective Functions 1: Non-example)
In Example 3.6 we showed that f is injective while Example 3.9 showed that f is not surjective; thus, f is not bijective.

Example 3.13 (Bijective Functions 2: Example)
In Example 3.7 we showed that g is injective while Example 3.10 showed that g is surjective; thus, g is bijective.

Example 3.14 (Bijective Functions 3: Non-example)
In Example 3.8 we showed that h is not injective while Example 3.11 showed that h is not surjective; thus, h is not bijective.

3.1.5 Permutations

Permutations are a specific type of function; in particular, permutations are bijections from a set to itself. More formally, if A is a set, the function $f : A \to A$ is a *permutation* if f is bijective.

Example 3.15 (Permutations: Examples)
We let $A = \{1, 2, 3, 4\}$.
We define $f : A \to A$ by

$$\begin{aligned} f(1) &\coloneqq 2 \\ f(2) &\coloneqq 3 \\ f(3) &\coloneqq 4 \\ f(4) &\coloneqq 1. \end{aligned} \tag{3.23}$$

Here, we see that f is a permutation.
Similarly, we can define $g : A \to A$ by

$$\begin{aligned} g(1) &\coloneqq 4 \\ g(2) &\coloneqq 1 \\ g(3) &\coloneqq 2 \\ g(4) &\coloneqq 3. \end{aligned} \tag{3.24}$$

Here, g is also a permutation. We can see that

$$\begin{aligned} f(g(1)) &= 1 \\ f(g(2)) &= 2 \\ f(g(3)) &= 3 \\ f(g(4)) &= 4. \end{aligned} \tag{3.25}$$

Naturally, this formally shows that f and g are inverses.

Example 3.16 (Permutations: Non-example)
As before, we let $A = \{1, 2, 3, 4\}$.
We define $h : A \to A$ by

$$\begin{aligned} h(1) &\coloneqq 2 \\ h(2) &\coloneqq 3 \\ h(3) &\coloneqq 4 \\ h(4) &\coloneqq 4. \end{aligned} \tag{3.26}$$

We see that h is not a permutation because it is not a bijection. In particular, h is not injective because $3, 4 \in A$ with $h(3) = h(4)$ yet $3 \neq 4$.

3.2 Bit Operations

Here we include a brief discussion of bit operations for the sake of completeness.

3.2.1 Definitions

Each bit is either `0` or `1`.

NOT *NOT* reverses a bit:

$$\begin{aligned} \neg \mathtt{0} &= \mathtt{1} \\ \neg \mathtt{1} &= \mathtt{0}. \end{aligned} \tag{3.27}$$

AND *AND* operates on two bits and returns `1` if both bits are `1` and returns `0` otherwise:

$$\begin{aligned} \mathtt{0} \wedge \mathtt{0} &= \mathtt{0} \\ \mathtt{1} \wedge \mathtt{0} &= \mathtt{0} \\ \mathtt{0} \wedge \mathtt{1} &= \mathtt{0} \\ \mathtt{1} \wedge \mathtt{1} &= \mathtt{1}. \end{aligned} \tag{3.28}$$

OR *OR* operates on two bits and returns `1` if at least one bit is `1` and returns `0` otherwise:

$$\begin{aligned} \mathtt{0} \vee \mathtt{0} &= \mathtt{0} \\ \mathtt{1} \vee \mathtt{0} &= \mathtt{1} \\ \mathtt{0} \vee \mathtt{1} &= \mathtt{1} \\ \mathtt{1} \vee \mathtt{1} &= \mathtt{1}. \end{aligned} \tag{3.29}$$

XOR *XOR* operates on two bits and returns `1` if the two bits are different and returns `0` otherwise:

$$\begin{aligned} \mathtt{0} \oplus \mathtt{0} &= \mathtt{0} \\ \mathtt{1} \oplus \mathtt{0} &= \mathtt{1} \\ \mathtt{0} \oplus \mathtt{1} &= \mathtt{1} \\ \mathtt{1} \oplus \mathtt{1} &= \mathtt{0}. \end{aligned} \tag{3.30}$$

The XOR operation is important because of the following property:

$$(\mathtt{x} \oplus \mathtt{y}) \oplus \mathtt{y} = \mathtt{x}. \tag{3.31}$$

for all bits `x` and `y`. We also have

$$\mathtt{x} \oplus \mathtt{0} = \mathtt{x}. \tag{3.32}$$

3.2.2 Bit Operations on Integers

Using an unsigned integer's binary representation, these bit operations can be extended to unsigned integers; the extension is straightforward and we only give examples.

Example 3.17 (Bitwise Operations)
We show some examples of bit operations using 8-bit unsigned integers:

- Bitwise NOT:

$$\begin{aligned} &\neg\texttt{11110010} = \texttt{00001101} \\ &\neg\texttt{11010011} = \texttt{00101100} \\ &\neg\texttt{00010110} = \texttt{11101001} \\ &\neg\texttt{10111011} = \texttt{01000100}. \end{aligned} \tag{3.33}$$

- Bitwise AND:

$$\begin{aligned} &\texttt{11110010} \wedge \texttt{11010011} = \texttt{11010010} \\ &\texttt{00010110} \wedge \texttt{10111011} = \texttt{00010010}. \end{aligned} \tag{3.34}$$

- Bitwise OR:

$$\begin{aligned} &\texttt{11110010} \vee \texttt{11010011} = \texttt{11110011} \\ &\texttt{00010110} \vee \texttt{10111011} = \texttt{10111111}. \end{aligned} \tag{3.35}$$

- Bitwise XOR:

$$\begin{aligned} &\texttt{11110010} \oplus \texttt{11010011} = \texttt{00100001} \\ &\texttt{00010110} \oplus \texttt{10111011} = \texttt{10101101}. \end{aligned} \tag{3.36}$$

  In general, we also have

$$(x \oplus y) \oplus y = x \tag{3.37}$$

  and

$$x \oplus 0 = x \tag{3.38}$$

  for all unsigned integers x and y.

3.3 Size of Numbers

In Public Key Cryptography (which we discuss in Chapter 9), we will need numbers to be of various magnitudes so that certain mathematical problems are deemed sufficiently difficult; the particular sizes are discussed in Chapter 15 when we talk about computational hardness assumptions. We frequently represent integers in binary expansion, so we use that to describe their sizes here.

Given $a \in \mathbb{N}$, we say that a is a k-bit number if $2^{k-1} \leq a < 2^k$; that is, the largest set bit in a's binary expansion is in the kth position.

Example 3.18 (Size of Numbers)
Code may be found at `examples/math_review/size_of_numbers.py`.

Here are some good order-of-magnitude estimates to keep in mind:

$$\begin{aligned} 2^{32} &\simeq 4 \cdot 10^9 \\ 2^{64} &\simeq 2 \cdot 10^{19} \\ 2^{128} &\simeq 3 \cdot 10^{38} \\ 2^{256} &\simeq 1 \cdot 10^{77} \\ 2^{512} &\simeq 1 \cdot 10^{154} \\ 2^{1024} &\simeq 2 \cdot 10^{308}. \end{aligned} \tag{3.39}$$

For reference, we note that the total number of particles in the known universe is estimated to be 10^{80}. Additionally, a standard year has $31536000 \simeq 2^{25}$ seconds in it.

When working with cryptography in practice, we may seek a certain level of security. For instance, we may want "128-bit security". This means that we want the most efficient algorithm for breaking the underlying cryptosystem to require *at least* 2^{128} operations. As we can see, this is a large number. Analogously, having k-bit security amounts to requiring the most efficient algorithm must perform at least 2^k operations.

Example 3.19 (How large is 2^{128}?)
Let us suppose that we have P processors performing F operations per second and run those processors for S seconds. The total number of operations T that will be performed in that amount of time will be

$$T = P \cdot F \cdot S \quad \text{operations.} \tag{3.40}$$

We now put in some concrete numbers. Assume $P = 2^{30}$ processors (approximately one billion) performing $F = 2^{30}$ operations per second (1 GHz) for one year $S = 2^{25}$. Then we have

$$\begin{aligned} T &= 2^{30} \cdot 2^{30} \cdot 2^{25} \\ &= 2^{85} \quad \text{operations.} \end{aligned} \tag{3.41}$$

At this rate, to perform 2^{128} operations would require $2^{128-85} = 2^{43} \simeq 9 \cdot 10^{12}$ years. It would take almost 10 trillion years at current capabilities for perform 2^{128} operations.

While we should never try to estimate how computing will advance in the future, it is clear that performing 2^{128} operations is a daunting task. Presumably, this is infeasible for at least the near future.

Converting between Powers of 2 and Powers of 10

We have the following useful approximations:

$$\begin{aligned} 10 &\simeq 2^3 \\ 100 &\simeq 2^7 \\ 1000 &\simeq 2^{10}. \end{aligned} \tag{3.42}$$

These approximations are slightly better:

$$\begin{aligned} 10 &\simeq 2^{3.3} \\ 100 &\simeq 2^{6.6} \\ 1000 &\simeq 2^{10}. \end{aligned} \tag{3.43}$$

Even so, these approximations are probably not needed in practice, especially if we just care about order-of-magnitude *estimates*.

Based on $10^3 = 1000 \simeq 1024 = 2^{10}$, we have the following:

$$\begin{aligned} 2^{10} &\simeq 10^3 \\ 2^{20} &\simeq 10^6 \\ 2^{30} &\simeq 10^9. \end{aligned} \tag{3.44}$$

In general, we have

$$2^{10k} \simeq 10^{3k}. \tag{3.45}$$

Example 3.20 (Power Conversions)
Code may be found at `examples/math_review/power_conversion.py`.

We will now work through some conversions. We see

$$\begin{aligned} 2^{86} &= 2^{6+10 \cdot 8} \\ &= 2^6 \cdot 2^{10 \cdot 8} \\ &\simeq 100 \cdot 10^{3 \cdot 8} \\ &= 10^{26}. \end{aligned} \tag{3.46}$$

A more exact approximation is

$$2^{86} \simeq 7.7 \cdot 10^{25}. \tag{3.47}$$

Thus, 10^{26} is a good order-of-magnitude estimate.

Similarly,

$$\begin{aligned} 10^{86} &= 10^{2+3\cdot 28} \\ &= 100 \cdot 10^{3\cdot 28} \\ &\simeq 2^7 \cdot 2^{280} \\ &= 2^{287}. \end{aligned} \tag{3.48}$$

We see that

$$\log_2(10^{86}) \simeq 285.7. \tag{3.49}$$

Thus, our approximation was not too far off: we are within a factor of 4.

If instead we use the more accurate approximation $100 \simeq 2^{6.6}$, then we have

$$10^{86} \simeq 2^{286.6}. \tag{3.50}$$

This is within a factor of 2.

3.4 Number Theory

We review some basic facts of number theory related to divisibility, prime numbers, and modular arithmetic. Additional material may be found in Appendix B.3. A more systematic treatment may be found in [119].

3.4.1 Divisibility and Prime Numbers

Given $a, b \in \mathbb{Z}$, we say a divides b, written as $a \mid b$, when $b = ca$ for $c \in \mathbb{Z}$; in this case, we call a a *divisor* of b. We write $a \nmid b$ when a does not divide b and say a is not a divisor of b. Because $a = 1 \cdot a$, we see that $1 \mid a$ and $a \mid a$; thus, 1 and a are always divisors of a.

If $p \in \mathbb{N} \setminus \{0, 1\}$ and we *only* have $1 \mid p$ and $p \mid p$, then we say that p is a *prime number*. We frequently use p and q to denote prime numbers. If $n \in \mathbb{N} \setminus \{0, 1\}$ and n is not prime, then we say that n is *composite*.

By convention, mathematicians do not consider 0 or 1 to be prime or composite numbers.

Example 3.21 (Prime and Composite Numbers)
We know that $15 = 3 \cdot 5$, so 15 is a composite number. We know that we only have $3 = 1 \cdot 3$ and $5 = 1 \cdot 5$, so 3 and 5 are prime numbers.

3.4.2 Greatest Common Divisor

Given $a, b \in \mathbb{Z}$ (with $a \neq 0$ or $b \neq 0$), $d \geq 0$ is the *greatest common divisor* of a and b, written as $\gcd(a, b) = d$, if d is a divisor of a and b and that if $r \geq 0$ is also a divisor of a and b then $r \leq d$. When $\gcd(a, b) = 1$, we say that a and b are *coprime*: they have no common divisors. Stated another way, coprime numbers have no common prime factors.

There are efficient algorithms to compute $\gcd(a, b)$; see Appendix B.3 for more details. One inefficient method involves computing the prime factorization of a and b and then collecting the common factors.

Example 3.22 (GCD Examples)
Code may be found at `examples/math_review/gcd.py`.

We have $3 = 1 \cdot 3$ and $5 = 1 \cdot 5$; this implies

$$\gcd(3, 5) = 1. \tag{3.51}$$

We know $15 = 3 \cdot 5$; this implies that

$$\begin{aligned} \gcd(15, 5) &= 5 \\ \gcd(15, 3) &= 3. \end{aligned} \tag{3.52}$$

We know $1872 = 2^4 \cdot 3^2 \cdot 13$ and $1080 = 2^3 \cdot 3^3 \cdot 5$; this implies that

$$\begin{aligned} \gcd(1872, 1080) &= 2^3 \cdot 3^2 \\ &= 72. \end{aligned} \tag{3.53}$$

This is worked out in Example B.8.

If $\gcd(a, b) = d$, then we can always find $x, y \in \mathbb{Z}$ such that

$$ax + by = d. \tag{3.54}$$

There are efficient algorithms to compute this; see Appendix B.3 for more details.

Example 3.23
Code may be found at `examples/math_review/gcd.py`.

We continue the previous example for $\gcd(1872, 1080) = 72$. We see

$$1872 \cdot (-4) + 1080 \cdot 7 = 72. \tag{3.55}$$

This is worked out in Example B.9.

3.4.3 Congruent Numbers

Let $n > 1$ be an integer. Given $a, b \in \mathbb{Z}$, we say that a is congruent to b modulo n if $n \mid a - b$; that is, we have $a - b = kn$ for some $k \in \mathbb{Z}$. We also write this as $a \equiv b \mod n$.

Example 3.24 (Examples of Congruent Numbers Mod 2)
We let $n = 2$. In this case, we see that $1 \equiv 11 \mod 2$ because

$$1 - 11 = (-5) \cdot 2. \tag{3.56}$$

Similarly, $2 \equiv 10 \mod 2$:

$$2 - 10 = (-4) \cdot 2. \tag{3.57}$$

Continuing on, we see that all even numbers are congruent modulo 2 and all odd numbers are congruent modulo 2.

We show this now. Let $a = 2k$ and $b = 2\ell$. Then

$$a - b = 2(k - \ell). \tag{3.58}$$

Thus, $a \equiv b \mod 2$. This gives us the result for even numbers. If $c = 2m+1$ and $d = 2n+1$, then we see

$$c - d = 2(m - n). \tag{3.59}$$

Thus, $c \equiv d \mod 2$. This gives us the result for odd numbers.

We just showed that all even numbers are congruent to 0 modulo 2 and all odd numbers are congruent to 1 modulo 2.

3.4.4 Modular Arithmetic

Let $n > 1$ be an integer. Given $a \in \mathbb{Z}$, we can always compute $q, r \in \mathbb{Z}$ such that

$$a = qn + r, \tag{3.60}$$

where q the *quotient* and $0 \leq r < n$ the *remainder*; this is called *Euclidean Division* and is discussed in Appendix B.3. From our previous discussion, we see that

$$a \equiv r \mod n. \tag{3.61}$$

Using this fact, we will show that addition and multiplication are always well-defined in modular arithmetic. From there, exponentiation and other operations follow. The only operation which requires care is division, which we will look at separately.

Addition We let $a_1, a_2 \in \mathbb{Z}$ with

$$\begin{aligned} a_1 &= q_1 n + r_1 \\ a_2 &= q_2 n + r_2, \end{aligned} \tag{3.62}$$

where $q_i \in \mathbb{Z}$ and $0 \leq r_i < n$. Then we see

$$\begin{aligned} a_1 + a_2 &= (q_1 + q_2)\, n + (r_1 + r_2) \\ &= r_1 + r_2 \mod n. \end{aligned} \tag{3.63}$$

This shows that addition behaves as we would expect.

Multiplication We let a_1 and a_2 be as before. Then

$$\begin{aligned} a_1 a_2 &= (q_1 q_2 n + r_1 q_2 + r_2 q_1)\, n + r_1 r_2 \\ &= r_1 r_2 \mod n. \end{aligned} \tag{3.64}$$

This shows that multiplication behaves as we would expect.

Division We need to be careful in order to define division. Within the rational or real numbers, when we say that x is the multiplicative inverse of a, we mean that

$$ax = 1. \tag{3.65}$$

We look for something similar in modular arithmetic. Thus, if $a \in \mathbb{Z} \setminus \{0\}$, then we look for an $x \neq 0$ such that

$$ax \equiv 1 \mod n. \tag{3.66}$$

This implies that there is $k \in \mathbb{Z}$ such that

$$ax - 1 = -nk, \tag{3.67}$$

or rather

$$ax + nk = 1. \tag{3.68}$$

The above equality implies that

$$\gcd(a, n) = 1. \tag{3.69}$$

Thus, if $\gcd(a, n) = 1$, then a has a multiplicative inverse modulo n.

Example 3.25 (Modular Arithmetic Example)
Code may be found at `examples/math_review/modular_arithmetic.py`.

We let $n = 256$. We have

$$\begin{aligned} 27 + 241 &\equiv 12 \mod 256 \\ 27 \cdot 241 &\equiv 107 \mod 256. \end{aligned} \tag{3.70}$$

Because $\gcd(27, 256) = 1$ and $\gcd(241, 256) = 1$, 27 and 241 have multiplicative inverses. We find that

$$\begin{aligned} 27 \cdot 19 &\equiv 1 \mod 256 \\ 241 \cdot 17 &\equiv 1 \mod 256. \end{aligned} \tag{3.71}$$

3.5 Computational Complexity

We review the basics of computational complexity in order to talk about the computational cost of solving certain problems. That is to say, we care about how long certain operations take when we look at asymptotically large values.

3.5.1 Big-O Notation

We begin by reviewing big-O notation. Let $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$ with $f(n) \geq 0$ and $g(n) \geq 0$. We write $f(n) = O(g(n))$ when there is some $C > 0$ and $N \in \mathbb{N}$ such that

$$f(n) \leq Cg(n), \quad n \geq N. \tag{3.72}$$

That is, f is asymptotically bounded above by g.

We write $f(n) = \Omega(g(n))$ when there is some $C > 0$ and $N \in \mathbb{N}$ such that

$$f(n) \geq Cg(n), \quad n \geq N. \tag{3.73}$$

That is, f is asymptotically bounded below by g.

Finally, we write $f(n) = \Theta(g(n))$ when $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. That is, f is asymptotically bounded above and below by g. In general, we are interested in *tightly* bounding functions asymptotically; lose bounds are not very valuable.

A further discussion of Big-O notation may be found in [40, Chapter 3] or [73, Chapter 1.2.11.1].

Example 3.26 (Example of Big-O Notation)
We let $f(n) = n^2 + 10n + 5$ and $g_1(n) = n^3$. Then we see

$$f(n) \leq 20 \cdot g_1(n), \quad n \geq 1. \tag{3.74}$$

Thus, $f(n) = O(g_1(n))$.

We now let $g_2(n) = n^2$. In this case, we have

$$f(n) \leq 16 \cdot g_2(n), \quad n \geq 1. \tag{3.75}$$

Thus, $f(n) = O(g_2(n))$. We see that g_1 is a larger upper bound for f while g_2 matches the growth of f more closely.

We now focus on making the statement "g_2 matches the growth of f more closely" more precise; in particular, we will show that g_2 asymptotically bounds f above and below. We see that

$$f(n) \geq g_2(n), \quad n \geq 1. \tag{3.76}$$

Thus, $f(n) = \Omega(g_2(n))$. Together, we have $f(n) = \Theta(g_2(n))$.

Example 3.27 (Another Example of Big-O Notation)
We consider the function $f(n) = 100\,[\ln n]^2$.

We begin by comparing this function to $g_1(n) = n$. Looking at $N = e^{10} \simeq 2.2 \cdot 10^4$, we have

$$f(N) = 10^4 < 2.2 \cdot 10^4 \simeq g_1(N). \tag{3.77}$$

This holds for all $n \geq N$, so we have $f(n) = O(g_1(n))$. Because g_1 grows much more quickly than f, this bound is not very valuable.

We set $g_2 = [\ln n]^2$. Then we see that

$$f(n) = 100\,[\ln n]^2 \leq 1000\,[\ln n]^2 \tag{3.78}$$

and

$$f(n) = 100\,[\ln n]^2 \geq [\ln n]^2\,. \tag{3.79}$$

These inequalities hold for all n. Thus, we have $f(n) = \Theta(g_2(n))$.

3.5.2 Standard Complexity Classes

We will now discuss some broad complexity classes.

Generally, we think of two classes: polynomial algorithms and exponential algorithms. Polynomial algorithms have time and space requirements bounded above by a polynomial. Exponential algorithms have time and space requirements bounded above by an exponential function.

When working in computational (or algorithmic) number theory, we seek algorithms which are polynomial in the *bits* of n, not polynomial in n. Thus, we seek algorithms with total complexity cost

$$C(n) = O\left([\log n]^c\right) \tag{3.80}$$

for some constant $c > 0$. This is called a *polynomial algorithm*; its run time is bounded above by a polynomial in the bits of n.

If an algorithm has total complexity cost

$$C(n) = O\left(n^c\right) \tag{3.81}$$

for some constant $c > 0$, then the algorithm is *exponential* in the bits of n. This is called an *exponential algorithm*; its run time is bounded above by a function exponential in the bits of n. Because all polynomial algorithms are trivially exponential algorithms, we generally assume the cost is bounded below by an exponential as well. In the case of probabilistic algorithms, the *expected* run time is exponential.

It is possible to talk about superpolynomial algorithms (algorithms which grow faster than any polynomial) and subexponential algorithms (algorithms which grow slower than any exponential) but we will not discuss them at this point. We do note that there are many open

questions in computer science related to the asymptotic complexity of various algorithms. This is related to the computational complexity of certain mathematical problems which we will discuss in Chapter 15.

3.5.3 Complexity of Various Operations

There are polynomial algorithms for addition, multiplication, exponentiation, greatest common divisor, and many others. Currently, there are *no known* polynomial algorithms for integer factorization. The fastest algorithms for factoring integers are subexponential algorithms.

Chapter 4

Mathematical Review: Groups, Rings, and Fields

We now build on our mathematical discussion from Chapter 3. Here, we focus on the algebraic objects known as groups, rings, and fields. These mathematical objects enable us to discuss the specifics of Public Key Cryptography.

To assist in this discussion, we will first begin with intuition and examples before giving the formal definition; we include non-examples as well.

4.1 Groups

4.1.1 Why do we care about Groups?

Groups arise in many different areas. In cryptography, we will encounter them in the Diffie-Hellman Key Exchange and digital signatures. For groups to be used in practice, we will need to encode them; that is, we will need to know how to represent them on a computer.

4.1.2 Intuition and Examples

Groups are mathematical objects which we will frequently encounter. Informally, a group is a generalization of the integers under addition, so we start by looking at them first.

Example 4.1 (Integers under addition)
The standard example of a group is to consider the integers $\mathbb{Z}$ under addition.

Given $a, b \in \mathbb{Z}$, we know that $a + b \in \mathbb{Z}$. This means that the integers are *closed* under addition: adding two integers will always give us another integer.

Addition is also *associative*. That is, for $a, b, c \in \mathbb{Z}$, we have

$$(a + b) + c = a + (b + c)\,. \tag{4.1}$$

In this way, we do not have to worry about which way we add integers; we will always get the same result regardless of order.

There is also a special integer $0 \in \mathbb{Z}$. It is special in that adding 0 to a gives us a:

$$0 + a = a + 0 = a. \tag{4.2}$$

No other integer has this property.

We also know the (additive) inverse of a: $-a$. This is because

$$a + (-a) = 0. \tag{4.3}$$

It also turns out that the (additive) inverse is *unique*.

We note that every integer can be written as repeated additions of 1. If $a \geq 0$, then we have

$$a = \underbrace{1 + 1 + \cdots + 1}_{a \text{ times}} \quad a \geq 0. \tag{4.4}$$

For this to apply to negative numbers, we allow for adding 1's additive inverse -1. If $a < 0$, then

$$a = \underbrace{(-1) + (-1) + \cdots + (-1)}_{|a| \text{ times}} \quad a < 0. \tag{4.5}$$

In this way, 1 *generates* the group of integers under addition.

We also notice that addition is *commutative*: given $a, b \in \mathbb{Z}$, we have

$$a + b = b + a. \tag{4.6}$$

Although not all groups have this property, the ones we care about do.

Example 4.2 (Rationals under addition)
Similar to the integers under addition, the rational numbers $\mathbb{Q}$ under addition also form a group.

Given $a, b \in \mathbb{Q}$ with $a = p/q$ and $b = r/s$, we know

$$\begin{aligned} a + b &= \frac{p}{q} + \frac{r}{s} \\ &= \frac{ps + qr}{qs} \in \mathbb{Q}. \end{aligned} \tag{4.7}$$

This shows us that the rationals are closed under addition.

We know that addition of rationals is associative and commutative. Furthermore, $0 \in \mathbb{Q}$ is the identity element. If $a = \frac{p}{q}$ as above, then the inverse is

$$-a = \frac{-p}{q} \tag{4.8}$$

because

$$\begin{aligned} a + (-a) &= \frac{p}{q} + \frac{-p}{q} \\ &= \frac{pq - pq}{q^2} \\ &= 0. \end{aligned} \tag{4.9}$$

This shows that the rationals under addition form a group.

Example 4.3 (Integers under modular addition)
We now look at another standard example: integers under modular arithmetic. Let $n > 1$. Then

$$\mathbb{Z}_n = \{0, 1, \cdots, n-1\}. \tag{4.10}$$

For $a, b \in \mathbb{Z}_n$, we know there is some $c \in \mathbb{Z}_n$ such that

$$a + b = c \mod n. \tag{4.11}$$

This shows us that $\mathbb{Z}_n$ is closed under addition.

We know $0 \in \mathbb{Z}_n$ and we always have

$$0 + a = a + 0 = a \tag{4.12}$$

for $a \in \mathbb{Z}_n$. Thus, 0 is the additive identity.

Given $a \in \mathbb{Z}_n \setminus \{0\}$, we know

$$a + (n - a) = 0 \mod n, \tag{4.13}$$

Thus, the additive inverse of a is $n - a$; 0 is its own additive inverse. Therefore, we see that $(\mathbb{Z}_n, +)$ is a group.

Example 4.4 (Positive Rationals under multiplication)
Similar to the integers under addition, the positive rational numbers $\mathbb{Q}^+$ under multiplication form a group. In this case, the multiplicative identity is 1.

Example 4.5 (Nonexample: Natural numbers under addition)
We know that the natural numbers under addition are very similar to the integers under addition. We know that we can always add two natural numbers, so that $a, b \in \mathbb{N}$ implies $a + b \in \mathbb{N}$; thus, the naturals are closed under addition. Addition is also associative and commutative.

We also have the identity $0 \in \mathbb{N}$, so

$$a + 0 = a \tag{4.14}$$

for all $a \in \mathbb{N}$.

Unfortunately, we do not have additive inverses; this is because the natural numbers do not include the negative integers. That is, for $a \in \mathbb{N} \setminus \{0\}$, there is no $b \in \mathbb{N}$ such that

$$a + b = 0. \tag{4.15}$$

Thus, $(\mathbb{N}, +)$ is *not* a group.

4.1.3 Formal Definition

Definition 4.1 (Group)**.** A group is a set G together with a binary operation $\cdot$ such that, given $a, b \in G$, $a \cdot b \in G$; that is, G is closed under the binary operation $\cdot$.

The binary operation also satisfies the following properties:

- Associativity: for all $a, b, c \in G$, we have

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)\,. \tag{4.16}$$

- Identity element: there is an element $e \in G$ such that for all $a \in G$,

$$a \cdot e = e \cdot a = a. \tag{4.17}$$

- Inverses: for every $a \in G$ there is a $b \in G$ such that

$$a \cdot b = b \cdot a = e. \tag{4.18}$$

  We henceforth denote the inverse of a as a^{-1}.

Formally, we write that $(G, \cdot)$ is a group. Informally, we write that G is a group when the operation is understood. Additionally, we frequently write ab for $a \cdot b$.

4.1.4 Continued Discussion

Although the generic group operation was listed as "multiplication", we could very well have used addition. We will sometimes refer to some groups as "multiplicative groups" while others as "additive groups"; this just means we use the multiplication sign $\cdot$ or the addition sign $+$ to denote the group operation. Nothing changes, as we can use both symbols, but there are conventions. In additive groups, the identity element is usually denoted 0; in multiplicative groups, the identity element is usually denoted 1.

Because $(\mathbb{Z}, +)$ is a group, we know that groups can have an infinite number of elements. In what follows, we will be particularly interested in *finite groups*: groups that have a finite number of elements. We let $|G|$ denote the order of the group (the number of elements in G).

At times, we may drop explicit reference to the operation if it is understood. For instance, we may refer to $\mathbb{Z}$ as a group, when more formally we should say $(\mathbb{Z}, +)$ is a group in order to reference both the *set* (the integers $\mathbb{Z}$) and the *operation* (integer addition). Even so, this is long, tedious, and painful, so we will, at times, forego the formalities when the operation is clear from the context.

4.1.5 More Definitions

Let G be a group and $H \subseteq G$; more explicitly, $(G, \cdot)$ is a group and H is a subset of G. We say that H is a *subgroup* of G, written as $H \leq G$, if H with the group operation inherited from G is also a group. Every group G has at least two subgroups: G (the entire group) and $\{e\}$ (the group that is just the identity element e). A subgroup is called a *proper subgroup* when $H \leq G$ and $H \neq G$. If H is a proper subgroup, then we may write $H < G$. The group $\{e\}$ is called the *trivial* subgroup.

Example 4.6 (Subgroup Example: $2\mathbb{Z} \leq \mathbb{Z}$)
We let

$$2\mathbb{Z} \coloneqq \{\ldots, -4, -2, 0, 2, 4, \ldots\}; \tag{4.19}$$

that is, $2\mathbb{Z}$ is the set of even integers. Naturally, we have $2\mathbb{Z} \subseteq \mathbb{Z}$. For every $a \in 2\mathbb{Z}$ there is some $b \in \mathbb{Z}$ so that $a = 2b$.

Given $x, y \in 2\mathbb{Z}$, we set $x = 2m$ and $y = 2n$ for $m, n \in \mathbb{Z}$. We then see

$$\begin{aligned} x + y &= 2m + 2n \\ &= 2(m + n) \in 2\mathbb{Z}. \end{aligned} \tag{4.20}$$

Thus, $2\mathbb{Z}$ is closed under addition. Because addition on $\mathbb{Z}$ is associative, addition on $2\mathbb{Z}$ is also associative.

Because $0 \in \mathbb{Z}$ and

$$0 + x = x \in 2\mathbb{Z}, \tag{4.21}$$

we still have the same identity element $0 \in 2\mathbb{Z}$.

We let $x \in 2\mathbb{Z}$ with $x = 2m$ for $m \in \mathbb{Z}$. If $y = -2m$, then $y \in \mathbb{Z}$ and

$$\begin{aligned} x + y &= 2m + (-2m) \\ &= 0 \in 2\mathbb{Z}. \end{aligned} \tag{4.22}$$

Thus, we see that every element in $2\mathbb{Z}$ has an additive inverse within $2\mathbb{Z}$. We have just shown that $(2\mathbb{Z}, +)$ is a group; because $2\mathbb{Z} \subseteq \mathbb{Z}$, $2\mathbb{Z}$ is a subgroup of $\mathbb{Z}$.

In general, for $n \in \mathbb{N} \setminus \{0\}$ we let

$$n\mathbb{Z} \coloneqq \{\ldots, -2n, -n, 0, n, 2n, \ldots\}. \tag{4.23}$$

Then $n\mathbb{Z} \leq \mathbb{Z}$. This is a *proper* subgroup when $n \geq 2$.

For a multiplicative group $(G, \cdot)$ and $g \in G$, we define

$$\langle g \rangle \coloneqq \{g^n \mid n \in \mathbb{Z}\}. \tag{4.24}$$

In this case, we see that $\langle g \rangle$ contains all group elements of the form g^n for $n \in \mathbb{Z}$. This is similar to Eqs. (4.4) and (4.5). Here, we have

$$g^n = \underbrace{g \cdot g \cdots g}_{n \text{ times}} \quad n \geq 0 \tag{4.25}$$

and

$$g^n = \underbrace{g^{-1} \cdot g^{-1} \cdots g^{-1}}_{|n| \text{ times}} \quad n < 0. \tag{4.26}$$

In this case, $\langle g \rangle$ is the subgroup of G which consists of products of g or g^{-1}; this allows us to write $\langle g \rangle \leq G$. We say G is a *cyclic group* if there is a $g \in G$ such that for all $h \in G$ there is an $n \in \mathbb{Z}$ such that $h = g^n$. In this case, we have $G = \langle g \rangle$ and say that g is a *generator* of G. Cyclic groups are important in cryptography because these groups are used in the Diffie-Hellman Key Exchange; they are also used to construct digital signatures.

Example 4.7 (Cyclic Group: $\mathbb{Z}$)
As we saw previously, 1 is a generator of the integers $\mathbb{Z}$ under addition. Thus, we can write $\langle 1 \rangle = \mathbb{Z}$. We note that -1 is also a generator, so $\langle -1 \rangle = \mathbb{Z}$.

Example 4.8 (Non-cyclic Group: $\mathbb{Q}$)
We can consider the rational numbers $\mathbb{Q}$ under addition. For any $a \in \mathbb{Q} \setminus \{0\}$, we know that and $\frac{a}{2} \in \mathbb{Q} \setminus \{0\}$. Even so, we see that $\frac{a}{2} \notin \langle a \rangle$. Thus, $\mathbb{Q}$ is not a cyclic group.

In all of the groups we have looked up to this point, all of our group operations satisfy $a \cdot b = b \cdot a$; that is, the group operation is *commutative*. We say that G is an *abelian group* if for all $a, b \in G$ we have $a \cdot b = b \cdot a$. This need not always be the case, but we will only consider abelian groups here because the groups which normally arise in cryptography are abelian.

We note that addition and multiplication are familiar commutative operations; we always have $a + b = b + a$ and $a \cdot b = b \cdot a$ for real numbers. We know that subtraction and division are *not* commutative; in general, we have $a - b \neq b - a$ and $\frac{a}{b} \neq \frac{b}{a}$.

We will usually denote $a \cdot b$ as ab from now on when there is no cause for confusion.

4.1.6 Encoding Groups

When working with the group $\mathbb{Z}_n$, we can encode $k \in \mathbb{Z}_n$ by its binary representation. Other groups may have different representations.

4.2 Rings

4.2.1 Why do we care about Rings?

Rings arise in many different areas. We use rings to ease our way into the discussion about fields in Chapter 4.3.

4.2.2 Intuition and Examples

Rings are mathematical objects which we will sometimes encounter. They are a generalization of the integers when we look at both addition *and* multiplication of integers. We begin with some examples.

Example 4.9 (Integers under addition and multiplication)
We know that we can add and multiply integers and remain in the set of integers. In this way, the integers are closed under addition and closed under multiplication. We also note that both addition and multiplication are associative. We know that both addition is commutative; multiplication is also commutative, but here we emphasize the commutative nature of addition.

In the case of addition, we know that $0 \in \mathbb{Z}$ is the additive inverse: for every $a \in \mathbb{Z}$, we have

$$0 + a = a + 0 = a. \tag{4.27}$$

Similarly, we have that $1 \in \mathbb{Z}$ is the multiplicative inverse: we have

$$1 \cdot a = a \cdot 1 = a \tag{4.28}$$

for all $a \in \mathbb{Z}$.

We also know that addition and multiplication follow certain rules. In particular, given $a, b, c \in \mathbb{Z}$, we have

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)\,; \tag{4.29}$$

that is, we can *distribute* multiplication over addition. We also have

$$(a + b) \cdot c = (a \cdot c) + (b \cdot c)\,. \tag{4.30}$$

Example 4.10 (Integers modulo n)
We now look at modular arithmetic. To be concrete, we look at $\mathbb{Z}_6$. This satisfies all of the properties of the integers under addition and multiplication.

One interesting property of $\mathbb{Z}_6$ but not $\mathbb{Z}$ is that $2, 3 \in \mathbb{Z}_6$ with $2 \neq 0$ and $3 \neq 0$ but

$$2 \cdot 3 = 0 \mod 6. \tag{4.31}$$

This is an example where two nonzero elements may be multiplied together to equal zero. Thus, we can see that there is something distinctly different between $\mathbb{Z}$ and $\mathbb{Z}_6$.

4.2.3 Formal Definition

Definition 4.2 (Ring)**.** A ring is a set R together with two binary operations addition $+$ and multiplication $\cdot$. First, R is closed under addition and multiplication. Furthermore, we have the following properties:

- $(R, +)$ is an abelian group with additive identity 0. This implies that addition is associative.

- Multiplication is associative on R; that is, for all $a, b, c \in R$, we have

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)\,. \tag{4.32}$$

- There is multiplicative identity $1 \in R$; that is, for all $a \in R$, we have

$$1 \cdot a = a \cdot 1 = a. \tag{4.33}$$

  Furthermore, $0 \neq 1$.

- We have the following distribution laws between multiplication and addition. For all $a, b, c \in R$, we have

$$\begin{aligned} a \cdot (b + c) &= (a \cdot b) + (a \cdot c) \\ (b + c) \cdot a &= (b \cdot a) + (c \cdot a) \end{aligned} \tag{4.34}$$

We formally write $(R, +, \cdot)$ is a ring.

The above definition holds for all rings. A *commutative ring* is one where multiplication is commutative; that is $a \cdot b = b \cdot a$ for all $a, b \in R$. We will focus on commutative rings here because those will arise in our work moving forward. Thus, our distribution law reduces to

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)\,. \tag{4.35}$$

4.2.4 Continued Discussion

We say that $a \in R \setminus \{0\}$ is a *zero divisor* if there exists $b \in R \setminus \{0\}$ such that $ab = 0$.

Example 4.11 (Example of Zero Divisors)
We continue to look at the ring $\mathbb{Z}_6$. We have $2, 3 \in \mathbb{Z}_6$ with $2 \neq 0$ and $3 \neq 0$, yet we know

$$2 \cdot 3 = 6 \equiv 0 \mod 6. \tag{4.36}$$

This implies that 2 and 3 are zero divisors in $\mathbb{Z}_6$.

More generally, let $n = pq$ for distinct primes p and q. Then $p, q \in \mathbb{Z}_n$ and we know

$$p \cdot q = n \equiv 0 \mod n. \tag{4.37}$$

This shows that p and q are zero divisors in $\mathbb{Z}_n$.

Example 4.12 (Non-example of Zero Divisors)
While we may not have used this language before, $\mathbb{Z}$ has no zero divisors. We know (although we have not shown) that for $a, b \in \mathbb{Z}$ with $ab = 0$ implies that $a = 0$ or $b = 0$.

This shows that the ring $(\mathbb{Z}, +, \cdot)$ is very different from the rings $(\mathbb{Z}_n, +, \cdot)$ when n is composite.

4.2.5 Encoding Rings

When working with the ring $\mathbb{Z}_n$, we can encode $k \in \mathbb{Z}_n$ by its binary representation. Other rings have different representations.

4.3 Fields

4.3.1 Why do we care about Fields?

Fields arise in many different areas. In cryptography, we use fields in Diffie-Hellman Key Exchange and digital signatures. Additionally, fields are used when working with elliptic curves.

4.3.2 Intuition and Examples

Field are common mathematical objects. All fields are rings but have additional properties. Informally, a field is a generalization of the rational numbers with addition and multiplication. In particular, every nonzero element has a multiplicative inverse. We begin with some examples.

Example 4.13 (The Rational Numbers)
The rational numbers are a standard example of a field. For $a \in \mathbb{Q} \setminus \{0\}$, we can write

$$a = \frac{m}{n} \tag{4.38}$$

for $m, n \in \mathbb{Z} \setminus \{0\}$. Then $\frac{n}{m} \in \mathbb{Q}$ and

$$\frac{m}{n} \cdot \frac{n}{m} = 1. \tag{4.39}$$

Thus, a has multiplicative inverse. The rational numbers $\mathbb{Q}$ have an infinite number of elements.

Example 4.14 (The Field $(\mathbb{Z}_{11}, +, \cdot)$)
Code may be found at `examples/math_review/finite_field_inverses.py`.

Let us look at $\mathbb{Z}_{11}$. We have the following multiplication facts:

$$\begin{aligned}
1 \cdot 1 &= 1 \mod 11 & 6 \cdot 2 &= 1 \mod 11 \\
2 \cdot 6 &= 1 \mod 11 & 7 \cdot 8 &= 1 \mod 11 \\
3 \cdot 4 &= 1 \mod 11 & 8 \cdot 7 &= 1 \mod 11 \\
4 \cdot 3 &= 1 \mod 11 & 9 \cdot 5 &= 1 \mod 11 \\
5 \cdot 9 &= 1 \mod 11 & 10 \cdot 10 &= 1 \mod 11.
\end{aligned} \tag{4.40}$$

This shows that every nonzero element in $\mathbb{Z}_{11}$ has a multiplicative inverse. This, along with the other ring properties, makes $\mathbb{Z}_{11}$ a field. In this case, we write it as $\mathbb{F}_{11}$ to emphasize the field nature. The field $\mathbb{F}_{11}$ has a finite number of elements.

4.3.3 Formal Definition

Definition 4.3 (Field)**.** A field is a set F together with two binary operations addition $+$ and multiplication $\cdot$ which have the following properties:

- $(F, +)$ is an abelian group with additive identity 0.
- $(F^*, \cdot)$ is an abelian group with multiplicative identity 1. Here, $F^* \coloneqq F \setminus \{0\}$.
- We have $0 \neq 1$.
- We have the following distribution law between multiplication and addition. For all $a, b, c \in F$, we have

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)\,. \tag{4.41}$$

We formally say that $(F, +, \cdot)$ is a field.

4.3.4 Continued Discussion

Difference Between Rings and Fields

We note that a field is a commutative ring with no zero divisors and where every nonzero element has a multiplicative inverse.

Finite Fields

While the fields of the rationals $\mathbb{Q}$, the reals $\mathbb{R}$, and complex numbers $\mathbb{C}$ are more familiar, in cryptography we will be particularly interested in *finite fields*. As one may guess, a field $\mathbb{F}$ is *finite* if the number of elements is finite; that is, $|\mathbb{F}| < \infty$.

The finite fields we will focus on are $\mathbb{F}_p$; here p is a prime number. We will look at more examples of these fields below. There are additional types of finite fields, but the finite fields $\mathbb{F}_p$ are our primary focus. Additional material on fields may be found in Appendix B.4.

4.3.5 More Examples

Example 4.15 (Integers modulo a prime)
We now show that $(\mathbb{F}_p, +, \cdot)$ is a field, where

$$\mathbb{F}_p \coloneqq \{0, 1, 2, \ldots, p-1\} \tag{4.42}$$

and p is prime. We let

$$\mathbb{F}_p^* \coloneqq \mathbb{F}_p \setminus \{0\}\,, \tag{4.43}$$

so $\mathbb{F}_p^*$ are the nonzero elements of $\mathbb{F}_p$.

Let $a \in \mathbb{F}_p^*$ for prime $p > 2$. Because p is prime and $1 \leq a \leq p-1$, $\gcd(a, p) = 1$. Thus, there exists x and y such that

$$ax + py = 1. \tag{4.44}$$

If we look at the above modulo p, we see

$$ax = 1 \mod p. \tag{4.45}$$

Therefore, a has a multiplicative inverse $x \mod p$. This holds for all $a \in \mathbb{F}_p^*$, so $(\mathbb{F}_p, +, \cdot)$ is a field.

Example 4.16 (More examples with $\mathbb{F}_p$)
There are some unusual properties that we will discuss about $\mathbb{F}_p$. Throughout this example, all arithmetic will be performed modulo p.

For all $x \in \mathbb{F}_p$, we have

$$p \cdot x = 0. \tag{4.46}$$

By this, we mean

$$\underbrace{x + x + \cdots + x}_{p \text{ times}} = 0. \tag{4.47}$$

In particular, we have

$$p \cdot 1 = 0; \tag{4.48}$$

that is,

$$\underbrace{1 + 1 + \cdots + 1}_{p \text{ times}} = 0. \tag{4.49}$$

This is not the case in the more familiar fields of $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$. In particular,

$$\underbrace{1 + 1 + \cdots + 1}_{n \text{ times}} = n \tag{4.50}$$

for every $n \in \mathbb{N}$. It is not possible to add 1 to itself and arrive at 0.

Additionally, for $x, y \in \mathbb{F}_p$, we have

$$(x + y)^p = x^p + y^p. \tag{4.51}$$

This follows from the previous property and the Binomial Theorem[1]. Although this is fascinating, we will not discuss it further.

Example 4.17 (Even more examples with $\mathbb{F}_p$)
Code may be found at `examples/math_review/finite_field_powers.py`.

We now fix a value of p in order to work on some examples. We let

[1] https://en.wikipedia.org/wiki/Binomial_theorem

$$\begin{aligned} p &= 65537 \\ &= 2^{16} + 1. \end{aligned} \tag{4.52}$$

Addition modulo p is well understood. We will take some time to look at examples of multiplication. In particular, we will look at exponentiation.

We see

$$\begin{aligned} 3^0 &= 1 & 3^{32} &= 61869 \\ 3^1 &= 3 & 3^{64} &= 19139 \\ 3^2 &= 9 & 3^{128} &= 15028 \\ 3^4 &= 81 & 3^{256} &= 282 \\ 3^8 &= 6561 & 3^{512} &= 13987 \\ 3^{16} &= 54449 & 3^{1024} &= 8224. \end{aligned} \tag{4.53}$$

Once we reach a large enough exponent, say 16, there does not appear to be any relationship between the exponent x and the resulting number $3^x \mod p$.

To get a group from the field $\mathbb{F}_p$, we can look at the subgroup generated by 3 under multiplication; that is, we can look at

$$H \coloneqq \left\{3^x \mod p \mid x \in \mathbb{Z}\right\}. \tag{4.54}$$

Then $H \leq \mathbb{F}_p^*$; that is, H is a subgroup of the group $\left(\mathbb{F}_p^*, \cdot\right)$. In general, if we choose prime p large enough, we can use H as a group for the Diffie-Hellman Key Exchange; this will be discussed in Chapter 9.3.

4.3.6 Encoding Fields

When working with the field $\mathbb{F}_p$, we can encode $k \in \mathbb{F}_p$ by its binary representation. Other fields have different representations.

4.4 Concluding Discussion

It would be a valid question to ask if everything discussed here is absolutely necessary to begin to understand cryptography. The answer is *yes* if one is really interested in getting more than just a superficial view of Public Key Cryptography. This is because Public Key Cryptography looks to work in groups where certain problems are deemed computationally infeasible. We will discuss computational infeasibility more in Chapter 15.

Chapter 5

Mathematical Review: Elliptic Curves, Pairings, and Interpolation

In this chapter we continue on to more advanced mathematics. Understanding the information here is not required before reading other parts of the document. Even so, the more advanced parts of this document will require some familiarity with these topics.

5.1 Important Information

- Public Key Cryptography, such as the Diffie-Hellman Key Exchange, requires cyclic groups. Many cyclic groups may be constructed from elliptic curves with greater flexibility than the other groups discussed in Chapter 4.1 or Example 4.17.
- Bilinear pairings allow for many interesting possibilities; in particular, they allow for digital signatures like BLS signatures [29]. More generally, they allow for Pairing-Based Cryptography as discussed in Chapter 12.
- Lagrange Interpolation is used in the secret sharing protocols in Chapter 14.

5.2 Elliptic Curves

5.2.1 Why do we care about Elliptic Curves?

Elliptic curves are fascinating and extremely useful in number theory. We will give a brief introduction that will set the stage for what we need in Public Key Cryptography.

We begin by noting *why* we care about elliptic curves. Public Key Cryptography requires groups of various sizes and properties. It turns out that there is a lot of freedom when constructing specific elliptic curves, and it is easy to make many different kinds of groups. This freedom is what allows elliptic curves to be chosen to have particular properties that are useful in Public Key Cryptography.

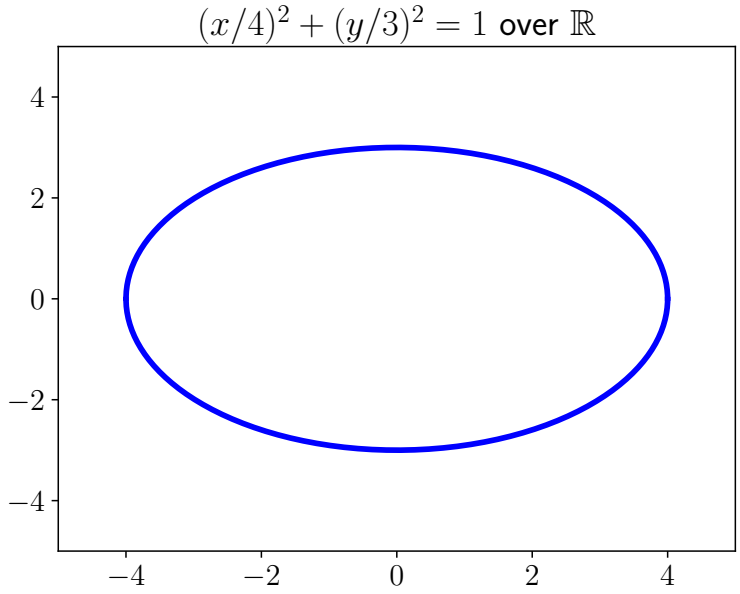


Figure 5.1: Here is a plot of an ellipse over $\mathbb{R}$. Ellipses are *not* elliptic curves.

5.2.2 Elliptic Curves are *not* Ellipses

First, we note that an elliptic curve is *not* an ellipse. We recall that an ellipse over the reals has the form

$$\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 = 1 \tag{5.1}$$

for constants $a, b > 0$ and $x, y \in \mathbb{R}$. Figure 5.1 shows an example of an ellipse.

Even though elliptic curves are not ellipses, they are useful for jumping into our discussion of elliptic curves. Ellipses describe a collection of points which satisfy a polynomial equation; in this case, a particular quadratic equation. Additionally, we have some notion of geometry by looking at the points which satisfy the equation. Elliptic curves arise in a similar manner but from a different polynomial equation.

We will never mention ellipses again.

5.2.3 Elliptic Curves over the Reals

We will now look at some examples of elliptic curves over $\mathbb{R}$. To do this, we will look at $(x, y) \in \mathbb{R}^2$ which satisfy

$$y^2 = x^3 + ax + b \tag{5.2}$$

for $a, b \in \mathbb{R}$. Example plots can be found in Figures 5.2 and 5.3.

First, we notice the symmetry about the x-axis. We expect this symmetry because we have $y^2 = \cdots$ in Eq. (5.2). Furthermore, we note that by changing a and b, we change characteristics of the curve. In Figure 5.2 we vary a; in Figure 5.3 we vary b.

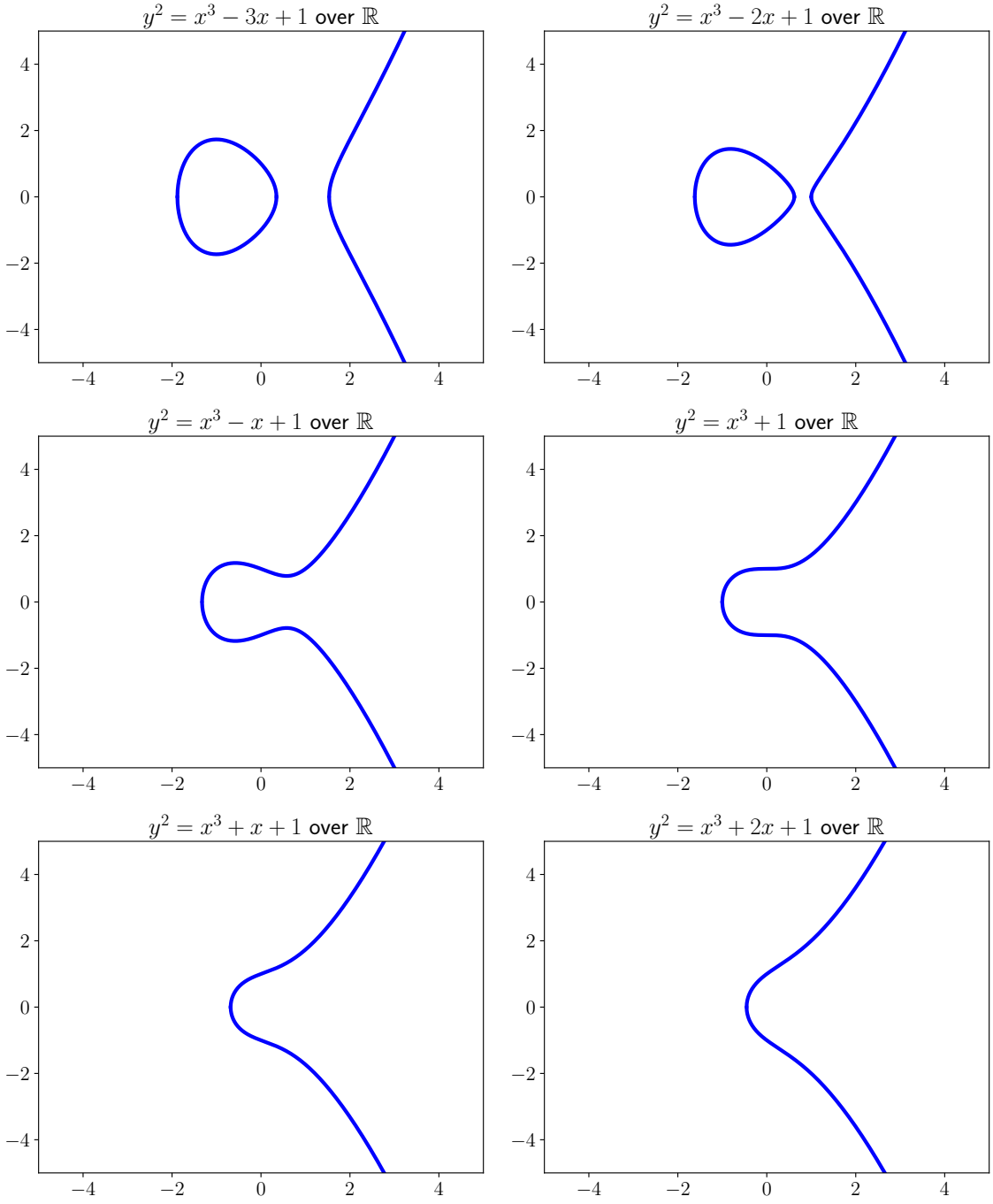


Figure 5.2: Here are plots of elliptic curves over $\mathbb{R}$. The (x, y) points satisfy the equation $y^2 = x^3 + ax + b$; we vary a and hold b constant.

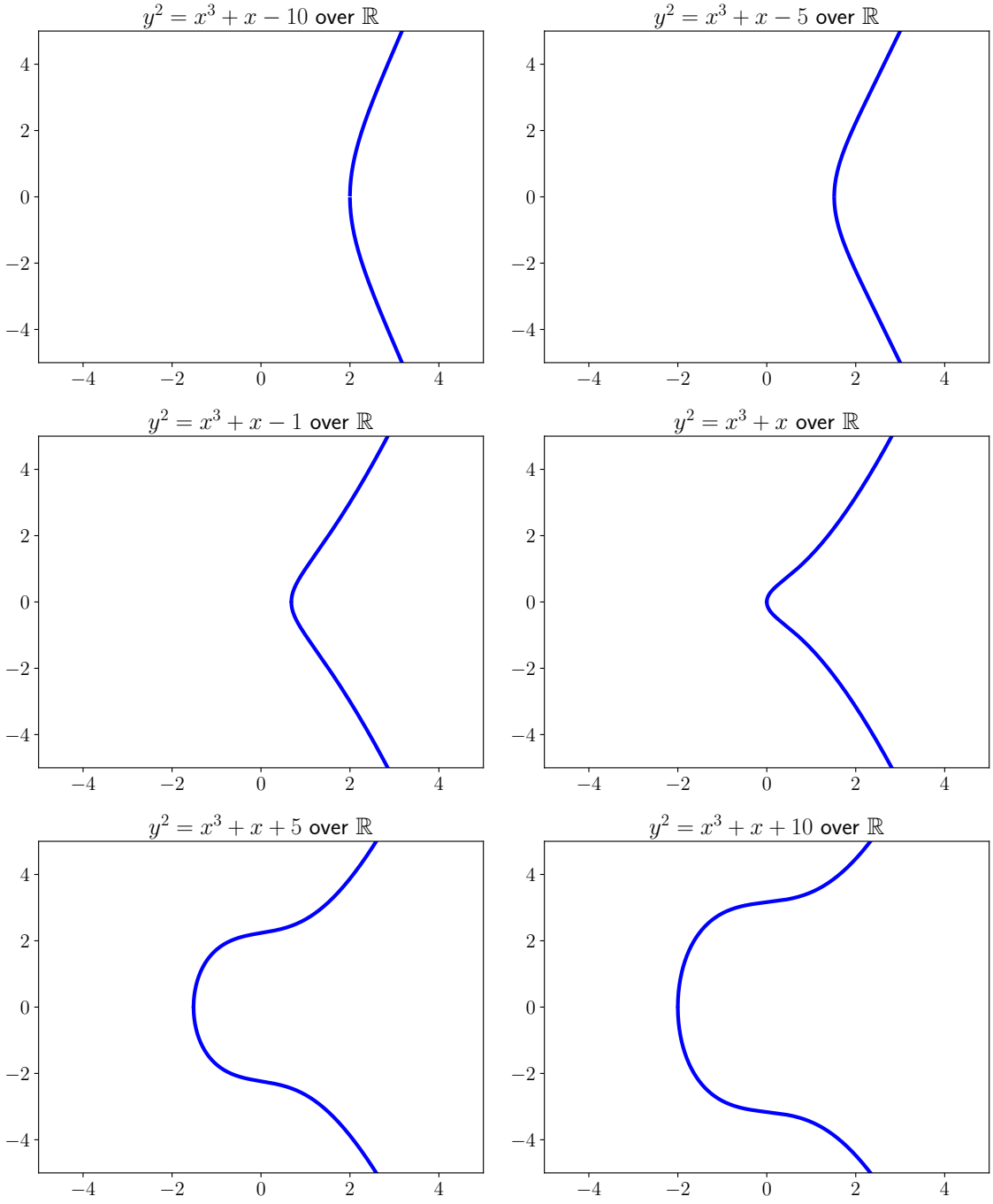


Figure 5.3: Here are additional plots of elliptic curves over $\mathbb{R}$. The (x, y) points satisfy the equation $y^2 = x^3 + ax + b$; we vary b and hold a constant.

5.2.4 Elliptic Curve Addition

We previously stated that elliptic curves give more freedom to construct groups. At this point, it is not clear what the group structure is, as we have just seen the equation defining the curve and looked at some plots.

We now define addition on points of elliptic curves. As before, we will be looking at points (x, y) which satisfy

$$y^2 = x^3 + ax + b. \tag{5.3}$$

We also include a special point $\mathcal{O}$; this point is the additive identity on the elliptic curve. Thus, for any point P on the elliptic curve, we have

$$\mathcal{O} + P = P. \tag{5.4}$$

We suppose that $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ with $P \neq \mathcal{O}$ and $Q \neq \mathcal{O}$; that is, P and Q are points on the elliptic curve which are not the identity element. We want to compute R such that

$$P + Q = R. \tag{5.5}$$

We have a few different cases to consider:

- Case 1: $x_1 \neq x_2$ and $y_1 \neq y_2$

  This the general case. We set

$$\begin{aligned} s &\coloneqq \frac{y_2 - y_1}{x_2 - x_1} \\ x_3 &\coloneqq s^2 - x_1 - x_2 \\ y_3 &\coloneqq s\left(x_1 - x_3\right) - y_1. \end{aligned} \tag{5.6}$$

  In this case, we set

$$R \coloneqq \left(x_3, y_3\right). \tag{5.7}$$

- Case 2: $x_1 = x_2$ and $y_1 \neq y_2$

  In this case, we have $Q = -P$, so

$$R \coloneqq \mathcal{O}. \tag{5.8}$$

- Case 3: $x_1 = x_2$ and $y_1 = y_2$

  In this case, we need to add a point to itself; this is also called *point doubling*, and we want to compute $P + P = R$. We set

$$\begin{aligned} s &\coloneqq \frac{3x_1^2 + a}{2y_1} \\ x_3 &\coloneqq s^2 - 2x_1 \\ y_3 &\coloneqq s\left(x_1 - x_3\right) - y_1. \end{aligned} \tag{5.9}$$

In this case, we set

$$R \coloneqq \left(x_3, y_3\right). \tag{5.10}$$

This, when including addition by the identity element, is the entire addition formula; one reference is [69, Proposition 9.70]. Although we have not shown it, elliptic curves form abelian groups; this means that

$$P + Q = Q + P \tag{5.11}$$

for all points P and Q.

As mentioned above, if we have $P = (x, y)$, the additive inverse $-P$ satisfies

$$-P \coloneqq (x, -y). \tag{5.12}$$

Naturally, we have

$$P + (-P) = \mathcal{O}. \tag{5.13}$$

The addition law described here depends on the particular form of elliptic curve. Additional types of elliptic curves are discussed below, and the addition formula described here does not always apply. In fact, due to the nature of the addition formula (involving conditional checks and different paths for different points), certain Elliptic Curve Cryptography (ECC) algorithms specifically choose a different form of elliptic curve. The branch logic required to implement the above addition formula will leak secret information if it is not carefully implemented [37].

Plots for elliptic curve addition in Examples 5.1 and 5.2 can be found in Figure 5.4.

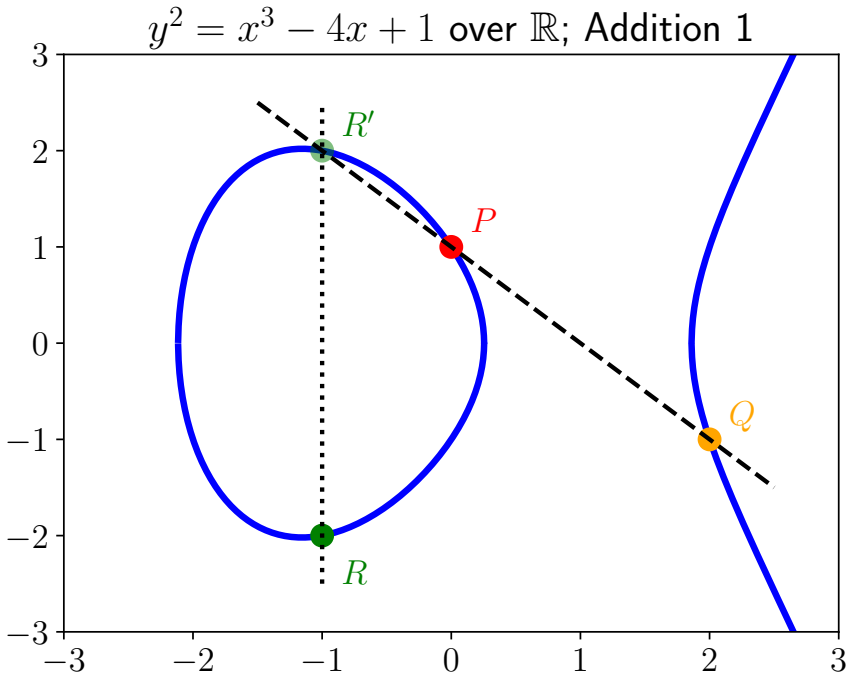
Example 5.1 (Elliptic Curve Addition over Reals 1)
We now look at addition on the elliptic curve

$$E : y^2 = x^3 - 4x + 1 \tag{5.14}$$

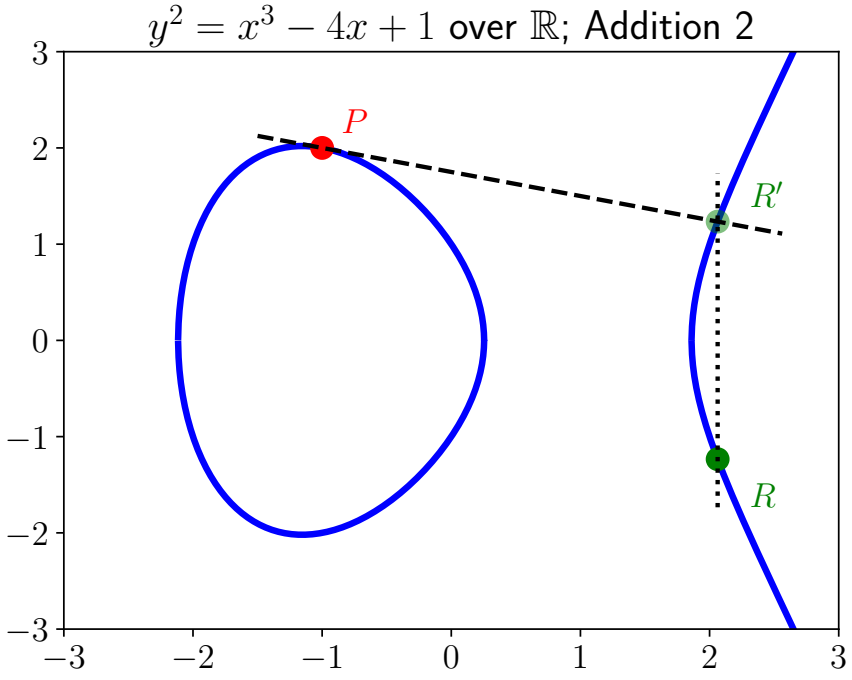
over $\mathbb{R}$. We want to add the two points

$$\begin{aligned} P &= (0, 1) \\ Q &= (2, -1). \end{aligned} \tag{5.15}$$

Using the addition formula, we find



(a) Plot of elliptic curve addition over $\mathbb{R}$ for Example 5.1; this is an example of adding distinct points. We have $P + Q = R$. This figure shows $(0, 1) + (2, -1) = (-1, -2)$.



(b) Plot of elliptic curve addition over $\mathbb{R}$ for Example 5.2; this is an example of *point doubling*. We have $P + P = R$. This may also be written as $2 \cdot P = R$. This figure shows $2 \cdot (-1, 2) = \left(\frac{33}{16}, -\frac{79}{64}\right)$.

Figure 5.4: Here we plot elliptic curve addition over $\mathbb{R}$ for Examples 5.1 and 5.2. Figure 5.4a shows the addition of distinct points while Figure 5.4b shows the addition of repeated points (point doubling).

$$\begin{aligned} s &= -1 \\ x_3 &= -1 \\ y_3 &= -2. \end{aligned} \tag{5.16}$$

This gives us the point

$$R = (-1, -2)\,. \tag{5.17}$$

Thus, we have

$$(0, 1) + (2, -1) = (-1, -2). \tag{5.18}$$

See Figure 5.4a for a plot.

To understand how R is determined, we look at the line connecting P and Q. This line will intersect the elliptic curve at another point R'; it can be shown that this will always happen. The point R is defined to be the reflection of R' across the x-axis.

Example 5.2 (Elliptic Curve Addition over Reals 2)
We use the same elliptic curve in Example 5.1:

$$E : y^2 = x^3 - 4x + 1 \tag{5.19}$$

over $\mathbb{R}$.

In this example, we look at adding a point to itself; this is also called *point doubling*. In this case, we have the point

$$P = (-1, 2)\,. \tag{5.20}$$

We see

$$\begin{aligned} s &= -\frac{1}{4} \\ x_3 &= \frac{33}{16} \\ y_3 &= -\frac{79}{64}. \end{aligned} \tag{5.21}$$

This gives us the point

$$R = \left(\frac{33}{16}, -\frac{79}{64}\right). \tag{5.22}$$

Thus, we have

$$(-1, 2) + (-1, 2) = \left(\frac{33}{16}, -\frac{79}{64}\right). \tag{5.23}$$

This may also be written as

$$2 \cdot (-1, 2) = \left(\frac{33}{16}, -\frac{79}{64} \right). \tag{5.24}$$

See Figure 5.4b for a plot.

To understand how R is determined, we now look at the tangent line of P on the elliptic curve. This line will intersect the elliptic curve at another point R'; it can be shown that this will always happen. The point R is defined to be the reflection of R' across the x-axis.

5.2.5 Elliptic Curves over General Fields

We let K be a field. For concreteness, the reader may assume $K = \mathbb{R}$ as before. We are interested in looking at points $(x, y) \in K^2$ which satisfy the equation

$$E : y^2 = x^3 + ax + b \tag{5.25}$$

for $a, b \in K$. Elliptic curves given in the form in Eq. (5.25) are said to be in *Weierstrass form*. We will see other forms of elliptic curves in Chapter 5.2.10 when we discuss some specific elliptic curves which are used in practice. The addition law we discussed in Chapter 5.2.4 is the same provided the field operations over K are used.

The set of points of an elliptic curve along with the addition formula gives rise to an abelian group. We can look at cyclic subgroups of the elliptic curves to specify a Discrete Logarithm Problem.

We note that there are some technical restrictions on a and b in Eq. (5.25), but we do not discuss the specifics here.

5.2.6 Elliptic Curves over Finite Fields

We now focus on looking at elliptic curves over the field $\mathbb{F}_p$ for prime p. Example plots can be found in Figures 5.5 and 5.6 where we look at elliptic curves over $\mathbb{F}_{71}$.

The elliptic curves are the "same" as the ones in Figures 5.2 and 5.3. That is, the *equation* describing each of them is the same; the only difference is that we are now looking for solutions over a different field; we are looking for solutions over $\mathbb{F}_{71}$ rather than $\mathbb{R}$. By looking at the plots, there does not appear to any relationship between the parameters and the points on the elliptic curve. We still note the symmetry about the middle of the graph, though. This comes from the $y^2 = \cdots$ in the equation definition.

To understand the symmetry about the middle of the graph, we will spend a bit more time discussing this. First, we remember that, in our finite fields, we have

$$-y = p - y \mod p. \tag{5.26}$$

To be concrete, we will focus on $y^2 = x^3 + 2x + 1$ over $\mathbb{F}_{71}$; see Figure 5.7a. We can see *two* lines of symmetry: one line at $y = 0$; another line at $y = p/2$. These two lines of symmetry arise from $-y = p - y \mod p$: we are free in where we put the "negative" numbers. We can place the "negative" numbers between $p/2$ and p as seen in Figure 5.7b; or we can place the "negative" numbers between $-p/2$ and 0 as seen in Figure 5.7c. In all of our plots, we will

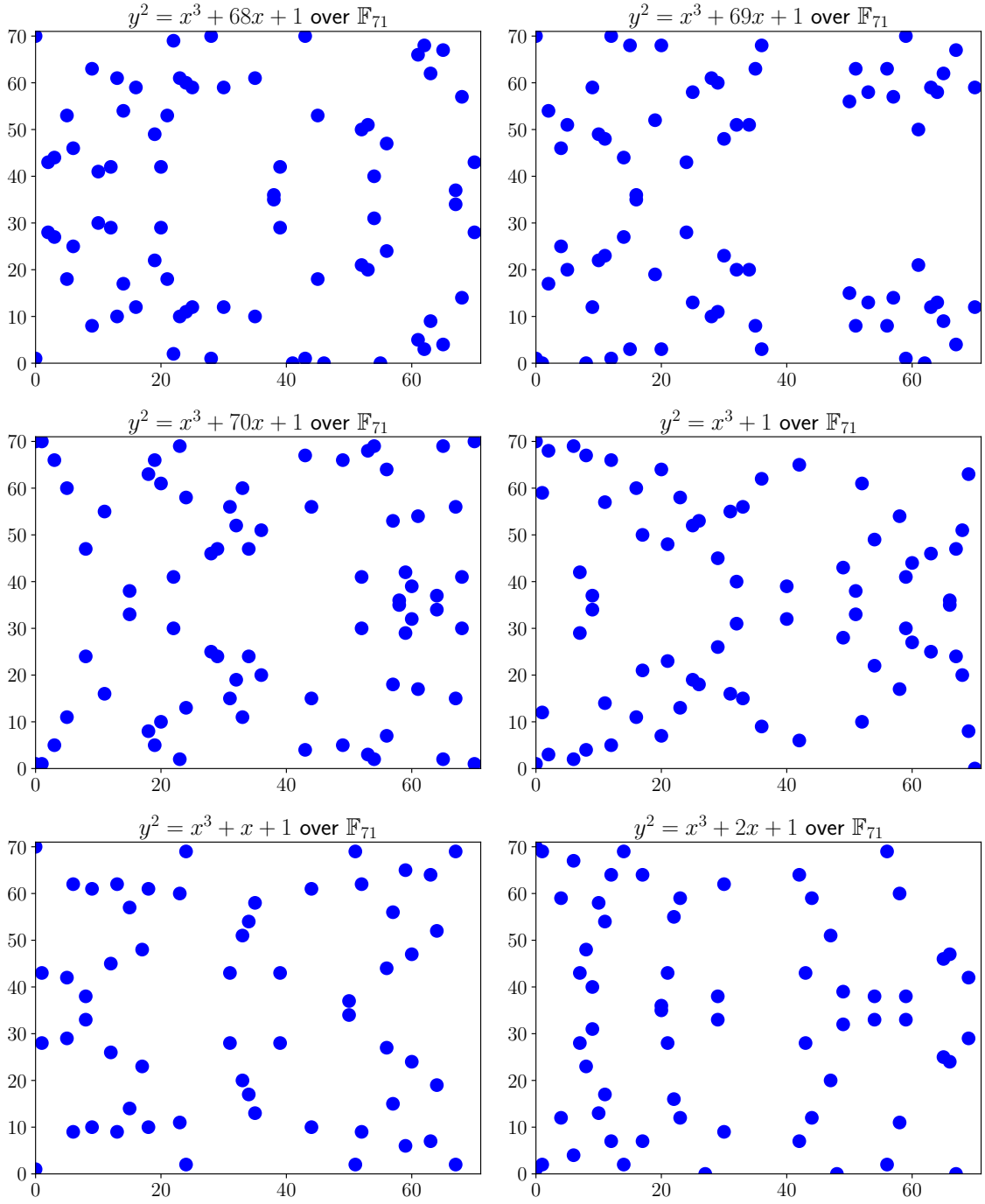


Figure 5.5: Here are plots of elliptic curves over $\mathbb{F}_{71}$. Here, we are keeping b constant.

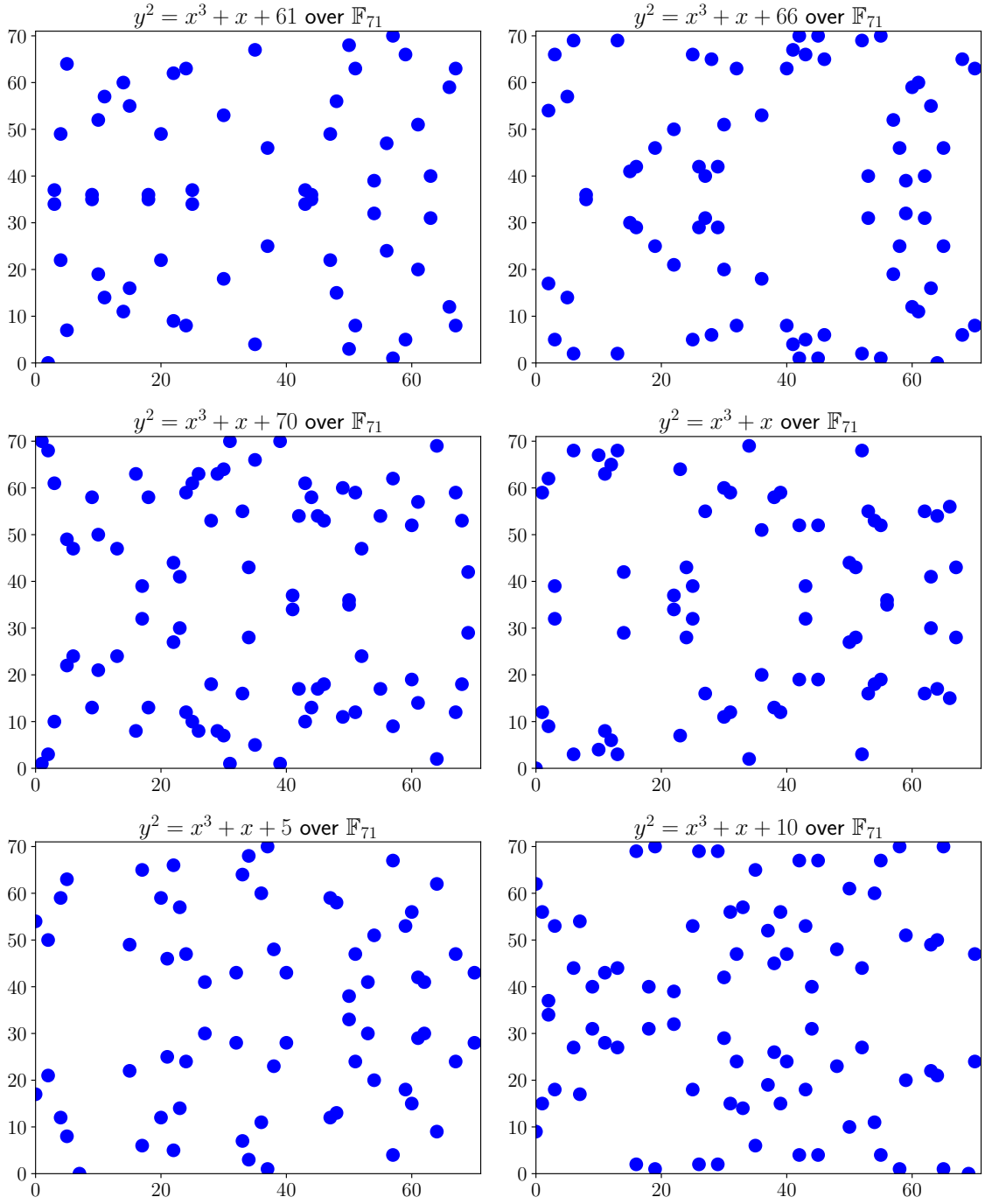
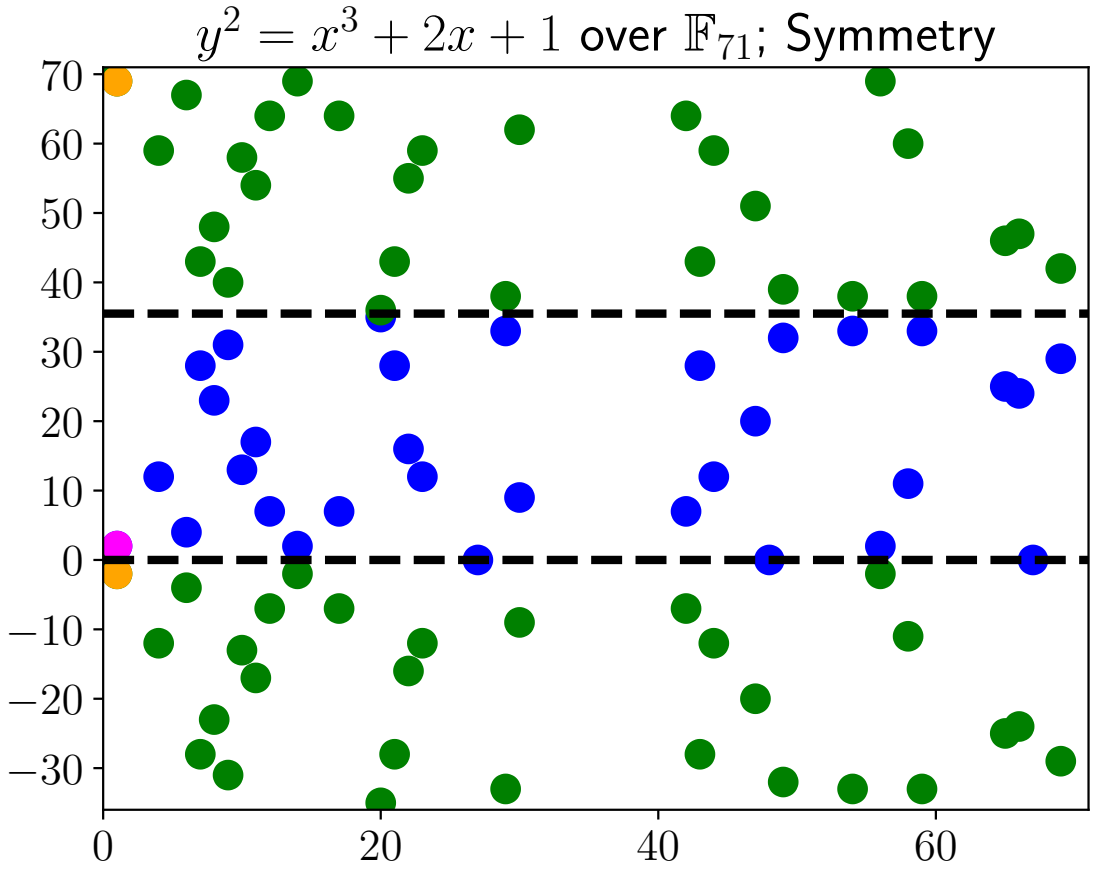
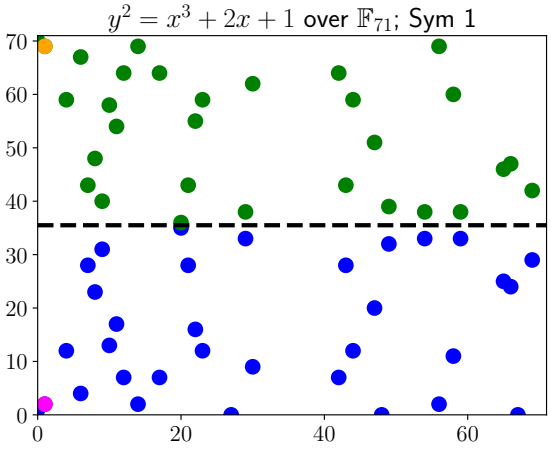


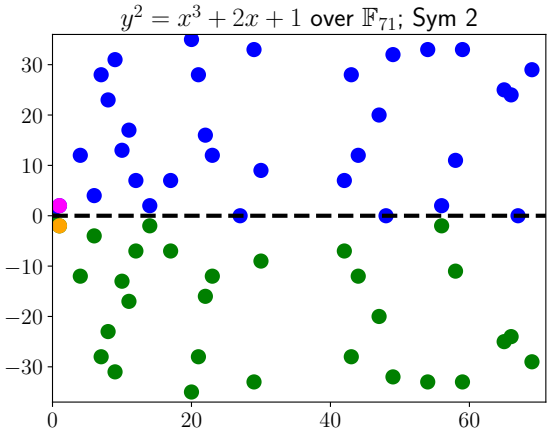
Figure 5.6: Here are additional plots of elliptic curves over $\mathbb{F}_{71}$. Here, we are keeping a constant.



(a) Plot with all y values in $[-p/2, p]$.



(b) Plot with y values in $[0, p]$.



(c) Plot with y values in $[-p/2, p/2]$.

Figure 5.7: Here have a plot showing the symmetry of elliptic curves over $\mathbb{F}_{71}$. The "positive" points are in blue with y-values between 0 and $p/2$; the "negative" points are in green with y-values between $p/2$ and p. Because $-y = p - y \mod p$, the negative points are equivalent to those with y values between $-p/2$ and 0.

use the first convention and always have plot y values between 0 and p. Within Figure 5.7, we have a distinguished point $(1, 2)$. The additive inverse of this point is $(1, -2)$. This is equivalent to $(1, 69)$ over $\mathbb{F}_{71}$.

All of the previous plots looked at elliptic curves over $\mathbb{F}_{71}$. Even so, the particular form of elliptic curve depends on the finite field as well. Figure 5.8 includes some plots where we keep the equation fixed but change the prime p.

We have not talked about methods to determine how many points are on a specific elliptic curve. There are methods to compute the exact number of elements on an elliptic curve, but we will not discuss them here. Here is an easy bound on the number of points:

Theorem 5.3 (Hasse's Theorem on Elliptic Curves [120, Theorem V.1.1])
Let $\#E(\mathbb{F}_p)$ denote the number of elements of an elliptic curve $E/\mathbb{F}_p$. Then we have the following bound:

$$|\#E(\mathbb{F}_p) - (p+1)| \leq 2\sqrt{p}. \tag{5.27}$$

This means that the number of points on the elliptic curve $E(\mathbb{F}_p)$ is essentially p.

We now look at some examples of addition on elliptic curves. By looking at the points and the elliptic curve, there does not appear to be any structure.

Example 5.4 (Addition of Elliptic Curves over Finite Fields)
Code may be found at `examples/math_review/elliptic_curve_addition.py`.

We will focus on the elliptic curve

$$E : y^2 = x^3 + 2x + 1 \tag{5.28}$$

over $\mathbb{F}_{71}$.

We compute the following values:

$$\begin{aligned}
(17, 7) + (20, 35) &= (58, 60) \\
(17, 7) + (21, 28) &= (65, 25) \\
(17, 7) + (22, 16) &= (4, 59) \\
(17, 7) + (23, 12) &= (10, 58) \\
(17, 7) + (29, 33) &= (8, 48)\,.
\end{aligned} \tag{5.29}$$

We plot the results of addition in Figure 5.9.

5.2.7 Elliptic Curve Scalar Multiplication

In Chapter 5.2.8, we will want to repeatedly add a point on an elliptic curve to itself. That is, we will want to compute $k \cdot P$ for $k \in \mathbb{Z}$ and $P \in E(\mathbb{F}_p)$. This is defined as

$$k \cdot P \coloneqq \underbrace{P + P + \cdots + P}_{k \text{ times}}, \quad k \geq 0. \tag{5.30}$$

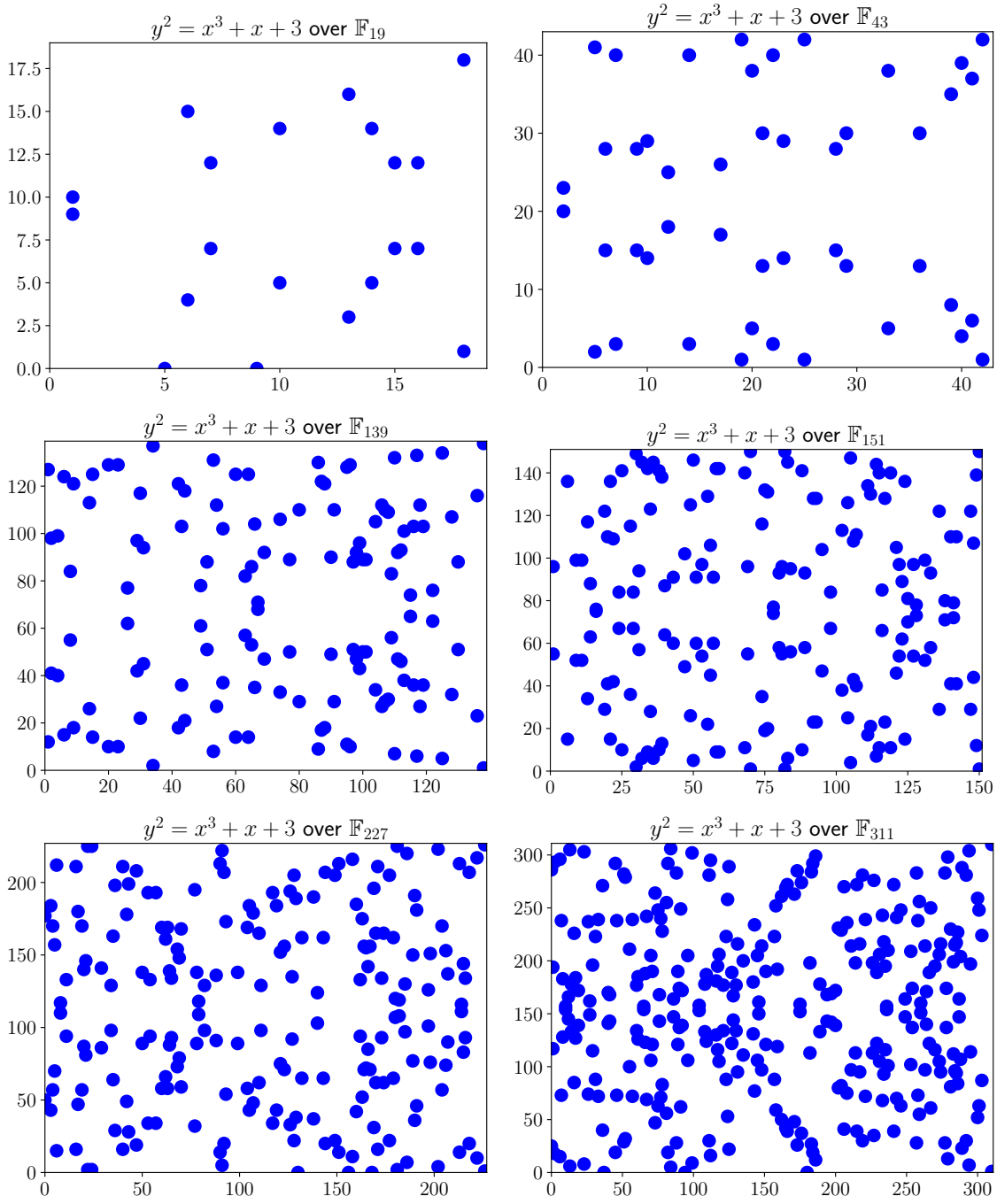
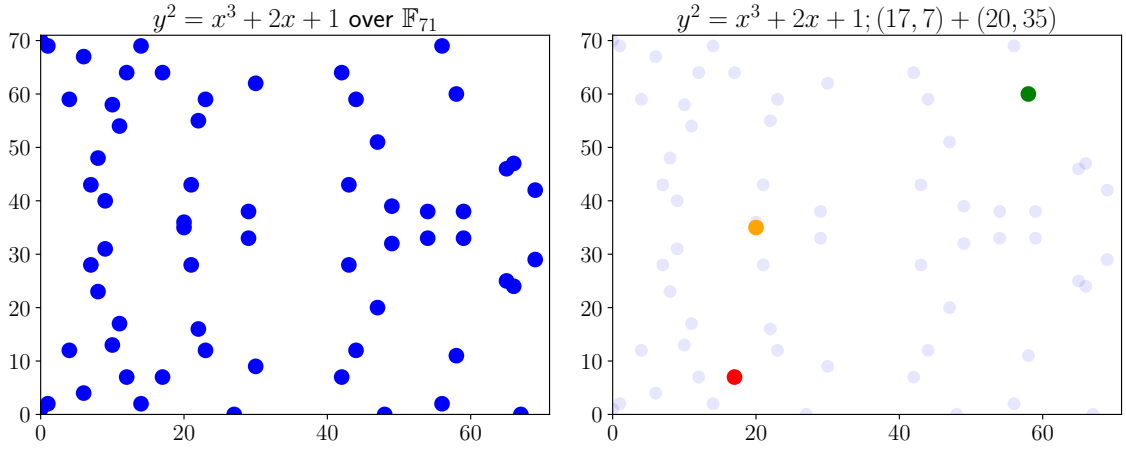
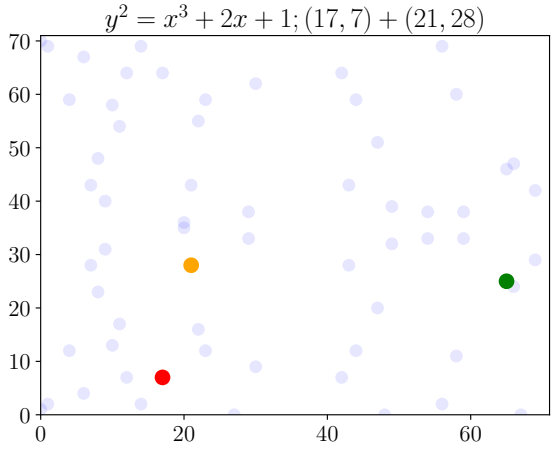


Figure 5.8: Here are plots of elliptic curves over $\mathbb{F}_p$ for primes $p \in \{19, 43, 139, 151, 227, 311\}$.

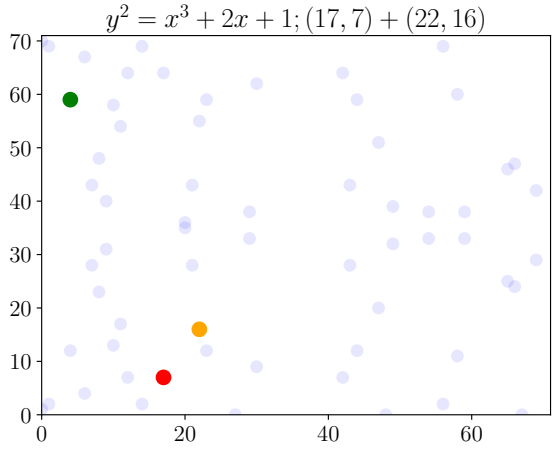


(a) All points on $y^2 = x^3 + 2x + 1$ over $\mathbb{F}_{71}$.

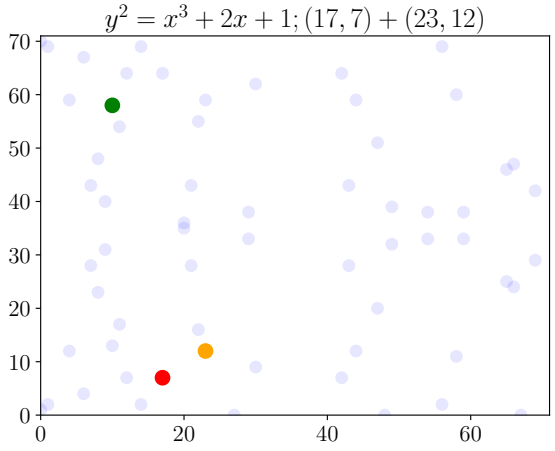
(b) $(17, 7) + (20, 35) = (58, 60)$



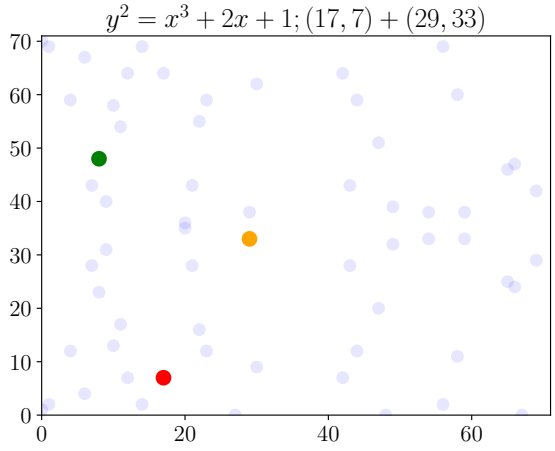
(c) $(17, 7) + (21, 28) = (65, 25)$



(d) $(17, 7) + (22, 16) = (4, 59)$



(e) $(17, 7) + (23, 12) = (10, 58)$



(f) $(17, 7) + (29, 33) = (8, 48)$

Figure 5.9: Here have plots of addition on elliptic curves over $\mathbb{F}_{71}$. The elliptic curves are all the same: $y^2 = x^3 + 2x + 1$. The first plot shows the points on the elliptic curve; the other plots show the addition $P + Q = R$. Throughout the plots, $P = (17, 7)$.

Algorithm 5.1 Double-and-add formula for elliptic curve scalar multiplication

Require: P point on elliptic curve; $k \in \mathbb{N}$

```
procedure DoubleAndAdd(P, k)
    if k = 0 then
        return 𝒪
    else if k = 1 then
        return P
    else if k ≡ 0 mod 2 then
        return DoubleAndAdd(P + P, k/2)
    else
        return DoubleAndAdd(P + P, (k − 1)/2) + P
    end if
end procedure
```

Similarly, we have

$$k \cdot P := \underbrace{-P - P - \cdots - P}_{-k \text{ times}}, \quad k < 0. \tag{5.31}$$

This is called *elliptic curve point multiplication* or *elliptic curve scalar multiplication*. We remember that if $P \in E(\mathbb{F}_p)$ and E is in Weierstrass form with $P = (x, y)$, then $-P = (x, -y)$. Thus, point negation has negligible cost.

We recall that, in practice, we will define elliptic curves over $\mathbb{F}_p$ with p being a 256-bit prime number. Thus, k will also be a 256-bit number. Computing $k \cdot P$ by performing

$$\begin{aligned} 2P &= P + P \\ 3P &= P + 2P \\ 4P &= P + 3P \\ 5P &= P + 4P \\ &\vdots \end{aligned} \tag{5.32}$$

would take around 2^{256} elliptic curve additions. There is not a fast enough computer to perform the calculation using this method.

Thankfully, there is a better method called the *double-and-add* method; see Alg. 5.1 for a formal specification using recursion. We will walk through the method in Example 5.5.

While this is an efficient algorithm to compute $k \cdot P$, the branch logic will *leak information* about k. For instance, this type of operation occurs when k could be a private key. Thus, if this operation is not performed in a secure method *independent* of k, the private key could be revealed [136].

We note that the double-and-add formula may be used when computing large exponentiations as well; the algorithms are the same except for switching the operations from doubling and addition to squaring and multiplication.

Example 5.5 (Double-and-Add Example)
Code may be found at `examples/math_review/elliptic_curve_double-and-add.py`.

We continue using the elliptic curve

$$E : y^2 = x^3 + 2x + 1 \tag{5.33}$$

defined over $\mathbb{F}_{71}$ as before. We want to compute

$$29 \cdot (0, 1) \,. \tag{5.34}$$

We will use Alg. 5.1. We first see that

$$(0, 1) + (0, 1) = (1, 69) \,. \tag{5.35}$$

We have

$$\textsc{DoubleAndAdd}\left[(0, 1), 29\right] = (0, 1) + \textsc{DoubleAndAdd}\left[(1, 69), 14\right] . \tag{5.36}$$

Next, we see

$$(1, 69) + (1, 69) = (4, 59) \,. \tag{5.37}$$

We then compute

$$\textsc{DoubleAndAdd}\left[(1, 69), 14\right] = \textsc{DoubleAndAdd}\left[(4, 59), 7\right] . \tag{5.38}$$

In the next step, we find

$$(4, 59) + (4, 59) = (56, 2) \,. \tag{5.39}$$

After this, we have

$$\textsc{DoubleAndAdd}\left[(4, 59), 7\right] = (4, 59) + \textsc{DoubleAndAdd}\left[(56, 2), 3\right] . \tag{5.40}$$

Another addition gives us

$$(56, 2) + (56, 2) = (67, 0) \,. \tag{5.41}$$

The final iteration gives us

$$\begin{aligned} \textsc{DoubleAndAdd}\left[(56, 2), 3\right] &= (56, 2) + \textsc{DoubleAndAdd}\left[(67, 0), 1\right] \\ &= (56, 2) + (67, 0) \,. \end{aligned} \tag{5.42}$$

Our algorithm has stopped.

At this point, we can add all of the points we have computed. Doing this, we see

$$\begin{aligned} 29 \cdot (0,1) &= (0,1) + \{(4,59) + [(56,2) + (67,0)]\} \\ &= (0,1) + \{(4,59) + (56,69)\} \\ &= (0,1) + (4,12) \\ &= (8,48)\,. \end{aligned} \tag{5.43}$$

This took us 7 additions on the elliptic curve; this is much less than 29.

In fact, using Alg. 5.1, the computation is logarithmic in k; that is, it will take $O(\log(k)^c)$ steps for small constant $c > 0$.

5.2.8 Subgroups of Elliptic Curves over Finite Fields

We now have the addition formula and the ability to efficiently add points. *This* allows us to create subgroups of elliptic curves over finite fields for Elliptic Curve Cryptography.

We let $P \in E(\mathbb{F}_p)$ be a point on an elliptic curve. We define the following subgroup:

$$\langle P \rangle \coloneqq \{k \cdot P \mid k \in \mathbb{Z}\}\,. \tag{5.44}$$

Naturally, we have $\langle P \rangle \leq E(\mathbb{F}_p)$; more explicitly, $\langle P \rangle$ is the subgroup of the elliptic curve $E(\mathbb{F}_p)$ formed by looking at all the points on the elliptic curve which we can get by adding P to itself. We can always look at subgroups of this form over general elliptic curves; when we restrict our attention to finite fields, then the elliptic curves are finite groups. These finite groups allow us to look at the *Discrete Logarithm Problem*: If $Q \in \langle P \rangle$, determine $x \in \mathbb{Z}$ such that

$$Q = x \cdot P. \tag{5.45}$$

For many elliptic curves, this is difficult problem. This is discussed more in Chapter 15.

See Figure 5.10 for example cyclic subgroups of elliptic curves over finite fields.

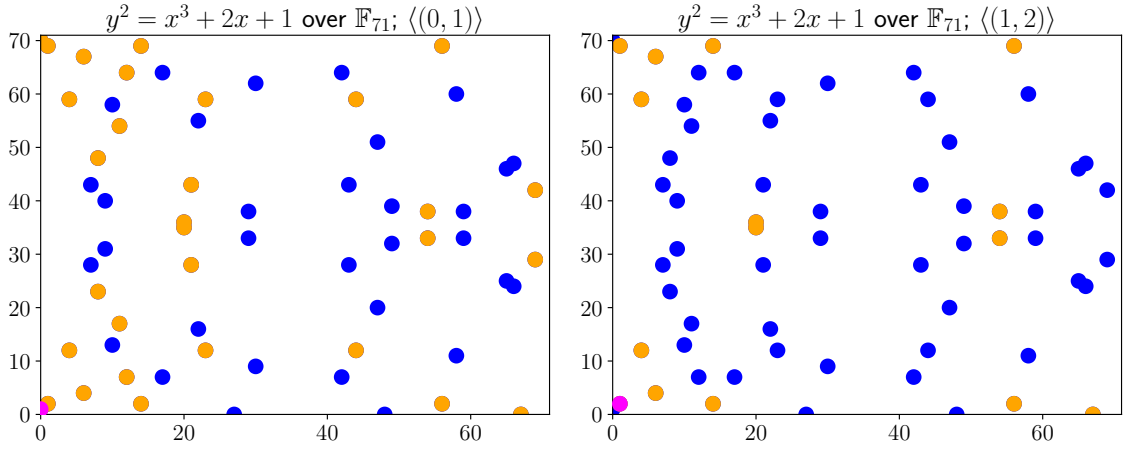
Example 5.6 (Subgroups of Elliptic Curves)
Code may be found at `examples/math_review/elliptic_curve_subgroup.py`.

In this example we look at subgroups on an elliptic curve. We will focus on the elliptic curve

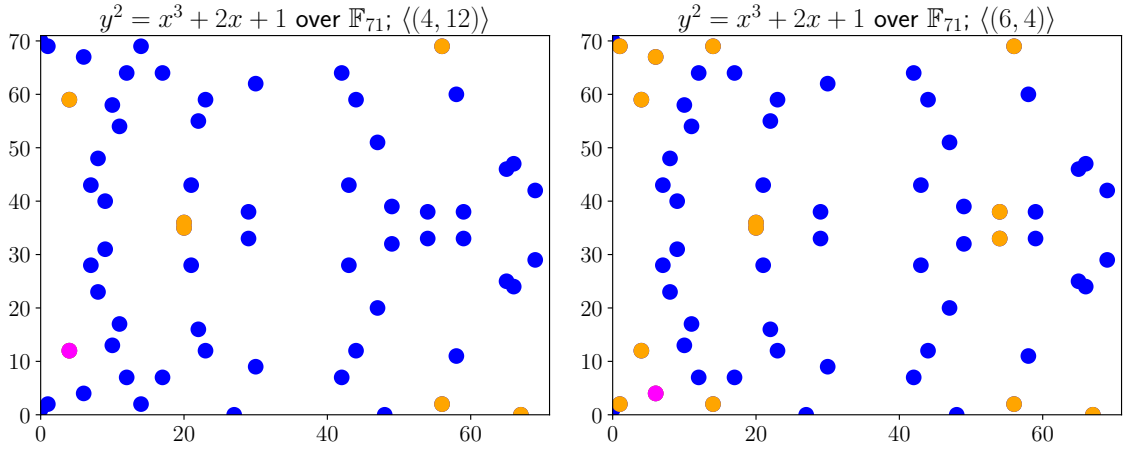
$$E : y^2 = x^3 + 2x + 1 \tag{5.46}$$

over $\mathbb{F}_{71}$. Plots of various subgroups are found in Figure 5.10.

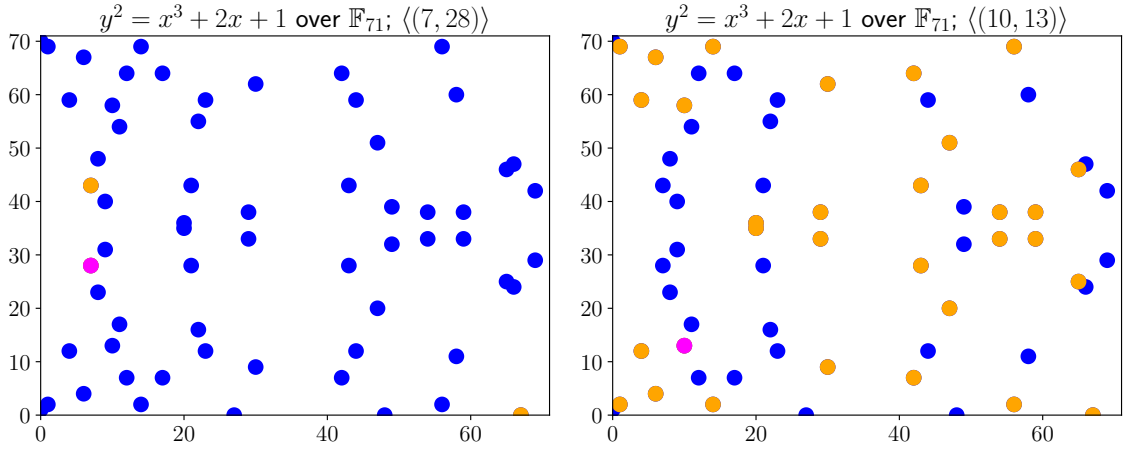
We will focus on $\langle (1,2) \rangle$; this can be seen in Figure 5.10b. If we list out all the points of the subgroup, we see



(a) Subgroup generated by $(0, 1)$ (b) Subgroup generated by $(1, 2)$



(c) Subgroup generated by $(4, 12)$ (d) Subgroup generated by $(6, 4)$



(e) Subgroup generated by $(7, 28)$ (f) Subgroup generated by $(10, 13)$

Figure 5.10: Here are plots of elliptic curves over $\mathbb{F}_{71}$. The elliptic curves are all the same: $y^2 = x^3 + 2x + 1$. In each plot, we also show different cyclic subgroups; subgroups generated by different base points.

$$
\begin{aligned}
1 \cdot (1,2) &= (1,2) & 9 \cdot (1,2) &= (6,67) \\
2 \cdot (1,2) &= (4,12) & 10 \cdot (1,2) &= (20,35) \\
3 \cdot (1,2) &= (14,2) & 11 \cdot (1,2) &= (54,33) \\
4 \cdot (1,2) &= (56,69) & 12 \cdot (1,2) &= (56,2) \\
5 \cdot (1,2) &= (54,38) & 13 \cdot (1,2) &= (14,69) \\
6 \cdot (1,2) &= (20,36) & 14 \cdot (1,2) &= (4,59) \\
7 \cdot (1,2) &= (6,4) & 15 \cdot (1,2) &= (1,69) \\
8 \cdot (1,2) &= (67,0) & 16 \cdot (1,2) &= \mathcal{O}.
\end{aligned} \tag{5.47}
$$

As we look at the point distribution and compare it with the rest of the points, there does not appear to be any structure between the points in the subgroup or not. It is this apparent lack of structure that makes elliptic curve groups great for Public Key Cryptography.

We note that the subgroups in Figures 5.10b and 5.10d are the same; this means that the points $(1,2)$ and $(6,4)$ generate the same subgroups.

Example 5.7 (Elliptic Curve Discrete Logarithm)
We continue the previous example with the same elliptic curve and subgroup.

We want to find x such that

$$x \cdot (1,2) = (54,33)\,. \tag{5.48}$$

Because the addition formula for points on elliptic curves is "random", there does not appear to be any better way to find x than to just list out every possibility. From looking at all the scalar multiplications in Eq. (5.47), we see that $x = 11$.

We note that generic methods for solving the elliptic curve Discrete Logarithm Problem are discussed in Chapter 15.1.

5.2.9 Elliptic Curve Point Compression

Due to the fact that elliptic curves satisfy an algebraic equation, we are able to *compress* points on the elliptic curve so that they take up less space.

We assume that p is an odd prime. As before, let us look at an elliptic curve $E/\mathbb{F}_p$ and suppose that we have a point on the elliptic curve $P = (x,y)$. We know that we have the relation

$$y^2 = x^3 + ax + b. \tag{5.49}$$

for some specified $a, b \in \mathbb{F}_p$.

If we only knew x, then we can solve for y:

$$y = \pm\sqrt{x^3 + ax + b}\,. \tag{5.50}$$

The challenge is to then determine *which* y value is the appropriate one. We see that such a square root must exist, so we set

$$s \coloneqq \sqrt{x^3 + ax + b}. \tag{5.51}$$

Note well that this square root is a *modular square root*, not a regular square root. Thus, we have

$$y = \pm s. \tag{5.52}$$

Now, we remember that all of these operations are modular arithmetic operations modulo a prime p. Thus, we know $s \in \{0, 1, 2, \cdots, p-1\}$. We also remember that

$$-s = p - s \mod p. \tag{5.53}$$

Because we are assuming that p is an odd prime, we know there are two possibilities: s is even and $p - s$ is odd; or s is odd and $p - s$ is even.

We can use these facts to compress the point $P = (x, y)$ on an elliptic curve; this allows for more efficient transmission. For instance, suppose that p is an n-bit prime. In this case, the uncompressed form of representing P requires $2n$ bits. The compressed form only requires $n + 1$ bits: n bits to store x and 1 bit to store whether y is even or odd. By using the compressed form, storage is essentially halved. This assumes that the particular elliptic curve being used is public knowledge so that a and b are known.

We did not mention *how* to compute s in Eq. (5.51). Methods for computing square roots in finite fields may be found in Appendix B.4.

Example 5.8 (Elliptic Curve Point Compression)
Code may be found at `examples/math_review/elliptic_curve_compression.py`.

We continue to work with the elliptic curve

$$E : y^2 = x^3 + 2x + 1 \tag{5.54}$$

over $\mathbb{F}_{71}$.

If we want to compress the point $(17, 7)$, then we need to send $(17, \texttt{0b1})$; we send `0b1` because the y-coordinate 7 is odd.

Suppose we receive the compressed point $(47, \texttt{0b0})$. Thus, the x-coordinate is 47 and the y-coordinate is even. We see that

$$\begin{aligned} t &\coloneqq x^3 + ax + b \mod p \\ &= 47^3 + 2 \cdot 47 + 1 \mod 71 \\ &= 45. \end{aligned} \tag{5.55}$$

We can verify that

$$51^2 \mod 71 = 45. \tag{5.56}$$

Thus, 51 is a square root of 45. It follows that

$$\begin{aligned} s &\coloneqq \sqrt{t} \mod p \\ &= 51. \end{aligned} \tag{5.57}$$

We note, though, that s is odd although we were told y is even; thus, we have

$$\begin{aligned} y &\coloneqq p - s \\ &= 20. \end{aligned} \tag{5.58}$$

Therefore, the point in uncompressed form is

$$P = (47, 20)\,. \tag{5.59}$$

As we can see from the above example, the point compression for elliptic curves trades storage for computation. Compression may be worthwhile in situations where storage is relatively expensive.

5.2.10 Examples of Specific Curves

We now include some specific curves used in practice. We also bring them up because some of them have forms different than the Weierstrass form we have been using.

While it is very easy to come up with additional elliptic curves, developing new elliptic curves for cryptography requires *extreme care*. Doing this is *highly discouraged*, as it amounts to "rolling your own crypto". This was frowned upon in Chapter 2.

Whenever possible, use a well-tested library developed by experienced cryptologists. If this is not possible, *proceed with caution*.

Secp256k1

This elliptic curve is used by both Bitcoin [92] and Ethereum [135, Appendix F] for digital signatures. It has the form

$$E : y^2 = x^3 + 7 \tag{5.60}$$

over the field $\mathbb{F}_q$ with

$$q = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1. \tag{5.61}$$

It was defined in [34, Section 2.4.1].

Curve25519

This curve [14] is used in X25519, the Diffie-Hellman Key Exchange based on this curve. We have

$$E : y^2 = x^3 + 486662x^2 + x \tag{5.62}$$

over the prime field $\mathbb{F}_q$, where

$$q = 2^{255} - 19. \tag{5.63}$$

This is a *Montgomery curve*. A general Montgomery curve has the form

$$M_{A,B} : By^2 = x^3 + Ax^2 + x \tag{5.64}$$

over a field K with $A, B \in K$. Having this specific form allows for some operations to be efficiently computed.

Ed25519

This is one of the elliptic curves used for the Edwards Curve Digital Signature Algorithm (EdDSA) [18]. It is birationally equivalent to Curve25519 [14]. We have

$$E : -x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2 \tag{5.65}$$

over the field $\mathbb{F}_q$ with

$$q = 2^{255} - 19. \tag{5.66}$$

This is a *twisted Edwards curve*. In general, a twisted Edwards curve has the form

$$E_{E,a,d} : ax^2 + y^2 = 1 + dx^2y^2 \tag{5.67}$$

over a field K with $a, d \in K$ distinct. There is a minor technical restriction on the field K which we do not mention.

5.2.11 Encoding Elliptic Curves over Finite Fields

When working with the elliptic curves $E/\mathbb{F}_p$, we can encode (x, y) using its binary representation; encoding the identity element $\mathcal{O}$ may require care. It may be useful to use point compression described in Chapter 5.2.9 depending on the situation.

5.3 Bilinear Pairings

5.3.1 Why do we care about Bilinear Pairings?

We care about bilinear pairings because they give us a lot of nice mathematical features that we can use. In particular, bilinear pairings give us *short digital signatures*. These started with BLS signatures based on the Weil pairing [29], a particular type of bilinear pairing.

Bilinear pairings also allow for threshold group signatures. These can be formed as part of a distributed key generation protocol and are discussed in Chapter 14.

5.3.2 Formal Definition

A bilinear pairing is a special type of function:

Definition 5.1 (Bilinear Pairing)**.** We let G_1, G_2, and G_T be finite groups with $|G_1| = |G_2| = |G_T|$. We say $e : G_1 \times G_2 \to G_T$ is a *bilinear pairing* if for all $h_1 \in G_1$, $h_2 \in G_2$, and $a, b \in \mathbb{Z}$ we have

$$e\left(h_1^a, h_2^b\right) = \left[e\left(h_1, h_2\right)\right]^{ab}. \tag{5.68}$$

5.3.3 Discussion

It is straightforward to *use* the previous definition. The challenge is *finding* groups with which to build such a bilinear pairing. This is nontrivial.

At this point, all bilinear pairings arise from elliptic curves. The exact construction is complex and we do not discuss it here. We will use bilinear pairings in Chapter 12 when we discuss Pairing-Based Cryptography. Having a solid understanding of the mathematics behind pairings and how they are constructed from elliptic curves requires advanced knowledge of elliptic curves as discussed in [120].

5.4 Lagrange Interpolation

We spend some time discussing Lagrange Interpolation.

5.4.1 Why do we care about interpolation?

Within cryptography, *Lagrange Interpolation* over finite fields is used in secret sharing protocols and distributed key generation. More generally, interpolation is useful because it attempts to approximate a complex function with a simpler polynomial.

We will start by looking at interpolation over real data. After working through examples, we will transition to interpolation over finite fields as this is our primary focus.

5.4.2 Lagrange Interpolation over the Reals

Given a set of data points $\{(x_k, y_k)\}_{k=1}^n$ with $\{x_k\}_{k=1}^n$ distinct, we want to find the interpolating polynomial of minimal degree which agrees with the data. That is, we want a polynomial p of at most degree $n-1$ such that

$$p(x_k) = y_k, \quad k \in \{1, \cdots, n\}. \tag{5.69}$$

We now proceed to construct such a polynomial. To do this, we set

$$
\begin{aligned}
L(x) &:= \sum_{k=1}^{n} y_k \ell_k(x) \\
\ell_k(x) &:= \prod_{\substack{1 \le j \le n \\ j \neq k}} \frac{x - x_j}{x_k - x_j} \\
&= \frac{x - x_1}{x_k - x_1} \cdot \frac{x - x_2}{x_k - x_2} \cdots \frac{x - x_{k-1}}{x_k - x_{k-1}} \cdot \frac{x - x_{k+1}}{x_k - x_{k+1}} \cdots \frac{x - x_n}{x_k - x_n}.
\end{aligned} \tag{5.70}
$$

We note that ℓ_k is a polynomial of degree $n-1$; this implies that L is a polynomial of degree at most $n-1$. We note that

$$
\begin{aligned}
\ell_k(x_j) &= \begin{cases} 1, & k = j \\ 0, & k \neq j \end{cases} \\
&= \delta_{kj}.
\end{aligned} \tag{5.71}
$$

Here, δ_{kj} is the Kronecker delta. It follows that

$$
\begin{aligned}
L(x_j) &= \sum_{k=1}^{n} y_k \ell_k(x_j) \\
&= \sum_{k=1}^{n} y_k \delta_{kj} \\
&= y_j.
\end{aligned} \tag{5.72}
$$

Thus, this is the interpolating polynomial which agrees with the data. Furthermore, the degree of the polynomial is at most $n-1$. Such polynomials are unique.

5.4.3 Examples of Lagrange Interpolation over the Reals

Example 5.9

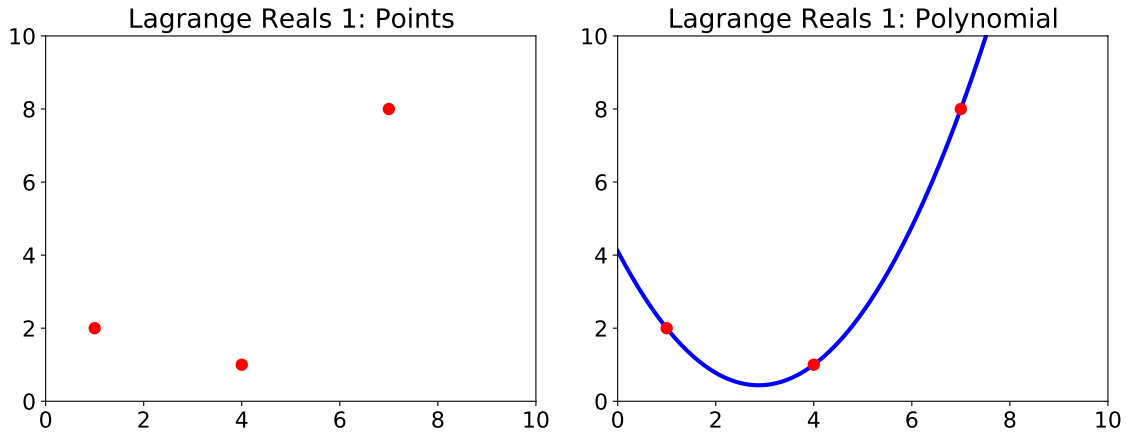
Code may be found at `examples/math_review/lagrange_reals_1.py`.

We begin by interpolating the data $\{(1,2),(4,1),(7,8)\}$. See the data in Figure 5.11a.

We start by computing the ℓ_k polynomials:

$$
\begin{aligned}
\ell_1(x) &= \frac{x-4}{1-4} \cdot \frac{x-7}{1-7} \\
\ell_2(x) &= \frac{x-1}{4-1} \cdot \frac{x-7}{4-7} \\
\ell_3(x) &= \frac{x-1}{7-1} \cdot \frac{x-4}{7-4}.
\end{aligned} \tag{5.73}
$$

In this case, we have the interpolating polynomial



(a) Lagrange interpolation data points (b) Lagrange interpolation polynomial

Figure 5.11: Here are data points and Lagrange interpolating polynomial for Example 5.9.

$$L(x) = 2 \cdot \frac{x-4}{1-4} \cdot \frac{x-7}{1-7} + 1 \cdot \frac{x-1}{4-1} \cdot \frac{x-7}{4-7} + 8 \cdot \frac{x-1}{7-1} \cdot \frac{x-4}{7-4}. \tag{5.74}$$

This may be reduced to

$$L(x) = \frac{1}{9}\left[4x^2 - 23x + 37\right]. \tag{5.75}$$

The data points with this polynomial can be found in Figure 5.11b. We see that

$$\begin{aligned} L(1) &= 2 \\ L(4) &= 1 \\ L(7) &= 8. \end{aligned} \tag{5.76}$$

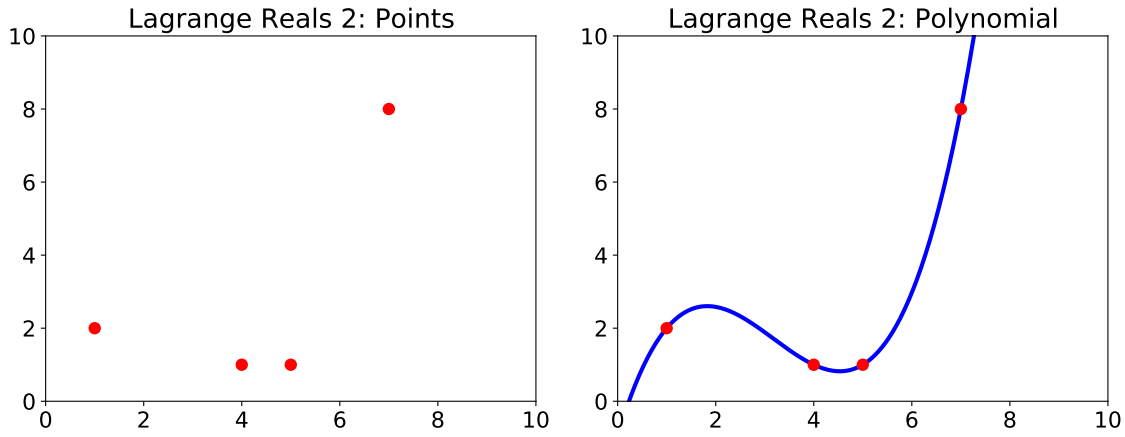
Example 5.10

Code may be found at `examples/math_review/lagrange_reals_2.py`.

We interpolate the data $\{(1,2),(4,1),(5,1),(7,8)\}$; this is the same data from the previous example with the additional point $(5,1)$. See the data in Figure 5.12a.

We again compute the ℓ_k polynomials:

$$\begin{aligned} \ell_1(x) &= \frac{x-4}{1-4} \cdot \frac{x-5}{1-5} \cdot \frac{x-7}{1-7} \\ \ell_2(x) &= \frac{x-1}{4-1} \cdot \frac{x-5}{4-5} \cdot \frac{x-7}{4-7} \\ \ell_3(x) &= \frac{x-1}{5-1} \cdot \frac{x-4}{5-4} \cdot \frac{x-7}{5-7} \\ \ell_4(x) &= \frac{x-1}{7-1} \cdot \frac{x-4}{7-4} \cdot \frac{x-5}{7-5}. \end{aligned} \tag{5.77}$$



(a) Lagrange interpolation data points (b) Lagrange interpolation polynomial

Figure 5.12: Here are data points and Lagrange interpolating polynomial for Example 5.10.

The resulting Lagrange interpolation polynomial is

$$L(x) = \frac{1}{72}\left[13x^3 - 124x^2 + 323x - 68\right]. \tag{5.78}$$

The data points with this polynomial can be found in Figure 5.12b.

5.4.4 Problems with Lagrange Interpolation over the Reals

We note that using Lagrange Interpolation can lead to problems when attempting to approximate certain types of functions (data). In particular, even with smooth functions (functions with infinitely many derivatives), it is possible that the polynomial approximation given by Lagrange Interpolation does not converge to the underlying function. An example of this is shown in Figure 5.13; this is called *Runge's Phenomenon*.

There are mathematical reasons which can explain this, but *none* of those difficulties matter to us because we are not particularly interested in interpolating functions over $\mathbb{R}$. Rather, we are interested in interpolating polynomials over finite fields.

5.4.5 Lagrange Interpolation over Finite Fields

We are particularly interested in performing Lagrange Interpolation over finite fields. This will be used in secret sharing protocols.

There is effectively no difference between interpolation over $\mathbb{R}$ and interpolation over $\mathbb{F}_p$. This is because all of the operations in $\mathbb{R}$ directly translate into the equivalent operations in $\mathbb{F}_p$.

5.4.6 Examples of Lagrange Interpolation over Finite Fields

We use the same examples as before except that now we use the finite field $\mathbb{F}_{73}$.

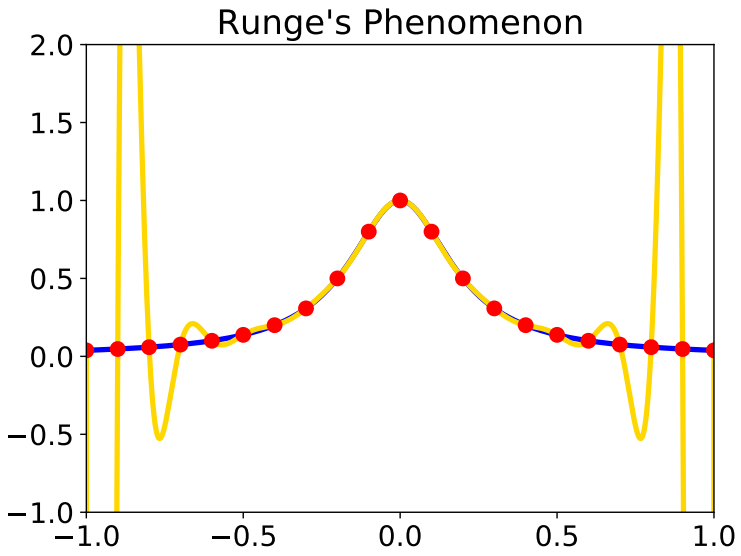


Figure 5.13: Here is an example of Runge's phenomenon. Even though the polynomial approximation of the function $f(x) = [1 + 25x^2]^{-1}$ is accurate close to 0, the approximation is very poor near 1 and -1.

Example 5.11
Code may be found at `examples/math_review/lagrange_finite_1.py`.

We begin by interpolating the data $\{(1, 2), (4, 1), (7, 8)\}$ as before in Example 5.9; in this case, we are interpolating over $\mathbb{F}_{73}$. See the data in Figure 5.14a.

In this case, we have the interpolating polynomial

$$L(x) = 41x^2 + 38x + 69. \tag{5.79}$$

The data points with this polynomial can be found in Figure 5.14b. We see that

$$\begin{aligned} L(1) &= 2 \\ L(4) &= 1 \\ L(7) &= 8. \end{aligned} \tag{5.80}$$

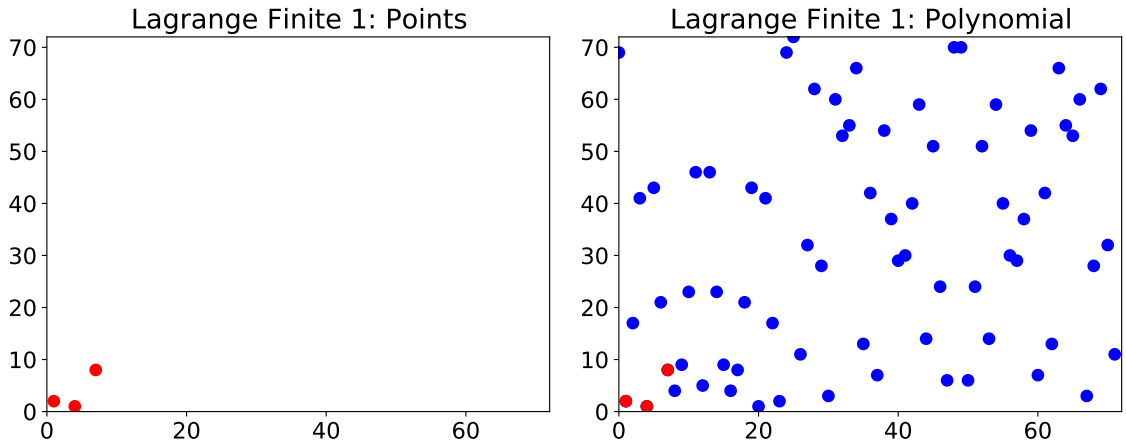
Example 5.12
Code may be found at `examples/math_review/lagrange_finite_2.py`.

We interpolate the data $\{(1, 2), (4, 1), (5, 1), (7, 8)\}$ as in Example 5.10; in this case, we are interpolating over $\mathbb{F}_{73}$. See the data in Figure 5.15a.

In this case, we have the interpolating polynomial

$$L(x) = 60x^3 + 51x^2 + 42x + 68. \tag{5.81}$$

The data points with this polynomial can be found in Figure 5.15b. We see that



(a) Lagrange interpolation data points (b) Lagrange interpolation polynomial

Figure 5.14: Here are data points and Lagrange interpolating polynomial for Example 5.11. This example uses the finite field $\mathbb{F}_{73}$.

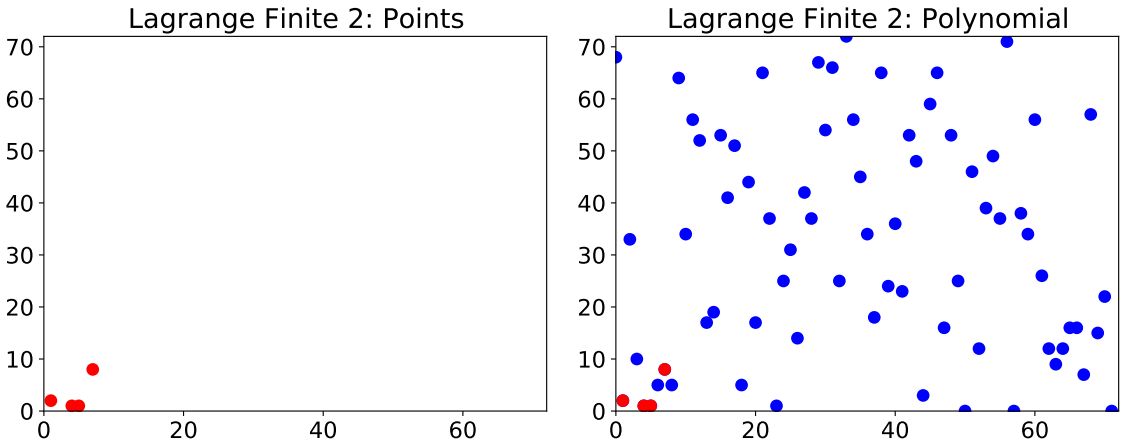
$$\begin{aligned} L(1) &= 2 \\ L(4) &= 1 \\ L(5) &= 1 \\ L(7) &= 8. \end{aligned} \tag{5.82}$$

5.4.7 Further Generalizations of Lagrange Interpolation

It is possible to generalize Lagrange Interpolation more. In particular, by looking at Eq. (5.70), we see that all that is required of the data $\{(x_i, y_i)\}_{i=1}^n$ is that $\{x_i\}$ must be elements of a field $\mathbb{F}$ and $\{y_i\}$ must be elements where it makes sense to perform multiplication by $\mathbb{F}$. Although this may appear to be an unnecessary generalization, this will be used when computing threshold digital signatures in Chapter 14.6; in that setting, we are essentially interpolating over group elements parameterized by elements in a finite field.

5.5 Conclusion of Mathematical Review

A lot of important and difficult material was covered in the previous chapters. There is still more mathematics that could be learned and that would be useful for cryptography. With that said, these are the main ideas and the most useful. Studying additional mathematics would be recommended to anyone who wants to dive deeper into cryptography; see Chapter 16 for additional resources.



(a) Lagrange interpolation data points (b) Lagrange interpolation polynomial

Figure 5.15: Here are data points and Lagrange interpolating polynomial for Example 5.12. This example uses the finite field $\mathbb{F}_{73}$.

Chapter 6

Symmetric Key Cryptography

We now begin our discussion of Symmetric Key Cryptography, which involves one secret key. This distinguishes it from Public Key Cryptography, which we will discuss in Chapter 9, which uses *two* keys: one public key and one private key.

6.1 The Need for Symmetric Key Cryptography

Alice and Bob would like to communicate with each other over an insecure channel. In particular, they know that eavesdropper Eve will frequently be intercepting their communication, and they do not want her to be find out what they are saying.

Throughout Symmetric Key Cryptography, we assume that Alice and Bob have a shared secret key that they can use for communication. This secret key has been shared through a secure channel; in particular, Eve does *not* know what the key is, although she does know everything about the encryption scheme they are using. This ensures that Alice and Bob follow *Kerchoff's principle* [69, Page 5]:

> The cipher method must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience.

The messages that Alice and Bob send to each other shall be called the *plaintext*. When the plaintext has been encrypted, the resulting encrypted message shall be called the *ciphertext* (also spelled *cyphertext*).

6.2 Unbreakable Encryption: the One-Time Pad

6.2.1 Definition

Suppose Alice has a message $m \in \{0,1\}^L$ to send Bob and wants to encrypt m. Further suppose that Alice previously shared a secret key $s \xleftarrow{\$} \{0,1\}^L$ with Bob. Here, the key s is a uniformly random bit string. To encrypt m using the one-time pad, the ciphertext c is

$$c \coloneqq s \oplus m, \tag{6.1}$$

where $\oplus$ is the XOR operation. She then sends the ciphertext c to Bob over an insecure channel (like the Internet). Because Bob also knows the secret key s, he is able to recover the message m by the XOR operation:

$$\begin{aligned} s \oplus c &= s \oplus (s \oplus m) \\ &= m. \end{aligned} \tag{6.2}$$

6.2.2 Examples

We now look at some examples of encryption using the one-time pad.

Example 6.1 (One-Time Pad 1)
Code may be found at `examples/symmetric/one-time_pad_1.py`.

We want to encrypt the message

$$m = \texttt{00112233445566778899aabbccddeeff} \tag{6.3}$$

using the one-time pad with the secret key

$$s = \texttt{aa31553ea3fd3f015c4bffac26447514}. \tag{6.4}$$

The resulting ciphertext is

$$\begin{aligned} c &= s \oplus m \\ &= \texttt{aa20770de7a85976d4d25517ea999beb}. \end{aligned} \tag{6.5}$$

Decrypting the message, we see

$$\begin{aligned} c \oplus s &= \texttt{00112233445566778899aabbccddeeff} \\ &= m, \end{aligned} \tag{6.6}$$

as expected. See Listing 6.1.

Example 6.2 (One-Time Pad 2)
Code may be found at `examples/symmetric/one-time_pad_2.py`.

We provide another example of encryption using the one-time pad. Here, we encrypt the same message

$$m = \texttt{00112233445566778899aabbccddeeff} \tag{6.7}$$

using a different secret key:

$$s = \texttt{09fa37f922072f9cd36cfbc4936d5712}. \tag{6.8}$$

A different key produces a different ciphertext:

Listing 6.1: Encryption with the one-time pad 1

```
#!/usr/bin/env python3

message = bytes.fromhex('00112233445566778899aabbccddeeff')
key =     bytes.fromhex('aa31553ea3fd3f015c4bffac26447514')

ciphertext = bytes(a ^ b for a,b in zip(key, message))
# aa20770de7a85976d4d25517ea999beb

message_prime = bytes(a ^ b for a,b in zip(key, ciphertext))
# 00112233445566778899aabbccddeeff

assert message == message_prime # Valid
```

Listing 6.2: Encryption with the one-time pad 2

```
#!/usr/bin/env python3

message = bytes.fromhex('00112233445566778899aabbccddeeff')
key =     bytes.fromhex('09fa37f922072f9cd36cfbc4936d5712')

ciphertext = bytes(a ^ b for a,b in zip(key, message))
# 09eb15ca665249eb5bf5517f5fb0b9ed

message_prime = bytes(a ^ b for a,b in zip(key, ciphertext))
# 00112233445566778899aabbccddeeff

assert message == message_prime # Valid
```

$$\begin{aligned} c &= s \oplus m \\ &= \texttt{09eb15ca665249eb5bf5517f5fb0b9ed}. \end{aligned} \tag{6.9}$$

Decrypting the message, we see

$$\begin{aligned} c \oplus s &= \texttt{00112233445566778899aabbccddeeff} \\ &= m, \end{aligned} \tag{6.10}$$

as expected. See Listing 6.2.

6.2.3 Discussion

The one-time pad is a symmetric key encryption scheme that *cannot* be broken. This is called *perfect security*, *information-theoretic security*, or *unconditional security*. Even with *unlimited computation*, the scheme *cannot be broken*. A proof that the one-time pad is perfectly secure can be found in [69, Theorem 2.10]. It is also possible to prove that perfect security is possible only when the total number of keys is greater than or equal to the total number of messages; see [69, Theorem 2.11]. From this, it follows that any encryption scheme with perfect security must be equivalent to the one-time pad

The referenced proofs rely on the fact that s is a uniformly random bit string the same size of the message. The secret key s must be *truly random*; anything less is *not sufficient*. Acquiring true randomness is difficult. This shows the cost of perfect security: the secret key must be a truly random secret shared beforehand and the same size as the message. This is why other encryption schemes are used in practice: *computational security* enables smaller secret keys.

Anyone who claims to have an unbreakable encryption scheme must either have the equivalent of a one-time pad or he is lying; there are no other possibilities.

6.3 Encryption Schemes

We recall that the primary goal of all encryption schemes is to convert a plaintext message into a ciphertext which is indistinguishable from a random bit string. This random-looking ciphertext is then transmitted over an insecure channel. Upon receiving the ciphertext, it may be decrypted back into the original message.

There are two primary classes of symmetric key encryption algorithms: stream ciphers and block ciphers. Stream ciphers act on the message one bit at a time (in a stream) while block ciphers operate on chunks (or blocks) of bits at a time.

We mention that encryption schemes do not provide *message integrity*; that is, encrypting a message does not ensure that the message was not corrupted during transit. See Chapter 6.7 for more information about message integrity using a message authentication code.

The discussion here is meant to give a high-level overview of stream ciphers and block ciphers. The focus is on general ideas, not current best practices and a thorough description of specific algorithms. In particular, *do not attempt to write encryption schemes based on the material presented here.*

6.3.1 Stream Ciphers

Throughout this section, let $L > 0$ denote the length of the message in bits. In general, we will assume that L is significantly larger than our k-bit keys.

Discussion

In a certain sense, stream ciphers are a natural extension of the one-time pad. Here, a secret key is stretched into a cryptographically secure stream of bits. More formally, let $G : \{0,1\}^k \to \{0,1\}^L$. Then given a secret key $s \xleftarrow{\$} \{0,1\}^k$, the ciphertext is

$$c = G(s) \oplus m. \tag{6.11}$$

Because we are assuming Bob also knows s, he can decrypt the ciphertext:

$$m = G(s) \oplus c. \tag{6.12}$$

In practice, an *initialization vector* may used in a stream cipher. In this case, we would have $G : \{0,1\}^k \times \{0,1\}^\ell \to \{0,1\}^L$. We would then choose the secret key $s \xleftarrow{\$} \{0,1\}^k$ and initialization vector $\mathsf{IV} \xleftarrow{\$} \{0,1\}^\ell$. The resulting ciphertext would be

$$c = G(s, \mathsf{IV}) \oplus m. \tag{6.13}$$

In this case, while s must be kept secret, the initialization vector IV is sent in the clear (unencrypted) along with the ciphertext c. This enables the reuse of the secret key and also ensures that encrypting the same message under the same secret key but different initialization vector results in a completely different ciphertext. Different protocols have different requirements of initialization vectors. In every case, an initialization vector should *never* be reused.

The important part of a stream cipher is the function G. This function must produce an output which is impractical to distinguish from true randomness. This should hold for every secret key and initialization vector combination.

Initialization vectors are related to nonces. A nonce is a ***n****umber used only* ***once***. Nonces should *never* be reused; nonce reuse may *break* the cryptosystem by leaking the private key.

Examples

Stream ciphers operate on the plaintext in a continuous manner. Some standard examples include RC4 [108], Salsa20 [17], and ChaCha [13].

RC4 Designed by Rivest in 1987 for RSA Security; originally a trade secret, the algorithm was made public in 1994 after being reverse engineered [108]. RC4 *should not be used* because it is a cryptographically-broken stream cipher [101, 108].

Salsa20 In the case of Salsa20, the secret key is 256 bits and the nonce is 64 bits [17]; a version with a 192 bit nonce is also available [15]. Developed by Daniel Bernstein in 2005, the cipher uses only additions, rotations, and XOR operations; ciphers which only use these operations are called *ARX ciphers*.

ChaCha The design of ChaCha [13] is a modification of Salsa20 cipher; ChaCha was also designed by Bernstein. While the original specification uses a 64 bit nonce, one standardization uses a 96 bit nonce [94].

6.3.2 Block Ciphers

Discussion

As mentioned previously, a block cipher acts on blocks of bits at a time. The main property of block ciphers is that they are *keyed* permutations. We suppose that the secret key is k bits and the block is n bits. In this case, the block cipher would be a function $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ such that for all $s \in \{0,1\}^k$, $E(s,\cdot) : \{0,1\}^n \to \{0,1\}^n$ is a permutation which appears random.

In order to encrypt a message $m \in \{0,1\}^n$, Alice chooses a secret key $s \xleftarrow{\$} \{0,1\}^k$; Bob receives s through a secure channel. The resulting ciphertext is

$$c \coloneqq E(s,m). \tag{6.14}$$

Because $E(s,\cdot)$ is a permutation (a bijection), it has an inverse permutation $D(s,\cdot)$. Thus, Bob can now decrypt the ciphertext to retrieve the plaintext:

$$m = D(s,c). \tag{6.15}$$

Different methods (called modes) need to be used to encrypt messages longer than n bits. There are ways to combine the plaintexts and ciphertexts to thoroughly scramble the inputs and have the ciphertext look like a random bit string. Some modes of operation require the use of an initialization vector.

We mention one mode that *should not be used:* Electronic Codebook (ECB). It *should not be used* because it does not thoroughly scramble information. If we wish to encrypt a message with blocks m_1, m_2, and m_3, then encrypting with ECB mode would be

$$\begin{aligned} c_1 &\coloneqq E(s,m_1) \\ c_2 &\coloneqq E(s,m_2) \\ c_3 &\coloneqq E(s,m_3), \end{aligned} \tag{6.16}$$

where s is the secret key. This leaks too much information because, generally speaking, the message blocks m_i will be highly structured. Also, if $m_i = m_j$ for $i \neq j$, then we clearly will have $c_i = c_j$. Any secure block cipher mode will not leak this information. We end with one quote about ECB [51, Chapter 4.2]:

> Do not ever use ECB for anything. It has serious weaknesses, and is only included here so that we can warn you away from it.

Instead of using ECB, consider the algorithm in Figure 6.1.

Examples

Some standard examples of block ciphers include AES [123], DES [122], Twofish [117], and Blowfish [116].

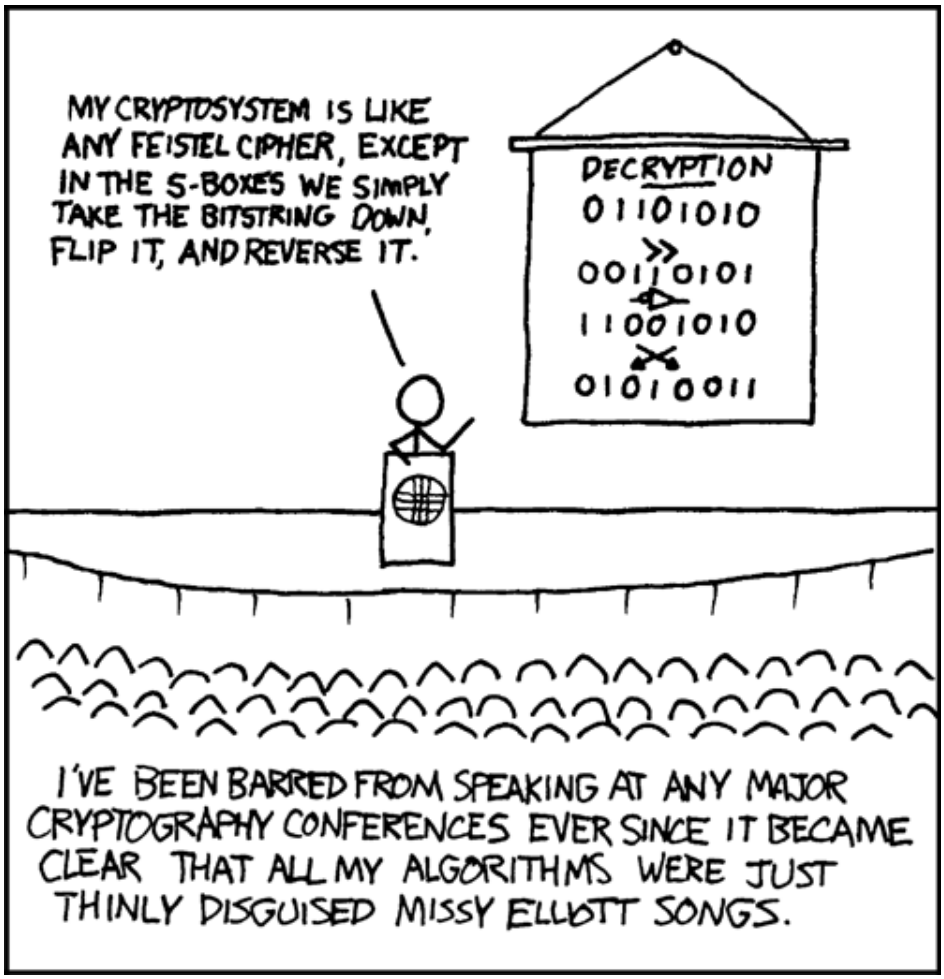


Figure 6.1: Here we have an example of a symmetric key encryption algorithm. Created by Randall Munroe on `xkcd`; posted online at https://xkcd.com/153/.

AES The *Advanced Encryption Standard* has a block size of 128 bits and key sizes of 128, 192, and 256 bits [123]. AES uses the Rijndael cipher [41]. When using AES, care should be taken so that timing information does not leak the private key [12, 133].

DES The *Data Encryption Standard* was originally standardized in 1977 [122]. DES should not be used because it is a cryptographically-broken block cipher [70]; it has a block size of 64 bits and a key size of 56 bits. The small block size and key size makes it vulnerable to attacks. and *should not be used.* AES was in fact designed to replace DES.

Triple DES (3DES) is a variant of DES which performs encryption using three different keys [5, 68]. While it is still currently approved for certain instances ([5] was published in 2017), it is "Depreciated through 2023" and "Disallowed after 2023" [6]; it was also recommended that 3DES be depreciated in 2018 [64].

All of this points to the fact that DES and Triple DES *should not be used.*

Twofish A block cipher designed by Bruce Schneier in 1998 as part of the AES Competition [117]. Twofish made it to the final round of the competition but was not selected as the standard.

Blowfish Blowfish is a cipher designed by Bruce Schneier in 1993 [116]. Its old design and small block size (64 bits like DES) means it should not be be used. In 2007, Schneier recommended[1] switching from Blowfish to Twofish.

6.4 Cryptographic Hash Functions

Cryptographic hash functions are very important in cryptography. They are discussed in Chapters 7 and 8.

6.5 Key Derivation Functions

A *key derivation function* (KDF) is a function which takes a key (perhaps a password, passphrase, or shared secret) and uses it to derive *cryptographic* keys. While the input key material may not be uniformly random bit strings, the derived cryptographic keys *are* uniformly random bit strings. This is important because cryptographic protocols usually assume that the cryptographic keys are uniformly random. In this way, using the raw input key as a cryptographic key would decrease the security of the system.

The algorithm *assumes* the input key has sufficient entropy; problems arise when this assumption is violated. In practice, a salt may also be used so that repeated input keys produce independent derived keys. Key derivation functions based on hash functions are discussed in Chapter 8.8. Key derivation functions may also be used to store passwords. This aspect is discussed in Chapter 8.10.

6.6 Cryptographically-Secure Pseudorandom Number Generators

6.6.1 Discussion

In the cryptographic setting, the pseudorandom number generators that are important are *cryptographically-secure pseudorandom number generator* (CSPRNGs). These are random number generators where it is impractical to guess the next bit better than average. This is *not* the case for standard pseudorandom number generators such as the Linear Congruential Generator [74, Chapter 3.2.1], Permuted Congruential Generator [95], or the Mersenne Twister [85]. Their focus is on certain statistical properties and passing certain statistical tests; this is necessary but *not sufficient* in cryptography. In particular, we note that the

[1] https://www.schneier.com/news/archives/2007/12/bruce_almighty_schne.html

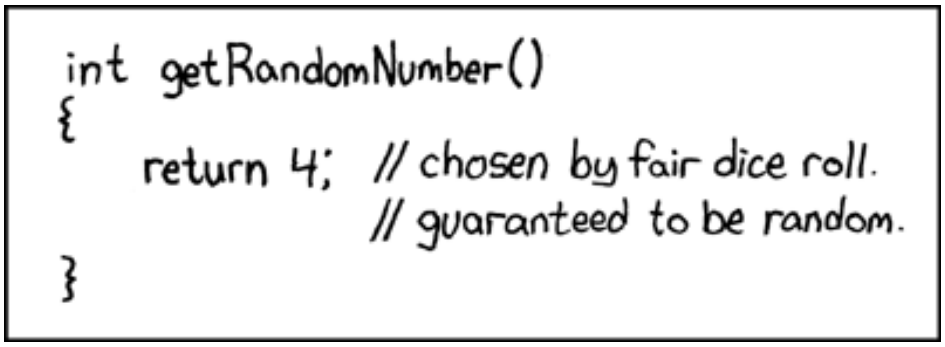


Figure 6.2: A secure random number generator written in the C programming language. Created by Randall Munroe on `xkcd`; posted online at https://xkcd.com/221/.

outputs of LCGs *fall in planes* [84] and that the internal state recovery of PCGs is *practical* [31]. Also, we note that an online challenge[2] is devoted to cloning the internal state of the Mersenne Twister based solely on its output.

Cryptographically-secure pseudorandom number generators are similar to stream ciphers, in that they take an initial random seed and stretch it into random output. The difference is that cryptographically-secure pseudorandom number generators are able to be reseeded periodically with additional randomness so that it is impractical to determine previous outputs.

We mention that even the *notion* of randomness is quite complex and we will not discuss it further; one reference for further discussion is [74, Chapter 3.5].

6.6.2 Examples

Some examples of cryptographically-secure pseudorandom number generators are Fortuna [50, Chapter 10][51, Chapter 9], Hash_DRBG [8, Section 10.1.1], and HMAC_DRBG [8, Section 10.1.2]. In general, individuals should not worry about specific cryptographically-secure pseudorandom number generators functions. It should be possible to perform a generic call to retrieve cryptographically random bits within a cryptographic library; this is sufficient for normal users. Do *not* use Dual_EC_DRBG [7, Section 10.3.1], as the standard implementation is thought to have a backdoor [21]. Figure 6.2 shows an example of a secure software random number generator while Figure 6.3 shows an example of a secure hardware random number generator.

Cryptographically-secure pseudorandom number generators are used for constructing private keys, initialization vectors, nonces, and related numbers which need to be drawn from a uniform bit distribution. They are also used when constructing large prime numbers.

Unless an algorithm has been *specifically designed* to be a cryptographically-secure pseudorandom number generators, it *should not used* when cryptographic random numbers are required. Otherwise, this could happen: your Ethereum private keys could be brute-forced due to insufficient randomization[3]. *You have been warned.*

[2] https://cryptopals.com/sets/3/challenges/23
[3] https://github.com/johguse/profanity/issues/61

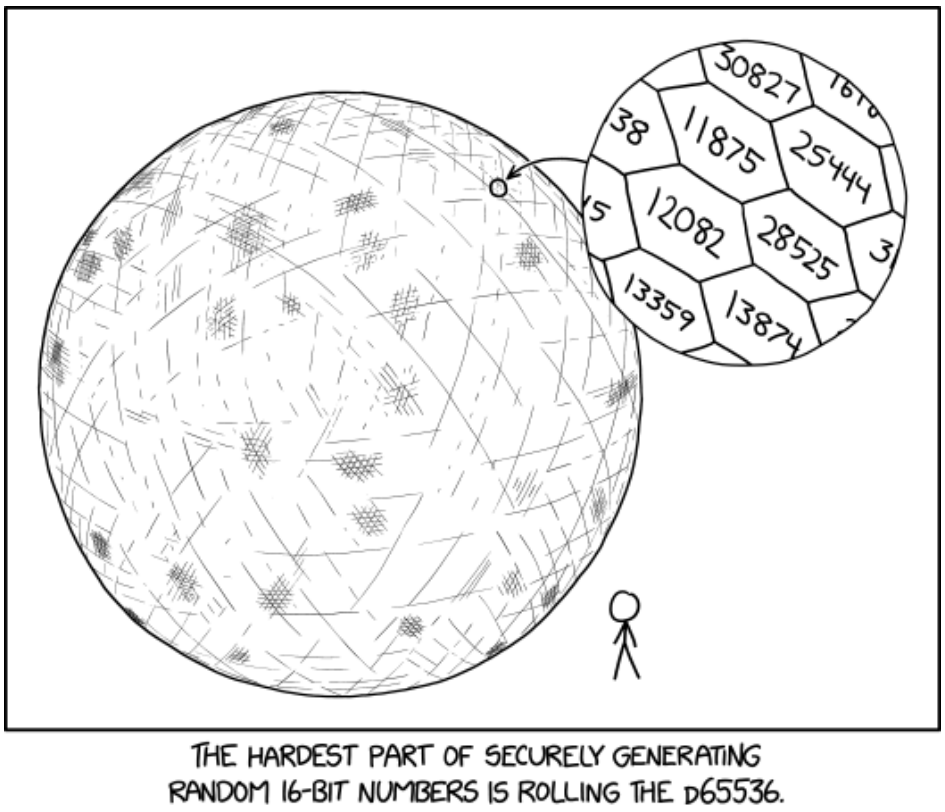


Figure 6.3: A secure hardware random number generator. Created by Randall Munroe on `xkcd`; posted online at https://xkcd.com/2626/.

6.7 Message Authentication Codes

6.7.1 Discussion

A *message authentication code* (MAC) ensures the integrity of a message. This occurs through the use of a shared secret key between parties.

We suppose that Alice and Bob have a shared key used for message authentication. In this case, Alice would take her message, compute a *tag* t using her secret key k and the message m, and send the pair (m, t) to Bob. Bob would be able to use his copy of the secret key to validate the pair (m, t). If the tag is valid, then Bob can be certain that Alice sent him the message m and that it was not corrupted during transit.

It is important to note that message authentication codes do not ensure nonrepudiation; that is, if Alice sends Bob a message with valid tag, Bob cannot prove to Charlie that Alice actually sent him the message. This is because Bob also has access to the (shared) secret key, and he could have constructed a valid tag for this message. Additionally, he would have to share his secret key with Charlie for him to validate the message. Nonrepudiation is possible with *digital signatures*. Digital signatures are briefly mentioned in Chapter 9 and discussed more thoroughly in Chapter 10. We also wish to emphasize that message authentication codes do not involve encryption; in the example above, Alice sent the raw message to Bob. Combining encryption with message integrity results in *authenticated encryption*

and is discussed below in Chapter 6.8.

6.7.2 Examples

One example of a message authentication code is Poly1305 [16]. A generic way to build a message authentication code is based on a hash function: a *Hash-based Message Authentication Code*, or HMAC. HMAC based on the SHA-2 cryptographic hash function would be written as HMAC-SHA-2. The HMAC construction is briefly discussed in Chapter 8.7; a more thorough discussion can be found in Appendix A.2.1.

6.8 Authenticated Encryption

It is important to note that encryption, by itself, does *not* ensure message integrity; that is, nothing in the encryption process ensures that the entire message was correctly delivered. *Authenticated encryption* is the process of accomplishing both privacy and integrity by using an encryption scheme with a message authentication code. In practice, this combination is highly desirable. One example is using the ChaCha20 stream cipher and Poly1305 message authentication code [94, 103].

Chapter 7

Cryptographic Hash Functions

This chapter is devoted to introducing cryptographic hash functions. Hash functions are used all over cryptography.

Note: In Chapters 7 and 8, we will frequently use the MD5 and SHA-1 hash functions in the examples due to their small output size. As explained below, MD5 and SHA-1 *should never be used in practice.*

7.1 The Need for Cryptographic Hash Functions

In cryptography, we want to be able to work with arbitrary data; the challenge, of course, is that it may be difficult to design algorithms which can handle arbitrary-length data. It would be better if, instead of working with the raw data, we would work with a "fingerprint" or "hash" of the data; in this way, algorithms would be able to work with identifiers of a fixed size. For this to be useful in a cryptographic setting, we would want hashes to be effectively unique. Furthermore, additional intuitive properties are desired: each hash should be easy to compute and appear random; the only way to determine a hash is to explicitly compute it; and it should be hard to find two different files with the same hash.

Thus, we are interested in cryptographic hash functions; these are hash functions with additional properties to defend against cryptographic adversaries.

7.2 Desired Properties of Cryptographic Hash Functions

In the cryptographic setting, the hash functions we are concerned about are generally called *cryptographic hash functions*. We let $H : \{0,1\}^* \to \{0,1\}^b$ be a cryptographic hash function with an output of b bits. We want H to satisfy a number of properties:

Easy to Compute Given an input x, we want to be able to easily compute $H(x)$.

Discussion: Most hash functions have some inherent message length limitations, but in practice these lengths are extremely long and so are treated as infinite.

Random Output Given an input x, we want $H(x)$ to be a "random" element of $\{0,1\}^b$.

Discussion: One of the important assumptions involving hash functions is that it should not be possible to obtain *any* information about the output without explicitly computing $H(x)$. In particular, even if x' is somehow related to x, $H(x')$ should be completely unrelated to $H(x)$ provided $x \neq x'$.

Preimage Resistance Given an output y, it should be impractical to find x such that

$$H(x) = y. \tag{7.1}$$

Discussion: We expect that every possible output has some input (in fact, many inputs) which maps to it, but the challenge is that it is not clear how to find such inputs efficiently. We expect that the best choice is just to try all possible inputs (or rather, around 2^b possible inputs) to find x.

Second-Preimage Resistance Given x_1, it should be impractical to find a distinct x_2 such that

$$H(x_1) = H(x_2). \tag{7.2}$$

Discussion: Given x_1, we realize it should be possible to find x_2 which results in the same hash output, but the emphasis here is that there should be no efficient method for finding such pairs. We expect the best choice would be to try around 2^b inputs to find x_2.

Collision Resistance It should be impractical to find any $x_1 \neq x_2$ such that

$$H(x_1) = H(x_2). \tag{7.3}$$

Discussion: We expect there to be no efficient method to find *any* pairs x_1 and x_2 that hash to the same output. We expect the best choice would be to try around $2^{b/2}$ inputs to find x_1 and x_2.

Although here we have restricted ourselves to hash functions from bit strings to bit strings, it is possible to expand the definition from any set to any other set. In particular, we look at hashing to elliptic curves in Chapter 12. Some standard hash functions will be discussed below in Chapter 7.5.

Note well: It is important to note that hashing data *does not*, by itself, have anything to do with encrypting said data. Additionally, the relationship between preimage resistance, second-preimage resistance, and collision resistance is intricate; see [19, 110] for extended discussions about the relationships between them.

7.3 Random Oracle: The Ideal Cryptographic Hash Function

A *random oracle* is an idealized form of a cryptographic hash function. Put another way, when cryptographers write proofs they may assume that all parties have access to a random oracle; in practice, the random oracle is instantiated with a specific hash function.

Suppose we have a random oracle $H : \{0,1\}^* \to \{0,1\}^b$. A random oracle $H(\cdot)$ has the property that, for any $x \in \{0,1\}^*$, the output $H(x)$ is uniformly distributed on $\{0,1\}^b$.

7.4 Additional Properties

There are additional properties that we want in a cryptographic hash function:

Correlation Resistance Even if x and y are correlated, $H(x)$ and $H(y)$ should be unrelated.

Discussion: This provides greater detail related to the **Random Output** property. It is common that correlated inputs are hashed. In fact, in some instances x and y may vary by only *one bit*, yet we want the outputs $H(x)$ and $H(y)$ to be completely unrelated.

Length-extension Resistance If $H(x) = H(y)$ with $x \neq y$, then for any bit string z of positive length, $H(x\|z)$ and $H(y\|z)$ are independent.

Discussion: Again, this provides greater detail related to the **Random Output** property. *There are hash functions which do not satisfy this property*; see Chapter 7.7.1.

7.5 Examples

We will now spend some time looking at the outputs of various hash functions. Code may be found at `examples/hash_functions/`. One way to gain some understanding about hash functions is to repeatedly hash data. For each hash function, we will look at three different examples:

1. "" (the empty byte slice);
2. "The quick brown fox jumps over the lazy dog" (converted into bytes)
3. "The quick brown fox jumps over the lazy dog." (converted into bytes; note the period).

Although the last 2 values are very similar, we would expect there to be drastic changes in the hash output; this happens in all the examples below. This drastic change is known as the *avalanche effect*: any change in input should result in a vastly different output, as we expect the outputs to be completely unrelated.

MD5 Designed by Ronald Rivest and published in 1992 [107] with an output of 128 bits (16 bytes). At this point, due to the small size and significant attacks against it, MD5 should no longer be used in any situation where a hash function is necessary. Example hashes can be found in Listing 7.1.

SHA-1 Designed by NIST and published in 1995 [124], SHA-1 has an output of 160 bits (20 bytes). Although more secure than MD5, there are also significant attacks against SHA-1 and it should not be used. Example hashes can be found in Listing 7.2.

SHA-2 Due to the attacks against SHA-1, NIST released SHA-2 in 2001 [125]. The standard had multiple algorithms with different output sizes; one commonly used algorithm is SHA-2-256 (also called SHA-256), which has a 256 bit (32 byte) output. Although still subject to certain attacks (described below), it is still considered secure. SHA-2-256 example hashes can be found in Listing 7.3.

SHA-3 The continued cryptanalysis of MD5, SHA-1, and SHA-2 caused NIST to have a competition (similar to the competition which produced the block cipher AES, the Advanced Encryption Standard [123]). In the end, Keccak [22] became the basis for SHA-3 [46]. Like SHA-2, there are multiple versions with different output lengths; a common one is SHA-3-256 with 256 bits (32 bytes) of output. SHA-3-256 example hashes can be found in Listing 7.4.

7.6 Domain Separation

It may happen that, due to resource constraints, only *one* hash function may be implemented. Sometimes it is desired to instantiate different hash functions for different uses. We can do this with *domain separation*: adding a value to specify the specific domain. This will ensure that the queries to the underlying hash function are unique regardless of input.

In particular, if H is a hash function, we can make two different hash functions by specifying

$$\begin{aligned} H_0(x) &:= H(\mathtt{0}\|x) \\ H_1(x) &:= H(\mathtt{1}\|x), \end{aligned} \tag{7.4}$$

where $\mathtt{0}$ and $\mathtt{1}$ are bits. When viewed as random oracles, this is not a problem. Care must be required when this is used in practice, though; this is because we can run into issues when working with hash functions in the real world. This is discussed more in Chapter 7.7.

7.7 Known Challenges with Certain Hash Functions

Here we briefly discuss some of the known difficulties which may arise when working with hash functions in practice.

Listing 7.1: Output from the MD5 hash function

```
#!/usr/bin/env python3

import hashlib

# Example MD5 hashes

# Hash [] (the empty byte array)
md5 = hashlib.md5()
string_data = ''
data = bytes.fromhex(string_data)
md5.update(data)
md5.hexdigest()
# d41d8cd98f00b204e9800998ecf8427e

# Hash 'The quick brown fox jumps over the lazy dog'
md5 = hashlib.md5()
string_data = 'The quick brown fox jumps over the lazy dog'
data = str.encode(string_data)
md5.update(data)
md5.hexdigest()
# 9e107d9d372bb6826bd81d3542a419d6

# Hash 'The quick brown fox jumps over the lazy dog.'
md5 = hashlib.md5()
string_data = 'The quick brown fox jumps over the lazy dog.'
data = str.encode(string_data)
md5.update(data)
md5.hexdigest()
# e4d909c290d0fb1ca068ffaddf22cbd0
```

Listing 7.2: Output from the SHA-1 hash function

```
#!/usr/bin/env python3

import hashlib

# Example SHA-1 hashes

# Hash [] (the empty byte array)
sha1 = hashlib.sha1()
string_data = ''
data = bytes.fromhex(string_data)
sha1.update(data)
sha1.hexdigest()
# da39a3ee5e6b4b0d3255bfef95601890afd80709

# Hash 'The quick brown fox jumps over the lazy dog'
sha1 = hashlib.sha1()
string_data = 'The quick brown fox jumps over the lazy dog'
data = str.encode(string_data)
sha1.update(data)
sha1.hexdigest()
# 2fd4e1c67a2d28fced849ee1bb76e7391b93eb12

# Hash 'The quick brown fox jumps over the lazy dog.'
sha1 = hashlib.sha1()
string_data = 'The quick brown fox jumps over the lazy dog.'
data = str.encode(string_data)
sha1.update(data)
sha1.hexdigest()
# 408d94384216f890ff7a0c3528e8bed1e0b01621
```

Listing 7.3: Output from the SHA-2-256 hash function

```
#!/usr/bin/env python3

import hashlib

# Example SHA-2 hashes

# Hash [] (the empty byte array)
sha2 = hashlib.sha256()
string_data = ''
data = bytes.fromhex(string_data)
sha2.update(data)
sha2.hexdigest()
# e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855

# Hash 'The quick brown fox jumps over the lazy dog'
sha2 = hashlib.sha256()
string_data = 'The quick brown fox jumps over the lazy dog'
data = str.encode(string_data)
sha2.update(data)
sha2.hexdigest()
# d7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592

# Hash 'The quick brown fox jumps over the lazy dog.'
sha2 = hashlib.sha256()
string_data = 'The quick brown fox jumps over the lazy dog.'
data = str.encode(string_data)
sha2.update(data)
sha2.hexdigest()
# ef537f25c895bfa782526529a9b63d97aa631564d5d789c2b765448c8635fb6c
```

Listing 7.4: Output from the SHA-3-256 hash function

```
#!/usr/bin/env python3

import hashlib

# Example SHA-3 hashes

# Hash [] (the empty byte array)
sha3 = hashlib.sha3_256()
string_data = ''
data = bytes.fromhex(string_data)
sha3.update(data)
sha3.hexdigest()
# a7ffc6f8bf1ed76651c14756a061d662f580ff4de43b49fa82d80a4b80f8434a

# Hash 'The quick brown fox jumps over the lazy dog'
sha3 = hashlib.sha3_256()
string_data = 'The quick brown fox jumps over the lazy dog'
data = str.encode(string_data)
sha3.update(data)
sha3.hexdigest()
# 69070dda01975c8c120c3aada1b282394e7f032fa9cf32f4cb2259a0897dfc04

# Hash 'The quick brown fox jumps over the lazy dog.'
sha3 = hashlib.sha3_256()
string_data = 'The quick brown fox jumps over the lazy dog.'
data = str.encode(string_data)
sha3.update(data)
sha3.hexdigest()
# a80f839cd4f83f6c3dafc87feae470045e4eb0d366397d5c6ce34ba1739f734d
```

7.7.1 Length Extension Attacks

The MD5, SHA-1, and SHA-2 hash functions are based on the Merkle-Damgård construction [43, 87]; this is discussed in Appendix A.1.1. In this case, given the length of x and $H(x)$ (but not x explicitly), there exists y such that for any z it is easy to compute

$$H(x\|y\|z). \tag{7.5}$$

Here, y is related to the internal padding of the underlying hash function. Because this padding is publicly known, it is trivial to determine it based solely on the length of the input. This implies that one must be careful in order to use hash functions in message authentication codes; see Chapter 8.7 for more information.

Not all hash functions are susceptible to this attack. Keccak and SHA-3 were designed to resist all known attacks; their resistance is based on the fact that they are built using a sponge construction (discussed in Appendix A.1.2) and not all their internal state is output. Similarly, SHA-2-512/256 is not affected because its internal state is truncated before output (512 bit internal state truncated to 256 bits for output). In these instances, a portion of the internal state is not output, so total internal state recovery would require guessing at least 256 bits thereby making this attack impractical.

7.7.2 Other Attacks

We note that the collision resistance of MD5 and SHA-1 are *significantly below* their original security level (respectively 64 bits and 80 bits). It is *practical* to find MD5 and SHA-1 hash collisions [82, 83, 88, 100, 128, 130]; preimage resistance has been attacked [113].

In order to show just how practical hash collisions are for MD5 and SHA-1, see Listing 7.5 for an MD5 hash collision [126] and Listing 7.6 and Figure 7.1 for a SHA-1 hash collision [127].

7.8 Properties of a Secure Cryptographic Hash Function

Ideally, a hash function will be as close to a random oracle as possible. So, a good hash function will have all of the properties from Chapters 7.2 and 7.4.

This is not always possible in practice. As discussed previously, MD5 and SHA-1 do not satisfy all of the standard hash properties (particularly collision resistance); this makes them insecure. Thus, MD5 and SHA-1 should only by used when *required* for legacy systems.

Due to length extension attacks, SHA-2 cannot satisfy all the properties of a random oracle. With that said, the best attacks against its collision resistance are no where close to working on the entire hash function (just weaker versions). In this way, SHA-2 is secure provided that care is taken to ensure that it is not possible to exploit length extension attacks. By design, SHA-3 is not susceptible to length extension attacks. This makes it closer to a random oracle.

In conclusion, it would be acceptable to use either SHA-2 or SHA-3 in situations where a cryptographic hash function is required. If using SHA-2, ensure that it is not possible to exploit length extension attacks.

Listing 7.5: Example MD5 Collision from Marc Stevens [126]. Each message is 64 bytes.

```
#!/usr/bin/env python3

import hashlib

# Example of MD5 collision

# Hash of message 1
md5 = hashlib.md5()
string_1 = '4dc968ff 0ee35c20 9572d477 7b721587 \
            d36fa7b2 1bdc56b7 4a3dc078 3e7b9518 \
            afbfa200 a8284bf3 6e8e4b55 b35f4275 \
            93d84967 6da0d155 5d8360fb 5f07fea2'
data_1 = bytes.fromhex(string_1)
md5.update(data_1)
hash_md5_1 = md5.hexdigest()
# 008ee33a9d58b51cfeb425b0959121c9
sha1 = hashlib.sha1()
sha1.update(data_1)
hash_sha1_1 = sha1.hexdigest()
# c6b384c4968b28812b676b49d40c09f8af4ed4cc

# Hash of message 2
md5 = hashlib.md5()
sha1 = hashlib.sha1()
string_2 = '4dc968ff 0ee35c20 9572d477 7b721587 \
            d36fa7b2 1bdc56b7 4a3dc078 3e7b9518 \
            afbfa202 a8284bf3 6e8e4b55 b35f4275 \
            93d84967 6da0d1d5 5d8360fb 5f07fea2'
data_2 = bytes.fromhex(string_2)
md5.update(data_2)
hash_md5_2 = md5.hexdigest()
# 008ee33a9d58b51cfeb425b0959121c9
sha1 = hashlib.sha1()
sha1.update(data_2)
hash_sha1_2 = sha1.hexdigest()
# c728d8d93091e9c7b87b43d9e33829379231d7ca

assert hash_md5_1  == hash_md5_2  # Passes
assert hash_sha1_1 != hash_sha1_2 # Passes
```

Listing 7.6: Example SHA-1 Collision from [127]. Each file is a valid PDF.

```
#!/usr/bin/env python3

import hashlib

# Example of SHA-1 collision

def md5(fname):
    hash_md5 = hashlib.md5()
    with open(fname, "rb") as f:
        for chunk in iter(lambda: f.read(4096), b""):
            hash_md5.update(chunk)
    return hash_md5.hexdigest()

def sha1(fname):
    hash_sha1 = hashlib.sha1()
    with open(fname, "rb") as f:
        for chunk in iter(lambda: f.read(4096), b""):
            hash_sha1.update(chunk)
    return hash_sha1.hexdigest()

file1 = "../../figures/hash_functions/shattered-1.pdf"
file2 = "../../figures/hash_functions/shattered-2.pdf"

# File 1
hash_md5_1  = md5(file1)
# ee4aa52b139d925f8d8884402b0a750c
hash_sha1_1 = sha1(file1)
# 38762cf7f55934b34d179ae6a4c80cadccbb7f0a

# File 2
hash_md5_2  = md5(file2)
# 5bd9d8cabc46041579a311230539b8d1
hash_sha1_2 = sha1(file2)
# 38762cf7f55934b34d179ae6a4c80cadccbb7f0a

assert hash_md5_1  != hash_md5_2  # Passes
assert hash_sha1_1 == hash_sha1_2 # Passes
```

(a) `shattered-1.pdf`

(b) `shattered-2.pdf`

Figure 7.1: Here are two different files (valid PDFs) which produce the same SHA-1 hash value: `38762cf7f55934b34d179ae6a4c80cadccbb7f0a`. See https://shattered.io/ for more information.

Chapter 8

Applications of Cryptographic Hash Functions

This chapter is devoted to applications of cryptographic hash functions.

8.1 Ethereum Addresses

This section assumes some knowledge of the Ethereum blockchain and elliptic curves.

8.1.1 Address Definition

We recall that Ethereum uses SECP256K1 for its Elliptic Curve Cryptography; its private keys are 256-bit nonnegative (unsigned) integers (32 bytes in length; 64 hexadecimal characters). Each private key k produces a public key

$$P = k \cdot G; \tag{8.1}$$

here, G is the specified base point of the elliptic curve. Uncompressed, the public key is two 256-bit unsigned integers (64 bytes). With the compressed or uncompressed designation byte, a public key is either 65 bytes uncompressed or 33 bytes compressed.

This is a significant amount of space. In order to save space, Ethereum [135, Eq. 314] defines the address of every public key. This is computed by performing the KECCAK hash of the public key and using the rightmost 20 bytes.

8.1.2 Colliding Addresses

While each private key corresponds to a unique public key, using only 20 bytes of a hash output reduces the unique number of addresses to 2^{160}. Because there are approximately 2^{256} unique private and public keys, this means there are *many* public keys with the same address; a rough estimate would be that there are approximately 2^{96} public keys per address.

The good news is that, due to KECCAK being a strong hash function with collision resistance, finding any 2 public keys which map to the same address would require the

computation of approximately 2^{80} hash values. Although large, it *may* be possible for this many hash operations to be computed by a *well-funded and dedicated* adversary.

In practice, an adversary would want to find a private key corresponding to a *particular* address. This converts a collision attack into a second-preimage attack. This instead requires approximately 2^{160} hash operations to find a valid public key. Considering the belief that performing 2^{128} operations is impractical, performing 2^{160} operations is also impractical. Thus, there is little concern for colliding addresses in Ethereum even though they certainly exist.

8.1.3 Example

Example 8.1 (Ethereum Address)
Code may be found at `examples/hash_applications/ethereum_addr.py`.

We have the private key

$$\begin{aligned} k &= 2^{248} + 1 \\ &= \texttt{0100000000000000000000000000000000000000000000000000000000000001}; \end{aligned} \tag{8.2}$$

the second line lists the private key in hexadecimal form. The corresponding public key in hexadecimal form is

$$\begin{aligned} &P = k \cdot G = \\ &(\texttt{e4dbb4350d84eabec1d67e40a398a78a8e6d719d86914393fca83b88dbe927af}, \\ &\texttt{b80fe66bf659859889a544623c945d0bd80d855f649e8c197be3aa41fe0390f8}). \end{aligned} \tag{8.3}$$

By computing the Keccak-256 hash of this and taking the rightmost 20 bytes, we see

$$\texttt{addr} = \texttt{7691ee0343b9a529675e1a8a70197b3b704f90b7}. \tag{8.4}$$

This address is much more manageable than the public key.

8.2 Commitment Schemes

A *commitment scheme* is a method to commit to a specific value without revealing any information about the value; at a later point in time, it is possible to reveal the committed value and prove that this particular value was the commitment. Ideally, a commitment scheme is *hiding* and *binding*: *hiding* because no information about the committed value is revealed; *binding* because the committed value cannot be changed after it is committed.

Suppose we have our hash function $H : \{0,1\}^* \to \{0,1\}^n$. If $m \in \{0,1\}^*$ is the value to be committed, we choose our random value $r \xleftarrow{\$} \{0,1\}^n$. The commitment is then

$$c \coloneqq H(r \| m). \tag{8.5}$$

Listing 8.1: Commitment scheme using MD5

```
#!/usr/bin/env python3

import hashlib

rPreStr = '00'*16
rPre = bytes.fromhex(rPreStr)
md5 = hashlib.md5()
md5.update(rPre)
r = md5.digest()
# 4ae71336e44bf9bf79d2752e234818a5

mStr = '01'
m = bytes.fromhex(mStr)
# 01

md5 = hashlib.md5()
md5.update(r)
md5.update(m)
c = md5.digest()
# ff95213ae708e6443c5f0ae2d8f5fe41
```

At a future point in time, revealing the value m and the random value r will show that m was the value previously committed. This is due to the fact that it is difficult to determine $r' \in \{0,1\}^n$ and $m' \in \{0,1\}^*$ such that $c = H(r' \| m')$.

Example 8.2 (Commitment Scheme)
Code may be found at `examples/hash_applications/commitment_scheme.py`.

We look at a commitment scheme using MD5; see Listing 8.1. We are seeking to commit to the byte value

$$m \coloneqq \texttt{01}. \tag{8.6}$$

The random value r is the MD5 hash of 16 zero bytes:

$$r \coloneqq \texttt{4ae71336e44bf9bf79d2752e234818a5}. \tag{8.7}$$

The resulting commitment c is then

$$\begin{aligned} c &\coloneqq \mathrm{MD5}\,(r \| m) \\ &= \texttt{ff95213ae708e6443c5f0ae2d8f5fe41}. \end{aligned} \tag{8.8}$$

At a later point in time, m and r may be revealed.

Note well: We note that in an actual commitment scheme, such a random value *should not be used*; the value should be truly random or drawn from a cryptographically-secure pseudorandom number generator.

It is not necessary that r be a bit string of length n in the commitment scheme; this was chosen for convenience. Even so, it must be long enough so that it is impractical to test every possible value for r and m. If the random value r was not included in the commitment, problems may arise. If the commitment was just $H(m)$, it may be possible to compute all possible values and determine which value is committed; thus, the scheme would not be hiding.

8.3 Hash Chains

Hash chains use hash functions recursively. If H is a hash function, then a hash chain of x is

$$x, H(x), H(H(x)), H(H(H(x))), H(H(H(H(x)))), \cdots \tag{8.9}$$

This can be more compactly written as

$$H^0(x), H^1(x), H^2(x), H^3(x), H^4(x), \cdots \tag{8.10}$$

where we have

$$H^n(x) = \begin{cases} x & n = 0 \\ H(H^{n-1}(x)) & n \geq 1 \end{cases}. \tag{8.11}$$

Depending on the circumstance, the initial value (x or $H^0(x)$) may not be included.

Hash chains enable easy iteration in the forward direction. By using a secure hash function, it is impractical to iterate in reverse, as doing so would require computing hash function preimages.

Example 8.3 (Hash Chain)
Code may be found at `examples/hash_applications/hash_chain.py`.

We look at a hash chain using MD5; see Listing 8.2. The initial byte array is

$$\begin{aligned} x &= H^0(x) \\ &= \texttt{00112233445566778899aabbccddeeff}. \end{aligned} \tag{8.12}$$

Iterating MD5 five times gives us the value

$$H^5(x) = \texttt{e94712ae81bf26d81b0e7475fbf6b3f8}. \tag{8.13}$$

Listing 8.2: Hash chain using MD5

```
#!/usr/bin/env python3

import hashlib

xStr = '00112233445566778899aabbccddeeff'
x0 = bytes.fromhex(xStr)
# 00112233445566778899aabbccddeeff

md5 = hashlib.md5()
md5.update(x0)
x1 = md5.digest()
# 6e8311168ee16d6aa1aa48c64145003c

md5 = hashlib.md5()
md5.update(x1)
x2 = md5.digest()
# 7a69fffa917aafaa21e54379fa990232

md5 = hashlib.md5()
md5.update(x2)
x3 = md5.digest()
# 30367bcf47ed16e934dc13e4c270e869

md5 = hashlib.md5()
md5.update(x3)
x4 = md5.digest()
# 5aef06528360f7185f08110aacb45f5d

md5 = hashlib.md5()
md5.update(x4)
x5 = md5.digest()
# e94712ae81bf26d81b0e7475fbf6b3f8
```

8.4 Merkle Trees I

Another useful application of hash functions are *Merkle trees*. In this section, we will focus on Merkle trees involving leaves which are a power-of-two. We will look at trees with an arbitrary number of leaves in Chap. 8.5. In Chap. 8.6, we will look at *Sparse Merkle trees*.

8.4.1 Intuition

It is challenging to send large amounts of data at one time. It would be convenient to chop the data into small pieces and then send those small pieces of data separately. The challenge is then to ensure the validity of each small piece of data.

A *Merkle tree* can help solve this problem. It uses one cryptographic hash value to represent the *entire* collection of data. At the same time, individual blocks of data can be retrieved along with its associated *Merkle Proof*: a proof which shows the individual data block is valid.

8.4.2 Definition

The algorithm defining the Merkle tree root hash can be found in Alg. 8.1; see Alg. 8.2 for the proof of inclusion. These definitions are modified from [30, Chapter 8.9]. This is valid only for trees with leaves which are a power-of-two; as mentioned, we will discuss the situation involving an arbitrary number of leaves in Chap. 8.5.

8.4.3 Continued Discussion

As mentioned above, there is an associated proof of inclusion for each piece of data. These individual proofs are inexpensive. It is possible to perform multiple inclusion-proofs at once; this is a called a *Merkle Multi-Proof*. We will not give a formal definition of a multi-proof but see Example 8.8 for an example.

We note the following details which are useful but which will not be discussed any further here. In addition to proofs of inclusion, it is also possible to have *proofs of exclusion*; doing this requires additional assumptions on the input data [42]. Furthermore, some implementations may use prefixes when hashing; the purpose of this is to provide second-preimage resistance [80, 81]. The need for this will be shown in Example 8.9.

8.4.4 Examples

Example 8.4 (2-Layer Merkle Tree)
Code may be found at `examples/hash_applications/merkle_tree_md5_2.py`.

We begin by starting with an example involving 4 blocks of data; see the Merkle tree in Figure 8.1a for the example.

In this case, we see that we have the following:

Algorithm 8.1 Compute Merkle Tree root hash

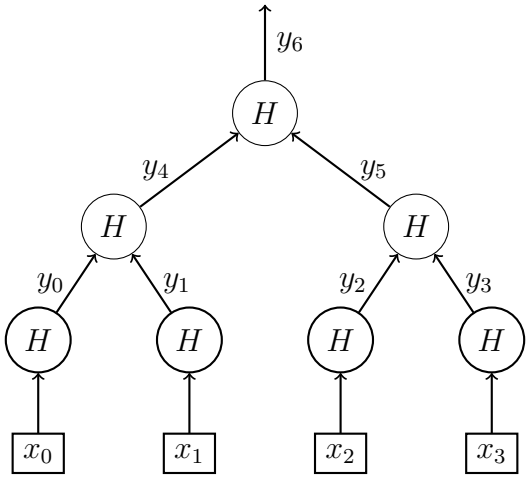
Require: $n = 2^k$ for some $k \geq 1$; H is a hash function.

procedure MerkleTreeRoot($x_0, x_1, \cdots, x_{n-1}$)
  for $i = 0; i < n; i{+}{+}$ **do**
    $y_i \coloneqq H(x_i)$ ▷ Compute leaf hashes
  end for
  for $i = 0; i < n - 1; i{+}{+}$ **do**
    $y_{n+i} \coloneqq H(y_{2i} || y_{2i+1})$ ▷ Compute internal hashes
  end for
  return y_{2n-2} ▷ Return root hash
end procedure

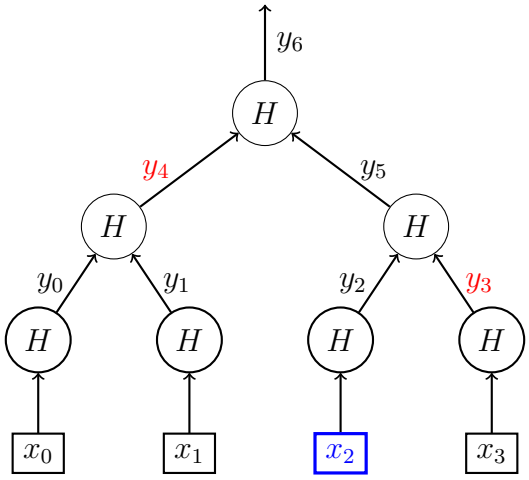
Algorithm 8.2 Validate Merkle Proof of Inclusion

Require: $n = 2^k$ for some $k \geq 1$; H is a hash function; $\text{len}(\pi) = \log_2(n)$.

procedure ValidateMerkleProof(x, i, y_{2n-2}, π)
  $\hat{y}_i \coloneqq H(x)$ ▷ Compute leaf hash
  $\mu \coloneqq i$
  for $k = 0; k < \log_2(n); k{+}{+}$ **do**
    if μ odd **then**
      $\ell \coloneqq (\mu - 1)/2$
      $\nu \coloneqq n + \ell$
      $\hat{y}_\nu \coloneqq H(\pi_k \| \hat{y}_\mu)$
    else
      $\ell \coloneqq \mu/2$
      $\nu \coloneqq n + \ell$
      $\hat{y}_\nu \coloneqq H(\hat{y}_\mu \| \pi_k)$
    end if
    $\mu \coloneqq \nu$
  end for
  if $\hat{y}_{2n-2} = y_{2n-2}$ **then**
    return `true`
  else
    return `false`
  end if
end procedure



(a) 2-layer Merkle Tree



(b) 2-layer Merkle Tree with Merkle Proof of x_2.

Figure 8.1: Here we have a 2-layer Merkle tree. There is also a Merkle Proof for x_2.

$$\begin{aligned} y_0 &= H(x_0) & y_4 &= H(y_0 \| y_1) \\ y_1 &= H(x_1) & y_5 &= H(y_2 \| y_3) \\ y_2 &= H(x_2) & y_6 &= H(y_4 \| y_5). \\ y_3 &= H(x_3) \end{aligned} \tag{8.14}$$

In this case, y_6 is our root hash.

To make this more practical, we will work through an example using MD5; see Listing 8.3. We have the data

$$\begin{aligned} x_0 &= \texttt{00000000000000000000000000000000} \\ x_1 &= \texttt{11111111111111111111111111111111} \\ x_2 &= \texttt{22222222222222222222222222222222} \\ x_3 &= \texttt{33333333333333333333333333333333}. \end{aligned} \tag{8.15}$$

After working through all the hashing in Eq. (8.14), we have the root hash

$$y_6 = \texttt{40c0b71ca488d334d266beecac02a16c}. \tag{8.16}$$

Example 8.5 (2-Layer Merkle Tree Proof)
Code may be found at `examples/hash_applications/merkle_tree_md5_2_proof.py`.

We will now look at Merkle Proof showing that x_2 is a member of this tree at position 2; see Figure 8.1b and Listing 8.4. Our data and proof is

$$\begin{aligned} x_2 &= \texttt{22222222222222222222222222222222} \\ y_3 &= \texttt{28bfcf057ec5a48f18c3154c1f2bd324} \\ y_4 &= \texttt{b05df6fba6c1c53e8ddb98ffd386ffc8}. \end{aligned} \tag{8.17}$$

Using these values, we see

$$\begin{aligned} \hat{y}_2 &= H(x_2) \\ &= \texttt{fbc3cf71d993ca7bec2664357ccdac2b} \\ \hat{y}_5 &= H(\hat{y}_2, y_3) \\ &= \texttt{8f7a2a2dcd297872689e177953270f37} \\ \hat{y}_6 &= H(y_4, \hat{y}_5) \\ &= \texttt{40c0b71ca488d334d266beecac02a16c}. \end{aligned} \tag{8.18}$$

Because $\hat{y}_6 = y_6$, we have shown that x_2 is the data at position 2 in our Merkle tree. This completes the proof; it required computing 3 hashes.

Listing 8.3: 2-Layer Merkle Tree using MD5

```
#!/usr/bin/env python3

import hashlib

# Set up data
x0 = bytes.fromhex(''.join('00' for i in range(16)))
x1 = bytes.fromhex(''.join('11' for i in range(16)))
x2 = bytes.fromhex(''.join('22' for i in range(16)))
x3 = bytes.fromhex(''.join('33' for i in range(16)))

# Make Merkle Tree
md5 = hashlib.md5(); md5.update(x0); y0 = md5.digest()
# 4ae71336e44bf9bf79d2752e234818a5
md5 = hashlib.md5(); md5.update(x1); y1 = md5.digest()
# 8057b6feaa62d90126274cf9ba31c642
md5 = hashlib.md5(); md5.update(x2); y2 = md5.digest()
# fbc3cf71d993ca7bec2664357ccdac2b
md5 = hashlib.md5(); md5.update(x3); y3 = md5.digest()
# 28bfcf057ec5a48f18c3154c1f2bd324

md5 = hashlib.md5(); md5.update(y0); md5.update(y1)
y4 = md5.digest()
# b05df6fba6c1c53e8ddb98ffd386ffc8
md5 = hashlib.md5(); md5.update(y2); md5.update(y3)
y5 = md5.digest()
# 8f7a2a2dcd297872689e177953270f37

md5 = hashlib.md5(); md5.update(y4); md5.update(y5)
y6 = md5.digest()
# 40c0b71ca488d334d266beecac02a16c
```

Listing 8.4: 2-Layer Merkle Proof using MD5

```
#!/usr/bin/env python3

import hashlib

root = bytes.fromhex('40c0b71ca488d334d266beecac02a16c')

# Set up data and proof
x2 = bytes.fromhex('22222222222222222222222222222222')
y3 = bytes.fromhex('28bfcf057ec5a48f18c3154c1f2bd324')
y4 = bytes.fromhex('b05df6fba6c1c53e8ddb98ffd386ffc8')


# Merkle Proof
md5 = hashlib.md5(); md5.update(x2);     yHat2 = md5.digest()
# fbc3cf71d993ca7bec2664357ccdac2b
md5 = hashlib.md5(); md5.update(yHat2); md5.update(y3)
yHat5 = md5.digest()
# 8f7a2a2dcd297872689e177953270f37
md5 = hashlib.md5(); md5.update(y4);     md5.update(yHat5)
yHat6 = md5.digest()
# 40c0b71ca488d334d266beecac02a16c

assert yHat6 == root # Valid
```

Example 8.6 (3-Layer Merkle Tree)
Code may be found at `examples/hash_applications/merkle_tree_md5_3.py`.

A 3-layer Merkle tree can be found in Figure 8.2a; also see Listing 8.5. In this case, we see that we have the following:

$$\begin{aligned} y_0 &= H(x_0) & y_8 &= H(y_0\|y_1) \\ y_1 &= H(x_1) & y_9 &= H(y_2\|y_3) \\ y_2 &= H(x_2) & y_{10} &= H(y_4\|y_5) \\ y_3 &= H(x_3) & y_{11} &= H(y_6\|y_7) \\ y_4 &= H(x_4) & y_{12} &= H(y_8\|y_9) \\ y_5 &= H(x_5) & y_{13} &= H(y_{10}\|y_{11}) \\ y_6 &= H(x_6) & y_{14} &= H(y_{12}\|y_{13}). \\ y_7 &= H(x_7) \end{aligned} \tag{8.19}$$

Here, y_{14} is the root hash.

For our data, we set

$$\begin{aligned} x_0 &= \texttt{00000000000000000000000000000000} \\ x_1 &= \texttt{01010101010101010101010101010101} \\ x_2 &= \texttt{02020202020202020202020202020202} \\ x_3 &= \texttt{03030303030303030303030303030303} \\ x_4 &= \texttt{04040404040404040404040404040404} \\ x_5 &= \texttt{05050505050505050505050505050505} \\ x_6 &= \texttt{06060606060606060606060606060606} \\ x_7 &= \texttt{07070707070707070707070707070707}. \end{aligned} \tag{8.20}$$

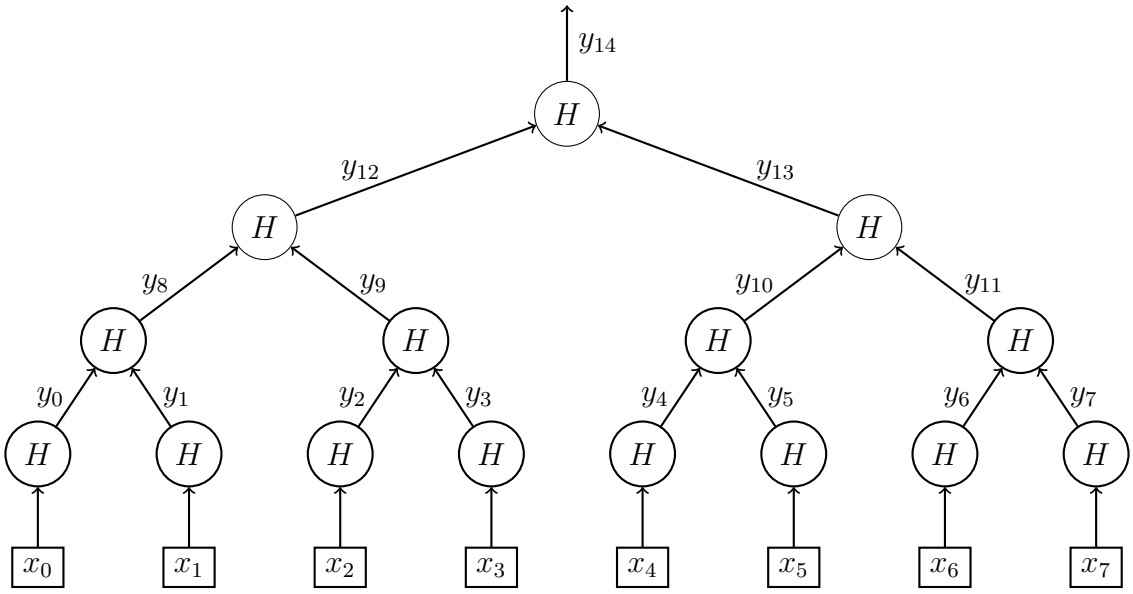
We have the root hash

$$y_{14} = \texttt{18446692e822581d02b096e1b77c9fff}. \tag{8.21}$$

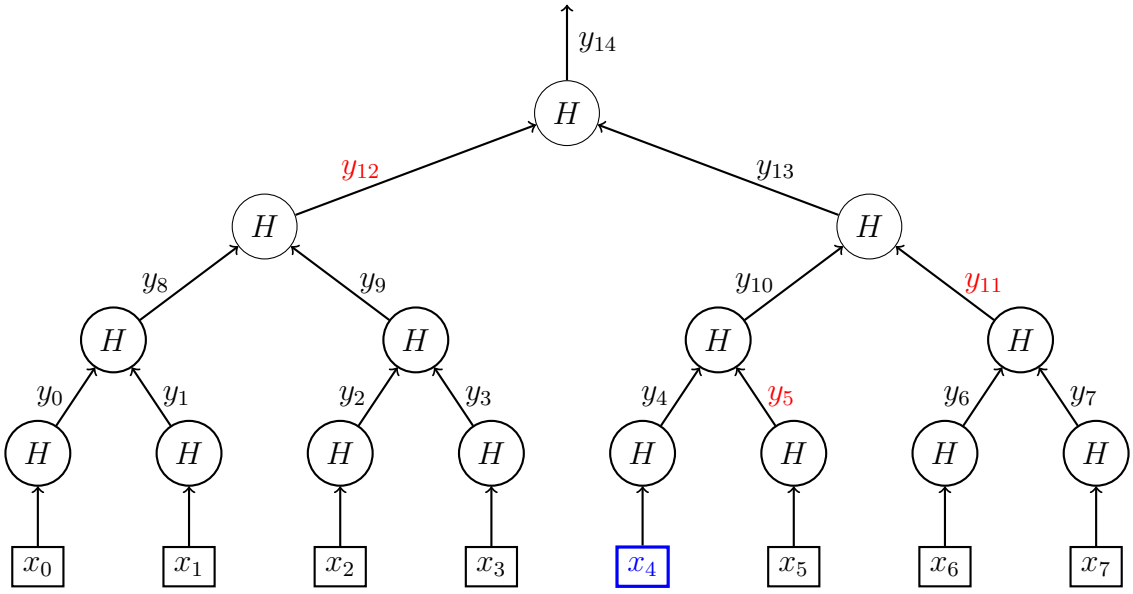
Example 8.7 (3-Layer Merkle Tree Proof)
Code may be found at `examples/hash_applications/merkle_tree_md5_3_proof.py`.

We continue our previous example to show a Merkle proof showing x_4 is an element of our Merkle tree; see Figure 8.2b and Listing 8.6. Our data and proof is

$$\begin{aligned} x_4 &= \texttt{04040404040404040404040404040404} \\ y_5 &= \texttt{af52282db55243f4c147ba5d7fb1155a} \\ y_{11} &= \texttt{70d0669eae8c7a5dce3b3ff4ccf4adbb} \\ y_{12} &= \texttt{a3f21dba8fa8de359220c29c00467556}. \end{aligned} \tag{8.22}$$



(a) 3-layer Merkle Tree



(b) 3-layer Merkle Tree with Merkle Proof of x_4

Figure 8.2: Here is a 3-layer Merkle tree. There is also a Merkle Proof for x_4.

Listing 8.5: 3-Layer Merkle Tree using MD5

```
#!/usr/bin/env python3

import hashlib

# Set up data
x0 = bytes.fromhex(''.join('00' for i in range(16)))
x1 = bytes.fromhex(''.join('01' for i in range(16)))
x2 = bytes.fromhex(''.join('02' for i in range(16)))
x3 = bytes.fromhex(''.join('03' for i in range(16)))
x4 = bytes.fromhex(''.join('04' for i in range(16)))
x5 = bytes.fromhex(''.join('05' for i in range(16)))
x6 = bytes.fromhex(''.join('06' for i in range(16)))
x7 = bytes.fromhex(''.join('07' for i in range(16)))

# Make Merkle Tree
md5 = hashlib.md5(); md5.update(x0);  y0 = md5.digest()
md5 = hashlib.md5(); md5.update(x1);  y1 = md5.digest()
md5 = hashlib.md5(); md5.update(x2);  y2 = md5.digest()
md5 = hashlib.md5(); md5.update(x3);  y3 = md5.digest()
md5 = hashlib.md5(); md5.update(x4);  y4 = md5.digest()
md5 = hashlib.md5(); md5.update(x5);  y5 = md5.digest()
md5 = hashlib.md5(); md5.update(x6);  y6 = md5.digest()
md5 = hashlib.md5(); md5.update(x7);  y7 = md5.digest()

md5 = hashlib.md5(); md5.update(y0);  md5.update(y1)
y8  = md5.digest()
md5 = hashlib.md5(); md5.update(y2);  md5.update(y3)
y9  = md5.digest()
md5 = hashlib.md5(); md5.update(y4);  md5.update(y5)
y10 = md5.digest()
md5 = hashlib.md5(); md5.update(y6);  md5.update(y7)
y11 = md5.digest()

md5 = hashlib.md5(); md5.update(y8);  md5.update(y9)
y12 = md5.digest()
md5 = hashlib.md5(); md5.update(y10); md5.update(y11)
y13 = md5.digest()

md5 = hashlib.md5(); md5.update(y12); md5.update(y13)
y14 = md5.digest()
# 18446692e822581d02b096e1b77c9fff
```

Listing 8.6: 3-Layer Merkle Proof using MD5

```
#!/usr/bin/env python3

import hashlib

root = bytes.fromhex('18446692e822581d02b096e1b77c9fff')

# Set up data and proof
x4  = bytes.fromhex('040404040404040404040404040404')
y5  = bytes.fromhex('af52282db55243f4c147ba5d7fb1155a')
y11 = bytes.fromhex('70d0669eae8c7a5dce3b3ff4ccf4adbb')
y12 = bytes.fromhex('a3f21dba8fa8de359220c29c00467556')

# Merkle Proof
md5 = hashlib.md5(); md5.update(x4);      yHat4 = md5.digest()
# b40ebeba833c1c07e74d9e4c6ebb8230
md5 = hashlib.md5(); md5.update(yHat4);   md5.update(y5)
yHat10 = md5.digest()
# 524856d86cb1006e9e9e9149f36b2875
md5 = hashlib.md5(); md5.update(yHat10); md5.update(y11)
yHat13 = md5.digest()
# da95d6882598892592ccd54bc89f7bdb
md5 = hashlib.md5(); md5.update(y12);     md5.update(yHat13)
yHat14 = md5.digest()
# 18446692e822581d02b096e1b77c9fff

assert yHat14 == root # Valid
```

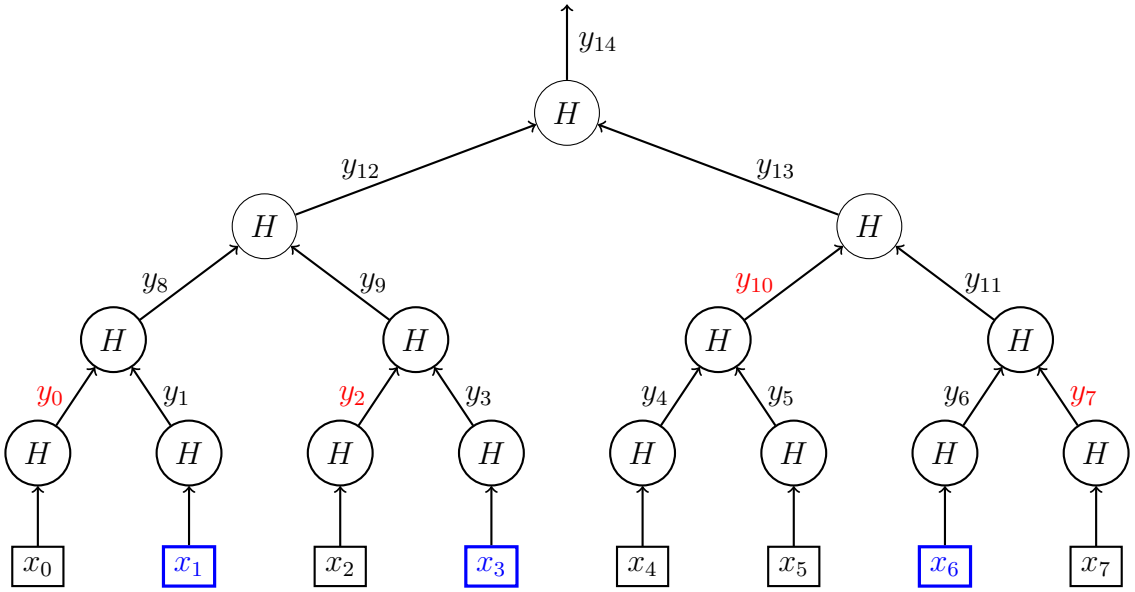


Figure 8.3: Here is a 3-layer Merkle tree with a Merkle Multi-Proof of x_1, x_3, and x_6.

Using these values, we see

$$\begin{aligned}
\hat{y}_4 &= H(x_4) \\
&= \texttt{b40ebeba833c1c07e74d9e4c6ebb8230} \\
\hat{y}_{10} &= H(\hat{y}_4, y_5) \\
&= \texttt{524856d86cb1006e9e9e9149f36b2875} \\
\hat{y}_{13} &= H(\hat{y}_{10}, y_{11}) \\
&= \texttt{da95d6882598892592ccd54bc89f7bdb} \\
\hat{y}_{14} &= H(y_{12}, \hat{y}_{13}) \\
&= \texttt{18446692e822581d02b096e1b77c9fff}. \qquad (8.23)
\end{aligned}$$

Because $\hat{y}_{14} = y_{14}$ in Eq. (8.21), we have shown that x_4 is the data at position 4 in our Merkle tree. This completes the proof; it required computing 4 hashes.

Example 8.8 (3-Layer Merkle Tree Multi-Proof)
Code may be found at `examples/hash_applications/merkle_tree_md5_3_multi-proof.py`.

We continue the previous example working with a 3-layer Merkle tree by showing an example of a Merkle Multi-Proof. In this case, we want to proof the inclusion of x_1, x_3, and x_6; see Figure 8.3 and Listing 8.7.

We have the data

$$\begin{aligned}
x_1 &= \texttt{01010101010101010101010101010101} \\
x_3 &= \texttt{03030303030303030303030303030303} \\
x_6 &= \texttt{06060606060606060606060606060606}. \qquad (8.24)
\end{aligned}$$

Listing 8.7: 3-Layer Merkle Multi-Proof using MD5

```
#!/usr/bin/env python3

import hashlib

root = bytes.fromhex('18446692e822581d02b096e1b77c9fff')

# Set up data and proof
x1  = bytes.fromhex('01010101010101010101010101010101')
x3  = bytes.fromhex('03030303030303030303030303030303')
x6  = bytes.fromhex('06060606060606060606060606060606')
y0  = bytes.fromhex('4ae71336e44bf9bf79d2752e234818a5')
y2  = bytes.fromhex('437b25ad27df2f61eb14c6400ae98309')
y7  = bytes.fromhex('a4aa9a02aa65f31d17dce9f944a57ab2')
y10 = bytes.fromhex('524856d86cb1006e9e9e9149f36b2875')

# Merkle Proof
md5 = hashlib.md5(); md5.update(x1);      yHat1 = md5.digest()
md5 = hashlib.md5(); md5.update(x3);      yHat3 = md5.digest()
md5 = hashlib.md5(); md5.update(x6);      yHat6 = md5.digest()

md5 = hashlib.md5(); md5.update(y0);      md5.update(yHat1)
yHat8  = md5.digest()
md5 = hashlib.md5(); md5.update(y2);      md5.update(yHat3)
yHat9  = md5.digest()
md5 = hashlib.md5(); md5.update(yHat6);   md5.update(y7)
yHat11 = md5.digest()

md5 = hashlib.md5(); md5.update(yHat8);   md5.update(yHat9)
yHat12 = md5.digest()
md5 = hashlib.md5(); md5.update(y10);     md5.update(yHat11)
yHat13 = md5.digest()

md5 = hashlib.md5(); md5.update(yHat12); md5.update(yHat13)
yHat14 = md5.digest()

assert yHat14 == root # Valid
```

Our proof is

$$\begin{aligned} y_0 &= \texttt{4ae71336e44bf9bf79d2752e234818a5} \\ y_2 &= \texttt{437b25ad27df2f61eb14c6400ae98309} \\ y_7 &= \texttt{a4aa9a02aa65f31d17dce9f944a57ab2} \\ y_{10} &= \texttt{524856d86cb1006e9e9e9149f36b2875}. \end{aligned} \tag{8.25}$$

We have the following computed values:

$$\begin{aligned} \hat{y}_1 &= H(x_1) \\ &= \texttt{24311d9abc4077123c2c9a167afbe754} \\ \hat{y}_3 &= H(x_3) \\ &= \texttt{b09931959951caa2ac110feebbd16750} \\ \hat{y}_6 &= H(x_6) \\ &= \texttt{e5f636381c9702b63aa666ef2a6f8e20} \\ \hat{y}_8 &= H(y_0 \| \hat{y}_1) \\ &= \texttt{f8a9500bb41d7312abd46b0d01404fca} \\ \hat{y}_9 &= H(y_2 \| \hat{y}_3) \\ &= \texttt{8bfbb2226583b489fbc3cdfee7adc2c7} \\ \hat{y}_{11} &= H(\hat{y}_6, y_7) \\ &= \texttt{70d0669eae8c7a5dce3b3ff4ccf4adbb} \\ \hat{y}_{12} &= H(\hat{y}_8 \| \hat{y}_9) \\ &= \texttt{a3f21dba8fa8de359220c29c00467556} \\ \hat{y}_{13} &= H(y_{10} \| \hat{y}_{11}) \\ &= \texttt{da95d6882598892592ccd54bc89f7bdb} \\ \hat{y}_{14} &= H(\hat{y}_{12} \| \hat{y}_{13}) \\ &= \texttt{18446692e822581d02b096e1b77c9fff}. \end{aligned} \tag{8.26}$$

Because $\hat{y}_{14} = y_{14}$, we have shown that x_1, x_3, and x_6 are valid and are at the specified positions. This completes the proof. This proof requires computing 9 hash values, which is less than the 12 hash computations required to prove the inclusions separately ($12 = 3 \cdot 4$; 3 proofs involving 4 hash computations each).

Example 8.9 (3-Layer Merkle Tree Security)
Code may be found at `examples/hash_applications/merkle_tree_md5_3_security.py`.

We continue the previous example working with a 3-layer Merkle tree by showing what can go wrong by using the standard definitions given in Algs. 8.1 and 8.2.

In particular, we will show that the value

$$x \coloneqq y_4 \| y_5 \tag{8.27}$$

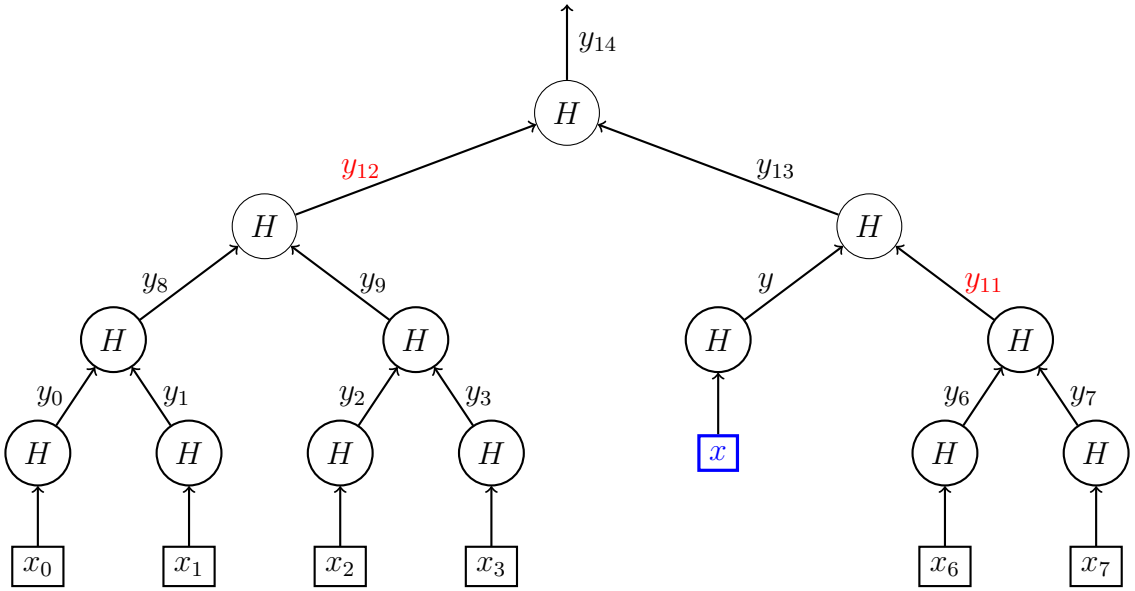


Figure 8.4: Here is a 3-layer Merkle tree with a Merkle Proof for showing that $x := y_4||y_5$ is a "leaf node" in the tree. This is possible if there is not difference between hashing leaf nodes and interior nodes in a Merkle tree.

is a "leaf node" in the tree; see Figure 8.4 and Listing 8.8. The values y_4 and y_5 are derived from the data in Eq. (8.20).

Our data and proof is

$$\begin{aligned} x &= \texttt{b40ebeba833c1c07e74d9e4c6ebb8230af52282db55243f4c147ba5d7fb1155a} \\ y_{11} &= \texttt{70d0669eae8c7a5dce3b3ff4ccf4adbb} \\ y_{12} &= \texttt{a3f21dba8fa8de359220c29c00467556}. \end{aligned} \tag{8.28}$$

Using these values, we see

$$\begin{aligned} \hat{y} &= H(x) \\ &= \texttt{524856d86cb1006e9e9e9149f36b2875} \\ \hat{y}_{13} &= H(\hat{y}, y_{11}) \\ &= \texttt{da95d6882598892592ccd54bc89f7bdb} \\ \hat{y}_{14} &= H(y_{12}, \hat{y}_{13}) \\ &= \texttt{18446692e822581d02b096e1b77c9fff}. \end{aligned} \tag{8.29}$$

Because $\hat{y}_{14} = y_{14}$ in Eq. (8.21), we have "proven" that x is a leaf in our Merkle tree. The reason for this will be discussed more in Chapter 8.5.1.

Listing 8.8: Security of 3-Layer Merkle Proof using MD5

```
#!/usr/bin/env python3

import hashlib

root = bytes.fromhex('18446692e822581d02b096e1b77c9fff')

# Set up data and proof
x   = bytes.fromhex('\
b40ebeba833c1c07e74d9e4c6ebb8230af52282db55243f4c147ba5d7fb1155a')
y11 = bytes.fromhex('70d0669eae8c7a5dce3b3ff4ccf4adbb')
y12 = bytes.fromhex('a3f21dba8fa8de359220c29c00467556')

# Merkle Proof
md5 = hashlib.md5(); md5.update(x);   y = md5.digest()
# 524856d86cb1006e9e9e9149f36b2875
md5 = hashlib.md5(); md5.update(y);   md5.update(y11)
yHat13 = md5.digest()
# da95d6882598892592ccd54bc89f7bdb
md5 = hashlib.md5(); md5.update(y12); md5.update(yHat13)
yHat14 = md5.digest()
# 18446692e822581d02b096e1b77c9fff

assert yHat14 == root # Valid
```

8.5 Merkle Trees II

The previous section focused on generic Merkle trees when the number of leaves are a power-of-two. We now look at the more general situation involving an arbitrary number of leaves. First, we start by looking more closely at Merkle tree security concerns.

8.5.1 Merkle Tree Security

In Example 8.9, we were able to "prove" that another data value was present in the Merkle tree. The fact such easy "proofs" exist and may be readily found scares cryptographers (who have a "professional paranoia mindset" [51, Chapter 1.12]). This is unacceptable; it should be impractical to find any data and a valid inclusion proof if that data is not present within the Merkle tree.

This problem arises because the same hash algorithm is used for both leaf and interior nodes. One way to get around this is to use domain separation within the definition of the leaf hashes and interior hashes; this choice is used in [80, 81]. In particular, leaf hashes begin with the zero byte `0x00` while other hashes (interior hashes) start with the one byte `0x01`. The stated reason is that "this domain separation is required to give second preimage resistance." [81, Section 2.1.1].

8.5.2 Merkle Trees with Arbitrary Leaves

Our Merkle tree definitions in Algs. 8.1 and 8.2 only allowed for leaves which were a power-of-two. While it would be possible to always use this by padding with a specific value, we would prefer to not have this restriction.

To get around this, a convention must be followed. One choice may be found in [80, 81]; another convention is specified in the OpenZeppelin Merkle tree library [96].

Certificate Transparency Convention

We will first look at the convention in [80, 81], although we will use the general hash from [81] rather than SHA-2-256 from [80]. The precise definition for computing the Merkle tree root hash may be found in Alg. 8.3; the definition is defined using recursion. Some visualizations of these trees may be found in Figures 8.5, 8.6, 8.7, and 8.8. Algorithms for constructing inclusion proofs and verifying them may be found in [81]. As mentioned above, second preimage security is ensured by using domain separation.

OpenZeppelin Convention

A slightly different convention is specified in the OpenZeppelin Merkle tree library [96]; this convention has been used on the Ethereum blockchain within smart contracts. The definition for computing the Merkle tree root hash may be found in Alg. 8.4. Second preimage security is ensured by double-hashing the leaf nodes. Some visualizations of these trees may be found in Figures 8.9, 8.10, 8.11, and 8.12. Additional algorithms may be found in [96].

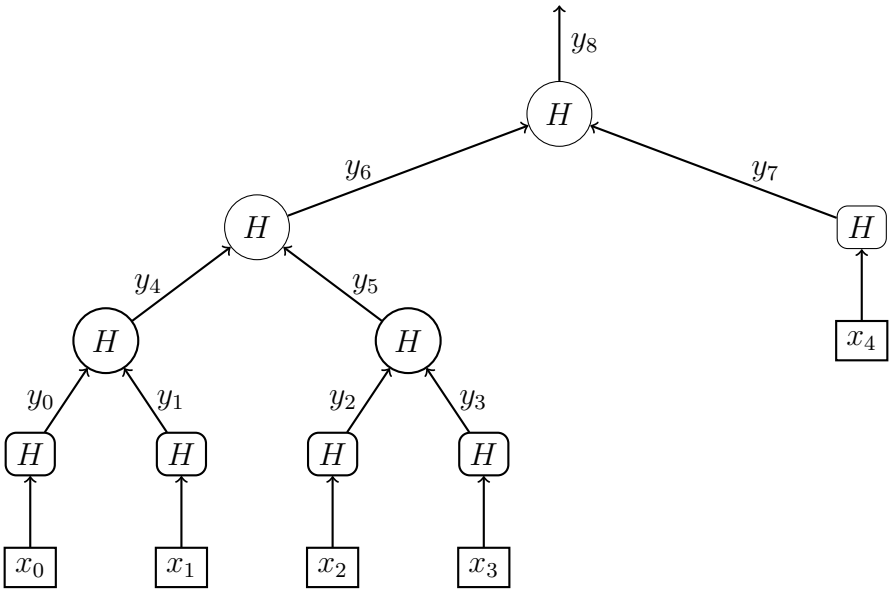


Figure 8.5: Merkle Tree with 5 leaves using recursive definition from [81].

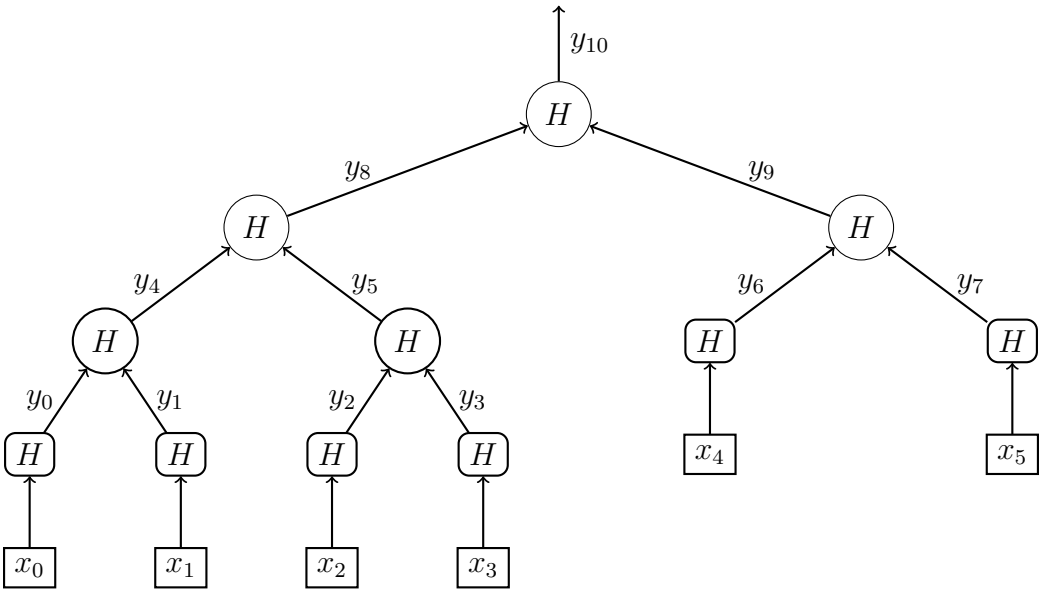


Figure 8.6: Merkle Tree with 6 leaves using recursive definition from [81].

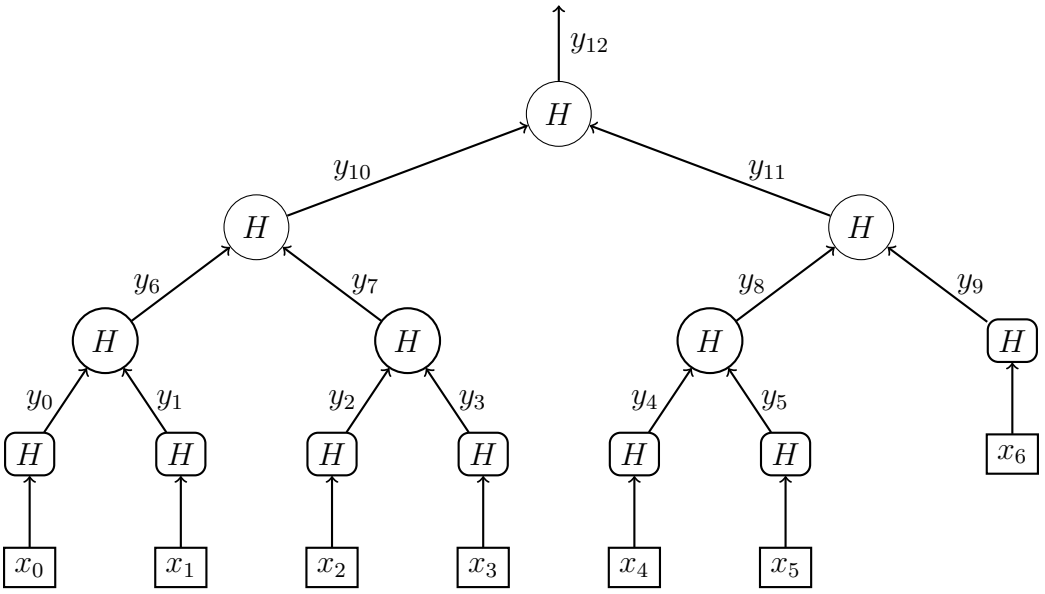


Figure 8.7: Merkle Tree with 7 leaves using recursive definition from [81].

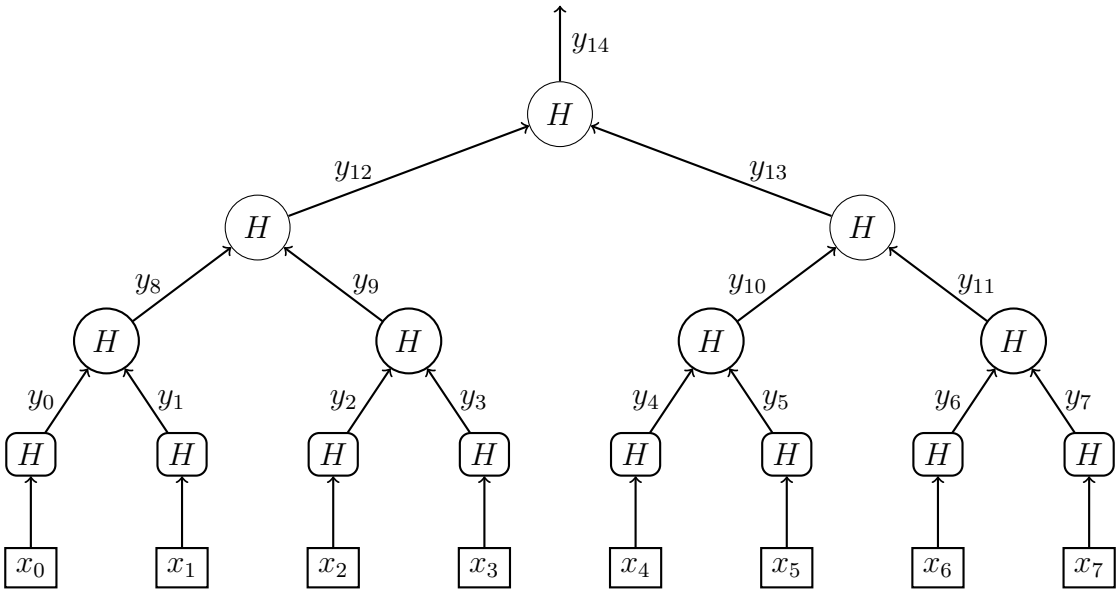


Figure 8.8: Merkle Tree with 8 leaves using recursive definition from [81].

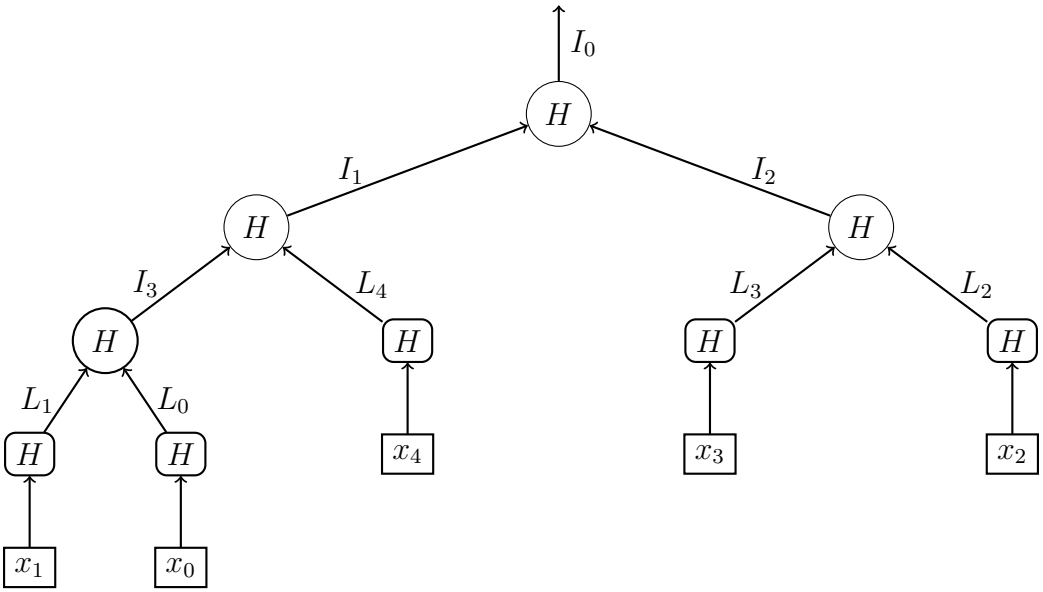


Figure 8.9: Merkle Tree with 5 leaves using definition from [96].

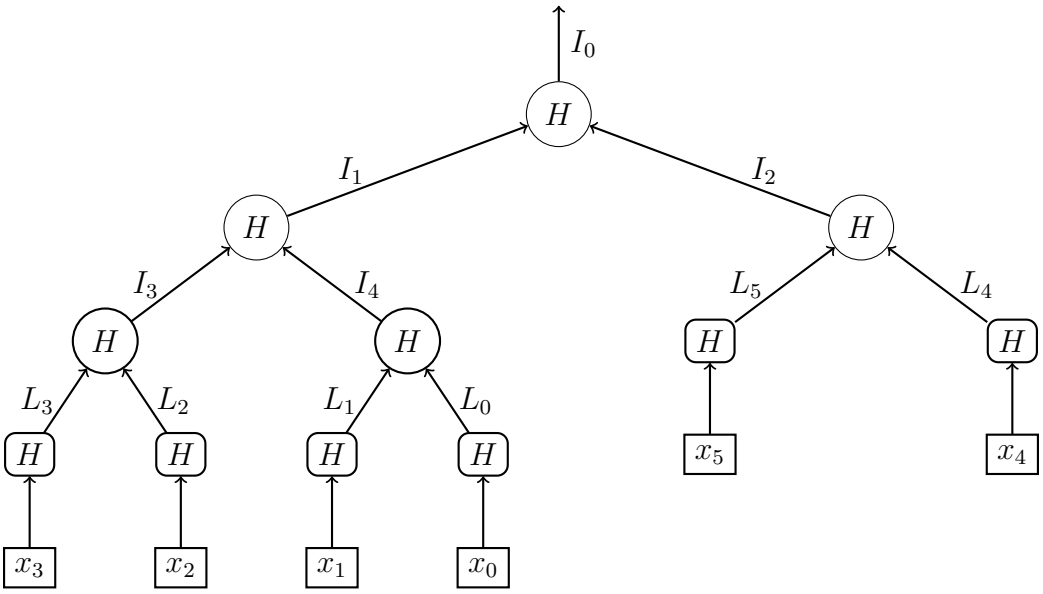


Figure 8.10: Merkle Tree with 6 leaves using definition from [96].

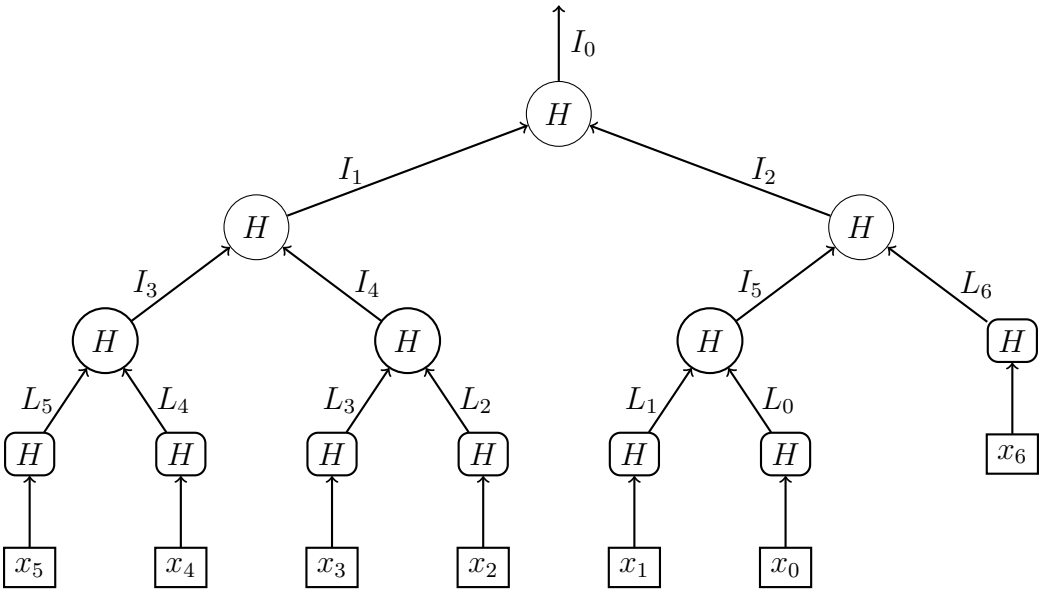


Figure 8.11: Merkle Tree with 7 leaves using definition from [96].

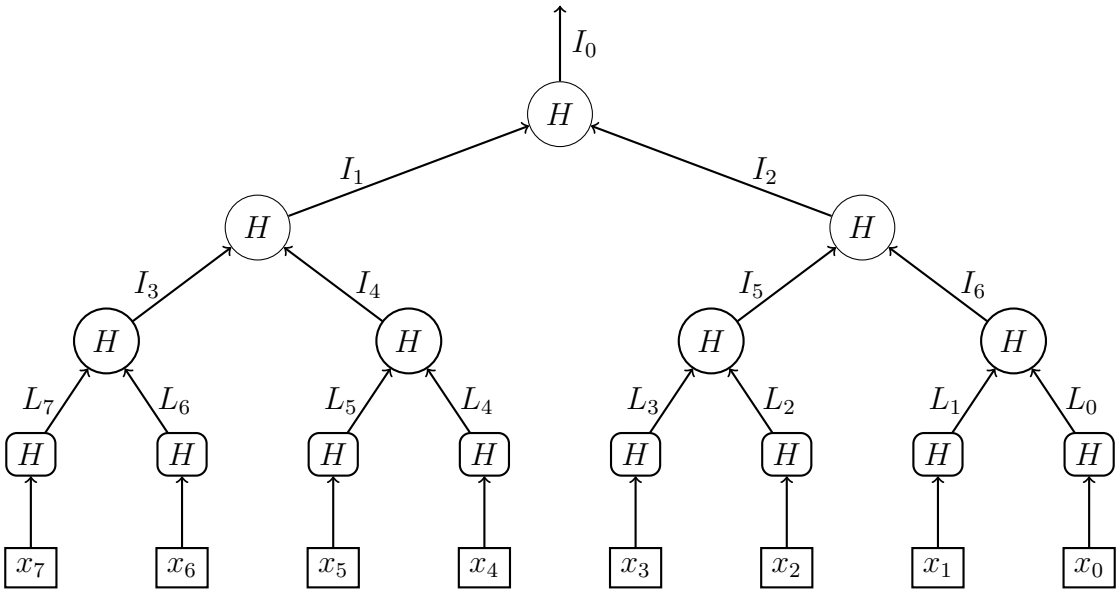


Figure 8.12: Merkle Tree with 8 leaves using definition from [96].

Algorithm 8.3 Compute Merkle Tree root hash using recursion from [81]

procedure MerkleRootCTV($x_0, x_1, \cdots, x_{n-1}$)
  if $n = 0$ **then**
    return Hash() ▷ Hash of empty byte string
  else if $n = 1$ **then**
    return Hash(0x00, x_0)
  else
    $k := \max\left\{2^\ell \mid \ell \in \mathbb{N}_0 \wedge 2^\ell < n\right\}$ ▷ k is largest power-of-two strictly less than n
    $h_0 :=$ MerkleRootCTV($x_0, x_1, \cdots, x_{k-1}$)
    $h_1 :=$ MerkleRootCTV($x_k, x_{k+1}, \cdots, x_{n-1}$)
    return Hash(0x01, h_0, h_1)
  end if
end procedure

8.6 Sparse Merkle Trees

8.6.1 Discussion

We continue our discussion of Merkle trees by noting another variation: *Sparse Merkle trees*. In this case, the number of leaves may be very large; however, *most* of the leaves are empty, so it is still practical to work with the tree.

In particular, using a 256-bit hash function such as SHA-2-256 or SHA-3-256, there are a maximum of $2^{256} \simeq 10^{77}$ leaves.

Additionally, while when we discussed Merkle trees in Chapters 8.4 and 8.5 we placed no restriction on the key order, in Sparse Merkle trees the keys will be sorted. Furthermore, if there is no data for a particular key, it is handled in a special way.

8.6.2 Conventions

There are various conventions for Sparse Merkle trees; this is needed in order to know how to handle empty leaves. Some references include [42, 59, 65, 79, 106].

8.7 Hash-based Message Authentication Code

As mentioned in Chapter 6.7, message authentication codes are used for message authentication; Bob can be certain Alice sent him the message and that it was not corrupted in transit. One method for implementing a message authentication code is based on a hash function, hence Hash-based Message Authentication Code (HMAC) [10, 77]. Any hash function may be used, but Hash-based Message Authentication Codes based on MD5 or SHA-1 should *only* be used if *required*; see [128].

Further discussion about Hash-based Message Authentication Code is relegated to Appendix A.2.1.

Algorithm 8.4 Compute Merkle Tree root hash from [96]

```
1:  procedure MerkleRootOZ(data)
2:      n ≔ Length(data)
3:      leaves ≔ Make(array, n)                ▷ Initialize an empty array of length n
4:      for k = 0; k < n; k++ do
5:          leaves[k] ≔ LeafHash(data[k])
6:      end for
7:      tree ≔ Make(array, 2n − 1)             ▷ Initialize an empty array of length 2n − 1
8:      for k = 0; k < n; k++ do
9:          tree[2n − 2 − k] ≔ leaves[k])
10:     end for
11:     for k = n − 2; k ≥ 0; k−− do
12:         tree[k] ≔ InteriorHash(tree[2k + 1], tree[2k + 2])
13:     end for
14:     return tree[0]                         ▷ Return the root hash
15: end procedure
16:
17: procedure LeafHash(x)
18:     return Hash(Hash(x))
19: end procedure
20:
21: procedure InteriorHash(a, b)
22:     if int(a) < int(b) then
23:         return Hash(a, b)
24:     else
25:         return Hash(b, a)
26:     end if
27: end procedure
```

8.8 HMAC-based Key Derivation Function

We mentioned in Chapter 6.5 that key derivation functions take initial key material which has sufficient entropy (but which may not be uniform) and converts it into a uniform cryptographically-strong key. One way this can be done is through the use of a hash function via the HMAC construction; see [76, 78]. In this case, it is called an HMAC-based Key Derivation Function (HKDF). The main idea is to first compress the available entropy; this compressed entropy is then expanded into cryptographic keys. For instance, it could be used to derive an encryption key from a shared secret between Alice and Bob.

Further discussion about HMAC-based Key Derivation Function is relegated to Appendix A.2.3.

8.9 Mask Generation Functions and Extendable Output Functions

A mask generation function is like a hash function except it has a variable digest length; in particular, the length may be made arbitrarily long. There are times when this is useful, especially if the digest of a hash function is too short. A similar concept is an extendable output function.

Further discussion about mask generation functions and extendable output functions is relegated to Appendix A.2.4.

8.10 Password Hashing

We briefly describe another application of cryptographic hash functions: safe password storage. As mentioned in Chapter 6.5, key derivation functions are also used for protecting passwords.

Password storage is difficult. One way to store passwords is to not store the password *itself* but to rather store its *hash*. In order to make each password hash unique, each password is also given a random *salt*. A salt is a bit string which is uniformly random used for domain separation; the proper length will be discussed below.

If a salt is not used (or a salt is repeated), then it is possible to make *rainbow tables*: find a list of the most common passwords and hash them, either by themselves or together with the repeated salt. This is why it is important to not reuse salts: reusing salts makes it more efficient to crack multiple passwords. Using a repeated salt is almost as bad as using no salt at all. This is also why it is important to use salts of sufficient length and entropy: it should be impractical to make a rainbow table of every possible salt value, and it should not be possible to *derive* the salt from related information.

While cryptographic hash functions like SHA-2 or SHA-3 may be used, when security is important, it is important to use a hash function *specifically designed* for password hashing. Some examples are `scrypt` [98, 99], `Argon2` [23], and balloon hashing [28]. When using these hash functions, it is possible to tune the difficulty parameter so that users are only mildly affected by password validation, but adversaries have a much more difficult time when attempting to brute-force potential passwords. The difficulty parameter ensures that password cracking is difficult even when performed offline. We note that even though PBKDF2 [90] is recommended as of 2017, it was designed in 2000 [66] and should not be used due to the insufficient protection it supplies [25]. In [25], the authors recommend *not* using `bcrypt` [104] and PBKDF2 [66]. The `bcrypt` key derivation function is derived from the Blowfish cipher [116]. Both PBKDF2 and `bcrypt` allow for varying the computation time, but the fact that they do not have large memory requirements enables the construction of dedicated hardware for inexpensive attacks.

There have been a number of instances where user passwords have been hacked; see the examples in [25] and Figure 8.13. Suggestions for strong passwords may be found in Figure 8.14. As we can see from Figure 8.15, password reuse is insecure.

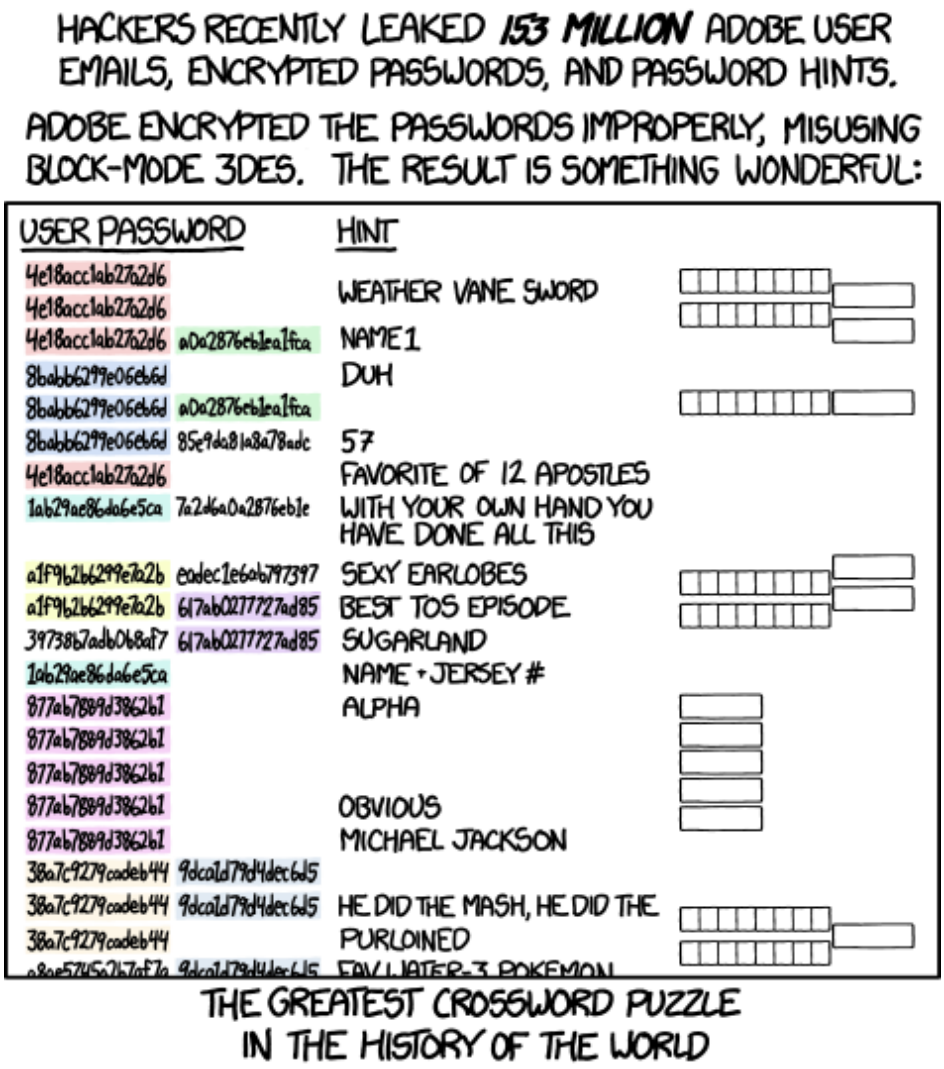


Figure 8.13: Here we have an example of how *not* to store passwords; also, see Figure 8.15. Created by Randall Munroe on `xkcd`; posted online at https://xkcd.com/1286/.

Salt Length

It is important that a salt be long enough so that it is impractical to form rainbow tables. Because of this, salts should be at least be 128 bits (16 bytes). Ideally, salts should be 256 bits (32 bytes). Salts may be longer, but the *purpose* of a salt is to ensure a unique output even if the same password is used multiple times. To make the password derivation more difficult, use a hash function designed for password hashing and choose the appropriate difficulty setting with iteration count and memory requirements.

8.11 Concluding Discussion of Hash Functions

There has been a lot of information presented here on hash functions. From all the applications discussed, it is obvious that hash functions are used all over cryptography. Also, see

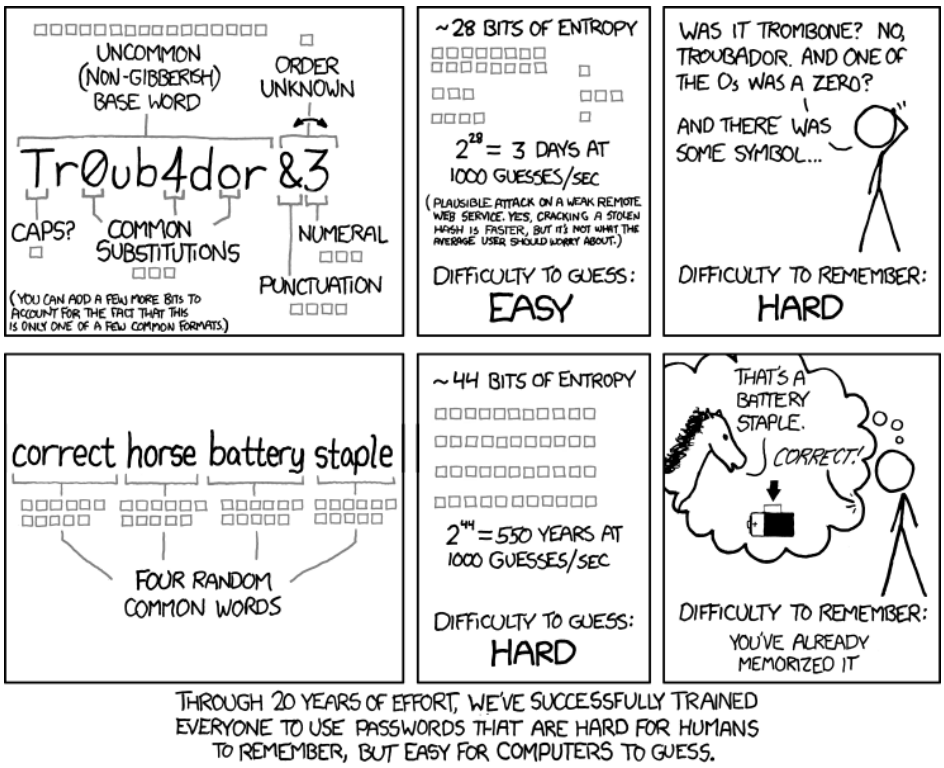


Figure 8.14: Here we learn about strong passwords. Created by Randall Munroe on `xkcd`; posted online at https://xkcd.com/936/.

the discussion of geohashing in Figure 8.16.

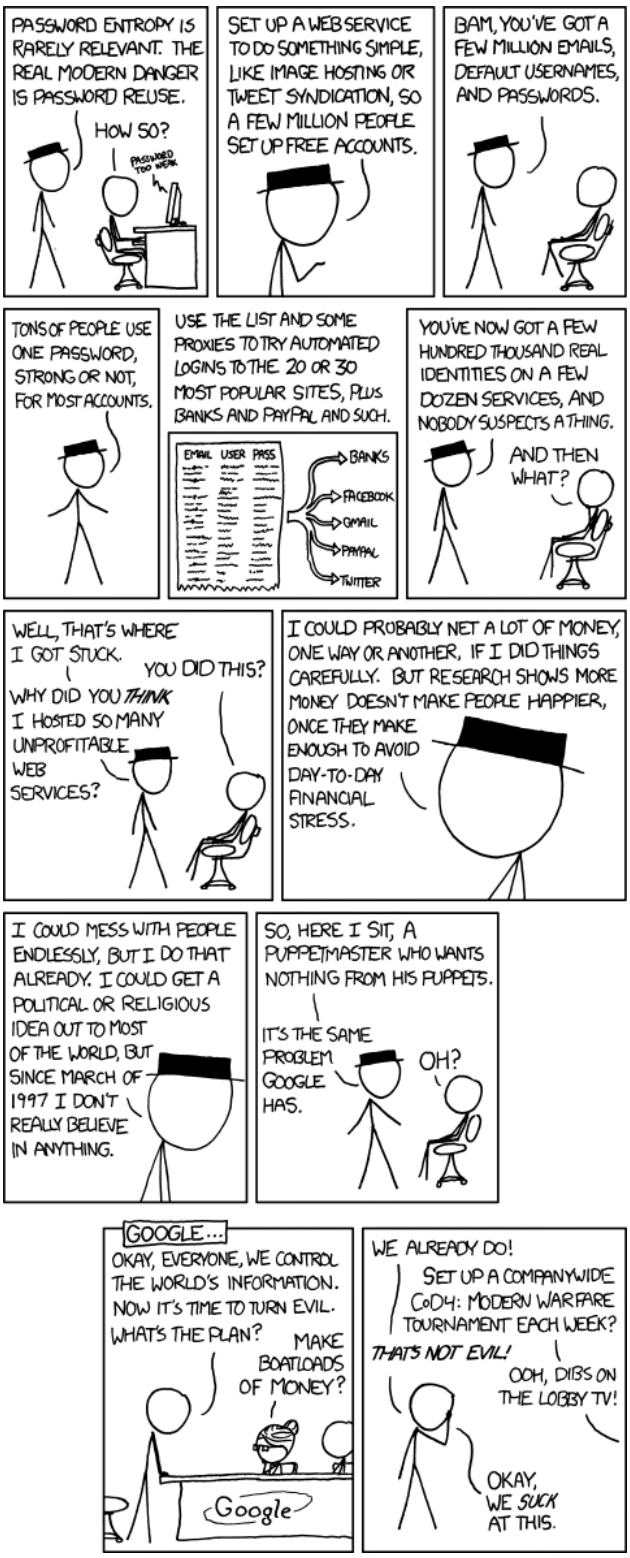


Figure 8.15: Here we learn why it is a bad idea to reuse passwords. Created by Randall Munroe on `xkcd`; posted online at https://xkcd.com/792/.

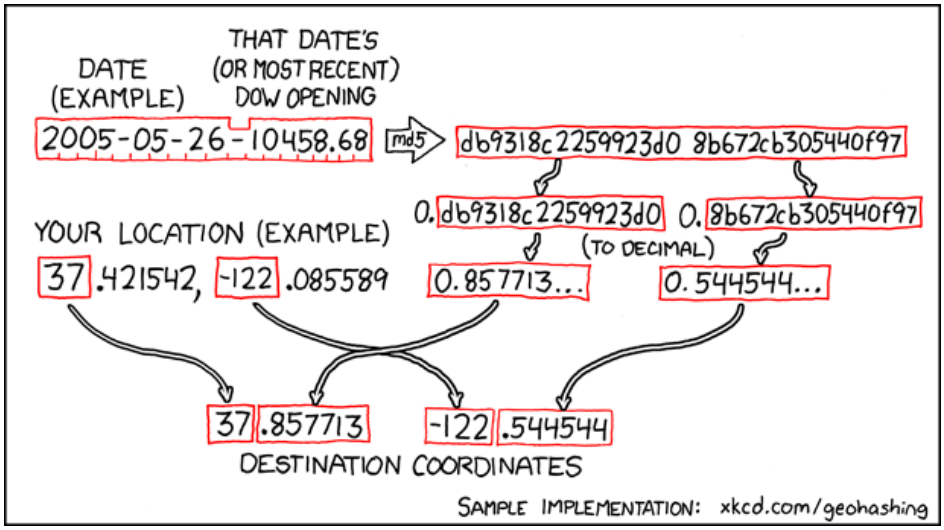


Figure 8.16: Geohashing is a frequently-overlooked application of cryptographic hash functions. Created by Randall Munroe on `xkcd`; posted online at https://xkcd.com/426/.

Chapter 9

Public Key Cryptography

We now talk about Public Key Cryptography.

9.1 The Need for Public Key Cryptography

Alice and Bob would like to communicate with each other. Unfortunately, there is a problem: they have never met and are not able to meet beforehand. Thus, they are not able to share a secret through a secure channel. Is there some way that Alice and Bob can still securely communicate over an insecure channel? The answer is yes, and communication is possible using *Public Key Cryptography*.

In each case, Alice and Bob have *two* keys: a *public key* (known to everyone) and a *private key* (known only to the individual). We recall that in Symmetric Key Cryptography, Alice and Bob *share* a secret key. Due to the nature of the private key, it should never be shared; see Figure 9.1.

The Diffie-Hellman Key Exchange was first described in by Diffie and Hellman in [44]. The first method for performing public key encryption and digital signatures was the original RSA paper [109] by Rivest, Shamir, and Adleman. We will not discuss Public Key Cryptography based on RSA; these notes focus on Elliptic Curve Cryptography.

9.2 Brief Discussion of Public Key Infrastructure

Public Key Cryptography *assumes* the ability for Alice and Bob to publicly post information and allow everyone to know that this information is valid. This is handled by Public Key Infrastructure (PKI). This is an important aspect of Public Key Cryptography and is also very challenging; we will not go into details at this time. For simplicity, we will assume that such infrastructure is present.

9.3 Diffie-Hellman Key Exchange

Here, we discuss how Alice and Bob, who have never met, can construct a shared secret. This shared secret may then be used derive a secret key for symmetric key encryption.

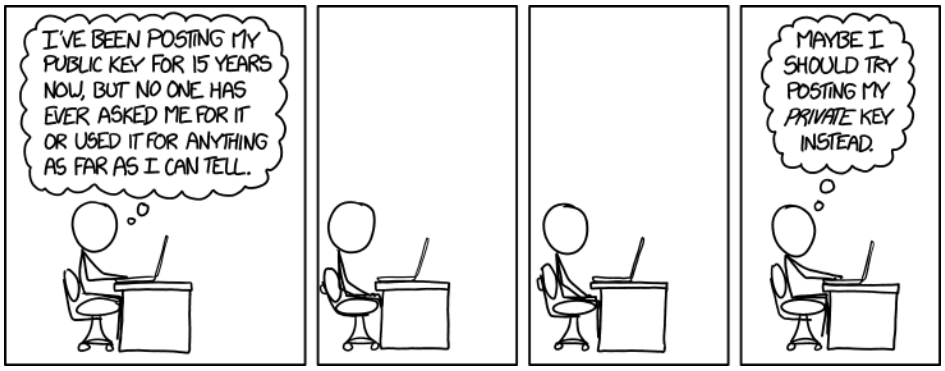


Figure 9.1: Even with a lack of attention, posting one's private key is *highly* discouraged. Created by Randall Munroe on `xkcd`; posted online at https://xkcd.com/1553/.

9.3.1 Intuition and Discussion

We would like to use mathematics to help Alice and Bob communicate.

Let G be a cyclic group with generator g and order $|G| = N$. W let $a, b \xleftarrow{\$} \mathbb{Z}_N^*$ be private keys and $g^a, g^b \in G$ be the corresponding public keys. We want to be able to combine this information in some way to derive a shared secret that Alice and Bob can use to securely communicate over an insecure channel. At the same time, it should be impractical for eavesdropper Eve to derive the shared secret from the public information.

If Alice has Bob's public key g^b, then she can compute

$$\left(g^b\right)^a = g^{ab}. \tag{9.1}$$

Similarly, if Bob has Alice's public key g^a, then he can compute

$$\left(g^a\right)^b = g^{ab}. \tag{9.2}$$

Thus, g^{ab} is a secret that both Alice and Bob can compute. This is their *shared secret*.

If it is difficult to compute g^{ab} given g^a and g^b, then this method enables Alice and Bob to compute a shared secret which is hard for Eve to determine. This shared secret can then be used to derive a secret key for secure communication using symmetric key encryption. *Deriving* a key from the shared secret and not using the shared secret directly is important for security. Some form of *key derivation function* (KDF) is generally used; we briefly mentioned KDFs based on hash functions in Chapter 8.8.

Example 9.1 (DH Example 1)
Code may be found at `examples/public/diffie-hellman_1.py`.

We now go through an example. Let us look at $\mathbb{F}_{7919}$; we let $7 \in \mathbb{F}_{7919}$ be the base point. It can be shown that $\langle 7 \rangle = \mathbb{F}_{7919}^*$ so that $|\langle 7 \rangle| = 7918$; for more information, see Appendix B.4.1 and Example B.13.

We let $a = 3545$ and $b = 6614$. Here, a is Alice's private key and b is Bob's private key. In this case, we have

$$\begin{aligned} A &:= 7^{3545} \mod 7919 \\ &= 5490 \\ B &:= 7^{6614} \mod 7919 \\ &= 407, \end{aligned} \tag{9.3}$$

where A is Alice's public key and B is Bob's public key.

Now, Alice knows B and computes B^a to learn their shared secret:

$$\begin{aligned} B^a &= 407^{3545} \mod 7919 \\ &= 4439. \end{aligned} \tag{9.4}$$

Similarly, Bob knows A and computes A^b:

$$\begin{aligned} A^b &= 5490^{6614} \mod 7919 \\ &= 4439. \end{aligned} \tag{9.5}$$

They can now derive a secret key from their shared secret 4439.

Example 9.2 (DH Example 2)
Code may be found at `examples/public/diffie-hellman_2.py`.

We now go through another example. The setup is the same: we look at $\mathbb{F}_{7919}$ and let $7 \in \mathbb{F}_{7919}$ be the base point.

We let $a = 1347$ and $b = 4862$. Then

$$\begin{aligned} A &:= 7^{1347} \mod 7919 \\ &= 683 \\ B &:= 7^{4862} \mod 7919 \\ &= 2710. \end{aligned} \tag{9.6}$$

Alice computes

$$\begin{aligned} B^a &= 2710^{1347} \mod 7919 \\ &= 7114. \end{aligned} \tag{9.7}$$

Bob computes

$$\begin{aligned} A^b &= 683^{4862} \mod 7919 \\ &= 7114. \end{aligned} \tag{9.8}$$

They can derive a secret key from their shared secret 7114.

9.3.2 Formal Definition

We now give the formal definition. This definition holds for any cyclic group.

Definition 9.1. Let $G = \langle g \rangle$ and $|G| = N$. Choose $a, b \xleftarrow{\$} \mathbb{Z}_N^*$ as private keys and set $A = g^a$ and $B = g^b$. Then $A, B \in G$ are public keys.

The *shared secret* is $g^{ab} \in G$. We have

$$\begin{aligned} B^a &= \left(g^b\right)^a \\ &= g^{ab} \end{aligned} \tag{9.9}$$

and

$$\begin{aligned} A^b &= (g^a)^b \\ &= g^{ab}. \end{aligned} \tag{9.10}$$

9.3.3 Potential Confusion between Public and Private Keys

Care must be taken to avoid confusion between private keys and public keys. From the definition, the private key is a element of $\mathbb{Z}_N^*$ while the public key is an element of G; that is, the private key is an *integer* and the public key is a *group element*.

Potential confusion may arise from the fact that when we are using subgroups of $\mathbb{F}_p$ (so that $G \leq \mathbb{F}_p$), then both the private key *and* the public key are integers. This may cause some initial confusion when trying to understand the Diffie-Hellman Key Exchange. In Chapter 11.2, we look at the Diffie-Hellman Key Exchange using subgroups of elliptic curves. In that case, the private key is an integer while the public key is a point on an elliptic curve.

9.3.4 Security of the Diffie-Hellman Key Exchange

A discussion of the security of the Diffie-Hellman Key Exchange can be found in Chapter 15.

9.4 Elgamal Encryption

We now describe the Elgamal encryption scheme [48].

The Diffie-Hellman Key Exchange made it possible to compute a shared secret from two public keys; that shared secret is the used to derive a secret key for symmetric key encryption. We now will consider the situation where encryption is based on the public key.

9.4.1 Formal Definition

Definition 9.2. We let G be a cyclic group with generator g of order N.

Key Generation Let $x \xleftarrow{\$} \mathbb{Z}_N^*$ be the private key. The public key is g^x.

Encryption Alice wishes to encrypt a message M to Bob.

Convert M in a reversible way to $m \in G$. Let $B = g^b$ be Bob's public key. Choose a nonce $y \xleftarrow{\$} \mathbb{Z}_N^*$. Set $s = B^y$; we note $s = g^{yb}$. Compute $c_1 = g^y$ and $c_2 = m \cdot s$. Alice sends (c_1, c_2) to Bob.

Decryption Bob receives (c_1, c_2) from Alice.

Set $s = c_1^b$. Because b is Bob's private key and $c_1 = g^y$,

$$\begin{aligned} s &= c_1^b \\ &= (g^y)^b \\ &= g^{yb}. \end{aligned} \tag{9.11}$$

Thus, s is the secret used by Alice; it is their "shared secret". Compute $m = c_2 \cdot s^{-1}$. Convert m into the message M.

9.4.2 Discussion and Examples

As it currently stands, one of the challenges is that y is a nonce: it must be random and *never* reused. If a message M and ciphertext pair (c_1, c_2) is known, then the secret s can be easily computed.

Example 9.3 (Elgamal Encryption)
Code may be found at `examples/public/elgamal_encryption.py`.

We use the same group as before: $\mathbb{F}_{7919}$ with generator $7 \in \mathbb{F}_{7919}$ as the base point. We recall that $\langle 7 \rangle = \mathbb{F}_{7919}^*$ and $|\langle 7 \rangle| = 7918$.

Encryption Alice wants to send the message $m = 1234$ to Bob. Bob has the private key $b = 3119$ and public key

$$\begin{aligned} g^b &= 7^{3119} \mod 7919 \\ &= 2106. \end{aligned} \tag{9.12}$$

Thus, $B = 2106$. Alice chooses random $y = 3185$. In this case,

$$\begin{aligned} c_1 &= g^y \\ &= 7^{3185} \mod 7919 \\ &= 2478. \end{aligned} \tag{9.13}$$

We also have the shared secret

$$\begin{aligned} s &= B^y \\ &= 2106^{3185} \mod 7919 \\ &= 1582. \end{aligned} \tag{9.14}$$

Finally, we have

$$\begin{aligned} c_2 &= m \cdot s \\ &= 1234 \cdot 1582 \mod 7919 \\ &= 4114. \end{aligned} \tag{9.15}$$

Thus, Alice sends $(c_1, c_2) = (2478, 4114)$ to Bob.

Decryption Bob receives $(c_1, c_2) = (2478, 4114)$ from Alice. We recall Bob's private key is $b = 3119$. Bob computes

$$\begin{aligned} s &= c_1^b \\ &= 2478^{3119} \mod 7919 \\ &= 1582. \end{aligned} \tag{9.16}$$

This is the shared secret used for encryption. Bob now needs to compute s^{-1}. We see $1582 \cdot 3519 = 1 \mod 7919$, so $s^{-1} = 3519$. Then we see

$$\begin{aligned} m &= c_2 \cdot s^{-1} \\ &= 4114 \cdot 3519 \mod 7919 \\ &= 1234. \end{aligned} \tag{9.17}$$

Thus, Bob correctly decrypted Alice's message.

9.5 Digital Signatures

Suppose Alice sends Bob a (non-secret) message. Can we ensure that Bob can prove to Charlie that Alice did send him that message? Yes; this can be done with a *digital signature*.

Due to the important nature of digital signatures, a thorough discussion will be postponed until Chapter 10.

9.6 Public Key Encryption in Practice

Public key encryption (in the form of Elgamal) requires *orders of magnitude* more work than the symmetric key encryption algorithms discussed in Chapter 6.

In practice, a symmetric key is generated and used to encrypt a message. Public key encryption is then used to encrypt the symmetric key. At this point, the encrypted symmetric key and encrypted message are then sent over an insecure channel from Alice to Bob. This is a *hybrid encryption scheme*. Although this is a very important topic in practice, we will not discuss it further; one reference is [69, Chapter 12.3].

Furthermore, it is important to note that some of these encryption schemes require the use of random numbers. It is *critical* that cryptographically-secure random numbers are used; cryptographically-secure pseudorandom number generators are discussed in Chapter 6.6. It is be better to use methods which do not rely on random numbers due to the difficulty in producing random numbers of sufficient quality required for cryptographic protocols; see [1] for a discussion on the randomness requirements for cryptography and [102] for a discussion on deterministic nonces.

9.7 Ciphertext Malleability

A ciphertext is *malleable* if it can be changed into another valid ciphertext for a related message. In general, this is an undesirable feature.

Example 9.4 (Malleability of Elgamal Encryption)
Let us assume that (c_1, c_2) is a valid ciphertext to Bob for a message m. We will show that (c_1, tc_2) is *another* valid ciphertext to Bob for message tm; here, $t \in G$ is arbitrary.

First, we see

$$c_1^b = s \tag{9.18}$$

as before. From here, we also have

$$\begin{aligned} s^{-1} \cdot [t \cdot c_2] &= s^{-1} \cdot [t \cdot (m \cdot s)] \\ &= t \cdot m. \end{aligned} \tag{9.19}$$

Thus, we may arbitrarily modify a valid ciphertext into another valid ciphertext.

We mention in passing that the original RSA encryption scheme [109] is malleable.

Chapter 10

Digital Signatures

This chapter is devoted to digital signatures. We will describe what they are as well as go through examples.

10.1 The Need for Digital Signatures

Digital signatures allow for *nonrepudiation*; that is, if Alice sends Bob a message m with a valid digital signature σ of m, Bob can be certain that Alice did indeed send him that message. Only if Eve stole Alice's signing (private) key would Eve be able to forge a signature. Thus, if Alice's signing key remains secure, it is not possible for Alice to later claim that she did not sign the message m. In particular, we want to ensure that Bob can present (m, σ) to Charlie as proof that Alice sent message m to Bob.

We note that digital signatures are different from message authentication codes; see Chapter 6.7. A message authentication code *does not* allow for nonrepudiation. Because of this, Bob is not able to use a MAC to prove to Charlie that Alice sent him a message because both Bob and Alice have the private key used in a MAC. Even though MACs do not allow for nonrepudiation, the fact remains that they require *orders of magnitude* less computation.

10.2 Clearing Up Potential Confusion about Digital Signatures

Digital signatures are sometimes explained as "performing public key encryption backwards." Although from a historical perspective this is understandable, this is *completely incorrect*. Digital signatures *should not* be thought of as the "reverse" of encryption. The *purpose* of a digital signature is enable Bob to prove to Charlie that Alice sent him a message; encryption *does not* play any role.

Now, it is understandable how this confusion arose. In the original RSA paper [109], public key encryption and digital signatures are both discussed. The authors describe a digital signature scheme which is merely their public key encryption scheme backwards. Of course, the original method described in [109] *should not be used* because it is *not secure*.

Additionally, it should be noted that Ralph Merkle in his dissertation [87, Chapter 1.4] makes a similar statement. As modern cryptography developed, it became clear that it was best to separate the notions of an encryption scheme and digital signature scheme due to the different security objectives.

This history explains why some may incorrectly describe digital signatures as "reverse encryption."

10.3 Elgamal Signatures

Elgamal signatures [48] are related to Elgamal encryption previously discussed in Chapter 9.4.

10.3.1 Formal Definition

Definition 10.1. We fix a prime p and let $g \in \mathbb{F}_p^*$ generate $\mathbb{F}_p^*$. Also, let $H : \{0,1\}^* \to \mathbb{Z}_{p-1}$ be a hash function.

Key Generation Choose $x \xleftarrow{\$} \mathbb{Z}_{p-1}^*$. Set $y = g^x \mod p$. Then x is the private key and y is the public key.

Signature Generation Choose a nonce $k \xleftarrow{\$} \mathbb{Z}_{p-1}^*$ such that $\gcd(k, p-1) = 1$. Set

$$\begin{aligned} r &= g^k \mod p \\ s &= [H(m) - xr]\, k^{-1} \mod p - 1. \end{aligned} \tag{10.1}$$

If $s = 0$, choose another k and repeat the process. The signature is (r, s).

Signature Verification The signature (r, s) is valid if $r, s \in \mathbb{F}_p^*$ and

$$g^{H(m)} = y^r r^s \mod p. \tag{10.2}$$

10.3.2 Verification

Suppose we receive (r, s) as previously defined. In this case, we see

$$H(m) = ks + xr \mod (p-1). \tag{10.3}$$

This allows the following equalities:

$$\begin{aligned} g^{H(m)} &= g^{ks+xr} \\ &= \left(g^k\right)^s \left(g^x\right)^r \\ &= r^s y^r. \end{aligned} \tag{10.4}$$

Thus, we have the desired equality.

The Necessary Requirement of $s \neq 0$

.

We briefly describe why we must have $s \neq 0$. Suppose $s = 0$. Then we have

$$H(m) = xr \mod (p-1). \tag{10.5}$$

This implies that

$$x = H(m)r^{-1} \mod (p-1). \tag{10.6}$$

Thus, if $s = 0$, then Alice leaks her private key. *This should never happen.*

10.3.3 Security Considerations

If the nonce k is reused, the security of the scheme breaks because the private key can be computed; the reader should work out the details. A similar derivation can be found in Chapter 10.4.4.

10.3.4 Example

Example 10.1 (Elgamal Signature)
Code may be found at `examples/signatures/elgamal_signature.py`.

We will again work within $\mathbb{F}_p$ with $p = 7919$ and base point $g = 7$. Alice chooses her private key $x = 5654$; her public key is

$$\begin{aligned} y &= g^x \mod p \\ &= 5581. \end{aligned} \tag{10.7}$$

Alice has a message m she wishes to sign. We have $H(m) = 2689$.

Signature Generation Alice chooses her nonce $k = 3331$; we note $\gcd(7918, 3331) = 1$. Additionally, we see In this case, we have

$$\begin{aligned} r &= g^k \mod p \\ &= 3521. \end{aligned} \tag{10.8}$$

We then have

$$\begin{aligned} s &= [H(m) - xr]\, k^{-1} \mod (p-1) \\ &= 585. \end{aligned} \tag{10.9}$$

The signature is $(3521, 585)$.

Signature Verification We now want to verify the signature $(3521, 585)$ from Alice with public key $y = 5581$. The message m has the hash $H(m) = 2689$.

We look at the left-hand side:

$$g^{H(m)} \mod p = 367. \tag{10.10}$$

We look at the right-hand side:

$$y^r r^s \mod p = 367. \tag{10.11}$$

Thus, our signature is valid.

10.4 Schnorr Signatures

One difficulty with Elgamal signatures is that they are large. Schnorr signatures produce smaller signatures.

10.4.1 Formal Definition

Definition 10.2. We let $G = \langle g \rangle$ be a cyclic group with prime order q. We fix a hash function $H : \{0,1\}^* \to \mathbb{Z}_q$. Here, $m \in \{0,1\}^*$ is the message to be signed.

Key Generation Let $x \xleftarrow{\$} \mathbb{Z}_q^*$ be the signing key. Then $y = g^x$ is the public key.

Signature Generation Choose a nonce $k \xleftarrow{\$} \mathbb{Z}_q^*$. Set $r = g^k$. Let

$$e = H(r \| m); \tag{10.12}$$

here, we have a deterministic binary representation for r. Set

$$s = k - xe \mod q. \tag{10.13}$$

The signature is (s, e).

Signature Verification We receive the message m along with (s, e). Let

$$\begin{aligned} r_v &= g^s y^e \\ e_v &= H(r_v \| m). \end{aligned} \tag{10.14}$$

The signature is valid if $e_v = e$.

10.4.2 Verification

We now verify the result. In this case, if we use the values as defined, we see

$$\begin{aligned} r_v &= g^s y^e \\ &= g^{k-xe} g^{xe} \\ &= g^k. \end{aligned} \tag{10.15}$$

Therefore, $r_v = r$ and

$$\begin{aligned} e_v &= H(r_v \| m) \\ &= H(r \| m) \\ &= e. \end{aligned} \tag{10.16}$$

Thus, the signature is valid.

10.4.3 Group Realization

Here, we presented the Schnorr signature in a general cyclic group. In practice, this was originally designed to be implemented in a subgroup of order q in $\mathbb{F}_p$ for large prime p. This is similar to the situation in DSA in Chapter 10.5. More explicitly, let p be a large prime with

$$p = rq + 1. \tag{10.17}$$

Furthermore, let $h \in \mathbb{F}_p^*$ be a primitive element; that is, let h be a generator so that $\langle h \rangle = \mathbb{F}_p^*$. Then set

$$g = h^r \mod p. \tag{10.18}$$

In this case, $|\langle g \rangle| = q$ and so $\langle g \rangle$ is the desired cyclic group of prime order q.

10.4.4 Security Considerations

Care must be taken to ensure the nonce is not reused. Nonce reuse easily leads to recovery of the private key. In fact, mere *bias* in the nonce distribution can lead to private key recovery; see [32]. The use of deterministic nonces have been recommended for some time [102].

Private Key Recovery from Nonce Reuse

We show what happens when the nonce is reused.

Suppose Alice signs distinct messages m and m' but reuses the nonce k to save computation. That is, we have $r = g^k$ with

$$\begin{aligned} e &= H(r\|m) \\ s &= k - xe \\ e' &= H(r\|m') \\ s' &= k - xe'. \end{aligned} \tag{10.19}$$

Somehow, the adversary Eve knows that k is reused; this could be determined by verifying the signatures and then noticing the computed r values are the same. Using this information, we see

$$\begin{aligned} s - s' &= (k - xe) - (k - xe') \\ &= x\left(e' - e\right), \end{aligned} \tag{10.20}$$

so it follows that

$$x = \frac{s - s'}{e' - e} \mod q. \tag{10.21}$$

Therefore, Eve has now learned Alice's private key.

10.4.5 Example

Example 10.2 (Schnorr Signature)
Code may be found at `examples/signatures/schnorr_signature.py`.

We have the following:

$$\begin{aligned} p &= cq + 1 \\ q &= 2^{32} + 15 \\ c &= 12. \end{aligned} \tag{10.22}$$

It can be shown that p and q are prime numbers and that

$$h = 7 \tag{10.23}$$

is a generator for $\mathbb{F}_p^*$. Our q-order subgroup is generated by

$$\begin{aligned} g &= h^c \mod p \\ &= 13841287201. \end{aligned} \tag{10.24}$$

See Chapter 13.2 for more information.

Alice chooses her private key and public key so that

$$\begin{aligned} x &= 2981464014 \\ y &= g^x \mod p \\ &= 2966837275. \end{aligned} \tag{10.25}$$

Alice wants to sign the message

$$m = \texttt{00010203}. \tag{10.26}$$

Signature Generation Alice chooses

$$\begin{aligned} k &= 458074281 \\ r &= g^k \mod p \\ &= 2699985659. \end{aligned} \tag{10.27}$$

She then computes

$$\begin{aligned} e &= \mathrm{MD5}\,(r \| m) \mod q \\ &= 1756904634. \end{aligned} \tag{10.28}$$

We then have

$$\begin{aligned} s &= (k - xe) \mod q \\ &= 1743605188. \end{aligned} \tag{10.29}$$

The signature is $(1743605188, 1756904634)$.

Signature Verification Bob receives $(1743605188, 1756904634)$ along with the message $m = \texttt{00010203}$. Alice has the public key

$$y = 2966837275. \tag{10.30}$$

Bob computes

$$\begin{aligned} r_v &= g^s y^e \mod p \\ &= 2699985659. \end{aligned} \tag{10.31}$$

He then sees

$$\begin{aligned} e_v &= \mathrm{MD5}\,(r_v \| m) \mod q \\ &= 1756904634. \end{aligned} \tag{10.32}$$

Bob determines the signature is valid because $e = e_v$.

10.4.6 Schnorr Signature Patent

Claus Schnorr held the patent on Schnorr signatures until it expired in 2008. This led to the development of the Digital Signature Algorithm (discussed in Chapter 10.5).

10.5 Digital Signature Algorithm (DSA)

The Digital Signature Algorithm (DSA) is another signature algorithm. It is more complicated than a Schnorr signature and based on the Elgamal signature scheme in Chapter 10.3.

10.5.1 Formal Definition

Definition 10.3. Let p be a prime. Choose $g \in \mathbb{F}_p$ such that $|\langle g \rangle| = q$, a prime. Fix a hash function $H : \{0,1\}^* \to \mathbb{Z}_q$. Here, $m \in \{0,1\}^*$ is the message to be signed.

Key Generation Choose $x \xleftarrow{\$} \mathbb{Z}_q^*$ and let $y = g^x \mod p$. We have public key y and private key x.

Signature Generation Choose a nonce $k \xleftarrow{\$} \mathbb{Z}_q^*$. Set

$$r = \left[g^k \mod p\right] \mod q. \tag{10.33}$$

If $r = 0$, restart and choose another k. Set

$$s = k^{-1} \left[H(m) + xr\right] \mod q. \tag{10.34}$$

If $s = 0$, restart and choose another k. The signature is (r, s).

Signature Verification Let m be a message with signature (r, s). First, verify $0 < r < q$ and $0 < s < q$. Set

$$\begin{aligned} w &= s^{-1} \mod q \\ u_1 &= H(m) \cdot w \mod q \\ u_2 &= r \cdot w \mod q \\ v &= [g^{u_1} y^{u_2} \mod p] \mod q. \end{aligned} \tag{10.35}$$

The signature is valid if $v = r$.

10.5.2 Verification

Suppose we have message m and signature (r, s) as defined above. Then we have

$$\begin{aligned} u_1 &= H(m) \cdot s^{-1} \mod q \\ u_2 &= r \cdot s^{-1} \mod q. \end{aligned} \tag{10.36}$$

It follows that

$$\begin{aligned} g^{u_1} y^{u_2} &= g^{H(m)s^{-1}} y^{rs^{-1}} \\ &= g^{H(m)s^{-1}} g^{xrs^{-1}} \\ &= g^{[H(m)+xr]s^{-1}} \\ &= g^k. \end{aligned} \tag{10.37}$$

Therefore, we see

$$\begin{aligned} v &= [g^{u_1} y^{u_2} \mod p] \mod q \\ &= \left[g^k \mod p\right] \mod q \\ &= r, \end{aligned} \tag{10.38}$$

as desired.

The Necessary Requirement of $r \neq 0$

If $r = 0$, then we also have $u_2 = 0$. This implies that we have a valid signature for *any* public key; additionally, k can also be recovered. Thus, we cannot let $r = 0$.

The Necessary Requirement of $s \neq 0$

If $s = 0$, then we cannot compute the modular inverse in verification. Thus, we cannot verify the signature.

10.5.3 Security Considerations

Reusing the nonce k can lead to recovery of the private key. The reader should work out the details.

10.5.4 Example

Example 10.3 (DSA Signature)
Code may be found at `examples/signatures/dsa_signature.py`.

We use the same group from Example 10.2; namely, we have

$$\begin{aligned} p &= cq + 1 \\ q &= 2^{32} + 15 \\ c &= 12 \\ h &= 7 \\ g &= h^c \mod p \\ &= 13841287201. \end{aligned} \tag{10.39}$$

Alice chooses her private key and public key so that

$$\begin{aligned} x &= 3713319676 \\ y &= g^x \mod p \\ &= 43016495308. \end{aligned} \tag{10.40}$$

Alice wants to sign the same message

$$m = \texttt{00010203} \tag{10.41}$$

as before; this time, she wants to use DSA.

Signature Generation Alice chooses

$$\begin{aligned} k &= 2073840447 \\ r &= \left[g^k \mod p\right] \mod q \\ &= 1886505440. \end{aligned} \tag{10.42}$$

She then computes

$$\begin{aligned} s &= k^{-1}\left[H(m) + xr\right] \mod q \\ &= 2123944771. \end{aligned} \tag{10.43}$$

The signature is $(1886505440, 2123944771)$.

Signature Verification Bob receives $(1886505440, 2123944771)$ along with the message $m = \texttt{00010203}$. Alice has the public key

$$y = 43016495308. \tag{10.44}$$

Bob computes

$$\begin{aligned} w &= s^{-1} \mod q \\ &= 1156096597 \\ u_1 &= H(m) \cdot w \mod q \\ &= 938259556 \\ u_2 &= r \cdot w \mod q \\ &= 75869898 \\ v &= (g^{u_1} y^{u_2} \mod p) \mod q \\ &= 1886505440. \end{aligned} \tag{10.45}$$

Bob determines the signature is valid because $r = v$.

10.6 Problems with DSA

There are some critical problems with DSA.

It is critical that the nonce k be sufficiently random. As noted above, nonce reuse allows for the recovery of the signing key. Furthermore, if k is not chosen uniformly but comes from a biased distribution as a result of poor random number generation, then information about the signing key can be leaked. This information may lead to signing key recovery; see [93]. Because of this, it is *critical* that k be chosen carefully. It is *recommended* to deterministically choose k [102].

The fact there are subexponential algorithms for factoring integers leads to subexponential algorithms for breaking DSA. This is discussed more in Chapter 11.1.

10.7 Comparison of Digital Signature Schemes

We now perform a comparison between the different signature algorithms and why one might prefer one to the others.

Given the standard security requirements (discussed in Chapter 15; see Table 15.1), the above signature algorithms require p to be a 3072-bit prime. In the case of DSA, q is required to be a 256-bit prime.

10.7.1 Public Key Length

We begin by discussing the public key length:

- Elgamal: 3072 bits; 384 bytes
- Schnorr: 3072 bits; 384 bytes
- DSA: 3072 bits; 384 bytes

In each case, the public key length is the same.

10.7.2 Signature Length

We now discuss the signature length:

- Elgamal: $2 \cdot 3072$ bits; 768 bytes
- Schnorr: $3072 + 256$ bits; 416 bytes
- DSA: $2 \cdot 256$ bits; 64 bytes

In each case, DSA clearly has the smallest signature. Schnorr also produces a small signature.

10.7.3 Computational Complexity

Finally, we discuss the computational complexity for signature generation:

- Elgamal: 1 exponentiation, 1 modular inverse, 1 hash
- Schnorr: 1 exponentiation, 1 hash
- DSA: 1 exponentiation, 1 modular inverse, 1 hash

Here, we see that DSA and Elgamal have essentially the same complexity for signing. We remember that when working in $\mathbb{F}_p$, exponentiation and modular inversion have similar complexity. Furthermore, hashing is cheap. Thus, Schnorr signatures have the smallest complexity.

10.7.4 Conclusion

After comparing major aspects of these digital signatures, we see that Schnorr signatures can be computed the fastest while having signatures that are not too large. One challenge, as noted above, is that Schnorr signatures were patented. For this reason, DSA was developed by NIST to be algorithm that everyone could use. It results in a smaller signature but larger computational cost when compared with Schnorr.

As noted, *all* of these algorithms have problems when k is not uniformly random. Some attacks can be found [32, 93]. The use of deterministic nonces is *strongly encouraged* [102].

10.8 Malleable Signatures

Similar to encryption schemes, digital signature schemes may also be malleable. We mention that the original RSA signature algorithm [109] is malleable.

Chapter 11

Elliptic Curve Cryptography

We now begin our discussion of Elliptic Curve Cryptography; that is, cryptography which uses elliptic curves. In this chapter we focus on a variant of the Diffie-Hellman Key Exchange which uses a cyclic group based on elliptic curves. We also present digital signatures based on elliptic curves which are generalizations of Schnorr signatures and DSA.

11.1 Problems with Subgroups of Finite Fields

One of the difficulties with $\mathbb{F}_p^*$ is that each prime p uniquely determines the potential multiplicative subgroups. The security depends on the factorization of $p-1$; this factorization will determine how difficult it is to solve the Discrete Logarithm Problem, which is the basis for security. Additionally, there are subexponential algorithms which can solve the Discrete Logarithm Problem over $\mathbb{F}_p$; a brief discussion of the *index calculus algorithm* may be found in [69, Chapter 10.3]. These facts imply that the size of p must be chosen quite large; see Chapter 15 and Table 15.1 for more information. Large values of p implies greater computational and storage costs. This makes them challenging to use in resource-constrained environments.

Elliptic curves solve this problem. For any given prime p, there are many elliptic curves which can be defined over the finite field $\mathbb{F}_p$. We have $\#E(\mathbb{F}_p) \simeq p$; see Theorem 5.3 for a more information. Given the freedom of both the underlying finite field and the elliptic curve equation, it would seem reasonable that certain elliptic curves could be constructed with desirable properties; this is the case. Furthermore, there are large classes of elliptic curves where the best known algorithms for solving the Discrete Logarithm Problem are *generic*; that is, the methods work on all groups and do not take advantage of the algebraic structure of the elliptic curve.

For 128-bit security, it is suggested that p be a 3072-bit prime for DSA and DH performed inside subgroups of $\mathbb{F}_p^*$. Storing a 3072-bit public key requires 384 bytes. This is significant storage requirement. A similar security level requires an elliptic curve over a 256-bit prime field; the public key is 64 bytes (uncompressed). As mentioned before, see the discussion in Chapter 15 and Table 15.1. Of course, the smaller size of public key requires more complicated group arithmetic, and care must be taken that these methods are computed securely.

11.2 Elliptic Curve Diffie-Hellman (ECDH)

11.2.1 Discussion

Elliptic Curve Diffie-Hellman (ECDH) is the Diffie-Hellman Key Exchange over groups based on elliptic curves. The definition is exactly the same as Def. 9.1 in Chapter 9.3; the only difference is that we are using a different cyclic group.

11.2.2 Examples

Plots for Examples 11.1 and 11.2 may be found in Figure 11.1.

Example 11.1 (Elliptic Curve Diffie-Hellman 1)
Code may be found at `examples/ecc/ecdh_1.py`.

We continue with the elliptic curve

$$E : y^2 = x^3 + 2x + 1 \tag{11.1}$$

over $\mathbb{F}_{71}$. This curve was previously used in Chapter 5.2. We will use this curve along with the base point $P = (0, 1)$. We can see that $|\langle(0, 1)\rangle| = 32$; that is, the point $(0, 1)$ generates a subgroup of order 32.

Alice chooses her private key so that

$$\begin{aligned} a &= 29 \\ A &= a \cdot P \\ &= 29 \cdot (0, 1) \\ &= (8, 48)\,. \end{aligned} \tag{11.2}$$

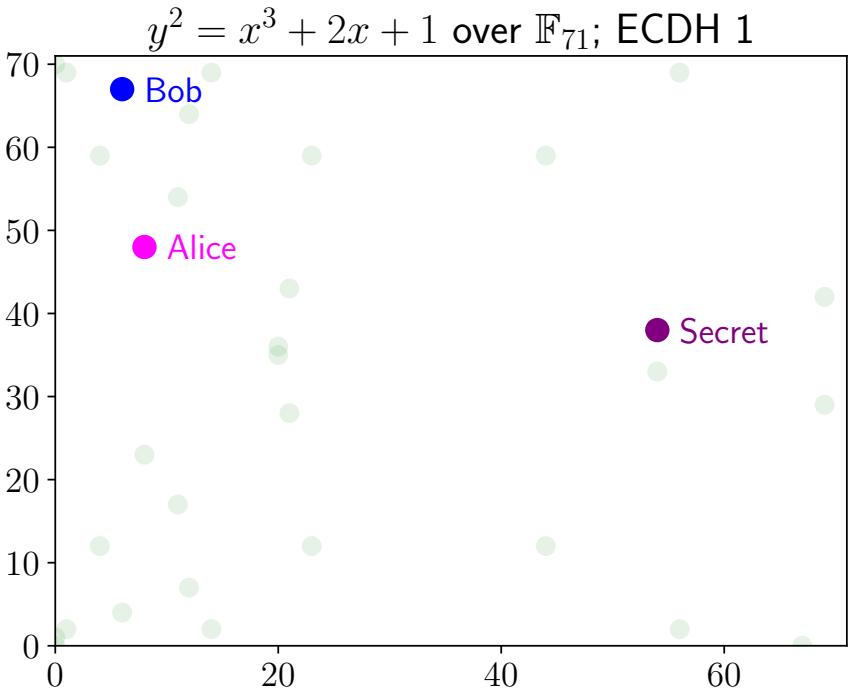
Similarly, Bob chooses his private key so that

$$\begin{aligned} b &= 14 \\ B &= b \cdot P \\ &= 14 \cdot (0, 1) \\ &= (6, 67)\,. \end{aligned} \tag{11.3}$$

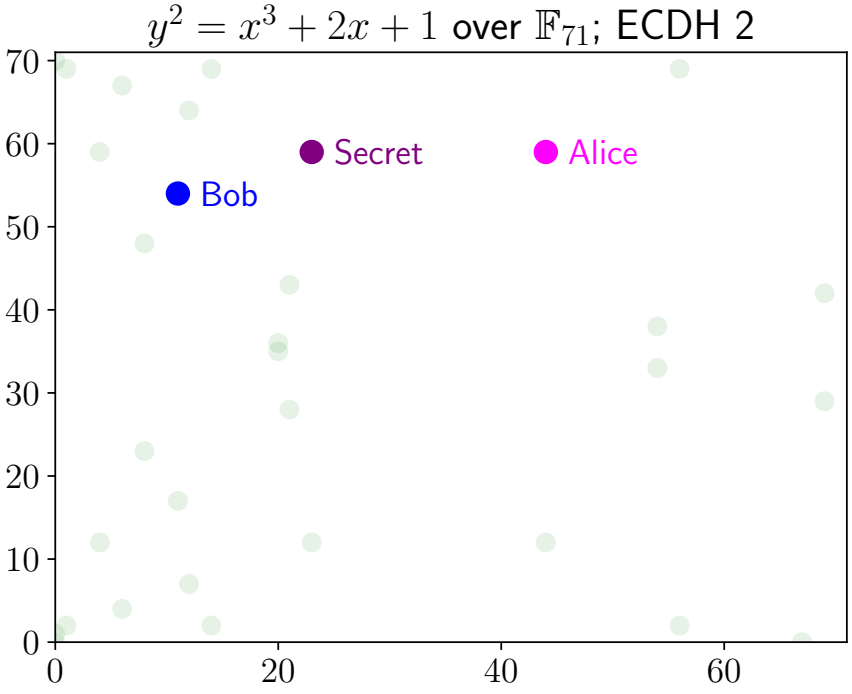
Alice retrieves Bob's public key and computes

$$\begin{aligned} K &= a \cdot B \\ &= 29 \cdot (6, 67) \\ &= (54, 38)\,. \end{aligned} \tag{11.4}$$

Bob learns Alice's public key and determines



(a) Plot of Alice's public key $A = (8, 48)$, Bob's public key $B = (6, 67)$, and the shared secret $K = (54, 38)$ for Example 11.1.



(b) Plot of Alice's public key $A = (44, 59)$, Bob's public key $B = (11, 54)$, and the shared secret $K = (23, 59)$ for Example 11.2.

Figure 11.1: Here we plot Alice's public key $A = a \cdot P$, Bob's public key $B = b \cdot P$, and the shared secret $K = (ab) \cdot P$ for Examples 11.1 and 11.2.

$$\begin{aligned} K &= b \cdot A \\ &= 14 \cdot (8, 48) \\ &= (54, 38)\,. \end{aligned} \tag{11.5}$$

They can now use this shared secret to derive a secret key for secure communication. See Figure 11.1a for a plot of

$$\begin{aligned} A &= (8, 48) \\ B &= (6, 67) \\ K &= (54, 38). \end{aligned} \tag{11.6}$$

Example 11.2 (Elliptic Curve Diffie-Hellman 2)
Code may be found at `examples/ecc/ecdh_2.py`.

We continue with the setup we used in Example 11.1 with the elliptic curve

$$E : y^2 = x^3 + 2x + 1 \tag{11.7}$$

over $\mathbb{F}_{71}$ and base point $P = (0, 1)$. The order of the group is 32.

Alice chooses

$$\begin{aligned} a &= 17 \\ A &= a \cdot P \\ &= 17 \cdot (0, 1) \\ &= (44, 59) \end{aligned} \tag{11.8}$$

while Bob chooses

$$\begin{aligned} b &= 27 \\ B &= b \cdot P \\ &= 27 \cdot (0, 1) \\ &= (11, 54)\,. \end{aligned} \tag{11.9}$$

Alice retrieves Bob's public key and computes

$$\begin{aligned} K &= a \cdot B \\ &= 17 \cdot (11, 54) \\ &= (23, 59)\,. \end{aligned} \tag{11.10}$$

Bob learns Alice's public key and determines

$$\begin{aligned} K &= b \cdot A \\ &= 27 \cdot (44, 59) \\ &= (23, 59)\,. \end{aligned} \tag{11.11}$$

They can now use this shared secret to derive a secret key for secure communication. See Figure 11.1b for a plot of

$$\begin{aligned} A &= (44, 59) \\ B &= (11, 54) \\ K &= (23, 59). \end{aligned} \tag{11.12}$$

11.3 Elliptic Curve DSA (ECDSA)

The Elliptic Curve Digital Signature Algorithm (ECDSA) reformulates DSA (Def. 10.3) into a digital signature algorithm for elliptic curves. There are many similarities with some differences due to different groups.

11.3.1 Formal Definition

Definition 11.1. Let $E/\mathbb{F}_p$ be an elliptic curve with $G \in E(\mathbb{F}_p)$ a specified base point and generator. We assume $\#E(\mathbb{F}_p) = n$ is prime. We fix a hash function $H : \{0,1\}^* \to \mathbb{Z}_n$.

Key Generation Choose private key $d \xleftarrow{\$} \mathbb{Z}_n^*$. The public key is $Q = d \cdot G$.

Signature Generation We let m be the message to be signed. Choose a nonce $k \xleftarrow{\$} \mathbb{Z}_n^*$. Set

$$\begin{aligned} (x, y) &= k \cdot G \\ r &= x \mod n \\ s &= k^{-1}\left[H(m) + rd\right] \mod n. \end{aligned} \tag{11.13}$$

If $r = 0$, choose another k and repeat. If $s = 0$, choose another k and repeat. The signature is (r, s).

Signature Verification We receive a signature (r, s) from Q for message m. First, confirm $0 < r < n$, $0 < s < n$, and $n \cdot Q = \mathcal{O}$. Set

$$\begin{aligned} w &= s^{-1} \mod n \\ u_1 &= H(m)w \mod n \\ u_2 &= rw \mod n \\ (x, y) &= u_1 \cdot G + u_2 \cdot Q. \end{aligned} \tag{11.14}$$

The signature is valid if $r = x \mod n$.

11.3.2 Verification

Let (r, s) be as previously defined for the signature for message m and public key Q. Then we see

$$\begin{aligned} (x, y) &= u_1 \cdot G + u_2 \cdot Q \\ &= \left(H(m)s^{-1}\right) \cdot G + \left(drs^{-1}\right) \cdot G \\ &= s^{-1} \left[H(m) + dr\right] \cdot G \\ &= k \cdot G. \end{aligned} \tag{11.15}$$

Thus, we have a valid signature.

The Necessary Requirement of $r \neq 0$

If $r = 0$, the signature will be valid for any public key. This should not happen.

The Necessary Requirement of $s \neq 0$

If $s = 0$, then we cannot compute the modular inverse during signature verification. Thus, we cannot verify the signature.

Nonce Reuse

Reusing the nonce k leads to recovery of the private key. The reader should work out the details.

11.3.3 ECDSA Malleability

We note that there is some malleability to the signature. We mean this: let (r, s) be the signature of message m for public key Q. As noted previously in Eq. (11.15), we have

$$\begin{aligned} (x, y) &= u_1 \cdot G + u_2 \cdot Q \\ r &= x \mod n. \end{aligned} \tag{11.16}$$

We now look at the pair $(r, -s)$. In this case, we see

$$\begin{aligned}\tilde{w} &= (-s)^{-1} \mod n \\ &= -w \\ \tilde{u}_1 &= H(m)w \mod n \\ &= -u_1 \\ \tilde{u}_2 &= rw \mod n \\ &= -u_2. \end{aligned} \tag{11.17}$$

This implies that

$$\begin{aligned}\tilde{u}_1 \cdot G + \tilde{u}_2 \cdot Q &= -u_1 \cdot G - u_2 \cdot Q \\ &= -\left[u_1 \cdot G + u_2 \cdot Q\right] \\ &= -\left(x, y\right) \\ &= \left(x, -y\right). \end{aligned} \tag{11.18}$$

Thus, $(r, -s)$ is *also* a valid ECDSA signature for message m under the public key Q.

While this type of malleability is not of much concern in practice, we point this out because it will come up when we discuss Recoverable ECDSA in Ethereum in Chapter 11.6.2.

11.4 Problems with ECDSA

The same problems from biased nonces which arise in DSA also arise in ECDSA. Additionally, the nature of scalar multiplication in elliptic curves requires care to ensure the nonce k is not leaked from side-channel analysis even when using deterministic nonces; we recall that leaking a nonce results in the recovery of the private key. These side-channel attacks are *entirely practical* [37, 61, 132].

11.5 Edwards Curve DSA (EdDSA)

Edwards Curve Digital Signature Algorithm (EdDSA) was developed to be a digital signature algorithm which would overcome many of the pitfalls inherent to previous signature algorithms. In particular, the nonce is deterministic in its definition. Once a private key is computed, there is no further need for cryptographic random numbers. This reduces the likelihood of poor implementations. One of the authors of EdDSA wrote a blog post[1] about its design; later, he gave his thoughts[2] on why EdDSA is more immune to attacks than ECDSA.

We do not present a formal definition of EdDSA; this can be found in references [18, 20, 63]. The main advantages come from its deterministic nature (the nonce k is deterministic based on the private key and message), no branch logic (a specific curve type is chosen with

[1] https://blog.cr.yp.to/20140323-ecdsa.html
[2] https://blog.cr.yp.to/20191024-eddsa.html

one addition formula), its speed, and the small size of public keys and signatures. The design helps ensure immunity to side-channel attacks.

11.6 Recoverable ECDSA

In Ethereum, the ECDSA signatures used are *Recoverable* ECDSA signatures. This means that, given the message and associated signature, it is possible to *recover* the corresponding public key. The goal of this section to understand this. First, we work through the process in general before focusing on the particular case of Ethereum.

11.6.1 Deriving the Public Key from the ECDSA Signature

We work through the details of deriving the public key from the message and its ECDSA signature. We will use the same notation from Chapter 11.3 and review what we know:

- Elliptic curve $E/\mathbb{F}_p$ with generator $G \in E(\mathbb{F}_p)$.
- The number of points on the curve $\#E(\mathbb{F}_p) = n$ is prime.
- We have a hash function $H : \{0,1\}^* \to \mathbb{Z}_n$.
- We have private key $d \xleftarrow{\$} \mathbb{Z}_n^*$ with public key $Q = d \cdot G$.
- We have nonce $k \xleftarrow{\$} \mathbb{Z}_n^*$.
- We have a valid signature (r, s) for message m where

$$\begin{aligned} (x, y) &= k \cdot G \\ r &= x \mod n \\ s &= k^{-1}\left[H(m) + rd\right] \mod n. \end{aligned} \tag{11.19}$$

  A valid signature implies $r \neq 0$ and $s \neq 0$.

- We set $P = k \cdot G = (x, y)$. The valid signature also implies that

$$\begin{aligned} w &= s^{-1} \mod n \\ u_1 &= H(m)w \mod n \\ u_2 &= rw \mod n \\ P &= u_1 \cdot G + u_2 \cdot Q. \end{aligned} \tag{11.20}$$

  This means we can solve for the public key Q:

$$Q = u_2^{-1}\left[P - u_1 \cdot G\right]. \tag{11.21}$$

We now walk through the recovery process. We know that $(x, y) = k \cdot G$, so that $x \in \mathbb{F}_p$. First, we see $r = x \mod n$ so that $r \leq x$. Thus, we have

$$x = \alpha n + r \tag{11.22}$$

for $\alpha \geq 0$ and $r \in \mathbb{Z}_n$.

Previously, we did not make any assumptions about n and x; we must now make some assumptions in order to proceed. We either have $p \leq n$ or $p > n$.

- Case 1: $p \leq n$

  Because we already know $r \leq x$, the restriction $p \leq n$ implies that $x < n$ so that $x = r$. This implies that $\alpha = 0$ in Eq. (11.22).

  Now that we have determined x, we can find y such that

  $$y^2 = x^3 + ax + b. \tag{11.23}$$

  The points

  $$\begin{aligned} R_1 &= (x, y) \\ R_2 &= (x, p - y) \end{aligned} \tag{11.24}$$

  are both potential values for $P = k \cdot G$ (assuming $y \neq 0$); because k is unknown, we do not know which is correct. From here, there are 2 possibilities for the public key Q:

  $$\begin{aligned} Q_1 &= u_2^{-1} \left[R_1 - u_1 \cdot G \right] \\ Q_2 &= u_2^{-1} \left[R_2 - u_1 \cdot G \right]. \end{aligned} \tag{11.25}$$

  Nothing else can be determined without additional information.

- Case 2: $p > n$

  In this situation, we need to be careful. Here, we know that $x \geq r$. From the Hasse Theorem (Theorem 5.3) we have $\alpha \in \{0, 1\}$ in Eq. (11.22). This implies we have two possibilities for x: $x_1 = r$ and $x_2 = r + n$. From here, we set y_1 and y_2 such that

  $$\begin{aligned} y_1^2 &= x_1^3 + ax_1 + b \\ y_2^2 &= x_2^3 + ax_2 + b. \end{aligned} \tag{11.26}$$

  In this case, there are 4 possible values for P:

$$\begin{aligned} R_1 &= (x_1, y_1) \\ R_2 &= (x_1, p - y_1) \\ R_3 &= (x_2, y_2) \\ R_4 &= (x_2, p - y_2) \,. \end{aligned} \tag{11.27}$$

Therefore, there are 4 possibilities for the public key Q:

$$\begin{aligned} Q_1 &= u_2^{-1} \left[R_1 - u_1 \cdot G \right] \\ Q_2 &= u_2^{-1} \left[R_2 - u_1 \cdot G \right] . \\ Q_3 &= u_2^{-1} \left[R_3 - u_1 \cdot G \right] \\ Q_4 &= u_2^{-1} \left[R_4 - u_1 \cdot G \right] . \end{aligned} \tag{11.28}$$

Nothing else can be determined without additional information.

As noted, additional information is required in order to recover the specific public key. For instance, it is sufficient to specify which x value (corresponding to $\alpha = 0$ or $\alpha = 1$) as well as the parity of y (whether y is even or odd).

11.6.2 Recoverable ECDSA in Ethereum

We now walk through Recoverable ECDSA in Ethereum.

We review material from the Ethereum Yellowpaper [135, Appendix F]. Note that the paper has been updated and we are referencing the *Berlin* version; this is important because different versions use different conventions.

First, we note some differences with the Recoverable ECDSA from above: instead of the pair (r, s), we have the triple (r, s, v). Here, we have the restrictions

$$\begin{aligned} r &\in \{1, 2, \cdots, n-1\} \\ s &\in \left\{ 1, 2, \cdots, \frac{n+1}{2} \right\} \\ v &\in \{0, 1\} \,. \end{aligned} \tag{11.29}$$

Here, we are letting

$$n = \texttt{secp256k1n} \tag{11.30}$$

from [135, Eq. 313]; the restrictions in Eq. (11.29) are from [135, Eqs. 310–312]. Thus, n is the number of unique private keys; this agrees with our notation.

In this case, r is the x-coordinate; see the discussion in [135, Appendix F]. From the wording, it appears that the possibility of $r = x + n$ is *not allowed*; we do note this occurs

for a small fraction of all possible public keys (approximately 2^{-128} of all public keys). The additional restriction on s is due to concerns about signature malleability as discussed in Chapter 11.3.3; with this restriction, there is only *one* valid ECDSA signature for each message. This is discussed in Ethereum Improvement Proposal 2 (EIP-2)[3]. In order to uniquely determine the y coordinate, v specifies whether y is even ($v = 0$) or odd ($v = 1$); we note that previous versions of the Ethereum Yellowpaper [134, Appendix F] used a different convention for v: $v = 27$ for even y-coordinate and $v = 28$ for odd y-coordinate.

In summary: given the triple (r, s, v), the pair (r, s) is used in signature verification while v uniquely determines the parity of the y-coordinate of the public key.

11.7 Mnemonic Seed Phrases and BIP-39

11.7.1 The Challenge of Long Passwords

One of the challenges with cryptography in practice is using private keys. In the case of Elliptic Curve Cryptography with the desired 128-bit security, the private key is 256 bits. This is a long password; it is 64 hexadecimal characters. If it is going to have full entropy (that is, be truly random), it would take considerable effort to remember the password.

Mnemonic Seed Phrases were developed to make this easier; this is described in Bitcoin Improvement Proposal 39 (BIP-39)[4] and we will work to understand that material now.

The main idea is to convert a password that is difficult to remember into another form which is easier to remember or store correctly; additionally, we will be able to easily rederive the private key material when necessary. Using the password

$$\begin{aligned}&\text{password}\\ &:= \texttt{9e6291970cb44dd94008c79bcaf9d86f18b4b49ba5b2a04781db7199ed3b9e4e},\end{aligned} \tag{11.31}$$

would be difficult in practice; 64 hexadecimal characters is long and there is no apparent pattern. On the other hand, the fact that

$$\begin{aligned}&\text{SHA-3-256}(\texttt{0000000000000000000000000000000000000000000000000000000000000000})\\ &= \texttt{9e6291970cb44dd94008c79bcaf9d86f18b4b49ba5b2a04781db7199ed3b9e4e}\end{aligned} \tag{11.32}$$

makes remembering the password easier: the password is the SHA-3-256 hash of 32 zero bytes (64 zero hexadecimal characters). A mnemonic seed phrase is similar, in that it converts randomness into a more understandable form.

11.7.2 Mnemonic Seed Phrases

A *Mnemonic Seed Phrase* is a list of words which are used to then derive a seed; this seed can then be used to construct private keys. The method of derivation is important; one

[3] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.md
[4] https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki

method of standardization is BIP-39[5]. We note that BIP-39 uses PBKDF2 [90] to derive the seed; as mentioned previously in Chapter 8.10, the use of PBKDF2 is discouraged due to its insufficient security [25]. We will discuss PBKDF2 in Appendix A.2.5. Given the fact that BIP-39 was created in 2013, it is understandable why it was recommended at that time.

Generating the Mnemonic

In order to construct a mnemonic seed phrase, random bits are required. BIP-39 requires 128, 160, 192, 224, or 256 bits of randomness. We let $n \in \{128, 160, 192, 224, 256\}$ denote the number of random bits and $\mathtt{r} \in \{0,1\}^n$ to be the random bit string; note that we have $n = 32k$ for $k \in \{4, 5, 6, 7, 8\}$. Next, we compute the checksum

$$\mathtt{c} \coloneqq \text{SHA-2-256}(r)[:k]; \tag{11.33}$$

that is, hash the random bits and take the first k bits for the checksum.

At this point, we now derive the mnemonic phrase from the $\mathtt{r}\|\mathtt{c}$ bits. We set

$$\mathtt{R} \coloneqq \mathtt{r}\|\mathtt{c}. \tag{11.34}$$

From our previous work, $\mathtt{R}$ is a bit string of length $11 \cdot 3k$. Split $\mathtt{R}$ into $3k$ bit words of 11 bits each. For each bit word, look up the corresponding mnemonic word. The resulting $3k$ words (in that order) is the Mnemonic Seed Phrase. We will use the BIP-39 English word list[6].

From Mnemonic to Seed

Deriving the seed from the mnemonic uses PBKDF2. If a passphrase is not used, the salt is "" (the empty string); otherwise, the passphrase is prepended with "mnemonic" and this is used as the salt. From our discussion in Chapter 8.10, a salt is used to ensure that the same password produces a different hash. If the mnemonic is constructed as previously described, the probability is small that the same two mnemonics would ever be generated. Within PBKDF2, the pseudorandom function is HMAC-SHA-2-512 and the iteration count is 2048. This 512-bit seed can then be used to derive private keys. Note that the method for deriving the seed is *independent* of the method used to construct the mnemonic phrase.

The entropy of the 512-bit seed depends on the entropy of the mnemonic. Even so, using PBKDF2 to derive the seed will help ensure it is challenging to derive the seed without knowing the mnemonic phrase.

11.7.3 Examples of Mnemonic Seed Phrases with BIP-39

We now work through some examples using the BIP-39 standard. *Do not use these mnemonic seed phrases!*

[5] https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki
[6] https://github.com/bitcoin/bips/blob/master/bip-0039/english.txt

Example 11.3 (BIP-39 128-bit Mnemonic Example)
Code may be found at `examples/ecc/bip-39_ex_0.py`.

We begin with an example which is a test vector. In this case, we let

$$\mathtt{r} \coloneqq \mathtt{00000000000000000000000000000000}; \tag{11.35}$$

that is, our randomness is 16 zero bytes. In this case, we find

$$\begin{aligned}&\text{SHA-2-256}(\mathtt{r}) = \\ &\mathtt{374708fff7719dd5979ec875d56cd2286f6d3cf7ec317a3b25632aab28ec37bb}.\end{aligned} \tag{11.36}$$

Thus, the first 4 bits are $\mathtt{3}_{16}$. Thus, we have

$$\mathtt{R} \coloneqq \mathtt{00000000000000000000000000000003}. \tag{11.37}$$

We now need to convert this into our phrase. Because we have 132 bits, this is twelve 11-bit words. All words except the last one is all zeros; the last word is the word $\mathtt{00000000011}_2 = 3$. Looking at the word list, we have the mnemonic

$$\begin{aligned}\text{words} \coloneqq\ &\text{abandon abandon abandon abandon} \\ &\text{abandon abandon abandon abandon} \\ &\text{abandon abandon abandon about.}\end{aligned} \tag{11.38}$$

Example 11.4 (BIP-39 MD5 Mnemonic Example)
Code may be found at `examples/ecc/bip-39_ex_md5.py`.

This example will be more involved. This time, we let

$$\begin{aligned}\mathtt{r} &\coloneqq \text{MD5}(\mathtt{00000000000000000000000000000000}) \\ &= \mathtt{4ae71336e44bf9bf79d2752e234818a5};\end{aligned} \tag{11.39}$$

that is, our randomness is the MD5 hash of 16 zero bytes. In this case, we find

$$\begin{aligned}&\text{SHA-2-256}(\mathtt{r}) = \\ &\mathtt{4455ba3e9c7f3ad867f3b0f0882f0826c21322773af394f6050c36527c6d5c74}.\end{aligned} \tag{11.40}$$

Thus, the first 4 bits are $\mathtt{4}_{16}$. Thus, we have

$$\mathtt{R} \coloneqq \mathtt{4ae71336e44bf9bf79d2752e234818a54}. \tag{11.41}$$

We now split this into 11-bit words:

$$\begin{aligned}\text{bit words} \coloneqq\ &\mathtt{01001010111}\ \ \mathtt{00111000100}\ \ \mathtt{11001101101}\ \ \mathtt{11001000100} \\ &\mathtt{10111111100}\ \ \mathtt{11011111101}\ \ \mathtt{11100111010}\ \ \mathtt{01001110101} \\ &\mathtt{00101110001}\ \ \mathtt{00011010010}\ \ \mathtt{00000110001}\ \ \mathtt{01001010100}.\end{aligned} \tag{11.42}$$

By converting this into integers, we find

$$\begin{aligned} \text{integers} \coloneqq\ & \texttt{599} \quad \texttt{452} \quad \texttt{1645} \quad \texttt{1604} \\ & \texttt{1532} \quad \texttt{1789} \quad \texttt{1850} \quad \texttt{629} \\ & \texttt{369} \quad \texttt{210} \quad \texttt{49} \quad \texttt{596}. \end{aligned} \tag{11.43}$$

This gives us the words

$$\begin{aligned} \text{words} \coloneqq\ & \text{enough decade soap silk} \\ & \text{sauce text trap exchange} \\ & \text{comfort bottom alert enhance}. \end{aligned} \tag{11.44}$$

Example 11.5 (BIP-39 SHA-1 Mnemonic Example)
Code may be found at `examples/ecc/bip-39_ex_sha1.py`.

We proceed as we did with MD5; this time, we use SHA-1:

$$\begin{aligned} \texttt{r} &\coloneqq \text{SHA-1}(\texttt{0000000000000000000000000000000000000000}) \\ &= \texttt{6768033e216468247bd031a0a2d9876d79818f8f}; \end{aligned} \tag{11.45}$$

that is, our randomness is the SHA-1 hash of 20 zero bytes. In this case, we find

$$\begin{aligned} &\text{SHA-2-256}(\texttt{r}) = \\ &\texttt{4c811531c1f2290d58da39cace9720192b0563354b3f9172584682e913c92c75}. \end{aligned} \tag{11.46}$$

Thus, the first 5 bits are $\texttt{01001}_2$. Thus, we have

$$\texttt{R} \coloneqq \texttt{6768033e216468247bd031a0a2d9876d79818f8f}_{16} \| \texttt{01001}_2. \tag{11.47}$$

We now split this into 11-bit words:

$$\begin{aligned} \text{bit words} \coloneqq\ & \texttt{01100111011} \quad \texttt{01000000000} \quad \texttt{11001111100} \quad \texttt{01000010110} \\ & \texttt{01000110100} \quad \texttt{00010010001} \quad \texttt{11101111010} \quad \texttt{00000110001} \\ & \texttt{10100000101} \quad \texttt{00010110110} \quad \texttt{01100001110} \quad \texttt{11011010111} \\ & \texttt{10011000000} \quad \texttt{11000111110} \quad \texttt{00111101001}. \end{aligned} \tag{11.48}$$

By convert these into integers, we find

$$\begin{aligned} \text{integers} \coloneqq\ & \texttt{827} \quad \texttt{512} \quad \texttt{1660} \quad \texttt{534} \\ & \texttt{564} \quad \texttt{145} \quad \texttt{1914} \quad \texttt{49} \\ & \texttt{1285} \quad \texttt{182} \quad \texttt{782} \quad \texttt{1751} \\ & \texttt{1216} \quad \texttt{1598} \quad \texttt{489}. \end{aligned} \tag{11.49}$$

This gives us the words

$$\begin{aligned}\text{words} \coloneqq\ & \text{guess divorce sort drift}\\ & \text{educate banana urban alert}\\ & \text{pass bitter gift sustain}\\ & \text{object sick diamond.}\end{aligned} \tag{11.50}$$

Example 11.6 (BIP-39 SHA-3 Mnemonic Example)
Code may be found at `examples/ecc/bip-39_ex_sha3.py`.

To derive random bits, we use the SHA-3 hash of 32 zero bytes.

$$\begin{aligned}\texttt{r} &\coloneqq \text{SHA-3}(\texttt{0000000000000000000000000000000000000000000000000000000000000000})\\ &= \texttt{9e6291970cb44dd94008c79bcaf9d86f18b4b49ba5b2a04781db7199ed3b9e4e}.\end{aligned} \tag{11.51}$$

In this case, we find

$$\begin{aligned}&\text{SHA-2-256}(\texttt{r}) =\\ &\texttt{75d0e0663934a19f8e2b848f40cb0f446f6dd7c7e4bfd4b8bde87286b3b2f3bb}.\end{aligned} \tag{11.52}$$

Thus, the first 8 bits are $\texttt{75}_{16}$. Thus, we have

$$\texttt{R} \coloneqq \texttt{9e6291970cb44dd94008c79bcaf9d86f18b4b49ba5b2a04781db7199ed3b9e4e75}. \tag{11.53}$$

We now split this into 11-bit words:

$$\begin{aligned}\text{bit words} \coloneqq\ & \texttt{10011110011}\ \ \texttt{00010100100}\ \ \texttt{01100101110}\ \ \texttt{00011001011}\\ & \texttt{01000100110}\ \ \texttt{11101100101}\ \ \texttt{00000000001}\ \ \texttt{00011000111}\\ & \texttt{10011011110}\ \ \texttt{01010111110}\ \ \texttt{01110110000}\ \ \texttt{11011110001}\\ & \texttt{10001011010}\ \ \texttt{01011010010}\ \ \texttt{01101110100}\ \ \texttt{10110110010}\\ & \texttt{10100000010}\ \ \texttt{00111100000}\ \ \texttt{01110110110}\ \ \texttt{11100011001}\\ & \texttt{10011110110}\ \ \texttt{10011101110}\ \ \texttt{01111001001}\ \ \texttt{11001110101}.\end{aligned} \tag{11.54}$$

By converting these into integers, we find

$$\begin{aligned}\text{integers} \coloneqq\ & \texttt{1267}\ \ \texttt{164}\ \ \texttt{814}\ \ \texttt{203}\\ & \texttt{550}\ \ \texttt{1893}\ \ \texttt{1}\ \ \texttt{199}\\ & \texttt{1246}\ \ \texttt{702}\ \ \texttt{944}\ \ \texttt{1777}\\ & \texttt{1114}\ \ \texttt{722}\ \ \texttt{884}\ \ \texttt{1458}\\ & \texttt{1282}\ \ \texttt{480}\ \ \texttt{950}\ \ \texttt{1817}\\ & \texttt{1270}\ \ \texttt{1262}\ \ \texttt{969}\ \ \texttt{1653}.\end{aligned} \tag{11.55}$$

This gives us the words

$$\begin{aligned} \text{words} \coloneqq\ & \text{oyster behind grape bonus} \\ & \text{dynamic uncover ability body} \\ & \text{orange fit invite taste} \\ & \text{mercy fog hub rent} \\ & \text{park despair item tobacco} \\ & \text{paddle oven junior solid.} \end{aligned} \tag{11.56}$$

Chapter 12

Pairing-Based Cryptography

Ethereum [135] has a bilinear pairing included in its precompiled contracts with the ability to perform a pairing check, elliptic curve addition, and elliptic curve point multiplication. This makes it possible to compute BLS signatures [29]. In this chapter we will talk about BLS signatures and what else can be accomplished with a bilinear pairing. All of these operations may be performed within Solidity smart contracts. On the Ethereum blockchain, smart contracts are programs which may automatically perform actions in response to changes in state. As a brief aside, we begin by discussing cryptography in Ethereum in Chapter 12.1.

Throughout this chapter, we are assuming the bilinear pairing is over elliptic curves.

12.1 Cryptography in Ethereum

We take a brief digression to spend some time talking about Ethereum [135]. The main focus here will be on the cryptographic protocols available within Ethereum.

Ethereum is a blockchain that allows for smart contracts. A *smart contract* is a program which is able to perform operations in response to inputs. These operations include the ability to perform signature verification and other cryptographic operations such as hashing. Other arithmetical operations are available as well.

In particular, Ethereum has the ability to cheaply perform certain operations involving Pairing-Based Cryptography. One available type of bilinear pairing is based on a Barreto–Naehrig Curve [9]. In particular, let $e : G_1 \times G_2 \to G_T$ be the bilinear pairing. We can cheaply perform addition and scalar multiplication in G_1. We also have the ability to perform a pairing check. Given $a_i \in G_1$ and $b_i \in G_2$ for $i \in \{1, \cdots, \ell\}$, we define

$$\textsc{PairingCheck}\left(a_1, b_1, a_2, b_2, \cdots, a_\ell, b_\ell\right) := \begin{cases} \texttt{true} & \text{if } 1 = \prod_{k=1}^{\ell} e(a_k, b_k) \\ \texttt{false} & \text{otherwise} \end{cases}. \tag{12.1}$$

When we restrict ourselves to the case when $\ell = 2$, we see that

$$\textsc{PairingCheck}\left(a_1, b_1, a_2, b_2\right) = \texttt{true} \tag{12.2}$$

when

$$e(a_1, b_1) \cdot e(a_2, b_2) = 1 \tag{12.3}$$

or

$$e(a_1, b_1^{-1}) = e(a_2, b_2). \tag{12.4}$$

This is the form that will be used when performing BLS signature validation; we discuss BLS signatures in Chapter 12.2.

At this time, no inexpensive operations in G_2 are available in Ethereum [135]. Thus, in order to ensure valid information in G_2, we will need to submit the corresponding information in G_1 and then validate it using a PairingCheck.

12.2 BLS Signatures

We previously talked about digital signatures in Chapters 10 and 11. We now talk about digital signatures which arise from bilinear pairings and discuss their importance. This builds on our discussion of bilinear pairings from Chapter 5.3.

Let us suppose $e : G_1 \times G_2 \to G_T$ is our bilinear pairing with $|G_1| = |G_2| = |G_T| = q$ for q prime. Let $g_2 \in G_2$ be a generator. Let $x \xleftarrow{\$} \mathbb{Z}_q$ be the private key and g_2^x be the public key. Let us suppose that $H : \{0,1\}^* \to G_1$ is a hash function. In this setting, H is a hash-to-curve function, as we map arbitrary bit strings to elements of G_1. Given a message $m \in \{0,1\}^*$, we have the BLS signature

$$\sigma := [H(m)]^x . \tag{12.5}$$

To verify this signature, we check that

$$e(\sigma, g_2) \stackrel{?}{=} e(H(m), g_2^x) . \tag{12.6}$$

This is equivalent to

$$e\left(\sigma, g_2^{-1}\right) \cdot e(H(m), g_2^x) \stackrel{?}{=} 1. \tag{12.7}$$

This can be computed by performing a PairingCheck:

$$\text{PairingCheck}(\sigma, g_2^{-1}, H(m), g_2^x) \stackrel{?}{=} \texttt{true}. \tag{12.8}$$

As mentioned previously, this operation may be performed cheaply on Ethereum [135].

This signature scheme was originally referred to as a *short signature* due to the size of σ; see the original publication [29]. When using the bilinear pairing on Ethereum, BLS signatures are 64 bytes uncompressed and 33 bytes compressed; this is smaller than any of the other signatures mentioned in Chapter 10.7. The public keys are also 128 bytes uncompressed.

12.3 Hash-to-Curve Functions

For BLS signatures, we require a method of hashing arbitrary messages into cyclic groups. In the previous section, we *assumed* such functions exist. Now we describe how they are formed. We want to map into an elliptic curve $E/\mathbb{F}_p$. Furthermore, we will assume throughout that we have access to a standard hash function $H : \{0,1\}^* \to \{0,1\}^b$. In practice, H will likely be SHA-2, SHA-3, or Keccak.

In this section, we switch from multiplicative notation to additive notation for groups.

12.3.1 One Insecure Hash-to-Curve Function

Constructing a hash-to-curve function is nontrivial; we now present a method that should *never* be used because it breaks security assumptions.

Suppose we have a public key $x \cdot P$ and standard hash function $H : \{0,1\}^* \to \{0,1\}^b$. There are some who may want to use the hash-to-curve function

$$H_{2C}(m) \coloneqq H(m) \cdot P. \tag{12.9}$$

In this way, for Alice with public key $a \cdot P$, a signature would be

$$\begin{aligned} \sigma &\coloneqq a \cdot (H(m) \cdot P) \\ &= (a \cdot H(m)) \cdot P. \end{aligned} \tag{12.10}$$

This is *horrible* and completely breaks all security, because we do not even need to know the private key to construct this signature: all that is needed is the public key $x \cdot P$:

$$\begin{aligned} H(m) \cdot (x \cdot P) &= (H(m) \cdot x) \cdot P \\ &= (x \cdot H(m)) \cdot P. \end{aligned} \tag{12.11}$$

This is a clear example of why care is required when designing cryptographic protocols. Using this method, it is possible for forge arbitrary signatures for any public key.

12.3.2 Nondeterministic Hash-to-Curve Algorithm

The first hash-to-curve algorithms used a nondeterministic method; it was called MapToGroup in [29]. The method is simple in practice, but deterministic methods are preferred because the number of computational operations may be bounded. We will not discuss nondeterministic methods any further.

12.3.3 Initial Hash-to-Curve Discussion

There does not appear to be a good way to map an arbitrary bit string into a group. To simplify the matter, we use a two-step process: first, we hash arbitrary bit strings to a finite

field; next, map from the finite field into the elliptic curve. This two-step process is now standard; see [33, 52, 129]. In the end, we have a hash function $H : \{0,1\}^* \to G$ with

$$H(m) = f(H_1(m)) + f(H_2(m)). \tag{12.12}$$

In this case, $H_i : \{0,1\}^* \to \mathbb{F}$ is a hash function to a finite field while $f : \mathbb{F} \to G$ is a deterministic mapping from a finite field $\mathbb{F}$ to the elliptic curve G. Furthermore, H_1 and H_2 are independent hash functions.

12.3.4 Hashing to Finite Fields

We first focus on mapping deterministically into $\mathbb{F}_p$. This is an easy task. The main challenge will be to control the nonuniformity.

Suppose we have a random oracle $\widetilde{H} : \{0,1\}^* \to \mathbb{Z}_N$. We are interested in knowing how much $H(m) \mod p$ deviates from the uniform distribution. It turns out that the deviations are small provided N is large enough. In particular, if p is an n-bit prime number and we want k bits of security, then we need $N \geq 2^{n+k}$. By choosing $N = 2^m$ for some m, we can instantiate the random oracle $\widetilde{H}$ using a standard hash function. The details are worked out in Appendix A.2.6.

Overall, it is relatively easy ensure the nonuniformity is negligible. This describes the hash functions H_1 and H_2 in Eq. (12.12).

12.3.5 Deterministic Mapping from Finite Field to Elliptic Curve

Deterministically mapping from a finite field to an elliptic curve is much more challenging. The exact method strongly depends on the specific type of elliptic curve. The good news is that there are generic mappings for a wide class of elliptic curves; see [33, 52, 129]. This deterministic mapping is f in Eq. (12.12).

12.3.6 Ensuring Hash-to-Curve Uniformity

At this point, we can deterministically map arbitrary data into an elliptic curve. Hash function values should be uniformly distributed across the entire range, though.

The previous work does not ensure this property. It may be the case that the deterministic mapping only maps into a portion of the elliptic curve; in this way, not every point on the elliptic curve may be reached. This results in a nonuniform hash function. Thus, ensuring uniformity requires additional care. The details are worked on in in [33, 52, 129]. In the cases from the papers previously cited, it turns out that all that is required are 2 independent realizations of different finite field points mapping to the elliptic curve. This is why we require H_1 and H_2 to be independent hash functions and then add their deterministic outputs in Eq. (12.12).

12.4 Multi-Signatures

As previously discussed in Chapter 12.2, BLS signatures allow for efficient signature generation and validation. We will now focus on n-out-of-n multi-signature schemes. That is, n participants want to sign the same message, and we want to prove that all of them signed it. Threshold signature schemes are more involved and will be discussed in Chapter 14.6 after working through secret sharing and distributed key generation protocols. In a threshold signature scheme, only a *subset* of participants are required to sign a message for it to be valid for the entire group.

Let us suppose that Alice and Bob wish to sign a message together. As in Chapter 12.2, we are assuming that we are working with BLS signatures with a bilinear pairing e and hash-to-curve function H. We suppose that Alice has private key x_A and public key $A = g_2^{x_A}$; Bob has private and public keys x_B and $B = g_2^{x_B}$.

Let us suppose that Alice and Bob both sign message $m \in \{0,1\}^*$ with signatures σ_A and σ_B. In this case, we know that

$$\begin{aligned} e\left(\sigma_A, g_2\right) &= e\left(H(m), g_2^{x_A}\right) \\ e\left(\sigma_B, g_2\right) &= e\left(H(m), g_2^{x_B}\right). \end{aligned} \tag{12.13}$$

Suppose we set

$$\sigma = \sigma_A \sigma_B. \tag{12.14}$$

In this case, we know

$$\begin{aligned} e\left(\sigma, g_2\right) &= e\left(\sigma_A \sigma_B, g_2\right) \\ &= e\left(\sigma_A, g_2\right) \cdot e\left(\sigma_B, g_2\right) \\ &= e\left(H(m), g_2^{x_A}\right) \cdot e\left(H(m), g_2^{x_B}\right) \\ &= e\left(H(m), g_2\right)^{x_A} \cdot e\left(H(m), g_2\right)^{x_B} \\ &= e\left(H(m), g_2\right)^{x_A + x_B} \\ &= e\left(H(m), g_2^{x_A + x_B}\right) \\ &= e\left(H(m), A \cdot B\right). \end{aligned} \tag{12.15}$$

Thus, in order to prove that both Alice and Bob signed the message m, we only need to combine the signatures and the public keys. This reduces the cost from 2 signature verifications to 1 signature verification.

The same property holds in general. In particular, suppose we have private keys $\{x_k\}_{k=1}^n$ and public keys $\{g_2^{x_k}\}_{k=1}^n$. If each participant signs the same message m producing signatures $\{\sigma_k\}_{k=1}^n$, we set

$$\begin{aligned}\sigma &= \prod_{k=1}^{n} \sigma_k \\ X &= \prod_{k=1}^{n} g_2^{x_k}.\end{aligned} \tag{12.16}$$

We then see

$$\begin{aligned}e(\sigma, g_2) &= e\left(\prod_{k=1}^{n} \sigma_k, g_2\right) \\ &= \prod_{k=1}^{n} e(\sigma_k, g_2) \\ &= \prod_{k=1}^{n} e(H(m), g_2^{x_k}) \\ &= \prod_{k=1}^{n} e(H(m), g_2)^{x_k} \\ &= e(H(m), g_2)^{\sum_{k=1}^{n} x_k} \\ &= e\left(H(m), g_2^{\sum_{k=1}^{n} x_k}\right) \\ &= e(H(m), X).\end{aligned} \tag{12.17}$$

As we can see, the n-out-of-n case is the same as the 2-out-of-2 case with Alice and Bob. In the n-out-of-n case, we reduce the computation from n signature verifications to 1 signature verification at the cost of aggregating n signatures and public keys. Because signature verification is expensive, this is generally worthwhile.

12.5 Concluding Discussion

We have only scratched the surface of Pairing-Based Cryptography. Additional material may be found in [47].

Chapter 13

Zero-Knowledge Proofs

We now give a brief introduction to zero-knowledge proofs.

13.1 The Need for Zero-Knowledge Proofs

The purpose of zero-knowledge proofs is to *prove* something is true without revealing *any additional information*.

We note that all digital signatures are a form of zero-knowledge proof. As discussed in Chapter 10, If Alice signs a message m resulting in signature σ and sends (m, σ) to Bob, he is able to prove to Charlie that Alice did send him the message m. In this case, Charlie sees the message along with the signature and knows that Alice signed the message and sent the pair to Bob, but he does not learn any additional information; in particular, he does not gain the ability to forge signatures of Alice. Zero-knowledge proofs attempt to generalize this to proofs about other knowledge.

13.2 Group for Examples

Throughout this chapter we will be using a particular finite cyclic group for the examples. We have the following:

$$\begin{aligned} p &= rq + 1 \\ q &= 2^{32} + 15 \\ r &= 12. \end{aligned} \tag{13.1}$$

It can be shown that p and q are prime numbers. Additionally, it can also be shown that

$$h = 7 \tag{13.2}$$

is a generator for $\mathbb{F}_p^*$. Thus, we set

$$g = h^r \mod p. \tag{13.3}$$

From this, we know that $|\langle g \rangle| = q$.

When we need two generators, we set

$$\begin{aligned} h_1 &= 7 \\ h_2 &= 6 \end{aligned} \tag{13.4}$$

and

$$\begin{aligned} g_1 &= h_1^r \mod p \\ g_2 &= h_2^r \mod p. \end{aligned} \tag{13.5}$$

It can also be shown that 6 is also a generator of $\mathbb{F}_p^*$; see Appendix B.4.1 for more information.

13.3 Proving knowledge of Discrete Logarithms

We now work through zero-knowledge proofs involving discrete logarithms; this section uses examples found in [38].

Throughout we suppose $G = \langle g \rangle$ is a finite cyclic group of order q. Furthermore, we suppose that we have a hash function $H : \{0,1\}^* \to \{0,1\}^b$.

13.3.1 Knowledge of Discrete Logarithm

Definition

We let $x \xleftarrow{\$} \mathbb{Z}_q^*$ and set $y = g^x$. To prove knowledge of x,

1. We let $v \xleftarrow{\$} \mathbb{Z}_q$ and set $t = g^v$; t is the *commitment*.
2. We have the *challenge*

$$c = H(g, y, t). \tag{13.6}$$

3. We have the *response*

$$r = v - cx \mod q. \tag{13.7}$$

The challenge-response pair (c, r) can be verified by anyone. The proof involves constructing the commitment

$$t' = g^r y^c \tag{13.8}$$

and verifying

$$c = H(g, y, t'). \tag{13.9}$$

As noted in [38], this is essentially a Schnorr signature of the message (g, y). Schnorr signatures are discussed in Chapter 10.4.

Verification

If (c, r) is as defined, then we see

$$\begin{aligned} t' &= g^r y^c \\ &= g^{v-cx} \cdot (g^x)^c \\ &= g^v \cdot g^{-cx} \cdot g^{xc} \\ &= g^v \\ &= t. \end{aligned} \tag{13.10}$$

Thus, we have

$$\begin{aligned} H(g, y, t') &= H(g, y, t) \\ &= c. \end{aligned} \tag{13.11}$$

Example

Example 13.1 (Knowledge of Discrete Logarithm)
Code may be found at `examples/zk_proofs/dl_proof_1.py`.

We use the cyclic group described in Chapter 13.2; see Listing 13.1.

Secret Information In this case, we let

$$\begin{aligned} g &= 13841287201 \\ y &= 34063925739. \end{aligned} \tag{13.12}$$

We have

$$y = g^x \mod p \tag{13.13}$$

with

$$x = 710124910. \tag{13.14}$$

Setup Challenge-Response To make our challenge-response, we set

$$\begin{aligned} v &= 979360098 \\ t &= g^v \mod p \\ &= 26735074229. \end{aligned} \tag{13.15}$$

This allows us to compute

$$\begin{aligned} c &= \text{MD5}\,(g, y, t) \mod q \\ &= 2451489814 \\ r &= v - cx \mod q \\ &= 3624355949. \end{aligned} \tag{13.16}$$

This implies that $(2451489814, 3624355949)$ is the challenge-response for knowledge of the discrete logarithm for $(13841287201, 34063925739)$.

Validate Challenge-Response To validate the zero-knowledge proof, we see

$$\begin{aligned} t' &= g^r y^c \mod p \\ &= 26735074229 \\ c' &= \text{MD5}\,(g, y, t') \mod q \\ &= 2451489814. \end{aligned} \tag{13.17}$$

The proof is valid because $c = c'$.

Intuition

We take some time to try to explain what is happening intuitively.

We are building a challenge-response test where the challenge is a pseudorandom number (from a hash function). For an arbitrary (c', r') pair, we have

$$\begin{aligned} t' &= g^{r'} y^{c'} \\ &= g^{r' + c'x}. \end{aligned} \tag{13.18}$$

If x is unknown, then this will appear to be a random element of G. Given a hash function (with a sufficiently large output), it should be computational infeasible to ensure

$$c' = H(g, y, t') \tag{13.19}$$

unless x is known.

13.3.2 Knowledge of Two Discrete Logarithms

Definition

In this case, we assume g_1 and g_2 are independent generators of G. For $x \xleftarrow{\$} \mathbb{Z}_q^*$, we set $y_1 = g_1^x$ and $y_2 = g_2^x$. We wish to prove that x is known.

We proceed in a manner similar as before:

1. We let $v \xleftarrow{\$} \mathbb{Z}_q$ and set $t_1 = g_1^v$ and $t_2 = g_2^v$.

Listing 13.1: Zero-knowledge proof of knowledge of discrete logarithm

```
#!/usr/bin/env python3
import hashlib
num_bytes = 5
r = 12; q = 2**32 + 15; p = r*q + 1
h = 7;  g = pow(h, r, p)
# g: 13841287201

xPreStr = '00'*16; xPre = bytes.fromhex(xPreStr)
md5 = hashlib.md5(); md5.update(xPre)
x = int(md5.hexdigest(), 16) % q
y = pow(g, x, p)
# y: 34063925739

vPreStr = '11'*16; vPre = bytes.fromhex(vPreStr)
md5 = hashlib.md5(); md5.update(vPre)
v = int(md5.hexdigest(), 16) % q
t = pow(g, v, p)

t_bytes = t.to_bytes(num_bytes, 'big')
g_bytes = g.to_bytes(num_bytes, 'big')
y_bytes = y.to_bytes(num_bytes, 'big')

md5 = hashlib.md5()
md5.update(g_bytes); md5.update(y_bytes); md5.update(t_bytes)
c = int(md5.hexdigest(), 16) % q # c: 2451489814
r = (v - c*x) % q                # r: 3624355949

tPrime = (pow(g, r, p) * pow(y, c, p)) % p
tPrime_bytes = tPrime.to_bytes(num_bytes, 'big')
md5 = hashlib.md5()
md5.update(g_bytes); md5.update(y_bytes); md5.update(tPrime_bytes)
cPrime = int(md5.hexdigest(), 16) % q
assert c == cPrime # Valid
```

2. We have the *challenge*

$$c = H(g_1, y_1, g_2, y_2, t_1, t_2). \tag{13.20}$$

3. We have the *response*

$$r = v - cx \mod q. \tag{13.21}$$

The challenge-response pair (c, r) can be verified by anyone. The proof involves constructing the commitment

$$\begin{aligned} t_1' &= g_1^r y_1^c \\ t_2' &= g_2^r y_2^c \end{aligned} \tag{13.22}$$

and confirming

$$c = H(g_1, y_1, g_2, y_2, t_1', t_2'). \tag{13.23}$$

Verification

If (c, r) is as defined, then we see

$$\begin{aligned} t_i' &= g_i^r y_i^c \\ &= g_i^{v-cx} \cdot (g_i^x)^c \\ &= g_i^v \cdot g_i^{-cx} \cdot g_i^{xc} \\ &= g_i^v \\ &= t_i. \end{aligned} \tag{13.24}$$

Thus, we have

$$\begin{aligned} H(g_1, y_1, g_2, y_2, t_1', t_2') &= H(g_1, y_1, g_2, y_2, t_1, t_2) \\ &= c. \end{aligned} \tag{13.25}$$

Example

Example 13.2 (Knowledge of Two Discrete Logarithms)
Code may be found at `examples/zk_proofs/dl_proof_2.py`.

We use the cyclic group described in Chapter 13.2; see Listing 13.2.

Secret Information In this case, we let

$$\begin{aligned} g_1 &= 13841287201 \\ g_2 &= 2176782336 \\ y_1 &= 34063925739 \\ y_2 &= 42077908773. \end{aligned} \tag{13.26}$$

We have

$$\begin{aligned} y_1 &= g_1^x \\ y_2 &= g_2^x \end{aligned} \tag{13.27}$$

with

$$x = 710124910. \tag{13.28}$$

Setup Challenge-Response To make our challenge-response, we set

$$\begin{aligned} v &= 979360098 \\ t_1 &= g_1^v \mod p \\ &= 26735074229 \\ t_2 &= g_2^v \mod p \\ &= 14645668233. \end{aligned} \tag{13.29}$$

This allows us to compute

$$\begin{aligned} c &= \text{MD5}\left(g_1, y_1, g_2, y_2, t_1, t_2\right) \mod q \\ &= 820804763 \\ r &= v - cx \mod q \\ &= 1837574845. \end{aligned} \tag{13.30}$$

This implies that $(820804763, 1837574845)$ is the challenge-response for knowledge of the two discrete logarithms $(13841287201, 34063925739)$ and $(2176782336, 42077908773)$.

Validate Challenge-Response To validate the zero-knowledge proof, we see

$$\begin{aligned} t_1' &= g_1^r y_1^c \mod p \\ &= 26735074229 \\ t_2' &= g_2^r y_2^c \mod p \\ &= 14645668233 \\ c' &= \text{MD5}\left(g_1, y_1, g_2, y_2, t_1', t_2'\right) \mod q \\ &= 820804763. \end{aligned} \tag{13.31}$$

The proof is valid because $c = c'$.

13.3.3 Knowledge of Multiple Discrete Logarithms

The situation for two values can be generalized to multiple values.

13.3.4 Knowledge of Linear Combination of Two Discrete Logarithms

Definition

We now suppose that g_1 and g_2 are distinct generators of G and $x_1, x_2 \xleftarrow{\$} \mathbb{Z}_q^*$. We want to prove that

$$\begin{aligned} y_1 &= g_1^{x_1} \\ y_2 &= g_2^{x_2} \\ a_1 x_1 + a_2 x_2 &= b \mod q \end{aligned} \tag{13.32}$$

for specified $(a_1, a_2) \in \mathbb{Z}_q^2$. That is, we know the discrete logarithms and a linear combination of them. This is more involved than the previous proofs.

1. Let

$$(v_1, v_2) \xleftarrow{\$} \left\{(u_1, u_2) \in \mathbb{Z}_q^2 \mid a_1 u_1 + a_2 u_2 = 0 \mod q\right\}. \tag{13.33}$$

   We have the commitments

$$\begin{aligned} t_1 &= g_1^{v_1} \\ t_2 &= g_2^{v_2}. \end{aligned} \tag{13.34}$$

2. We have the challenge

$$c = H(g_1, y_1, g_2, y_2, a_1, a_2, b, t_1, t_2). \tag{13.35}$$

3. We have the responses

$$\begin{aligned} r_1 &= v_1 - c x_1 \mod q \\ r_2 &= v_2 - c x_2 \mod q. \end{aligned} \tag{13.36}$$

Listing 13.2: Zero-knowledge proof of knowledge of two discrete logarithms

```
#!/usr/bin/env python3
import hashlib
num_bytes = 5
r = 12; q = 2**32 + 15; p = r*q + 1
h1 = 7; g1 = pow(h1, r, p) # g1: 13841287201
h2 = 6; g2 = pow(h2, r, p) # g2:  2176782336

xPreStr = '00'*16; xPre = bytes.fromhex(xPreStr)
md5 = hashlib.md5(); md5.update(xPre)
x = int(md5.hexdigest(), 16) % q
y1 = pow(g1, x, p); y2 = pow(g2, x, p)
# y1: 34063925739; y2: 42077908773

vPreStr = '11'*16; vPre = bytes.fromhex(vPreStr)
md5 = hashlib.md5(); md5.update(vPre)
v = int(md5.hexdigest(), 16) % q;
t1 = pow(g1, v, p); t2 = pow(g2, v, p)

t1_bytes = t1.to_bytes(num_bytes, 'big')
t2_bytes = t2.to_bytes(num_bytes, 'big')
g1_bytes = g1.to_bytes(num_bytes, 'big')
g2_bytes = g2.to_bytes(num_bytes, 'big')
y1_bytes = y1.to_bytes(num_bytes, 'big')
y2_bytes = y2.to_bytes(num_bytes, 'big')

md5 = hashlib.md5()
md5.update(g1_bytes); md5.update(y1_bytes); md5.update(g2_bytes)
md5.update(y2_bytes); md5.update(t1_bytes); md5.update(t2_bytes)
c = int(md5.hexdigest(), 16) % q # c:  820804763
r = (v - c*x) % q                # r: 1837574845

t1Prime = (pow(g1, r, p) * pow(y1, c, p)) % p
t2Prime = (pow(g2, r, p) * pow(y2, c, p)) % p
t1Prime_bytes = t1Prime.to_bytes(num_bytes, 'big')
t2Prime_bytes = t2Prime.to_bytes(num_bytes, 'big')
md5 = hashlib.md5()
md5.update(g1_bytes);       md5.update(y1_bytes)
md5.update(g2_bytes);       md5.update(y2_bytes)
md5.update(t1Prime_bytes); md5.update(t2Prime_bytes)
cPrime = int(md5.hexdigest(), 16) % q
assert c == cPrime # Valid
```

The proof is (c, r_1, r_2).

The verify the proof, we set

$$\begin{aligned} t'_1 &= g_1^{r_1} y_1^c \\ t'_2 &= g_2^{r_2} y_2^c. \end{aligned} \tag{13.37}$$

The proof is valid if

$$c = H(g_1, y_1, g_2, y_2, a_1, a_2, b, t'_1, t'_2) \tag{13.38}$$

and

$$a_1 r_1 + a_2 r_2 = -cb \mod q. \tag{13.39}$$

Verification

We now show that if (c, r_1, r_2) is as defined then the proof is valid.

First, we see

$$\begin{aligned} t'_i &= g_i^{r_i} y_i^c \\ &= g_i^{v_i - cx_i} \cdot (g_i^{x_i})^c \\ &= g_i^{v_i} \\ &= t_i. \end{aligned} \tag{13.40}$$

Thus, the commitments are valid, so we have that

$$\begin{aligned} H(g_1, y_1, g_2, y_2, a_1, a_2, b, t'_1, t'_2). &= H(g_1, y_1, g_2, y_2, a_1, a_2, b, t_1, t_2) \\ &= c. \end{aligned} \tag{13.41}$$

We next see that

$$\begin{aligned} a_1 r_1 + a_2 r_2 &= a_1 (v_1 - cx_1) + a_2 (v_2 - cx_2) \mod q \\ &= (a_1 v_1 + a_2 v_2) - c (a_1 x_1 + a_2 x_2) \mod q \\ &= -c (a_1 x_1 + a_2 x_2) \mod q \\ &= -cb \mod q. \end{aligned} \tag{13.42}$$

Between the first and second lines, we rearrange terms. The first term from the second line is zero from the assumptions on (v_1, v_2). The fourth line follows from the third by assumption as well.

Example

Example 13.3 (Knowledge of Linear Combination of Two Discrete Logarithms)
Code may be found at `examples/zk_proofs/dl_proof_linear.py`.

We use the cyclic group described in Chapter 13.2; see Listing 13.3. We note that some of the output is missing due to its length.

Secret Information In this case, we let

$$\begin{aligned} g_1 &= 13841287201 \\ g_2 &= 2176782336 \\ y_1 &= 34063925739 \\ y_2 &= 6808289188 \end{aligned} \tag{13.43}$$

with

$$\begin{aligned} y_1 &= g_1^{x_1} \\ y_2 &= g_2^{x_2} \end{aligned} \tag{13.44}$$

for

$$\begin{aligned} x_1 &= 710124910 \\ x_2 &= 285758093. \end{aligned} \tag{13.45}$$

We have

$$a_1 x_1 + a_2 x_2 = b \tag{13.46}$$

with

$$\begin{aligned} a_1 &= 521 \\ a_2 &= 523 \\ b &= 4030483429. \end{aligned} \tag{13.47}$$

Setup Challenge-Response To make our challenge-response, we set

$$\begin{aligned} v_1 &= 979360098 \\ v_2 &= 2333891425 \\ t_1 &= g_1^{v_1} \mod p \\ &= 26735074229 \\ t_2 &= g_2^{v_2} \mod p \\ &= 47139203830. \end{aligned} \tag{13.48}$$

This allows us to compute

$$\begin{aligned} c &= \text{MD5}\,(g_1, y_1, g_2, y_2, a_1, a_2, b, t_1, t_2) \mod q \\ &= 1236818951 \\ r_1 &= v_1 - cx_1 \mod q \\ &= 1952959998 \\ r_2 &= v_2 - cx_2 \mod q \\ &= 3581019185. \end{aligned} \tag{13.49}$$

This implies that $(1236818951, 1952959998, 3581019185)$ is the challenge-response for knowledge of the linear combination of discrete logarithms.

Validate Challenge-Response To validate the zero-knowledge proof, we see

$$\begin{aligned} t_1' &= g_1^{r_1} y_1^c \mod p \\ &= 26735074229 \\ t_2' &= g_2^{r_2} y_2^c \mod p \\ &= 47139203830 \\ c' &= \text{MD5}\,(g_1, y_1, g_2, y_2, a_1, a_2, b, t_1', t_2') \mod q \\ &= 1236818951. \end{aligned} \tag{13.50}$$

We see that $c = c'$ and that

$$\begin{aligned} a_1 r_1 + a_2 r_2 &\mod q = 4147159721 \\ -cb &\mod q = 4147159721. \end{aligned} \tag{13.51}$$

Thus, we have

$$a_1 r_1 + a_2 r_2 = -cb \mod q. \tag{13.52}$$

The proof is valid because $c = c'$ and the correct linear combination.

13.3.5 Knowledge of Linear Combination of Multiple Discrete Logarithms

We can easily generalize the results from the previous section to a linear combination of n discrete logarithm values.

Listing 13.3: Zero knowledge proof of knowledge of linear combination of two discrete logarithms

```
#!/usr/bin/env python3
import hashlib
num_bytes = 5
r = 12; q = 2**32 + 15; p = r*q + 1
h1 = 7; g1 = pow(h1, r, p) # g1: 13841287201
h2 = 6; g2 = pow(h2, r, p) # g2:  2176782336

x1PreStr = '00'*16; x1Pre = bytes.fromhex(x1PreStr)
md5 = hashlib.md5(); md5.update(x1Pre)
x1 = int(md5.hexdigest(), 16) % q
x2PreStr = 'ff'*16; x2Pre = bytes.fromhex(x2PreStr)
md5 = hashlib.md5(); md5.update(x2Pre)       # y1:  34063925739
x2 = int(md5.hexdigest(), 16) % q            # y2:   6808289188
y1 = pow(g1, x1, p); y2 = pow(g2, x2, p)     # a1: 521; a2: 523
a1 = 521; a2 = 523; b = (a1*x1 + a2*x2) % q # b:    4030483429
a2Inv = pow(a2, q-2, q); assert (a2Inv * a2) % q == 1
v1 = v1Bar
v2 = (v2Bar - a2Inv*alpha) % q

t1 = pow(g1, v1, p); t2 = pow(g2, v2, p)

md5 = hashlib.md5()
md5.update(g1_bytes); md5.update(y1_bytes); md5.update(g2_bytes)
md5.update(y2_bytes); md5.update(a1_bytes); md5.update(a2_bytes)
md5.update(b_bytes);  md5.update(t1_bytes); md5.update(t2_bytes)
c  = int(md5.hexdigest(), 16) % q # c:  1236818951
r1 = (v1 - c*x1) % q              # r1: 1952959998
r2 = (v2 - c*x2) % q              # r2: 3581019185

t1Prime = (pow(g1, r1, p) * pow(y1, c, p)) % p
t2Prime = (pow(g2, r2, p) * pow(y2, c, p)) % p
md5 = hashlib.md5()
md5.update(g1_bytes); md5.update(y1_bytes); md5.update(g2_bytes)
md5.update(y2_bytes); md5.update(a1_bytes); md5.update(a2_bytes)
md5.update(b_bytes);
md5.update(t1Prime_bytes); md5.update(t2Prime_bytes)
cPrime = int(md5.hexdigest(), 16) % q
assert c == cPrime
assert ((a1*r1 + a2*r2) % q) == ((-c*b) % q) # Valid
```

13.4 Concluding Discussion

We walked through some basic examples of zero-knowledge proofs by focusing on proving knowledge of discrete logarithms. There is much more which could be said on this topic but we will not at this time. See [38] for more information and a more thorough discussion.

We note some extensions of zero-knowledge proofs include Non-Interactive Zero-Knowledge Proofs (NIZKs) [105, 112], Zero-Knowledge Succinct Non-interactive ARguments of Knowledge (zk-SNARKs) [24], and Zero-Knowledge Scalable Transparent ARguments of Knowledge (zk-STARKs) [3, 11]. The interest in zero-knowledge proofs may be related to interest in Ethereum [135] and related cryptocurrencies.

Chapter 14

Secret Sharing Protocols

We now consider the situation where individuals wish to share secret information. Perhaps a collection of individuals wish to split a private key between multiple parties; this way, a specified threshold will need to work together to use the private key.

If the reader is not familiar with Lagrange Interpolation, it would be a good idea for him to review the material in Chapter 5.4, as it will be used extensively in this chapter. Furthermore, the reader is assumed to also be familiar with bilinear pairings and BLS signatures as discussed in Chapter 12.

Notation Convention: Throughout this chapter we will be looking at (t, n)-threshold schemes. By this we mean that $t + 1$ participants are required to work together in order to learn the secret and that t individuals working together gain no additional information. We point this out because there are conflicting conventions throughout the literature.

14.1 The Need for Secret Sharing Protocols

Suppose that Alice, Bob, and Charlie start a business. Being equals, they want a majority to be able to withdraw money for business expenses at any time. At the same time, they do not fully trust each other. They would like to split the private key between the three of them so that 2 out of 3 participants will be required to open the vault. Alice, Bob, and Charlie will need to use a form of secret sharing to share their private vault key. The challenge is that Alice should know *nothing* about the private key without assistance from either Bob or Charlie.

We will start by discussing the original secret sharing protocol (Shamir's Secret Sharing) [118] in Chapter 14.2. From here, we will allow for the possibility of malicious actors in Chapter 14.3. Finally, we discuss the solution to Alice, Bob, and Charlie's problem when we give an overview of distributed key generation (DKG) in Chapter 14.4; the DKG protocol is discussed in Chapter 14.5. As a follow up to DKG, we discuss how to make group signatures in Chapter 14.6.

14.2 Shamir's Secret Sharing

14.2.1 Setup

We now discuss Shamir's Secret Sharing protocol [118].

We want a (t, n)-sharing protocol, where n is the total number of participants and we require $t + 1$ secret shares to recover the original secret. To that end, let P_i be the ith participant and let $\mathcal{P} = \{P_i\}_{i=1}^{n}$ be the set of all participants. *Note well:* this protocol requires *trusted setup*; that is, whoever is producing the secret shares must be honest.

To do this, let $q \in \mathbb{N}$ be a prime sufficiently large. Also, let $s \xleftarrow{\$} \mathbb{F}_q$ be the secret that we wish to distribute. Additionally, we choose random coefficients $c_1, c_2, \cdots, c_t \xleftarrow{\$} \mathbb{F}_q$; we set $c_0 = s$. At this point, we have the secret polynomial

$$p(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_t x^t. \tag{14.1}$$

We use this secret polynomial to derive the secret shares. We let $(i, p(i))$ be the share for participant P_i. Because $\{c_i\}_{i=0}^{n}$ are chosen uniformly at random in $\mathbb{F}_q$, $p(i)$ will appear to be randomly distributed in $\mathbb{F}_q$.

14.2.2 Secret Reconstruction

We now show how to reconstruct the secret $s = c_0$.

We let $\mathcal{R} \subseteq \mathcal{P}$ be a valid subset of participants who are able to reconstruct the secret; that is, we have $|\mathcal{R}| = t + 1$. We now denote the secret share for participant P_k as

$$s_k \coloneqq p(k). \tag{14.2}$$

This is meant to simplify the equations below.

We review some information related to Lagrange Interpolation from Chapter 5.4. The Lagrange interpolating polynomial for $\{(x_k, y_k)\}_{k=1}^{n}$ is

$$\begin{aligned} L(x) &\coloneqq \sum_{k=1}^{n} y_k \ell_k(x) \\ \ell_k(x) &\coloneqq \prod_{\substack{1 \leq j \leq n \\ j \neq k}} \frac{x - x_j}{x_k - x_j}. \end{aligned} \tag{14.3}$$

These equations were previously presented in Eq. (5.70). In this setting, our data is $\{(k, s_k)\}_{P_k \in \mathcal{R}}$. Thus, we have the following interpolating polynomial:

$$\begin{aligned} L(x) &= \sum_{P_k \in \mathcal{R}} s_k \ell_k(x) \\ \ell_k(x) &= \prod_{\substack{P_j \in \mathcal{R} \\ j \neq k}} \frac{x-j}{k-j} \\ &= \prod_{\substack{P_j \in \mathcal{R} \\ j \neq k}} \frac{j-x}{j-k}. \end{aligned} \tag{14.4}$$

In order to compute the secret s, we need to evaluate $L(0)$. This gives us

$$\begin{aligned} s &= \sum_{P_k \in \mathcal{R}} s_k R_k \\ R_k &= \prod_{\substack{P_j \in \mathcal{R} \\ j \neq k}} \frac{j}{j-k}. \end{aligned} \tag{14.5}$$

14.2.3 Examples

We work through some examples to understand the entire process.

Example 14.1 (Alice, Bob, and Charlie share a secret)
Code may be found at `examples/secret_sharing/secret_sharing_1.py`.

As mentioned previously, Alice, Bob, and Charlie formed a business and want to share a secret key to their bank account whereby 2 out of 3 must sign off on any withdrawals. Thus, they want a $(1, 3)$-threshold system.

We begin by choosing $q = 7919$. We have the secret $s = 2310$ and polynomial

$$p(x) = 2310 + 4673x. \tag{14.6}$$

In this case, we see Alice, Bob, and Charlie receive the shares

$$\begin{aligned} \text{Alice} &= (1, p(1)) \\ &= (1, 6983) \\ \text{Bob} &= (2, p(2)) \\ &= (2, 3737) \\ \text{Charlie} &= (3, p(3)) \\ &= (3, 491)\,. \end{aligned} \tag{14.7}$$

We now look at all three possible share combinations: Alice and Bob, Alice and Charlie, and Bob and Charlie.

- Alice and Bob:

  In this case, we have that

$$\begin{aligned} s &= s_1 R_1 + s_2 R_2 \\ s_1 &= 6983 \\ s_2 &= 3737 \\ R_1 &= \frac{2}{2-1} \mod 7919 \\ R_2 &= \frac{1}{1-2} \mod 7919. \end{aligned} \tag{14.8}$$

  We reduce these modulo q to find

$$\begin{aligned} R_1 &= \frac{2}{2-1} \mod 7919 \\ &= 2 \\ R_2 &= \frac{1}{1-2} \mod 7919 \\ &= -1 \mod 7919 \\ &= 7918. \end{aligned} \tag{14.9}$$

  This reduces to

$$\begin{aligned} s &= 6983 \cdot 2 + 3737 \cdot 7918 \mod 7919 \\ &= 2310. \end{aligned} \tag{14.10}$$

  Thus, we have arrived at the shared secret s.

- Alice and Charlie:

  In this case, we have that

$$\begin{aligned} s &= s_1 R_1 + s_3 R_3 \\ s_1 &= 6983 \\ s_3 &= 491 \\ R_1 &= \frac{3}{3-1} \mod 7919 \\ R_3 &= \frac{1}{1-3} \mod 7919. \end{aligned} \tag{14.11}$$

In this case, we actually have to do some operations modulo q. We see that

$$\begin{aligned} R_1 &= \frac{3}{2} \mod 7919 \\ &= 3961 \\ R_3 &= -\frac{1}{2} \mod 7919 \\ &= 3959. \end{aligned} \tag{14.12}$$

This reduces to

$$\begin{aligned} s &= 6983 \cdot 3961 + 491 \cdot 3959 \mod 7919 \\ &= 2310. \end{aligned} \tag{14.13}$$

Thus, we have arrived at the shared secret s.

- Bob and Charlie:
  In this case, we have that

$$\begin{aligned} s &= s_2 R_2 + s_3 R_3 \\ s_2 &= 3737 \\ s_3 &= 491 \\ R_2 &= \frac{3}{3-2} \mod 7919 \\ R_3 &= \frac{2}{2-3} \mod 7919. \end{aligned} \tag{14.14}$$

We see that

$$\begin{aligned} R_2 &= 3 \\ R_3 &= \frac{2}{2-3} \mod 7919 \\ &= -2 \mod 7919 \\ &= 7917. \end{aligned} \tag{14.15}$$

This reduces to

$$\begin{aligned} s &= 3737 \cdot 3 + 491 \cdot 7917 \mod 7919 \\ &= 2310. \end{aligned} \tag{14.16}$$

Thus, we have arrived at the shared secret s.

In each case, every pair of participants are able to derive the shared secret.

Example 14.2 (Alice, Bob, Charlie, and Dave share a secret)
Code may be found at `examples/secret_sharing/secret_sharing_2.py`.

After making some good progress on their business, Alice, Bob, and Charlie decided to add Dave as a business partner. Again, with all treating each other as equals, they want the ability for a strict majority of them to be required to derive the secret key for the bank account. Thus, they now need to create a new secret and use a $(2, 4)$ threshold system.

As before, they choose $q = 7919$. This time, they have the secret $s = 1770$ and the secret polynomial

$$p(x) = 1770 + 6540x + 2889x^2. \tag{14.17}$$

In this case, we see Alice, Bob, Charlie, and Dave receive the shares

$$\begin{aligned} \text{Alice} &= (1, p(1)) \\ &= (1, 3280) \\ \text{Bob} &= (2, p(2)) \\ &= (2, 2649) \\ \text{Charlie} &= (3, p(3)) \\ &= (3, 7796) \\ \text{Dave} &= (4, p(4)) \\ &= (4, 2883)\,. \end{aligned} \tag{14.18}$$

We will look at two examples of deriving the shared secret: the triple Alice, Bob, and Dave as well as the triple Alice, Charlie, and Dave.

- Alice, Bob, and Dave:

  In this case, we have that

$$\begin{aligned} s &= s_1 R_1 + s_2 R_2 + s_4 R_4 \\ s_1 &= 3280 \\ s_2 &= 2649 \\ s_4 &= 2883 \\ R_1 &= \frac{2}{2-1} \cdot \frac{4}{4-1} \mod 7919 \\ R_2 &= \frac{1}{1-2} \cdot \frac{4}{4-2} \mod 7919 \\ R_4 &= \frac{1}{1-4} \cdot \frac{2}{2-4} \mod 7919. \end{aligned} \tag{14.19}$$

  Performing these operations we find

$$\begin{aligned}
R_1 &= \frac{2}{2-1} \cdot \frac{4}{4-1} \mod 7919 \\
&= \frac{8}{3} \mod 7919 \\
&= 5282 \\
R_2 &= \frac{1}{1-2} \cdot \frac{4}{4-2} \mod 7919 \\
&= -2 \mod 7919 \\
&= 7917 \\
R_4 &= \frac{1}{1-4} \cdot \frac{2}{2-4} \mod 7919 \\
&= \frac{1}{3} \mod 7919 \\
&= 2640.
\end{aligned} \tag{14.20}$$

This reduces to

$$\begin{aligned}
s &= 3280 \cdot 5282 + 2649 \cdot 7917 + 2883 \cdot 2640 \mod 7919 \\
&= 1770.
\end{aligned} \tag{14.21}$$

Thus, we have arrived at the shared secret s.

- Alice, Charlie, and Dave:

  In this case, we have that

$$\begin{aligned}
s &= s_1 R_1 + s_3 R_3 + s_4 R_4 \\
s_1 &= 3280 \\
s_3 &= 7796 \\
s_4 &= 2883 \\
R_1 &= \frac{3}{3-1} \cdot \frac{4}{4-1} \mod 7919 \\
R_3 &= \frac{1}{1-3} \cdot \frac{4}{4-3} \mod 7919 \\
R_4 &= \frac{1}{1-4} \cdot \frac{3}{3-4} \mod 7919.
\end{aligned} \tag{14.22}$$

  Performing these operations we find

$$\begin{aligned} R_1 &= \frac{3}{3-1} \cdot \frac{4}{4-1} \mod 7919 \\ &= 2 \\ R_3 &= \frac{1}{1-3} \cdot \frac{4}{4-3} \mod 7919 \\ &= -2 \mod 7919 \\ &= 7917 \\ R_4 &= \frac{1}{1-4} \cdot \frac{3}{3-4} \mod 7919 \\ &= 1. \end{aligned} \tag{14.23}$$

This reduces to

$$\begin{aligned} s &= 3280 \cdot 2 + 7796 \cdot 7917 + 2883 \cdot 1 \mod 7919 \\ &= 1770. \end{aligned} \tag{14.24}$$

Thus, we have arrived at the shared secret s.

In both cases we were able to derive the shared secret key.

14.2.4 Discussion

This protocol works well but has some problems. First, it requires *trusted setup*; there has to be a trusted party who determines the secret shares and distributes them to each participant. Additionally, we assume each participant *correctly shares* his portion of the secret. Furthermore, once a participant has shared his secret, there is nothing stopping the other participants from stealing it.

The required truthfulness of the dealer as well as the other participants has lead to further development of secret sharing protocols which are designed to reduce these requirements. Even so, the protocols discussed below build upon the material here.

We have not explicitly mentioned where q is used. The size of q defends the system against brute force attack. In practice, q needs to be large enough so that it is impractical to guess the key. Thus, choosing q as a 256-bit prime number would be reasonable. Of course, all of this depends on the desired level of security.

14.3 Verifiable Secret Sharing

Verifiable Secret Sharing protocols seek to combat the shortcomings discussed in Chapter 14.2.4 of Shamir's Secret Sharing protocol. Feldman's scheme [49] is frequently used but by itself is not secure [54, 55].

Due to the fact that VSS protocols are used when attempting to distribute a key between individuals, we do not mention VSS protocols any further but rather proceed to discuss distributed key generation protocols.

Function Name	Function Definition
`toBytes`	serialize object to its binary representation
`fromBytes`	deserialize object from its binary representation
ECAdd	Elliptic curve addition
ECMult	Elliptic curve scalar multiplication
PairingCheck	Validate bilinear pairing tuple

Table 14.1: Useful functions in Ethereum for the distributed key generation protocol.

14.4 Distributed Key Generation Overview and Setup

In this section we consider the problem of *distributed key generation*. Here, a collection of individuals want to distribute a private key between them in such a way that a certain portion must work together to derive it. This secret key will be used to digitally sign messages. Because of the nature of the secret key, it is *imperative* that the key not actually be constructed; rather, portions of the secret key should be used to construct partial signatures which may then be combined in a deterministic way to arrive at a valid *group signature*.

The basis for this discussion is the Ethereum Distributed Key Generation whitepaper [114]. This paper uses a variation of Feldman's VSS scheme [49] while making adjustments based on [54, 55] for security. We seek a (t, n)-threshold system whereby $t + 1$ participants are required to create a valid group signature.

We will focus on the actual implementation [114, Section 7]. While this will make our discussion specific to this particular setting, the main ideas of distributed key generation protocols should be clear. Doing this will require the use of functions particular to Ethereum; see Table 14.1 for a description of them.

The material here will require knowledge of Pairing-Based Cryptography from Chapter 12 in general and knowledge of cryptography on Ethereum [135] in Chapter 12.1 in particular. Due to the fact that smart contracts can only perform elliptic curve operations in G_1, information in G_2 will be submitted and then verified with a PairingCheck call.

The ith participant will be denoted P_i and $\mathcal{P} \coloneqq \{P_i\}_{i=1}^n$ will the collection of all participants. By working together, the participants will construct a *master public key* mpk along with its corresponding *master secret key* msk. The master secret key will *never* explicitly be formed; even so, the group will be able to work together to combine partial signatures into valid group signatures. Threshold signatures will be discussed in Chapter 14.6.

14.4.1 DKG Overview

Broadly speaking, in the distributed key generation protocol, each participant is a dealer who shares a secret key under a threshold. Thus, participants perform *simultaneous* Shamir's Secret Sharing protocols. The secret key for the entire group (the *master secret key*) will be the sum of the individual secrets. In order to protect against misbehavior, this protocol contains opportunities to prove that other participants are malicious.

14.4.2 DKG Setup

During the protocol, we will extensively use the bilinear pairing $e : G_1 \times G_2 \to G_T$. Here, we assume $|G_1| = |G_2| = |G_T| = q$ for prime q. We are assuming that $g_1, h_1 \in G_1$ are independent generators; that is, $\text{dlog}_{g_1} h_1$ is unknown. We also assume that $h_2 \in G_2$ is a generator. Furthermore, although not used here, we assume $H : \{0,1\}^* \to G_1$ is a hash function onto G_1; this hash-to-curve function will be used when making digital signatures. These signatures will be BLS signatures [29].

14.5 Distributed Key Generation Protocol

The distributed key generation protocol is split into a number of different phases. We begin by giving an overview of each phase before discussing them in detail.

We note that there are variations of communication models for different DKG protocols. Some assume secure communication between each pair of participants while others broadcast encrypted messages. The particulars matter in that the communication model determines how proofs of security are performed, but that is not our primary focus; the goal here is to give an *introduction* to DKG protocols. In our case, we will assume the ability to broadcast messages to all participants.

Additionally, we will assume the distributed key generation protocol restarts upon any malicious behavior. While this is not strictly necessary, this may be useful in certain situations.

DKG Protocol Overview

- **Registration Phase:** During this phase the participants register public keys to enable secure communication throughout the protocol.
- **Share Submission Phase:** Each participant submits encrypted shares and commitments which are then broadcast to all participants.
- **Share Dispute Phase:** Anyone who incorrectly submitted his secret share information may be accused.
- **Key Share Submission Phase:** All users submit their portion of the master public key.
- **MPK Submission Phase:** One user submits the master public key.
- **GPK Submission Phase:** All users submit their group public key.
- **GPK Dispute Phase:** Anyone who incorrectly shared his group public key may be accused.
- **Completion Phase:** The DKG process has finished. At this point, all information is valid and may be used to compute valid group signatures.

14.5.1 Registration Phase

Participant $P_i \in \mathcal{P}$ will choose a secret key $\mathrm{sk}_i \xleftarrow{\$} \mathbb{F}_q^*$; this will determine the corresponding public key

$$\mathrm{pk}_i \coloneqq g_1^{\mathrm{sk}_i}. \tag{14.25}$$

This pair $(\mathrm{sk}_i, \mathrm{pk}_i)$ *will not* be used for signing; rather, it will be used for secure communication over an insecure channel.

The public key pk_i will be submitted by participant P_i and broadcast to all participants. In this way, every participant will know P_i's public key.

14.5.2 Share Submission Phase

As mentioned in the overview, each participant will now perform Shamir's Secret Sharing protocol in parallel to share a secret.

Participant P_i will choose $s_i \xleftarrow{\$} \mathbb{F}_q$ to be his secret. He will then choose his private polynomial $f_i : \mathbb{F}_q \to \mathbb{F}_q$:

$$f_i(x) = c_{i,0} + c_{i,1}x + c_{i,2}x^2 + \cdots + c_{i,t}x^t. \tag{14.26}$$

Here, $c_{i,0} = s_i$ and $c_{i,j} \xleftarrow{\$} \mathbb{F}_q$ for $j \in \{1, \cdots, t\}$.

The secret share from P_i to P_j is

$$s_{i\to j} \coloneqq f_i(j). \tag{14.27}$$

In order to ensure honesty, P_i will have the commitments

$$C_{i,j} \coloneqq g_1^{c_{i,j}}. \tag{14.28}$$

These commitments then result in the corresponding public polynomial $F_i : \mathbb{F}_q \to G_1$:

$$F_i(x) = C_{i,0}C_{i,1}^{x}C_{i,2}^{x^2}\cdots C_{i,t}^{x^t}. \tag{14.29}$$

These public polynomials make it possible to prove that secrets were correctly shared. This is because we have

$$F_i(j) = g_1^{f_i(j)}. \tag{14.30}$$

In order to ensure the secret shares stay secret, the shares are encrypted under a symmetric key encryption scheme based on the Diffie-Hellman Key Exchange; the algorithms for encryption and decryption will be denoted Encrypt and Decrypt. Full algorithmic details may be found here.

The encrypted form of the secret share from P_i to P_j is given by

$$\overline{\mathsf{s}}_{i\to j} \coloneqq \textsc{Encrypt}\left(\mathrm{sk}_i, \mathrm{pk}_j, s_{i\to j}, j\right). \tag{14.31}$$

Using these encrypted shares, participant P_i broadcasts the following message:

$$\{\overline{\mathrm{s}}_{i\to 1}, \overline{\mathrm{s}}_{i\to 2}, \cdots, \overline{\mathrm{s}}_{i\to i-1}, \overline{\mathrm{s}}_{i\to i+1}, \cdots, \overline{\mathrm{s}}_{i\to n}, C_{i,0}, C_{i,1}, \cdots, C_{i,t}\}\,. \tag{14.32}$$

More explicitly, all of the necessary encrypted shares are sent along with the coefficients of the public polynomial. Implicitly, P_i will also send the secret share $s_{i\to i}$ to himself. A cryptographic hash of the encrypted shares and commitments will be stored should an accusation need to be made in the future; accusations will be discussed below.

At this time, all participants are expected to have received all broadcast messages. If any participant who registered failed to submit shares, it is possible for the protocol to continue; that is the path chosen in [114]. It may also be the case that the protocol could restart. The particular choice depends on the use case.

In our case, we will assume that all participants submit share information. After receiving share information, each share must be decrypted and verified. If P_j received $\overline{\mathrm{s}}_{i\to j}$ from P_i, he then computes

$$\hat{s}_{i\to j} \coloneqq \textsc{Decrypt}\left(\mathrm{sk}_j, \mathrm{pk}_i, \overline{\mathrm{s}}_{i\to j}, j\right). \tag{14.33}$$

If the share is valid, we will have the following equality:

$$g_1^{\hat{s}_{i\to j}} = F_i(j). \tag{14.34}$$

This follows from Eq. (14.30). If the above equality does not hold, then P_i incorrectly shared his secret with P_j. An accusation must be performed in the next phase.

Secret Share Encryption Algorithm

We now discuss the encryption scheme used. This uses a shared secret to derive bit stream for an XOR cipher. The index of the participant who *receives* the message is included in order to ensure that each stream is unique. It requires the use of a hash function with output the size of $\log_2(q)$; that is, if q is a k-bit number, we require the output of the hash function to be at least k bits. See Alg. 14.1 for the full specification.

14.5.3 Share Dispute Phase

We now assume that Eq. (14.34) does not hold; that is, P_i sent P_j an invalid share. We now require that P_j *prove* that P_i is malicious.

We know from Eq. (14.30) what the desired equality should be. Thus, to prove P_i's share is invalid, we must decrypt the encrypted share and then show the desired equality of Eq. (14.34) does not hold. From the decryption algorithm in Alg. 14.1, we know that we must compute the Diffie-Hellman shared secret.

This Diffie-Hellman shared secret k_{ij} can easily be computed. The challenge is how can we *prove* this is, in fact, the shared secret? We recall that we have the following equalities:

$$\begin{aligned} g_1^{\mathrm{sk}_j} &= \mathrm{pk}_j \\ \mathrm{pk}_i^{\mathrm{sk}_j} &= k_{ij}. \end{aligned} \tag{14.35}$$

Algorithm 14.1 Encryption scheme in the distributed key generation protocol

procedure ENCRYPT(sk_i,pk_j,$s_{i\to j}$,j)
  $k \coloneqq \text{pk}_j^{\text{sk}_i}$ ▷ Compute the Diffie-Hellman Key Exchange shared secret
  $\texttt{kX} \coloneqq \texttt{toBytes}(k_x)$ ▷ Use the x coordinate
  $\texttt{j} \coloneqq \texttt{toBytes}(j)$
  $\texttt{s} \coloneqq \texttt{toBytes}(s_{i\to j})$
  $\texttt{hashKJ} \coloneqq \text{HASH}(\texttt{kX} \| \texttt{j})$
  $\overline{\texttt{s}}_{i\to j} \coloneqq \texttt{s} \oplus \texttt{HashKJ}$
  return $\overline{\texttt{s}}_{i\to j}$
end procedure

procedure DECRYPT(sk_i,pk_j,$\overline{\texttt{s}}_{i\to j}$,$j$)
  $k \coloneqq \text{pk}_j^{\text{sk}_i}$ ▷ Compute the Diffie-Hellman Key Exchange shared secret
  $\texttt{kX} \coloneqq \texttt{toBytes}(k_x)$ ▷ Use the x coordinate
  $\texttt{j} \coloneqq \texttt{toBytes}(j)$
  $\texttt{hashKJ} \coloneqq \text{HASH}(\texttt{kX} \| \texttt{j})$
  $\texttt{s} \coloneqq \overline{\texttt{s}}_{i\to j} \oplus \texttt{HashKJ}$
  $s_{i\to j} \coloneqq \texttt{fromBytes}(\texttt{s})$
  return $s_{i\to j}$
end procedure

procedure DECRYPTSS(k,$\overline{\texttt{s}}_{i\to j}$,$j$)
  $\texttt{kX} \coloneqq \texttt{toBytes}(k_x)$ ▷ Use the x coordinate
  $\texttt{j} \coloneqq \texttt{toBytes}(j)$
  $\texttt{hashKJ} \coloneqq \text{HASH}(\texttt{kX} \| \texttt{j})$
  $\texttt{s} \coloneqq \overline{\texttt{s}}_{i\to j} \oplus \texttt{HashKJ}$
  $s_{i\to j} \coloneqq \texttt{fromBytes}(\texttt{s})$
  return $s_{i\to j}$
end procedure

Because the public keys pk_i and pk_j are public knowledge, if we can show that the pairs (g_1, pk_j) and (pk_i, k) have the same discrete logarithm, then this *proves* that k is actually the shared secret k_{ij}. To do this, we can use discrete logarithm equality proofs from Chapter 13.3.2; see Alg. 14.2 for full details.

Thus, P_j can broadcast the Diffie-Hellman shared secret k_{ij} along with discrete logarithm proof $\pi(k_{ij})$ which may be verified by everyone. At this point, all users may decrypt $\overline{\texttt{s}}_{i\to j}$ using DECRYPTSS:

$$\hat{s}_{i\to j} \coloneqq \text{DECRYPT}SS\left(k_{ij}, \overline{\texttt{s}}_{i\to j}, j\right). \tag{14.36}$$

Everyone can verify that

$$g_1^{\hat{s}_{i\to j}} \neq F_i(j). \tag{14.37}$$

This proves that P_i shared an invalid share with P_j; this is a malicious action.

Algorithm 14.2 Discrete logarithm equality proof and validation

Require: Group order q

procedure DLEQ-PROOF(x_1,y_1,x_2,y_2,α)
  $w \xleftarrow{\$} \mathbb{Z}_q^*$ ▷ Generate proof that $y_1 = x_1^\alpha$ and $y_2 = x_2^\alpha$ for secret α
  $t_1 \coloneqq x_1^w$
  $t_2 \coloneqq x_2^w$
  $c \coloneqq \text{HASH}(x_1, y_1, x_2, y_2, t_1, t_2)$
  $r \coloneqq w - c\alpha \mod q$
  $\pi \coloneqq (c, r)$
  return π
end procedure

procedure DLEQ-VERIFY(x_1,y_1,x_2,y_2,π) ▷ Validate that $y_1 = x_1^\alpha$ and $y_2 = x_2^\alpha$
  $t_1' \coloneqq y_1^c x_1^r$
  $t_2' \coloneqq y_2^c x_2^r$
  $c' \coloneqq \text{HASH}(x_1, y_1, x_2, y_2, t_1', t_2')$
  if $c' = c$ **then**
    return `true`
  else
    return `false`
  end if
end procedure

When this is implemented on Ethereum, P_i's encrypted shares and commitments would need to be resubmitted. After confirming their validity by rehashing the data, all accusation logic may be performed by a smart contract. If we do not have equality Eq. (14.34), then P_j just proved P_i is malicious because he submitted an invalid share; if we have equality Eq. (14.34), then P_j is malicious for submitting an invalid accusation.

14.5.4 Key Share Submission Phase

If no one submitted any invalid shares, then the next step is for each participant to submit his portion of the master public key.

To do this, P_i must submit $h_1^{s_i}$ along with proof

$$\pi(h_1^{s_i}) \coloneqq \text{DLEQ-PROOF}(g_1, g_1^{s_i}, h_1, h_1^{s_i}, s_i) \tag{14.38}$$

and $h_2^{s_i}$. This is possible because $g_1^{s_i} = C_{i,0}$ is public knowledge.

A smart contract call will confirm the validity of $h_1^{s_i}$ given $\pi(h_1^{s_i})$. A PAIRINGCHECK will confirm the validity of $h_2^{s_i}$ by requiring

$$\text{PAIRINGCHECK}(h_1^{s_i}, h_2^{-1}, h_1, h_2^{s_i}) = \texttt{true}. \tag{14.39}$$

The values $h_1^{s_i}$, $\pi(h_1^{s_i})$, and $h_2^{s_i}$ are then broadcast to all users. The values $\{h_1^{s_i}\}_{P_i \in \mathcal{P}}$ are stored for the next phase.

14.5.5 MPK Submission Phase

After all key shares have been submitted, we can compute the *master public key* mpk:

$$\text{mpk} \coloneqq \prod_{P_i \in \mathcal{P}} h_2^{s_i}. \tag{14.40}$$

The corresponding *master secret key* msk is defined similarly:

$$\text{msk} \coloneqq \sum_{P_i \in \mathcal{P}} s_i. \tag{14.41}$$

The mpk may be validated upon submission because we assume that $\{h_1^{s_i}\}_{P_i \in \mathcal{P}}$ are stored. Then, we can form

$$\text{mpk}^* \coloneqq \prod_{P_i \in \mathcal{P}} h_1^{s_i} \tag{14.42}$$

and then confirm

$$\textsc{PairingCheck}(\text{mpk}^*, h_2^{-1}, h_1, \text{mpk}) = \texttt{true}. \tag{14.43}$$

The reason we should not use $\prod_{P_i \in \mathcal{P}} g_1^{s_i}$ as the master public key is because the $g_1^{s_i}$ values are public knowledge after the initial share. Because of this, some participants could attempt to bias the distribution of $\prod_{P_i \in \mathcal{P}} g_1^{s_i}$ based on the order of submission; see [54, 55] for further discussion.

14.5.6 GPK Submission Phase

At this point, we have constructed the mpk and distributed the msk between all participants. We now focus on how to compute valid group signatures. Doing this requires signing under a special private key. These partial signatures may then be combined in a deterministic way to form a valid group signature.

Participant P_j has a *group secret key* gsk_j:

$$\text{gsk}_j \coloneqq \sum_{P_i \in \mathcal{P}} s_{i \to j}. \tag{14.44}$$

There is also the corresponding *group public key* gpk_j:

$$\text{gpk}_j \coloneqq h_2^{\text{gsk}_j}. \tag{14.45}$$

Based on the definition of gsk_j, we have the following:

$$\begin{aligned} g_1^{\text{gsk}_j} &= g_1^{\sum_{P_i \in \mathcal{P}} s_{i \to j}} \\ &= \prod_{P_i \in \mathcal{P}} g_1^{s_{i \to j}} \\ &= \prod_{P_i \in \mathcal{P}} g_1^{f_i(j)} \\ &= \prod_{P_i \in \mathcal{P}} F_i(j). \end{aligned} \tag{14.46}$$

In this way, $g_1^{\text{gsk}_j}$ may be constructed from public information.

We now define

$$\text{gpk}_j^* \coloneqq \prod_{P_i \in \mathcal{P}} F_i(j). \tag{14.47}$$

The definition of gpk_j^* in Eq. (14.47) along with the derivation in Eq. (14.46) shows us that

$$\text{gpk}_j^* = g_1^{\text{gsk}_j}. \tag{14.48}$$

This implies that gpk_j is a valid group public key if

$$\textsc{PairingCheck}(\text{gpk}_j^*, h_2^{-1}, g_1, \text{gpk}_j) = \texttt{true}. \tag{14.49}$$

This fact gives us the ability to verify all gpk_j submissions.

14.5.7 GPK Dispute Phase

Given a submission gpk_j, all parties can compute gpk_j^* from Eq. (14.47) and verify

$$\textsc{PairingCheck}(\text{gpk}_j^*, h_2^{-1}, g_1, \text{gpk}_j) = \texttt{true}. \tag{14.50}$$

This logic may also be performed by a smart contract, although care must be taken to ensure efficient computation.

14.5.8 Completion Phase

After allowing sufficient time for accusations of malicious gpk_j submissions, the distributed key generation process is complete.

14.6 Threshold Signatures

14.6.1 Deriving the Master Secret Key from the Group Secret Keys

Now that we have constructed the master public key, we will focus on computing group signatures. We let $\mathcal{R} \subseteq \mathcal{P}$ be subset such that $|\mathcal{R}| = t + 1$.

We recall the definition of gsk_j in Eq. (14.44):

$$\text{gsk}_j = \sum_{P_i \in \mathcal{P}} s_{i \to j}. \tag{14.51}$$

We remember that $s_{i \to j}$ is the secret share from P_i to P_j. This secret share $s_{i \to j}$ is an evaluation of P_i's private polynomial. Written another way, we have

$$\text{gsk}_j = \sum_{P_i \in \mathcal{P}} f_i(j). \tag{14.52}$$

To make valid group signatures, we need to combine signatures signed by the gsk_j keys. In order to understand how this works, we will show how we can derive the master secret key from the group secret keys.

We begin by recalling the definition of msk from Eq. (14.41):

$$\text{msk} = \sum_{P_i \in \mathcal{P}} s_i. \tag{14.53}$$

From Eq. (14.26) and the surrounding discussion, we know that

$$s_i = f_i(0). \tag{14.54}$$

This implies that msk can be rewritten in terms of evaluations of a private polynomials:

$$\text{msk} = \sum_{P_i \in \mathcal{P}} f_i(0). \tag{14.55}$$

As we have stated before, during the share distribution phase P_i performed Shamir's Secret Sharing. Given $\mathcal{R}$ as before, this means that the members of $\mathcal{R}$ can work together to compute

$$\begin{aligned} f_i(0) &= \sum_{P_j \in \mathcal{R}} f_i(j) R_j \\ R_j &= \prod_{\substack{P_k \in \mathcal{R} \\ k \neq j}} \frac{k}{k - j}. \end{aligned} \tag{14.56}$$

This is coming from Chapter 14.2; *note the indices are different.* Rewriting this, we have

$$\begin{aligned} s_i &= \sum_{P_j \in \mathcal{R}} s_{i \to j} R_j \\ R_j &= \prod_{\substack{P_k \in \mathcal{R} \\ k \neq j}} \frac{k}{k - j}. \end{aligned} \tag{14.57}$$

If we sum over $P_i \in \mathcal{P}$, then we have

$$\begin{aligned}\text{msk} &= \sum_{P_i \in \mathcal{P}} s_i \\ &= \sum_{P_i \in \mathcal{P}} \left[\sum_{P_j \in \mathcal{R}} s_{i \to j} R_j \right] \\ &= \sum_{P_j \in \mathcal{R}} \left[\sum_{P_i \in \mathcal{P}} s_{i \to j} \right] R_j \\ &= \sum_{P_j \in \mathcal{R}} \text{gsk}_j R_j. \end{aligned} \tag{14.58}$$

We can use this equality to construct valid group signatures.

In more technical mathematical language, we are performing Lagrange Interpolation over a finite field.

14.6.2 Constructing Valid Group Signatures

We can now use the work from the previous section to show how we are able to combine partial signatures into a valid group signature.

The group wants to sign the message $m \in \{0,1\}^*$. We suppose that $\mathcal{R} \subseteq \mathcal{P}$ be as before; that is, we have $|\mathcal{R}| = t + 1$. We now suppose that they have the individual signatures

$$\sigma_j \coloneqq [H(m)]^{\text{gsk}_j}. \tag{14.59}$$

We define the constants

$$R_j = \prod_{\substack{P_k \in \mathcal{R} \\ k \neq j}} \frac{k}{k - j}. \tag{14.60}$$

We set

$$\sigma \coloneqq \prod_{P_j \in \mathcal{R}} \sigma_j^{R_j}. \tag{14.61}$$

This is a valid group signature of m under the master public key mpk. To see this, we note

$$
\begin{aligned}
e\left(\sigma, h_2\right) &= e\left(\prod_{P_j \in \mathcal{R}} \sigma^{R_j}, h_2\right) \\
&= \prod_{P_j \in \mathcal{R}} e\left(\sigma^{R_j}, h_2\right) \\
&= \prod_{P_j \in \mathcal{R}} e\left(H(m)^{\mathrm{gsk}_j R_j}, h_2\right) \\
&= \prod_{P_j \in \mathcal{R}} e\left(H(m), h_2^{\mathrm{gsk}_j R_j}\right) \\
&= e\left(H(m), \prod_{P_j \in \mathcal{R}} h_2^{\mathrm{gsk}_j R_j}\right) \\
&= e\left(H(m), h_2^{\sum_{P_j \in \mathcal{R}} \mathrm{gsk}_j R_j}\right) \\
&= e\left(H(m), h_2^{\mathrm{msk}}\right) \\
&= e\left(H(m), \mathrm{mpk}\right). \qquad (14.62)
\end{aligned}
$$

Thus, we have a valid group signature from combining individual signatures in a deterministic manner. The constants only depend on which partial signatures are used in the construction. Additionally, they individually may be validated before combining them because the gpk_j are public knowledge. It may be verified by ensuring

$$\textsc{PairingCheck}\left(\sigma, h_2^{-1}, H(m), \mathrm{mpk}\right) = \texttt{true}. \qquad (14.63)$$

In more technical mathematical language, we are performing Lagrange Interpolation over a group parameterized by elements of a finite field.

14.7 Distributed Key Generation Example

We now walk through a trimmed down example of the distributed key generation protocol in order to see how it would work out in a smaller setting. In particular, we walk through the construction of the master secret key.

Example 14.3 (Alice, Bob, Charlie, and Dave distribute a key)
Code may be found at `examples/secret_sharing/dkg.py`.

As before, Alice, Bob, Charlie, and Dave have started a business. This time, though, they wish to perform the distributed key generation protocol in order to construct their own master secret key. They want to require that 3 members work together to sign a message for the entire group. Thus, they want a $(2, 4)$ protocol. As before, we will be working in $\mathbb{F}_{7919}$.

- Alice has the private polynomial

$$f_1(x) = 5853 + 2695x + 4447x^2 \tag{14.64}$$

and secret shares

$$\begin{aligned} s_{1\to 1} &= 5076 \\ s_{1\to 2} &= 5274 \\ s_{1\to 3} &= 6447 \\ s_{1\to 4} &= 676. \end{aligned} \tag{14.65}$$

- Bob has the private polynomial

$$f_2(x) = 87 + 1150x + 7782x^2 \tag{14.66}$$

and secret shares

$$\begin{aligned} s_{2\to 1} &= 1100 \\ s_{2\to 2} &= 1839 \\ s_{2\to 3} &= 2304 \\ s_{2\to 4} &= 2495. \end{aligned} \tag{14.67}$$

- Charlie has the private polynomial

$$f_3(x) = 6991 + 3631x + 5061x^2 \tag{14.68}$$

and secret shares

$$\begin{aligned} s_{3\to 1} &= 7764 \\ s_{3\to 2} &= 2821 \\ s_{3\to 3} &= 81 \\ s_{3\to 4} &= 7463. \end{aligned} \tag{14.69}$$

- Dave has the private polynomial

$$f_4(x) = 5304 + 2184x + 2803x^2 \tag{14.70}$$

and secret shares

$$\begin{aligned} s_{4\to 1} &= 2372 \\ s_{4\to 2} &= 5046 \\ s_{4\to 3} &= 5407 \\ s_{4\to 4} &= 3455. \end{aligned} \tag{14.71}$$

The master secret key is the sum of the constant terms:

$$\begin{aligned} \text{msk} &= 5853 + 87 + 6991 + 5304 \mod 7919 \\ &= 2397. \end{aligned} \tag{14.72}$$

We now look at computing the gsk_j's:

- Alice has received the secret shares

$$\begin{aligned} s_{1\to 1} &= 5076 \\ s_{2\to 1} &= 1100 \\ s_{3\to 1} &= 7764 \\ s_{4\to 1} &= 2372. \end{aligned} \tag{14.73}$$

  She then determines

$$\begin{aligned} \text{gsk}_1 &= s_{1\to 1} + s_{2\to 1} + s_{3\to 1} + s_{4\to 1} \mod q \\ &= 5076 + 1100 + 7764 + 2372 \mod 7919 \\ &= 474. \end{aligned} \tag{14.74}$$

- Bob has received the secret shares

$$\begin{aligned} s_{1\to 2} &= 5274 \\ s_{2\to 2} &= 1839 \\ s_{3\to 2} &= 2821 \\ s_{4\to 2} &= 5046. \end{aligned} \tag{14.75}$$

  He then determines

$$\begin{aligned} \text{gsk}_2 &= s_{1\to 2} + s_{2\to 2} + s_{3\to 2} + s_{4\to 2} \mod q \\ &= 5274 + 1839 + 2821 + 5046 \mod 7919 \\ &= 7061. \end{aligned} \tag{14.76}$$

- Charlie has received the secret shares

$$\begin{aligned} s_{1\to 3} &= 6447 \\ s_{2\to 3} &= 2304 \\ s_{3\to 3} &= 81 \\ s_{4\to 3} &= 5407. \end{aligned} \tag{14.77}$$

  He then determines

$$\begin{aligned} \text{gsk}_3 &= s_{1\to 3} + s_{2\to 3} + s_{3\to 3} + s_{4\to 3} \mod q \\ &= 6447 + 2304 + 81 + 5407 \mod 7919 \\ &= 6320. \end{aligned} \tag{14.78}$$

- Dave has received the secret shares

$$\begin{aligned} s_{1\to 4} &= 676 \\ s_{2\to 4} &= 2495 \\ s_{3\to 4} &= 7463 \\ s_{4\to 4} &= 3455. \end{aligned} \tag{14.79}$$

  He then determines

$$\begin{aligned} \text{gsk}_4 &= s_{1\to 4} + s_{2\to 4} + s_{3\to 4} + s_{4\to 4} \mod q \\ &= 676 + 2495 + 7463 + 3455 \mod 7919 \\ &= 6170. \end{aligned} \tag{14.80}$$

We previously calculated the R_j coefficients in Example 14.2, so we will reuse those here.

- Alice, Bob, and Dave can combine their group secret keys and coefficients to find

$$\begin{aligned} \text{gsk}_1 R_1 + \text{gsk}_2 R_2 + \text{gsk}_4 R_4 \mod q &= 474 \cdot 5282 + 7061 \cdot 7917 + 6170 \cdot 2640 \mod 7919 \\ &= 2397. \end{aligned} \tag{14.81}$$

  This is the master secret key.

- Alice, Charlie, and Dave can combine their group secret keys and coefficients to find

$$\begin{aligned} \text{gsk}_1 R_1 + \text{gsk}_3 R_3 + \text{gsk}_4 R_4 \mod q &= 474 \cdot 2 + 6320 \cdot 7917 + 6170 \cdot 1 \mod 7919 \\ &= 2397. \end{aligned} \tag{14.82}$$

  This is the master secret key.

Chapter 15

Computational Hardness Assumptions

In this chapter we discuss some of the hardness assumptions inherent to Public Key Cryptography. We will focus on those related to discrete logarithm systems because they are important to Elliptic Curve Cryptography.

When talking about computational infeasibility, we assume that it is not possible to do 2^{128} operations. Table 15.1 describes key length requirements and has been included from [69, Page 381]; the notation has been slightly modified to match ours.

15.1 Discrete Logarithm Problem

15.1.1 Formal Definition

Definition 15.1 (Discrete Logarithm Problem)**.** Given a finite cyclic group $G = \langle g \rangle$ and $h \xleftarrow{\$} G$, compute x such that

$$h = g^x. \tag{15.1}$$

15.1.2 General Methods for Solving DLP

We assume that $|G| = n$ is a k-bit number. There are some well known general methods for solving the Discrete Logarithm Problem. These include

- Baby Step, Giant Step: the total cost is $O(2^{k/2})$ space and $O(2^{k/2})$ time.
- Pollard's Rho: the total cost is $O(1)$ space and expected $O(2^{k/2})$ time; this is a probabilistic algorithm.

Both of these algorithms are *exponential* algorithms; the constants hidden in the $O(\cdot)$ are small. They are also called *square root* algorithms because the cost is approximately the square root of $|G|$. This implies that if we want k-bit security, then $|G|$ must be a $2k$-bit number. These methods are discussed in [69, Chapter 10].

If n is not prime, it is sometimes possible to speed up these attacks even further. Because of this, secure algorithms should use cyclic groups whose size is prime or which contains a *large* prime factor.

Effective Key Length	**Discrete Logarithm**	
	Order-q Subgroup of $\mathbb{F}_p^*$	**Elliptic Curve Group Order q**
112	p: 2048, q: 224	224
128	p: 3072, q: 256	256
192	p: 7680, q: 384	384
256	p: 15360, q: 512	512

Table 15.1: Estimated key lengths for specified security levels [69, Page 381]. The effective key length determines the required number of operations. An effective key length of 128 bits means that it will take approximately 2^{128} operations to break the cryptosystem. As we can see, the order of elliptic curve groups are *significantly smaller* than those required for subgroups of $\mathbb{F}_p$.

15.1.3 Discussion

When we are designing a cryptographic system which relies on the Discrete Logarithm Problem for security, we must ensure that it is impractical to solve using all known methods. Thus, the generic methods described ensure that $|G|$ must be at least 2^{2k} for k-bit security. This estimate only holds when these generic algorithms are the best algorithms.

When solving the Discrete Logarithm Problem in $\mathbb{F}_p$, though, better methods exist. One such method is the *index calculus algorithm* and its complexity is *subexponential* in $\log_2(p)$ [69, Chapter 10]. Because of this, key lengths in $\mathbb{F}_p$ must be *significantly* larger. From Table 15.1, we see that p must be a 3072-bit prime for 128-bit security.

We can contrast this with solving the Discrete Logarithm Problem on subgroups of elliptic curves. There are no known algorithms which beat the generic algorithms when solving DLP over elliptic curves (with some well-known exceptions). From Table 15.1, we see that we need to choose a 256-bit prime p when we want 128-bit security. This means that public keys will be much smaller in cryptosystems built using elliptic curves than $\mathbb{F}_p$.

15.2 Diffie-Hellman Problems

15.2.1 Formal Definitions

Definition 15.2 (Computational Diffie-Hellman Problem)**.** Fix a finite cyclic group $G = \langle g \rangle$ of order q and $a, b \xleftarrow{\$} \mathbb{Z}_q$. Let $A = g^a$ and $B = g^b$. Given A and B, compute g^{ab}.

Definition 15.3 (Decisional Diffie-Hellman Problem)**.** Fix a finite cyclic group $G = \langle g \rangle$ of order q and $a, b, c \xleftarrow{\$} Z_q$. Let $A = g^a$, $B = g^b$, and $C = g^c$. Distinguish g^{ab} and C.

15.2.2 General Methods for Solving CDH and DDH

It is clear that *if* we can solve DLP, then we can solve CDH. This follows because if we can solve DLP, then we could solve for a in

$$A = g^a \tag{15.2}$$

and then compute

$$\begin{aligned} g^{ab} &= \left(g^b\right)^a \\ &= B^a. \end{aligned} \tag{15.3}$$

It is not clear that if we can solve CDH then we can solve DLP; this would imply that DLP and CDH are equivalent. Here is a paper which discusses a variant of CDH and its relationship to DLP [36]

Similarly, we can see that if we can solve CDH then we can solve DDH. The reverse is not thought to be true [69, Pages 340–1]. Additionally, we note that DDH is easy when given a bilinear pairing. A survey paper of DDH may be found in [27].

15.2.3 Additional Variations

We mention that these two Diffie-Hellman Problems (CDH and DDH) are the primary DHPs. There are variations on these which also show up in the literature, but they are usually variations on these two problems.

Chapter 16

Conclusion

The material we covered here should give an individual the necessary background to begin to understand the mathematics behind cryptography. Naturally, much more could be learned. We now discuss general paths of study and additional resources.

16.1 Suggestions for Further Study

For those interested in further study, here are some suggestions for learning cryptography at a more rigorous level. The books are the same; the difference is order.

16.1.1 General STEM and Non-Mathematicians

The path described here is based on the author's thoughts and what he thinks would be helpful.

- **Understanding Cryptography** by Christof Paar and Jan Pelzl [97]. This book gives a broad introduction to cryptography. It includes many exercises, all of which should be completed or at least attempted. The presentation of these notes was influenced by this textbook.
- **An Introduction to Mathematical Cryptography** by Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman [60]. This book should help by providing the opportunity to learn mathematics at a deeper level than what may be learned from *Understanding Cryptography*. With it being more mathematical, it will also help individuals develop the necessary mathematical maturity for rigorous cryptography. Additional *useful* mathematical material may be found in [119].
- **Introduction to Modern Cryptography** by Jonathan Katz and Yehuda Lindell [69]. This textbook is the standard reference for teaching *rigorous* cryptography focused on definitions, theorems, and proofs. This is a challenging work and assumes a level of mathematical maturity.

16.1.2 Mathematicians

The path described here is based on the author's personal experience. Here, we assume that an individual has the equivalent of a bachelor's degree in mathematics and is comfortable with theorems, definitions, and proofs; no particular knowledge of number theory is assumed. The books described are the same as above but the author recommends a different order.

- **An Introduction to Mathematical Cryptography** by Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman [60]. This book will help increase mathematical ability as it pertains to cryptography. It also introduces some useful probability theory. This will help mathematicians understand where mathematics fits into cryptography. If additional mathematical material is desired, see [119].
- **Understanding Cryptography** by Christof Paar and Jan Pelzl [97]. This book gives a broad introduction to cryptography. Although the parts devoted to mathematics are not difficult, the book does an excellent job of giving an overview of the major areas of cryptography. In this way, a broad understanding of cryptography will be gained.
- **Introduction to Modern Cryptography** by Jonathan Katz and Yehuda Lindell [69]. This textbook is the standard reference for teaching *rigorous* cryptography focused on proofs. With a background in mathematics, the level of mathematical maturity should not be difficult; the challenge may be learning a new field.

16.2 Online Resources

Here are non-exhaustive lists of online resources. The author has used some, but not all, of these resources.

16.2.1 Online Cryptography Resources

- The Cryptopals website: https://cryptopals.com/. This website helps you learn cryptography by solving problems.
- IACR website: https://iacr.org/. The International Association for Cryptologic Research is a professional society devoted to cryptography research. IACR also has a preprint server that has many interesting cryptography papers: https://eprint.iacr.org/.
- Online cryptography notes: there are a number of online cryptography notes. These are all around the level of *Introduction to Modern Cryptography* [69].
  - A Graduate Course in Applied Cryptography[1] by Dan Boneh and Victor Shoup [30]. This is a work in progress that is mostly complete but does not have references.

[1] https://toc.cryptobook.us/

  - An Intensive Introduction to Cryptography[2] by Boaz Barak. This is a work in progress.
  - A Course in Cryptography[3] by Rafael Pass and Abhi Shelat. Pass's A Course in Discrete Structures[4] may also be helpful. Both of these are aimed at the advanced undergraduate level.
  - Introduction to Modern Cryptography[5] by Phillip Rogaway and Mihir Bellare.
  - Lecture Notes on Cryptography[6] by Shafi Goldwasser and Mihir Bellare. Some of the material here is borrowed from Rogaway and Bellare.

- Dan Boneh has an online cryptography class[7].

16.2.2 Online Mathematics Resources

- AMS Open Math Notes[8]: The American Mathematical Society has a collection of notes for standard mathematics courses at both the undergraduate and graduate level. Although learning mathematics in the classroom is usually ideal, these notes may help individuals learn additional mathematics.

- AIM Open Textbook Initiative[9]: The American Institute of Mathematics has a list of open textbooks covering many subjects including abstract algebra, number theory, and introduction to proofs.

16.3 Other Good Books

There are many other good books on cryptography and mathematics. Below are some books the author has purchased while trying to learn cryptography:

- **A Computational Introduction to Number Theory and Algebra** by Victor Shoup [119]. This is a useful book that gives an introduction to both number theory and abstract algebra at the undergraduate level. It covers a lot of useful material and may be found online[10]. *This book would be helpful when trying to build mathematical maturity.*

- **Cryptography Engineering** by Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno [51]. In some ways, this is an updated version of *Applied Cryptography* by Bruce Schneier [115], that was written in 1996; although useful at the time, it is out-of-date

[2] https://intensecrypto.org/public/
[3] https://www.cs.cornell.edu/courses/cs4830/2010fa/lecnotes.pdf
[4] https://www.cs.cornell.edu/~rafael/discmath.pdf
[5] https://www.cs.ucdavis.edu/~rogaway/classes/227/spring05/book/main.pdf
[6] https://cseweb.ucsd.edu/~mihir/papers/gb.pdf
[7] https://www.coursera.org/learn/crypto
[8] https://www.ams.org/open-math-notes
[9] https://aimath.org/textbooks/approved-textbooks/
[10] https://shoup.net/ntb/

and does not cover significant advances. Cryptography Engineering focuses on the *engineering* aspects of cryptography rather than the mathematics. This is useful when implementing cryptographic algorithms in order to know the common pitfalls and how to avoid them. This is considered the second edition to *Practical Cryptography* [50].

- **Foundations of Cryptography** (Volumes I and II) by Oded Goldreich [56, 57]. These books are standard textbooks on theoretical cryptography and rigorous proofs. They are more advanced than [69].

- **Handbook of Applied Cryptography** by Alfred Menezes, Paul van Oorschot, and Scott Vanstone [86]. This has been a standard cryptography reference. Being written in 1997, it is starting to be out of date like *Applied Cryptography* [115]. Elliptic curves are only briefly mentioned.

- **Handbook of Elliptic and Hyperelliptic Curve Cryptography** by Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren [39]. This is a standard reference for Elliptic Curve Cryptography. This text generally assumes advanced mathematical knowledge; *do not* try to learn Elliptic Curve Cryptography here.

- **Security Engineering** by Ross Anderson [2]. Although not focused on cryptography, this book helps place cryptography in the overall context of security. He talks about ways cryptographic implementations may leak information; this results in the notion of *side-channel attacks*. When implementing cryptographic algorithms, it is *critical* to not accidentally leak information through side-channels. Operations as simple as comparing bytes may leak timing information and result in leaking the private key.

Of course, not all of these books need to be purchased at once. Furthermore, the author has not yet thoroughly worked through all of these books. It may be good to hold off on purchasing the older books as well as the advanced books until they are needed.

16.4 Useful Areas of Mathematics and Books

Any serious interest in understanding advanced cryptography requires mathematics *at least* at the undergraduate level. Here are some useful areas of mathematics and potential books for study. In general, the goal would be to understand algebra and number theory; some knowledge of probability theory is helpful as well.

- Abstract Algebra: Here we gave a gentle introduction to groups, rings, and fields. A more rigorous treatment of these algebraic objects would be of great value for anyone who wants to have a better understanding of the underlying mathematics of cryptography. One standard abstract algebra book is *Abstract Algebra* by Dummit and Foote [45]. Other books may give an easier first pass on the subject, though; see Chapter 16.2.2 for online resources.

- Number Theory: as we saw, number theory is extremely useful in Public Key Cryptography. The study of elliptic curves also generally falls under number theory as well. After working through *A Computational Introduction to Number Theory and Algebra* by Victor Shoup [119], it would be helpful to spend some time on elliptic curves. One undergraduate book is *Rational Points on Elliptic Curves* by Silverman and Tate [121].

- Probability Theory: although not necessarily a part of mathematics, additional knowledge of probability is always useful. One standard undergraduate textbook is *Introduction to Probability Models* by Ross [111].

More advanced areas include

- Elliptic Curves: Advanced knowledge of Elliptic Curve Cryptography will require a deep understanding of elliptic curves. The standard graduate level textbook on elliptic curves is *The Arithmetic of Elliptic Curves* by Silverman [120]. Another book that focuses on elliptic curves in cryptography is *Elliptic Curves: Number Theory and Cryptography* by Lawrence Washington [131]. Additional study of commutative algebra would probably be helpful.

- Advanced Number Theory: Spending additional time learning number theory would complement further study on elliptic curves. One book to start may be *A Course in Number Theory and Cryptography* by Neal Koblitz [75].

The books included here are meant to be useful guides when attempting to learn advanced mathematics. Online notes may likely be found that cover the material suggested here; see potential mathematical notes and textbooks in Chapter 16.2.2. When attempting to learn mathematics through self-study, the greatest challenge will likely be gaining mathematical maturity; that is, the ability to write rigorous proofs, formalize intuitive logic, and recognize invalid logic.

Good luck.

Appendix A

Additional Cryptography

Here is material which was deemed not strictly necessary for the text, yet the author felt some individuals may find it useful. The difficulty level may vary from section to section.

A.1 Construction of Cryptographic Hash Functions

We discuss two standard methods for constructing cryptographic hash functions: the Merkle-Damgård construction and the sponge construction. The Merkle-Damgård construction is found in many hash functions such as MD5 [107], SHA-1 [124], and SHA-2 [125]. The sponge construction is the basis for KECCAK [22]/SHA-3 [46].

A.1.1 Merkle-Damgård Construction

The Merkle-Damgård construction was described independently by Merkle [87] and Damgård [43]. The main idea is to take the long message, separate the message into blocks, and operate on those blocks in an iterative fashion. The hash is then the result of this iterative process.

The construction is built around a *compression function* $h : \{0,1\}^b \times \{0,1\}^n \to \{0,1\}^b$. We begin by a discussion of hashing short messages before we discuss hashing longer messages. The desired hash output is b bits; the input message will be split up into blocks of n bits for processing. The compression function will be used to mix together the input message.

Hashing Short Messages

Let $m \in \{0,1\}^n$ be our message to hash. We let $\mathsf{IV} \in \{0,1\}^b$ be a public *initialization vector*. We can then define our hash of m:

$$H(m) \coloneqq h(\mathsf{IV}, m). \tag{A.1}$$

If h thoroughly scrambles messages, then it should be difficult to determine m from $H(m)$.

Hashing Long Messages

In practice, the messages which need to be hashed may be long; in particular, they may be longer than n bits.

Let us suppose we have a message $m \in \{0,1\}^{\ell n}$; that is, we have

$$m = m_0 \| m_1 \| \cdots \| m_{\ell-1} \tag{A.2}$$

with $m_i \in \{0,1\}^n$. We now define the hash of m by an iterative process:

$$\begin{aligned} h_0 &= h(\mathsf{IV}, m_0) \\ h_1 &= h(h_0, m_1) \\ h_2 &= h(h_1, m_2) \\ &\vdots \\ h_{\ell-1} &= h(h_{\ell-2}, m_{\ell-1}) \\ H(m) &\coloneqq h_{\ell-1}. \end{aligned} \tag{A.3}$$

In this way, we iteratively mix the entire input together to obtain the final hash output.

Message Padding

At this point, we have always assumed that our messages have been multiples of n-bits; this will not always be the case in practice. Thus, hash functions take a message as input, *pad* the message, and then hash the padded message. Within the padding, it is common to include the message length. Including the message length will ensure that messages of different lengths will *always* result in different padded messages. Padded messages will always have the following form:

$$\mathsf{pad}(m) = m_0 \| m_1 \| \cdots \| m_{\ell-1}. \tag{A.4}$$

Here, the appropriate padding scheme ensures that $m_i \in \{0,1\}^n$.

For a general message m, we have

$$\begin{aligned} \mathsf{pad}(m) &= m_0 \| m_1 \| \cdots \| m_{\ell-1} \\ h_0 &= h(\mathsf{IV}, m_0) \\ h_1 &= h(h_0, m_1) \\ h_2 &= h(h_1, m_2) \\ &\vdots \\ h_{\ell-1} &= h(h_{\ell-2}, m_{\ell-1}) \\ H(m) &\coloneqq h_{\ell-1}. \end{aligned} \tag{A.5}$$

See Figure A.1 for a graphical representation; see Alg. A.1 for the specification.

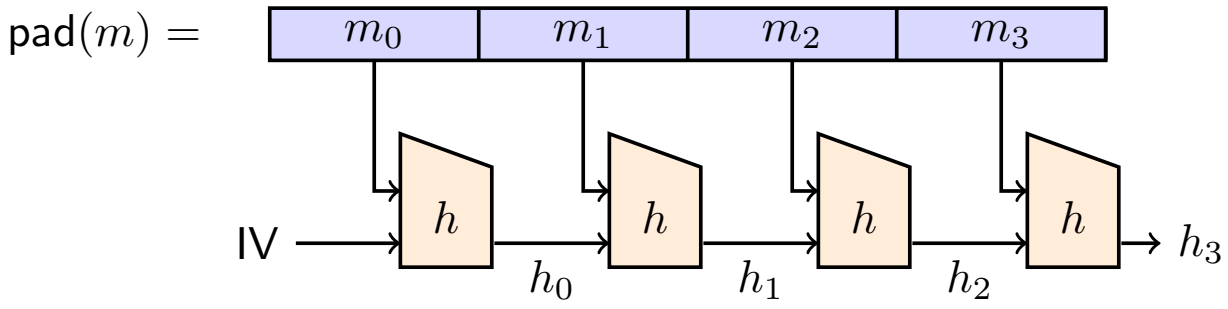


Figure A.1: Here is an example of the Merkle-Damgård construction. Based on a diagram from [62]. The message is padded so that it is a multiple of the block size. After this, the compression function h iteratively processes the message blocks.

Algorithm A.1 Merkle-Damgård construction of hash functions

```
1: procedure MERKLE-DAMGÅRD(m)
2:     m_0‖m_1‖···‖m_{k-1} := pad(m)          ▷ Pad m appropriately
3:     h_0 := h(IV, m_0)                      ▷ IV is the initialization vector
4:     for i = 1; i < k; i++ do
5:         h_i := h(h_{i-1}, m_i)
6:     end for
7:     return h_{k-1}
8: end procedure
```

With the particular message padding, there is usually some large but finite bound on the size of the message. In particular, MD5, SHA-1, and SHA-2-256 all have a maximum message length of $2^{64} - 1$ bits; SHA-2-512 has a maximum message length of $2^{128} - 1$ bits. This message length restriction results from the particulars of the padding process.

To put this length restriction in context, we note that 2^{64} bits is 2^{61} bytes or 2 exbibytes. One terabyte of data (10^{12}) is essentially one tebibyte (2^{40}); thus, the message length is limited to approximately one million terabytes.

Internal State and Length Extension Attacks

We will now discuss length extension attacks and how they affect hash functions based on the Merkle-Damgård construction.

Suppose we are given $\text{len}(x)$ and $H(x)$. That is, we are told the length of the unpadded message x and its digest, but not the message itself. Then for any message y, we can easily compute $H(\mathsf{pad}(x)\|y)$.

First, we suppose that

$$\mathsf{pad}(x) = x_0\|x_1\|\cdots\|x_{\ell-1}. \tag{A.6}$$

The iterative process tells us that

$$\begin{aligned} h_0 &= h(\mathsf{IV}, x_0) \\ &\vdots \\ h_{\ell-1} &= h(h_{\ell-2}, x_{\ell-1}) \\ H(x) &\coloneqq h_{\ell-1}. \end{aligned} \tag{A.7}$$

Now, we do not know about x_i but we *do know* $H(x) = h_{\ell-1}$; this is the *entire* internal state of the hash function once we have finished processing $\mathsf{pad}(x)$. We also know

$$\mathsf{pad}\left(\mathsf{pad}(x)\|y\right) = x_0\|x_1\|\cdots\|x_{\ell-1}\|y_0\|y_1\|\cdots\|y_{k-1}. \tag{A.8}$$

The explicit way to compute $H(\mathsf{pad}(x)\|y)$ would be the following:

$$\begin{aligned} h_0 &= h(\mathsf{IV}, x_0) \\ &\vdots \\ h_{\ell-1} &= h(h_{\ell-2}, x_{\ell-1}) \\ h_\ell &= h(h_{\ell-1}, y_0) \\ &\vdots \\ h_{\ell+k-2} &= h(h_{\ell+k-3}, y_{k-1}) \\ H(\mathsf{pad}(x)\|y) &\coloneqq h_{\ell+k-2}. \end{aligned} \tag{A.9}$$

But because we have $H(x) = h_{\ell-1}$, we can instead compute $H(\mathsf{pad}(x)\|y)$ by

$$\begin{aligned} h_\ell &= h(h_{\ell-1}, y_0) \\ h_{\ell+1} &= h(h_\ell, y_1) \\ &\vdots \\ h_{\ell+k-2} &= h(h_{\ell+k-3}, y_{k-1}) \\ H(\mathsf{pad}(x)\|y) &\coloneqq h_{\ell+k-2}. \end{aligned} \tag{A.10}$$

From our discussion about properties of random oracles, $H(\mathsf{pad}(x)\|y)$ should always be independent of $H(x)$ (even though $\mathsf{pad}(x) = x\|z$ for some z), and the only way to learn $H(\mathsf{pad}(x)\|y)$ should be to explicitly query the random oracle. We see this is not the case for the Merkle-Damgård construction.

Length extension attacks only affect those hash functions which output the entire internal state; as mentioned, these include MD5, SHA-1, SHA-2-256, and SHA-2-512. This *does not* affect SHA-2-512/224 or SHA-2-512/256; in both cases, the 512 bit internal state is truncated to 224 or 256 bits. In this way, the entire internal state is not learned, so this attack fails.

Algorithm and Variant		Internal State (bits)	Output Size (bits)	Block Size (bits)
MD5		128	128	512
SHA-1		160	160	512
SHA-2	SHA-2-256	256	256	512
	SHA-2-512	512	512	1024
SHA-3	SHA-3-256	1600	256	1088
	SHA-3-512	1600	512	576

Table A.1: Comparison of various cryptographic hash functions. We know that MD5, SHA-1, and SHA-2 are all based on the Merkle-Damgård construction; thus, their internal state and output size are the same. SHA-3 is based on sponge functions; its internal state is 1600 bits; the block size (rate) depends on the output. In the case of SHA-3-256, the capacity is $512 = 2 \cdot 256$ while the rate is $1088 = 1600 - 512$; for SHA-3-512, the capacity is $1024 = 2 \cdot 512$ while the rate is $576 = 1600 - 1024$. This table is based on the table from information from [107], [125, Figure 1], and [46, Section 6.1].

Example Merkle-Damgård Parameters

We now list the parameters used in some standard Merkle-Damgård-based hash functions; see Table A.1. The block size of MD5, SHA-1, and SHA-2-256 is 512 bits while the internal state and output size are equal (128, 160, and 256 bits, respectively). In the case of SHA-2-512, the block size is 1024 bits while the internal state and output size is 512 bits.

A.1.2 Sponge Construction

Attacks on MD5 and SHA-1 led to the *NIST hash function competition* in an effort to design a more secure hash function. One of the requirements was that all submitted hash functions must not be susceptible to length extension attacks.

Sponge Function

The winner of the competition was Keccak. These functions are based on *sponge functions*. This requires a random permutation $f : \{0,1\}^b \to \{0,1\}^b$. Here, we have integers $r + c = b$ with $r, c \geq 1$; r is called the *rate* and c is called the *capacity*. The input message m is padded and split into multiples of r blocks. These blocks are "absorbed" into the sponge. After all the blocks have been absorbed, the desired number of output bits are "squeezed" out. See Figure A.2 for a graphical representation; see Alg. A.2 for the specification. SHA-3 is an official hash function for the NIST standard; NIST modified the original padding scheme from Keccak.

More formally, we first pad the message to a multiple of the rate with the appropriate padding scheme:

$$\mathsf{pad}(m) = m_0 \| m_1 \| \cdots \| m_{k-1}. \tag{A.11}$$

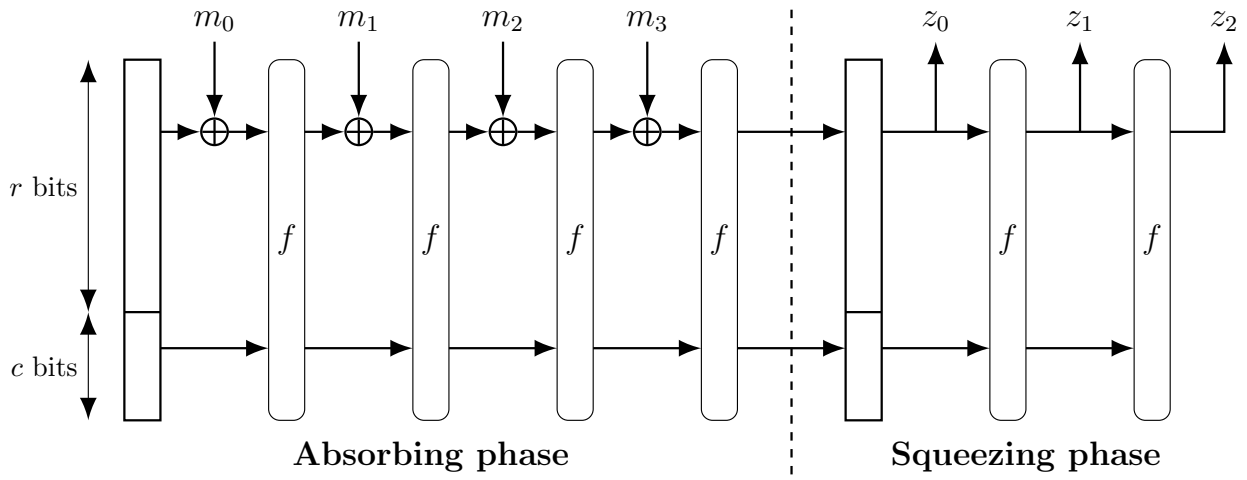


Figure A.2: Here is an example of the sponge construction. Based on a diagram from [62]. Here, the message m is first padded so that its total length is a multiple of the rate r. The sponge then absorbs the message into its internal state. After this, the desired output is squeezed out.

Algorithm A.2 Sponge construction of hash functions; based on [22, Alg. 1]

Require: $r < b$

procedure SPONGE(m, ℓ)
  $m_0 \| m_1 \| \cdots \| m_{k-1} \coloneqq \mathsf{pad}(m, r)$ ▷ Pad m to a multiple of r
  $s \coloneqq 0^b$ ▷ s is initialized to b zero bits
  for $i = 0$; $i < k$; $i{+}{+}$ **do** ▷ Absorbing Phase
    $s \coloneqq s \oplus (m_i \| 0^{b-r})$
    $s \coloneqq f(s)$
  end for
  $z \coloneqq \mathsf{trunc}(s, r)$ ▷ Squeezing Phase
  while $\text{len}(z) < \ell$ **do**
    $s \coloneqq f(s)$
    $z \coloneqq z \| \mathsf{trunc}(s, r)$
  end while
  return $\mathsf{trunc}(z, \ell)$
end procedure

Here, we have $m_i \in \{0,1\}^r$. We then absorb the message into the internal state of the sponge:

$$\begin{aligned} s_0 &= 0^b \\ \hat{s}_0 &= s_0 \oplus \left(m_0 \| 0^{b-r}\right) \\ s_1 &= f(\hat{s}_0) \\ \hat{s}_1 &= s_1 \oplus \left(m_1 \| 0^{b-r}\right) \\ &\vdots \\ \hat{s}_{k-1} &= s_{k-2} \oplus \left(m_{k-1} \| 0^{b-r}\right) \\ s_k &= f(\hat{s}_{k-1}) \end{aligned} \tag{A.12}$$

Here, 0^b and 0^{b-r} mean b zero bits and $b-r$ zero bits, respectively. At this point, the sponge has completed the absorbing phase. We need to squeeze the sponge to produce the desired output. If ℓ bits of output are desired with $jr < \ell \leq (j+1)\, r$, then compute

$$\begin{aligned} z_0 &= \mathsf{trunc}(s_k, r) \\ s_{k+1} &= f(s_k) \\ z_1 &= \mathsf{trunc}(s_{k+1}, r) \\ s_{k+2} &= f(s_{k+1}) \\ &\vdots \\ z_j &= \mathsf{trunc}(s_{k+j}, r) \\ z &= \mathsf{trunc}(z_0 \| z_1 \| \cdots \| z_j, \ell) \\ H(m) &\coloneqq z. \end{aligned} \tag{A.13}$$

Here, trunc refers to returning the specified number of bits. This defines the output of a hash function based on the sponge construction.

Sponge Function Security

For a given internal state b, there is a tradeoff between rate and capacity. A larger rate leads to faster computation because there are fewer blocks to process. A larger capacity gives the sponge greater security because less information about the internal state is revealed. Generic attacks against sponge functions use at least $2^{c/2}$ operations [22, Chapter 5], so this gives a lower bound on the capacity. Thus, if k bits of security is desired, it would make sense for $c = 2k$.

Due to the fact that the entire internal state is not revealed during the squeezing phase, sponge hash functions may be made immune to length extension attacks provided the capacity is large enough; that is, if the capacity is large enough, the best method is only brute force.

Example Sponge Parameters

We now list the sponge parameters used in KECCAK and SHA-3.

Padding Differences KECCAK and SHA-3 have slightly different padding schemes:

$$\begin{aligned} \mathsf{pad}_{\text{KECCAK}}(m) &\coloneqq m \| \mathtt{10^*1} \\ &= m_0 \| m_1 \| \cdots \| m_{\ell-1} \\ \mathsf{pad}_{\text{SHA-3}}(m) &\coloneqq m \| \mathtt{0110^*1} \\ &= m_0 \| m_1 \| \cdots \| m_{\ell-1}. \end{aligned} \tag{A.14}$$

Here, we have $m_i \in \{0,1\}^r$, where r is the rate. The notation $\mathtt{0}^*$ means that we add zero or more $\mathtt{0}$ bits for the padding scheme. In the case of KECCAK, we input is appended so that the message length is a multiple of the rate r (examples are discussed next). Other SHA-3 functions have different padding schemes as well.

Rate and Capacity Examples A comparison between hash functions may be found in Table A.1. Although KECCAK is defined for a number of internal states, SHA-3 focuses only on those with internal state $b = 1600$ bits, as this was the primary submission. We recall that for sponge functions the *block size* is called the *rate*. In SHA-3, due to generic attacks, the capacity is twice the desired security level. In this way, SHA-3-256 has capacity 512 and rate 1088 while SHA-3-512 has capacity 1024 and rate 576.

A.2 Applications of Cryptographic Hash Functions

A.2.1 Hash-based Message Authentication Code (HMAC)

We now spend time discussing Hash-based Message Authentication Codes (HMAC) [10, 77].

As mentioned in Chapter 6.7, a *message authentication code* ensures message integrity. This results from using a secret key k together with a message m to produce a tag t. For message authentication codes to be useful, it should be impractical for adversary Eve to produce a valid tag even if she has seen multiple valid message-tag pairs.

Throughout this section we will work with a hash function $H : \{0,1\}^* \to \{0,1\}^b$. This specific construction is designed for hash functions based the Merkle-Damgård construction discussed in Appendix A.1.1. A message authentication code based on the sponge construction in Appendix A.1.2 will be discussed in Appendix A.2.2.

Initial (Incorrect) Guess for Hash-based MAC

We let $k \in \{0,1\}^\ell$ be our secret key; in practice, we frequently have $\text{len}(k) = b$.

One potential method for constructing a message authentication code based on a hash function would be the following: Given a message m and secret key k, compute

$$t \coloneqq H(k \| m). \tag{A.15}$$

Algorithm A.3 Hash-based Message Authentication Code

Require: Hash is a hash function with internal block size B

procedure HMAC(k, m)
  ipad $\coloneqq$ 0x3636$\cdots$36 ▷ The byte 36 is repeated B times
  opad $\coloneqq$ 0x5c5c$\cdots$5c ▷ The byte 5c is repeated B times
  if LENGTH(k) $\leq B$ **then**
    $k' \coloneqq k \| 0000 \cdots 00$ ▷ Concatenate k with byte 00 until k' has length B
  else
    $k'' \coloneqq$ HASH(k)
    $k' \coloneqq k'' \| 0000 \cdots 00$ ▷ Concatenate k'' with byte 00 until k' has length B
  end if
  $k_1 \coloneqq k' \oplus$ ipad
  $k_2 \coloneqq k' \oplus$ opad
  $t' \coloneqq$ Hash(k_1, m)
  $t \coloneqq$ Hash(k_2, t')
  return t
end procedure

While this could be used, the challenge with this method is that, if H is based on the Merkle-Damgård construction, this is susceptible to length extension attacks. Given a valid (m, t), Eve would be able to construct valid MAC tags for all messages of the form pad$(m)\|y$. *This should not be possible.*

HMAC Definition

The Hash-based Message Authentication Code (HMAC) [10, 77] was constructed so that length extension attacks are not possible. Given a key k and message m, we compute the following:

$$\begin{aligned} k_1 &\coloneqq k \oplus \mathsf{ipad} \\ k_2 &\coloneqq k \oplus \mathsf{opad} \\ t' &\coloneqq H(k_1 \| m) \\ t &\coloneqq H(k_2 \| t'). \end{aligned} \tag{A.16}$$

Here, ipad and opad are two different public byte constants which ensure that k produces to different keys for each application of the hash function: one key for the inner application of the hash function, and a second key for the outer application. The two applications of the hash function ensure that it is not possible to perform length extension attacks on the protocol. The HMAC definition can be combined to

$$\begin{aligned} t &\coloneqq \mathsf{HMAC}(k, m) \\ \mathsf{HMAC}(k, m) &\coloneqq H\left([k \oplus \mathsf{opad}] \| H\left([k \oplus \mathsf{ipad}] \| m\right)\right). \end{aligned} \tag{A.17}$$

Algorithm A.4 Keccak Message Authentication Code

Require: Hash is a hash function based on KECCAK with internal rate r

1: **procedure** KMAC(k, m)
2: $k' \coloneqq \mathsf{pad}(k, r)$ ▷ Pad k to a multiple of r bytes with zero bytes
3: $t \coloneqq \mathsf{Hash}(k', m)$
4: **return** t
5: **end procedure**

The formal specification may be found in Alg. A.3. The minimum recommended key size is length of the hash function output [77].

This scheme has been proven secure [53]. Because the MD5 and SHA-1 hash functions are compromised, HMAC-MD5 and HMAC-SHA-1 *should not be used* [72, 128]; they should *only* be used when *required* for legacy systems. Although it may be the case that HMAC-SHA-1 is still secure [100, Section 3.3], the fact remains that SHA-1 is a *compromised* hash function.

Example A.1 (HMAC-MD5)
Code may be found at `examples/app_crypto/hmac_md5.py`.

We compute the output of HMAC using MD5; see Listing A.1. We have the following key and message pair:

$$\begin{aligned}\mathsf{key} &= \mathsf{MD5}(\texttt{1313131313131313131313131313131313}) \\ &= \texttt{e885a118b42df3823f4999d4135ce08f} \\ \mathsf{msg} &= \texttt{4242424242424242424242424242424242424242424242424242424242424242}.\end{aligned} \tag{A.18}$$

The resulting value is

$$\mathsf{HMAC\text{-}MD5}\left(\mathsf{key}, \mathsf{msg}\right) = \texttt{7fc82df5f026fab8f8f880bf73a8bb08}. \tag{A.19}$$

A.2.2 Keccak Message Authentication Code (KMAC)

KECCAK-MAC is a message authentication code based on the KECCAK sponge function. While it would be possible to also use the HMAC construction from Appendix A.2.1, the KMAC construction is more efficient because it requires only one hash operation instead of two.

We now define KMAC. As before, we let k be the secret key and m be the message; we let H be a hash function based on the KECCAK sponge function with rate r. The tag is then

$$\begin{aligned}t &\coloneqq \mathsf{KMAC}(k, m) \\ &= H\left(\mathsf{pad}(k, r) \| m\right).\end{aligned} \tag{A.20}$$

Listing A.1: HMAC using MD5

```
#!/usr/bin/env python3

import hashlib
import hmac

# Set up key and message
preKey = bytes.fromhex(''.join('13' for i in range(16)))
md5 = hashlib.md5(); md5.update(preKey)
key = md5.digest()
# e885a118b42df3823f4999d4135ce08f
msg = bytes.fromhex(''.join('42' for i in range(32)))
# 42424242424242424242424242424242424242424242424242424242424242424242

# Pad key
padding = md5.block_size-len(key)
zeroPad = bytes.fromhex(''.join('00' for i in range(padding)))
keyPrime = key + zeroPad

# Derive inner and outer keys
ipad = bytes.fromhex(''.join('36' for i in range(md5.block_size)))
opad = bytes.fromhex(''.join('5c' for i in range(md5.block_size)))
key1 = bytes([a^b for (a,b) in zip(keyPrime, ipad)])
key2 = bytes([a^b for (a,b) in zip(keyPrime, opad)])

# Compute value
md5 = hashlib.md5(); md5.update(key1); md5.update(msg)
tmp = md5.digest()
md5 = hashlib.md5(); md5.update(key2); md5.update(tmp)
t = md5.digest()
# 7fc82df5f026fab8f8f880bf73a8bb08

# Correct value
hmac_md5 = hmac.new(key, digestmod=hashlib.md5)
hmac_md5.update(msg)
ret = hmac_md5.digest()

assert t == ret # Valid hmac
```

Here, $\mathsf{pad}(k, r)$ means that we pad k to a multiple of r bits. KMAC is formally specified as a derived SHA-3 function [71]. In [71], additional padding is used for domain separation. By padding to a multiple of the rate, it is possible to precompute the internal state so that the first iteration has already been processed. This would allow for faster computation if many tags need to be calculated for the same key. The formal specification may be found in Alg. A.4.

Example A.2 (KMAC using SHA-3)
Code may be found at `examples/app_crypto/kmac_sha3.py`.

We compute the output of KMAC using SHA-3; see Listing A.2. We have the following key and message pair:

$$\begin{aligned}\mathsf{key} &= \mathsf{SHA\text{-}3}(1313131313131313131313131313131313131313131313131313131313131313)\\ &= \texttt{0554a7d12c7a1bfcb47088e98851c036ca90c3e2d2b0ef6d88b341d86ec6bd4a}\\ \mathsf{msg} &= \texttt{42424242424242424242424242424242424242424242424242424242424242}.\end{aligned} \tag{A.21}$$

The resulting value is

$$\begin{aligned}&\mathsf{KMAC\text{-}SHA\text{-}3}\,(\mathsf{key}, \mathsf{msg}) =\\ &\texttt{b4033d41cef0553003b8e1b189fae0c44283a4b1870f59cdd8a8133a7d31c585}.\end{aligned} \tag{A.22}$$

A.2.3 HMAC-based Key Derivation Function (HKDF)

Here we will discuss the HMAC-based Key Derivation Function (HKDF) protocol, attempting to condense the main ideas from the primary sources [76, 78].

Purpose of Key Derivation Functions

The purpose of a *key derivation function* is to convert *nonuniform* randomness into *uniform* randomness. Nonuniform randomness may arise in a variety of places. One example is the shared secret of a Diffie-Hellman Key Exchange: the shared secret is a random group element but not a *uniformly* random bit string. Most symmetric key encryption protocols assume a *uniform* random key, so merely using the shared secret would violate the assumptions of the protocol; violating cryptographic protocol security assumptions is *always* a bad idea.

Traditional KDF Protocol

As stated in [76, Section 8], the traditional KDF scheme is given as

$$T := \mathsf{Hash}\,(\mathsf{skm}\|1\|\mathsf{ctx})\,\mathsf{Hash}\,(\mathsf{skm}\|2\|\mathsf{ctx})\,\mathsf{Hash}\,(\mathsf{skm}\|3\|\mathsf{ctx})\cdots \tag{A.23}$$

Here, skm is the source key material, ctx is the context information, and ℓ is the output in bits; The result is truncated to the appropriate length. A formal specification may be found in Alg. A.5.

Listing A.2: KMAC using SHA-3

```
#!/usr/bin/env python3

import hashlib

# Set up key and message
preKey = bytes.fromhex(''.join('13' for i in range(32)))
sha3 = hashlib.sha3_256(); sha3.update(preKey)
key = sha3.digest()
# 0554a7d12c7a1bfcb47088e98851c036ca90c3e2d2b0ef6d88b341d86ec6bd4a
msg = bytes.fromhex(''.join('42' for i in range(32)))
# 4242424242424242424242424242424242424242424242424242424242424242

# Pad key
padding = sha3.block_size-len(key)
zeroPad = bytes.fromhex(''.join('00' for i in range(padding)))
keyPrime = key + zeroPad

# Compute value
sha3 = hashlib.sha3_256(); sha3.update(keyPrime); sha3.update(msg)
t = sha3.digest()
# b4033d41cef0553003b8e1b189fae0c44283a4b1870f59cdd8a8133a7d31c585
```

Algorithm A.5 Traditional Key Derivation Function

Require: Hash is a hash function with output size n

procedure TRADITIONALKDF(skm, ctx, ℓ)
  $t \coloneqq \lceil \ell / n \rceil$
  $T \coloneqq$ "" ▷ Empty byte string
  for $i = 1$; $i \leq t$; $i{+}{+}$ **do**
    $T \coloneqq T \| \mathsf{Hash}(\mathsf{skm} \| i \| \mathsf{ctx})$
  end for
  return $\mathsf{trunc}(T, \ell)$
end procedure

Algorithm A.6 NIST Key Derivation Function (One-Step Key Derivation [4, Section 4])

Require: Hash is a hash function with output size n

procedure NIST-KDF(skm, ctx, ℓ)
  $t \coloneqq \lceil \ell / n \rceil$
  $T \coloneqq$ "" ▷ Empty byte string
  for $i = 1$; $i \leq t$; $i{+}{+}$ **do**
    $T \coloneqq T \| \mathsf{Hash}(i \| \mathsf{skm} \| \mathsf{ctx})$
  end for
  return $\mathsf{trunc}(T, \ell)$
end procedure

The NIST KDF scheme slightly modifies the traditional scheme by switching the order of the counter and the source key material:

$$T \coloneqq \mathsf{Hash}\left(1 \| \mathsf{skm} \| \mathsf{ctx}\right) \mathsf{Hash}\left(2 \| \mathsf{skm} \| \mathsf{ctx}\right) \mathsf{Hash}\left(3 \| \mathsf{skm} \| \mathsf{ctx}\right) \cdots \tag{A.24}$$

A formal specification may be found in Alg. A.6. This version (One-Step Key Derivation) is still approved by NIST [4, Section 4]. In both the Traditional and NIST KDF specification, the counter is 4 bytes (32 bits), and the maximum number of iterations is 2^{32}.

It is noted that both instances, the inputs of the hash function are highly correlated; there are only a few bits which differ between hash calls. While this is fine when using a random oracle, the hash functions used in practice are not perfect. HMAC-based Key Derivation Function was designed to counter these problems [76].

HKDF Protocol

The HMAC-based Key Derivation Function protocol is based on an *extract-then-expand* principle: nonuniform randomness is first compressed into a uniformly random key; this uniformly random key is then used to derive additional uniformly random bits. From [76], we have

$$\mathsf{HKDF}\left(\mathsf{salt}, \mathsf{skm}, \mathsf{ctx}, \ell\right) \coloneqq \mathsf{trunc}\left(T(1) \| T(2) \| \cdots \| T(t), \ell\right), \tag{A.25}$$

Algorithm A.7 HMAC-based Key Derivation Function

Require: HMAC is a Hash-based MAC which uses a hash function with output size n

procedure HKDF(salt, skm, ctx, ℓ)
  $t := \lceil \ell / n \rceil$
  $T := ""$ ▷ Empty byte string
  prk := HMAC(salt, skm)
  out := HMAC(prk, ctx$\|1$)
  $T := T \| \text{out}$
  for $i = 2$; $i \leq t$; $i{+}{+}$ **do**
    out := HMAC(prk, out$\|$ctx$\|i$)
    $T := T \| \text{out}$
  end for
  return trunc(T, ℓ)
end procedure

where salt is a salt, skm is the source key material, ctx is the context information, and ℓ is the length of output in bits. We recall that a salt is used to ensure that different source key materials produce different pseudorandom keys. Salts are closely related to nonces. From here, we define $T(i)$ by

$$\begin{aligned} \text{prk} &:= \text{HMAC}(\text{salt}, \text{skm}) \\ T(1) &:= \text{HMAC}(\text{prk}, \text{ctx} \| 1) \\ T(i) &:= \text{HMAC}(\text{prk}, T(i-1) \| \text{ctx} \| i) \quad i \in \{2, \cdots, t\}. \end{aligned} \tag{A.26}$$

The source key material (possibly with a salt) is concentrated into a pseudorandom key prk. From there, this pseudorandom key is not output directly but rather is used to derive additional pseudorandom bits. In order to help resist any correlation affects between repeated HMAC calls, the pseudorandom bits from one iteration are fed into the message for the next iteration; this helps ensure that the HMAC instantiations are independent. The context information enables the same source key material to be reused in different settings while ensuring independent keys. A formal specification may be found in Alg. A.7. In [78], the counter is 1 byte and the number of iterations are limited to 255.

We note that all of the operations here which use HMAC could also be replaced with calls to KMAC, and this is done in Example A.4 below.

As seen in Examples A.3 and A.4, calling HKDF with the same skm, salt, and ctx values but varying lengths will cause the shorter outputs to be contained within longer outputs. Although within the HMAC-based Key Derivation Function specification there is no strict requirement that the length of the output be included within the ctx information, the author of this work feels this should be encouraged if there is the slightest chance for variable-length outputs to occur in practice.

Example A.3 (HKDF using HMAC-SHA-2-256)
Code may be found at `examples/app_crypto/hkdf_hmac_sha2.py`.

We compute the output of HKDF using HMAC-SHA-2-256; see Listing A.3. We have the following source key material, salt, and context:

$$\begin{aligned}\mathsf{skm} &= \mathsf{MD5}(\texttt{01010101010101010101010101010101}) \| \texttt{00...00} \\ &= \texttt{24311d9abc4077123c2c9a167afbe7540000000000000000000000000000000} \\ \mathsf{salt} &= \texttt{02020202020202020202020202020202020202020202020202020202020202}.\end{aligned} \tag{A.27}$$

We let ctx be empty. In this instance, we are modeling the situation where some of the source key material has entropy while other parts of it do not has as much entropy. We have the following results:

$$\begin{aligned}&\mathsf{HKDF\text{-}HMAC\text{-}SHA\text{-}2}\,(\mathsf{skm}, \mathsf{salt}, \mathsf{ctx}, 16) = \\ &\texttt{209a24fb9c313a19d8ece110a922a75c} \\ &\mathsf{HKDF\text{-}HMAC\text{-}SHA\text{-}2}\,(\mathsf{skm}, \mathsf{salt}, \mathsf{ctx}, 32) = \\ &\texttt{209a24fb9c313a19d8ece110a922a75c7fdccb0ea20fff64eacf0bc621e688fe} \\ &\mathsf{HKDF\text{-}HMAC\text{-}SHA\text{-}2}\,(\mathsf{skm}, \mathsf{salt}, \mathsf{ctx}, 64) = \\ &\texttt{209a24fb9c313a19d8ece110a922a75c7fdccb0ea20fff64eacf0bc621e688fe}\backslash \\ &\texttt{9d905f252edb3b56a13609eaab6c13cd446a073ea08423c32b2fc009271d828c}.\end{aligned} \tag{A.28}$$

As we can see, the output looks pretty random.

Example A.4 (HKDF using KMAC-SHA-3-256)
Code may be found at `examples/app_crypto/hkdf_kmac_sha3.py`.

We compute the output of HKDF using KMAC-SHA-3-256; see Listing A.4. We use the same values for skm, salt, and ctx as in Example A.3. We have the following results:

$$\begin{aligned}&\mathsf{HKDF\text{-}KMAC\text{-}SHA\text{-}3}\,(\mathsf{skm}, \mathsf{salt}, \mathsf{ctx}, 16) = \\ &\texttt{f9b8dbec9e6ba6c6862cc45cfc408e0b} \\ &\mathsf{HKDF\text{-}KMAC\text{-}SHA\text{-}3}\,(\mathsf{skm}, \mathsf{salt}, \mathsf{ctx}, 32) = \\ &\texttt{f9b8dbec9e6ba6c6862cc45cfc408e0b33b125aecff2705650f191b7cb599b74} \\ &\mathsf{HKDF\text{-}KMAC\text{-}SHA\text{-}3}\,(\mathsf{skm}, \mathsf{salt}, \mathsf{ctx}, 64) = \\ &\texttt{f9b8dbec9e6ba6c6862cc45cfc408e0b33b125aecff2705650f191b7cb599b74}\backslash \\ &\texttt{02c32a28defa349720c44c449319f9cff77b211fa5262de491334409115c712d}.\end{aligned} \tag{A.29}$$

A.2.4 Mask Generation Function and Extendable Output Function

We now discuss mask generation functions and extendable output functions.

Listing A.3: HKDF using HMAC-SHA-2

```
#!/usr/bin/env python3

import hashlib
import hmac
import math

def hkdf(length: int, skm, salt: bytes = b"", ctx: bytes = b""):
    sha2 = hashlib.sha256()
    hash_len = sha2.digest_size
    if len(salt) == 0:
        salt = bytes([0] * hash_len)
    prk = hmac.new(salt, skm, digestmod=hashlib.sha256).digest()
    t = b""
    okm = b""
    for i in range(math.ceil(length / hash_len)):
        t = hmac.new(prk,
                     t + ctx + bytes([i + 1]),
                     digestmod=hashlib.sha256).digest()
        okm += t
    return okm[:length]

skmPre  = bytes.fromhex("01" * 16)
skm = hashlib.md5(skmPre).digest() + bytes.fromhex("00" * 16)
# 24311d9abc4077123c2c9a167afbe754000000000000000000000000000000000
salt = bytes.fromhex("02" * 32)
# 0202020202020202020202020202020202020202020202020202020202020202
ctx = b""

okm1 = hkdf(16, skm, salt, ctx)
# 209a24fb9c313a19d8ece110a922a75c
okm2 = hkdf(32, skm, salt, ctx)
# 209a24fb9c313a19d8ece110a922a75c7fdccb0ea20fff64eacf0bc621e688fe
okm3 = hkdf(64, skm, salt, ctx)
# 209a24fb9c313a19d8ece110a922a75c7fdccb0ea20fff64eacf0bc621e688fe\
# 9d905f252edb3b56a13609eaab6c13cd446a073ea08423c32b2fc009271d828c
```

Listing A.4: HKDF using KMAC-SHA-3

```
#!/usr/bin/env python3

import hashlib
import math

# This version of KMAC does not follow the NIST spec
def kmac(key: bytes, msg: bytes):
    sha3 = hashlib.sha3_256()
    key_len = len(key)
    assert key_len < sha3.block_size
    padding = bytes.fromhex("00" * (sha3.block_size - key_len))
    sha3.update(key); sha3.update(padding); sha3.update(msg)
    return sha3.digest()

def hkdf(length: int, skm, salt: bytes = b"", ctx: bytes = b""):
    sha3 = hashlib.sha3_256()
    hash_len = sha3.digest_size
    if len(salt) == 0:
        salt = bytes([0] * hash_len)
    prk = kmac(salt, skm)
    t = b""
    okm = b""
    for i in range(math.ceil(length / hash_len)):
        t = kmac(prk, t + ctx + bytes([i + 1]))
        okm += t
    return okm[:length]

skmPre  = bytes.fromhex("01" * 16)
skm = hashlib.md5(skmPre).digest() + bytes.fromhex("00" * 16)
# 24311d9abc4077123c2c9a167afbe754000000000000000000000000000000000
salt = bytes.fromhex("02" * 32)
# 0202020202020202020202020202020202020202020202020202020202020202
ctx = b""

okm1 = hkdf(16, skm, salt, ctx)
# f9b8dbec9e6ba6c6862cc45cfc408e0b
okm2 = hkdf(32, skm, salt, ctx)
# f9b8dbec9e6ba6c6862cc45cfc408e0b33b125aecff2705650f191b7cb599b74
okm3 = hkdf(64, skm, salt, ctx)
# f9b8dbec9e6ba6c6862cc45cfc408e0b33b125aecff2705650f191b7cb599b74\
# 02c32a28defa349720c44c449319f9cff77b211fa5262de491334409115c712d
```

Algorithm A.8 Mask Generation Function 1

Require: Hash is a hash function with output size n
1: **procedure** MGF1(seed, ℓ)
2: $t \coloneqq \lceil \ell / n \rceil$
3: $T \coloneqq$ “” ▷ Empty byte string
4: **for** $i = 0$; $i < t$; $i{+}{+}$ **do**
5: $T \coloneqq T \| \mathsf{Hash}(\mathsf{seed} \| i)$
6: **end for**
7: **return** $\mathsf{trunc}(T, \ell)$
8: **end procedure**

Hash functions are very useful, but when a random output is needed with length greater than the message digest length, something else must be done. This was needed in the case of the RSA encryption or signature schemes [35, 89]. This lead to the development of a mask generation function. The particular scheme in [89] is given here:

$$\mathsf{MGF}(\mathsf{seed}, \ell) \coloneqq \mathsf{Hash}(\mathsf{seed} \| 0)\, \mathsf{Hash}(\mathsf{seed} \| 1)\, \mathsf{Hash}(\mathsf{seed} \| 2) \cdots \tag{A.30}$$

In this case, Hash is a hash function, seed is the data to be hashed, and ℓ is the specified output length. The numbers are encoded as 32-bit unsigned integers, and the result is truncated to the desired length; thus, there is a maximum of 2^{32} iterations. One reference defining a padding scheme which uses a mask generation function is from 1998 [67].

This definition is for “MGF1”; no additional versions appear to have been specified. A formal specification may be found in Alg. A.8. This is very similar to the definition of the Traditional KDF from Alg. A.5 (the only difference being a lack of notion for context), and this also means that it has similar problems.

Later, when defining the SHA-3 standard [46], additional functions, called extendable output functions, were specified which were like hash functions but with the ability to produce arbitrarily long output [46, Section 6.2]. This is due in part to the sponge function structure of SHA-3.

The difference between mask generation functions and extendable output functions may be that a mask generation function is designed to use a hash function to produce arbitrarily long output while an extendable output function is a “hash function” designed to produce arbitrarily long output.

In summary, the mask generation function as defined in [89] *should not be used except for backward compatibility* due to the weakness of its definition. If possible, use an extendable output function such as SHAKE128 or SHAKE256 [46, 71]; otherwise, use HMAC-based Key Derivation Function with HMAC or KMAC.

A.2.5 PBKDF2

Although we have mentioned that Password-Based Key Derivation Function 2 [90] should not be used [25], we *will* describe it here in order to help the reader understand how it works and how it attempts to increase the difficulty over standard cryptographic hash functions. Even so, *do not use this method to store passwords!* It was originally designed in 2000 [66].

Algorithm A.9 Password-Based Key Derivation Function Version 2 with HMAC

Require: HMAC is a Hash-based MAC which uses a hash function with output size n

procedure PBKDF2(pw, salt, c, ℓ)
  $t \coloneqq \lceil \ell / n \rceil$
  $T \coloneqq$ "" ▷ Empty byte string
  for $i = 1$; $i \leq t$; $i{+}{+}$ **do**
    $U_1 \coloneqq \mathsf{HMAC}(\mathsf{pw}, \mathsf{salt} \| i)$
    for $k = 2$; $k \leq c$; $k{+}{+}$ **do**
      $U_k \coloneqq \mathsf{HMAC}(\mathsf{pw}, U_{k-1})$
    end for
    $\mathsf{out} \coloneqq U_1 \oplus U_2 \oplus \cdots \oplus U_c$
    $T \coloneqq T \| \mathsf{out}$
  end for
  return $\mathsf{trunc}(T, \ell)$
end procedure

One of the difficulties of protecting passwords with standard cryptographic hash functions is that specialized hardware can be designed to quickly compute them. To slow down specialized hardware like application-specific integrated circuits (ASICs)[1], it is necessary to add computations which *must* be performed sequentially.

We now give the specification from [90]; we use HMAC as the required pseudorandom function:

$$\mathsf{PBKDF2}\left(\mathsf{pw}, \mathsf{salt}, c, \ell\right) \coloneqq T_1 \| T_2 \| \cdots \| T_k. \tag{A.31}$$

Here, pw is the password, salt is the salt, c is the number of iterations, and ℓ is the derived key output length.

We have

$$T_i \coloneqq U_{1,i} \oplus U_{2,i} \oplus \cdots \oplus U_{c,i} \tag{A.32}$$

where

$$\begin{aligned} U_{1,i} &\coloneqq \mathsf{HMAC}\left(\mathsf{pw}, \mathsf{salt} \| i\right) \\ U_{2,i} &\coloneqq \mathsf{HMAC}\left(\mathsf{pw}, U_{1,i}\right) \\ U_{3,i} &\coloneqq \mathsf{HMAC}\left(\mathsf{pw}, U_{2,i}\right) \\ &\vdots \\ U_{c,i} &\coloneqq \mathsf{HMAC}\left(\mathsf{pw}, U_{c-1,i}\right). \end{aligned} \tag{A.33}$$

This feed-forward scheme is similar to what we saw in the HMAC-based Key Derivation Function scheme in Appendix A.2.3. Greater computation can be required by having a large

[1] https://en.wikipedia.org/wiki/Application-specific_integrated_circuit

c value, as these require many *sequential* HMAC iterations. This causes a slowdown in the total operation. A formal specification may be found in Alg. A.9.

We note that although this is a good starting place for password storage, recent work has shown that the protection PBKDF2 offers is insufficient against dedicated attackers [25]. This is because it is possible to build dedicated hardware to efficiently compute PBKDF2. In particular, we note that the T_i values could be built up iteratively, so all the $U_{j,i}$ values do not need to be stored. As such, the algorithm does not have large memory requirements. *Do not use this algorithm to store passwords!*

A.2.6 Hash to Finite Field

Here we will work out the details proving the bounds for hashing to finite field.

Suppose we have a random oracle $H : \{0,1\}^* \to \mathbb{Z}_N$. We are interested in knowing how much $H(m) \mod p$ deviates from the uniform distribution.

We suppose that $N > p$ and $p \nmid N$; that is, p is not a divisor of N. In this case, find q and r such that

$$N = qp + r. \tag{A.34}$$

Here, we have $q \geq 1$ and $0 \leq r < p$. We have $r \geq 1$ because $p \nmid N$.

We let X be a random variable uniformly distributed on $\mathbb{Z}_N$ and $X_p \coloneqq X \mod p$. Then we have the following:

$$\begin{aligned} \mathcal{P}\left(X_p = \ell\right) &= \frac{q+1}{N}, \quad \ell \in \{0, \cdots, r-1\} \\ \mathcal{P}\left(X_p = \ell\right) &= \frac{q}{N}, \quad \ell \in \{r, \cdots, N-1\}. \end{aligned} \tag{A.35}$$

Here, $\mathcal{P}$ is the probability of an event occurring. Thus, $\mathcal{P}(X_p = \ell)$ denotes the probability that the random variable X_p equals the value ℓ.

We now determine how far X_p deviates from the uniform distribution U_p:

$$\begin{aligned} \Delta(X_p, U_p) &\coloneqq \sum_{\ell=0}^{p-1} |\mathcal{P}(X_p = \ell) - \mathcal{P}(U_p = \ell)| \\ &= \sum_{\ell=0}^{r-1} \left| \frac{q+1}{N} - \frac{1}{p} \right| + \sum_{\ell=r}^{p-1} \left| \frac{q}{N} - \frac{1}{p} \right| \\ &= r\frac{qp + p - N}{pN} + (p-r)\frac{N - qp}{pN} \\ &= \frac{2r(p-r)}{pN} \\ &\leq \frac{p}{2N}. \end{aligned} \tag{A.36}$$

Thus, the relative size of p and N controls the deviation from uniformity. In this way, if the p is n-bit prime number and we desire k bits of security, then we need $N \geq 2^{n+k}$. After

hashing into $\mathbb{Z}_N$, performing a reduction modulo p will result in a sufficiently uniform hash distribution.

In some sense, we are free in how we choose N provided it is sufficiently large. Because we have hash functions which output bits, it would be easy to hash to some power of 2. In practice, our prime p will be somewhat large; generally, at least 200 bits, and usually 256 bits or more. If we want at least 128 bits of security, then a standard 256-bit hash function will not be sufficient.

Because of this, using the HMAC-based Key Derivation Function previously discussed in Appendix A.2.3 appears to be a reasonable decision. Although we are not interested in keys *per se*, we are looking to map random inputs to random outputs. These outputs will then be deterministically mapped to points on an elliptic curve.

Appendix B

Additional Mathematics

Here is material which was deemed not required for the primary purpose of this text, yet the author felt some individuals may find it useful. It also will fill out some of the details which were glossed over because they were not strictly necessary.

The mathematical level of the different topics will vary. Even so, the goal is that this material will still be accessible for more determined readers. The material here should be considered a brief introduction. A more thorough book on number theory may be found here [119].

B.1 The Real Numbers

When listing off the standard symbols in Chapter 3.1.2, we defined the real numbers $\mathbb{R}$ with

$$\mathbb{R} \coloneqq \{\text{the set of all real numbers}\}, \tag{B.1}$$

while the other sets were more specific. That is, we never took time to *actually define* the real numbers.

The reason for this is that it is *not possible* to define the real numbers without an advanced mathematical discussion. This is because the property which distinguishes the real numbers $\mathbb{R}$ from the rational numbers $\mathbb{Q}$ is the *least-upper-bound property*: every nonempty set of real numbers which is bounded above has a least upper bound. The rationals do not have the least-upper-bound property; as a result, there are "gaps" in the rationals. Some of the numbers "missing" from the rationals are $\sqrt{2}$, π, and e; these numbers are *irrational*. The reals *do* have the least upper bound property, so there are no "gaps"; it contains the rational and irrational numbers.

A rigorous discussion of the properties of real numbers is required before attempting to prove all the results of calculus. All of this is done in a mathematics course usually called *Real Analysis*.

B.2 Set Theory

We now spend a bit more time discussing set theory and set operations. We will need the following result to prove two sets are equal:

Theorem B.1 (Set Equality)
For sets A and B, we have $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.

Proof. Let us suppose that $A = B$, then for $x \in A$ we have $x \in B$ so $A \subseteq B$. Similarly, for $x \in B$ we have $x \in A$ so $B \subseteq A$.

If $A \neq B$, then they do not contain the same elements. Thus, there is either some $x \in A$ such that $x \notin B$ (implying that $A \not\subseteq B$) or some $x \in B$ such that $x \notin A$ (implying that $B \not\subseteq A$). In either case, we either have $A \not\subseteq B$ or $B \not\subseteq A$. □

B.2.1 Additional Set Operations

From Eq. (3.3), we recall

$$\begin{aligned} A \cup B &\coloneqq \{x \mid x \in A \text{ or } x \in B\} \\ A \cap B &\coloneqq \{x \mid x \in A \text{ and } x \in B\} \\ A \setminus B &\coloneqq \{a \in A \mid a \notin B\}. \end{aligned} \tag{B.2}$$

These are not the only set operations, though.

First, we say that X is a *universal set* if all of the sets we will work with will be subsets of X. When working with a universal set X and a set $A \subseteq X$, we have the *set complement* or *complement*:

$$A^c \coloneqq \{x \in X \mid x \notin A\}. \tag{B.3}$$

This may also be written in terms of the set minus operation:

$$X \setminus A = \{x \in X \mid x \notin A\}. \tag{B.4}$$

The complement notation A^c may be preferred to the set minus notation $X \setminus A$ when there are many set operations in the same expression.

For $A \subseteq X$, we always have

$$\begin{aligned} A \cup A^c &= X \\ A \cap A^c &= \emptyset. \end{aligned} \tag{B.5}$$

Example B.2 (Set Complements)
We let our universal set be the set of natural numbers $\mathbb{N}$. In this case, we define the sets

$$\begin{aligned} A &= \{0, 1, 2, 3, 4\} \\ B &= \{1, 2, 3, \cdots\}. \end{aligned} \tag{B.6}$$

In this case, we see

$$\begin{aligned} A^c &= \{5, 6, 7, \cdots\} \\ B^c &= \{0\}. \end{aligned} \tag{B.7}$$

We can also take unions and intersections in a more general way. Specifically, let $\mathcal{A}$ be a nonempty indexing set; that is, $\mathcal{A}$ is a set which will be used to index other sets. Given a collection of sets $\{A_\alpha\}_{\alpha\in\mathcal{A}}$, we define

$$\begin{aligned}\bigcup_{\alpha\in\mathcal{A}} A_\alpha &\coloneqq \{x \mid x \in \mathcal{A}_\alpha \text{ for some } \alpha \in \mathcal{A}\}\\ \bigcap_{\alpha\in\mathcal{A}} A_\alpha &\coloneqq \{x \mid x \in \mathcal{A}_\alpha \text{ for all } \alpha \in \mathcal{A}\}\,.\end{aligned} \tag{B.8}$$

These are generalizations of the standard union and intersection operations.

Example B.3 (Set Unions and Intersections)
We now look at some set unions and intersections. First, we will let $\mathbb{N}$ be our indexing set. We see

$$\bigcup_{k\in\mathbb{N}} \{k\} = \mathbb{N}. \tag{B.9}$$

In this case, we are just taking the union of the set which contains each natural number; naturally, this is then just the natural numbers $\mathbb{N}$.

We also see

$$\bigcap_{k\in\mathbb{N}} \{k\} = \emptyset. \tag{B.10}$$

This example shows that, instead of taking the union, we take the intersection of all the sets which contain each natural number, the result is the empty set.

A more involved example is this:

$$\bigcap_{k\in\mathbb{N}} \{k, k+1, k+2, \cdots\} = \emptyset. \tag{B.11}$$

This shows that if you take the of natural numbers starting at k for each $k \in \mathbb{N}$ and then intersect them all, the resulting set contains no natural numbers. This follows from the fact that there is no largest natural number.

Intersections need not always be empty, as this next example shows:

$$\bigcap_{k\in\mathbb{N}} \{-k, -k+1, -k+2, \cdots, k-2, k-1, k\} = \{0\}\,. \tag{B.12}$$

B.2.2 Relationships Between Set Operations

We now discuss a very useful result: De Morgan's Laws.

Theorem B.4 (De Morgan's Laws)
Given a nonempty index set $\mathcal{A}$, we have the following:

$$\begin{aligned}\left[\bigcup_{\alpha\in\mathcal{A}} A_\alpha\right]^c &= \bigcap_{\alpha\in\mathcal{A}} A_\alpha^c \\ \left[\bigcap_{\alpha\in\mathcal{A}} A_\alpha\right]^c &= \bigcup_{\alpha\in\mathcal{A}} A_\alpha^c.\end{aligned} \tag{B.13}$$

This result is useful because it helps us rearrange set operations into forms which are more amenable.

In the case where we just have one intersection and union, De Morgan's Laws reduce to the following:

$$\begin{aligned}[A\cup B]^c &= A^c\cap B^c \\ [A\cap B]^c &= A^c\cup B^c.\end{aligned} \tag{B.14}$$

Proof. We will use Theorem B.1 in our proof.

Let

$$x\in\left[\bigcup_{\alpha\in\mathcal{A}} A_\alpha\right]^c. \tag{B.15}$$

This implies that

$$x\notin\bigcup_{\alpha\in\mathcal{A}} A_\alpha, \tag{B.16}$$

from which it follows that $x\notin A_\alpha$ for all $\alpha\in\mathcal{A}$. Therefore we have

$$x\in\bigcap_{\alpha\in\mathcal{A}} A_\alpha^c. \tag{B.17}$$

Because x was arbitrary, we have

$$\left[\bigcup_{\alpha\in\mathcal{A}} A_\alpha\right]^c\subseteq\bigcap_{\alpha\in\mathcal{A}} A_\alpha^c. \tag{B.18}$$

We now suppose

$$x\in\bigcap_{\alpha\in\mathcal{A}} A_\alpha^c. \tag{B.19}$$

This means that $x\in A_\alpha^c$ for all $\alpha\in\mathcal{A}$. It follows that $x\notin A_\alpha$ for all $\alpha\in\mathcal{A}$. Thus, we have

$$x\notin\bigcup_{\alpha\in\mathcal{A}} A_\alpha. \tag{B.20}$$

This is equivalent to

$$x \in \left[\bigcup_{\alpha\in\mathcal{A}} A_\alpha\right]^c. \tag{B.21}$$

Because x was arbitrary, we have

$$\bigcap_{\alpha\in\mathcal{A}} A_\alpha^c \subseteq \left[\bigcup_{\alpha\in\mathcal{A}} A_\alpha\right]^c. \tag{B.22}$$

Taking Eqs. (B.18) and (B.22) along with Theorem B.1 shows us that

$$\left[\bigcup_{\alpha\in\mathcal{A}} A_\alpha\right]^c = \bigcap_{\alpha\in\mathcal{A}} A_\alpha^c. \tag{B.23}$$

This completes the proof of the first equality. The second is left as an exercise to the reader. □

B.3 Number Theory

We now have a longer discussion related to greatest common divisors and how to compute them using the Euclidean Algorithm. We also talk about the Extended Euclidean Algorithm.

B.3.1 Euclidean Division

We will need the following properties of the natural numbers and integers:

Proposition B.5 (Well-Ordering Principle)
Every nonempty set of natural numbers contains a minimal element.

Proposition B.6 (No Zero Divisors)
If $a, b \in \mathbb{Z}$ and $ab = 0$, then $a = 0$ or $b = 0$.

In our discussion about number theory, we have repeatedly used the following result:

Proposition B.7 (Euclidean Division)
Given $a, b \in \mathbb{N}$ with $b \geq 1$, there are *unique* values $q, r \in \mathbb{N}$ such that

$$a = qb + r. \tag{B.24}$$

Here, q is called the *quotient* and r is called the *remainder*.

We give a formal proof now. We will use Propositions B.5 and B.6.

Proof of Proposition B.7. If $a = 0$, then we see that we must have $q = 0$ and $r = 0$.

Thus, we assume that $a \geq 1$. Define

$$S \coloneqq \{a - qb \mid q \in \mathbb{N} \text{ and } (a - qb \geq 0)\}. \tag{B.25}$$

That is, S contains all values of the form $a - qb$ which are nonnegative for $q \in \mathbb{N}$. Because $a \geq 1$, $0 \in \mathbb{N}$, and

$$a = a - 0 \cdot b, \tag{B.26}$$

we have that $a \in S$. Thus, S is nonempty.

As defined, S is a nonempty collection of natural numbers. We set $r \coloneqq \min S$; such a minimum exists by Proposition B.5. Now, $r \in S$, so we see that we have

$$a = qb + r \tag{B.27}$$

for some $q \in \mathbb{N}$.

If $r \geq b$, then it would follow that

$$\begin{aligned} a &= qb + r \\ &= (qb + b) + (r - b) \\ &= (q + 1)\, b + \bar{r}. \end{aligned} \tag{B.28}$$

Here, we have $\bar{r} = r - b \geq 0$. Thus, we see that if this were possible, then $\bar{r} \in S$ and $\bar{r} < r$. By the minimality of r, this is not possible. Thus, we have $0 \leq r < b$. This finishes the proof of existence.

We now prove uniqueness. First, we suppose that

$$\begin{aligned} a &= qb + r \\ a &= q'b + r' \end{aligned} \tag{B.29}$$

for $q, q' \in \mathbb{N}$, $0 \leq r < b$, and $0 \leq r' < b$. This implies that

$$(q - q')\, b = r - r'. \tag{B.30}$$

Thus, we see that $b \mid r - r'$. Now, $0 \leq r' < b$ implies that $0 \geq -r' > -b$, so it follows that

$$b > r \geq r - r' > r - b \geq -b. \tag{B.31}$$

By combining these inequalities, we find

$$|r - r'| < b. \tag{B.32}$$

However, we also know that $b \mid r - r'$, so this implies that $r - r' = 0$. Thus, we have

$$(q - q')\, b = 0. \tag{B.33}$$

Because $b \neq 0$, we have $q - q' = 0$; this follows from Proposition B.6. This finishes the proof of uniqueness. □

We note that while we stated Proposition B.7 as applying for $a \geq 0$ and $b \geq 1$, we could have modified it to the more general situation when $a \in \mathbb{Z}$, $b \neq 0$, and $0 \leq r < |b|$. Even so, it is not of much additional value, and the proof is essentially the same.

Algorithm B.1 Perform Euclidean division

Require: $a \geq 0$, $b \geq 1$
1: **procedure** EUCLIDEANDIVISION(a, b)
2: $q \coloneqq \lfloor a/b \rfloor$
3: $r \coloneqq a - qb$
4: **return** q, r
5: **end procedure**

Algorithm B.2 Compute the greatest common divisor

Require: $a \geq b \geq 1$
1: **procedure** EUCLIDEANALGORITHM(a, b)
2: $r_0 \coloneqq a$
3: $r_1 \coloneqq b$
4: q_1, $r_2 \coloneqq$ EUCLIDEANDIVISION(r_0, r_1)
5: $k \coloneqq 2$
6: **while** $r_k \neq 0$ **do**
7: q_k, $r_{k+1} \coloneqq$ EUCLIDEANDIVISION(r_{k-1}, r_k)
8: $k \coloneqq k + 1$
9: **end while**
10: $d \coloneqq r_{k-1}$ ▷ Greatest Common Divisor
11: **return** d
12: **end procedure**

B.3.2 Euclidean Algorithm

To compute the greatest common divisor (gcd), we use the Euclidean Algorithm; see Alg. B.2. This is an efficient algorithm.

We will prove that it terminates, bound the number of iterations, and prove that it converges to the gcd.

Proof that Alg. B.2 produces the Greatest Common Divisor. We will start by proving convergence before we bound the number of iterations.

Proof of Convergence

First, we see that

$$r_0 \geq r_1 > r_2 > \cdots \tag{B.34}$$

with $r_k \geq 0$. This shows that $\{r_k\}$ is a strictly decreasing sequence of natural numbers (after r_0) bounded below by 0; thus, r_k will reach 0 in at most b steps. This shows the algorithm will eventually terminate.

At each step, we have

$$r_{k-1} = q_k r_k + r_{k+1}. \tag{B.35}$$

At the end, we have

$$r_{N-1} = q_N r_N. \tag{B.36}$$

We set

$$d \coloneqq r_N, \tag{B.37}$$

and we will prove that d is the gcd.

We notice that $d \mid r_{N-1}$. We also see

$$\begin{aligned} r_{N-2} &= q_{N-1} r_{N-1} + r_N \\ &= (q_{N-1} q_N + 1)\, d; \end{aligned} \tag{B.38}$$

thus, we also have $d \mid r_{N-2}$. By continuing the same logic, we find that $d \mid r_k$ for all $k \in \{0, 1, \cdots, N\}$. In particular, we have $d \mid r_0$ and $d \mid r_1$. Written another way, we have $d \mid a$ and $d \mid b$. It follows that d is a *common divisor* of a and b.

Let $\ell \geq 0$ be any other divisor of a and b; in particular, suppose

$$\begin{aligned} a &= \alpha \ell \\ b &= \beta \ell. \end{aligned} \tag{B.39}$$

Because $r_0 = a$ and $r_1 = b$, we have that $\ell \mid r_0$ and $\ell \mid r_1$. We know

$$a = q_1 b + r_2. \tag{B.40}$$

Rearranging this, we see

$$\begin{aligned} r_2 &= a - q_1 b \\ &= \ell\,(\alpha - q_1 \beta)\,. \end{aligned} \tag{B.41}$$

This implies that we also have $\ell \mid r_2$. Continuing this logic, we see that $\ell \mid r_k$ for all $k \in \{0, 1, 2, \cdots, N\}$. It follows that $\ell \mid d$ because $d = r_N$. Because $\ell \mid d$, we know that $\ell \leq d$. Thus, we see that d is greater than or equal to *all* divisors of a and b; it is thus the *greatest common divisor*.

Bounding the number of iterations

We already showed that we will converge in at most b steps. We now produce a much better bound.

To do this, we need to look at the general recurrence relation more closely:

$$r_k = q_{k+1} r_{k+1} + r_{k+2}. \tag{B.42}$$

We assume that $r_k \geq r_{k+1}$. We always have $0 \leq r_{k+2} < r_{k+1}$, but we can say more: we always have

$$r_{k+2} < \frac{r_k}{2}. \tag{B.43}$$

We consider two cases:

- Case 1: $r_{k+1} > \frac{r_k}{2}$

  In this case, we see

  $$0 \leq r_k - r_{k+1} < \frac{r_k}{2}. \tag{B.44}$$

  This equation combined with $r_{k+1} > \frac{r_k}{2}$ shows us that we have

  $$0 \leq r_k - r_{k+1} < r_{k+1}. \tag{B.45}$$

  This means that $q_{k+1} = 1$. Thus, we have

  $$\begin{aligned} r_k &= r_{k+1} + (r_k - r_{k+1}) \\ &= r_{k+1} + r_{k+2} \end{aligned} \tag{B.46}$$

  with

  $$r_{k+2} < \frac{r_k}{2}. \tag{B.47}$$

- Case 2: $r_{k+1} \leq \frac{r_k}{2}$

  Because we always have

  $$r_{k+2} < r_{k+1} \tag{B.48}$$

  and we are assuming $r_{k+1} \leq \frac{r_k}{2}$, we see that

  $$r_{k+2} < \frac{r_k}{2}. \tag{B.49}$$

  By repeatedly applying this fact, we arrive at the bound

  $$r_{2k} < \frac{r_0}{2^k}. \tag{B.50}$$

We recall that we always have $r_N \geq 1$ and $r_{N+1} = 0$. If $N = 2k$, then we see

$$r_N < \frac{r_0}{2^k}. \tag{B.51}$$

The fact that $r_N \geq 1$ allows us to write

$$1 < \frac{r_0}{2^k}. \tag{B.52}$$

Because $N = 2k$, it follows that

$$N < 2 \log_2 r_0. \tag{B.53}$$

If $N = 2k + 1$, then we have

$$r_N < r_{2k} < \frac{r_0}{2^k}. \tag{B.54}$$

We then see

$$N < 2 \log_2 r_0 + 1. \tag{B.55}$$

Thus, we always have

$$N \leq 1 + 2 \log_2 r_0. \tag{B.56}$$

This shows that we need N steps for $N \leq 1 + 2 \log_2 a$ when $a \geq b \geq 0$. □

From our discussion of computational complexity, we have just shown that Alg. B.2 is a polynomial time algorithm to compute the gcd.

Example B.8 (Using the Euclidean Algorithm)
Code may be found at `examples/app_math/euclidean_alg.py`.

We now use Alg. B.2 to work through Example 3.22: Compute gcd $(1080, 1872)$. We have

$$\begin{aligned} r_0 &= 1872 \\ r_1 &= 1080. \end{aligned} \tag{B.57}$$

We see

$$\begin{aligned} 1872 &= 1 \cdot 1080 + 792 \\ q_1 &= 1 \\ r_2 &= 792. \end{aligned} \tag{B.58}$$

In the next step, we have

$$\begin{aligned} 1080 &= 1 \cdot 792 + 288 \\ q_2 &= 1 \\ r_3 &= 288. \end{aligned} \tag{B.59}$$

The following step gives

$$\begin{aligned} 792 &= 2 \cdot 288 + 216. \\ q_3 &= 2 \\ r_4 &= 216. \end{aligned} \tag{B.60}$$

Continuing on, we find

$$\begin{aligned} 288 &= 1 \cdot 216 + 72 \\ q_4 &= 1 \\ r_5 &= 72. \end{aligned} \tag{B.61}$$

At last we have

$$\begin{aligned} 216 &= 3 \cdot 72 \\ q_5 &= 3 \\ r_6 &= 0. \end{aligned} \tag{B.62}$$

The process has terminated and we have determined that

$$\gcd(1080, 1872) = 72. \tag{B.63}$$

B.3.3 Extended Euclidean Algorithm

We also include the Extended Euclidean Algorithm; see Alg. B.3. This algorithm produces the gcd d as well as x and y such that

$$ax + by = d. \tag{B.64}$$

In this case, x and y are called *Bézout coefficients*, and Eq. (B.64) is called the *Bézout identity*.

Proof that Alg. B.3 produces $ax + by = \gcd(a, b)$. We see that this is an extension of Alg. B.2; thus, we know that $d = \gcd(a, b)$.

We begin by showing

$$as_k + bt_k = r_k \tag{B.65}$$

for all k. First, we see that it holds for $k = 0$ and $k = 1$:

$$\begin{aligned} a \cdot 1 + b \cdot 0 &= a \\ a \cdot 0 + b \cdot 1 &= b. \end{aligned} \tag{B.66}$$

We suppose that the equation is valid for r_k and r_{k-1}. Then, we see

$$\begin{aligned} r_{k+1} &= r_{k-1} - q_k r_k \\ &= (as_{k-1} + bt_{k-1}) - q_k (as_k + bt_k) \\ &= a(s_{k-1} - q_k s_k) + b(t_{k-1} - q_k t_k) \\ &= as_{k+1} + bt_{k+1}. \end{aligned} \tag{B.67}$$

Algorithm B.3 Solve $ax + by = \gcd(a, b)$ for x and y

Require: $a \geq b \geq 0$

procedure ExtendedEuclideanAlgorithm(a, b)
  $r_0 \coloneqq a$
  $s_0 \coloneqq 1$
  $t_0 \coloneqq 0$
  $r_1 \coloneqq b$
  $s_1 \coloneqq 0$
  $t_1 \coloneqq 1$
  $q_1, r_2 \coloneqq \text{EuclideanDivision}(r_0, r_1)$
  $s_2 \coloneqq s_0 - q_1 s_1$
  $t_2 \coloneqq t_0 - q_1 t_1$
  $k \coloneqq 2$
  while $r_k \neq 0$ **do**
    $q_k, r_{k+1} \coloneqq \text{EuclideanDivision}(r_{k-1}, r_k)$
    $s_{k+1} \coloneqq s_{k-1} - q_k s_k$
    $t_{k+1} \coloneqq t_{k-1} - q_k t_k$
    $k \coloneqq k + 1$
  end while
  $d \coloneqq r_{k-1}$ ▷ Greatest Common Divisor
  $x \coloneqq s_{k-1}$
  $y \coloneqq t_{k-1}$
  return d, x, y
end procedure

Thus, it also holds for r_{k+1}. Continuing this argument, we have

$$as_k + bt_k = r_k \tag{B.68}$$

for all $k \in \{0, 1, \cdots, N\}$.

We now focus on the final case:

$$as_N + bt_N = r_N. \tag{B.69}$$

We already know that $d = r_N = \gcd(a, b)$. With $x = s_N$ and $y = t_N$, we have

$$ax + by = d, \tag{B.70}$$

as desired. □

Example B.9 (Using the Extended Euclidean Algorithm)
Code may be found at `examples/app_math/extended_euclidean_alg.py`.

We now use Alg. B.3. to work through Example 3.23: Compute x and y such that

$$1080x + 1872y = \gcd(1080, 1872). \tag{B.71}$$

We will reuse information we already computed in Example B.8.

We start by listing the initial values:

$$\begin{aligned} r_0 &= 1872 \\ s_0 &= 1 \\ t_0 &= 0 \\ r_1 &= 1080 \\ s_1 &= 0 \\ t_1 &= 1. \end{aligned} \tag{B.72}$$

We see

$$\begin{aligned} 1872 &= 1 \cdot 1080 + 792 \\ q_1 &= 1 \\ r_2 &= 792 \\ s_2 &= s_0 - q_1 s_1 \\ &= 1 \\ t_2 &= t_0 - q_1 t_1 \\ &= -1. \end{aligned} \tag{B.73}$$

In the next step, we have

$$\begin{aligned} 1080 &= 1 \cdot 792 + 288 \\ q_2 &= 1 \\ r_3 &= 288 \\ s_3 &= s_1 - q_2 s_2 \\ &= -1 \\ t_3 &= t_1 - q_2 t_2 \\ &= 2. \end{aligned} \tag{B.74}$$

The following step gives

$$\begin{aligned} 792 &= 2 \cdot 288 + 216. \\ q_3 &= 2 \\ r_4 &= 216 \\ s_4 &= s_2 - q_3 s_3 \\ &= 3 \\ t_4 &= t_2 - q_3 t_3 \\ &= -5. \end{aligned} \tag{B.75}$$

Continuing on, we find

$$\begin{aligned} 288 &= 1 \cdot 216 + 72 \\ q_4 &= 1 \\ r_5 &= 72 \\ s_5 &= s_3 - q_4 s_4 \\ &= -4 \\ t_5 &= t_3 - q_4 t_4 \\ &= 7. \end{aligned} \tag{B.76}$$

At last we have

$$\begin{aligned} 216 &= 3 \cdot 72 \\ q_5 &= 3 \\ r_6 &= 0. \end{aligned} \tag{B.77}$$

This concludes Alg. B.3.

The process has terminated and we have determined that

$$\gcd(1872, 1080) = 72 \tag{B.78}$$

and

$$1872 \cdot (-4) + 1080 \cdot 7 = \gcd(1872, 1080). \tag{B.79}$$

B.4 Finite Fields

Throughout this section, we will be looking at the finite field $\mathbb{F}_p$. We have the following theorems:

Theorem B.10 (Fermat's Little Theorem [91, Lemma 2.1.16])
For all $a \in \mathbb{F}_p^*$, we have

$$a^{p-1} = 1 \mod p. \tag{B.80}$$

Theorem B.11 ($\mathbb{F}_p^*$ is cyclic [91, Theorem 2.1.37])
Let p be prime. Then $\left(\mathbb{F}_p^*, \cdot\right)$ is a cyclic group of order $p - 1$.

B.4.1 Generating Primitive Elements

Given the fact that $\mathbb{F}_p^*$ is a cyclic group, we have the following definition:

Algorithm B.4 Compute a primitive element of a finite field

Require: p prime

procedure ComputeFiniteFieldGenerator(p)
  $q \coloneqq p - 1$
  $p_1^{n_1} p_2^{n_2} \cdots p_\ell^{n_\ell} \coloneqq \text{Factor}(q)$ ▷ Compute the prime factorization of $q = p - 1$
  while **do**
    $\alpha \xleftarrow{\$} \mathbb{F}_p^*$ ▷ Choose a random element in $\mathbb{F}_p^*$
    `IsGenerator` $\coloneqq$ `true`
    for $k \coloneqq 1; k \leq \ell; k{+}{+}$ **do**
      $q_k \coloneqq q/p_k$
      $\beta \coloneqq \alpha^{q_k}$
      if $\beta = 1$ **then**
        `IsGenerator` $\coloneqq$ `false`
        break
      end if
    end for
    if `IsGenerator` **then**
      break
    end if
  end while
  return α
end procedure

Definition B.1 (Primitive Element)**.** We say $g \in \mathbb{F}_p^*$ is a *primitive element* if g is a generator of $\mathbb{F}_p^*$.

Theorem B.11 does not *specify* a generator for $\mathbb{F}_p^*$; it merely states that one *exists*. In Chapter 13.2, we stated without proof that elements of $\mathbb{F}_p$ were primitive elements. We will now give an algorithm to construct such generators; see Alg. B.4.

Proving that Alg. B.4 produces a generator for $\mathbb{F}_p^*$ relies on this proposition:

Proposition B.12 (Generators of Finite Cyclic Groups)
Let G a finite cyclic group of order q with

$$q = p_1^{n_1} p_2^{n_2} \cdots p_\ell^{n_\ell}. \tag{B.81}$$

Then $h \in G$ is a generator if and only if

$$h^{q/p_k} \neq 1, \quad k \in \{1, \cdots, \ell\}. \tag{B.82}$$

To use Alg. B.4, we must compute the prime factorization of $p-1$. From here, we choose random elements until one satisfies the necessary constraints.

Example B.13 (Computing Finite Field Generators 1)
Code may be found at `examples/app_math/finite_field_generator_1.py`.

For the first example, we will compute some generators of $\mathbb{F}_{7919}^*$; we have used this group with generator 7 many times.

We start with

$$p = 7919. \tag{B.83}$$

We also see that

$$\begin{aligned} p - 1 &= 7918 \\ &= 2 \cdot 37 \cdot 107. \end{aligned} \tag{B.84}$$

We set

$$\begin{aligned} p_1 &= 2 \\ p_2 &= 37 \\ p_3 &= 107. \end{aligned} \tag{B.85}$$

We know that 2, 37, and 107 are prime numbers.

Instead of randomly generating elements, we will start at 2 and proceed forward. For ease of notation, we set

$$\begin{aligned} q_1 &= \frac{(p-1)}{p_1} \\ &= 3959 \\ q_2 &= \frac{(p-1)}{p_2} \\ &= 214 \\ q_3 &= \frac{(p-1)}{p_3} \\ &= 74. \end{aligned} \tag{B.86}$$

- 2: We see

$$\begin{aligned} 2^{q_1} \mod p &= 1 \\ 2^{q_2} \mod p &= 5823 \\ 2^{q_3} \mod p &= 6451. \end{aligned} \tag{B.87}$$

Because $2^{q_1} = 1$, 2 is not a generator of $\mathbb{F}_p^*$.

- 3: We see

$$\begin{aligned} 3^{q_1} &\mod p = 1 \\ 3^{q_2} &\mod p = 4930 \\ 3^{q_3} &\mod p = 4828. \end{aligned} \tag{B.88}$$

  Because $3^{q_1} = 1$, 3 is not a generator of $\mathbb{F}_p^*$.

- 4: We see

$$\begin{aligned} 4^{q_1} &\mod p = 1 \\ 4^{q_2} &\mod p = 6090 \\ 4^{q_3} &\mod p = 1056. \end{aligned} \tag{B.89}$$

  Because $4^{q_1} = 1$, 4 is not a generator of $\mathbb{F}_p^*$.

- 5: We see

$$\begin{aligned} 5^{q_1} &\mod p = 1 \\ 5^{q_2} &\mod p = 6 \\ 5^{q_3} &\mod p = 48. \end{aligned} \tag{B.90}$$

  Because $5^{q_1} = 1$, 5 is not a generator of $\mathbb{F}_p^*$.

- 6: We see

$$\begin{aligned} 6^{q_1} &\mod p = 1 \\ 6^{q_2} &\mod p = 1015 \\ 6^{q_3} &\mod p = 1. \end{aligned} \tag{B.91}$$

  Because $6^{q_1} = 1$, 6 is not a generator of $\mathbb{F}_p^*$.

- 7: We see

$$\begin{aligned} 7^{q_1} &\mod p = 7918 \\ 7^{q_2} &\mod p = 755 \\ 7^{q_3} &\mod p = 5549. \end{aligned} \tag{B.92}$$

  Because $7^{q_i} \neq 1$ for all i, 7 is a generator of $\mathbb{F}_p^*$.

Thus, we were able to find 1 generator in the first 6 possibilities. We note that the next generator is 13.

Example B.14 (Computing Finite Field Generators 2)
Code may be found at `examples/app_math/finite_field_generator_2.py`.

We now compute generators for the group from Chapter 13.2:

$$\begin{aligned} p &= rq + 1 \\ q &= 2^{32} + 15 \\ r &= 12. \end{aligned} \tag{B.93}$$

We see that we have the factorization

$$\begin{aligned} p - 1 &= rq \\ &= (12)\left(2^{32} + 15\right) \\ &= 2^2 \cdot 3 \cdot \left(2^{32} + 15\right). \end{aligned} \tag{B.94}$$

We accept the assertion that $2^{32} + 15$ is prime and set

$$\begin{aligned} p_1 &= 2 \\ p_2 &= 3 \\ p_3 &= 2^{32} + 15. \end{aligned} \tag{B.95}$$

Instead of randomly generating elements, we will start at 2 and proceed forward. For ease of notation, we set

$$\begin{aligned} q_1 &= \frac{(p-1)}{p_1} \\ &= 25769803866 \\ q_2 &= \frac{(p-1)}{p_2} \\ &= 17179869244 \\ q_3 &= \frac{(p-1)}{p_3} \\ &= 12. \end{aligned} \tag{B.96}$$

- 2: We see

$$\begin{aligned} 2^{q_1} \mod p &= 51539607732 \\ 2^{q_2} \mod p &= 1 \\ 2^{q_3} \mod p &= 4096. \end{aligned} \tag{B.97}$$

Because $2^{q_2} = 1$, 2 is not a generator of $\mathbb{F}_p^*$.

- 3: We see

$$\begin{aligned} 3^{q_1} &\mod p = 1 \\ 3^{q_2} &\mod p = 5944319764 \\ 3^{q_3} &\mod p = 531441. \end{aligned} \tag{B.98}$$

  Because $3^{q_1} = 1$, 3 is not a generator of $\mathbb{F}_p^*$.

- 4: We see

$$\begin{aligned} 4^{q_1} &\mod p = 1 \\ 4^{q_2} &\mod p = 1 \\ 4^{q_3} &\mod p = 16777216. \end{aligned} \tag{B.99}$$

  Because $4^{q_1} = 1$, 4 is not a generator of $\mathbb{F}_p^*$.

- 5: We see

$$\begin{aligned} 5^{q_1} &\mod p = 51539607732 \\ 5^{q_2} &\mod p = 5944319764 \\ 5^{q_3} &\mod p = 244140625. \end{aligned} \tag{B.100}$$

  Because $5^{q_i} \neq 1$ for all i, 5 is a generator of $\mathbb{F}_p^*$.

- 6: We see

$$\begin{aligned} 6^{q_1} &\mod p = 51539607732 \\ 6^{q_2} &\mod p = 5944319764 \\ 6^{q_3} &\mod p = 2176782336. \end{aligned} \tag{B.101}$$

  Because $6^{q_i} \neq 1$ for all i, 6 is a generator of $\mathbb{F}_p^*$.

- 7: We see

$$\begin{aligned} 7^{q_1} &\mod p = 51539607732 \\ 7^{q_2} &\mod p = 45595287968 \\ 7^{q_3} &\mod p = 13841287201. \end{aligned} \tag{B.102}$$

  Because $7^{q_i} \neq 1$ for all i, 7 is a generator of $\mathbb{F}_p^*$.

Thus, we were able to find 3 generators in the first 6 possibilities.

We note that Proposition B.12 *critically* relies on the factorization of the group order. It is an open problem to quickly determine a generator of a cyclic group; that is, it is an open problem to determine a polynomial time algorithm which computes a generator for a cyclic group without knowing the factorization of the group order [91, Remark 11.1.25]

B.4.2 Quadratic Residues and the Legendre Symbol

When working within $\mathbb{F}_p$, we want to know if $a \in \mathbb{F}_p$ has a square root; that is, we want to know if there is an $x \in \mathbb{F}_p$ such that

$$x^2 = a \mod p. \tag{B.103}$$

To this end, we have the following definitions:

Definition B.2 (Quadratic Residues and Quadratic Nonresidues)**.** Given $a \in \mathbb{F}_p^*$, we say that a is a *quadratic residue* if there is some $x \in \mathbb{F}_p^*$ such that

$$x^2 = a \mod p. \tag{B.104}$$

If this equation has no solution, then we say that a is a *quadratic nonresidue*.

We will sometimes shorten *quadratic residue* to *residue* and *quadratic nonresidue* to *nonresidue*.

We note the following interesting property:

$$\begin{aligned} R \cdot R &= R \\ R \cdot N &= N \\ N \cdot N &= R. \end{aligned} \tag{B.105}$$

Here, R denotes a residue and N denotes a nonresidue. While this may seem strange at first, we recall the following fact about the multiplication of real numbers:

$$\begin{aligned} (+) \cdot (+) &= (+) \\ (+) \cdot (-) &= (-) \\ (-) \cdot (-) &= (+)\,. \end{aligned} \tag{B.106}$$

We also note this fact about adding integers:

$$\begin{aligned} E + E &= E \\ E + O &= O \\ O + O &= E. \end{aligned} \tag{B.107}$$

Here, E denotes an even integer and O denotes an odd integer. In some ways, this property about quadratic residues and nonresidues may not be so strange.

From this definition, we can define another:

Definition B.3 (Legendre Symbol)**.** Given $a \in \mathbb{F}_p$, we can define the *Legendre symbol* as

$$\left(\frac{a}{p}\right) \coloneqq \begin{cases} 1, & a \text{ is a quadratic residue} \\ -1, & a \text{ is a quadratic nonresidue} \\ 0, & a = 0 \end{cases}. \tag{B.108}$$

One of the important properties of the Legendre symbol is that is it multiplicative: for all $a, b \in \mathbb{F}_p$, we have

$$\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right). \tag{B.109}$$

Additionally, it is not too difficult to show that there are equal numbers of residues and nonresidues.

Example B.15 (Computing Legendre Symbols by Squaring)
Code may be found at `examples/app_math/legendre_squaring.py`.

We will work with prime $p = 13$. We see the following:

$$\begin{aligned} 1^2 &= 1 \\ 2^2 &= 4 \\ 3^2 &= 9 \\ 4^2 &= 3 \\ 5^2 &= 12 \\ 6^2 &= 10 \\ 7^2 &= 10 \\ 8^2 &= 12 \\ 9^2 &= 3 \\ 10^2 &= 9 \\ 11^2 &= 4 \\ 12^2 &= 1. \end{aligned} \tag{B.110}$$

The numbers on the right-hand side are residues, while the numbers that do not show up are nonresidues. This implies the following:

$$\begin{aligned} \text{Residues of } p = 13: &\quad 1, 3, 4, 9, 10, 12 \\ \text{Nonresidues of } p = 13: &\quad 2, 5, 6, 7, 8, 11. \end{aligned} \tag{B.111}$$

To determine the residues and nonresidues for the previous example, we explicitly squared each element in $\mathbb{F}_p^*$. A more efficient method is this:

$$\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}}. \tag{B.112}$$

Example B.16 (Computing Legendre Symbols by Exponentiation)
Code may be found at `examples/app_math/legendre_exponentiation.py`.

We will work with prime $p = 7919$. We see the following:

$$\begin{aligned} 1^{\frac{p-1}{2}} &= 1 \\ 2^{\frac{p-1}{2}} &= 1 \\ 3^{\frac{p-1}{2}} &= 1 \\ 4^{\frac{p-1}{2}} &= 1 \\ 5^{\frac{p-1}{2}} &= 1 \\ 6^{\frac{p-1}{2}} &= 1 \\ 7^{\frac{p-1}{2}} &= -1. \end{aligned} \tag{B.113}$$

Thus, we see that all of these numbers have modular square roots in p except for 7.

B.4.3 Square Roots

In Eq. (B.112), we were given an efficient method for computing the Legendre symbol; however, knowing a square root *exists* is different from knowing its *value*.

Suppose that $a \in \mathbb{F}_p^*$ is a residue and $p = 3 \mod 4$. In this case, we know

$$a^{\frac{p-1}{2}} = 1. \tag{B.114}$$

We note the following:

$$\begin{aligned} \left(a^{\frac{p+1}{4}}\right)^2 &= a^{\frac{p+1}{2}} \\ &= a \cdot a^{\frac{p-1}{2}} \\ &= a. \end{aligned} \tag{B.115}$$

We used Eq. (B.114) going from line 2 to line 3.

This is a simple method for computing square roots in $\mathbb{F}_p^*$ when $p = 3 \mod 4$: Given $a \in \mathbb{F}_p^*$, we have

$$\sqrt{a} \coloneqq a^{\frac{p+1}{4}}. \tag{B.116}$$

We note that no deterministic algorithm is known when $p = 1 \mod 4$; even so, square roots may still be computed using the Tonelli–Shanks algorithm[1].

Example B.17 (Computing Square Roots by Exponentiation)
Code may be found at `examples/app_math/square_root.py`.

We will work with prime $p = 7919$; we note that $p = 3 \mod 4$. We see the following:

[1] https://en.wikipedia.org/wiki/Tonelli%E2%80%93Shanks_algorithm

$$\begin{aligned} 1^{\frac{p+1}{4}} &\mod p = 1 \\ 2^{\frac{p+1}{4}} &\mod p = 89 \\ 3^{\frac{p+1}{4}} &\mod p = 3023 \\ 4^{\frac{p+1}{4}} &\mod p = 2 \\ 5^{\frac{p+1}{4}} &\mod p = 1782 \\ 6^{\frac{p+1}{4}} &\mod p = 7720. \end{aligned} \tag{B.117}$$

We now verify the square roots:

$$\begin{aligned} 1^2 &\mod p = 1 \\ 89^2 &\mod p = 2 \\ 3023^2 &\mod p = 3 \\ 2^2 &\mod p = 4 \\ 1782^2 &\mod p = 5 \\ 7720^2 &\mod p = 6. \end{aligned} \tag{B.118}$$

We know that 7 does not have a square root. If we ignore this and compute

$$7^{\frac{p+1}{4}} \mod p = 4769, \tag{B.119}$$

then we find

$$\begin{aligned} 4769^2 \mod p &= 7912 \mod p \\ &= -7 \mod p. \end{aligned} \tag{B.120}$$

Thus, we *do* need to first compute the Legendre Symbol before attempting to compute its square root.

Bibliography

[1] Donald E. Eastlake 3rd, Steve Crocker, and Jeffrey I. Schiller. *Randomness Requirements for Security*. RFC 4086. June 2005. DOI: 10.17487/RFC4086. URL: https://www.rfc-editor.org/info/rfc4086 (cit. on p. 130).

[2] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2020 (cit. on p. 207).

[3] Tomer Ashur and Siemen Dhooghe. *MARVELlous: a STARK-Friendly Family of Cryptographic Primitives*. Cryptology ePrint Archive, Report 2018/1098. https://ia.cr/2018/1098. 2018 (cit. on p. 178).

[4] Elaine Barker, Lily Chen, and Richard Davis. "Recommendation for Key-Derivation Methods in Key-Establishment Schemes." In: *NIST Special Publication* 800-56CRev2 (2020). DOI: https://doi.org/10.6028/NIST.SP.800-56Cr2 (cit. on p. 222).

[5] Elaine Barker and Nicky Mouha. "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher." In: NIST Special Publication 800-67Rev2 (2017). DOI: https://doi.org/10.6028/NIST.SP.800-67r2 (cit. on p. 75).

[6] Elaine Barker and Allen Roginsky. "Transitioning the Use of Cryptographic Algorithms and Key Lengths." In: NIST Special Publication 800-131ARev2 (2018). DOI: https://doi.org/10.6028/NIST.SP.800-131Ar2 (cit. on p. 75).

[7] Elaine B. Barker and John Michael Kelsey. "Recommendation for Random Number Generation Using Deterministic Random Bit Generators." In: NIST Special Publication 800-90A (2012). DOI: http://dx.doi.org/10.6028/NIST.SP.800-90A (cit. on pp. 6, 77).

[8] Elaine B. Barker and John Michael Kelsey. "Recommendation for Random Number Generation Using Deterministic Random Bit Generators." In: NIST Special Publication 800-90ARev1 (2015). DOI: http://dx.doi.org/10.6028/NIST.SP.800-90Ar1 (cit. on pp. 6, 77).

[9] Paulo S. L. M. Barreto and Michael Naehrig. "Pairing-Friendly Elliptic Curves of Prime Order." In: *Selected Areas in Cryptography*. Ed. by Bart Preneel and Stafford Tavares. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 319–331. ISBN: 978-3-540-33109-4. URL: https://link.springer.com/content/pdf/10.1007/11693383_22.pdf (cit. on p. 159).

[10] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "Keying Hash Functions for Message Authentication." In: *Advances in Cryptology — CRYPTO '96*. Ed. by Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–15. ISBN: 978-3-540-68697-2. URL: https://link.springer.com/content/pdf/10.1007/3-540-68697-5_1.pdf (cit. on pp. 117, 216, 217).

[11] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. *Scalable, transparent, and post-quantum secure computational integrity*. Cryptology ePrint Archive, Report 2018/046. https://ia.cr/2018/046. 2018 (cit. on p. 178).

[12] Daniel J. Bernstein. "Cache-timing attacks on AES." In: (2005). URL: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf (cit. on p. 75).

[13] Daniel J. Bernstein. "ChaCha, a variant of Salsa20." In: *Workshop record of SASC*. Vol. 8. 2008, pp. 3–5. URL: https://cr.yp.to/chacha/chacha-20080120.pdf (cit. on p. 73).

[14] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records." In: *Public Key Cryptography - PKC 2006*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228. ISBN: 978-3-540-33852-9. URL: https://cr.yp.to/ecdh/curve25519-20060209.pdf (cit. on pp. 60, 61).

[15] Daniel J. Bernstein. "Extending the Salsa20 nonce." In: *Symmetric Key Encryption Workshop*. Vol. 2011. 2011. URL: https://cr.yp.to/snuffle/xsalsa-20110204.pdf (cit. on p. 73).

[16] Daniel J. Bernstein. "The Poly1305-AES Message-Authentication Code." In: *Fast Software Encryption*. Ed. by Henri Gilbert and Helena Handschuh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 32–49. ISBN: 978-3-540-31669-5. URL: https://cr.yp.to/mac/poly1305-20050329.pdf (cit. on p. 79).

[17] Daniel J. Bernstein. "The Salsa20 Family of Stream Ciphers." In: *New Stream Cipher Designs: The eSTREAM Finalists*. Ed. by Matthew Robshaw and Olivier Billet. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 84–97. ISBN: 978-3-540-68351-3. DOI: 10.1007/978-3-540-68351-3_8. URL: https://cr.yp.to/snuffle/salsafamily-20071225.pdf (cit. on p. 73).

[18] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-speed high-security signatures." In: *Journal of Cryptographic Engineering* 2.2 (2012), pp. 77–89. URL: https://ed25519.cr.yp.to/ed25519-20110926.pdf (cit. on pp. 61, 149).

[19] Daniel J. Bernstein and Andreas Hülsing. *Decisional second-preimage resistance: When does SPR imply PRE?* Cryptology ePrint Archive, Paper 2019/492. https://eprint.iacr.org/2019/492. 2019. URL: https://eprint.iacr.org/2019/492 (cit. on p. 81).

[20] Daniel J. Bernstein, Simon Josefsson, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "EdDSA for more curves." In: *Cryptology ePrint Archive* 2015 (2015). URL: https://eprint.iacr.org/2015/677.pdf (cit. on p. 149).

[21] Daniel J. Bernstein, Tanja Lange, and Ruben Niederhagen. "Dual EC: A Standardized Back Door." In: *The New Codebreakers: Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*. Ed. by Peter Y. A. Ryan, David Naccache, and Jean-Jacques Quisquater. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 256–281. ISBN: 978-3-662-49301-4. DOI: 10.1007/978-3-662-49301-4_17. URL: https://projectbullrun.org/dual-ec/documents/dual-ec-20150731.pdf (cit. on pp. 6, 77).

[22] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Cryptographic sponge functions." In: *Submission to NIST (Round 3)* (2011). URL: https://keccak.team/files/CSF-0.1.pdf (cit. on pp. 83, 209, 214, 215).

[23] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications." In: *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. 2016, pp. 292–302. DOI: 10.1109/EuroSP.2016.31. URL: https://orbilu.uni.lu/bitstream/10993/31652/1/Argon2ESP.pdf (cit. on p. 119).

[24] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. *From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again*. Cryptology ePrint Archive, Report 2011/443. https://ia.cr/2011/443. 2011 (cit. on p. 178).

[25] Jeremiah Blocki, Benjamin Harsha, and Samson Zhou. "On the Economics of Offline Password Cracking." In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 853–871. URL: https://arxiv.org/pdf/2006.05023.pdf (cit. on pp. 6, 119, 154, 227, 229).

[26] Hristo Bojinov, Daniel Sanchez, Paul Reber, Dan Boneh, and Patrick Lincoln. "Neuroscience Meets Cryptography: Designing Crypto Primitives Secure Against Rubber Hose Attacks." In: *21st USENIX Security Symposium (USENIX Security 12)*. 2012, pp. 129–141. URL: https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final25.pdf (cit. on p. 2).

[27] Dan Boneh. "The Decision Diffie-Hellman Problem." In: *International Algorithmic Number Theory Symposium*. Springer. 1998, pp. 48–63. URL: https://crypto.stanford.edu/~dabo/pubs/papers/DDH.pdf (cit. on p. 203).

[28] Dan Boneh, Henry Corrigan-Gibbs, and Stuart Schechter. *Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks*. Cryptology ePrint Archive, Report 2016/027. https://ia.cr/2016/027. 2016 (cit. on p. 119).

[29] Dan Boneh, Ben Lynn, and Hovav Shacham. "Short Signatures from the Weil Pairing." In: *Journal of Cryptology* 17.4 (2004), pp. 297–319. URL: https://link.springer.com/content/pdf/10.1007/s00145-004-0314-9.pdf (cit. on pp. 39, 61, 159–161, 188).

[30] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. Version 0.6. 2023. URL: https://toc.cryptobook.us/book.pdf (cit. on pp. 97, 205).

[31] Charles Bouillaguet, Florette Martinez, and Julia Sauvage. "Practical seed-recovery for the PCG Pseudo-Random Number Generator." In: *IACR Transactions on Symmetric Cryptology* (2020), pp. 175–196. URL: https://tosc.iacr.org/index.php/ToSC/article/download/8700/8292 (cit. on pp. 6, 77).

[32] Joachim Breitner and Nadia Heninger. *Biased Nonce Sense: Lattice Attacks against Weak ECDSA Signatures in Cryptocurrencies*. Cryptology ePrint Archive, Report 2019/023. https://ia.cr/2019/023. 2019 (cit. on pp. 135, 142).

[33] Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. "Efficient Indifferentiable Hashing into Ordinary Elliptic Curves." In: *Advances in Cryptology – CRYPTO 2010*. Ed. by Tal Rabin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 237–254. ISBN: 978-3-642-14623-7. URL: https://link.springer.com/content/pdf/10.1007/978-3-642-14623-7_13.pdf (cit. on p. 162).

[34] Daniel R. L. Brown. "SEC 2: Recommended Elliptic Curve Domain Parameters." In: *Standards for Efficient Cryptography* (2010). URL: https://www.secg.org/sec2-v2.pdf (cit. on p. 60).

[35] Daniel R. L. Brown. *What Hashes Make RSA-OAEP Secure?* Cryptology ePrint Archive, Paper 2006/223. https://eprint.iacr.org/2006/223. 2006. URL: https://eprint.iacr.org/2006/223 (cit. on p. 227).

[36] Daniel R. L. Brown and Robert P. Gallant. *The Static Diffie-Hellman Problem*. Cryptology ePrint Archive, Report 2004/306. https://ia.cr/2004/306. 2004 (cit. on p. 203).

[37] Billy Bob Brumley and Nicola Tuveri. "Remote Timing Attacks Are Still Practical." In: *European Symposium on Research in Computer Security*. Springer. 2011, pp. 355–371. URL: https://link.springer.com/content/pdf/10.1007/978-3-642-23822-2_20.pdf (cit. on pp. 44, 149).

[38] Jan Camenisch and Markus Stadler. *Proof Systems for General Statements about Discrete Logarithms*. Tech. rep. ETH Zurich, Department of Computer Science, 1997. URL: https://doi.org/10.3929/ethz-a-006651937 (cit. on pp. 166, 178).

[39] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Discrete Mathematics and Its Applications. CRC Press, 2006 (cit. on p. 207).

[40] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd ed. The MIT Press, 2009 (cit. on p. 24).

[41] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002, p. 238. ISBN: 3-540-42580-2 (cit. on p. 75).

[42] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. "Efficient Sparse Merkle Trees." In: *Nordic Conference on Secure IT Systems*. Springer. 2016, pp. 199–215. URL: https://eprint.iacr.org/2016/683.pdf (cit. on pp. 97, 117).

[43] Ivan Bjerre Damgård. "A Design Principle for Hash Functions." In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 416–427. URL: https://link.springer.com/content/pdf/10.1007%252F0-387-34805-0_39.pdf (cit. on pp. 88, 209).

[44] Whitfield Diffie and Martin Hellman. "New Directions in Cryptography." In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. URL: https://www-ee.stanford.edu/~hellman/publications/24.pdf (cit. on p. 124).

[45] David S. Dummit and Richard M. Foote. *Abstract Algebra*. 3rd ed. Prentice Hall Englewood Cliffs, NJ, 2004 (cit. on p. 207).

[46] Morris Dworkin. "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions." In: Federal Information Processing Standards Publication 202 (2015). DOI: https://doi.org/10.6028/NIST.FIPS.202 (cit. on pp. 83, 209, 213, 227).

[47] Nadia El Mrabet and Marc Joye. *Guide to Pairing-Based Cryptography*. Chapman and Hall/CRC Cryptography and Network Security. CRC Press, 2017 (cit. on p. 164).

[48] Taher Elgamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms." In: *IEEE Transactions on Information Theory* 31.4 (1985), pp. 469–472. URL: https://link.springer.com/content/pdf/10.1007/3-540-39568-7_2.pdf (cit. on pp. 127, 132).

[49] Paul Feldman. "A Practical Scheme for Non-interactive Verifiable Secret Sharing." In: *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*. IEEE. 1987, pp. 427–438. URL: https://www.cs.umd.edu/~gasarch/TOPICS/secretsharing/feldmanVSS.pdf (cit. on pp. 186, 187).

[50] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. Wiley, 2003 (cit. on pp. 77, 207).

[51] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. John Wiley & Sons, 2011 (cit. on pp. 74, 77, 112, 206).

[52] Pierre-Alain Fouque and Mehdi Tibouchi. "Indifferentiable Hashing to Barreto–Naehrig Curves." In: *International Conference on Cryptology and Information Security in Latin America*. Springer. 2012, pp. 1–17. URL: https://hal.inria.fr/hal-01094321/document (cit. on p. 162).

[53] Peter Gaži, Krzysztof Pietrzak, and Michal Rybár. *The Exact PRF-Security of NMAC and HMAC*. Cryptology ePrint Archive, Report 2014/578. https://ia.cr/2014/578. 2014 (cit. on p. 218).

[54] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. "Revisiting the Distributed Key Generation for Discrete-Log Based Cryptosystems." In: *RSA Security'03* (2002). URL: https://www.researchgate.net/profile/Stanislaw-Jarecki/publication/2558744_Revisiting_the_Distributed_Key_Generation_for_Discrete-Log_Based_Cryptosystems/links/54856b510cf283750c3715db/Revisiting-the-Distributed-Key-Generation-for-Discrete-Log-Based-Cryptosystems.pdf (cit. on pp. 186, 187, 193).

[55] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. "Secure Distributed Key Generation for Discrete-Log Based Cryptosystems." In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1999, pp. 295–310. URL: https://link.springer.com/content/pdf/10.1007/3-540-48910-X_21.pdf (cit. on pp. 186, 187, 193).

[56] Oded Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools*. Cambridge University Press, 2001 (cit. on p. 207).

[57] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, 2004 (cit. on p. 207).

[58] Christopher Gorman, Hunter Prendergast, Anthony Dean, Tom Bollich, and Adam Helfgott. *AliceNet*. Whitepaper. 2022. URL: https://github.com/alicenet/whitepaper (cit. on p. 1).

[59] Faraz Haider. "Compact Sparse Merkle Trees." In: (2018). URL: https://eprint.iacr.org/2018/955.pdf (cit. on p. 117).

[60] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. *An Introduction to Mathematical Cryptography*. 2nd ed. Undergraduate Texts in Mathematics. Springer, 2014 (cit. on pp. 204, 205).

[61] Jan Jancar, Vladimir Sedlacek, Petr Svenda, and Marek Sys. "Minerva: The curse of ECDSA nonces." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), pp. 281–308. URL: https://tches.iacr.org/index.php/TCHES/article/download/8684/8243/ (cit. on p. 149).

[62] Jérémy Jean. *TikZ for Cryptographers*. https://www.iacr.org/authors/tikz/. 2016 (cit. on pp. 211, 214).

[63] Simon Josefsson and Ilari Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC 8032. Jan. 2017. DOI: 10.17487/RFC8032. URL: https://rfc-editor.org/rfc/rfc8032.txt (cit. on p. 149).

[64] Benjamin Kaduk and Michiko Short. *Deprecate Triple-DES (3DES) and RC4 in Kerberos*. RFC 8429. Oct. 2018. DOI: 10.17487/RFC8429. URL: https://www.rfc-editor.org/info/rfc8429 (cit. on p. 75).

[65] Janakirama Kalidhindi, Alex Kazorian, Aneesh Khera, and Cibi Pari. *Angela: A Sparse, Distributed, and Highly Concurrent Merkle Tree*. Tech. rep. UC Berkeley, 2018. URL: https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F18/projects/reports/project1_report_ver3.pdf (cit. on p. 117).

[66] Burt Kaliski. *PKCS #5: Password-Based Cryptography Specification Version 2.0*. RFC 2898. Sept. 2000. DOI: 10.17487/RFC2898. URL: https://www.rfc-editor.org/info/rfc2898 (cit. on pp. 119, 227).

[67] Burt Kaliski and Jessica Staddon. *PKCS #1: RSA Cryptography Specifications Version 2.0*. RFC 2437. Oct. 1998. DOI: 10.17487/RFC2437. URL: https://www.rfc-editor.org/info/rfc2437 (cit. on p. 227).

[68] Phil R. Karn, William A. Simpson, and Perry E. Metzger. *The ESP Triple DES Transform*. RFC 1851. Sept. 1995. DOI: 10.17487/RFC1851. URL: https://www.rfc-editor.org/info/rfc1851 (cit. on p. 75).

[69] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. 3rd ed. Chapman and Hall/CRC Cryptography and Network Security. CRC Press, 2021 (cit. on pp. 5, 44, 69, 72, 130, 143, 201–205, 207).

[70] Scott G. Kelly. *Security Implications of Using the Data Encryption Standard (DES)*. RFC 4772. Dec. 2006. DOI: 10.17487/RFC4772. URL: https://www.rfc-editor.org/info/rfc4772 (cit. on pp. 6, 75).

[71] John Kelsey, Shu-jen Chang, and Ray Perlner. "SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash." In: NIST Special Publication 800-185 (2016). DOI: https://doi.org/10.6028/NIST.SP.800-185 (cit. on pp. 220, 227).

[72] Jongsung Kim, Alex Biryukov, Bart Preneel, and Seokhie Hong. *On the Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0 and SHA-1*. Cryptology ePrint Archive, Report 2006/187. https://ia.cr/2006/187. 2006 (cit. on p. 218).

[73] Donald Ervin Knuth. *The Art of Computer Programming*. 3rd ed. Vol. 1. Addison-Wesley, 1997 (cit. on p. 24).

[74] Donald Ervin Knuth. *The Art of Computer Programming*. 3rd ed. Vol. 2. Addison-Wesley, 1997 (cit. on pp. 76, 77).

[75] Neal Koblitz. *A Course in Number Theory and Cryptography*. 2nd ed. Vol. 114. Graduate Texts in Mathematics. Springer, 1994 (cit. on p. 208).

[76] Hugo Krawczyk. *Cryptographic Extraction and Key Derivation: The HKDF Scheme*. Cryptology ePrint Archive, Report 2010/264. https://ia.cr/2010/264. 2010 (cit. on pp. 118, 220, 222).

[77] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. Feb. 1997. DOI: 10.17487/RFC2104. URL: https://rfc-editor.org/rfc/rfc2104.txt (cit. on pp. 117, 216–218).

[78] Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. May 2010. DOI: 10.17487/RFC5869. URL: https://rfc-editor.org/rfc/rfc5869.txt (cit. on pp. 118, 220, 223).

[79] Ben Laurie and Emilia Kasper. "Revocation Transparency." In: *Google Research* 33 (Sept. 2012). URL: https://www.links.org/files/RevocationTransparency.pdf (cit. on p. 117).

[80] Ben Laurie, Adam Langley, and Emilia Kasper. *Certificate Transparency*. RFC 6962. June 2013. DOI: 10.17487/RFC6962. URL: https://www.rfc-editor.org/info/rfc6962 (cit. on pp. 97, 112).

[81] Ben Laurie, Adam Langley, Emilia Kasper, Eran Messeri, and Rob Stradling. *Certificate Transparency Version 2.0*. RFC 9162. Dec. 2021. DOI: 10.17487/RFC9162. URL: https://www.rfc-editor.org/info/rfc9162 (cit. on pp. 97, 112–114, 117).

[82] Arjen Lenstra, Xiaoyun Wang, and Benne de Weger. *Colliding X.509 Certificates*. Cryptology ePrint Archive, Report 2005/067. https://ia.cr/2005/067. 2005 (cit. on p. 88).

[83] Gaëtan Leurent and Thomas Peyrin. *SHA-1 is a Shambles - First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust*. Cryptology ePrint Archive, Report 2020/014. https://ia.cr/2020/014. 2020 (cit. on p. 88).

[84] George Marsaglia. "Random Numbers Fall Mainly in the Planes." In: *Proceedings of the National Academy of Sciences of the United States of America* 61.1 (1968), p. 25. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC285899/pdf/pnas00123-0038.pdf (cit. on pp. 6, 77).

[85] Makoto Matsumoto and Takuji Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator." In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), pp. 3–30 (cit. on p. 76).

[86] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. Discrete Mathematics and Its Applications. CRC Press, 1997 (cit. on p. 207).

[87] Ralph Charles Merkle. "Secrecy, Authentication, and Public Key Systems." PhD thesis. Stanford University, 1979. URL: https://www.merkle.com/papers/Thesis1979.pdf (cit. on pp. 88, 132, 209).

[88] Ondrej Mikle. *Practical Attacks on Digital Signatures Using MD5 Message Digest*. Cryptology ePrint Archive, Report 2004/356. https://ia.cr/2004/356. 2004 (cit. on p. 88).

[89] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. Nov. 2016. DOI: 10.17487/RFC8017. URL: https://www.rfc-editor.org/info/rfc8017 (cit. on p. 227).

[90] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. *PKCS #5: Password-Based Cryptography Specification Version 2.1*. RFC 8018. Jan. 2017. DOI: 10.17487/RFC8018. URL: https://www.rfc-editor.org/info/rfc8018 (cit. on pp. 119, 154, 227, 228).

[91] Gary L. Mullen and Daniel Panario. *Handbook of Finite Fields*. Discrete Mathematics and Its Applications. CRC Press, 2013 (cit. on pp. 244, 250).

[92] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System." In: (2008). URL: https://bitcoin.org/bitcoin.pdf (cit. on p. 60).

[93] Phong Q. Nguyen and Igor E. Shparlinski. "The Insecurity of the Digital Signature Algorithm with Partially Known Nonces." In: *Journal of Cryptology* 15.3 (2002). URL: https://link.springer.com/content/pdf/10.1007/s00145-002-0021-3.pdf (cit. on pp. 141, 142).

[94] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 8439. June 2018. DOI: 10.17487/RFC8439. URL: https://rfc-editor.org/rfc/rfc8439.txt (cit. on pp. 73, 79).

[95] Melissa E. O'Neill. "PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation." In: *ACM Transactions on Mathematical Software* (2014). URL: https://www.pcg-random.org/pdf/toms-oneill-pcg-family-v1.02.pdf (cit. on p. 76).

[96] OpenZeppelin. *Merkle Tree*. Commit `b3d9889`. 2023. URL: https://github.com/OpenZeppelin/merkle-tree (visited on 02/19/2023) (cit. on pp. 112, 115, 116, 118).

[97] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer, 2010 (cit. on pp. 204, 205).

[98] Colin Percival. *Stronger Key Derivation via Sequential Memory-Hard Functions*. 2009. URL: https://www.tarsnap.com/scrypt/scrypt.pdf (cit. on p. 119).

[99] Colin Percival and Simon Josefsson. *The scrypt Password-Based Key Derivation Function*. RFC 7914. Aug. 2016. DOI: 10.17487/RFC7914. URL: https://rfc-editor.org/rfc/rfc7914.txt (cit. on p. 119).

[100] Tim Polk, Sean Turner, and Paul E. Hoffman. *Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms*. RFC 6194. Mar. 2011. DOI: 10.17487/RFC6194. URL: https://www.rfc-editor.org/info/rfc6194 (cit. on pp. 6, 88, 218).

[101] Andrei Popov. *Prohibiting RC4 Cipher Suites*. RFC 7465. Feb. 2015. DOI: 10.17487/RFC7465. URL: https://www.rfc-editor.org/info/rfc7465 (cit. on pp. 6, 73).

[102] Thomas Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. RFC 6979. Aug. 2013. DOI: 10.17487/RFC6979. URL: https://rfc-editor.org/rfc/rfc6979.txt (cit. on pp. 4, 130, 135, 141, 142).

[103] Gordon Procter. *A Security Analysis of the Composition of ChaCha20 and Poly1305*. Cryptology ePrint Archive, Report 2014/613. https://ia.cr/2014/613. 2014 (cit. on p. 79).

[104] Niels Provos and David Mazieres. "A Future-Adaptable Password Scheme." In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 1999. 1999, pp. 81–91. URL: https://www.usenix.org/legacy/events/usenix99/full_papers/provos/provos.pdf (cit. on p. 119).

[105] Charles Rackoff and Daniel R. Simon. "Non-Interactive Zero-Knowledge Proof of Knowledge and Chosen Ciphertext Attack." In: *Annual International Cryptology Conference*. Springer. 1991, pp. 433–444. URL: https://link.springer.com/content/pdf/10.1007/3-540-46766-1_35.pdf (cit. on p. 178).

[106] Lum Ramabaja and Arber Avdullahu. "Compact Merkle Multiproofs." In: *CoRR* abs/2002.07648 (2020). arXiv: 2002.07648. URL: https://arxiv.org/abs/2002.07648 (cit. on p. 117).

[107] Ronald L. Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321. Apr. 1992. DOI: 10.17487/RFC1321. URL: https://rfc-editor.org/rfc/rfc1321.txt (cit. on pp. 83, 209, 213).

[108] Ronald L. Rivest and Jacob C. N. Schuldt. *Spritz—a spongy RC4-like stream cipher and hash function*. Cryptology ePrint Archive, Report 2016/856. https://ia.cr/2016/856. 2016 (cit. on p. 73).

[109] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Communications of the ACM* 21.2 (1978), pp. 120–126. URL: https://apps.dtic.mil/sti/pdfs/ADA606588.pdf (cit. on pp. 124, 130, 131, 142).

[110] Phillip Rogaway and Thomas Shrimpton. *Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance*. Cryptology ePrint Archive, Report 2004/035. https://ia.cr/2004/035. 2004 (cit. on p. 81).

[111] Sheldon M. Ross. *Introduction to Probability Models*. Academic Press, 2014 (cit. on p. 208).

[112] Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. "Non-Interactive Zero-Knowledge Proof Systems." In: *Conference on the Theory and Application of Cryptographic Techniques*. Springer. 1987, pp. 52–72. URL: https://link.springer.com/content/pdf/10.1007/3-540-48184-2_5.pdf (cit. on p. 178).

[113] Yu Sasaki and Kazumaro Aoki. "Finding Preimages in Full MD5 Faster Than Exhaustive Search." In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2009, pp. 134–152. URL: https://link.springer.com/content/pdf/10.1007/978-3-642-01001-9_8.pdf (cit. on p. 88).

[114] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. *ETHDKG: Distributed Key Generation with Ethereum Smart Contracts*. Cryptology ePrint Archive, Report 2019/985. https://eprint.iacr.org/2019/985. 2019 (cit. on pp. 187, 190).

[115] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 1996 (cit. on pp. 206, 207).

[116] Bruce Schneier. "Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)." In: *International Workshop on Fast Software Encryption*. Springer. 1993, pp. 191–204. URL: https://link.springer.com/content/pdf/10.1007/3-540-58108-1_24.pdf (cit. on pp. 6, 74, 76, 119).

[117] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *The Twofish Encryption Algorithm: A 128-bit Block Cipher*. John Wiley & Sons, Inc., 1999 (cit. on pp. 6, 74, 76).

[118] Adi Shamir. "How to Share a Secret." In: *Communications of the ACM* 22.11 (1979), pp. 612–613. URL: https://dl.acm.org/doi/pdf/10.1145/359168.359176 (cit. on pp. 179, 180).

[119] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2009. URL: https://shoup.net/ntb/ (cit. on pp. 20, 204–206, 208, 231).

[120] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*. 2nd ed. Vol. 106. Graduate Texts in Mathematics. Springer, 2009 (cit. on pp. 51, 62, 208).

[121] Joseph H. Silverman and John Torrence Tate. *Rational Points on Elliptic Curves*. Undergraduate Texts in Mathematics. Springer, 1992 (cit. on p. 208).

[122] National Bureau of Standards. "Data Encryption Standard." In: Federal Information Processing Standards Publication 46-3 (1977) (cit. on pp. 74, 75).

[123] National Institute of Standards and Technology. "Announcing the Advanced Encryption Standard (AES)." In: Federal Information Processing Standards Publication 197 (2001). DOI: https://doi.org/10.6028/NIST.FIPS.197 (cit. on pp. 74, 75, 83).

[124] National Institute of Standards and Technology. "Secure Hash Standard." In: Federal Information Processing Standards Publication 180-1 (1995) (cit. on pp. 83, 209).

[125] National Institute of Standards and Technology. "Secure Hash Standard." In: Federal Information Processing Standards Publication 180-4 (2015). URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf (cit. on pp. 83, 209, 213).

[126] Marc Stevens. 2012. URL: https://marc-stevens.nl/research/md5-1block-collision/ (cit. on pp. 88, 89).

[127] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. *The first collision for full SHA-1*. Cryptology ePrint Archive, Paper 2017/190. https://eprint.iacr.org/2017/190. 2017. URL: https://eprint.iacr.org/2017/190 (cit. on pp. 88, 90).

[128] Sean Turner. *Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms*. RFC 6151. Mar. 2011. DOI: 10.17487/RFC6151. URL: https://rfc-editor.org/rfc/rfc6151.txt (cit. on pp. 6, 88, 117, 218).

[129] Riad S. Wahby and Dan Boneh. *Fast and simple constant-time hashing to the BLS12-381 elliptic curve*. Cryptology ePrint Archive, Report 2019/403. https://ia.cr/2019/403. 2019 (cit. on p. 162).

[130] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*. Cryptology ePrint Archive, Report 2004/199. https://ia.cr/2004/199. 2004 (cit. on p. 88).

[131] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography*. Discrete Mathematics and Its Applications. CRC Press, 2008 (cit. on p. 208).

[132] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. "Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations." In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 1767–1784. URL: https://www.usenix.org/system/files/sec20-weiser.pdf (cit. on p. 149).

[133] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. "A Cache Timing Attack on AES in Virtualization Environments." In: *International Conference on Financial Cryptography and Data Security*. Springer. 2012, pp. 314–328. URL: https://fc12.ifca.ai/pre-proceedings/paper_70.pdf (cit. on p. 75).

[134] Gavin Wood. "Ethereum: A Secure Decentralised Generalised Transaction Ledger." In: *Ethereum project yellow paper* (2019). Commit `7e819ec`. URL: https://github.com/ethereum/yellowpaper (cit. on p. 153).

[135] Gavin Wood. "Ethereum: A Secure Decentralised Generalised Transaction Ledger." In: *Ethereum project yellow paper* (2022). Commit `4b05e0d`. URL: https://github.com/ethereum/yellowpaper (cit. on pp. 60, 92, 152, 159, 160, 178, 187).

[136] Yuval Yarom and Naomi Benger. *Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack*. Cryptology ePrint Archive, Report 2014/140. https://ia.cr/2014/140. 2014 (cit. on p. 54).

[137] Uwe Ziegenhagen. 2010. URL: https://texample.net/tikz/examples/set-operations-illustrated-with-venn-diagrams/ (cit. on p. 10).

Glossary

abelian group A group in which the underlying group operation is commutative. 32, 33, 36, 44, 47

associative The binary operation $\boxplus$ is associative if $(a \boxplus b) \boxplus c = a \boxplus (b \boxplus c)$ for all a, b, and c. Standard examples include addition and multiplication; standard nonexamples include subtraction and division. Informally, an operation is associative if parentheses may be arbitrarily rearranged. 27–29, 31, 33, 34

authenticated encryption A method for combining an encryption scheme with message integrity. 78, 79

bijective A function f is bijective (or is a bijection) if it is both injective and surjective. In this case, f has an inverse function. 13–15, 269

bilinear pairing A type of function which is linear in both of its arguments; used extensively in Pairing-Based Cryptography. 2, 39, 61, 62, 159, 160, 163, 179, 188, 203, 269

block cipher An encryption scheme which takes a secret key and encrypts messages based on a keyed-permutation. 6, 72, 74–76, 270

commutative The binary operation $\boxplus$ is commutative if $a \boxplus b = b \boxplus a$ for all a and b. Standard examples of commutative operations include addition and multiplication; standard nonexamples include subtraction and division. Informally, an operation is commutative if "the order of evaluation does not matter". 28, 29, 32–34, 266

commutative ring A ring in which multiplication is commutative. 34, 36

cyclic group A group which is generated by one element; that is, $(G, \cdot)$ is a cyclic group if $(G, \cdot)$ is a group and there is a $g \in G$ such that for all $h \in G$ there is an $x \in \mathbb{Z}$ such that $h = g^x$. 32, 39, 125, 127, 134, 135, 143, 144, 161, 167, 170, 175, 201, 244, 250, 266, 267

Diffie-Hellman Key Exchange A key exchange in Public Key Cryptography based on deriving a shared secret from public information. This comes from deriving a shared secret within a cyclic group. 27, 32, 35, 38, 39, 60, 124, 127, 143, 144, 189, 191, 220

Diffie-Hellman Problem Given a finite cyclic group $G = \langle g \rangle$ of order q with $a, b \xleftarrow{\$} \mathbb{Z}_q^*$, $A = g^a$, and $B = g^b$, solve for g^{ab} given A and B. 203

digital signature A method for authenticating data in Public Key Cryptography. Standard examples include DSA, ECDSA, and EdDSA. Digital signatures allow for nonrepudiation. 1, 2, 4, 27, 32, 35, 39, 60, 78, 124, 129, 131, 132, 142, 143, 160, 165, 188, 267, 269, 270

discrete logarithm For a cyclic group $G = \langle g \rangle$ and $h \in G$, if $h = g^x$, then x is the discrete logarithm of h with respect to g. 166, 168, 171, 172, 176, 178, 191, 201

Discrete Logarithm Problem Given a finite cyclic group $G = \langle g \rangle$ and $h \in G$, compute x such that $h = g^x$. 47, 56, 58, 143, 201, 202

distributed key generation A method for constructing a secret key between multiple participants and have it distributed between them. After the secret key is constructed, it is possible construct valid digital signatures for the entire group (called *group signatures*). 3, 61, 62, 163, 179, 186–188, 194, 197, 268

elliptic curve Defined over a field, an elliptic curve is a group defined by a collection of points which satisfy a certain polynomial equation as well as a group operation. Elliptic curves are used in Elliptic Curve Cryptography. 1–3, 35, 39–44, 46–62, 81, 92, 127, 143, 144, 146, 147, 149, 150, 159, 161, 162, 187, 202, 207, 208, 230, 267

Elliptic Curve Cryptography The area of cryptography which uses elliptic curves as the basis for security. Security is based on the assumption that certain mathematical operations involving elliptic curves are impractical to solve. 1, 2, 44, 56, 92, 124, 143, 153, 201, 207, 208, 267

encryption scheme A general method for converting a message (*plaintext*) into random-looking data (*ciphertext*). Alice and Bob use an encryption scheme to communicate over an insecure channel. 4, 5, 69, 72, 79, 127, 130, 132, 142, 190, 266, 268–270

Ethereum A blockchain with a built-in computational engine. It is possible to write smart contracts within Ethereum. 60, 77, 92, 93, 112, 149, 150, 152, 159, 160, 178, 187, 192, 270

extendable output function A deterministic method for producing a digital identifier of variable length; similar to a cryptographic hash function. Compare with mask generation function. 119, 224, 227, 269

field A field is a set together with two binary operations (addition and multiplication) with additional properties; importantly, every nonzero element has a multiplicative inverse. A standard example of a field is the rational numbers under addition and multiplication. 1, 27, 32, 35–38, 47, 60, 61, 67, 143, 207, 267, 268

finite cyclic group A group which is both finite and cyclic. 165, 166, 201, 202, 245, 267

finite field A field which has a finite number of elements. 36, 47, 51, 56, 59, 62, 65, 67, 68, 143, 161, 162, 196, 197, 229, 244

finite group A group which has a finite number of elements. 30, 56, 62

function A mapping which assigns an input from the domain to an output from the codomain. 8, 12–15, 62, 65, 66, 266

group A group is a set together with a binary operation which satisfies additional properties. Standard examples of groups include the integers under addition as well as the positive rationals under multiplication. 1, 27–32, 38, 39, 43, 58, 62, 67, 128, 139, 143, 144, 146, 147, 161, 202, 207, 246, 248, 266–268, 270

hash function A method for producing a constant-length digital identifier for data. The full name is **cryptographic hash function**. 2, 4, 6, 76, 79–83, 88, 92, 93, 95, 97, 98, 117–120, 123, 125, 132, 134, 138, 147, 150, 160–162, 166, 168, 188, 190, 209–213, 215–218, 222, 227, 228, 230, 267–270

Hash-based Message Authentication Code A general method for constructing a message authentication code from a hash function. 79, 117, 216, 217

HMAC-based Key Derivation Function A general method for constructing a key derivation function from a hash function. 118, 220, 222, 223, 227, 228, 230

initialization vector A bit string used to initialize an encryption schemes. Compare with nonce and salt. 73, 74, 77, 209, 211, 269, 270

injective A function f is injective if $f(x) = f(y)$ implies $x = y$; also called *one-to-one*. 12–15, 266

insecure channel A method for communication where no secrecy is assured. For instance, when Alice and Bob communicate over an insecure channel, they assume that Eve will learn everything they discuss. Because of this, there is no privacy on insecure channels. 69, 70, 72, 124, 125, 130, 189, 267

key derivation function A method for converting nonuniform randomness into uniform randomness. Also used to store passwords. 4, 6, 76, 118, 119, 125, 220, 268

Lagrange Interpolation A method of interpolation using Lagrange polynomials. Interpolation methods are used throughout mathematics, science, and engineering. Given distinct nodes with associated values, the resulting polynomial is the polynomial of minimal degree which interpolates the result (agrees with the data). Within Cryptography, *Lagrange Interpolation* is used within secret sharing protocols and distributed key generation. 2, 39, 62, 65, 67, 179, 180, 196, 197

mask generation function A deterministic method for producing a digital identifier of variable length; similar to a cryptographic hash function. It specifies one way to enable a hash function to have a variable output length. Compare with extendable output function. 119, 224, 227, 267

Merkle tree A data structure consisting of leaves and nodes which allow for efficient proofs of inclusion. Uses a hash function for security. 97, 99, 100, 103, 104, 107, 109, 110, 112, 117, 270

message authentication code A method in Symmetric Key Cryptography for ensuring message integrity. MACs do not allow for nonrepudiation; for nonrepudiation, see digital signature. 72, 78, 79, 88, 117, 131, 216, 218, 268

nonce A **n**umber used only **once**. Compare with salt and initialization vector. 73, 77, 128, 130, 132–135, 138, 139, 141, 142, 147–150, 223, 268, 270

number theory The division of mathematics devoted to studying the integers. This includes the study of prime numbers and related topics. 1, 20, 39, 205–208, 231, 235

one-time pad The only encryption scheme with perfect security. The challenge is that the key must be truly random and as long as the message. 3, 5, 69, 70, 72, 269, 270

Pairing-Based Cryptography The area of cryptography which uses bilinear pairings. 3, 39, 62, 159, 164, 187, 266

perfect security An encryption scheme has *perfect security* if it is not possible to break even with *unbounded* computation. The one-time pad is the only perfectly secure encryption scheme. Also called *information-theoretic security*. 3, 5, 72, 269

permutation A function $f : A \to A$ is a permutation if f is bijective. 15, 74, 213, 266

pseudorandom number generator A method for producing cryptographically-secure random numbers derived from a seed. The full name is **cryptographically-secure pseudorandom number generator**. One example is Fortuna. 6, 76, 77, 95, 130

Public Key Cryptography The area of cryptography which uses a two keys: a public key and a private key. 1–3, 18, 27, 38, 39, 58, 69, 124, 201, 208, 266, 267, 269

public key encryption An encryption scheme which uses public keys and private keys for secure communication; part of Public Key Cryptography. 124, 129–131

random oracle The ideal cryptographic hash function with all of its desired properties. It is a useful theoretical construct; in practice, it would be instantiated with a cryptographic hash function. 82, 83, 88, 162, 212, 222, 229

recursion See recursion. x, 54, 112, 117, 269

ring A ring is a set together with two binary operations (addition and multiplication) which satisfies additional properties. A standard example of rings include the integers under addition and multiplication. 27, 32–35, 207, 266

salt A random value used as additional data when working with cryptographic hash functions. When used within a hash function, a salt ensures domain separation. Compare with nonce and initialization vector. 76, 119, 120, 154, 223, 228, 268, 269

secure channel A method for communication without concern for adversaries. For instance, when Alice and Bob communicate over a secure channel, Eve will not learn what they discuss. There is assurance of privacy. 69, 74, 124

set A set is a collection of objects. 8, 9, 11–13, 15, 30, 33, 36, 81, 231, 232, 267, 268, 270

shared secret A secret that is shared between individuals. 76, 118, 124–128, 145–147, 182–186, 190, 191, 220, 266

smart contract A smart contract is a program which may automatically perform actions in response to changes in state. Smart contracts may be implemented on blockchains such as Ethereum. 112, 159, 187, 192, 194, 267

Sparse Merkle tree A Merkle tree where most of the leaves are empty. 97, 117

stream cipher An encryption scheme which takes a secret key and stretches it into a stream of randomness. 6, 72, 73, 77, 79, 270

subgroup A group which is a subset of a larger group when using the inherited group operation. 31, 32, 38, 56, 58, 127, 135, 136, 143, 144, 202

surjective A function $f : A \to B$ is surjective if for $b \in B$ there is an $a \in A$ such that $f(a) = b$; also called *onto*. 12–14, 266

Symmetric Key Cryptography The area of cryptography which uses one secret key. 2, 69, 124, 269, 270

symmetric key encryption An encryption scheme which uses one secret key for secure communication; part of Symmetric Key Cryptography. Examples include the one-time pad, stream ciphers, and block ciphers. 72, 75, 124, 125, 127, 129, 189, 220

zero-knowledge proof A proof in which no additional information is gained other than the fact that the statement is true. Digital signatures are a standard example of zero-knowledge proofs. 3, 165, 166, 168, 171, 176, 178