Securing Deployed Cryptographic Systems

by

Christina Garman

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

Baltimore, Maryland

August, 2017

Abstract

In 2015 more than 150 million records and $400 billion were lost due to publicly-reported criminal and nation-state cyberattacks in the United States alone. The failure of our existing security infrastructure motivates the need for improved technologies, and cryptography provides a powerful tool for doing this. There is a misperception that the cryptography we use today is a "solved problem" and the real security weaknesses are in software or other areas of the system. This is, in fact, not true at all, and over the past several years we have seen a number of serious vulnerabilities in the cryptographic pieces of systems, some with large consequences.

This thesis will discuss three aspects of securing deployed cryptographic systems. We will first explore the evaluation of systems in the wild, using the example of how to efficiently and effectively recover user passwords submitted over TLS encrypted with RC4, with applications to many methods of web authentication as well as the popular IMAP protocol for email. We will then address my work on developing tools to design and create cryptographic systems and bridge the often large gap between theory and practice by introducing AutoGroup+, a tool that automatically

ABSTRACT

translates cryptographic schemes from the mathematical setting used in the literature to that typically used in practice, giving both a secure and optimal output. We will conclude with an exploration of how to actually build real world deployable systems by discussing my work on developing decentralized anonymous credentials in order to increase the security and deployability of existing anonymous credentials systems.

Primary Reader: Matthew Green

Secondary Readers: Aviel Rubin and Nadia Heninger

Acknowledgments

I would like to thank a multitude of people for their support throughout my Ph.D.

First, thank you to my family for always supporting my throughout everything, including my perpetual life of being a student. Thank you to all my labmates, both for being my co-authors as well as my friends, and for making it fun to come to the office every day. This experience would not have been the same without you. Thank you to all of the other friends that I have made throughout the years, both at Hopkins and everywhere else. I do not think I would have made it through my Ph.D. without all of you. And I always appreciated getting to see old friends at conferences.

I would like to extend a sincere thank you to Susan Hohenberger for bringing me to Hopkins initially despite not knowing where she was going to be and then for supporting me in everything throughout my entire Ph.D. Thank you to Avi Rubin for stepping in when I first arrived at Hopkins to support me and help me find my way, and for subsequently always being there when I needed something. I appreciate you recommending me for the internship at Intel Labs. My thanks to you also for serving on my thesis committee. I also extend thanks to Nadia Heninger for serving

ACKNOWLEDGMENTS

on my thesis committee, as well as letting me visit for a summer. Thank you to Kenny Paterson and everyone at Royal Holloway for welcoming me for a summer and then for being there to advise me throughout the rest of my Ph.D. And lastly, but most importantly, thank you to Matt Green for being the best advisor that a graduate student could ask for and setting an example for me to strive towards. I would not be where I am today without your time, support, advice, and guidance. Thank you for helping me grow into the researcher and teacher that I am today.

And to everyone that I have inevitably forgotten to mention, thank you as well for everything that you have done for me throughout my time at Hopkins.

I would also like to graciously acknowledge the support for this work and thesis: the National Science Foundation (NSF) CNS-1414023 and the Office of Naval Research under contract N00014-14-1-0333.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

In 2015 more than 150 million records and $400 billion were lost due to publicly-reported criminal and nation-state cyberattacks in the United States alone [1, 2]. The failure of our existing security infrastructure motivates the need for improved technologies. Cryptography provides a powerful tool for doing this. There is a misperception that the cryptography we use today is a "solved problem" and the real security weaknesses are in software or other areas of the system. This is, in fact, not true at all. Over the past several years we have seen a number of serious vulnerabilities in the cryptographic pieces of systems, some with large consequences [3, 4, 5, 6]. These can be caused by various problems, including poor designs, difficulty of implementation, and insecure primitives.

CHAPTER 1. INTRODUCTION

One of the unique aspects of applied cryptography as a research field is that it is driven by practice, including security needs and goals and even attacks on real systems. My work has focused on the security of deployed cryptographic systems in all of its aspects, including the evaluation of real systems, developing improved tools to design and create those systems, and actually creating real, deployable systems. Each of these components is critical to achieving real world, secure cryptographic systems. Analyzing and evaluating real systems is often the best method that we currently have to determine if they are secure. Many of the errors that we find when analyzing these systems could have been prevented if designers and software engineers had better tools. And experience with developing and deploying real world systems is crucial to understanding the tooling needed to best help others and where the weakpoints are in the process. Understanding and working in each area provides you with the overall picture needed to be most effective in the field of applied cryptography.

This thesis will focus on work from each of the three areas, highlighting my experiences in the complete picture of applied cryptography research. I now describe the three areas, as well as the example work from each that will be discussed later on, in more detail.

1.1.1 Evaluating, Attacking, and Securing Existing Cryptographic Protocols

Cryptographic attacks are an important part of cryptographic development, testing, and security and often are the only way to truly demonstrate problems with already deployed systems. It is a community truism that "attacks only get better," but this is often insufficient to motivate vendors to fix existing schemes. We have often seen that so-called "theoretical" attacks are not taken seriously until researchers demonstrate that something is actually broken. It is important that these attacks be done in the labs by researchers where things can then be patched, instead of happening in the wild where they can compromise user security.

Despite many recent high-profile attacks on the RC4 encryption algorithm and many recommendations from experts to cease using it, in 2015 approximately 30% of all TLS traffic, including web browsing and email, was encrypted using RC4. Our research took the position that the only way to change industry adoption of RC4 was to demonstrate and deploy practical attacks against real systems that used the algorithm. To develop such a practical attack, we focused on recovering user passwords submitted over TLS encrypted with RC4. We were able to effectively and efficiently recover passwords by carefully exploiting statistical biases in the pseudorandom output of the generator and tailoring our attack to passwords. This has potential implications for many methods of web authentication as well as the popular IMAP protocol for

email. Demonstrating a believable, practical attack forced a change in the community and a recognition that RC4 was well and truly broken.

This sort of work can have demonstrable real world impact both within the community and for the general public. RC4 has now been removed from all major web browsers [7] and prohibited by the IETF in all TLS implementations [8].

1.1.2 Tools for Developing Secure Cryptographic Systems

The second part of my research deals with developing tools that can make it easier for both experts and non-experts to develop secure cryptographic systems. As we have seen repeatedly, developing secure cryptographic systems is hard. Whether it is difficulties in translating schemes developed by researchers to code, mistakes in implementation, or poor optimizations, we are often able to find flaws in deployed systems. Cryptographic automation is a new and promising area that is designed to help solve many of these problems and make developing secure systems easier and less error-prone. This approach has already shown promise. Various tools exist to help cryptographers create security proofs in an automated, computer-assisted manner [9, 10], while other lines of research help automate the creation of new cryptographic schemes [11, 12].

Pairing-based cryptography has become very popular over the last decade, as

this algebraic setting offers good functionality and efficiency. It is used in many popular cryptographic schemes, such as Identity-Based Encryption [13] and Attribute-Based Encryption [14], and pairing-commercializer Voltage Security was acquired by a major US company (HP) [15]. However, there is a huge security gap between how schemes are usually analyzed in the academic literature and how they are typically implemented. To address this, our contribution was the design, development and evaluation of a new software tool, AutoGroup+, that automatically translates from the mathematical setting used in the literature to that typically used in practice. The output of AutoGroup+ is secure and optimal based on the user's efficiency constraints [16]. AutoGroup+ is able to take a cryptographic scheme written in a special language, similar to LaTeX, as input and then output both C and Python code of the transformed scheme.

This sort of tool can be extremely valuable for both experts and non-experts. Experts can use it to check their results or reduce manual (often error-prone) work. Non-experts can use it to translate schemes from research into practical, useable schemes for their systems, without having to attempt to do the translation or implementation by hand. We hope that such tools will help aid in the future development of secure cryptographic systems.

1.1.3 New Domains for Cryptographic Applications

Because cryptography is so ubiquitous in the world today, the ability to actually develop and deploy secure systems is an important one. This is not an easy task, as many different design decisions and tradeoffs must be considered before one even reaches the point of designing the cryptography.

The third part of my research involves actually creating and implementing practical, useable (secure) systems to solve real world problems. While this thesis will not discuss it, our first work in this space [17] involves Bitcoin, the first electronic cash system to see widespread adoption. While Bitcoin offers the potential for new types of financial interaction, it has significant limitations regarding privacy. Specifically, because the Bitcoin transaction log is completely public, users have little privacy. We created Zerocoin, a cryptographic extension to Bitcoin that augments the protocol to allow for fully anonymous currency transactions and implemented a proof of concept system. Zerocoin leverages cryptographic zero-knowledge proofs and other techniques to provide a privacy layer on top of the already popular Bitcoin protocol. While this thesis will not include Zerocoin, this brief high-level overview is useful for introducing us to my work that will be discussed.

The applications of our work on Zerocoin go far beyond electronic cash. Indeed, they provided us with a new technique that allowed us to develop decentralized anonymous credentials in order to increase the security and deployability of existing anonymous credentials systems [18], which will be discussed in this thesis. Anonymous credentials

provide a powerful tool for making assertions about identity while maintaining privacy. However, a limitation of today's anonymous credential systems is the need for a trusted credential issuer — which is both a single point of failure and a target for compromise. Furthermore, the need for such a trusted issuer can make it challenging to deploy credential systems in practice, particularly in the ad hoc network setting (e.g., anonymous peer-to-peer networks) where no single party can be trusted with this responsibility. We proposed a novel anonymous credential scheme that eliminates the need for a trusted credential issuer and implemented the system to show its practicality. We also provided a number of practical applications for our techniques, including resource management in ad hoc networks and prevention of Sybil attacks.

1.1.4 Organization

In the next three chapters, we present the aforementioned work from each of these three areas. We begin by discussing cryptographic attacks, with my work on decrypting user passwords encrypted with RC4 in TLS. After describing how systems can fail and where things can go wrong, we move on to discuss ways to build more secure systems and prevent problems from ever occurring by using cryptographic automation and creating AutoGroup+. We conclude by discussing how to build cryptographic systems in practice, because it is both impossible to work in either of the previous areas and be a successful applied cryptography researcher without understanding this; this section will focus on our work on decentralized anonymous credentials. Each chapter is drawn

from the corresponding publication with little modification or additional content.

Chapter 2

Evaluating, Attacking, and Securing Existing Cryptographic Protocols

This chapter is based on joint work with Kenneth G Paterson and Thyla van der Merwe while visiting Royal Holloway, University of London. The paper was originally published in USENIX Security 2015 [19], while the complete version that appears in this chapter was published at [20].

It is a truism in the community that "attacks only get better", but this is often insufficient to motivate fixes to existing schemes. As we have seen many times, only when things are well and truly broken will they be fixed. This chapter demonstrates a real life example of this situation, where a scheme with known problems was continually used until a truly practical attack was demonstrated in a realistic application. We now describe our work on decrypting user passwords encrypted with TLS using RC4.

2.1 Introduction

The year 2013 was a(nother) bad one for the TLS protocol. After the trauma of BEAST in 2011 and CRIME in 2012, came the Lucky 13 attack on CBC-mode in TLS [21] and then, shortly after, attacks on RC4 in TLS [22] (see also [23, 24]). At the time, most TLS (and SSL) deployments used either CBC-mode or RC4, since TLS 1.2 with its more modern ciphersuites based on AES-GCM was barely deployed. Indeed, according to statistics obtained from the International Computer Science Institute (ICSI) Certificate Notary project,[1] in January 2013, there was a roughly 50/50 split between CBC-mode and RC4 usage in the wild. And, at that time, no major browser supported TLS 1.2 in its default settings.

Fast forward to February 2015: all mainstream browsers now support TLS 1.2, TLS 1.3 is under development in the IETF, and the IETF has just published an RFC deprecating the use of RC4 in TLS [8]. Moreover, statistics from SSL Pulse[2] show that server-side support for TLS 1.2 has grown rapidly, from less than 11.4% of the servers surveyed in January 2013 to 54.5% in February 2015. At the same time, large vendors have started to remove support for RC4. For example, Microsoft have made it possible to disable RC4 across a wide range of their products,[3] while CloudFlare, a major CDN provider, recently removed RC4 ciphersuites from all their server configurations.[4]

[1] The ICSI Certificate Notary project collects statistics from live upstream SSL/TLS traffic in a passive manner; see `http://notary.icsi.berkeley.edu`

[2] `https://www.trustworthyinternet.org/ssl-pulse/`

[3] See urlhttps://technet.microsoft.com/library/security/2868725.

[4] See `https://blog.cloudflare.com/end-of-the-road-for-rc4/`.

At first sight, then, it would seem that the world has paid close attention to the TLS attacks of 2013 and taken steps to remediate them. Most major implementations patched against Lucky 13,[5] while, according to [22], the only reasonable response to the RC4 attacks was to stop using RC4 in TLS. However, the reality is not so rosy. For example, SSL Pulse shows that, in February 2015, 74.5% of the roughly 150,000 sites surveyed still allowed negotiation of RC4. Even worse, a January 2015 survey[6] of about 400,000 of the Alexa top 1 million sites show that 3712 of them, or 0.79%, support *only* RC4 ciphersuites; meanwhile 8.75% force the use of RC4 in TLS 1.1 and 1.2, where better ciphers are available. And March 2015 data from the ICSI Certificate Notary project shows that more than 30% of SSL/TLS connections are still using RC4.[7]

It is instructive to examine the reasons why RC4 still remains so popular in TLS, and why deprecating its use seems to be so hard.

We assert that, first and foremost, this is because, while the attacks of [22] break RC4 in TLS in an academic sense, the attacks are far from being practical. For example, the preferred cookie-recovering attack in [22] needs around $2^{33} - 2^{34}$ encryptions of a 16-byte, base64-encoded secure cookie to reliably recover it. The number is so high because, with mainstream browsers and taking into account the verbosity of the HTTP protocol, the target cookie is not located near the start of the RC4 keystream,

[5] See `http://www.isg.rhul.ac.uk/tls/lucky13.html` for a list.

[6] `https://securitypitfalls.wordpress.com/2015/02/01/january-2015-scan-results/`

[7] An exact figure is hard to determine because of the "other" category in the relevant data which is currently running at 4.9% and which may include some RC4-protected traffic.

meaning that the strong, single-byte keystream biases in RC4 observed in [22] cannot be exploited. Rather, the preferred attack from [22] uses the much weaker, long-term Fluhrer-McGrew double-byte biases from [25]. This substantially increases the number of required encryptions before the plaintext cookie can be reliably recovered, to the point where, even with highly-tuned malicious JavaScript running in the victim's browser generating 6 million cookie-bearing HTTP POST requests per hour, the wall-clock time to execute the attack would still be on the order of 2000 hours using the experimental setup reported in [22]; moreover the attack would generate many Terabytes of network traffic. Thus the practical threat posed by the RC4 attacks reported in [22] is arguably quite limited.

The second reason for the continued popularity of RC4 in TLS is the presence of legacy implementations which support only RC4, or which would be vulnerable to the BEAST or Lucky 13 attacks on CBC-mode ciphersuites. Indeed, switching to RC4 was a widely recommended countermeasure to the BEAST attack. Websites are naturally loath to lose potential customers whose browsers are not equipped with the latest patches or TLS versions and ciphersuites. Countering this, recent data from CloudFlare[8] shows that very few TLS clients now actually *need* RC4 – 0.0009% of connections in the CloudFlare data. There is also a plethora of TLS deployment not protecting HTTP traffic, increasingly including smart metering systems, industrial control systems, and server-to-server communications. A further complexity, revealed

[8]See `http://blog.cloudflare.com/killing-rc4-the-long-goodbye/` and `http://blog.cloudflare.com/the-web-is-world-wide-or-who-still-needs-rc4/`

in the CloudFlare data, is the presence of TLS-based VPNs or firewalls that perform man-in-the-middle monitoring of SSL connections but that may need updating to stop them from using RC4 for secure connections.

A third reason is performance: in settings where hardware support for AES is not available (e.g. most mobile processors, including ARM processors), RC4 is fast, significantly outperforming AES-based ciphersuites. There are certainly more secure stream ciphers that are equally fast, or faster (for example, algorithms in the eStream portfoliom[9]) but they are not standardised for use in TLS, and their widespread deployment would take years.[10]

Nevertheless, it is a well-worn cliché that attacks only get better (i.e. stronger) with time. However, this is a cliché that happens to be true for TLS, with the BEAST, Lucky 13 and POODLE attacks all being illustrative examples. This paper presents another illustration of this phenomenon for RC4 in TLS. We present attacks recovering TLS-protected passwords whose ciphertext requirements are significantly reduced compared to those of [22]: we achieve a reduction from 2^{34} ciphertexts down to 2^{26} – 2^{28}. We also describe proof-of-concept implementations of these attacks against specific application-layer protocols making use of passwords, namely BasicAuth and IMAP.

[9] `http://www.ecrypt.eu.org/stream/`

[10] An exception is ChaCha20, which is experimentally deployed by Google in its Chrome browser and for which a specification is nearing completion in IETF, see `https://tools.ietf.org/html/draft-irtf-cfrg-chacha20-poly1305-08`.

2.1.1 Our Contributions

In this chapter, we revisit the statistical methods of [22], refining, extending and applying them to the specific problem of recovering TLS-protected passwords. Our target is to reduce as much as possible the ciphertext requirements of the original RC4 attacks from [22]. Our overall objective is to bring the use of RC4 in TLS closer to the point where it becomes indefensible and must be abandoned. This seems particularly important in view of persistent rumours about the ability of nation-state adversaries to break RC4 in real time, and the apparent need to strengthen the attacks in order to convince practitioners to move to better ciphers.

Passwords are a good target for our attacks because they are still very widely used on the Internet for providing user authentication, and are frequently protected using TLS to prevent them being passively eavesdropped. It is true that major websites use secure cookies for managing user authentication, but the authentication is usually bootstrapped via password entry. However, to build effective attacks, we need to find and exploit systems in which users' passwords are automatically and repeatedly sent under the protection of TLS, so that sufficiently many ciphertexts can be gathered for our statistical analyses.

2.1.1.0.1 Bayesian analysis

We present a formal Bayesian analysis that combines an *a priori* plaintext distribution with keystream distribution statistics to produce *a posteriori* plaintext likelihoods.

This analysis formalises and extends the procedure followed in [22] for single-byte attacks. There, only keystream distribution statistics were used (specifically, biases in the individual bytes in the early portion of the RC4 keystream) and plaintexts were assumed to be uniformly distributed, while here we also exploit (partial) knowledge of the plaintext distribution to produce a more accurate estimate of the *a posteriori* likelihoods. This yields a procedure that is optimal (in the sense of yielding a maximum *a posteriori* estimate for the plaintext) if the plaintext distribution is known exactly. In the context of password recovery, an *estimate* for the *a priori* plaintext distribution can be empirically formed by using data from password breaches or by synthetically constructing password dictionaries. We will demonstrate, via simulations, that this Bayesian approach improves performance (measured in terms of success rate of plaintext recovery for a given number of ciphertexts) compared to the approach in [22].

Our Bayesian analysis concerns vectors of consecutive plaintext bytes, which is appropriate given passwords as the plaintext target. This however means that the keystream distribution statistics also need to be for vectors of consecutive keystream bytes. Such statistics do not exist in the prior literature on RC4, except for the Fluher-McGrew biases [25] (which supply the distributions for adjacent byte pairs far down the keystream). Fortunately, in the early bytes of the RC4 keystream, the single-byte biases are dominant enough that a simple product distribution can be used as a reasonable estimate for the distribution on vectors of keystream bytes. We

also show how to build a more accurate approximation to the relevant keystream distributions using double-byte distributions. (Obtaining the double-byte distributions to a suitable degree of accuracy consumed roughly 4800 core-days of computation; for details, see Appendix A.) This approximation is not only more accurate but also *necessary* when the target plaintext is located further down the stream, where the single-byte biases disappear and where double-byte biases become dominant. Indeed, our double-byte-based approximation to the keystream distribution on vectors can be used to smoothly interpolate between the region where single-byte biases dominate and where the double-byte biases come into play (which is exhibited as a fairly sharp transition around position 256 in the keystream).

In the end, what we obtain is a formal algorithm that estimates the likelihood of each password in a dictionary based on both the *a priori* password distribution and the observed ciphertexts. This formal algorithm is amenable to efficient implementation using either the single-byte based product distribution for keystreams or the double-byte-based approximation to the distribution on keystreams. The dominant terms in the running time for both of the resulting algorithms is $\mathcal{O}(nN)$ where n is the length of the target password and N is the size of the dictionary used in the attack.

A major advantage of our new algorithms over the previous work in [22] is that they output a value for the likelihood of each password candidate, enabling these to be ranked and then tried in order in against a user's account. This fits neatly with how password authentication often works in practice: users are given a pre-determined

number of tries before their account locks out.

2.1.1.0.2 Evaluation

We evaluate and compare our password recovery algorithms through extensive simulations, exploring the relationships between the main parameters of our attack:

- The length n of the target password.
- The number S of available encryptions of the password.
- The starting position r of the password in the plaintext stream.
- The size N of the dictionary used in the attack, and the availability (or not) of an *a priori* password distribution for this dictionary.
- The number of tries T made (meaning that our algorithm is considered successful if it ranks the correct password amongst the top T passwords, i.e. the T passwords with highest likelihoods as computed by the algorithm).
- Which of our two algorithms is used (the one computing the keystream statistics using the product distribution or the one using a double-byte-based approximation).
- Whether the passwords are base64 encoded before being transmitted, or are sent as raw ASCII/Unicode.

Naturally, given the combinatorial explosion of possible parameter settings (and the cost of performing simulations), we focus on comparing the performance with all but one or two parameters or variables being fixed in each instance.

2.1.1.0.3 Proofs of concept

Our final contribution is to identify and apply our attacks to two specific and widely-deployed applications making use of passwords over TLS: BasicAuth and IMAP. In each case, we introduce the application and describe a proof-of-concept implementation of our attacks against it, giving an indication of the practicality of our attacks in each case.

For both applications, we have significant success rates with only $S = 2^{26}$ ciphertexts, in contrast to the 2^{34} ciphertexts required in [22]. This is because we are able to force the target passwords into the first 256 bytes of plaintext, where the single-byte keystream biases come into play. For example, with $S = 2^{26}$ ciphertexts, we would expect to recover a length 6 BasicAuth password with 44.5% success rate with $T = 5$ tries; the rate rises to 64.4% if $T = 100$ tries are made. In practice, many sites do not configure any limit on the number of BasicAuth attempts made by a client; moreover a study [26] showed that 84% of websites surveyed allowed for up to 100 password guesses (though these sites were not necessarily using BasicAuth as their authentication mechanism). As we will show, our result compares very favourably to the previous attacks and to random guessing of passwords without any reference to

the ciphertexts.

However, there is a downside too: to make use of the early, single-byte biases in RC4 keystreams, we have to repeatedly cause TLS connections to be closed and new ones to be opened. Because of latency in the TLS Handshake Protocol, this leads to a significant slowdown in the wall clock running time of the attack; for $S = 2^{26}$, a fairly low latency of 100ms, and exploiting browsers' propensity to open multiple parallel connections, we estimate a running time of around 300 hours for the attack. This is still more than 6 times faster than the 2000 hours estimated in [22].

2.1.2 Chapter Organization

In Section 2.2 we provide further background on the RC4 stream cipher, the TLS record protocol and its use of RC4, and password distributions. In Section 2.3 we present our attacks; we evaluate them via simulation in Section 2.4. In Section 2.5 we explore the application of our attacks to BasicAuth and IMAP. We conclude in Section 2.6.

2.2 Further Background

2.2.1 The RC4 algorithm

Originally a proprietary stream cipher designed by Ron Rivest in 1987, RC4 is remarkably fast when implemented in software and has a very simple description. Details of the cipher were leaked in 1994 and the cipher has been subject to public analysis and study ever since.

RC4 allows for variable-length key sizes, anywhere from 40 to 256 bits, and consists of two algorithms, namely, a *key scheduling algorithm* (KSA) and a *pseudo-random generation algorithm* (PRGA). The KSA takes as input an l-byte key and produces the initial internal state $st_0 = (i, j, \mathcal{S})$ for the PRGA; $\mathcal{S}$ is the canonical representation of a permutation of the numbers from 0 to 255 where the permutation is a function of the l-byte key, and i and j are indices for $\mathcal{S}$. The KSA is specified in Algorithm 1 where K represents the l-byte key array and $\mathcal{S}$ the 256-byte state array. Given the internal state st_r, the PRGA will generate a keystream byte Z_{r+1} as specified in Algorithm 2.

2.2.2 Single-byte biases in the RC4 Keystream

RC4 has several cryptographic weaknesses, notably the existence of various biases in the RC4 keystream, see for example [25, 27, 28, 29, 22]. Large single-byte biases are prominent in the early postions of the RC4 keystream. Mantin and Shamir [27]

Algorithm 1: RC4 key scheduling (KSA)

input : key K of l bytes
output : initial internal state st_0
begin
  for $i = 0$ **to** 255 **do**
    $\mathcal{S}[i] \leftarrow i$
  $j \leftarrow 0$
  for $i = 0$ **to** 255 **do**
    $j \leftarrow j + \mathcal{S}[i] + K[i \bmod l]$
    swap$(\mathcal{S}[i], \mathcal{S}[j])$
  $i, j \leftarrow 0$
  $st_0 \leftarrow (i, j, \mathcal{S})$
  return st_0

observed the first of these biases, in Z_2 (the second byte of the RC4 keystream), and showed how to exploit it in what they called a *broadcast attack*, wherein the same plaintext is repeatedly encrypted under different keys. AlFardan *et al.* [22] performed large-scale computations to estimate these early biases, using 2^{45} keystreams to compute the single-byte keystream distributions in the first 256 output positions. They also provided a statistical approach to recovering plaintext bytes in the broadcast attack scenario, and explored its exploitation in TLS. Much of the new bias behaviour they observed was subsequently explained in [30]. Unfortunately, from an attacker's perspective, the single-byte biases die away very quickly beyond position 256 in the RC4 keystream. This means that they can only be used in attacks to extract plaintext bytes which are found close to the start of plaintext streams. This was a significant complicating factor in the attacks of [22], where, because of the behaviour of HTTP in modern browsers, the target HTTP secure cookies were not so located.

Algorithm 2: RC4 keystream generator (PRGA)

input : internal state st_r
output : keystream byte Z_{r+1} updated internal state st_{r+1}
begin
  parse $(i, j, \mathcal{S}) \leftarrow st_r$
  $i \leftarrow i + 1$
  $j \leftarrow j + \mathcal{S}[i]$
  swap$(\mathcal{S}[i], \mathcal{S}[j])$
  $Z_{r+1} \leftarrow \mathcal{S}[\mathcal{S}[i] + \mathcal{S}[j]]$
  $st_{r+1} \leftarrow (i, j, \mathcal{S})$
  return (Z_{r+1}, st_{r+1})

2.2.3 Double-byte biases in the RC4 Keystream

Fluhrer and McGrew [25] showed that there are biases in adjacent bytes in RC4 keystreams, and that these so-called double-byte biases are persistent throughout the keystream. The presence of these long-term biases (and the absence of any other similarly-sized double-byte biases) was confirmed computationally in [22]. AlFardan *et al.* [22] also exploited these biases in their double-byte attack to recover HTTP secure cookies.

Because we wish to exploit double-byte biases in early portions of the RC4 keystream and because the analysis of [25] assumes the RC4 permutation $\mathcal{S}$ is uniformly random (which is not the case for early keystream bytes), we carried out extensive computations to estimate the initial double-byte keystream distributions: we used roughly 4800 core-days of computation to generate 2^{44} RC4 keystreams for random 128-bit RC4 keys (as used in TLS); we used these keystreams to estimate the

double-byte keystream distributions for RC4 in the first 512 positions.

While the gross behaviour that we observed is dominated by products of the known single-byte biases in the first 256 positions and by the Fluhrer-McGrew biases in the later positions, we did observe some new and interesting double-byte biases. Since these are likely to be of independent interest to researchers working on RC4, we report in more detail on this aspect of our work in Appendix A.

2.2.4 RC4 and the TLS Record Protocol

We provide an overview of the TLS Record Protocol with RC4 selected as the method for encryption and direct the reader to [22, 31, 32, 33] for further details.

Application data to be protected by TLS, i.e, a sequence of bytes or a record R, is processed as follows: An 8-byte sequence number `SQN`, a 5-byte header `HDR` and R are concatenated to form the input to an HMAC function. We let T denote the resulting output of this function. In the case of RC4 encryption, the plaintext, $P = T||R$, is XORed byte-per-byte with the RC4 keystream. In other words,

$$C_r = P_r \oplus Z_r,$$

for the r^{th} bytes of the ciphertext, plaintext and RC4 keystream respectively (for $r = 1, 2, 3 \dots$). The data that is transmitted has the form $\texttt{HDR}||C$, where C is the concatenation of the individual ciphertext bytes.

The RC4 algorithm is intialized in the standard way at the start of each TLS connection with a 128-bit encryption key. This key, K, is derived from the TLS master secret that is established during the TLS Handshake Protocol; K is either established via the the full TLS Handshake Protocol or TLS session resumption. The first few bytes to be protected by RC4 encryption is a `Finished` message of the TLS Handshake Protocol. We do not target this record in our attacks since this message is not constant over multiple sessions. The exact size of this message is important in dictating how far down the keystream our target plaintext will be located; in turn this determines whether or not it can be recovered using only single-byte biases. A common size is 36 bytes, but the exact size depends on the output size of the TLS PRF used in computing the `Finished` message and of the hash function used in the HMAC algorithm in the record protocol.

Decryption is the reverse of the process described above. As noted in [22], any error in decryption is treated as fatal – an error message is sent to the sender and all cryptographic material, including the RC4 key, is disposed of. This enables an active attacker to force the use of new encryption and MAC keys: the attacker can induce session termination, followed by a new session being established when the next message is sent over TLS, by simply modifying a TLS Record Protocol message. This could be used to ensure that the target plaintext in an attack is repeatedly sent under the protection of a fresh RC4 key. However, this approach is relatively expensive since it involves a rerun of the full TLS Handshake Protocol, involving

multiple public key operations and, more importantly, the latency involved in an exchange of 4 messages (2 complete round-trips) on the wire. A better approach is to cause the TCP connection carrying the TLS traffic to close, either by injecting sequences of `FIN` and `ACK` messages in both directions, or by injecting a `RST` message in both directions. This causes the TLS *connection* to be terminated, but not the TLS *session* (assuming the session is marked as "resumable" which is typically the case). This behaviour is codified in [33, Section 7.2.1]. Now when the next message is sent over TLS, a TLS session resumption instance of the Handshake Protocol is executed to establish a fresh key for RC4. This avoids the expensive public key operations and reduces the TLS latency to 1 round-trip before application data can be sent. On large sites, session resumption is usually handled by making use of TLS session tickets [34] on the server-side.

2.2.5 Passwords

Text-based passwords are arguably the dominant mechanism for authenticating users to web-based services and computer systems. As is to be expected of user-selected secrets, passwords do not follow uniform distributions. Various password breaches of recent years, including the Adobe breach of 150 million records in 2013 and the RockYou leak of 32.6 million passwords in 2009, attest to this with passwords such as `123456` and `password` frequently being counted amongst the most popular.[11] For

[11] A comprehensive list of data breaches, including password breaches, can be found at `http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/`.

example, our own analysis of the RockYou password data set confirmed this: the number of unique passwords in the RockYou dataset is 14,344,391, meaning that (on average) each password was repeated 2.2 times, and we indeed found the most common password to be `123456` (accounting for about 0.9% of the entire data set). Our later simulations will make extensive use of the RockYou data set as an attack dictionary. A more-fine grained analysis of it can be found in [35]. We also make use of data from the Singles.org breach for generating our target passwords. Singles.org is a now-defunct Christian dating website that was breached in 2009; religiously-inspired passwords such as `jesus` and `angel` appear with high frequency in its 12,234 distinct entries, making its frequency distribution quite different from that of the RockYou set.

There is an extensive literature regarding the reasons for poor password selection and usage, including [36, 37, 38, 39]. In [40], Bonneau formalised a number of different metrics for analysing password distributions and studied a corpus of 70M Yahoo! passwords (collected in a privacy-preserving manner). His work highlights the importance of careful validation of password guessing attacks, in particular, the problem of estimating attack complexities in the face of passwords that occur rarely – perhaps uniquely – in a data set, the so-called *hapax legomena* problem. The approach to validation that we adopt benefits from the analysis of [40], as explained further in Section 2.4.

2.3 Plaintext Recovery via Bayesian Analysis

In this section, we present a formal Bayesian analysis of plaintext recovery attacks in the broadcast setting for stream ciphers. We then apply this to the problem of extracting passwords, specialising the formal analysis and making it implementable in practice based only on the single-byte and double-byte keystream distributions.

2.3.1 Formal Bayesian Analysis

Suppose we have a candidate set of N plaintexts, denoted $\mathcal{X}$, with the *a priori* probability of an element $x \in \mathcal{X}$ being denoted p_x. We assume for simplicity that all the candidates consist of byte strings of the same length n. For example $\mathcal{X}$ might consist of all the passwords of a given length n from some breach data set, and then p_x can be computed as the relative frequency of x in the data set. If the frequency data is not available, then the uniform distribution on $\mathcal{X}$ can be assumed.

Next, suppose that a plaintext from $\mathcal{X}$ is encrypted S times, each time under independent, random keys using a stream cipher such as RC4. Suppose also that the first character of the plaintext always occurs in the same position r in the plaintext stream in each encryption. Let $c = (c_{ij})$ denote the $S \times n$ matrix of bytes in which row i, denoted $c^{(i)}$ for $0 \leq i < S$, is a vector of n bytes corresponding to the values in positions $r, \ldots, r + n - 1$ in ciphertext i. Let X be the random variable denoting the

(unknown) value of the plaintext.

We wish to form a maximum a posteriori (MAP) estimate for X, given the observed data c and the *a priori* probability distribution p_x, that is, we wish to maximise $\Pr(X = x \mid C = c)$ where C is a random variable corresponding to the matrix of ciphertext bytes.

Using Bayes' theorem, we have

$$\Pr(X = x \mid C = c) = \Pr(C = c \mid X = x) \cdot \frac{\Pr(X = x)}{\Pr(C = c)}.$$

Here the term $\Pr(X = x)$ corresponds to the *a priori* distribution p_x on $\mathcal{X}$. The term $\Pr(C = c)$ is independent of the choice of x (as can be seen by writing $\Pr(C = c) = \sum_{x \in \mathcal{X}} \Pr(C = c \mid X = x) \cdot \Pr(X = x)$). Since we are only interested in maximising $\Pr(X = x \mid C = c)$, we ignore this term henceforth.

Now, since ciphertexts are formed by XORing keystreams z and plaintext x, we can write

$$\Pr(C = c \mid X = x) = \Pr(W = w)$$

where w is the $S \times n$ matrix formed by XORing each row of c with the vector x and W is a corresponding random variable. Then to maximise $\Pr(X = x \mid C = c)$, it suffices to maximise the value of

$$\Pr(X = x) \cdot \Pr(W = w)$$

over $x \in \mathcal{X}$. Let $w^{(i)}$ denote the i-th row of the matrix w, so $w^{(i)} = c^{(i)} \oplus x$. Then $w^{(i)}$ can be thought of as a vector of keystream bytes (coming from positions $r, \ldots r+n-1$) induced by the candidate x, and we can write

$$\Pr(W = w) = \prod_{i=0}^{S-1} \Pr(Z = w^{(i)})$$

where, on the right-hand side of the above equation, Z denotes a random variable corresponding to a vector of bytes of length n starting from position r in the keystream. Writing $\mathcal{B} = \{\texttt{0x00}, \ldots \texttt{0xFF}\}$ for the set of bytes, we can rewrite this as:

$$\Pr(W = w) = \prod_{z \in \mathcal{B}^n} \Pr(Z = z)^{N_{x,z}}$$

where the product is taken over all possible byte strings of length n and $N_{x,z}$ is defined as:

$$N_{x,z} = |\{i : z = c^{(i)} \oplus x\}_{0 \le i < S}|,$$

that is, $N_{x,z}$ counts the number of occurrences of vector z in the rows of the matrix formed by XORing each row of c with candidate x. Putting everything together, our objective is to compute for each candidate $x \in \mathcal{X}$ the value:

$$\Pr(X = x) \cdot \prod_{z \in \mathcal{B}^n} \Pr(Z = z)^{N_{x,z}}$$

and then to rank these values in order to determine the most likely candidate(s).

Notice that the expressions here involve terms $\Pr(Z = z)$ which are probabilities of occurrence for n consecutive bytes of keystream. Such estimates are not generally available in the literature, and for the values of n we are interested in (corresponding to putative password lengths), obtaining accurate estimates for them by sampling many keystreams would be computationally prohibitive. For example, our computation for double-byte probabilities discussed in Appendix A involved 2^{44} keystreams and, with highly optimised code, consumed roughly 4800 core-days of computation. This yields the required probabilities only for $n = 2$. Moreover, the product $\prod_{z \in \mathcal{B}^n}$ involves 2^{8n} terms and is not amenable to calculation. Thus we must turn to approximate methods to make further progress.

Note also that taking $n = 1$ in the above analysis, we obtain exactly the same approach as was used in the single-byte attack in [22], except that we include the *a priori* probabilities $\Pr(X = x)$ whereas these were (implicitly) assumed to be uniform in [22].

2.3.2 Using a Product Distribution

Our task is to derive simplified ways of computing the expression

$$\Pr(X = x) \cdot \prod_{z \in \mathcal{B}^n} \Pr(Z = z)^{N_{x,z}}$$

and then apply these to produce efficient algorithms for computing (approximate) likelihoods of candidates $x \in \mathcal{X}$.

The simplest approach is to assume that the n bytes of the keystreams can be treated independently. For RC4, this is actually a very good approximation in the regime where single-byte biases dominate (that is, in the first 256 positions). Thus, writing $Z = (Z_r, \ldots, Z_{r+n-1})$ and $z = (z_r, \ldots, z_{r+n-1})$ (with the subscript r denoting the position of the first keystream byte of interest), we have:

$$\Pr(Z = z) \approx \prod_{j=0}^{n-1} \Pr(Z_{r+j} = z_{r+j}) = \prod_{j=0}^{n-1} p_{r+j,z}$$

where now the probabilities appearing on the right-hand side are single-byte keystream probabilities, as reported in [22] for example. Then writing $x = (x_0, \ldots, x_{n-1})$ and rearranging terms, we obtain:

$$\prod_{z \in \mathcal{B}^n} \Pr(Z = z)^{N_{x,z}} \approx \prod_{j=0}^{n-1} \prod_{z \in \mathcal{B}} p_{r+j,z}^{N_{x_j,z,j}}.$$

where $N_{y,z,j} = |\{i : z = c_{i,j} \oplus y\}_{0 \le i < S}|$ counts (now for single bytes instead of length n vectors of bytes) the number of occurrences of byte z in the column vector formed by XORing column j of c with a candidate byte y.

Notice that, as in [22], the counters $N_{y,z,j}$ for $y \in \mathcal{B}$ can all be computed efficiently by permuting the counters $N_{\texttt{0x00},z,j}$, these being simply counters for the number of occurrences of each byte value z in column j of the ciphertext matrix c.

In practice, it is more convenient to work with logarithms, converting products into sums, so that we evaluate for each candidate $x = (x_0, \ldots, x_{n-1})$ an expression of the form

$$\gamma_x := \log(p_x) + \sum_{j=0}^{n-1} \sum_{z \in \mathcal{B}} N_{x_j,z,j} \log(p_{r+j,z}).$$

Given a large set of candidates $\mathcal{X}$, we can streamline the computation by first computing the counters $N_{y,z,j}$, then, for each possible byte value y, the value of the inner sum $\sum_{z \in \mathcal{B}} N_{y,z,j} \log(p_{r+j,z})$, and then reusing these individual values across all the relevant candidates x for which $x_j = y$. This reduces the evaluation of γ_x for a single candidate x to $n + 1$ additions of real numbers.

The above procedure, including the various optimizations, is specified as an attack in Algorithm 3. We refer to it as our single-byte attack because of its reliance on the single-byte keystream probabilities $p_{r+j,z}$. It outputs a collection of approximate log likelihoods $\{\gamma_x : x \in \mathcal{X}\}$ for each candidate $x \in \mathcal{X}$. These can be further processed to extract, for example, the candidate with the highest score, or the top T candidates.

2.3.3 Double-byte-based Approximation

We continue to write $Z = (Z_r, \ldots, Z_{r+n-1})$ and $z = (z_r, \ldots, z_{r+n-1})$ and aim to find an approximation for $\Pr(Z = z)$ which lends itself to efficient computation of approximate log likelihoods as in our first algorithm. Now we rely on the double-byte

Algorithm 3: Single-byte attack

input : $c_{i,j} : 0 \le i < S, 0 \le j < n$ – array formed from S independent encryptions of fixed n-byte candidate X
r – starting position of X in plaintext stream
$\mathcal{X}$ – collection of N candidates
p_x – *a priori* probability of candidates $x \in \mathcal{X}$
$p_{r+j,z}$ $(0 \le j < n,\ z \in \mathcal{B})$ – single-byte keystream distribution

output : $\{\gamma_x : x \in \mathcal{X}\}$ – set of (approximate) log likelihoods for candidates in $\mathcal{X}$

begin

- **for** $j = 0$ **to** $n - 1$ **do**
  - **for** $z =$ `0x00` **to** `0xFF` **do**
    - $N'_{z,j} \leftarrow 0$
- **for** $j = 0$ **to** $n - 1$ **do**
  - **for** $i = 0$ **to** $S - 1$ **do**
    - $N'_{c_{i,j},j} \leftarrow N'_{c_{i,j},j} + 1$
- **for** $j = 0$ **to** $n - 1$ **do**
  - **for** $y =$ `0x00` **to** `0xFF` **do**
    - **for** $z =$ `0x00` **to** `0xFF` **do**
      - $N_{y,z,j} \leftarrow N'_{z \oplus y,j}$
    - $L_{y,j} = \sum_{z \in \mathcal{B}} N_{y,z,j} \log(p_{r+j,z})$,
- **for** $x = (x_0, \ldots, x_{n-1}) \in \mathcal{X}$ **do**
  - $\gamma_x \leftarrow \log(p_x) + \sum_{j=0}^{n-1} L_{x_j,j}$
- **return** $\{\gamma_x : x \in \mathcal{X}\}$

keystream distribution, writing

$$p_{s,z_1,z_2} := \Pr((Z_s, Z_{s+1}) = (z_1, z_2)), \quad s \geq 1, k_1, k_2 \in \mathcal{B}$$

for the probabilities of observing bytes (z_1, z_2) in the RC4 keystream in positions $(s, s+1)$. We estimated these probabilities for r in the range $1 \leq r \leq 511$ using 2^{44} RC4 keystreams – for details, see Appendix A; for larger r, these are well approximated by the Fluhrer-McGrew biases [25] (as was verified in [22]).

We now make the Markovian assumption that, for each j,

$$\begin{aligned}\Pr(Z_j = z_j \mid Z_{j-1} = z_{j-1} \wedge \cdots \wedge Z_0 = z_0) \\ \approx \Pr(Z_j = z_j \mid Z_{j-1} = z_{j-1}),\end{aligned}$$

meaning that byte j in the keystream can be modelled as depending only on the preceding byte and not on earlier bytes. We can write

$$\Pr(Z_j = z_j \mid Z_{j-1} = z_{j-1}) = \frac{\Pr(Z_j = z_j \wedge Z_{j-1} = z_{j-1})}{\Pr(Z_{j-1} = z_{j-1})}$$

where the numerator can then be replaced by p_{j-1,z_{j-1},z_j} and the denominator by $p_{j-1,z_{j-1}}$, a single-byte keystream probability. Then using an inductive argument and

our assumption, we easily obtain:

$$\Pr(Z = z) \approx \frac{\prod_{j=0}^{n-2} p_{r+j,z_j,z_{j+1}}}{\prod_{j=1}^{n-2} p_{r+j,z_j}}$$

giving an approximate expression for our desired probability in terms of single-byte and double-byte probabilities. Notice that if we assume that the adjacent byte pairs are independent, then we have $p_{r+j,z_j,z_{j+1}} = p_{r+j,z_j} \cdot p_{r+j+1,z_{j+1}}$ and the above expression collapses down to the one we derived in the previous subsection.

For candidate x, we again write $x = (x_0, \ldots, x_{n-1})$ and rearranging terms, we obtain:

$$\prod_{z \in \mathcal{B}^n} \Pr(Z = z)^{N_{x,z}} \approx \frac{\prod_{j=0}^{n-2} \prod_{z_1 \in \mathcal{B}} \prod_{z_2 \in \mathcal{B}} p_{r+j,z_1,z_2}^{N_{x_j,x_{j+1},z_1,z_2,j}}}{\prod_{j=1}^{n-2} \prod_{z \in \mathcal{B}} p_{r+j,z}^{N_{x_j,z,r+j}}}.$$

where $N_{y_1,y_1,z_1,z_2,j} = |\{i : z_1 = c_{i,j} \oplus y_1 \wedge z_2 = c_{i,j+1} \oplus y_2\}_{0 \le i < S}|$ counts (now for consecutive pairs of bytes) the number of occurrences of bytes (z_1, z_2) in the pair of column vectors formed by XORing columns $(j, j+1)$ of c with candidate bytes (y_1, y_2) (and where $N_{x_j,z,r+j}$ is as in our previous algorithm).

Again, the counters $N_{y_1,y_2,z_1,z_2,j}$ for $y_1, y_2 \in \mathcal{B}$ can all be computed efficiently by permuting the counters $N_{\texttt{0x00},\texttt{0x00},z_1,z_2,j}$, these being simply counters for the number of occurrences of pairs of byte values (z_1, z_2) in column j and $j+1$ of the ciphertext matrix c. As before, we work with logarithms, so that we evaluate for each candidate

$x = (x_0, \ldots, x_{n-1})$ an expression of the form

$$\gamma_x := \log(p_x) + \sum_{j=0}^{n-2} \sum_{z_1 \in \mathcal{B}} \sum_{z_2 \in \mathcal{B}} N_{x_j, x_{j+1}, z_1, z_2, j} \log(p_{r+j, z_1, z_2}) - \sum_{j=1}^{n-2} \sum_{z \in \mathcal{B}} N_{x_j, z, r+j} \log(p_{r+j, z}).$$

With appropriate pre-computation of the terms $N_{y_1,y_2,z_1,z_2,j} \log(p_{r+j,z_1,z_2})$ and $N_{y,z,r+j} \log(p_{r+j,z})$ for all y_1, y_2 and all y, the computation for each candidate $x \in \mathcal{X}$ can be reduced to roughly $2n$ floating point additions. The pre-computation can be further reduced by computing the terms for only those pairs (y_1, y_2) actually arising in candidates in $\mathcal{X}$ in positions $(j, j+1)$. We use this further optimisation in our implementation.

The above procedure is specified as an attack in Algorithm 4. We refer to it as our double-byte attack because of its reliance on the double-byte keystream probabilities p_{s,z_1,z_2}. It again outputs a collection of approximate log likelihoods $\{\gamma_x : x \in \mathcal{X}\}$ for each candidate $x \in \mathcal{X}$, suitable for further processing. Note that for simplicity of presentation, it involves a quintuply-nested loop to compute the values $N_{y_1,y_2,z_1,z_2,j}$; these values should of course be directly computed from the $(n-1) \cdot 2^{16}$ pre-computed counters $N'_{c_{i,j},c_{i,j+1},j}$ in an in-line fashion using the formula $N_{y_1,y_2,z_1,z_2,j} = N'_{z_1 \oplus y_1, z_2 \oplus y_2,,j}$.

Algorithm 4: Double-byte attack

input : $c_{i,j} : 0 \leq i < S, 0 \leq j < n$ – array formed from S independent encryptions of fixed n-byte candidate X
r – starting position of X in plaintext stream
$\mathcal{X}$ – collection of N candidates
p_x – *a priori* probability of candidates $x \in \mathcal{X}$
$p_{r+j,z}$ $(0 \leq j < n,\ z \in \mathcal{B})$ – single-byte keystream distribution
p_{r+j,z_1,z_2} $(0 \leq j < n-1,\ z_1, z_2 \in \mathcal{B})$ – double-byte keystream distribution

output : $\{\gamma_x : x \in \mathcal{X}\}$ – set of (approximate) log likelihoods for candidates in $\mathcal{X}$

begin

- **for** $j = 0$ **to** $n - 2$ **do**
  - **for** $z_1 =$ `0x00` **to** `0xFF` **do**
    - $N'_{z,j} \leftarrow 0$
    - **for** $z_2 =$ `0x00` **to** `0xFF` **do**
      - $N'_{z_1,z_2,j} \leftarrow 0$
- **for** $j = 0$ **to** $n - 2$ **do**
  - **for** $i = 0$ **to** $S - 1$ **do**
    - $N'_{c_{i,j},j} \leftarrow N'_{c_{i,j},j} + 1$
    - $N'_{c_{i,j},c_{i,j+1},j} \leftarrow N'_{c_{i,j},c_{i,j+1},j} + 1$
- **for** $j = 1$ **to** $n - 2$ **do**
  - **for** $y =$ `0x00` **to** `0xFF` **do**
    - **for** $z =$ `0x00` **to** `0xFF` **do**
      - $N_{y,z,j} \leftarrow N'_{z \oplus y,j}$
    - $L_{y,j} = \sum_{z \in \mathcal{B}} N_{y,z,j} \log(p_{r+j,z})$,
- **for** $j = 0$ **to** $n - 2$ **do**
  - **for** $y_1 =$ `0x00` **to** `0xFF` **do**
    - **for** $y_2 =$ `0x00` **to** `0xFF` **do**
      - **for** $z_1 =$ `0x00` **to** `0xFF` **do**
        - **for** $z_2 =$ `0x00` **to** `0xFF` **do**
          - $N_{y_1,y_2,z_1,z_2,j} \leftarrow N'_{z_1 \oplus y_1, z_2 \oplus y_2,,j}$
      - $L_{y_1,y_2,j} = \sum_{z_1 \in \mathcal{B}} \sum_{z_2 \in \mathcal{B}} N_{y_1,y_2,z_1,z_2,j} \log(p_{r+j,z_1,z_2})$,
- **for** $x = (x_0, \ldots, x_{n-1}) \in \mathcal{X}$ **do**
  - $\gamma_x \leftarrow \log(p_x) + \sum_{j=0}^{n-2} L_{x_j,x_{j+1},j} - \sum_{j=1}^{n-2} L_{x_j,j}$
- **return** $\{\gamma_x : x \in \mathcal{X}\}$

2.4 Simulation Results

2.4.1 Methodology

We performed extensive simulations of both of our attacks, varying the different parameters to evaluate their effects on success rates. We focus on the problem of password recovery, using the RockYou data set as an attack dictionary and the Singles.org data set as the set of target passwords. Except where noted, in each simulation, we performed 256 independent runs of the relevant attack. In each attack in a simulation, we select a password of some fixed length n from the Singles.org password data set according to the known *a priori* probability distribution for that data set, encrypt it S times in different starting positions r using random 128-bit keys for RC4, and then attempt to recover the password from the ciphertexts using the set of all passwords of length n from the entire RockYou data set (14 million passwords) as our candidate set $\mathcal{X}$. We declare success if the target password is found within the top T passwords suggested by the algorithm (according to the approximate likelihood measures γ_x). Our default settings, unless otherwise stated, are $n = 6$ and $T = 5$, and we try all values for r between 1 and $256 - n + 1$, where the single-byte biases dominate the behaviour of the RC4 keystreams. Typical values of S are 2^s where $s \in \{20, 22, 24, 26, 28\}$.

Using different data sets for the attack dictionary and the target set from which encrypted passwords are chosen is more realistic than using a single dictionary for both

purposes, not least because in a real attack, the exact content and *a priori* distribution of the target set would not be known. This approach also avoids the problem of *hapax legomena* highlighted in [40]. However, this has the effect of limiting the success rates of our attacks to less than 100%, since there are highly likely passwords in the target set (such as `jesus`) that do not occur at all, or only have very low *a priori* probabilities in the attack dictionary, and conversely. Figure 2.1 compares the use of the RockYou password distribution to attack Singles.org passwords with the less realistic use of the RockYou password distribution to attack RockYou itself. It can be seen that, for the particular choice of attack parameters ($S = 2^{24}$, $n = 6$, $T = 5$, double-byte attack), the effect on success rate is not particularly large. However, for other attack parameters, as we will see below, we observe a maximum success rate of around 80% for our attacks, whereas we would achieve 100% success rates if we used RockYou against itself. The observed maximum success rate could be increased by augmenting the attack dictionary with synthetically generated, site-specific passwords and by removing RockYou-specific passwords from the attack dictionary. We leave the development and evaluation of these improvements to future work.

Many data sets are available from password breaches. We settled on using RockYou for the attack dictionary because it was one of the biggest data sets in which all passwords and their associated frequencies were available, and because the distribution of passwords, while certainly skewed, was less skewed than for other data sets. We used Singles.org for the target set because the Singles.org breach occurred later than

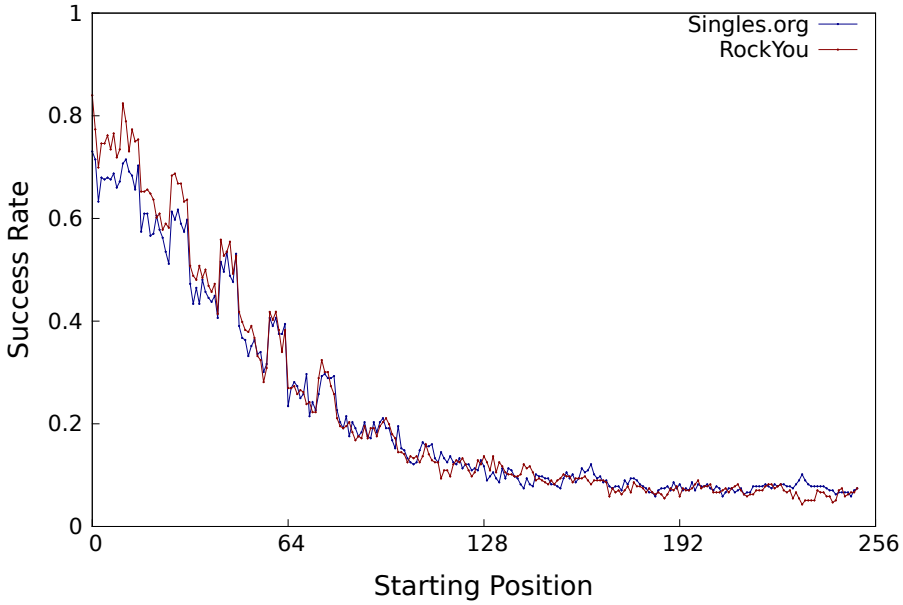


Figure 2.1: Recovery rate for Singles.org passwords using RockYou data set as dictionary, compared to recovery rate for RockYou passwords using RockYou data set as dictionary ($S = 2^{24}$, $n = 6$, $T = 5$, $1 \leq r \leq 251$, double-byte attack).

the RockYou breach, so that the former could reasonably used as an attack dictionary for the latter. Moreover, the Singles.org distribution being quite different from that for RockYou makes password recovery against Singles.org using RockYou as a dictionary more challenging for our attacks. A detailed evaluation of the extent to which the success rates of our attacks depend on the choice of attack dictionary and target set is beyond the scope of this current work.

A limitation of our approach is that we assume the password length n to be already known, whereas in reality this may not be the case. At least four potential solutions to this problem exist. Firstly, in specific applications, n may leak via analysis of packet lengths or other forms of traffic analysis. Secondly we can run our attacks for the

full range of password lengths, possibly adjusting the likelihood measure γ_x for each password candidate x to scale it appropriately by its length (except for the p_x term). A third approach is to augment the shorter passwords with the known plaintext that typically follows them in a specific targeted application protocol and then run our attacks for a fixed, but now longer, n. A fourth approach applies in protocols which use known delimiters to denote the end of a password (such as the = symbol seen at the end of Base64 encodings for certain username/password lengths); here, the idea is to adapt our general attacks to compute the likelihood that such a delimiter appears in each possible position, and generate an estimate for n by selecting the position for which the likelihood is highest.

2.4.2 Results

2.4.2.0.4 Single-Byte Attack

We ran the attack described in Algorithm 3 with our default parameters ($n = 6$, $T = 5$, $1 \leq r \leq 251$) for $S = 2^s$ with $s \in \{20, 22, 24, 26, 28\}$ and evaluated the attack's success rate. We used our default of 256 independent runs per parameter set. The results are shown in Figure 2.2. We observe that:

- The performance of the attack improves markedly as S, the number of ciphertexts, increases, but the success rate is bounded by 75%. We attribute this to the use of one dictionary (RockYou) to recover passwords from another (Singles.org)

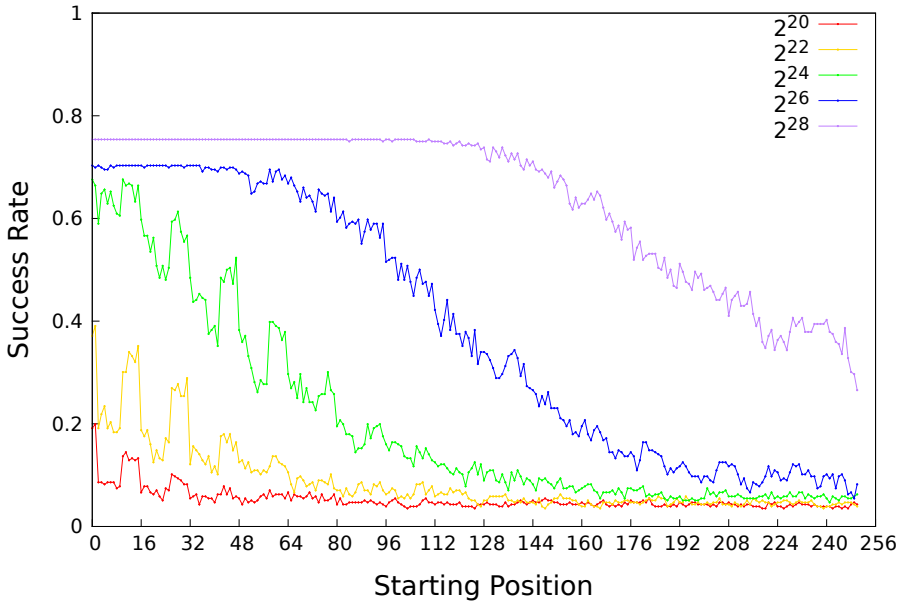


Figure 2.2: Recovery rates for single-byte algorithm for $S = 2^{20}, \ldots, 2^{28}$ ($n = 6$, $T = 5$, $1 \leq r \leq 251$).

– for the same attack parameters, we achieved 100% success rates when using RockYou against RockYou, for example.

- For 2^{24} ciphertexts we see a success rate of greater than 60% for small values of r, the position of the password in the RC4 keystream. We see a drop to below 50% for starting positions greater than 32. We note the effect of the key-length-dependent biases on password recovery; passwords encrypted at starting positions $16\ell - n, 16\ell - n + 1, \ldots, 16\ell - 1, 16\ell$, where $\ell = 1, 2, \ldots, 6$, have a higher probability of being recovered in comparison to neighbouring starting positions.

- For 2^{28} ciphertexts we observe a success rate of more than 75% for $1 \leq r \leq 120$.

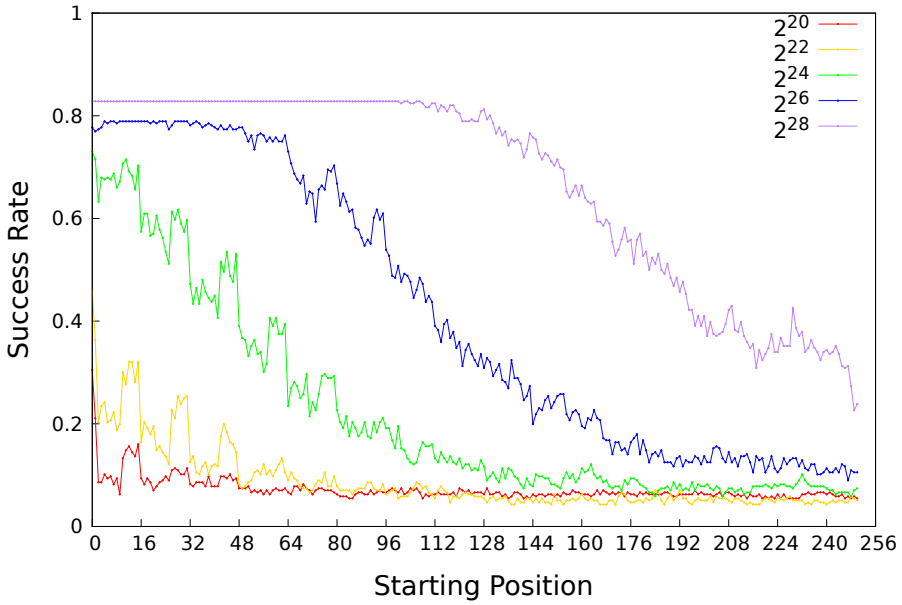


Figure 2.3: Recovery rates for double-byte algorithm for $S = 2^{20}, \ldots, 2^{28}$ ($n = 6$, $T = 5$, $1 \leq r \leq 251$).

2.4.2.0.5 Double-Byte Attack

Analogously, we ran the attack of Algorithm 4 for $S = 2^s$ with $s \in \{20, 22, 24, 26, 28\}$ and our defaults of $n = 6$, $T = 5$. The results for these simulations are shown in Figure 2.3. Note that:

- Again, at 2^{24} ciphertexts the effect of key-length-dependent biases is visible.
- For 2^{26} ciphertexts we observe a success rate that is greater than 78% for $r \leq 48$.

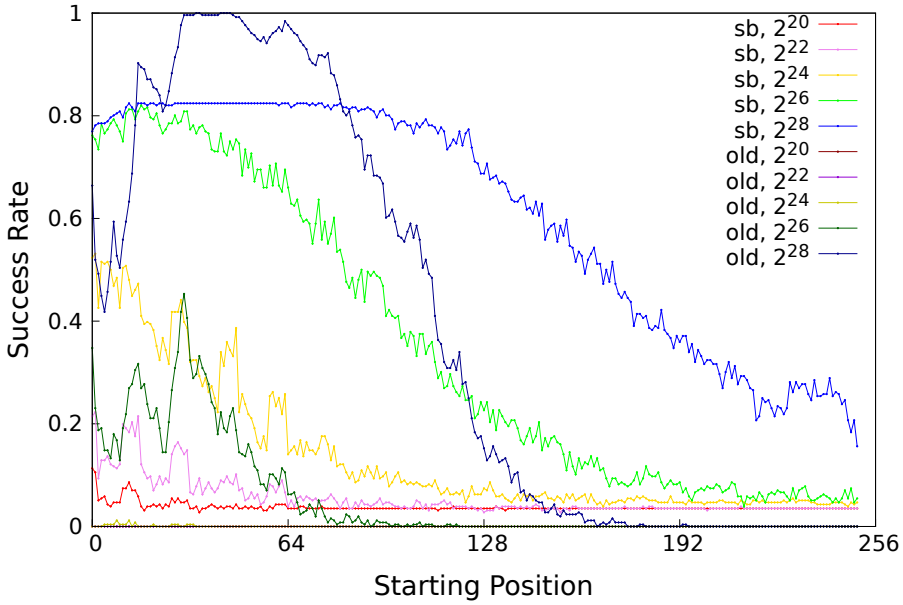


Figure 2.4: Performance of our single-byte algorithm versus a naive single-byte attack based on the methods of AlFardan *et al.* (labelled "old"). ($n = 6$, $T = 1$, $1 \leq r \leq 251$.)

2.4.2.0.6 Comparing the Single-Byte Attack with a Naive Algorithm

Figure 2.4 provides a comparison between our single-byte algorithm with $T = 1$ and a naive password recovery attack based on the methods of [22], in which the password bytes are recovered one at a time by selecting the highest likelihood byte value in each position and declaring success if all bytes of the password are recovered correctly. Significant improvement over the naive attack can be observed, particularly for high values of r. For example with $S = 2^{24}$, the naive algorithm essentially has a success rate of zero for every r, whereas our single-byte algorithm has a success rate that exceeds 20% for $1 \leq r \leq 63$. By way of comparison, an attacker knowing the password

length and using the obvious guessing strategy would succeed with probability 4.2% with a single guess, this being the *a priori* probability of the password `123456` amongst all length 6 passwords in the Singles.org dataset (and `123456` being the highest ranked password in the RockYou dictionary, so the first one that an attacker using this strategy with the RockYou dictionary would try). As another example, with $S = 2^{28}$ ciphertexts, a viable recovery rate is observed all the way up to $r = 251$ for our single-byte algorithm, whereas the naive algorithm fails badly beyond $r = 160$ for even this large value of S. Note however that the naive attack can achieve a success rate of 100% for sufficiently large S, whereas our attack cannot. This is because the naive attack directly computes a password candidate rather than evaluating the likelihood of candidates from a list which may not contain the target password. On the other hand, our attack trivially supports larger values of T, whereas the naive attack is not so easily modified to enable this feature.

2.4.2.0.7 Comparing the Single-Byte and Double-Byte Attacks

Figure 2.5 provides a comparison of our single-byte and double-byte attacks. With all other parameters equal, the success rates are very similar for the initial 256 positions. The reason for this is the absence of many strong double-byte biases that do not arise from the known single-byte biases in the early positions of the RC4 keystream.

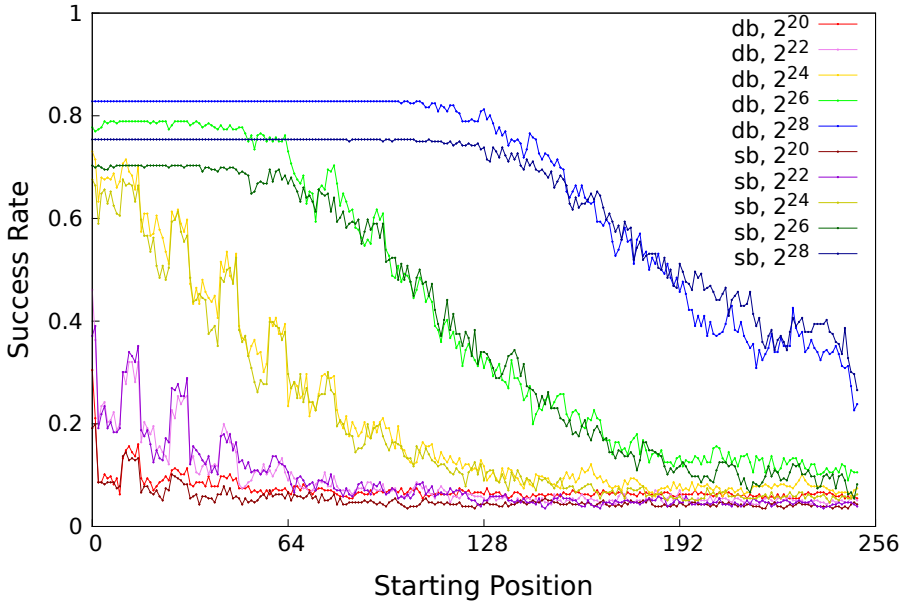


Figure 2.5: Recovery rate of single-byte versus double-byte algorithm for $S = 2^{20}, \ldots, 2^{28}$ ($n = 6$, $T = 5$, $1 \leq r \leq 251$).

2.4.2.0.8 Effect of the *a priori* Distribution

As a means of testing the extent to which our success rates are influenced by knowledge of the *a priori* probabilities of the candidate passwords, we ran simulations in which we tried to recover passwords sampled correctly from the Singles.org dataset but using a uniform *a priori* distribution for the RockYou-based dictionary used in the attack. Figure 2.6 shows the results ($S = 2^{24}$, $n = 6$, $T = 5$, double-byte attack) of these simulations, compared to the results we obtain by exploiting the *a priori* probabilities in the attack. It can be seen that a significant gain is made by using the *a priori* probabilities, with the uniform attack's success rate rapidly dropping to zero at around $r = 128$.

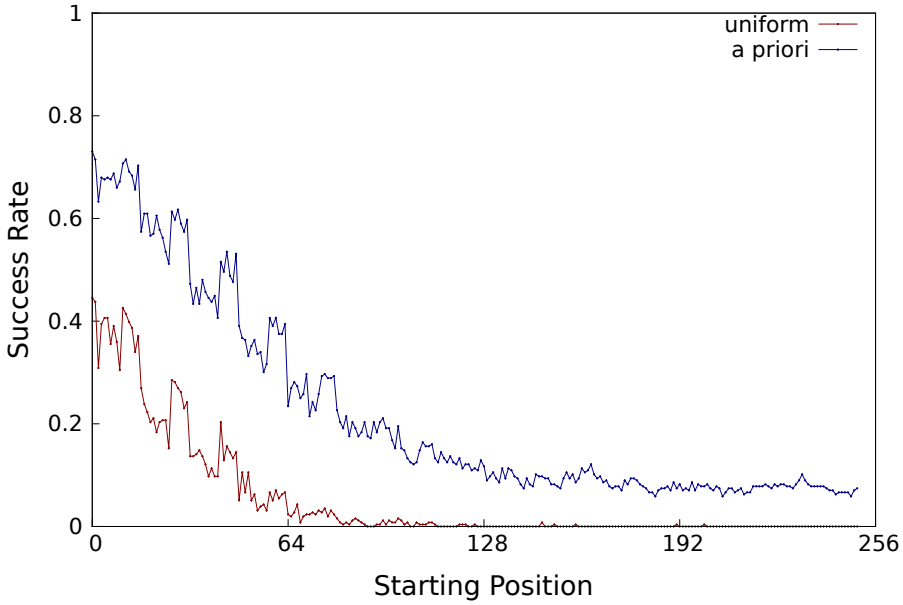
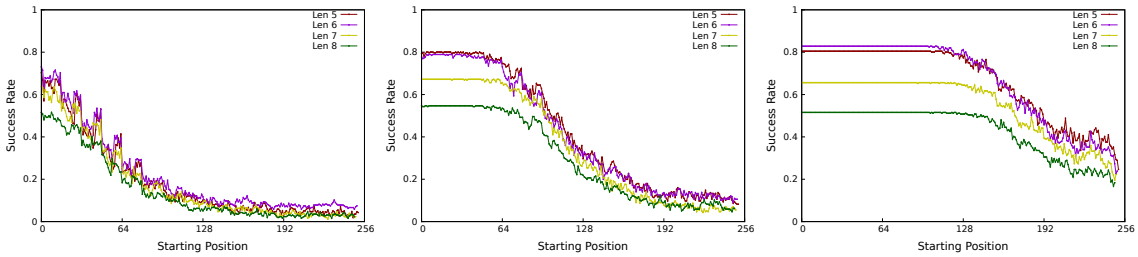


Figure 2.6: Recovery rate for uniformly distributed passwords versus known *a priori* distribution ($S = 2^{24}$, $n = 6$, $T = 5$, $1 \leq r \leq 251$, double-byte algorithm).

2.4.2.0.9 Effect of Password Length

Figure 2.7 shows the effect of increasing n, the password length, on recovery rates, with the sub-figures showing the performance of our double-byte attack for different numbers of ciphertexts ($S = 2^s$ with $s \in \{24, 26, 28\}$). Other parameters are set to their default values. As intuition suggests, password recovery becomes more difficult as the length increases. Also notable is that the ceiling on success rate of our attack decreases with increasing n, dropping from more than 80% for $n = 5$ to around 50% for $n = 8$. This is due to the fact that only 48% of the length 8 passwords in the Singles.org data set actually occur in the RockYou attack dictionary: our attack is doing as well as it can in this case, and we would expect stronger performance with



(a) 2^{24} ciphertexts (b) 2^{26} ciphertexts (c) 2^{28} ciphertexts

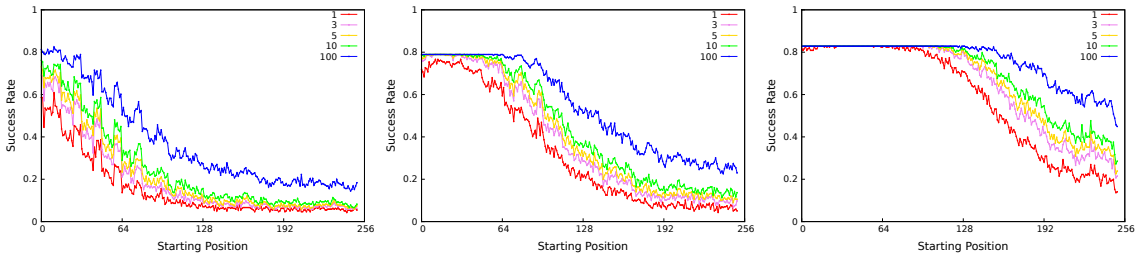
Figure 2.7: Effect of password length on recovery rate ($T = 5$, $1 \leq r \leq 251$, double-byte algorithm).

an attack dictionary that is better matched to the target site.

2.4.2.0.10 Effect of Increasing Try Limit T

Recall that the parameter T defines the number of password trials our attacks make. The number of permitted attempts for specific protocols like BasicAuth and IMAP is server-dependent and not mandated in the relevant specifications. Whilst not specific to our chosen protocols, a 2010 study [26] showed that 84% of websites surveyed allowed at least $T = 100$ attempts; many websites appear to actually allow $T = \infty$. Figure 2.8 shows the effect of varying T in our double-byte algorithm for different numbers of ciphertexts ($S = 2^s$ with $s \in \{24, 26, 28\}$). Other parameters are set to their default values. It is clear that allowing large values of T boosts the success rate of the attacks.

Note however that a careful comparison must be made between our attack with parameter T and the success rate of the obvious password guessing attack given T attempts. Such a guessing attack does not require any ciphertexts but instead uses

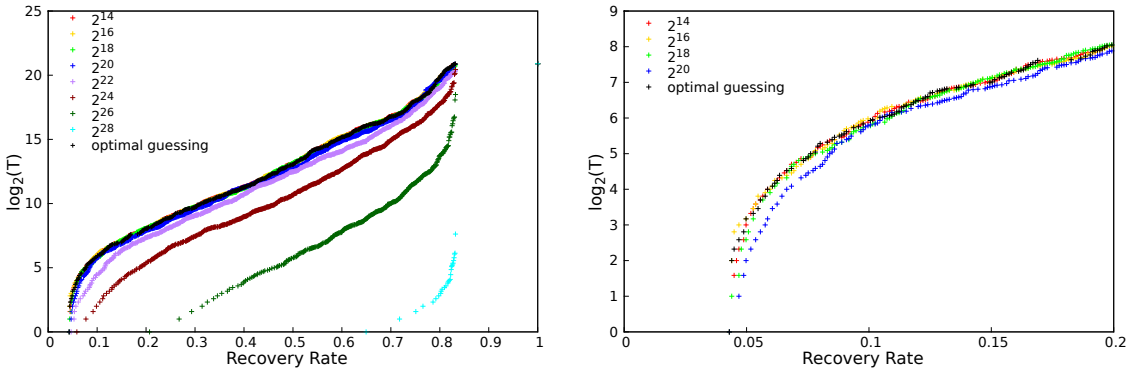


(a) 2^{24} ciphertexts (b) 2^{26} ciphertexts (c) 2^{28} ciphertexts

Figure 2.8: Effect of try limit T on recovery rate ($n = 6$, $1 \leq r \leq 251$, double-byte algorithm).

the *a priori* distribution on passwords in the attack dictionary (RockYou) to make guesses for the target password in descending order of probability, the success rate being determined by the *a priori* probabilities of the guessed passwords in the target set (Singles.org). Clearly, our attacks are only of value if they significantly out-perform this ciphertext-less attack.

Figure 2.9 shows the results of plotting $\log_2(T)$ against success rate α for $S = 2^s$ with $s \in \{14, 16, \ldots, 28\}$. The figure then illustrates the value of T necessary in our attack to achieve a given password recovery rate α for different values of S. This measure is related to the α-work-factor metric explored in [40] (though with the added novelty of representing a work factor when one set of passwords is used to recover passwords from a different set). To generate this figure, we used 1024 independent runs rather than the usual 256, but using a fixed set of 1024 passwords sampled according to the *a priori* distribution for Singles.org. This was in an attempt to improve the stability of the results (with small numbers of ciphertexts S, the success



(a) $\alpha \in [0, 1]$ (b) $\alpha \in [0, 0.2]$

Figure 2.9: Value of T required to achieve a given password recovery rate α for $S = 2^s$ with $s \in \{14, 16, \ldots, 28\}$ ($n = 6$, $r = 133$, double-byte algorithm).

rate becomes heavily dependent on the particular set of passwords selected and their *a priori* probabilities, while we wished to have comparability across different values of S). The success rates shown are for our double-byte attack with $n = 6$ and $r = 133$, this specific choice of r being motivated by it being the location of passwords for our BasicAuth attack proof-of-concept when the Chrome browser is used (similar results are obtained for other values of r). The graph also shows the corresponding work factor T as a function of α for the guessing attack (labeled "optimal guessing" in the figure).

Figure 2.9a shows that our attack far outperforms the guessing attack for larger values of S, with a significant advantage accruing for $S = 2^{24}$ and above. However, as Figure 2.9b shows, the advantage over the guessing attack for smaller values of S, namely 2^{20} and below, is not significant. This can be attributed to our attack

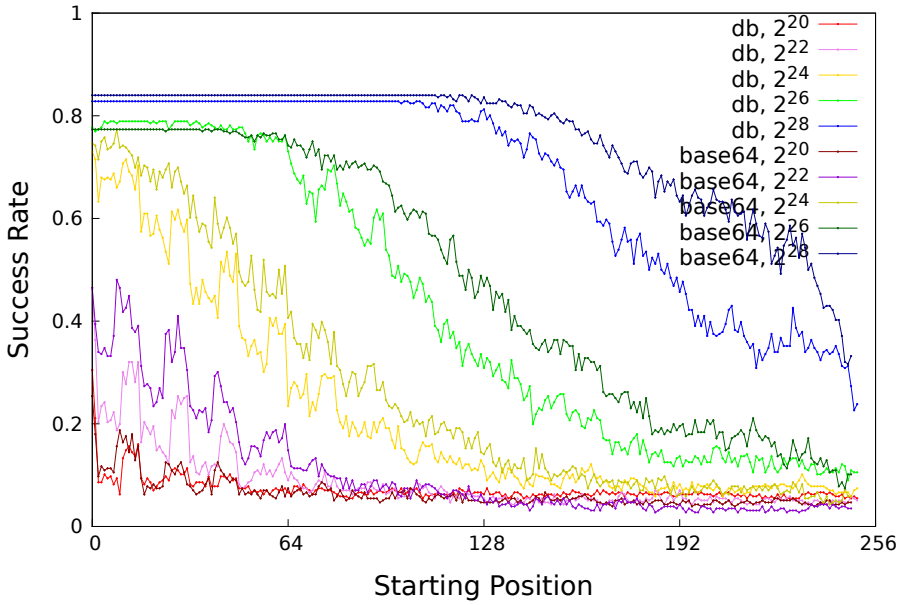


Figure 2.10: Recovery rate of Base64 encoded password versus a "normal" password for 6-character passwords ($T = 5$, $1 \leq r \leq 251$, double-byte algorithm).

simply not being able to compute stable enough statistics for these small numbers of ciphertexts. In turn, this is because the expected random fluctuations in the keystream distributions overwhelm the small biases; in short, the signal does not sufficiently exceed the noise for these low values of S.

2.4.2.0.11 Effect of Base64 Encoding

We investigated the effect of Base64 encoding of passwords on recovery rates, since many application layer protocols use such an encoding. The encoding increases the password length, making recovery harder, but also introduces redundancy, potentially helping the recovery process to succeed. Figure 2.10 shows our simulation results comparing the performance of our double-byte algorithm acting on 6-character

passwords and on Base64 encoded versions of them. It is apparent from the figure that the overall effect of the Base64 encoding is to help our attack to succeed. In practice, the start of the target password may not be well-aligned with the Base64 encoding process (for example, part of the last character of the username and/or a delimiter such as ":" may be jointly encoded with part of the first character of the password). This can be handled by building a special-purpose set of candidates $\mathcal{X}$ for each possibility. Handling this requires some care when mounting a real attack against a specific protocol; a detailed analysis is deferred to future work.

2.4.2.0.12 Shifting Attack

It was observed in [22] and elsewhere that for 128-bit keys, RC4 keystreams exhibit particularly large "key-length-dependent" biases at positions $r = 16\ell$, $\ell = 1, \ldots, 7$, with the bias size decreasing with increasing ℓ. These large biases boost recovery rates, as already observed in our discussion of Figure 2.2.

In certain application protocols and attack environments (such as HTTPS) it is possible for the adversary to incrementally pad the plaintext messages so that the unknown bytes are always aligned with positions having large keystream biases. Our algorithm descriptions and code are both easily modified to handle this situation, and we have conducted simulations with the resulting shift attack.

Figure 2.11 shows the results for the shift version of our double-byte algorithm. In the shift attack, the true number of ciphertexts is equal to $n \times S$, since we now use S

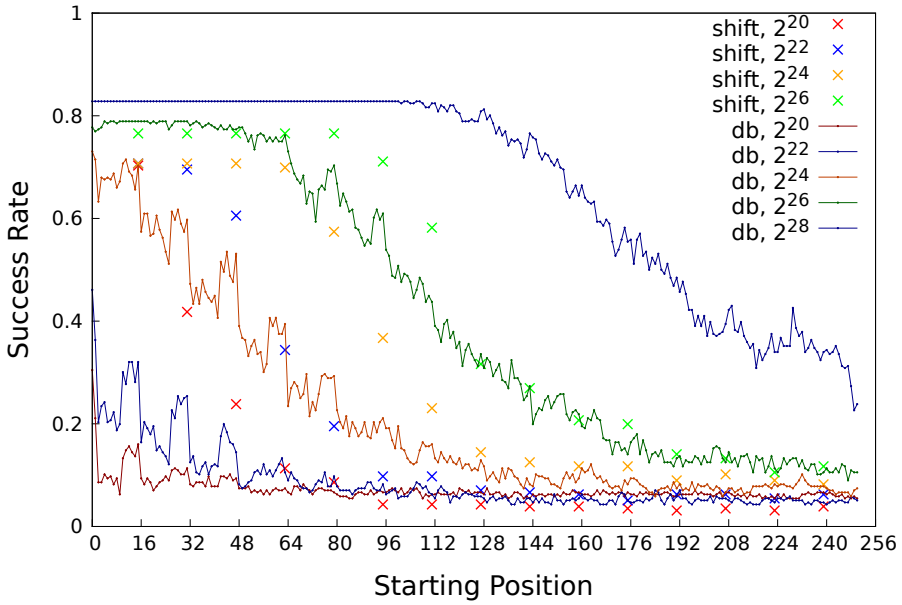


Figure 2.11: Recovery rate of shift attack versus double-byte algorithm ($n = 6$, $T = 5$, $1 \leq r \leq 251$).

ciphertexts at each of n shift positions. So a proper comparison would compare with one of our earlier attacks using an appropriately increased value of S. Making this adjustment, it can be seen that the success rate is significantly improved, particularly for small values of $r = 16\ell$ where the biases are biggest.

2.5 Practical Validation

In this section we describe proof-of-concept implementations of our attacks against two specific application-layer protocols running over TLS, namely BasicAuth and IMAP.

2.5.1 BasicAuth

2.5.1.0.13 Introducing BasicAuth

Defined as part of the HTTP/1.0 specification [41], the Basic Access Authentication scheme (BasicAuth) provides a means for controlling access to webpages and other protected resources. Here we provide a high-level overview of BasicAuth and direct the reader to [41] and [42] for further details.

BasicAuth is a challenge-response authentication mechanism: a server will present a client with a challenge to which the client must supply the correct response in order to gain access to the resource being requested. In the case of BasicAuth, the challenge takes the form of either a `401 Unauthorized` response message from an origin server, or a `407 Proxy Authentication Required` response message from a proxy server. BasicAuth requires that the client response contain legitimate user credentials – a username and password – in order for access to be granted. Certain web browsers may display a login dialog when the challenge is received and many browsers present users with the option of storing their user credentials in the browser, with the credentials thereafter being automatically presented on behalf of the user.

The client response to the challenge is of the form

`Authorization: Basic Base64(userid:password)` where `Base64`$(\cdot)$ denotes the Base64 encoding function (which maps 3 characters at a time onto 4 characters of output). Since the username and password are sent over the network as cleartext, BasicAuth

needs to be used in conjunction with a protocol such as TLS.[12]

2.5.1.0.14 Attacking BasicAuth

To obtain a working attack against BasicAuth, we need to ensure that two conditions are met:

- The Base64-encoded password included in the BasicAuth client response can be located sufficiently early in the plaintext stream.
- There is a method for forcing a browser to repeatedly send the BasicAuth client response.

We have observed that the first condition is met for particular browsers, including Google Chrome. For example, we inspected HTTPS traffic sent from Chrome to an iChair server.[13]. We observed the user's Base64-encoded password being sent with every HTTP(S) request in the same position in the stream, namely position $r = 133$ (this includes 16 bytes consumed by the client's `Finished` message as well as the 20-bytes consumed by the TLS Record Protocol tag). For Mozilla Firefox, the value of r was the less useful 349.

For the second condition, we adopt the methods used in the BEAST, CRIME and Lucky 13 attacks on TLS, and also used in attacking RC4 in [22]: we assume

[12]The Digest Access Authentication Scheme was introduced to address the cleartext transmission of passwords. See [42] for details.

[13]iChair is a popular system for conference reviewing, widely used in the cryptography research community and available from `http://www.baigneres.net/ichair`. It uses BasicAuth as its user authentication mechanism.

that the user visits a site `www.evil.com` which loads JavaScript into the user's browser; the JavaScript makes GET or POST requests to the target website at `https://www.good.com` by using `XMLHttpRequest` objects (this is permitted under Cross Origin Resource Sharing (CORS), a mechanism developed to allow JavaScript to make requests to a domain other than the one from which the script originates). The Base64-encoded BasicAuth password is automatically included in each such request. To force the password to be repeatedly encrypted at an early position in the RC4 keystream, we use a MITM attacker to break the TLS connection (by injecting sequences of TCP `FIN` and `ACK` messages into the connection). This requires some careful timing on the part of the JavaScript and the MITM attacker.

We built a proof-of-concept demonstration of these components to illustrate the principles. We set up a virtual network with three virtual machines each running Ubuntu 14.04, kernel version 3.13.0-32. On the first machine, we installed iChair. We configured the iChair web server to use RC4 as its default TLS cipher. The second machine was running the Chrome 38 browser and acted as the client in our attack. We installed the required JavaScript directly on this machine rather than downloading from another site. The third machine acted as the MITM attacker, required to intercept the TLS-protected traffic and to tear-down the TLS connections. We used the Python tool Scapy[14] to run an ARP poisoning attack on the client and server from the MITM so as to be able to intercept packets; with the connection

[14]Available at `http://www.secdev.org/projects/scapy/`.

hijacked we were able to force a graceful shutdown of the connection between the client and the server after the password-bearing record had been observed and recorded. We observed that forcing a graceful shutdown of each subsequent connection did allow for TLS resumption (rather than leading to the need for a full TLS Handshake run).

With this setup, the JavaScript running in the client browser sent successive HTTPS GET requests to the iChair server every 80ms. Our choice of 80ms was motivated by the fact that for our particular configuration, we observed a total time of around 80ms for TLS resumption, delivery of the password-bearing record and the induced shutdown of the TCP connection. This choice enabled us to capture 2^{16} encrypted password-bearing records in 1.6 hours (the somewhat greater than expected time here being due to anomalies in network behaviour). Running at this speed, the attack was stable over a period of hours.

We note that the latency involved in our setup is much lower than would be found in a real network in which the server may be many hops away from the client: between 500ms and 1000ms is typical for establishing an initial TLS connection to a remote site, with the latency being roughly half that for session resumptions. Notably, the cost of public key operations is not the issue, but rather the network latency involved in the round-trips required for TCP connection establishment and then running the TLS Handshake. However, browsers also open up multiple TLS connections in parallel when fetching multiple resources from a site, as a means of reducing the latency perceived by users; the maximum number of concurrent connections per server is 6 for both the

Chrome and Firefox browsers (though, we only ever saw roughly half this number in practice, even with low inter-request times). This means that, assuming a TLS resumption latency (including the client's TCP `SYN`, delivery of the password-bearing record and the final, induced TCP `ACK`) of 250ms and the JavaScript is running fast enough to induce the browser to maintain 6 connections in parallel, the amount of time needed to mount an attack with $S = 2^{26}$ would be on the order of 776 hours. If the latency was further reduced to 100ms (because of proximity of the server to the client), the attack execution time would be reduced to 312 hours.

Again setting $n = 6$, $T = 100$, $r = 133$ and using the simulation results displayed in Figure 2.10, we would expect a success rate of 64.4% for this setup (with $S = 2^{26}$). For $T = 5$, the corresponding success rate would be 44.5%.

We emphasise that we have not executed a complete attack on these scales, but merely demonstrated the feasibility of the attack in our laboratory setup.

2.5.2 The Internet Message Access Protocol

In Appendix B, we describe how our attacks can be applied to the Internet Message Access Protocol (IMAP), a common client-server protocol for dealing with e-mail. In summary, IMAP's `AUTHENTICATE PLAIN` SASL and `LOGIN` mechanisms lead to user passwords being sent over IMAP, with the password being protected by TLS thanks to the execution of the IMAP `STARTTLS` command; moreover, in our experimental setup, we saw the location of the password in the subsequent TLS session varying between

positions 102 and 108 (depending on the server to which the client connected, but with stable results on a per server basis). The main issue was the slow rate at which our email client polled the server, leading to a correspondingly slow rate of encrypted password transmission. We describe several ways by which this rate could be boosted in Appendix B.

2.6 Conclusion and Open Problem

We have presented plaintext recovery attacks that derive from a formal Bayesian analysis of the problem of estimating plaintext likelihoods given an *a priori* plaintext distribution, suitable keystream distribution information, and a large number of encryptions of a fixed plaintext under independent keys. We applied these ideas to the specific problem of recovering passwords encrypted by the RC4 algorithm with 128-bit keys as used in TLS, though they are of course more generally applicable – to uses of RC4 other than in TLS, and to stream ciphers with non-uniform keystream distributions in general. Using large-scale simulations, we have investigated the performance of these attacks under different settings for the main parameters.

We then studied the applicability of these attacks for two different application layer protocols, BasicAuth and IMAP. In both cases, for certain browsers and clients, the passwords were located at a favourable point in the plaintext stream and we could induce the password to be repeatedly encrypted under fresh, random keys. We built

a proof-of-concept implementation of both attacks. In both cases, it was difficult to arrange for the rate of generation of encryptions to be as high as desired for a speedy attack. For BasicAuth this was mainly due to the latency associated with TLS connection establishment (even with session resumption) rather than any fundamental barrier. For IMAP, the low rate of encryption was more due to the rate at which an IMAP client polls an IMAP server. We discussed ways in which this could be overcome.

Good-to-excellent password recovery success rates can be achieved using 2^{24} – 2^{28} ciphertexts in our attacks. We also demonstrated that our single-byte attack for password recovery significantly outperforms a naive password recovery attack based on the ideas of [22]. We observed an improvement over a guessing strategy even for low numbers (2^{22} or 2^{24}) of ciphertexts. By contrast to these numbers, the preferred double-byte attack of [22] required on the order of 2^{34} encryptions to recover a 16-byte cookie, though without incurring the time overheads arising from TLS session resumption that our approach incurs. In view of our results, we feel justified in claiming that we have significantly narrowed the gap between the feasibility results of [22] and our goal of achieving practical attacks on RC4 in TLS.

Our research has led to the identification of a number of areas for further work:

- Our Bayesian approach can also be applied to the situation where we model the plaintext as a word from a language described as a Markov model with memory. It would be interesting to investigate the extent to which this approach can

be applied to either password recovery or more general analysis of, say, typical HTTP traffic.

- We have focussed on the use of the single-byte biases described in [22] and the double-byte biases of Fluhrer and McGrew (and from our own extensive computations for the first 512 keystream positions). Other biases in RC4 keystreams are known, for example [28]. It is a challenge to integrate these in our Bayesian framework, with the aim being to further improve our attacks.

- We identified new double-byte biases early in the RC4 keystream which deserve a theoretical explanation.

- It would be an interesting challenge to develop algorithms for constructing synthetic, site-specific dictionaries along with *a priori* probability distributions. Existing work in this direction includes Marx's WordHound tool.[15]

- We identified several open questions in the discussion of our simulation results, including the effect of the choice of password data sets on success rates, and the evaluation of different methods for recovering the target password's length.

Acknowledgements

We would like to thank Google, Dan Kaminsky at White Ops and Ingo von Maurich at Ruhr Universität Bochum for their generous donation of computing resources. Dan

[15] `https://bitbucket.org/mattinfosec/wordhound`.

gave us free rein on a 512-core system for the 4800 core-days necessary to perform our double-byte keystream distribution estimates, while resources from Google and Ruhr Universität Bochum were used to conduct our attack simulations. We would also like to thank Alexei Melnikov for acting as our IMAP oracle.

Garman was funded by a generous grant from the Mozilla Foundation and supported by the Office of Naval Research under contract N00014-14-1-0333; Paterson was supported by an EPSRC Leadership Fellowship, EP/H005455/1; van der Merwe was supported by the EPSRC as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London.

Chapter 3

Tools for Developing Secure Cryptographic Systems

This chapter is based on joint work with Joseph A. Akinyele and Susan Hohenberger at Johns Hopkins University. The paper was originally published in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 1370–1381. ACM, 2015 [16], while the complete version that appears in this chapter was published at [43].

Automation is a growing area within the field of cryptography, as it is one method to help us see the greater deployment of cryptographic systems, but in a correct and secure manner. Various factors can influence the use of cryptography, and in this chapter we tackle the problem of the often large gap between the academic setting where cryptography is designed and the practical world where it is used. We do this

by creating AutoGroup+, a tool that securely and efficiently translates pairing based cryptography from the theoretical setting to the practical one.

3.1 Introduction

Automation is increasingly being explored as a means of assisting in the design or implementation of a cryptographic scheme. The benefits of using computer assistance include speed, accuracy, and cost.

Recently, automation for *pairing* (also called *bilinear*) cryptographic constructions (e.g., [11, 12, 44, 45]) has been under exploration. Since the seminal work of Boneh and Franklin [13], interest in pairings is strong: they have become a staple at top cryptography and security conferences, the open-source Charm library has been downloaded thousands of times worldwide and recently pairing-commercializer Voltage Security was acquired by a major US company (HP) [15].

Pairings are algebraic groups with special properties (see Section 3.2.1), which are often employed for their functionality and efficiency. There are different types of pairings: Type-I called "symmetric" is typically how schemes are presented *and proven secure* in the literature, because it is simpler and the complexity assumptions can be weaker; however, Type-III called "asymmetric" is typically the most efficient choice for an implementation in terms of bandwidth and computation time.

Unfortunately, translating a Type-I scheme into the Type-III scheme is complicated.

First, there may be thousands of different Type-III translations of a Type-I scheme and the "best" translation may depend on the application. For instance, one translation might optimize ciphertext size while another offers the fastest decryption time. Second, each new translation requires a new proof under Type-III assumptions. Exploring and analyzing all possible translations is clearly a great burden on a human cryptographer. Indeed a small subset of manual translations of a scheme or particular set of schemes is regarded as a publishable result in its own right, e.g., [46, 47, 48].

Given this translation hurdle, common practice today is to analyze a Type-I scheme, but then use ad-hoc means to derive a Type-III translation that is unproven and possibly non-optimal. The goal of this work is to address this problem by covering new ground in cryptographic automation.

Our Contribution: The AutoGroup+ Tool. Our primary contribution is the design, development, and performance evaluation of a new publicly-available[1] tool, AutoGroup+, that automatically translates pairing schemes from Type-I to Type-III. The output of AutoGroup+ is: (1) "secure" provided the input is "secure" (see Section 3.3.2) and (2) optimal based on the user's efficiency constraints (see Section 3.3.1.5).[2] The input is a computer-readable format of the Type-I construction, metadata about its security analysis, and user-specified efficiency constraints. The output is a translated Type-III construction (in text, C++, Python, or LaTeX) with metadata about its

[1] AutoGroup+ can be downloaded at `https://github.com/jhuisi/auto-tools`.

[2] These claims regard the cryptographic transformation and exclude any software or run-time errors.

security analysis. (See Figure 3.1.)

The audience for this tool is: (1) anyone wanting to implement a pairing construction, and (2) pairing construction designers. We highlight some features.

New Scheme Description Language (SDL) Database. The input to AutoGroup+ requires a computer-readable format of the Type-I construction, the Type-I complexity assumption(s), and the Type-I security proof. It was a challenge to create a means of translating human-written security proofs into SDL. We focused on a common type of proof exhibiting a certain type of black-box reduction.[3] We created a new SDL structure for representing assumptions and reductions of this type that may be of independent interest. Additionally, we did the tedious work of carefully transcribing five assumptions, eight reductions and improving the SDLs for nine popular constructions (from [12]). (See Appendix D for an example of a simple case.) One transcribed, however, these SDL files can be reused. We believe the future of cryptographic automation research will involve processing the assumptions and proofs; thus our database is made public as a testbed for future automation research.

Speed of Tool. AutoGroup+ took less than 21 seconds to process any of the test set, which included seven simple schemes (16 or less solutions), three medium schemes (256 to 512 solutions), and three complex schemes (1024 to 2048 solutions). (The preference for simple schemes was to compare with prior work.) This measures from SDL input to a C++ (or alternative) output. Speed is very important here for usage,

[3]The theoretical translation security results of [44] on which we will base our security are also limited to this class of proof.

because we anticipate that designers may iteratively use this tool like a compiler and implementors may want to try out many different efficiency optimizations.

In contrast, in CRYPTO 2014, Abe, Groth, Ohkubo and Tango [44] laid out an elegant theoretical framework for doing pairing translations in four steps. It left open the issue of whether their framework was practical to implement for a few reasons: (1) they automated only one of four steps (code not released), (2) their algorithm for this step was exponential time, and (3) they tested it on only simple and medium schemes, but their medium scheme took over 1.75 hours for *one* step. Our fully automated translation of that scheme took 6.5 seconds, which is much more in line with the "compiler"-like usage we anticipate.

We attribute our drastic efficiency improvement in part to our use of the Z3 SMT Solver. As described in Section 3.3, we encode the translation of the scheme, its assumption(s) and its reduction as a constraint-satisfaction problem and then use Z3 to quickly find the satisfying set of solutions.

New Results. We evaluated AutoGroup+ on 9 distinct constructions (plus 4 additional variations of one scheme), with various optimization priorities, for 48 bandwidth-optimizing translations. In Figure 3.8, we report the sizes compared to the symmetric case, which are significantly smaller. In Figure 3.9, we report on over 140 timing experiments resulting from the translations. Due both to the asymmetric setting and AutoGroup+'s optimizations, in most cases, the running times were reduced to less than 10% of the symmetric case. In Figure 3.10, we report on the effect that different

levels of complexity have on translation time for a single scheme.

In Section 3.5, we compare the performance of AutoGroup+ to prior automation works, published manual translations, and translations existing as source code in the Advanced Crypto Software Collection [49] and Charm library [50]. We discovered a few things. In fourteen points of comparison with AutoGroup, AutoGroup+ matches those solutions and provides a security validation and new assumptions, adding only a few additional seconds of running time. In three points of comparison with Abe et al. [44] and subsequent personal communications [51], our translated results match.

In the five points of overlap with ACSC and Charm, we are able to confirm the security and ciphertext-size optimality of one broadcast encryption and one hierarchical identity-based encryption implementation. We are also able to confirm the security of two signature implementations, although only one is signature-size optimal. These confirmations are new results. Our tool was able to confirm the ciphertext-size optimality, but not the security of the Charm implementation of Dual System Encryption [52] (meaning it may not be secure). That implementation made changes to the keys outside the scope of the translations here or in [12, 44]. However, our tool did find a secure translation with the same ciphertext-size.

Overall, our tests show that the tool can produce high-quality solutions in just seconds, demonstrating that pairing translations can be practically and securely performed by computers.

3.1.1 Prior Work

The desirability of translating Type-I to Type-III pairings is well documented. First, this is an exercise that cryptographers are still actively doing by hand. In PKC 2012, Ramanna, Chatterjee and Sarkar [46] nicely translated the dual system encryption scheme of Waters [52] from the Type-I pairing setting to a number of different Type-III possibilities. Recently, Chen, Lim, Ling, Wang and Wee [47, 48] presented an elegant semi-general framework for (re-)constructing various IBE, Inner-Product Encryption and Key-Policy Functional Encryption schemes in the Type-III setting, assuming the SXDH assumption holds.[4] These works go into deeper creative detail (changing the scheme or adding assumptions) than our automator, and thus mainly get better results, but then, these works appear to have taken significant human resources. In contrast, our work offers a computerized translation as a starting point.

The Advanced Crypto Software Collection (ACSC) [49], including the Charm library [50], contains many Type-III implementations of schemes that were published and analyzed in the Type-I format. To the best of our knowledge, there is no formal analysis of these converted schemes and thus also no guarantees that the translations are secure or optimal efficiency-wise for a user's specific application. (We remark that ACSC/Charm makes no claims that they *are* secure or optimal.) The public Github records for Charm show that it has been downloaded thousands of times;

[4] Informally, the SXDH assumption asserts that in a Type-III pairing group, there exist no efficient isomorphisms from $\mathbb{G}_1$ to $\mathbb{G}_2$ or from $\mathbb{G}_2$ to $\mathbb{G}_1$.

thus, it would be prudent to verify these implementations. (See our results on this in Section 3.5.)

In ACM CCS 2013, Akinyele, Green and Hohenberger [12] presented a publicly-available tool called AutoGroup, which offered an automated translation from Type-I to Type-III pairing schemes. This work employed sophisticated tools, such as the Z3 Satisfiability Modulo Theories (SMT) solver produced by Microsoft Research (see Section 3.2), to quickly find a set of *possible* assignments of elements into $\mathbb{G}_1$ or $\mathbb{G}_2$. There was not, however, any guarantee that the resulting translation remained secure. Indeed, Akinyele et al. [12] explicitly framed their results as follows: translation has two parts: (1) the search for an efficient translation, and (2) a security analysis of it. They automated the first part and left the security analysis to a human cryptographer. Since they made their source code public, we used it as a starting point and thus named our work after theirs.

While using AutoGroup is certainly faster than a completely manual approach, the lack of a security guarantee is a real drawback. At that time, there was simply no established theory on how to generalize these translations.

Fortunately, in CRYPTO 2014, Abe, Groth, Ohkubo and Tango [44] pushed the theory forward in this area. They elegantly formalized the notion that if certain dependencies from the Type-I complexity assumption(s) and the reduction in the security analysis were added to the dependencies imposed by the scheme itself, then there was a generic way to reason about the security of the translated scheme. Their

main theorem, which we will later use, can informally be stated as:

Theorem 3.1.1 (Informal [44]). Following the conversion method of [44], if the Type-I scheme is correct and secure in the generic Type-I group model, then its converted Type-III scheme is correct and secure in the generic Type-III group model.

There are four steps in their translation: (1) build a dependency graph between the group elements for each algorithm in the construction, the complexity assumption(s) and the security reduction (In the graph, elements are nodes and a directed edge goes from g to h if h is derived from g, such as $h = g^x$.), (2) merge all graphs into a single graph, (3) split this graph into two graphs (where elements of the first graph will be assigned to $\mathbb{G}_1$ and elements of the second assigned to $\mathbb{G}_2$), and (4) derive the converted scheme.

For the four schemes tested in [44], steps (1), (2), and (4) were done by hand. The algorithm for step (3) was *exponential* in two variables[5] and the Java program to handle step (3) reported taking 1.75 hours on a medium scheme. Thus, this is a great theory advance, but it left open the question of whether the entire translation could be efficiently automated as a "real-time" tool.

AutoGroup+ in a Nutshell. In short, prior work admitted a public tool that is fast, but possibly insecure [12], and a cryptographic framework that is slow, but secure [44]. Our goal was to realize the best of both worlds. Even though the implementations differed, we discovered that both works began by tracing generator to pairing dependencies,

[5]Their splitting algorithm runs exponentially in both the number of pairings and the bottom nodes (without outgoing edges) of the dependency graph. Thus, scalability is a real concern.

where [12] did this bottom up and [44] used a top down approach. Since both of these representations can be helpful for different optimizations, AutoGroup+ does both. It also traces these dependencies for the complexity assumptions and reductions. The pairings and hash variables in the combined dependency graph are translated into a formula and constraints, and then fed into a SMT solver. The output set is then efficiently searched for an optimal solution using the SMT solver again, then verified as a valid graph split (as formalized in [44]). Finally, if the split is valid, then a converted scheme and complexity assumption(s) are output.

3.2 Background

3.2.1 Pairings

Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ be groups of prime order p. A map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is an admissible *pairing* (also called a *bilinear map*) if it satisfies the following three properties:

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

computes $e(g, h)$.

A pairing generator is an algorithm that on input a security parameter 1^λ, outputs the parameters for a pairing group $(p, g, h, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ such that p is a prime in $\Theta(2^\lambda)$, $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ are groups of order p where g generates $\mathbb{G}_1$, h generates $\mathbb{G}_2$ and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is an admissible pairing.

The above pairing is called an *asymmetric* or Type-III pairing. This type of pairing is generally preferred in implementations for its efficiency. We also consider *symmetric* or Type-I pairings, where there is an efficient isomorphism $\psi : \mathbb{G}_1 \to \mathbb{G}_2$ (and vice versa) such that a symmetric map is defined as $e : \mathbb{G}_1 \times \psi(\mathbb{G}_1) \to \mathbb{G}_T$. We generally treat $\mathbb{G} = \mathbb{G}_1 = \mathbb{G}_2$ for simplicity and write $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$. These types of pairings are typically preferred for presenting constructions in the academic literature for two reasons. First, they are simpler from a presentation perspective, requiring fewer subscripts and other notations. More importantly, they are sometimes preferred because the underlying symmetric assumption on which the proof is based may be viewed as simpler or weaker than the corresponding asymmetric assumption.

We include current efficiency numbers for Type-I and Type-III groups in Appendix C, demonstrating the significant advantages of the latter.

3.2.2 The Z3 Satisfiability Modulo Theories (SMT) Solver

Our implementation also relies on the power of the state-of-the-art Z3 SMT solver [53] developed at Microsoft Research. SMT is a generalization of boolean satisfiability (or SAT) solving where the goal is to decide whether solutions exist to a given logical formula. The publicly available Z3 is one such tool that is highly efficient in solving constraint satisfaction problems and used in many different applications.

3.2.3 A Scheme Description Language (SDL) and Toolchain

This work builds on the efforts of prior automation works [11, 12] which include several tools such as a scheme description language (or SDL), an accompanying parser for SDL, a code generator that translates SDL schemes into executable code in either C++ or Python, and a LaTeX generator for SDL descriptions. We obtained all these prior tools from the publicly-available AutoTools GitHub repository.[6] Our code and SDL database will be made public in this repository as well. The SDL for the constructions are the same in AutoGroup and AutoGroup+; the difference is that the latter also includes SDL for assumptions and security reductions. Since we used the code of AutoGroup as a starting point, we derived our tool name from it.

[6]Project link: `https://github.com/jhuisi/auto-tools`

3.3 The AutoGroup+ System

As described in Section 3.1, AutoGroup+ is a new tool built to realize the best of both worlds from a prior tool called AutoGroup [12] (fast, but no security guarantees) and new theoretical insights [44] (secure, but exponential time and no public tool.)

3.3.1 How It Works

We begin with an illustration of the AutoGroup+ system in Figure 3.1. This system takes in the description of a symmetric (Type-I) pairing-based scheme S, together with metadata about its security and user-desired efficiency constraints, and outputs an asymmetric (Type-III) pairing-based translation S', together with metadata about its security. Informally, if S was secure, then S' will be both secure and optimal for the constraints set by the user over the space of "basic" translations.

3.3.1.1 Step 1: Generating Computer-Readable Inputs

AutoGroup+ operates on four inputs: an abstract description of the (1) scheme itself, (2) the complexity assumption(s) on which the scheme is based, (3) the black-box reduction in the scheme's proof of security, and (4) a set of efficiency optimization constraints specified by the user (e.g., optimize for smallest key or ciphertext size.). The abstract descriptions are all specified in a Scheme Description Language (SDL) [11, 12].

The need for SDL representations of the complexity assumptions and security

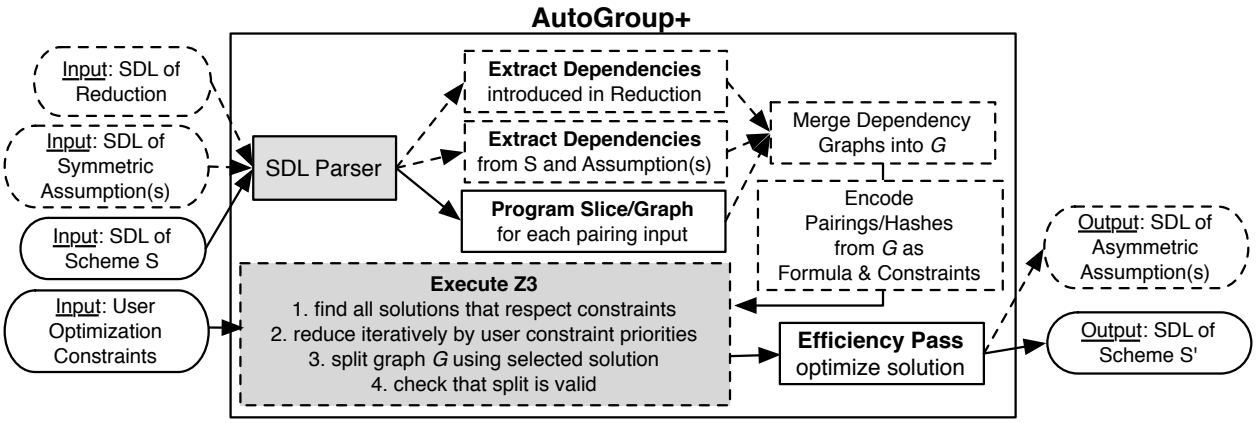


Figure 3.1: A high-level presentation of the AutoGroup+ tool. Components that are new or improved, over AutoGroup, are included with dashed lines. Both AutoGroup+ and AutoGroup use external tools Z3, SDL Parser and Code generator (omitted from the figure).

reductions are new challenges for this work. To run our Section 3.5 tests, we had to translate the text in the published papers to the SDL format by hand. This was a time-consuming and tedious task. However, we maximize the benefit of doing this, by making these SDL files publicly available. This enables anyone to check their correctness and provides a ready-made base of test files for any future automation exercises that require this deeper scheme analysis.

One novel and curious observation we made during these experiments was that *how* group elements were derived in the symmetric group impacted the dependency graphs and therefore the asymmetric results. To say this another way, two schemes computing the exact same elements, but in different ways, could have different dependency graphs and therefore different asymmetric translations. As a toy example, suppose a scheme has $\text{PK} = (g, A = g^a, B = g^b)$ and $\text{SK} = (\text{PK}, a, b)$. Now suppose that as part of a

signing algorithm, the holder of SK must compute the value $C = g^{ab}$. Suppose in Scheme 1, the signer computes this as $x = ab \mod p$ and $C = g^x$. Suppose in Scheme 2, the signer computes this as $C = A^b$. Then in the dependency graph for Scheme 1, there is a root node g, with nodes A and C hanging off it. Whereas for the graph of Scheme 2, there is a root node g with A off it, and C off of A. The importance of these differences comes alive when we attempt to split the graph (see Step 3.3.1.4). Suppose there is the pairing $e(A, C)$. Then in Scheme 1, the generator g must be split, but A can be assigned to $\mathbb{G}_1$ and C to $\mathbb{G}_2$, resulting in a 4 element public key. However, in Scheme 2, the generator g *and* the element A must be split, with $A_1 \in \mathbb{G}_1$ and $A_2 \in \mathbb{G}_2$, so that one can compute $C = (A_2)^b \in \mathbb{G}_2$. This results in a 5 element public key. The general rule is that the fewer unnecessary dependencies the better. Interestingly, Abe et al. [44] sometimes *added* dependencies that did not exist in the original schemes. For instance, for the Waters 2005 IBE [54], Waters clearly states to choose g_2, u', u_i as fresh random generators, but Abe et al. explicitly "assume" that they are generated from a separate generator g. For this particular scheme, this does not impact the asymmetric translations, but in theory it could.

Our experiments did not add any dependencies. We note that in this step, a human is not being tasked with any job but simple transcription of the input into a language the computer can understand.

3.3.1.1.1 System Limitations and Allowable Inputs

This system shares some of the same limitations as prior works [12, 44]. First, this is a junk-in-makes-junk-out system. AutoGroup+ assumes that the security reduction is correct, the complexity assumptions are true, and that the SDL was typed in correctly. If any of these turn out to be false, the output cannot be depended on. Fortunately, we can mitigate these risks as follows. The correctness of the security reductions might be verified automatically using a number of tools, such as EasyCrypt [10], but this likely requires further research. The pairing-based assumptions may be sanity-checked in the generic group model using the recently developed tool by Barthe et al. [45] from CRYPTO 2014. Finally, the SDL transcriptions can be verified in the usual crowd-based manner which we encourage by making them public.

Second, the system does not accept all possible schemes that might appear in the literature. AutoGroup+ supports only prime-order symmetric pairing schemes with a "standard" reduction analysis[7]. It can support most non-interactive assumptions. It can also support dynamic (also called q-based) assumptions, where the size of the assumption may grow depending on the usage of the scheme. It can also support interactive (also called oracle-based) assumptions such as the LRSW assumption behind the popular Camenisch-Lysyanskaya [55] pairing-based signatures.

Third, how the scheme hashes into pairing groups also may disqualify it from

[7]We refer the reader to Abe et al. [44] for a formal definition of the allowed reductions. Roughly, we mean an analysis where there is an efficient algorithm called a *reduction* that is successful in solving the hard problem (underlying the complexity assumption) given black-box access to an adversary that successfully attacks the scheme.

being translated. We now give an example of how to alter the Setup algorithm of the Waters 2005 IBE scheme [54], so that AutoGroup+ cannot translate it. (Indeed, it is not clear to us if a translation even exists.) In the original Setup algorithm, the master authority chooses a generator $g \in \mathbb{G}$ at random. Then public parameter g_1 is derived from g, while parameters $g_2, u_0, \ldots, u_n \in \mathbb{G}$ are chosen independently at random. Instead, suppose we treat the hash function $H : \{0,1\}^* \to \mathbb{G}$ as a random oracle. Let generator $g \in \mathbb{G}$ be computed as $g = H(ID)$, where ID is some string describing the master authority. Then g_1 is derived from g as before, but we set $g_2 = g^r, u_0 = g^{r_0}, \ldots, u_n = g^{r_n}$ for random $r, r_0, \ldots, r_n \in \mathbb{Z}_p$ (where p is the order of $\mathbb{G}$). It is easy to see that the public parameters have the same distribution as before (assuming the random oracle model); all we have changed is *how* the master authority samples these parameters. Thus, this variant of the Waters IBE remains secure in the symmetric setting, and yet it is not clear how to translate it to the asymmetric setting. We return to this example in Section 3.5.

These limitations also appear in the theoretical work of Abe et al. [44], and fortunately, these issues seem relatively rare and did not come up for any of the schemes we tested (except our hand-made counterexample). As in [12, 44], we note that if AutoGroup+ cannot produce a translation, it does not imply that a translation does not exist. A characterization of untranslatable schemes is an open theoretical problem.

3.3.1.2 Step 2: Extracting Algebraic Dependencies

Once AutoGroup+ has parsed all its input files, it begins processing them to graph the algebraic dependencies between source group elements in a scheme, assumption and reduction. All source group elements are nodes in the graph and a directed edge exists if there is a direct dependency between two elements. E.g., if $h = g^x$, then h is derived from g and we place an edge from g and h.

AutoGroup+ extracts the dependency graphs *automatically from the SDL* for each input file and builds a distinct graph from the SDL representations and metadata. AutoGroup+ defines two new procedures that programmatically extract the dependency graph for the assumption(s) as well as the reduction(s) (see Section 3.4 for an example). Then, AutoGroup+ reuses logic from AutoGroup to programmatically build the graph of the scheme by tracking the generators in the setup algorithm and by tracing backward from each pairing in the scheme. It merges the program slice (or trace) extracted for each pairing input into one dependency graph for the scheme. The resulting graphs are the same as those produced by Abe et al. [44] (except where we reduced dependencies by computing elements more directly as discussed in the last step.)

The work of Abe et al. [44] required a human to build (and later merge) these dependency graphs by hand and the graphs were constructed starting from the common generators downward. The AutoGroup work of Akinyele et al. [12] automatically derived these graphs *for the scheme only* from the SDL description of the scheme. They did not consider the assumptions or reduction dependencies. Indeed, AutoGroup only

graphed the dependencies as a traceback from the pairings, whereas AutoGroup+ also adds a top-down analysis from the assumption down to the pairings for the security logic.

3.3.1.3 Step 3: Merge Dependency Graphs

After extracting the dependencies, AutoGroup+ has a set of distinct graphs: one graph that represents dependencies from the setup, key generation, encryption/signature and decryption/verification algorithms, as well as a graph for each complexity assumption and one or more graphs for the reduction. These graphs are then systematically merged together using the metadata provided with the SDL inputs. The metadata includes a reduction map which relates the names of source group elements in the reduction to those in the assumption. We require this map to understand which nodes represent the same group element (across the scheme, assumption and reduction) to simplify merging into a single node. See the example in Section 3.4. AutoGroup+ programmatically checks the type information in the reduction map across all SDL inputs to ensure correctness during the merge.

3.3.1.4 Step 4: Assign Variables using the SMT Solver

This is the most complex step in the automation. In the symmetric setting, all group elements in the scheme were in $\mathbb{G}$. To move to the asymmetric setting, we must assign elements to either $\mathbb{G}_1$ or $\mathbb{G}_2$ in such a way that the dependencies between

elements are not violated (*e.g.*, if $h = g^x$, then both g, h must be in the same group) and such that for all variables a, b, if we have a pairing between them $e(a, b)$, then a and b must be in distinct source groups (*e.g.*, $a \in \mathbb{G}_1$ and $b \in \mathbb{G}_2$ or vice versa). Such an assignment may not be feasible (see such an example in Section 3.3.1.1) or it may require that one or more variables in the symmetric scheme be duplicated in the asymmetric scheme with one assigned to $\mathbb{G}_1$ and another to $\mathbb{G}_2$. E.g., in the symmetric setting if $g \in \mathbb{G}$, $a = g^x$ and $b = g^y$ and these elements are paired as $e(a, b)$, then in the asymmetric setting, g will be split into $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$, where $a = g_1^x$ and $b = g_2^y$, so that one can compute $e(a, b)$.

To *efficiently* make these variable assignments, AutoGroup+ follows the approach of AutoGroup in that it uses a powerful Z3 Satisfiability Modulo Theories (SMT) solver produced by Microsoft Research (see Section 3.2) to compute the set of all possible splits (*i.e.*, all possible variable assignment combinations) and then later identifies the best one. Z3 takes as input a logical formula and determines whether valid variable assignments exist that evaluate that formula to *true*. Similar to AutoGroup, AutoGroup+ expresses the pairing equations as a logical formula of conjunctions and inequality operations over binary variables. For example, $e(a, b) \cdot e(c, d)$ is translated to the logical formula $P1[0] \neq P1[1] \wedge P2[0] \neq P2[1]$ where $P1[0]$ is a reference to a, $P1[1]$ to b, and so on. AutoGroup+ simply follows the pairing identifier convention established by Abe et al. [44].

One major difference between AutoGroup+ and AutoGroup is that the former's

dependency graphs include dependencies based on the assumptions and reductions. The formula is derived from the pairings that occur in the graph (from the construction, reduction and assumption(s)) with a conjunction joining each pairing piece, plus extra constraints added for variables that cannot be duplicated (regarding hashing). This formula is then fed into the solver. The solver returns a set of 0 or 1 assignments for each variable. We then apply each solution to the merged dependency graph to generate the split (variables assigned to 0 on one side and the rest on the other).

3.3.1.5 Step 5: Search for Optimal Solution

There are often many (possibly thousands) of ways to translate a symmetric scheme into an asymmetric scheme; thus, we can end up with many feasible graph splits. Indeed, the output of the SMT solver in the last step is a *set* of assignments of the variables. In this step, we again use the SMT solver to deduce which assignment from this set is "best". AutoGroup+ allows selection of assignments based on a number of user-specified optimization constraints. For public-key encryption, the user can choose to minimize the public-key, assumption, secret key and/or ciphertext size. Similarly for signature schemes, the user can mimize the public-key parameters, assumption, and/or the signature size.

To select an optimal assignment, AutoGroup+ encodes these user requirements as parameters of some *objective function*. We then call the solver a second time with this objective function set to rank/narrow the given solutions to one. Depending on

the optimization goal, the objective function can be specified in one of two ways. If reducing public-key size or the assumption, then we are concerned with minimizing the duplication of source group elements. As such, we first specify an EvalGraph function that the solver uses to compute the splits for each element in the public key or assumption: $\mathsf{EvalGraph}(A_j, B, G) = S$, where $A_j = a_1, \ldots, a_n$ represents pairing input variable assignments for the j-th solution (each a_i variable is either $0 = \mathbb{G}_1$ or $1 = \mathbb{G}_2$), $B = b_1, \ldots, b_m$ represents the source group elements to minimize either in the assumption or public-key, and G represents the merged dependency graph.

Our search algorithm first applies the EvalGraph function to determine how the b_i values are assigned for each solution. Once the b_i values are assigned, we then compute $S = s_1, \ldots, s_m$ where each s_i corresponds to one of three values for each b_i assignment. That is, let a w_1 value denote a $\mathbb{G}_1$ only assignment, w_2 is $\mathbb{G}_2$ only, and $w_3 = w_1 + w_2$ is both a $\mathbb{G}_1$ and $\mathbb{G}_2$ assignment (or simply a split). We then set w_1 and w_2 to the group size of $\mathbb{G}_1$ and $\mathbb{G}_2$ for Type-III pairing curves (*e.g.*, BN256). Each solution is ranked in terms of splits and the total size of group elements in B. Our search returns the j-th solution that results in the fewest splits in B with the smallest overall size S_j. This overall size breaks ties between multiple solutions with the same number of splits.

$$\min_{j \in |A|} (\mathsf{CountSplits}(S_j), \sum_{i=1}^{m} S_{j,i}) \tag{3.1}$$

For the other optimization options (*i.e.*, secret-key, ciphertext, etc), we can reuse the objective function specified by AutoGroup as is:

$$\min_{j \in |A|} F(A_j, C, w_1, w_2) = \sum_{i=1}^{n} ((1 - a_i) \cdot w_1 + a_i \cdot w_2) \cdot c_i \tag{3.2}$$

where the A_j represents the j-th solution as before, $C = \{c_1, \ldots, c_n\}$ represent some *cost* associated with each a_i variable reference, and w_1 and w_2 correspond to *weights* (for different Type-III pairing curves) over groups $\mathbb{G}_1$ and $\mathbb{G}_2$. By encoding these cost values, it is feasible to create different weight functions that adhere to the user specified constraints. Once these functions are specified correctly, we minimize it across the set of assignments and return the solution that yields the lowest value. Thus, the combination of equations 3.1 and 3.2 yield all the possible ways a current user can optimize a given symmetric scheme. Further optimizations are future work.

Once the "best" solution is found, we have a CheckValidSplit procedure that verifies that the conditions (1) and (2) of a "valid split" hold as defined in Definition 3.3.1. If this solution satisfies these conditions, we are done. If not, we simply test the next best solution, because the solver caches all solutions and we record metadata about each solution in terms of efficiency and security.

3.3.1.6 Step 6: Evaluate and Process the Solution

Once a split is chosen, AutoGroup+ must reconstruct SDL for the asymmetric scheme *and assumption(s)*. It reuses the functionality provided by AutoGroup to construct the SDL as dictated by the split.[8] To output the new asymmetric assumptions, AutoGroup+ follows the logic of Abe et al. [44] (although they did not implement this step) and implements a new procedure that uses the graph split to reconstruct the asymmetric assumption(s). For each element in the asymmetric assumption, we learn the new assignments of the elements using the graph split and mechanically generate the asymmetric assumption SDL. Finally, we rely on existing tools [11, 12] to translate the new asymmetric SDL representation into executable code for C++ or Python, or simply LaTeX.

3.3.2 Analysis of AutoGroup+

We analyze AutoGroup+'s security and optimizations.

Security. At a high-level, the Abe et al. [44] security argument works as follows. In the Type-I setting, we treat $\mathbb{G}_1 = \mathbb{G}_2$ because there are efficient isomorphisms between these two groups. However, suppose we work in the generic Type-I group model, where elements are a black box and to compute this isomorphism, a party must utilize an oracle $\mathcal{O}$. Next, consider moving to a Type-III group, where every element (for which the discrete logarithm is known with respect to the base generators) is duplicated; that

[8] We further perform an efficiency check on the final scheme as previously done in AutoGroup.

is, for $h = g^x \in \mathbb{G}$, we have $h_1 = g_1^x \in \mathbb{G}_1$ and $h_2 = g_2^x \in \mathbb{G}_2$. Then in the generic Type-III group model, we can simulate having efficiently computable isomorphisms between these groups by exposing an oracle $\mathcal{O}'$ that on input $d_1 \in \mathbb{G}_1$ outputs $d_2 \in \mathbb{G}_2$ (or vice versa). In essence, by exposing the "corresponding" group element (through the oracle in the Type-III setting), we "allow" all necessary isomorphism computations for the scheme itself to operate, however, at the same time, we can argue that any adversary that breaks this scheme (with these elements exposed) can be turned into an attacker against the Type-I scheme, where these isomorphisms are natively computable. The resulting theorem was summarized in Theorem 3.1.1: namely, the Type-III conversion will be secure in the generic group model, if one follows the conversion criteria in [44] and the Type-I input was secure in the generic group model.

Thus, we must argue that the AutoGroup+ implementation satisfies the criteria in [44]. The dependency graphs are created and merged according to the same algorithm. (AutoGroup+ tracks some additional information on the side for optimization purposes.) What is required is that the splitting of the merged dependency graph satisfies Abe et al.'s notion of a "valid split."

Definition 3.3.1 (Valid Split [44]). Let $\Gamma = (V, E)$ be a dependency graph for $\Pi = (\mathsf{S}, \mathsf{R}, \mathsf{A})$, a tuple representing a scheme, reduction and assumption(s) that are in the set covered by the [44] translation. Let $P = (p_1[0], \ldots, p_n[1]) \subset V$ be pairing nodes. A pair of graphs $\Gamma_0 = (V_0, E_0)$ and $\Gamma_1 = (V_1, E_1)$ is a valid split of Γ with respect to $\mathsf{NoDup} \subseteq V$ if the following hold:

1. merging Γ_0 and Γ_1 recovers Γ,

2. for each $i \in \{0, 1\}$ and every $X \in V_i \backslash P$, the ancestor subgraph of X in Γ is included in Γ_i.

3. for each $i \in \{1, \ldots, n_p\}$ pairing nodes $p_i[0]$ and $p_i[1]$ are separately included in V_0 and V_1,

4. No node in $V_0 \cap V_1$ is included in NoDup. NoDup is a list of nodes that cannot be assigned to both V_0 and V_1.

In terms of AutoGroup+ security, conditions (1) and (2) are satisfied in the search procedure (step 5). That is, before we admit a split, we do these simple tests. Condition (3) is satisfied by the SMT solver with the logical formula encoding of pairing nodes (step 4). Condition (4) is also satisfied by the SMT solver (step 4). We encode the output of hashes as constraints over the logical formula; specifically, we ask the solver to find splits that keep hashes in $\mathbb{G}_1$. This is the only place we differ slightly. Abe et al. allow $\mathbb{G}_1$ or $\mathbb{G}_2$ assignment for hashes but not both. Our approach prioritizes solutions that preserve efficiency but we could give the user the option of relaxing this to match Abe et al. The translation back to SDL is fairly straightforward from the split.

Optimizations. In terms of optimality over the set of solutions admitted by the "valid split" method, AutoGroup+ finds the "best" one by searching over the entire set. It does this efficiently by turning the user-specified optimizations into the appropriate

objective function and passing this function into the SMT solver. Our experiments in Section 3.5 provide evidence that the tool is, indeed, finding the optimal solutions over the space of valid translations.

As discussed in Section 3.1.1, we do not rule out the existence of even better solutions that employ insights outside of this method (such as altering the construction or adding "stronger" assumptions, such as SXDH.)

3.4 An Automation Example with BB-HIBE

In this section, we illustrate each phase of the AutoGroup+ implementation described in Section 3.3 by showing the step-by-step translation of the Boneh-Boyen hierarchical identity-based encryption [56] (or BB HIBE) scheme. We begin by recalling the scheme: an efficient HIBE scheme (with $\ell = 2$) [57, §4.1] that is selective identity secure based on the standard Decisional Bilinear-Diffie Hellman (DBDH) assumption.

This scheme consists of four algorithms: Setup, KeyGen, Encrypt and Decrypt. The Setup algorithm takes as input a security parameter and defines public keys (ID) of depth ℓ as vectors of elements in $\mathbb{Z}_p^\ell$. We define $\ell = 2$, thus the identity is comprised of $\mathsf{ID} = (ID_1, ID_2) \in \mathbb{Z}_p^2$. The algorithm generates system parameters as follows. First, select a random generator $g \in \mathbb{G}$, a random $\alpha \in \mathbb{Z}_p$, and sets $g_1 = g^\alpha$. Then, pick random $h_1, h_2, g_2 \in \mathbb{G}$. Set the master public parameters $params = (g, g_1, g_2, h_1, h_2)$

and the master secret key $msk = {g_2}^{\alpha}$.

The KeyGen algorithm takes as input an $\mathsf{ID} = (ID_1, ID_2) \in {\mathbb{Z}_p}^2$, picks random $r_1, r_2 \in Z_p$ and outputs:

$$d_1 = {g_2}^{\alpha} \cdot ({g_1}^{ID_1} \cdot h_1)^{r_1} \cdot ({g_1}^{ID_2} \cdot h_2)^{r_2}, d_2 = g^{r_1}, d_3 = g^{r_2}$$

and the algorithm outputs $d_{ID} = (d_1, d_2, d_3)$

The Encrypt algorithm takes as input the public parameters $params$, an identity ID and a message $M \in \mathbb{G}_T$. To encrypt the message M under the public key $\mathsf{ID} = (ID_1, ID_2)$, picks a random $s \in \mathbb{Z}_p$ and computes:

$$C = (e(g_1, g_2)^s \cdot M, g^s, ({g_1}^{ID_1} \cdot h_1)^s, ({g_1}^{ID_2} \cdot h_2)^s)$$

and the algorithm outputs $C = (C_1, C_2, C_3, C_4)$.

The Decrypt algorithm takes as input a private key $d_{ID} = (d_1, d_2, d_3)$ and a ciphertext C and computes M as:

$$M = C_1 \cdot \frac{e(C_3, d_2) \cdot e(C_4, d_3)}{e(C_2, d_1)}$$

The scheme is based on the DBDH assumption.

Assumption 1 (Decisional Bilinear Diffie-Hellman). Let g generate group $\mathbb{G}$ of prime order $p \in \Theta(2^{\lambda})$ with mapping $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$. For all p.p.t. adversaries $\mathcal{A}$, the

following probability is negligible in λ:

$$|\tfrac{1}{2} - \Pr[a, b, c \leftarrow \mathbb{Z}_p, z \leftarrow \{0,1\}, A = g^a,$$

$$B = g^b, C = g^c, T_0 = e(g,g)^{abc}, T_1 \leftarrow \mathbb{G}_T;$$

$$z' \leftarrow \mathcal{A}(g, A, B, C, T_z) : z = z']|.$$

3.4.0.0.1 Step 1: Generating SDL Inputs

In order for AutoGroup+ to perform the translation, we first begin by transcribing the scheme, reduction and the DBDH assumption into SDL. We provide the SDL description of the above scheme, reduction and assumption in Appendix D. The reader will notice that the SDL descriptions closely and concisely follow the paper counterpart. This design is on purpose as to reduce the burden of transcribing these constructions for AutoGroup+ users. Indeed, in our experience the most time consuming and tedious part is in specifying the reductions accurately.

3.4.0.0.2 Step 2: Extracting the Dependencies

Once the SDLs have been generated along with the metadata and the user's desired optimization goal, the user can proceed with executing AutoGroup+ to begin deriving the dependency graphs for each input file. AutoGroup+ programmatically extracts the dependencies from the SDL descriptions starting with the assumption(s), then the reduction(s) and finally, the scheme. The dependency graph diagrams for BB

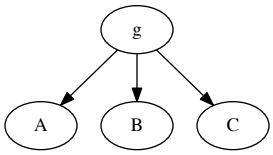


Figure 3.2: Dependency graph for the DBDH instance generated by AutoGroup+.

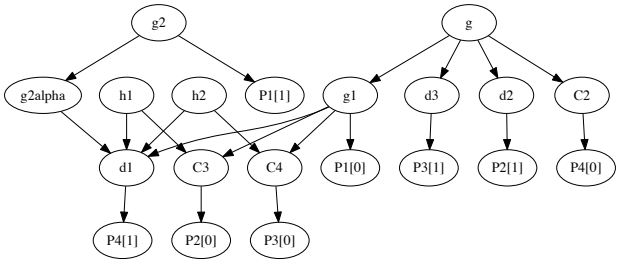


Figure 3.3: Dependency graph that merges Setup, KeyGen, Encrypt and Decrypt algorithms in BB HIBE and generated by AutoGroup+. For brevity, we only show the combined scheme graph and omit the smaller ones for each routine in the scheme. Note that nodes $P1$ through $P4$ represent unique pairing identifiers, with a 0 index representing a left-hand pairing element and a 1 the right.

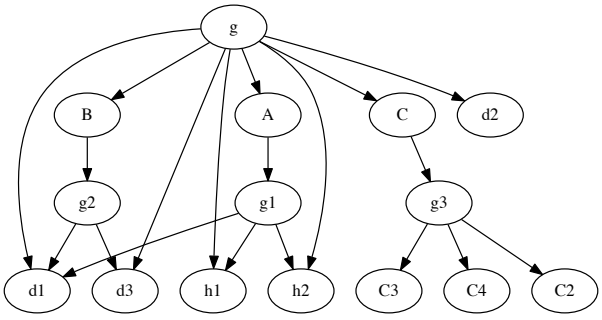


Figure 3.4: Dependency graph for the reduction to DBDH in BB HIBE. This graph was generated by AutoGroup+.

HIBE [57, §4.1] are included in Figures 3.2, 3.3, and 3.4. Note that these diagrams were generated automatically by our tool; we believe this feature provides more transparency to make it easier for humans to verify that the software is operating correctly. In "naming" the nodes of our dependency graphs, we closely follow the naming conventions that the user employed in the SDL, thus supporting the quick and easy verification.

3.4.0.0.3 Step 3: Merge the Graphs

In Figure 3.5, we show the third step in AutoGroup+ which is to merge the multiple dependency graphs (assumption, reduction and scheme graphs) into one single graph.

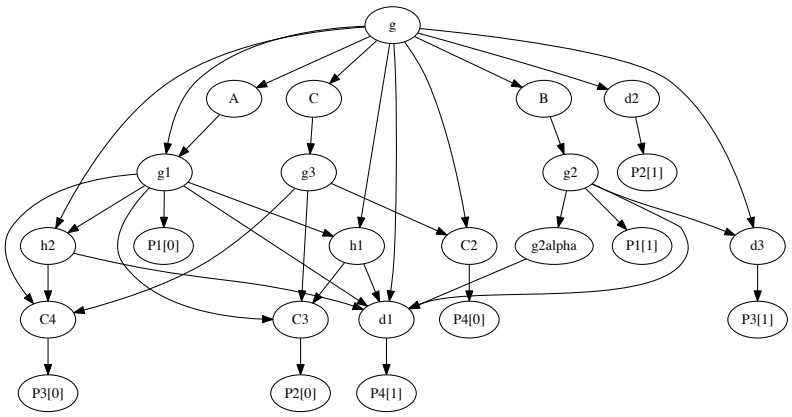


Figure 3.5: The merged dependency graph for the assumption, reduction to DBDH, and the BB HIBE scheme. This graph was generated by AutoGroup+.

3.4.0.0.4 Step 4: Assignment of Variables

With the merged graph, we encode the pairing equations as a logical formula as in AutoGroup but also encode certain group elements in the dependency graph as additional constraints to the solver (with optimization requirements):

$$P1[0] \neq P1[1] \wedge P2[0] \neq P2[1] \wedge P3[0] \neq P3[1] \wedge P4[0] \neq P4[1]$$

Recall that pairing identifiers (*e.g.*, $P2[0]$, $P2[1]$) are unique references which refer to pairing inputs from the scheme (*e.g.*, $e(C_3, d_2)$).

3.4.0.0.5 Step 5: Search for an Optimal Solution

In our BB HIBE example, the goal is to minimize the number of splits in the master public parameters *params*, so this requires specifying the following parameters of the EvalGraph function. Let $B = \{g, g_1, g_2, h_1, h_2\}$ be the set of elements in the public parameters we wish to minimize and let G be an encoding of the merged dependency graph shown in Figure 3.5. As reflected in Table 3.8, the solver identifies 16 possible solutions for the BB HIBE scheme and computes the following on each solution as $S_j = \mathsf{EvalGraph}(A_j, B, G)$ where A_j is the j-th set of possible variable assignments. Recall that EvalGraph simply applies a given solution to G and records how elements of B are assigned. From the set S, the solver finds an assignment that has the fewest number of duplicated public key elements with the smallest overall size. Based on this

criteria, the solver returned a optimal solution in the fifth step which consisted of 2 splits (i.e., two duplicated elements). The new public key elements are assigned as $B' = \{g, \tilde{g}, g_1, g_2, \tilde{g_2}, h_1, h_2\} \in \mathbb{G}_1^5 \times \mathbb{G}_2^2$. This constitutes only an addition of 2 group elements in $\mathbb{G}_2$.

3.4.0.0.6 Step 6: Assignment of Variables

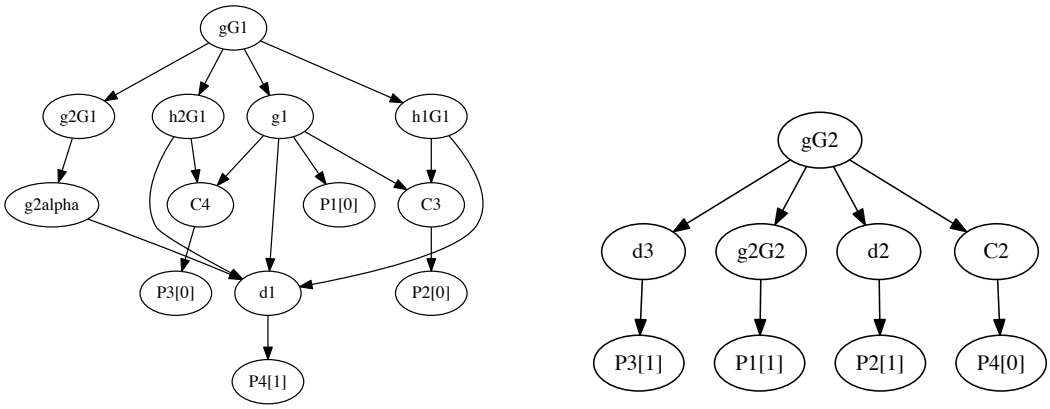
In the last step, AutoGroup+ splits the graph as dictated by the optimal solution found by the solver. The resulting graphs for $\mathbb{G}_1$ and $\mathbb{G}_2$ assignments for the BB HIBE scheme are shown in Figure 3.6. AutoGroup+ programmatically converts the split graph into an asymmetric translation for the scheme and assumption. We improve on code from AutoGroup to do the former translation and write a new module to do the latter (see Figure 3.7 for the graph split of co-DBDH). These resulting SDL files are provided in Appendix D.2. As mentioned before, there is a publicly-available tool (see Section 3.2.3) for automatically turning this SDL into C++, Python or LaTeX.

3.5 AutoGroup+: Experimental Evaluation

We tested AutoGroup+ on 9 schemes, with 3-4 optimization options and 4 different levels of BB HIBE, for 48 total translations.[9] Figure 3.8 summarizes the translation times and resulting scheme sizes.[10] To demonstrate the improvement in running

[9] Currently the tool does not support the assumption minimization option for schemes with more than one assumption. This is future work, although we would like to explore how valuable assumption minimization is to tool users.

[10] We only give details for two variations of BB HIBE because the results are similar for all levels.



(a) Showing $\mathbb{G}_1$ elements in the scheme (b) Showing $\mathbb{G}_2$ elements in the scheme

Figure 3.6: The dependency graphs for the asymmetric translation of BB HIBE scheme only (with PK optimization). This graph was generated by AutoGroup+.

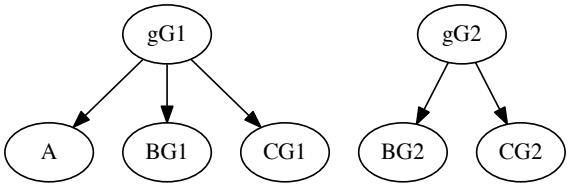


Figure 3.7: The dependency graph for the co-DBDH assumption and generated by AutoGroup+.

	Conversion Time	Number of Group Elements				Assumption	Num. Solutions
		Public Key	Secret Key	Ciphertext	Assumption		
ID-Based Enc.							
BB04 HIBE [56, §4] Symmetric (l = 2)	-	$\mathbb{G}^5$	$\mathbb{G}^3$	$\mathbb{G}^3 \times \mathbb{G}_T$	$\mathbb{G}^4 \times \mathbb{G}_T$	DBDH	
Asymmetric [Min. PK]	592 ms	$\mathbb{G}_1^5 \times \mathbb{G}_2^2$	$\mathbb{G}_1 \times \mathbb{G}_2^2$	$\mathbb{G}_1^2 \times \mathbb{G}_2 \times \mathbb{G}_T$	$\mathbb{G}_1^4 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		16
Asymmetric [Min. SK]	641 ms	$\mathbb{G}_1^5 \times \mathbb{G}_2^4$	$\mathbb{G}_1^3$	$\mathbb{G}_2^3 \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		16
Asymmetric [Min. CT]	626 ms	$\mathbb{G}_1^4 \times \mathbb{G}_2^5$	$\mathbb{G}_2^3$	$\mathbb{G}_1^3 \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		16
Asymmetric [Min. Assump]	582 ms	$\mathbb{G}_1^4 \times \mathbb{G}_2^5$	$\mathbb{G}_2^3$	$\mathbb{G}_1^3 \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		16
BB04 HIBE [56, §4] Symmetric (l = 9)	-	$\mathbb{G}^{12}$	$\mathbb{G}^{10}$	$\mathbb{G}^{10} \times \mathbb{G}_T$	$\mathbb{G}^4 \times \mathbb{G}_T$	DBDH	
Asymmetric [Min. PK]	20629 ms	$\mathbb{G}_1^{12} \times \mathbb{G}_2^2$	$\mathbb{G}_1 \times \mathbb{G}_2^9$	$\mathbb{G}_1^9 \times \mathbb{G}_2 \times \mathbb{G}_T$	$\mathbb{G}_1^4 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		2048
Asymmetric [Min. SK]	15714 ms	$\mathbb{G}_1^{12} \times \mathbb{G}_2^{11}$	$\mathbb{G}_1^{10}$	$\mathbb{G}_2^{10} \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		2048
Asymmetric [Min. CT]	15690 ms	$\mathbb{G}_1^{11} \times \mathbb{G}_2^{12}$	$\mathbb{G}_2^{10}$	$\mathbb{G}_1^{10} \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		2048
Asymmetric [Min. Assump]	20904 ms	$\mathbb{G}_1^{11} \times \mathbb{G}_2^{12}$	$\mathbb{G}_2^{10}$	$\mathbb{G}_1^{10} \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		2048
GENTRY06 [58, §3.1] Symmetric	-	$\mathbb{G}^3$	$\mathbb{Z}_p \times \mathbb{G}$	$\mathbb{G} \times \mathbb{G}_T^2$	$\mathbb{G}^{3+q} \times \mathbb{G}_T$	trunc. dec. q-ABDHE	
Asymmetric [Min. PK]	669 ms	$\mathbb{G}_1^2 \times \mathbb{G}_2^2$	$\mathbb{Z}_p \times \mathbb{G}_2$	$\mathbb{G}_1 \times \mathbb{G}_T^2$	$\mathbb{G}_1^{3+q} \times \mathbb{G}_2^{2+q} \times \mathbb{G}_T$		4
Asymmetric [Min. SK]	718 ms	$\mathbb{G}_1^2 \times \mathbb{G}_2^3$	$\mathbb{Z}_p \times \mathbb{G}_1$	$\mathbb{G}_2 \times \mathbb{G}_T^2$	$\mathbb{G}_1^{2+q} \times \mathbb{G}_2^{3+q} \times \mathbb{G}_T$		4
Asymmetric [Min. CT]	723 ms	$\mathbb{G}_1^2 \times \mathbb{G}_2^2$	$\mathbb{Z}_p \times \mathbb{G}_2$	$\mathbb{G}_1 \times \mathbb{G}_T^2$	$\mathbb{G}_1^{3+q} \times \mathbb{G}_2^{2+q} \times \mathbb{G}_T$		4
Asymmetric [Min. Assump]	676 ms	$\mathbb{G}_1^2 \times \mathbb{G}_2^2$	$\mathbb{Z}_p \times \mathbb{G}_2$	$\mathbb{G}_1 \times \mathbb{G}_T^2$	$\mathbb{G}_1^{3+q} \times \mathbb{G}_2^{1+q} \times \mathbb{G}_T$		4
WATERS05 [54, §4] Symmetric	-	$\mathbb{G}^{4+n}$	$\mathbb{G}^2$	$\mathbb{G}^2 \times \mathbb{G}_T$	$\mathbb{G}^4 \times \mathbb{G}_T$	DBDH	
Asymmetric [Min. PK]	725 ms	$\mathbb{G}_1^{4+n} \times \mathbb{G}_2^2$	$\mathbb{G}_1 \times \mathbb{G}_2$	$\mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_T$	$\mathbb{G}_1^4 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		8
Asymmetric [Min. SK]	770 ms	$\mathbb{G}_1^{4+n} \times \mathbb{G}_2^{3+n}$	$\mathbb{G}_1^2$	$\mathbb{G}_2^2 \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		8
Asymmetric [Min. CT]	767 ms	$\mathbb{G}_1^{3+n} \times \mathbb{G}_2^{4+n}$	$\mathbb{G}_2^2$	$\mathbb{G}_1^2 \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		8
Asymmetric [Min. Assump]	716 ms	$\mathbb{G}_1^{4+n} \times \mathbb{G}_2^{3+n}$	$\mathbb{G}_1^2$	$\mathbb{G}_2^2 \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		8
WATERS09 (DSE) [52, §3.1] Symmetric	-	$\mathbb{G}^{13} \times \mathbb{G}_T$	$\mathbb{G}^8 \times \mathbb{Z}_p$	$\mathbb{Z}_p \times \mathbb{G}^9 \times \mathbb{G}_T$	$(\mathbb{G}^4 \times \mathbb{G}_T), (\mathbb{G}^6), (\mathbb{G}^6)$	DBDH, DLIN, DLIN	
Asymmetric [Min. PK]	6217 ms	$\mathbb{G}_1^{10} \times \mathbb{G}_2^4 \times \mathbb{G}_T$	$\mathbb{G}_1^4 \times \mathbb{G}_2^4 \times \mathbb{Z}_p$	$\mathbb{G}_1^5 \times \mathbb{G}_2^4 \times \mathbb{G}_T$	$(\mathbb{G}_1^4 \times \mathbb{G}_2^3 \times \mathbb{G}_T), (\mathbb{G}_1^6 \times \mathbb{G}_2^6)$		256
Asymmetric [Min. SK]	5871 ms	$\mathbb{G}_1^7 \times \mathbb{G}_2^{13} \times \mathbb{G}_T$	$\mathbb{G}_1^8 \times \mathbb{Z}_p$	$\mathbb{G}_2^9 \times \mathbb{G}_T$	$(\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T), (\mathbb{G}_1^6 \times \mathbb{G}_2^6), (\mathbb{G}_1^6 \times \mathbb{G}_2^6)$		256
Asymmetric [Min. CT]	5858 ms	$\mathbb{G}_1^{13} \times \mathbb{G}_2^7 \times \mathbb{G}_T$	$\mathbb{G}_2^8 \times \mathbb{Z}_p$	$\mathbb{G}_1^9 \times \mathbb{G}_T$	$(\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T), (\mathbb{G}_1^6 \times \mathbb{G}_2^6), (\mathbb{G}_1^6 \times \mathbb{G}_2^6)$		256
Asymmetric [Min. Assump]	6228 ms	$\mathbb{G}_1^{12} \times \mathbb{G}_2^5 \times \mathbb{G}_T$	$\mathbb{G}_1^3 \times \mathbb{G}_2^5 \times \mathbb{Z}_p$	$\mathbb{G}_1^6 \times \mathbb{G}_2^3 \times \mathbb{G}_T$	$(\mathbb{G}_1^4 \times \mathbb{G}_2^2 \times \mathbb{G}_T), (\mathbb{G}_1^6 \times \mathbb{G}_2^6), (\mathbb{G}_1^6 \times \mathbb{G}_2^6)$		256
Broadcast Encryption							
BGW05 [59, §3.1] Symmetric (n users)	-	$\mathbb{G}^{2n+1}$	$\mathbb{G}$	$\mathbb{G}^3$	$\mathbb{G}^{2l+1} \times \mathbb{G}_T$	decision l-BDHE	
Asymmetric [Min. PK]	530 ms	$\mathbb{G}_1^{2n+1} \times \mathbb{G}_2^{2n}$	$\mathbb{G}_2$	$\mathbb{G}_1^2 \times \mathbb{G}_T$	$\mathbb{G}_1^{2l} \times \mathbb{G}_2^{2l+1} \times \mathbb{G}_T$		4
Asymmetric [Min. SK]	601 ms	$\mathbb{G}_1^{2n} \times \mathbb{G}_2^{2n+1}$	$\mathbb{G}_1$	$\mathbb{G}_2^2 \times G_T$	$\mathbb{G}_1^{2l} \times \mathbb{G}_2^{2l+1} \times \mathbb{G}_T$		4
Asymmetric [Min. CT]	587 ms	$\mathbb{G}_1^{2n+1} \times \mathbb{G}_2^{2n}$	$\mathbb{G}_2$	$\mathbb{G}_1^2 \times \mathbb{G}_T$	$\mathbb{G}_1^{2l} \times \mathbb{G}_2^{2l+1} \times \mathbb{G}_T$		4
Asymmetric [Min. Assump]	544 ms	$\mathbb{G}_1^{2n+1} \times \mathbb{G}_2^{2n}$	$\mathbb{G}_2$	$\mathbb{G}_1^2 \times \mathbb{G}_T$	$\mathbb{G}_1^{2l} \times \mathbb{G}_2^{2l+1} \times \mathbb{G}_T$		4
Signature							
ACDKNO [60, §5.3] Symmetric	-	$\mathbb{G}^{15}$	$\mathbb{G}^2$	$\mathbb{G}^8$	$(\mathbb{G}^4), (\mathbb{G}^6), (\mathbb{G}^6)$	CDH, DLIN, DLIN	
Asymmetric [Min. PK]	18216 ms	$\mathbb{G}_1^{14} \times \mathbb{G}_2^5$	$\mathbb{G}_2^2$	$\mathbb{G}_1 \times \mathbb{G}_2^7$	$(\mathbb{G}_1^2 \times \mathbb{G}_2^4), (\mathbb{G}_1^2 \times \mathbb{G}_2^6), (\mathbb{G}_1^2 \times \mathbb{G}_2^6)$		1024
Asymmetric [Min. Sig]	14689 ms	$\mathbb{G}_1^6 \times \mathbb{G}_2^{14}$	$\mathbb{G}_1^2$	$\mathbb{G}_1^8$	$(\mathbb{G}_1^4 \times \mathbb{G}_2^2), (\mathbb{G}_1^6 \times \mathbb{G}_2^2), (\mathbb{G}_1^6 \times \mathbb{G}_2^2)$		1024
Asymmetric [Min. Assump]	18135 ms	$\mathbb{G}_1^5 \times \mathbb{G}_2^{14}$	$\mathbb{G}_1^2$	$\mathbb{G}_1^7 \times \mathbb{G}_2$	$(\mathbb{G}_1^4 \times \mathbb{G}_2^2), (\mathbb{G}_1^6 \times \mathbb{G}_2^2), (\mathbb{G}_1^6 \times \mathbb{G}_2^2)$		1024
BLS [61, §2.2] Symmetric	-	$\mathbb{G}^2$	$\mathbb{Z}_p^*$	$\mathbb{G}$	$\mathbb{G}^4$	CDH	
Asymmetric [Min. PK]	515 ms	$\mathbb{G}_2^2$	$\mathbb{Z}_p^*$	$\mathbb{G}_1$	$(\mathbb{G}_1^4 \times \mathbb{G}_2^3), (\mathbb{G}_1^3 \times \mathbb{G}_2^3), (\mathbb{G}_1^3 \times \mathbb{G}_2^3)$		2
Asymmetric [Min. Sig]	556 ms	$\mathbb{G}_2^2$	$\mathbb{Z}_p^*$	$\mathbb{G}_1$	$(\mathbb{G}_1^4 \times \mathbb{G}_2^3), (\mathbb{G}_1^3 \times \mathbb{G}_2^3), (\mathbb{G}_1^3 \times \mathbb{G}_2^3)$		2
Asymmetric [Min. Assump]	517 ms	$\mathbb{G}_2^2$	$\mathbb{Z}_p^*$	$\mathbb{G}_1$	$(\mathbb{G}_1^4 \times \mathbb{G}_2^3), (\mathbb{G}_1^3 \times \mathbb{G}_2^3), (\mathbb{G}_1^3 \times \mathbb{G}_2^3)$		2
CL04 [55, §3.1] Symmetric	-	$\mathbb{G}^3$	$\mathbb{Z}_p^{*2}$	$\mathbb{G}^3$	$\mathbb{G}^3$	LRSW	
Asymmetric [Min. PK]	278 ms	$\mathbb{G}_1^3 \times \mathbb{G}_2$	$\mathbb{Z}_p^{*2}$	$\mathbb{G}_2^3$	$\mathbb{G}_1^3$		2
Asymmetric [Min. Sig]	328 ms	$\mathbb{G}_1 \times \mathbb{G}_2^3$	$\mathbb{Z}_p^{*2}$	$\mathbb{G}_1^3$	$\mathbb{G}_2^3$		2
Asymmetric [Min. Assump]	275 ms	$\mathbb{G}_1^3 \times \mathbb{G}_2$	$\mathbb{Z}_p^{*2}$	$\mathbb{G}_2^3$	$\mathbb{G}_1^3$		2
WATERS05 [54, §7] Symmetric	-	$\mathbb{G}^{4+n}$	$\mathbb{G}$	$\mathbb{G}^2$	$\mathbb{G}^4 \times \mathbb{G}_T$	DBDH	
Asymmetric [Min. PK]	724 ms	$\mathbb{G}_1^3 \times \mathbb{G}_2$	$\mathbb{G}_2^3$	$\mathbb{G}_1 \times \mathbb{G}_2$	$\mathbb{G}_1^4 \times \mathbb{G}_2^2 \times \mathbb{G}_T$		8
Asymmetric [Min. Sig]	721 ms	$\mathbb{G}_1^{4+n} \times \mathbb{G}_2^{3+n}$	$\mathbb{G}_1$	$\mathbb{G}_1^2$	$\mathbb{G}_1^3 \times \mathbb{G}_2^3 \times \mathbb{G}_T$		8
Asymmetric [Min. Assump]	755 ms	$\mathbb{G}_1^{4+n} \times \mathbb{G}_2^2$	$\mathbb{G}_1$	$\mathbb{G}_1 \times \mathbb{G}_2$	$\mathbb{G}_1^4 \times \mathbb{G}_2^2 \times \mathbb{G}_T$		8

Figure 3.8: A summary of the experimental evaluations of AutoGroup+ on a variety of schemes and optimization options. For the symmetric baseline with curve SS1536, elements in $\mathbb{G}$ are 1536 bits and $\mathbb{G}_T$ are 3072 bits. For the asymmetric translations with BN256, elements in $\mathbb{G}_1$ are 256 bits, $\mathbb{G}_2$ are 1024 bits, and $\mathbb{G}_T$ are 3072 bits. For BGW05, the private key size is listed for a single user.

	Time•			
	Setup	**Keygen**	**Encrypt/Sign**	**Decrypt/Verify**
ID-Based Enc.				
BB04 HIBE [56, §4] Symmetric (SS1536) (l = 2)	346.47 ms	84.75 ms	118.64 ms	133.48 ms
Asymmetric (BN256) [Min. PK]	5.09 ms	4.79 ms	12.92 ms	21.36 ms
Asymmetric (BN256) [Min. SK]	8.15 ms	2.95 ms	14.95 ms	21.32 ms
Asymmetric (BN256) [Min. CT]	9.84 ms	6.23 ms	12.38 ms	21.22 ms
Asymmetric (BN256) [Min. Assump]	9.08 ms	7.30 ms	12.27 ms	21.64 ms
BB04 HIBE [56, §4] Symmetric (SS1536) (l = 9)	892.69 ms	283.11 ms	217.39 ms	446.10 ms
Asymmetric (BN256) [Min. PK]	9.25 ms	17.64 ms	17.10 ms	70.84 ms
Asymmetric (BN256) [Min. SK]	20.53 ms	11.14 ms	24.36 ms	71.45 ms
Asymmetric (BN256) [Min. CT]	21.60 ms	27.02 ms	16.48 ms	72.03 ms
Asymmetric (BN256) [Min. Assump]	21.68 ms	31.96 ms	16.77 ms	70.48 ms
GENTRY06 [58, §3.1] Symmetric (SS1536)	172.30 ms	28.23 ms	137.79 ms	48.42 ms
Asymmetric (BN256) [Min. PK]	2.88 ms	2.47 ms	21.08 ms	10.01 ms
Asymmetric (BN256) [Min. SK]	4.22 ms	1.18 ms	22.46 ms	9.96 ms
Asymmetric (BN256) [Min. CT]	2.93 ms	2.53 ms	21.02 ms	10.02 ms
Asymmetric (BN256) [Min. Assump]	2.88 ms	2.53 ms	21.10 ms	10.09 ms
WATERS05 [54, §4] Symmetric (SS1536)	908.94 ms	29.78 ms	78.08 ms	111.76 ms
Asymmetric (BN256) [Min. PK]	10.31 ms	2.04 ms	11.98 ms	14.23 ms
Asymmetric (BN256) [Min. SK]	24.11 ms	1.37 ms	13.68 ms	14.11 ms
Asymmetric (BN256) [Min. CT]	25.39 ms	3.67 ms	11.25 ms	14.23 ms
Asymmetric (BN256) [Min. Assump]	23.81 ms	1.36 ms	13.71 ms	14.38 ms
WATERS09 (DSE) [52, §3.1] Symmetric (SS1536)	755.50 ms	195.27 ms	212.88 ms	414.79 ms
Asymmetric (BN256) [Min. PK]	23.13 ms	9.71 ms	13.70 ms	66.45 ms
Asymmetric (BN256) [Min. SK]	36.83 ms	7.07 ms	20.08 ms	66.42 ms
Asymmetric (BN256) [Min. CT]	34.41 ms	14.82 ms	11.08 ms	66.92 ms
Asymmetric (BN256) [Min. Assump]	29.90 ms	11.09 ms	13.03 ms	66.92 ms
Broadcast Encryption				
BGW05 [59, §3.1] Symmetric (SS1536) ($n = 10$)	376.84 ms	140.27 ms	86.96 ms	68.65 ms
Asymmetric (BN256) [Min. PK]	55.29 ms	13.98 ms	11.457 ms	6.13 ms
Asymmetric (BN256) [Min. SK]	38.45 ms	5.82 ms	12.49 ms	8.122 ms
Asymmetric (BN256) [Min. CT]	37.75 ms	12.32 ms	11.18 ms	6.27 ms
Asymmetric (BN256) [Min. Assump]	37.74 ms	12.31 ms	11.186 ms	6.12 ms
Signature				
ACDKNO [60, §5.3] Symmetric (SS1536)	395.23 ms	497.04 ms	275.99 ms	937.14 ms
Asymmetric (BN256) [Min. PK]	9.05 ms	17.19 ms	15.27 ms	147.62 ms
Asymmetric (BN256) [Min. Sig]	8.31 ms	22.65 ms	14.33 ms	152.60 ms
Asymmetric (BN256) [Min. Assump]	8.43 ms	22.23 ms	13.94 ms	147.77 ms
BLS [61, §] Symmetric (SS1536)	-	93.20 ms	92.61 ms	167.73 ms
Asymmetric (BN256) [Min. PK]	-	2.99 ms	0.74 ms	14.20 ms
Asymmetric (BN256) [Min. Sig]	-	3.00 ms	0.75 ms	14.20 ms
Asymmetric (BN256) [Min. Assump]	-	3.03 ms	0.69 ms	14.18 ms
CL04 [55, §3.1] (SS1536)	-	464.7 ms	178.18 ms	973.48 ms
Asymmetric (BN256) [Min. PK]	-	9.27 ms	15.12 ms	121.61 ms
Asymmetric (BN256) [Min. Sig]	-	14.54 ms	7.38 ms	119.16 ms
Asymmetric (BN256) [Min. Assump]	-	11.53 ms	15.32 ms	124.19 ms
WATERS05 [54, §7] (SS1536)	-	720.75 ms	29.72 ms	135.00 ms
Asymmetric (BN256) [Min. PK]	-	10.42 ms	2.02 ms	21.44 ms
Asymmetric (BN256) [Min. Sig]	-	25.60 ms	1.43 ms	23.13 ms
Asymmetric (BN256) [Min. Assump]	-	10.18 ms	2.01 ms	21.42 ms

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

Figure 3.9: A summary of the running times of the AutoGroup+ translations using curve BN256 as compared to the running times using the roughly security-equivalent symmetric curve SS1536 in MIRACL. The asymmetric setting plus AutoGroup+'s optimizations cut the running times by one or two orders of magnitude.

	Conversion Time	**Num. Solutions**
BB04 HIBE [56, §4] (l = 2)	-	-
Asymmetric [Min. PK]	592 ms	16
Asymmetric [Min. SK]	641 ms	16
Asymmetric [Min. CT]	626 ms	16
Asymmetric [Min. Assump]	582 ms	16
BB04 HIBE [56, §4] (l = 6)	-	-
Asymmetric [Min. PK]	2361 ms	256
Asymmetric [Min. SK]	2019 ms	256
Asymmetric [Min. CT]	2023 ms	256
Asymmetric [Min. Assump]	2375 ms	256
BB04 HIBE [56, §4] (l = 7)	-	-
Asymmetric [Min. PK]	4555 ms	512
Asymmetric [Min. SK]	3644 ms	512
Asymmetric [Min. CT]	3662 ms	512
Asymmetric [Min. Assump]	4519 ms	512
BB04 HIBE [56, §4] (l = 8)	-	-
Asymmetric [Min. PK]	9344 ms	1024
Asymmetric [Min. SK]	7148 ms	1024
Asymmetric [Min. CT]	7194 ms	1024
Asymmetric [Min. Assump]	9299 ms	1024
BB04 HIBE [56, §4] (l = 9)	-	-
Asymmetric [Min. PK]	20629 ms	2048
Asymmetric [Min. SK]	15714 ms	2048
Asymmetric [Min. CT]	15690 ms	2048
Asymmetric [Min. Assump]	20904 ms	2048

Figure 3.10: A summary of the conversion times of AutoGroup+ for various levels/degrees of complexity of BB04 HIBE [56, §4] and a variety of optimization options.

times due to both the asymmetric setting and AutoGroup+'s optimizations, Figure 3.9 includes over 140 timing experiments, showing drastic improvements. In Figure 3.10, we summarize the effect of scheme complexity on AutoGroup+ conversion time by varying the complexity of BB HIBE. We note that even given a more complex scheme than attempted by any other tool, AutoGroup+ still provides fast conversion times.

System Configuration. All of our benchmarks were executed on a standard workstation that has a 2.20GHz quad-core Intel Core i7-2720QM processor with 8GB RAM running Ubuntu 11.04 LTS, Linux Kernel version 2.6.38-16-generic (x86-64-bit architecture). Our measurements only use a single core of the Intel processor for consistency. The AutoGroup+ implementation utilizes the same building blocks as AutoGroup which include the MIRACL library (v5.5.4) and/or RELIC cryptographic toolkit [62], Charm v0.43 [50] in C++ or Python code, and the Z3 SMT solver (v4.3.2).

Limitations. In Section 3.3.1.1, we provide an example of a scheme which falls into a category of things that Abe et al. warned about and on which AutoGroup gets confused. AutoGroup tries to power through and split the hash output (which it cannot really do because the discrete log is unknown), so while it eventually outputs some SDL, this SDL is not a proper translation. Unlike AutoGroup, AutoGroup+ includes logic to output a warning when processing such inputs and continues trying to translate the scheme. If the verification check of a valid split fails (*e.g.*, due to hash split), then AutoGroup+ identifies the split as invalid and attempts checking the next best solution. If there are no such solutions, AutoGroup+ outputs no solution.

3.5.1 Comparison with ACSC/Charm

Our experiments have five schemes in common with public implementations in the Advanced Crypto Software Collection [49] and Charm [50]. Where we have matches, our new results confirm the security and optimality of those (unproven) implemented translations.

For Waters 2009 [52], we compare with the Charm implementation by Fan Zhang. For our PK-size optimization, our translation is 3 elements shorter (we split only g, whereas they split g, w, u, h.) For our ciphertext-size optimization, it looks the closest to theirs, but they do not match. Both translations have short ciphertexts leaving all base elements in $\mathbb{G}_1$. However, the Charm translation appears to have shifted some elements from the public key to the secret key and dropped some elements from the master secret key (e.g., we split v and include both in the MSK, because that is the naive way to do it, but they use the v split for $\mathbb{G}_1$ only in the Setup and then drop it from the MSK.) While we cannot confirm the security of this implementation using our tool (so we believe this is left as an open question), the tool did produce a translation with the same ciphertext-size that is secure.

For BGW 2005 [59], we compared with the C implementation on the ACSC website by Matt Steiner and Ben Lynn. Indeed, our translations that minimize the public parameters or ciphertext size are the same, and the same as their manual translation. We confirm security and PP/ciphertext-size optimality.

For BB HIBE [56], Charm has a full HIBE implementation. We tested it for a

minimum of 2 levels, but their implementation matches ours for ciphertext minimization, except that they add a precomputed pairing (element in $\mathbb{G}_T$) to the public key so that it does not have to be done per encryption. This impacts only efficiency. We confirm security and ciphertext-size optimality.

For CL [55], we can confirm that the Charm implementation is secure and public-key-size optimal. However, in the more likely event that one wants to minimize signature size, AutoGroup+ found a translation with a shorter signature.

For BLS [61], our translations also match. This is a simple case with only two translation options.

Charm [50] also includes variants of the Waters encryption and signature schemes [54] from 2005, but we translated the original schemes (as did [12, 44]), so our translations are not directly comparable to these Charm variants.

3.5.2 Comparison with Abe et al.

Abe et al. [44] tested their method on two encryption schemes: Waters 2005 [54] and Waters 2009 (Dual System Encryption) [52]. They looked at minimizing the size of the public key and the Type-III assumption. We conjecture that practitioners would be more interested in minimizing ciphertext or private key size, so our summary also includes those optimizations.

For Waters 2005, AutoGroup+ found the same construction as their semi-automated method. As remarked in Section 3.3.1.1, their dependency graph for this scheme

included some unnecessary dependencies. Waters [54] clearly states to choose g_2, u', u_i as fresh random generators, but Abe et al. explicitly "assume" that they are generated from a common generator g. From a functionality and security standpoint *of the Type-I scheme*, this distinction certainly does not matter. However, it does change the intermediate dependency graphs, which could in some cases affect the output (though it does not in this situation). Both their partial automation and our full automation of Waters 2005 took under one second.

For Waters 2009, AutoGroup+ first appeared to find a PK-optimized construction with one less group element than the PK-optimized construction of Abe et al. [44]. However, subsequent discussions [51] determined that this was merely the product of a different counting method; the numbers reported in this work are the correct ones for both AutoGroup+and the Abe et al. method.

In the original work [44], no schemes with interactive assumptions were reported on. In subsequent communcations [51], Abe et al. demonstrated a translation for the Camenisch-Lysyanskaya signatures [55] based on the interactive LSRW assumption. We derived the SDL files for the scheme, assumption and proof and ran it through AutoGroup+. The results matched.

Drawing and merging the dependency graphs by hand is tedious and becomes infeasible for a complex scheme like [60]. In addition, the Abe et al. graph splitting program took 1.75 hours for Waters09, whereas our tool handled everything in 6.5 seconds. Thus, we find that it is considerably easier and faster to transcribe the SDL

and use AutoGroup+.

3.5.3 Comparison with AutoGroup

The AutoGroup tool [12] was used as the starting point for our implementation, hence the name of AutoGroup+. Our 48 translation experiments overlap with AutoGroup in 14 points (seven schemes in common and they do fewer optimizations). For these 14, the tools found the same constructions. However, a major difference is that with AutoGroup+, we have security guarantees. This required us to write new SDL descriptions for all the assumptions and proofs involved.

Indeed, one crucial question was how the security logic would increase translation times. We focused our effort on leveraging an SMT Solver to help handle this security logic, which kept the running times of AutoGroup+ within a few seconds of AutoGroup.

In addition to the security logic we added, we also found that the public key optimization flag for encryption was not implemented. Because we wanted to compare our results with [44], we implemented it.

AutoGroup was tested on one signature scheme omitted here. Boneh-Boyen [63] has a nested proof structure that falls outside of the black box reductions considered in this work.

3.5.4 Comparison with manual translations

The Dual System Encryption scheme of Waters [52] has a few manual translations with a security analysis. Ramanna, Chatterjee and Sarkar [46] provide a variety of translations, one with the smallest public parameter/key size, at the cost of introducing some mild complexity assumptions. Similarly, Chen, Lim, Ling, Wang and Wee [47] presented a translation introducing the SXDH assumption, which achieved the shortest ciphertext size. These results are superior to those derived by AutoGroup+ and [12, 44], but it is not yet clear how to generalize and systematize the human creativity used.

3.6 Conclusions

Automation is the future for many cryptographic design tasks. This work successfully demonstrates automating a complex translation of a scheme from one algebraic setting to another. There was a demonstrated need for such a compiler both for pairing designers and implementors. Its realization combined and improved on contributions from the systems [12] and theory [44] communities. The result is a practical tool, AutoGroup+, that enables secure pairing translations for everyone.

3.7 Acknowledgments

The authors thank Masayuki Abe, Jens Groth, Miyako Ohkubo, Takeya Tango for very helpful discussions regarding this work and their prior CRYPTO 2014 work.

Chapter 4

New Domains for Cryptographic Applications

This chapter is based on joint work with Matthew Green and Ian Miers at Johns Hopkins University. The paper was originally published in Network and Distributed System Security Symposium (NDSS), 2013, ISOC [18], while the complete version that appears in this chapter was published at [64].

One of the barriers to the deployment of certain cryptographic schemes is often the need to have a trusted party for some part of them, whether it is setup or part of the actual execution of the scheme. These parties inherently require our trust, which we might not want to give them, and provide a large target for compromise, which might not always be detectable. In this chapter, we will discuss how to leverage a public, append-only ledger, like Bitcoin's blockchain, to remove the need for a trusted third

party in anonymous credentials schemes. These "decentralized" anonymous credentials are able to operate without a trusted credential issuer, allowing them to be practically deployed in a variety of settings.

4.1 Introduction

Traditionally, making statements about identity on the Internet, whether actual assertions of identity ("I am Spartacus") or about one's identity ("I am a gladiator") involves centralized providers who issue a credential attesting to that verification. These organizations, which include Certificate Authorities, DNS maintainers, or login providers like Google and Facebook, play a large role in securing internet infrastructure, email, and financial transactions. Our increasing reliance on these providers raises concerns about privacy and trust.

Anonymous credentials, introduced by Chaum [65] and developed in a line of subsequent works [66, 67, 68, 69, 70], represent a powerful solution to this privacy concern: they deprive even colluding credential issuers and verifiers of the ability to identify and track their users. Although credentials may involve direct assertions of identity, they may also be used for a large range of useful assertions, such as "my TPM says my computer is secure," "I have a valid subscription for content," "I have a certain reputation," or "I am eligible to vote."

Indeed, anonymous credentials have already seen several practical applications. The

most widely deployed example is the Direct Anonymous Attestation (DAA) portion of the Trusted Platform Module specification [71, 72]. DAA extends the standard attestation capabilities of the Trusted Platform Module to allow for anonymous attestations of TPM state and to admit *pseudonyms* that are cryptographically bound to the TPM's internal identity certificate.

Unfortunately, current anonymous credential systems such as DAA have a fundamental limitation: while identity certification itself can be performed by a variety of centralized and decentralized processes, all existing anonymous credential systems employ blind signatures and thus require the appointment of a central, *trusted* party to issue the credentials. This issuer represents a single point of failure and its signing key an obvious target for compromise, either of which can seriously damage the reliability of the credential system. Moreover, compromise or issuer malfeasance can be particularly difficult to detect in an anonymous credential system. As a result, in distributed settings such as ad hoc or peer-to-peer networks, it may be challenging to identify parties who can be trusted to play this critical role or verify that the trust is well placed. The ability to remove this trusted party or even verify their continued good behavior is a distinct advantage.

These challenges raise two questions: 1) is it possible to build practical anonymous credential systems where the process of issuing credentials — if not the establishment of identity itself — no longer depends on a trusted party? And 2) is it possible to do so without the need for a central party?

Our contribution. In this chapter we answer both questions in the affirmative, proposing a new technique for constructing anonymous credentials which does not rely on the continued integrity of signature keys. A consequence of this result is that our anonymous credential system can be instantiated on-demand and operated by an *ad hoc* group of mistrustful peers. We further show how to extend our credential scheme to create *updatable* (e.g., stateful) anonymous credentials in which users obtain new credentials based on changing properties of their identity.

As a basic ingredient, our protocols require the existence of a *public append-only ledger*. When the ledger is implemented using trusted hardware, or a central party who is audited by the rest of the network, we obtain a positive answer only to the first question. To answer both questions in the affirmative we require that 1) this ledger be maintained in a distributed manner that need not require a trusted party or parties and 2) the identity claims we are issuing credentials on must be verifiable by everyone participating in the system. We refer to this new primitive as a *decentralized anonymous credential* system and elaborate on its properties herein. We note that one promising instantiation of a decentralized ledger is the "block chain" construction used by Bitcoin [73] to implement a decentralized digital currency. Not only can this technology be used to actually construct a separate distributed ledger for identities, but using existing techniques for embedding small amounts of data in the block chain [74] we can leverage Bitcoin's existing ledger and protocol *without* modification to transform any reliable storage mechanism (whether a central server or a distributed

mechanism like a DHT) into an append-only ledger.

We show that our techniques have several immediate applications. They include:

- **Decentralized Direct Anonymous Attestation.** We show how to decentralize the Direct Anonymous Attestation protocol [71], allowing individual collections of nodes in an ad hoc or distributed system to securely assert properties of their system state. We provide an exemplary description of our decentralized (dDAA) construction.

- **Anonymous resource management in ad hoc networks.** Peer-to-peer networks are vulnerable to impersonation attacks, where a single party simulates many different peers in order to gain advantage against the network [75]. We show that our credentials may be useful in mitigating these attacks. The basic approach is to construct an *anonymous subscription service* [76, 77, 78] where parties may establish unique or costly pseudonyms (for example by submitting a valid TPM credential or paying a sum of digital currency). They can then assert possession on their identity under a specific set of restrictions, e.g., a limit to the number of requests they can make in each time period.

- **Auditable credentials.** Our techniques may also be used to extend existing centralized credential systems by allowing for public audit of issued credentials. This helps to guard against compromised credential issuers and allows the network to easily detect and revoke inappropriate credential grants. For example, in

Direct Anonymous Attestation (DAA) one might want to prevent a malicious DAA authority from covertly granting certificates to users who do not have a TPM or whose TPM did not attest.

Is decentralized credential issuance valuable? Before proceeding to describe our protocols, it is worth asking whether decentralizing the issuance of anonymous credentials is a useful goal at all. After all, identity credentialing is frequently a centralized process. One might ask: what do we gain by decentralizing the issuance of *anonymous* credentials?

A first response to this question is that most anonymous credential systems separate the process of *issuing* anonymous credentials from the process of *certifying* the underlying identity claims. Frequently, the claims being certified are publicly verifiable. For example, each TPM ships with an Endorsement Key (EK). Identity assertions using the EK could be publicly verifiable merely by checking the certificate chain on the EK certificate and engaging in a challenge/response protocol to ensure the TPM can read nonces encrypted to the EK.[1] The problem is that transactions conducted using this certificate are linked to the particular TPM device.

DAA solves this issue by having a central party issue new anonymous credentials to a device. Organizations must configure a local server to validate identity certifications and issue the corresponding anonymous credential. All this server does is transform a publicly verifiable identity assertion into an anonymous one. This adds a cumbersome

[1]Conceptually the TPM's EK can sign a statement and forgo any interactive issuing process. The TPM 1.1 spec places an arbitrary restriction against using the EK RSA key for signing.

step to the anonymous attestation system and also introduces a point of failure. Indeed, this pattern of a trusted party transforming existing credentials into an *anonymous* credential repeats in many settings. Allowing for the distributed issue of anonymous credentials, even if they can only certify centrally validated assertions, removes this additional point of trust.

An obvious question is why, if the identity assertion is publicly verifiable, do we need any transformation mechanism at all? Why not present the information we used to convince the authority to issue the credential to everyone? The issue is that proving an identity statement may reveal far more information than the statement itself. For example, a driver's license can prove to anyone that the bearer is over 21 but also reveals a whole host of other information that the statement that "some trusted mechanism says I am over 21" does not. Because anonymous credentials add a layer of indirection between certifying that an identity statement is true and actually showing that statement, they fix this issue and avoid linking any use of the credential to the information used to issue it.

A more interesting question is whether identity certification itself can be decentralized. At least for certain claims, this seems like a promising direction. For example, non–extended validation SSL certificates are simply an assertion that the bearer controls the specified domain.[2] Similarly, DNS names are generally an assertion that the owner was the first to register that name and wants it mapped to certain values

[2]In practice, CA's usually verify that the bearer controls some administrator email such as admin@domain or webmaster@domain.

(e.g., an IP address). In both cases, since these claims are publicly verifiable by simple criteria, a distributed set of entities can easily validate these claims for themselves.

In fact, a now largely unused fork of Bitcoin, Namecoin [79], shows that such modifications are readily achievable. Namecoin uses Bitcoin's append-only ledger mechanism to maintain such first-come first-serve name-value mappings. Individuals register a name and an owning public key. Provided they are the first to register that name, they can make arbitrary updates to the associated value by signing them with the registered key. A DNS system built atop this — DotBIT — is already in experimental deployment. Namecoin can also be used to maintain mappings from names to public keys. One could imagine more complex semantics for allowing name registration — e.g., proofs of work, proofs of payment, TPM attestations, publicly verifiable proofs of storage and retrievability of files [80] — supporting more sophisticated functionality than simple DNS.

4.1.1 Overview of Our Construction

We now provide a brief overview for our construction, which is inspired by the electronic cash proposals of Sander and Ta-Shma [81] and Miers et al. [82].

Issuing and showing credentials. The ability to establish identities and bind them to a public key ensures that users can assert their identity in a non-anonymous fashion, simply by issuing signatures from the corresponding secret key. Unfortunately, this does not immediately show us how to construct anonymous credentials, since traditional

anonymous credentials consist of a signature computed by a credential issuer. Since no central party exists to compute the credential signature, this approach does not seem feasible without elaborate (and inefficient) use of threshold cryptography.[3]

We instead take a different approach. To issue a new credential in our decentralized system, the user establishes an identity and related attributes as described above. She then attaches a vector commitment to her secret key $sk_{\mathcal{U}}$ along with the identity and attribute strings that are contained within her identity assertion. Finally, she includes a non-interactive proof that the credential is correctly constructed, i.e., that the attributes in the commitment correspond to those revealed in the identity assertion. The network will accept the identity assertion if and only if the assertion is considered correct and the attached proof is valid.

At a later point an individual can prove possession of such a credential by proving the following two statements in zero-knowledge:

1. She knows a commitment C_i in the set $(C_1, \ldots, C_N)$ of all credentials previously accepted to the block chain.

2. She knows the opening (randomness) for the commitment.

In addition to this proof, the user may simultaneously prove additional statements about the identity and attributes contained within the commitment C_i. The challenge in the above construction is to efficiently prove statements (1) and (2), i.e., without

[3]A possibility is to use ring signatures [83], which do not require a single trusted signer. Unfortunately, these signatures grow with the number of participating signers and require expensive communication to generate.

producing a proof that scales with N. Our solution, which adapts techniques from distributed e-cash systems [82], circumvents this problem by using an efficient publicly-verifiable accumulator [69] to gather the set of all previous commitments together. Using this accumulator in combination with an efficient membership proof due to Camenisch and Lysyanskaya [84], we are able to reduce the size of this proof to $O(\lambda)$ for security parameter λ, rather than the $O(N \cdot \lambda)$ proofs that would result from a naive OR proof.

Of course, merely applying these techniques does not lead to a *practical* credential system. A key contribution of this work is to supply a concrete instantiation of the above idea under well-studied assumptions and to prove that our construction provides for consistency of credentials (ensuring multiple users cannot pool their credentials), the establishment of pseudonyms, and a long set of extensions built upon anonymous credentials. Last but not least, we need to formally define and prove the security of a distributed anonymous credential scheme and provide some model for the distributed ledger. Our instantiation requires a single trusted setup phase, after which the trusted party is no longer required.[4]

4.1.2 Outline of This Chapter

The remainder of this chapter is organized as follows. In the next section we discuss how to get a distributed bulletin board. In §4.3 we discuss specific applications

[4]In §4.7 we discuss techniques for removing this trusted setup requirement.

for decentralized anonymous credentials and argue that these systems can be used to solve a variety of problems in peer-to-peer networks. In §4.4 we define the notion of a *decentralized anonymous credential scheme* and provide an ideal-world security definition. In §4.5 we describe the cryptographic building blocks of our construction, and in §4.6 we provide an overview of our basic construction as well as a specific instantiation based on the Discrete Logarithm and Strong RSA assumptions. In §4.7 we extend our basic construction to add a variety of useful features, including k-show credentials, stateful credentials, and credentials with hidden attributes. In §4.8 we describe the implementation and performance of a prototype library realizing our credential system. Finally, in §4.9, we show how to use our library to build a distributed version of anonymous attestation.

4.2 Real-World Bulletin Boards and Decentralized Bulletin Boards

A core component of our system is an append-only bulletin board we can use to post issued credentials. The board must provide two strong security guarantees: (1) that credentials must not be tampered with once added to the board and (2) all parties will share a consistent view of the board. For the distributed instantiation we additionally require (3) no party can control the addition of credentials to the board. We detail ways to achieve both distributed and centralized versions of such a bulletin

board here.

4.2.1 Bitcoin

Bitcoin is a distributed currency system [73], which has grown since 2009 to handle between \$2–\$5 million USD/day in transaction volume in a highly adversarial environment. The heart of Bitcoin is the block chain, which serves as an append-only bulletin board maintained in a distributed fashion by the Bitcoin peers. The block chain consists of a series of blocks connected in a hash chain.[5] Every Bitcoin block memorializes a set of transactions (containing an amount of bitcoin, a sender, and a recipient) that are collected from the Bitcoin broadcast network. Thus the network maintains a consensus about what transactions have occurred and how much money each user has.

Bitcoin peers, who are free to enter and leave the network, compete to generate the next block by trying to calculate $\mathcal{H}(block \mid\mid nonce) < t$ where $\mathcal{H}$ is a secure hash function and t is an adjustable parameter. This process is known as *mining*, and the difficulty level t is adjusted so that a block is created on average every 10 minutes. When a block is generated, it is broadcast to the network and, if valid, accepted as the next entry in the block chain. Bitcoin and related systems provide two incentives to miners: (1) mining a block (i.e., completing the proof of work) entitles them to a reward[6] and (2) nodes can collect fees from every transaction in a block they mine.

[5]For efficiency reasons, the hash chain is actually a Merkle Tree.

[6]For Bitcoin this reward is set at 25 BTC but will eventually diminish and be eliminated.

While Bitcoin uses the hash chain for the specific purpose of implementing an electronic currency, the usefulness of the Bitcoin bulletin board has already been recognized by several related applications. One spinoff of the Bitcoin concept is Namecoin [79], a fork of Bitcoin that uses the block chain to maintain key–value mappings. Namecoin is currently being used to implement an experimental DNS replacement, dotBIT [85]. Users pay a small fee to register a key–value pair along with a controlling public key. They can then make updates to the pair provided (1) the updates are signed by that key and (2) if necessary, they pay a transaction fee.[7] Due to this flexibility we use the Namecoin software in our implementations, but we stress that the same techniques can be used with nearly any hash chain based network, including mature deployments such as Bitcoin.

Because of the way Bitcoin's block chain is constructed, recently added blocks maybe be removed, and, more importantly, it is possible to introduce short-term forks in the block chain that could be used to convince a single party that a poisoned-pill credential was issued and hence identify them (see §4.4.3 for more details). One solution, which is commonly used in Bitcoin, is to wait until a block has several blocks on top of it (known as confirmations) before using it. Typically, waiting six blocks, or roughly 60 minutes, is sufficient. Of course, peers are free to show credentials based off blocks sooner than that as doing so does not make the show less secure. However it comes at an increased privacy risk.

[7]Currently, neither Namecoin nor Bitcoin require significant transaction fees.

4.2.2 A Central Ledger

An alternative to using Bitcoin's block chain technology is to simply use a central service to maintain an append-only ledger. This service must be trusted to give a consistent view of the credential ledger to all parties. The most effective way to do this is with trusted hardware (e.g., TPM attestations) that ensures that (1) the list is append only and (2) the same version of the list is shown to everyone for a given time period.

For lower security systems, it may be possible to simply run a service that signs the list and have users audit the system by periodically comparing the list they received. Similar mechanisms exist for auditing SSL authorities (e.g., Google's Certificate Transparency project). Tampering would not only be readily apparent but, due to the signature on the list, provable. This, however, only acts as a deterrent to tampering as it would not be detected until the next such comparison. As such tampering can identify users when they authenticate, we only recommend this approach when either the consequences of such a privacy breach are low or loss of reputation to an authority when its malfeasance is subsequently detected is prohibitively high.

4.2.3 A Hybrid Approach

A third approach is to use some reliable storage mechanism (e.g., a central server or a robust DHT) to store credential requests and insert checkpoints into Bitcoin's

actual block chain to ensure the ledger is append only. This can be done *without* any modification to Bitcoin. We can achieve this by periodically (e.g., every 10 minutes) inserting the digest of the ledger into the Bitcoin block chain. One way to accomplish this is by using CommitCoin [74] which encodes information into the keys used for transactions without destroying funds.[8]

Our one last technical challenge is to actually mark these transactions as checkpoints for anyone to see. To accomplish this we propose leveraging multi–sig transactions[9] where one key encodes the checkpoint with CommitCoin's techniques and another is a marker address that designates a checkpoint. For a distributed storage service, this requires that the network elect a node or set of nodes to hold the marker key and insert checkpoints and elect a new set of nodes with a fresh marker if the current set either fails to insert checkpoints or inserts too many (either case is a denial of service attack and will not compromise the integrity or anonymity of credentials).

4.3 Applications

In this section we discuss several of the applications facilitated by decentralized anonymous credentials. While we believe that these credential systems may have applications in a variety of environments, we focus specifically on settings where trusting a central credential issuer is not an option or where issued credentials must

[8]The naive approach replaces the public key specifying the recipient with the hash of the data, making it impossible to retrieve the funds. CommitCoin fixes this.

[9]Transactions that require signatures from multiple parties to redeem.

be publicly audited.

Mitigating Sybil attacks in ad hoc networks. Impersonation attacks can have grave consequences for both the security and resource allocation capabilities of ad hoc networks. A variety of solutions have been proposed to address this problem. One common approach is to require that clients solve computational puzzles [86]. For example, for a challenge c and a difficulty target t, find a nonce n such that $\mathcal{H}(c||n) < t$. Solving such a puzzle takes a meaningful amount of effort — thus deterring Sybil attacks — and, as anyone can hash n and c, is publicly verifiable. For a centralized service, this proof can be done once per client on registration. In a peer-to-peer system, however, far more complex mechanisms are needed to avoid having to provide a proof of work per each pair of interacting peers [86]. We stress that the issue with distributed approaches is not the lack of publicly verifiable puzzles but the number of puzzles and who they are sent to. This is even more difficult if we require the system to be anonymous.

Our solution to this problem is to use k-show anonymous credentials. In this approach, peers establish a single credential by solving a proof of work (similar to using a central service). This allows the peer to obtain a credential that can be used a limited number of times or a limited number of times within a given time period. When a peer exceeds the k-use threshold (e.g., by cloning the credential for a Sybil attack), the credential can be identified and revoked. We note that this proposal is a distributed variant of the *anonymous subscription service* concept, which was first

explored by Damgård et al. [76] and Camenisch et al. [77].

Managing resource usage. In networks where peers both contribute and consume resources, ensuring fair resource utilization can be challenging. For example, a storage network might wish to ensure peers provide as much storage as they consume [87] or ensure that peers fairly use network bandwith [88]. This can be problematic in networks that provide anonymity services (e.g., Tor), where peers may be reluctant to identify which traffic they originated. An anonymous credential system allows peers to identify their contributions to routing traffic in exchange for a credential which they can then use to originate traffic. Of course, we are restricted to issuing credentials on metrics which peers can publicly establish. Thankfully this is a fairly expressive set. Eigenspeed [89] allows peer-to-peer networks to form accurate bandwidth estimates for all peers even in the presence of active attackers. Similarly, there exist publicly verifiable proofs of retrievability that can be used to verify storage of a file [80]. Both of these are effective metrics for resource management.

4.4 Decentralized Anonymous Credentials

A traditional anonymous credential system has two types of participants: users and organizations. Users, who each have a secret key $sk_{\mathcal{U}}$, are known by pseudonyms both to each other and organizations. Nym_A^O, for example, is the pseudonym of user A to organization O. Decentralized anonymous credentials have no single party representing

the organization. Instead, this party is replaced with a quorum of users who enforce a specific *credential issuing policy* and collaboratively maintain a list of credentials thus far issued. For consistency with prior work, we retain the term "organization" for this group.

A distributed anonymous credential system consists of a global transaction ledger, a set of transaction semantics, as well as the following (possibly probabilistic) algorithms:

- $\mathsf{Setup}(1^\lambda) \to params$. Generates the system parameters.
- $\mathsf{KeyGen}(params) \to sk_{\mathcal{U}}$. Run by a user to generate her secret key.
- $\mathsf{FormNym}(params, \mathcal{U}, E, sk_{\mathcal{U}}) \to (Nym_{\mathcal{U}}^{E}, sk_{Nym_{\mathcal{U}}^{E}})$. Run by a user to generate a pseudonym $Nym_{\mathcal{U}}^{E}$ and an authentication key $sk_{Nym_{\mathcal{U}}^{E}}$ between a user $\mathcal{U}$ and some entity (either a user or an organization) E.
- $\mathsf{MintCred}(params,\ sk_{\mathcal{U}},\ Nym_{\mathcal{U}}^{O},\ sk_{Nym_{\mathcal{U}}^{O}},\ attrs,\ aux) \to (c, sk_c, \pi_M)$. Run by a user to generate a request for a credential from organization O. The request consists of a candidate credential c containing public attributes $attrs$; the user's key $sk_{\mathcal{U}}$; auxiliary data aux justifying the granting of the credential; and a proof π_M that (1) $Nym_{\mathcal{U}}^{O}$ was issued to the same $sk_{\mathcal{U}}$ and (2) the credential embeds $attrs$.
- $\mathsf{MintVerify}(params, c, Nym_{\mathcal{U}}^{O}, aux, \pi_M) \to \{0, 1\}$. Run by nodes in the organization to validate a credential. Returns 1 if π_M is valid, 0 otherwise.

- $RegNym(Nym_{\mathcal{U}}^{O}, \mathcal{U}, O)$: $\mathcal{U}$ logs into T_P with $sk_{\mathcal{U}}$ to register a nym with organization O. If she does not have an account, she first creates one. She gives T_P a unique random string $Nym_{\mathcal{U}}^{O}$ for use as her nym with O. T_P checks that the string is indeed unique and if so stores $(Nym_{\mathcal{U}}^{O}, \mathcal{U}, O)$ and informs $\mathcal{U}$.
- $MintCred(Nym_{\mathcal{U}}^{O}, O, attrs, aux)$: $\mathcal{U}$ logs into T_P authenticating with $sk_{\mathcal{U}}$. If $Nym_{\mathcal{U}}^{O}$ is not $\mathcal{U}$'s nym with O or $sk_{\mathcal{U}}$ is wrong, reject. Otherwise, T_P checks that aux justifies issuing a credential under O's issuing policy and if so generates a unique random id id and stores $(Nym_{\mathcal{U}}^{O}, \mathcal{U}, id, attrs)$. It then adds id to its public list of issued credentials for O.
- $ShowOnNym(Nym_{\mathcal{U}}^{O}, Nym_{\mathcal{U}}^{V}, O, V, attrs, \mathbf{C})$: $\mathcal{U}$ logs into T_P with $sk_{\mathcal{U}}$. If $Nym_{\mathcal{U}}^{O}$ is not $\mathcal{U}$'s nym with O or $Nym_{\mathcal{U}}^{V}$ is not $\mathcal{U}$'s nym with V, reject. Else, T_P checks if the tuple $(Nym_{\mathcal{U}}^{O}, \mathcal{U})$ exists, if id associated with that tuple is in the set of credentials $\mathbf{C}$ that $\mathcal{U}$ provided, and if the given attributes $attrs$ match the attributes associated with that tuple. If all conditions hold, T_P informs V that $Nym_{\mathcal{U}}^{V}$ has a credential from O in the set $\mathbf{C}$. V then retrieves the set of credentials $\mathbf{C}_O$ issued by O from T_P and accepts T_P's assertion if and only if $\mathbf{C} \subseteq \mathbf{C}_O$ and O's issuing policy is valid $\forall c' \in \mathbf{C}_O$.
- $GetCredList(O)$: T_P retrieves the list of credentials for organization O and returns it.

Figure 4.1: Ideal Functionality. Security of a basic distributed anonymous credential system.

- $\mathsf{Show}(params, sk_{\mathcal{U}}, Nym_{\mathcal{U}}^{V}, sk_{Nym_{\mathcal{U}}^{V}}, c, sk_c, \mathbf{C}_O) \rightarrow \pi_S$. Run by a user to non-interactively prove that a given set of attributes are in a credential c in the set of issued credentials $\mathbf{C}_O$ and that c was issued to the same person who owns $Nym_{\mathcal{U}}^{V}$. Generates and returns a proof π_S.

- $\mathsf{ShowVerify}(params, Nym_{\mathcal{U}}^{V}, \pi_S, \mathbf{C}_O) \rightarrow \{0, 1\}$. Run by a verifier to validate a shown credential. Return 1 if π_S is valid for $Nym_{\mathcal{U}}^{V}$, 0 otherwise.

We now describe how these algorithms are used in the context of an anonymous credential system.

4.4.1 Overview of the Protocol Semantics

To realize the full anonymous credential system, we integrate the above algorithms with a decentralized hash chain based bulletin board as follows. We assume a bulletin board such as Namecoin that provides a means for storing arbitrary key–value pairs.[10] We provide a concrete realization of our protocols in §4.6 and §4.8.

Formulating a pseudonym. Prior to requesting a new credential, the user executes the KeyGen algorithm to obtain sk_U and then runs the FormNym algorithm to obtain a pseudonym for use with this organization. This requires no interaction with the bulletin board, hence the user can perform these actions offline.

Obtaining a credential. To obtain a credential, the user places the organization name and some public identity assertion — for example, a TPM attestation and AIK certificate chain — into the auxiliary data field aux, then executes the MintCred routine to obtain a credential and a signature of knowledge on that information. She then formulates a transaction including both the resulting credential and the auxiliary data and broadcasts it into the hash chain network, along with (optionally) some sum of digital currency to pay for the transaction fees. She retains the secret portion of the credential.

Once received by the network, all parties can verify the correctness of the credential and the identity assertion using the MintVerify routine and whatever external

[10] While this functionality is supported by default in Namecoin, it is also possible to store arbitrary data in existing block chains such as the Bitcoin chain.

procedures are needed to verify the auxiliary data. This process can be conducted directly by the network nodes, or it can be validated after the fact by individual credential verifiers.

Showing a credential. When a user wishes to show a credential to some Verifier, she first scans through the bulletin board to obtain a set $\mathbf{C}_O$ consisting of all candidate credentials belonging to a specific organization. She next verifies each credential using the MintVerify routine (if she has not already done so) and validates the auxiliary identity certification information. She then runs the Show algorithm to generate a credential, which she transmits directly to the Verifier. The Verifier also collects the set of credentials in $\mathbf{C}_O$ and validates the credential using the ShowVerify routine. She accepts the credential certification if this routine outputs 1.

4.4.2 Security

We define our system in terms of an ideal functionality implemented by a trusted party T_P that plays the role that our cryptographic constructions play in the real system. All communication takes place through this ideal trusted party. Security and correctness for our system comes from a proof that this ideal model is indistinguishable from the real model provided the cryptographic assumptions hold. Our ideal functionality is outlined in Figure 4.1.

It consists of organizations who issue credentials and users who both prove that they have these credentials and verify such proofs. Organizations have only two things:

1) an efficient and publicly evaluable policy, $policy_O$, for granting credentials and 2) an append-only list of credentials meeting that policy maintained by the trusted party.

4.4.3 Trusting the Ledger

An obvious question is whether the append-only transaction ledger is necessary at all. Indeed, if the list of valid credentials can be evaluated by a set of untrusted nodes, then it seems that a user (Prover) could simply maintain a credential list compiled from network broadcasts and provide this list to the Verifier during a credential show. However, this approach can enable sophisticated attacks where a malicious Verifier manipulates the Prover's view of the network to include a poisoned-pill credential that — although valid by the issuing heuristic — was not broadcast to anyone else. When the Prover authenticates, she has completely identified herself.

The distributed transaction ledgers employed by networks such as Bitcoin and Namecoin provide a solution to this problem, as their primary purpose is to ensure a shared view among a large number of nodes in an adversarial network. In practice this is accomplished by maintaining a high degree of network connectivity and employing computational *proofs of work* to compute a hash chain.

For an attacker to execute the poisoned credential attack against such a ledger, she would need to both generate and maintain a false view of the network to delude the Prover. This entails both simulating the Prover's view of the rest of the network complete with *all* its computational power and forging any assurances the Prover might

expect from known peers about the present state of the network. If the Prover has a reasonable estimate of the actual network's power (e.g., she assumes it monotonically increases), then an attacker must actually have equivalent computational power to the entirety of the network to mount such an attack. For the purposes of this paper we assume such active attacks are impossible even if the attacker controls a simple majority of the computational power. Attackers are still free to attempt any and all methods of retroactively identifying a user and mount any other active attacks.

4.5 Preliminaries

We make use of the following complexity assumptions and cryptographic building blocks to construct our scheme.

4.5.1 Complexity Assumptions

The security of our scheme relies on the following two complexity assumptions:

Strong RSA Assumption [90, 91]. Given a randomly generated RSA modulus n and a random element $y \in \mathbb{Z}_n^*$, it is hard to compute $x \in \mathbb{Z}_n^*$ and integer exponent $e > 1$ such that $x^e \equiv y \bmod n$. We can restrict the RSA modulus to those of the form pq, where $p = 2p' + 1$ and $q = 2q' + 1$ are safe primes.

Discrete Logarithm (DL) Assumption [92]. Let G be a cyclic group with generator g. Given $h \in G$, it is hard to compute x such that $h = g^x$.

4.5.2 Cryptographic Building Blocks

Zero-knowledge proofs. In a zero-knowledge protocol [93] a user (the prover) proves a statement to another party (the verifier) without revealing anything about the statement other than that it is true. Our constructions use zero-knowledge proofs that can be instantiated using the technique of Schnorr [94], with extensions due to, e.g., [95, 96, 97, 98]. We convert these into *non-interactive* proofs by applying the Fiat-Shamir heuristic [99]. When we use these proofs to authenticate auxiliary data, we refer to the resulting non-interactive proofs as *signatures of knowledge* as defined in [100].

When referring to these proofs we will use the notation of Camenisch and Stadler [101]. For instance, $\mathsf{NIZKPoK}\{(x, y) : h = g^x \ \wedge \ c = g^y\}$ denotes a non-interactive zero-knowledge proof of knowledge of the elements x and y that satisfy both $h = g^x$ and $c = g^y$. All values not enclosed in ()'s are assumed to be known to the verifier. Similarly, the extension $\mathsf{ZKSoK}[m]\{(x, y) : h = g^x \ \wedge \ c = g^y\}$ indicates a *signature of knowledge* on message m.

Accumulators [82]. An accumulator allows us to combine many values into one smaller value (the accumulator). We then have a single element, called the witness, that allows us to attest to the fact that a given value is actually part of the accumulator. Our constructions use an accumulator based on the Strong RSA assumption. The accumulator we use was first proposed by Benaloh and de Mare [102] and later improved by Baric and Pfitzmann [90] and Camenisch and Lysyanskaya [69]. We

describe the accumulator using the following algorithms:

- $\mathsf{AccumSetup}(\lambda) \rightarrow params$. On input a security parameter, sample primes p, q (with polynomial dependence on the security parameter), compute $N = pq$, and sample a seed value $u \in QR_N, u \neq 1$. Output (N, u) as $params$.

- $\mathsf{Accumulate}(params, \mathbf{C}) \rightarrow A$. On input $params$ (N, u) and a set of prime numbers $\mathbf{C} = \{c_1, \ldots, c_i \mid c \in [\mathcal{A}, \mathcal{B}]\}$,[11] compute the accumulator A as $u^{c_1 c_2 \cdots c_n} \bmod N$.

- $\mathsf{GenWitness}(params, v, \mathbf{C}) \rightarrow \omega$. On input $params$ (N, u), a set of prime numbers $\mathbf{C}$ as described above, and a value $v \in \mathbf{C}$, the witness ω is the accumulation of all the values in $\mathbf{C}$ besides v, i.e., $\omega = \mathsf{Accumulate}(params, \mathbf{C} \setminus \{v\})$.

- $\mathsf{AccVerify}(params, A, v, \omega) \rightarrow \{0, 1\}$. On input $params$ (N, u), an element v, and witness ω, compute $A' \equiv \omega^v \bmod N$ and output 1 if and only if $A' = A$, v is prime, and $v \in [\mathcal{A}, \mathcal{B}]$ as defined previously.

For simplicity, the description above uses the full calculation of A. Camenisch and Lysyanskaya [69] observe that the accumulator may also be *incrementally* updated, i.e., given an existing accumulator A_n it is possible to add an element x and produce a new accumulator value A_{n+1} by computing $A_{n+1} = A_n^x \bmod N$.[12]

[11] "Where $\mathcal{A}$ and $\mathcal{B}$ can be chosen with arbitrary polynomial dependence on the security parameter, as long as $2 < \mathcal{A}$ and $\mathcal{B} < \mathcal{A}^2$." [84] For a full description, see [84, §3.2 and §3.3].

[12] This allows the network to maintain a running value of the accumulator and prevents individual nodes from having to recompute it [82].

Camenisch and Lysyanskaya [69] show that the accumulator satisfies a strong *collision-resistance* property if the Strong RSA assumption is hard. Informally, this ensures that no *p.p.t.* adversary can produce a pair (v, ω) such that $v \notin \mathbf{C}$ and yet $\mathsf{AccVerify}$ is satisfied. Additionally, they describe an efficient zero-knowledge proof of knowledge that a committed value is in an accumulator. We convert this into a non-interactive proof using the Fiat-Shamir transform and refer to the resulting proof using the following notation:

$$\mathsf{NIZKPoK}\{(v, \omega) : \mathsf{AccVerify}((N, u), A, v, \omega) = 1\}.$$

Verifiable Random Functions. A pseudorandom function (PRF) [103] is an efficiently computable function whose output cannot be distinguished (with non-negligible advantage) from random by a computationally bounded adversary. We denote the pseudorandom function as $f_k(\cdot)$, where k is a randomly chosen key. A number of PRFs possess efficient proofs that a value is the output of a PRF on a set of related public parameters. Two examples of this are the Dodis-Yampolskiy (DY) PRF [104] and the Naor-Reingold PRF [105].

Pedersen Commitments. A commitment scheme allows a user to bind herself to a chosen value without revealing that value to the recipient of the commitment. This commitment to the value ensures that the user cannot change her choice (i.e., *binding*), while simultaneously ensuring that the recipient of the commitment does not learn

anything about the value it contains (i.e., *hiding*) [106]. In Pedersen commitments [107], the public parameters are a group G of prime order q, and generators $(g_0, \ldots, g_m)$. In order to commit to the values $(v_1, \ldots, v_m) \in \mathbb{Z}_q^m$, pick a random $r \in \mathbb{Z}_q$ and set $C = \text{PedCom}(v_1, \ldots, v_m; r) = g_0^r \prod_{i=1}^m g_i^{v_i}$.

4.6 A Concrete Instantiation

We now provide a concrete instantiation of our construction and prove the security of our construction under the Discrete Logarithm and Strong RSA assumptions.

4.6.1 Overview of the Construction

Alice's pseudonym with a given organization/user is an arbitrary identity that she claims in a transaction. She tags this value with a Pedersen commitment to her secret key *sk* and *signs* the resulting transaction using a signature of knowledge that she knows the secret key. There is no separate process for registering a pseudonym: instead they are simply used in issue and show to allow operations to be linked if necessary. Alice's credential c is a vector Pedersen commitment to both *sk* and a set of public attributes $attrs = a_0, \ldots, a_m$, which Alice also includes in her credential. To issue a credential, Alice provides the network with a credential, a pseudonym, her attributes, optionally some auxiliary data justifying the credential issue (e.g., a proof of work that Alice is not a Sybil), and a proof that (1) the commitment and the pseudonym

contain the same secret key and (2) the attributes are in some allowed set. If all of this validates, the entry is added to the ledger. Alice shows the credential under a different pseudonym by proving in zero-knowledge that (1) she knows a credential on the ledger from the organization, (2) the credential opens to the same sk as her pseudonym, and (3) it has some attributes.

4.6.2 The Construction

The full construction is provided in Figure 4.2. We use Pedersen commitments and a Strong RSA based accumulator to instantiate the core of the protocol. The proofs of knowledge in the Show algorithm are conducted using Schnorr-style proofs modified using the Fiat-Shamir heuristic as in previous work [94, 69]. The implementation of the proofs are similar to those used by Miers et al. in [82].

Theorem 4.6.1. The basic distributed anonymous credential system described in Figure 4.2 is secure in the random oracle model under the Strong RSA and the Discrete Logarithm assumptions.

We provide a sketch of the proof of Theorem 4.6.1 in Appendix E.

[13] "Where $\mathcal{A}$ and $\mathcal{B}$ can be chosen with arbitrary polynomial dependence on the security parameter, as long as $2 < \mathcal{A}$ and $\mathcal{B} < \mathcal{A}^2$." [84]. For a full description, see [84, §3.2 and §3.3].

- $\mathsf{Setup}(1^\lambda) \rightarrow params$. On input a security parameter λ, run $\mathsf{AccumSetup}(1^\lambda)$ to obtain the values (N, u). Next generate primes p, q such that $p = 2^w q + 1$ for $w \geq 1$. Let $\mathbb{G}$ be an order-q subgroup of $\mathbb{Z}_p^\star$, and select random generators $g_0, \ldots, g_n$ such that $\mathbb{G} = \langle g_0 \rangle = \cdots = \langle g_n \rangle$. Output $params = (N, u, p, q, g_0, \ldots, g_n)$.
- $\mathsf{KeyGen}(params) \rightarrow sk$. On input a set of parameters $params$, select and output a random master secret $sk \in \mathbb{Z}_q$.
- $\mathsf{FormNym}(params, sk) \rightarrow (Nym, sk_{Nym})$. Given a user's master secret sk, select a random $r \in \mathbb{Z}_q$ and compute $Nym = g_0^r g_1^{sk}$. Set $sk_{Nym} = r$ and output (Nym, sk_{Nym}).
- $\mathsf{MintCred}(params, sk, Nym_{\mathcal{U}}^O, sk_{Nym_{\mathcal{U}}^O}, attrs, aux) \rightarrow (c, sk_c, \pi_M)$. Given a nym $Nym_{\mathcal{U}}^O$ and its secret key $sk_{Nym_{\mathcal{U}}^O}$; attributes $attrs = (a_0, \ldots, a_m) \in \mathbb{Z}_q$; and auxiliary data aux, select a random $r' \in \mathbb{Z}_q$ and compute $c = g_0^{r'} g_1^{sk} \prod_{i=0}^{m} g_{i+2}^{a_i}$ such that $\{c \text{ prime} \mid c \in [\mathcal{A}, \mathcal{B}]\}$.[13] Set $sk_c = r'$ and output (c, sk_c, π_M) where π_M is a signature of knowledge on aux that the nym and the credential both belong to the same master secret sk, i.e.:

$$\pi_M = \mathsf{ZKSoK}[aux]\{(sk, r', r) : c = g_0^{r'} g_1^{sk} \prod_{i=0}^{m} g_{i+2}^{a_i} \ \wedge \ Nym_{\mathcal{U}}^O = g_0^r g_1^{sk}\}$$

  Finally, submit the resulting values $(c, \pi_M, attrs, Nym_{\mathcal{U}}^O, aux)$ to the public transaction ledger.
- $\mathsf{MintVerify}(params, c, attrs, Nym_{\mathcal{U}}^O, aux, \pi_M) \rightarrow \{0, 1\}$. Given a credential c, attributes $attrs$, a nym $Nym_{\mathcal{U}}^O$, and proof π_M, verify that π_M is the signature of knowledge on aux. If the proof verifies successfully, output 1, otherwise output 0. The organization nodes should accept the credential to the ledger if and only if this algorithm returns 1.
- $\mathsf{Show}(params, sk, Nym_{\mathcal{U}}^V, sk_{Nym_{\mathcal{U}}^V}, c, attrs, sk_c, \mathbf{C}_O) \rightarrow \pi_S$. Given a user's master secret sk; a nym $Nym_{\mathcal{U}}^V$ between the user and the verifier and its secret key $sk_{Nym_{\mathcal{U}}^V}$; a credential c and its secret key sk_c; the attributes $(a_0, \ldots, a_m)$ used in the credential; and a set of credentials $\mathbf{C}$, compute $A = \mathsf{Accumulate}(params, \mathbf{C}_O)$ and $\omega = \mathsf{GenWitness}(params, c, \mathbf{C}_O)$ and output the following proof of knowledge:

$$\pi_S = \mathsf{NIZKPoK}\{(sk, \omega, r', c, r, Nym_{\mathcal{U}}^V) : \mathsf{AccVerify}(params, A, c, \omega) = 1 \ \wedge \ c = g_0^{r'} g_1^{sk} \prod_{i=0}^{m} g_{i+2}^{a_i} \ \wedge \ Nym_{\mathcal{U}}^V = g_0^r g_1^{sk}\}$$

- $\mathsf{ShowVerify}(params, Nym_{\mathcal{U}}^V, \pi_S, \mathbf{C}_O) \rightarrow \{0, 1\}$. Given a nym $Nym_{\mathcal{U}}^V$, proof of possession of a credential π_S, and the set of credentials issued by organization O $\mathbf{C}_O$, first compute $A = \mathsf{Accumulate}(params, \mathbf{C}_O)$. Then verify that π_S is the aforementioned proof of knowledge on c, $\mathbf{C}_O$, and $Nym_{\mathcal{U}}^V$ using the known public values. If the proof verifies successfully, output 1, otherwise output 0.

Figure 4.2: Our basic decentralized anonymous credential scheme.

4.7 Extensions

We consider extending the basic system in several ways.

4.7.1 k-show Credentials

Damgård et al. [76] first suggested a credential system where users could only authenticate once per time period. Camenisch et al. [77] independently proposed a significantly more efficient construction that allows for up to k authentications per time period, with the ability to revoke all cloned credentials if a credential was used beyond this limit. Camenisch et al. suggested that these techniques might be used to build anonymous subscription services, allowing users to access a resource (such as a website) within reasonable bounds. We briefly show that these same techniques can be applied to our basic credential system.

In the system of [77] an authority issues a credential on a user's secret seed s. To show a credential for the i^{th} time in validity period t, the user generates a serial number S using a verifiable random function (VRF) as $S = f_s(0||t||i)$. She also includes a non-interactive zero-knowledge proof that this serial number is correctly structured.[14] This technique can be applied to our construction provided we can securely store a seed for the VRF. This is easy: the user simply generates a random seed s and includes this value in the commitment she stores in the transaction ledger. We note

[14]The re-use of a credential would result in a repeated serial number, and yet the nature of the VRF's output (for an honest user) ensures that attackers cannot link individual shows.

that for the trivial case of one-time show credentials, we can simply reveal the seed. For k-show, the user provably evaluates the VRF on the seed plus a secret counter.[15]

4.7.2 Credentials with Hidden Attributes

In our basic construction of §4.6, users provide a full list of attributes when requesting and showing credentials. While this is sufficient for many applications, there exist cases where a user might wish to conceal the attributes requested or shown, opting instead to prove statements about them, e.g., proving knowledge of a secret key or proving that an attribute is within a certain range. There are two simple ways to do this. First, we can simply use multi-message commitments where each message is an attribute. This increases the size of our zero-knowledge proofs (they are linear in the number of messages in a commitment) but does not change our schemes. A more efficient construction is to encode the attributes in one single value and then prove statements about that committed value rather than reveal it. For example, one could prove that a given bit corresponding to a certain attribute was set. One could also use the first x bits for attribute one, the next x bits for attribute two, etc. and use range proofs [108, 109, 110, 111] to reveal only those attributes we want to display.

[15] Camenisch et al. [77] describe a further extension that reveals the user's identity in the event of a credential double-show. We omit the details here for space reasons but observe that the same technique can be applied to our construction.

4.7.3 Stateful Credentials

A stateful anonymous credential system [106] is a variant of an anonymous credential system where credential attributes encode some state that can be updated by issuing new credentials. This credential issuance is typically conditioned on the user showing a previous credential and offering proof that the new credential should be updated as a function of the original.

Intuitively, we can already have this capability quite easily due to the fact that our credentials are non-interactively issued. We can make stateful credentials simply by changing the policy by which we issue credentials: to issue a credential in a new state s_1, we require a user to demonstrate that they had a credential in state s_0 and discard it by revealing its single use serial number.

We construct a "single show" credential c embedding some state *state* in the attributes and a serial number S. Users are free to show c as many times as they like without revealing the serial number. However, to update the state of the credential, they must author a transaction that shows the original credential and reveals the serial number S and "mint" a new candidate credential c' containing the updated state $state'$ (hidden inside of a commitment) and a proof that there exists a valid relationship between the state encoded in c and the new state in c' (for example, that the attributes have been incremented).

This requires only minor extensions to our basic scheme composing the existing secure functionality. In this case we add an Update algorithm that operates similarly

- $\mathsf{Update}(params, sk, c, sk_c, \mathbf{C}_O, update\_relation, state') \rightarrow (c', sk'_c, \pi_u)$. Given a credential c and associated secret key sk_c, a set of credentials $\mathbf{C}_O$, an updated state $state' = (s'_0, \ldots, s'_m) \in \mathbb{Z}_q$, and an update relation $update\_relation$, generate a fresh random serial number $S' \in \mathbb{Z}_q$ and random value $r' \in \mathbb{Z}_q$ to form a new credential $c' = g_0^{r'} g_1^{sk} g_2^{S'} \prod_{i=0}^{m} g_{i+3}^{s'_i}$ with the aforementioned restrictions. Compute $A = \mathsf{Accumulate}(params, \mathbf{C}_O)$ and $\omega = \mathsf{GenWitness}(params, c, \mathbf{C}_O)$. Output (c', sk'_c, π_u) where $sk'_c = (S', state', r')$ and

$$\begin{aligned}\pi_u = \mathsf{NIZKPoK}\{&(sk, \omega, c, state, r, c', S', state', r') : \\ &\mathsf{AccVerify}(params, A, c, \omega) = 1 \\ &\wedge c = g_0^r g_1^{sk} g_2^S \prod_{i=0}^{m} g_{i+3}^{s_i} \ \wedge \ c' = g_0^{r'} g_1^{sk} g_2^{S'} \prod_{i=0}^{m} g_{i+3}^{s'_i} \\ &\wedge \ update\_relation(state, state') = 1\}\end{aligned}$$

- $\mathsf{UpdateVerify}(params, c, \mathbf{C}_O, \pi_u) \rightarrow \{0, 1\}$. Given a stateful credential c, a credential set $\mathbf{C}_O$, and proof π_u, output 1 if π_u is correct, the proved state transition is a legal one, and the serial number S was not previously used. Otherwise 0.

Figure 4.3: Extensions for a stateful anonymous credential system. $update\_relation(\ldots) = 1$ denotes that the update encodes some arbitrary state transition (e.g. $\forall i \ s'_i = s_i + 1$).

to MintCred but includes the earlier credential and a proof of its construction. A valid proof of the existing credential now becomes a condition for the organization accepting the updated credential into the ledger. We provide a description of this new algorithm in Figure 4.3.

4.8 Integrating with Proof-of-work Bulletin Boards

We provide a basic implementation of our credential scheme as a library and construct a basic example using Namecoin as the bulletin board. Our prototype system allows users to prove they have a (fresh) commitment to some attributes in an issued credential. For our purposes it is sufficient to merely reveal the content of that commitment (the attributes) in its entirety during a show. However, selectively disclosable attributes are trivially realizable, see §4.7.2.

4.8.1 Integration

Namecoin integration is straightforward. Namecoin provides a built in mechanism for storing key–value pairs which, by convention, have a namespace as a prefix. It also provides a basic functionality to scan the list of existing names. Thus we can scan for credentials, validate them, and then accumulate them. It is then simply matter of generating and verifying proofs against that computed accumulator value.

For Alice to obtain a credential, she:

1. Pays a very small fee (currently 0.0064 USD) to purchase some name in the system's namespace by registering a public key as the owner of the name. This corresponds to a transaction looking like:

```
1 665a...  OP_2DROP
OP_DUP  OP_HASH160 6c1abe34
OP_EQUALVERIFY  OP_CHECKSIG
```

2. Prepares a fresh credential with some attributes and any supporting documentation necessary for her identity claim and stores the private portion of the credential.

3. Updates, using the public key from step 1, her registered name to contain a credential and its supporting documentation.

```
2 642f7...  7b...
OP_2DROP OP_2DROP
OP_DUP OP_HASH160
14d...  OP_EQUALVERIFY OP_CHECKSIG
```

Once this update is confirmed, Alice has a fully formed credential.

To show the credential to Bob, Alice:

1. Scans through the list of added names and retrieves all candidate credentials.

2. Checks the supporting documentation for each candidate and puts valid ones in $\mathbf{C}$.

3. Runs Show with the public parameters, the private portion of her credentials, and $\mathbf{C}$ and sends the result to Bob.

4. Bob does steps 1 and 2 and computes $\mathbf{C}$ himself.

5. Bob runs ShowVerify on Alice's supplied credential and $\mathbf{C}$ to verify it.

Alice has now proved she has a credential to Bob.

What the supporting documentation is and how it is verified is an application specific problem. For some applications, merely having paid the tiny registration fee may be sufficient and no verification is necessary. For others, some digital signature may need to be verified or some assertion about resource management (e.g., a proof of storage/retrievability) may need to be verified. Without modifications to Namecoin/Bitcoin, any assertion must be verifiable by all participants.[16] We consider one such application in the next section.

4.8.2 Operating Cost

Namecoin is not free to use as purchasing a name costs a small (less than 0.10 USD as of 12/1/2013) amount of money. This fee is necessary both to prevent mass name hoarding and to provide an economy to pay the miners who maintain the block chain. This cost must minimally be paid by users when creating a credential. For certain applications (e.g., k-anonymous credentials), relying parties must also post data on the block chain (e.g., double spend tags and serial numbers). This, again, costs a small fee. As such, there are monetary costs to using such an identity scheme.

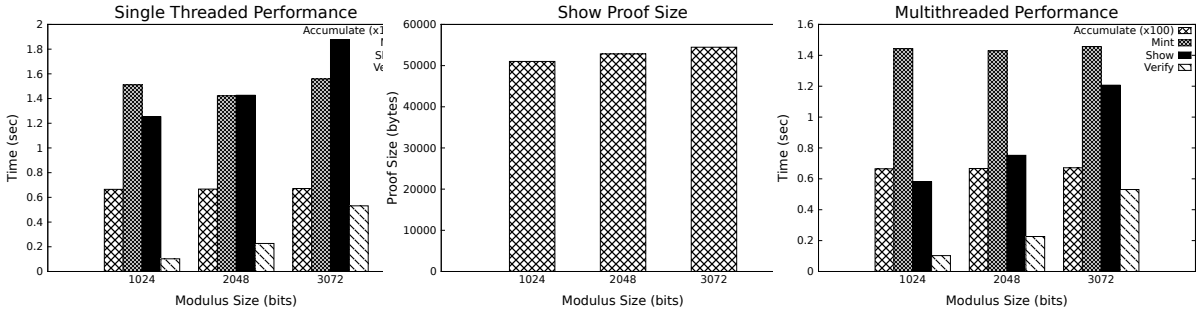
[16] With modifications, identity assertions can be validated as part of the consensus protocol, abrogating relying parties from validating credential issue and allowing the use of ephemeral supporting documentation.

4.8.3 Latency

A third consideration for the limited show credentials is the latency of inserting items into the block chain. Because completely meaningful proofs of work take time, some time must elapse in any such system. Namecoin and Bitcoin both aim to create blocks every 10 minutes. Thus, the naive wait time from a block is about 5 minutes. Propagation delays in the network and transaction volume, however, skew this distribution. While historical data for Namecoin is not available, for Bitcoin it takes slightly less than 9 minutes for a transaction to first be confirmed. In practice, it then takes multiple confirmations to solidify the transaction's place in the block chain. Variants of Bitcoin operate with faster confirmation times (e.g., Feathercoin, which aims to get a block every 2.5 minutes), though it is not yet clear if the more aggressive of these are entirely stable.

Given these latency constraints, our system, at least built on top of proof of work based bulletin boards, is not suitable for applications that require fast credential issue or quick detection of multi-spends across mutually distrusting parties.[17] A side effect of this is that double spend prevention mechanisms for fast transactions need to rely on detection and punishment (e.g., forfeiture of an escrowed value), not prevention.

[17] Obviously, parties could cooperate and maintain a faster store of double spend tags, alleviating this problem.



(a) Times for operations measured in seconds. These operations do not include the time required to compute the accumulator.

(b) Library show proof sizes measured in bytes as a function of RSA modulus size.

(c) Multithreaded library performance as a function of parameter size.

Figure 4.4: Library performance as a function of parameter size.

4.8.4 Performance

We now examine the performance of our anonymous credential system. There are four underlying operations: minting a credential, verifying that the mint is correct, showing a credential, and verifying that show. Showing and verifying credentials also entail computing the accumulation of all or all but one of the current credentials. However, both the accumulator and the witnesses can be computed incrementally as credentials are added: for each added credential, the nodes must update both the accumulator and the witness for each credential they intend to show. Because this cost is both amortized for any individual credential show or verify, it does not come into play. Hence, we measure the accumulation cost separately and run our other benchmarks with a precomputed witness and accumulator. We also give measurements for our performance with different security parameters. See Figure 4.4.

All experiments were conducted on a 2010 MacPro with 16GB of RAM and two 2.4GHz quad core Xeon E5620 processors running OSX 10.8.3. Experiments were measured in seconds via wall clock run time and were repeated for 500 iterations. Because of the speed of accumulating elements, we measure accumulator time in seconds per 100 accumulations.

The primary performance bottleneck for our library is the complexity of the proof of knowledge generated during the credential show. Because this double discrete logarithm proof uses cut-and-choose techniques, we need to perform between 80 and 128 iterations depending on the security parameter. This entails approximately 800-1000 exponentiations. Luckily, the same cryptographic requirements that force these iterations also mandate that such computations are independent and hence they can easily be parallelized. To exploit this, we make use of OpenMP to parallelize proof generation and verification. As shown in Figure 4.4c, this offers significant performance benefits.

Unfortunately, OpenSSL, which we use for the computations underpinning our system, is not fully parallelizable due to the fact that its PRNG is synchronous. The resulting locks around RNG usage prevent us from effectively parallelizing portions of our code for showing a credential. It also causes problems when minting a credential. The resource intensive portion of credential mint is creating commitments and then testing if they are prime. This requires random numbers both directly for commitment generation and indirectly for primality testing which uses randomized Miller-Rabin.

We believe further performance gains could be realized by using a parallelizable RNG (e.g., Intel's RDRand instruction).

4.9 Example Application: Distributed Direct Anonymous Attestation (dDAA)

In the original TPM 1.1b specification [112], attestations are signed by a TPM's Attestation Identity Key (AIK). Each TPM device can generate many AIKs, and prior to use each AIK public key is signed by a trusted third party called a Privacy CA, creating an AIK certification. The purpose of this awkward mechanism is to provide a layer of indirection between attestations and the manufacturer-specified keys programmed into the device, such as the permanent Endorsement Key (EK). By introducing a third party, it becomes possible to create many identities and thus remove the risk that a given device could be linked to all of its endorsements.

Direct Anonymous Attestation (DAA) [71], replaced the privacy CA with a cryptographically sound group signature scheme. Instead of signing attestations with an AIK, a TPM signs attestations with a private key for a group signature scheme that preserves the signer's anonymity. The TPM obtains the group signing key from a DAA authority by authenticating non-anonymously to the authority with the AIK. Because the group signature key is used anonymously, the authority can never link its use to the AIK key that caused its issue. Unfortunately, the integrity of this process

depends fundamentally on the integrity of the software running in the DAA authority. This makes deploying a DAA instance somewhat problematic: each organization is responsible for deploying and securing this DAA authority, and any compromise of this server opens the door for credential theft or denial of service. Given the critical role envisioned for TPM attestations, this may inhibit the deployment of DAA systems.

We propose a new TPM design that retains the privacy advantages of DAA without the need to run a separate DAA server for each deployment. The advantage of our approach is that organizations may still maintain separate trust environments for private assertions by TPM devices but without the need to run a vulnerable separate server. Our solution requires one modification to the existing TPM infrastructure, namely that the TPM be updated to include a (non-anonymous) signing key, with a permission level similar to that of the Endorsement Key (EK). We will refer to this key as the Endorsement Signing Key, or ESK, and assume that each new TPM will ship with an Endorsement Signing Key Certificate signed by the manufacturer.[18] Given this modification we show how to use our anonymous credential scheme as a replacement for DAA.

To obtain a credential in the new scheme, the TPM runs the MintCred routine, securely store the resulting sk and transmitting the resulting credential up to the block chain along with a signature under the TPM's Endorsement Signing Key (ESK).[19]

[18]The TPM Endorsement Key and Endorsement Key Certificate would be sufficient for this role. However this key is limited by the specification to performing a decryption-only role, largely for privacy reasons. Our sole modification is to allow signing with this key, or a second key of a similar nature.

[19]In principle this ESK signature can be replaced with an AIK signature with no changes to the

This signature authenticates the credential as having been generated by a valid TPM. Once the credential and signature are validated, they can be accumulated by verifiers. The TPM can later attest to a particular configuration by running a modified version of Show that ensures π_S is a signature of knowledge on the attestation values (i.e., the program configuration registers (PCRs) and an optional nonce). Running ShowVerify with the appropriate modifications for checking the signature of knowledge validates the attestation. We stress that even though the TPM's ESK (or AIK) is on the ledger with the issued credential, showing a credential never identifies which issued credential was used and hence does not identify the ESK (or AIK).

4.10 Related Work

Anonymous credentials. Introduced by Chaum [65] and developed in a line of subsequent works (e.g., [66, 67, 68]), anonymous credentials allow a user to prove that she has a credential issued by some organization, without revealing anything about herself other than that she has the credential. Under standard security definitions, even if the verifier and credential issuer collude, they cannot determine when the credential was issued, who it was issued to, or when it was or will be used. A common construction involves issuing a credential by obtaining a signature from an organization on a committed value (e.g., using the signature scheme of [69]) then

TPM mechanism, but AIKs will have to be issued by a trusted third party.

proving in zero-knowledge that one has a signature under the organization's public key on that value. The contents of the commitment may be revealed outright or various properties can proved on the committed values (e.g., Alice can prove she is over 21 years old). Extensions to this work describe credentials that can only be shown anonymously a limited number of times [77] or delegated to others [70]. All of these schemes require issuing organizations to maintain a secret key.

Bitcoin and append-only ledgers. Our construction relies on the existence of a *distributed append-only transaction ledger*, a technology that makes up the core component of the Bitcoin distributed currency: the log of all currency transactions called the block chain [73]. These ledgers are maintained by an ad hoc group of network nodes who are free to enter and leave the network (there is no key provisioning necessary for them to join). A typical transaction ledger consists of a sequence of blocks of data that are widely replicated among the participating nodes, with each block connected to the previous block using a hash chain. Nodes compete for the opportunity to add new blocks of transactions to the ledger by producing a partial hash collision over the new data and the hash of the last block in the chain. The hash collision serves two purposes: first, it is a computationally-difficult-to-forge authenticator of the ledger and second, since finding a partial hash collision involves substantial computational effort, the peer who finds it is chosen "at random" with a probability proportional to the rate at which he can compute such partial collisions. As a result, an ad hoc group of mutually distrusting and potentially dishonest peers can correctly manage

such a ledger provided that a majority of their computational power is held by honest parties. Recent experience with Bitcoin and Namecoin provides evidence that this assumption holds in practice.

Namecoin. Namecoin [79] is a decentralized identity system that uses the same block chain technology as Bitcoin. Namecoin's primary purpose is to associate names with arbitrary data. A user can claim a name provided (1) they pay the price in NMC for it and (2) it is unclaimed. At that point, an entry is inserted into the block chain mapping the name to a public key and some arbitrary data. The public key allows the owner to update the data by signing a new record. The data allows for various uses. If it is an IP address, then one has a distributed DNS system (such a system, .bit, is already deployed). On the other hand, if it is a public key, the result is a basic PKI. The first-come first-served nature of Namecoin seems somewhat anachronistic, however it replicates in miniature the way normal DNS names are generally assigned, where the first person to claim the name gets it. Similarly, standard (non–extended validation) SSL certificates for a domain are typically issued to anyone who can demonstrate control of a domain (usually via an email to admin@domain).

4.11 Conclusion

In this work we constructed a distributed anonymous credential system and several extensions. Our constructions are secure in the random oracle model under standard

cryptographic assumptions provided there exists a trustworthy global append-only ledger. To realize such a ledger we propose using the block chain system already in real world use with the distributed cryptographic currency Bitcoin. Although we are limited in the class of identity assertions we can certify, we argue that several basic assertions are of particular use in peer-to-peer systems, as they can be used to mitigate Sybil attacks, ensure fair resource usage, and protect users' anonymity while verifying their computer's correctness.

Future work. We leave two open problems for future work. First, the proofs in this work assumed the security of a transaction ledger. We leave a precise formal model of the ledger, which attacks are allowable, and what bounds may be placed on their consequence as an open problem. Second, the efficiency of our construction can be improved. Although all of our algorithms are efficient (in that they do not scale with the size of the ledger), the need for double-discrete logarithm proofs leads to somewhat large proof sizes when showing a credential (roughly 50KB for modest parameters). Our construction may be optimized for certain applications that do not require the full flexibility of our construction. For example, schemes not requiring selective disclosure of credentials require about half that proof size. At the same time, we hope that advances in bilinear accumulators, mercurial commitments, or lattice based techniques may provide a more efficient construction. We are particularly hopeful that generic work in verifiable computation [113, 114] will offer drastically smaller proof sizes without resorting to bespoke proofs and protocols.

4.12 Acknowledgments

This work was supported by the U.S. Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contract FA8750-11-2-0211.

The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

Chapter 5

Conclusion

Deploying secure cryptographic protocols has many challenges, and this thesis discussed various ways to increase the security and deployability of real-world cryptographic systems. We first discussed the benefit of analyzing already deployed cryptographic protocols by demonstrating weaknesses in the popular RC4 algorithm and showing how to decrypt user passwords encrypted with RC4 in TLS. We then explored how to make it easier to develop and deploy secure and efficient cryptographic systems and take theoretical work to practice by discussing cryptographic automation and AutoGroup+. We concluded by showing how to build secure real-world systems as well as work to remove barriers of deployment by detailing our work on building decentralized anonymous credentials using a blockchain. While these works are of course not a complete solution to the problem of securing deployed cryptographic systems, they advance the understanding of the problem and propose various different

avenues for solutions.

Bibliography

[1] "Privacy Rights Clearinghouse: Data Breaches," https://www.privacyrights.org/data-breaches.

[2] S. Morgan, "Cyber Crime Costs Projected To Reach $2 Trillion by 2019," http://www.forbes.com/sites/stevemorgan/2016/01/17/cyber-crime-costs-projected-to-reach-2-trillion-by-2019, January 17, 2016.

[3] S. Gallagher, "New JavaScript hacking tool can intercept PayPal, other secure sessions," http://arstechnica.com/business/2011/09/new-javascript-hacking-tool-can-intercept-paypal-other-secure-sessions/, September 21, 2011.

[4] "The Heartbleed Bug," http://heartbleed.com/, April 29, 2014.

[5] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect forward secrecy: How

Diffie-Hellman fails in practice," in *22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[6] D. Goodin, "Once seen as bulletproof, 11 million+ ashley madison passwords already cracked," http://arstechnica.com/security/2015/09/once-seen-as-bulletproof-11-million-ashley-madison-passwords-already-cracked/, September 10, 2015.

[7] F. Y. Rashid, "Google, mozilla, microsoft browsers will dump rc4 encryption," Available at http://www.infoworld.com/article/2979527/security/google-mozilla-microsoft-browsers-dump-rc4-encryption.html, 2015.

[8] A. Popov, "Prohibiting RC4 Ciphersuites," RFC 7465, Internet Engineering Task Force, February 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7465.txt

[9] "The Coq Proof Assistant," Available at https://coq.inria.fr/.

[10] "EasyCrypt: Computer-Aided Cryptographic Proofs," Available at https://www.easycrypt.info/trac/.

[11] J. A. Akinyele, M. Green, S. Hohenberger, and M. W. Pagano, "Machine-generated algorithms, proofs and software for the batch verification of digital signature schemes," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 474–487.

[12] J. A. Akinyele, M. Green, and S. Hohenberger, "Using smt solvers to automate

design tasks for encryption and signature schemes," in *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security.* ACM, 2013, pp. 399–410.

[13] D. Boneh and M. K. Franklin, "Identity-based encryption from the Weil pairing," in *CRYPTO*, 2001, pp. 213–229.

[14] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proceedings of the 13th ACM conference on Computer and communications security.* Acm, 2006, pp. 89–98.

[15] S. Krishnamurthy, "HP to Acquire Voltage Security to Expand Data Encryption Security Solutions for Cloud and Big Data," http://www.voltage.com/blog/releases, February 9, 2015.

[16] J. A. Akinyele, C. Garman, and S. Hohenberger, "Automating fast and secure translations from type-i to type-iii pairing schemes," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2015, pp. 1370–1381.

[17] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous distributed e-cash from bitcoin," in *Security and Privacy (SP), 2013 IEEE Symposium on.* IEEE, 2013, pp. 397–411.

[18] C. Garman, M. Green, and I. Miers, "Decentralized anonymous credentials," in

Network and Distributed System Security Symposium (NDSS), 2013. ISOC, 2013.

[19] C. Garman, K. G. Paterson, and T. V. der Merwe, "Attacks only get better: Password recovery attacks against rc4 in tls," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 113–128. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/garman

[20] C. Garman, K. G. Paterson, and T. van der Merwe, "Attacks only get better: Password recovery attacks against RC4 in TLS," full version of this paper. Available from http://www.isg.rhul.ac.uk/tls/RC4mustdie.html.

[21] N. AlFardan and K. G. Paterson, "Lucky 13: Breaking the TLS and DTLS record protocols," in *IEEE Symposium on Security and Privacy*, 2013. [Online]. Available: http://www.isg.rhul.ac.uk/tls/Lucky13.html

[22] N. J. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt, "On the Security of RC4 in TLS," in *Proceedings of the 22nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 305–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534793

[23] T. Isobe, T. Ohigashi, Y. Watanabe, and M. Morii, "Full plaintext recovery attack on broadcast RC4," in *Preproceedings of FSE*, 2013.

[24] T. Ohigashi, T. Isobe, Y. Watanabe, and M. Morii, "How to recover any byte of plaintext on RC4," in *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, ser. Lecture Notes in Computer Science, T. Lange, K. E. Lauter, and P. Lisonek, Eds., vol. 8282. Springer, 2013, pp. 155–173. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-43414-7_8

[25] S. R. Fluhrer and D. McGrew, "Statistical analysis of the alleged RC4 keystream generator," in *FSE*, ser. Lecture Notes in Computer Science, B. Schneier, Ed., vol. 1978. Springer, 2000, pp. 19–30.

[26] J. Bonneau and S. Preibusch, "The password thicket: Technical and market failures in human authentication on the web," in *9th Annual Workshop on the Economics of Information Security, WEIS 2010, Harvard University, Cambridge, MA, USA, June 7 - 8*, 2010. [Online]. Available: http://weis2010.econinfosec.org/papers/session3/weis2010_bonneau.pdf

[27] I. Mantin and A. Shamir, "A practical attack on broadcast RC4," in *FSE*, ser. Lecture Notes in Computer Science, M. Matsui, Ed., vol. 2355. Springer, 2001, pp. 152–164.

[28] I. Mantin, "Predicting and distinguishing attacks on RC4 keystream generator," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, R. Cramer, Ed., vol. 3494. Springer, 2005, pp. 491–506.

[29] S. Sen Gupta, S. Maitra, G. Paul, and S. Sarkar, "(Non-) random sequences from (non-) random permutations – analysis of RC4 stream cipher," *Journal of Cryptology*, vol. 27, no. 1, pp. 67–108, 2012.

[30] S. Sarkar, S. Sen Gupta, G. Paul, and S. Maitra, "Proving TLS-attack related open biases of RC4," *IACR Cryptology ePrint Archive*, vol. 2013, p. 502, 2013. [Online]. Available: http://eprint.iacr.org/2013/502

[31] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," Internet Engineering Task Force, RFC 2246, Jan. 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2246.txt

[32] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," Internet Engineering Task Force, RFC 4346, Apr. 2006. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4346.txt

[33] ——, "The Transport Layer Security (TLS) Protocol Version 1.2," Internet Engineering Task Force, RFC 5246, Aug. 2008. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5246.txt

[34] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State," RFC 5077 (Proposed Standard), Internet Engineering Task Force, Jan. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5077.txt

[35] M. Weir, S. Aggarwal, M. P. Collins, and H. Stern, "Testing metrics for password creation policies by attacking large sets of revealed passwords," in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, Eds. ACM, 2010, pp. 162–175. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866327

[36] A. Adams and M. A. Sasse, "Users are not the enemy," *Commun. ACM*, vol. 42, no. 12, pp. 40–46, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/322796.322806

[37] D. Florencio and C. Herley, "A Large-scale Study of Web Password Habits," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 657–666. [Online]. Available: http://doi.acm.org/10.1145/1242572.1242661

[38] M. Zviran and W. J. Haga, "Password Security: An Empirical Study," *J. Manage. Inf. Syst.*, vol. 15, no. 4, pp. 161–185, Mar. 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=1189462.1189470

[39] J. Yan, A. Blackwell, R. Anderson, and A. Grant, "Password Memorability and Security: Empirical Results," *IEEE Security and Privacy*, vol. 2, no. 5, pp. 25–31, Sep. 2004. [Online]. Available: http://dx.doi.org/10.1109/MSP.2004.81

[40] J. Bonneau, "The science of guessing: Analyzing an anonymized

corpus of 70 million passwords," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 2012, pp. 538–552. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SP.2012.49

[41] T. Berners-Lee, R. Fielding, and H. Frystyk, "The Hypertext Transfer Protocol âĂŤ HTTP/1.0," RFC 1945 (Informational), Internet Engineering Task Force, May 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1945.txt

[42] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, "HTTP Authentication: Basic and Digest Access authentication," RFC 2617 (Informational), Internet Engineering Task Force, June 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2617.txt

[43] J. A. Akinyele, C. Garman, and S. Hohenberger, "Automating fast and secure translations from type-i to type-iii pairing schemes," Cryptology ePrint Archive, Report 2015/290, 2015, http://eprint.iacr.org/2015/290.

[44] M. Abe, J. Groth, M. Ohkubo, and T. Tango, "Converting cryptographic schemes from symmetric to asymmetric bilinear groups," in *CRYPTO*, 2014, pp. 241–260.

[45] G. Barthe, E. Fagerholm, D. Fiore, J. C. Mitchell, A. Scedrov, and B. Schmidt, "Automated analysis of cryptographic assumptions in generic group models," in *CRYPTO 2014*, 2014, pp. 95–112.

[46] S. C. Ramanna, S. Chatterjee, and P. Sarkar, "Variants of Waters' dual system primitives using asymmetric pairings - (extended abstract)," in *PKC '12*, 2012, pp. 298–315.

[47] J. Chen, H. W. Lim, S. Ling, H. Wang, and H. Wee, "Shorter IBE and signatures via asymmetric pairings," in *Pairing-Based Cryptography–Pairing 2012*. Springer, 2013, pp. 122–140.

[48] ——, "Shorter identity-based encryption via asymmetric pairings," *Des. Codes Cryptography*, vol. 73, no. 3, pp. 911–947, 2014.

[49] A. Contributors, "Advanced crypto software collection," http://hms.isi.jhu.edu/acsc/.

[50] J. A. Akinyele, C. Garman, I. Miers, M. W. Pagano, M. Rushanan, M. Green, and A. D. Rubin, "Charm: a framework for rapidly prototyping cryptosystems," *Journal of Cryptographic Engineering*, vol. 3, no. 2, pp. 111–128, 2013, http://www.charm-crypto.com/Main.html.

[51] M. Abe, J. Groth, M. Ohkubo, and T. Tango, 2015, private communications.

[52] B. Waters, "Dual system encryption: Realizing fully secure ibe and hibe under simple assumptions," in *CRYPTO*, 2009, pp. 619–636.

[53] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Proceedings of the Theory and practice of Software*, ser. TACAS'08/ETAPS'08, 2008, pp. 337–340.

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

[55] J. Camenisch and A. Lysyanskaya, "Signature schemes and anonymous credentials from bilinear maps," in *CRYPTO*, vol. 3152, 2004, pp. 56–72.

[56] D. Boneh and X. Boyen, "Efficient selective-id secure identity-based encryption without random oracles," in *Advances in Cryptology - EUROCRYPT 2004*, 2004, vol. 3027, pp. 223–238.

[57] ——, "Efficient selective-id secure identity based encryption without random oracles," Cryptology ePrint Archive, Report 2004/172, 2004, http://eprint.iacr.org/.

[58] C. Gentry, "Practical identity-based encryption without random oracles," in *EUROCRYPT*, 2006, pp. 445–464.

[59] D. Boneh, C. Gentry, and B. Waters, "Collusion resistant broadcast encryption with short ciphertexts and private keys," in *CRYPTO'05*, 2005, pp. 258–275.

[60] M. Abe, M. Chase, B. David, M. Kohlweiss, R. Nishimaki, and M. Ohkubo, "Constant-size structure-preserving signatures: Generic constructions and simple assumptions," Cryptology ePrint Archive, Report 2012/285, 2012, http://eprint.iacr.org/.

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

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

[63] D. Boneh and X. Boyen, "Short signatures without random oracles," in *EUROCRYPT*, vol. 3027, 2004, pp. 382–400.

[64] C. Garman, M. Green, and I. Miers, "Decentralized anonymous credentials," Cryptology ePrint Archive, Report 2013/622, 2013, http://eprint.iacr.org/2013/622.

[65] D. Chaum, "Security without identification: transaction systems to make big brother obsolete," *Communications of the ACM*, 1985. [Online]. Available: http://doi.acm.org/10.1145/4372.4373

[66] S. A. Brands, *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy*. Cambridge, MA, USA: MIT Press, 2000.

[67] J. Camenisch and A. Lysyanskaya, "An efficient system for non-transferable anonymous credentials with optional anonymity revocation," ser. EUROCRYPT, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=647086.715698

[68] ——, "A signature scheme with efficient protocols," ser. SCN'02, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=1766811.1766838

[69] ——, "Dynamic accumulators and application to efficient revocation of anonymous credentials," in *CRYPTO*, 2002.

[70] M. Belenkiy, M. Chase, M. Kohlweiss, and A. Lysyanskaya, "P-signatures and noninteractive anonymous credentials," in *Theory of Cryptography*, 2008.

[71] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *CCS '04*. New York, NY, USA: ACM, 2004, pp. 132–145. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030103

[72] "TPM Main Specification v1.2." [Online]. Available: http://www.trustedcomputinggroup.org/resources/tpm_main_specification

[73] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[74] J. Clark and A. Essex, "Commitcoin: Carbon dating commitments with bitcoin," in *Financial Cryptography and Data Security*. Springer, 2012, pp. 390–398.

[75] J. R. Douceur, "The sybil attack," in *Peer-to-Peer Systems*, 2002. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-45748-8_24

[76] I. Damgård, K. Dupont, and M. Ø. Pedersen, "Unclonable group identification," ser. EUROCRYPT, 2006. [Online]. Available: http://dx.doi.org/10.1007/11761679_33

[77] J. Camenisch, S. Hohenberger, M. Kohlweiss, A. Lysyanskaya, and M. Meyerovich, "How to win the clonewars: efficient periodic n-

times anonymous authentication," ser. CCS, 2006. [Online]. Available: http://doi.acm.org/10.1145/1180405.1180431

[78] M. Z. Lee, A. M. Dunn, B. Waters, E. Witchel, and J. Katz, "Anon-pass: Practical anonymous subscriptions," in *IEEE Security and Privacy*, 2013.

[79] "Namecoin," Available at http://namecoin.info/. [Online]. Available: http://namecoin.info/

[80] H. Shacham and B. Waters, "Compact proofs of retrievability," in *Advances in Cryptology-ASIACRYPT 2008*. Springer, 2008, pp. 90–107.

[81] T. Sander and A. Ta-Shma, "Auditable, anonymous electronic cash (extended abstract)," in *CRYPTO*, 1999.

[82] I. Miers, C. Garman, M. Green, and A. Rubin, "Zerocoin: Anonymous distributed e-cash from bitcoin," in *IEEE Security and Privacy*, 2013.

[83] R. L. Rivest, A. Shamir, and Y. Tauman, "How to leak a secret," in *ASIACRYPT*, 2001.

[84] J. Camenisch and A. Lysyanskaya, "Dynamic accumulators and application to efficient revocation of anonymous credentials," in *CRYPTO*, 2002, extended Abstract. [Online]. Available: http://cs.brown.edu/~anna/papers/camlys02.pdf

[85] "Dot-bit," Available at http://dot-bit.org/. [Online]. Available: http://dot-bit.org/

[86] N. Borisov, "Computational puzzles as sybil defenses," in *Peer-to-Peer Computing, 2006. Sixth IEEE International Conference on.* IEEE, 2006, pp. 171–176.

[87] T.-W. J. Ngan, D. S. Wallach, and P. Druschel, "Enforcing fair sharing of peer-to-peer resources," in *Peer-to-Peer Systems II*, 2003. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-45172-3_14

[88] D. Obenshain, T. Tantillo, A. Newell, C. Nita-Rotaru, and Y. Amir, "Intrusion-tolerant cloud monitoring and control," in *LADIS*, 2012.

[89] R. Snader and N. Borisov, "Eigenspeed: secure peer-to-peer bandwidth evaluation." in *IPTPS*, 2009, p. 9.

[90] N. Barić and B. Pfitzmann, "Collision-free accumulators and fail-stop signature schemes without trees," in *EUROCRYPT*, 1997.

[91] E. Fujisaki and T. Okamoto, "Statistical zero knowledge protocols to prove modular polynomial relations," in *CRYPTO*, 1997.

[92] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, 1976.

[93] O. Goldreich, S. Micali, and A. Wigderson, "Proofs that yield nothing but their validity and a methodology of cryptographic protocol design," in *FOCS*, 1986.

[94] C.-P. Schnorr, "Efficient signature generation for smart cards," *Journal of Cryptology*, vol. 4, no. 3, pp. 239–252, 1991.

[95] R. Cramer, I. Damgård, and B. Schoenmakers, "Proofs of partial knowledge and simplified design of witness hiding protocols," in *CRYPTO*, 1994.

[96] J. Camenisch and M. Michels, "Proving in zero-knowledge that a number n is the product of two safe primes," in *EUROCRYPT*, 1999.

[97] J. L. Camenisch, "Group signature schemes and payment systems based on the discrete logarithm problem," Ph.D. dissertation, ETH Zürich, 1998.

[98] S. Brands, "Rapid demonstration of linear relations connected by boolean operators," in *EUROCRYPT*, 1997.

[99] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *CRYPTO*, 1986.

[100] M. Chase and A. Lysyanskaya, "On signatures of knowledge," in *CRYPTO*, vol. 4117 of LNCS, 2006, pp. 78–96.

[101] J. Camenisch and M. Stadler, "Efficient group signature schemes for large groups," in *CRYPTO*, 1997.

[102] J. Benaloh and M. de Mare, "One-way accumulators: a decentralized alternative to digital signatures," in *EUROCRYPT*, 1994.

[103] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *Journal of the ACM*, 1986.

[104] Y. Dodis and A. Yampolskiy, "A verifiable random function with short proofs and keys," ser. PKC, 2005.

[105] M. Naor and O. Reingold, "Number-theoretic constructions of efficient pseudo-random functions," *Journal of the ACM (JACM)*, 2004.

[106] S. Coull, M. Green, and S. Hohenberger, "Access controls for oblivious and anonymous systems," in *TISSEC*, 2011.

[107] T. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *CRYPTO*, 1991.

[108] J. Camenisch, R. Chaabouni *et al.*, "Efficient protocols for set membership and range proofs," in *Advances in Cryptology-ASIACRYPT 2008*. Springer, 2008, pp. 234–252.

[109] F. Boudot, "Efficient proofs that a committed number lies in an interval," in *EUROCRYPT 2000*. Springer, 2000, pp. 431–444.

[110] J. Groth, "Non-interactive zero-knowledge arguments for voting," in *Applied Cryptography and Network Security*. Springer, 2005, pp. 467–482.

[111] H. Lipmaa, "On diophantine complexity and statistical zero-knowledge arguments," in *Advances in Cryptology-ASIACRYPT 2003*. Springer, 2003, pp. 398–415.

[112] "TPM Main Specification v1.1." [Online]. Available: http://www.trustedcomputinggroup.org/resources/tcpa_main_specification_version_11b

[113] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *Proceedings of the 33rd Annual International Cryptology Conference*, ser. CRYPTO '13, 2013, pp. 90–108.

[114] B. Parno, C. Gentry, J. Howell, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, ser. Oakland '13, 2013, pp. 238–252.

[115] S. S. Gupta, S. Maitra, G. Paul, and S. Sarkar, "RC4: (Non-) random words from (non-)random permutations," *IACR Cryptology ePrint Archive*, vol. 2011, p. 448, 2011.

[116] M. Crispin, "Internet Message Access Protocol âĂŤ Version 4rev1," RFC 3051 (Informational), Internet Engineering Task Force, March 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3051.txt

[117] J. Myers, "IMAP4 Authentication Mechanisms," RFC 1731 (Informational), Internet Engineering Task Force, December 1994. [Online]. Available: http://www.ietf.org/rfc/rfc1731.txt

[118] K. Zeilenga, "IMAP4 Authentication Mechanisms," RFC 4616 (Informational),

Internet Engineering Task Force, August 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4616.txt

[119] R. Siemborski and A. Gulbrandsen, "IMAP Extension for Simple Authentication and Security Layer (SASL) Initial Client Response," RFC 4959 (Informational), Internet Engineering Task Force, September 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4959.txt

[120] B. Leiba, "IMAP4 IDLE command," RFC 2177 (Proposed Standard), Internet Engineering Task Force, Jun. 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2177.txt

[121] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5th ed., 2012, http://gmplib.org/.

[122] S. D. Galbraith, "Supersingular curves in cryptography," in *ASIACRYPT*, 2001, pp. 495–513.

[123] D. Page, N. Smart, and F. Vercauteren, "A comparison of MNT curves and supersingular curves," *Applicable Algebra in Eng,Com and Comp*, vol. 17, no. 5, pp. 379–392, 2006.

[124] P. S. L. M. Barreto and M. Naehrig, "Pairing-friendly elliptic curves of prime order," in *SAC*, vol. 3897, 2006, pp. 319–331, http://cryptojedi.org/papers/#pfcpo.

[125] S. D. Galbraith, K. G. Paterson, and N. P. Smart, "Pairings for cryptographers," 2006, cryptology ePrint Archive: Report 2006/165.

Appendix A

Double-byte biases in the RC4 keystream distribution

As mentioned in Section 2.2, we estimated the initial double-byte keystream distributions for RC4 in the first 512 positions using roughly 4800 core-days of computation to generate 2^{44} RC4 keystreams for random 128-bit RC4 keys (as used in TLS). As noted there, while the gross behaviour that we observed is dominated by products of the known single-byte biases in the first 256 positions and by the Fluhrer-McGrew biases in the later positions, we did observe some new and interesting double-byte biases.

In Figure A.1, for instance, the influence of the single-byte key-length-dependent bias [115], and the single-byte r-bias [22] are evident. The former can be observed as the strong vertical line at $Z_{16} = \texttt{0xE0}$, while the latter can be seen as the lines at

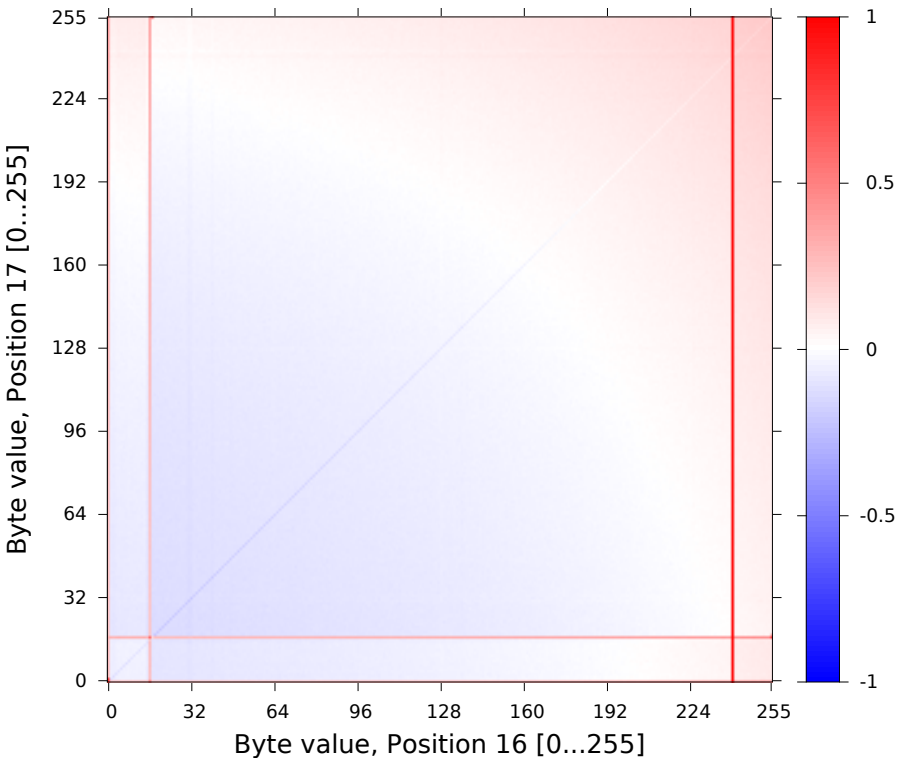


Figure A.1: Measured biases for RC4 keystream byte pair (Z_{16}, Z_{17}). The colouring scheme encodes the strength of the bias, i.e., the deviation from the expected probability of $1/2^{16}$, scaled by a factor of 2^{22}, capped at a maximum of 1.

$Z_{16} = \texttt{0x10}$ and $Z_{17} = \texttt{0x11}$. The faint diagonal line appears to be a new double-byte bias (that is not accounted for as a product of single-byte biases). It appears in many early positions. For example, it is at least twice as strong as that arising in the product distribution for at least 64 of the 256 possible byte values from positions (Z_3, Z_4) up to positions (Z_{110}, Z_{111}). It then gradually disappears, but reappears at around positions (Z_{192}, Z_{193}) (albeit as a positive bias) and persists up to positions (Z_{257}, Z_{258}) (changing sign again at (Z_{255}, Z_{256})).

The presence of horizontal and vertical lines in Figure A.1 and the absence of other strong biases, which is typical for the early positions, indicates that the adjacent

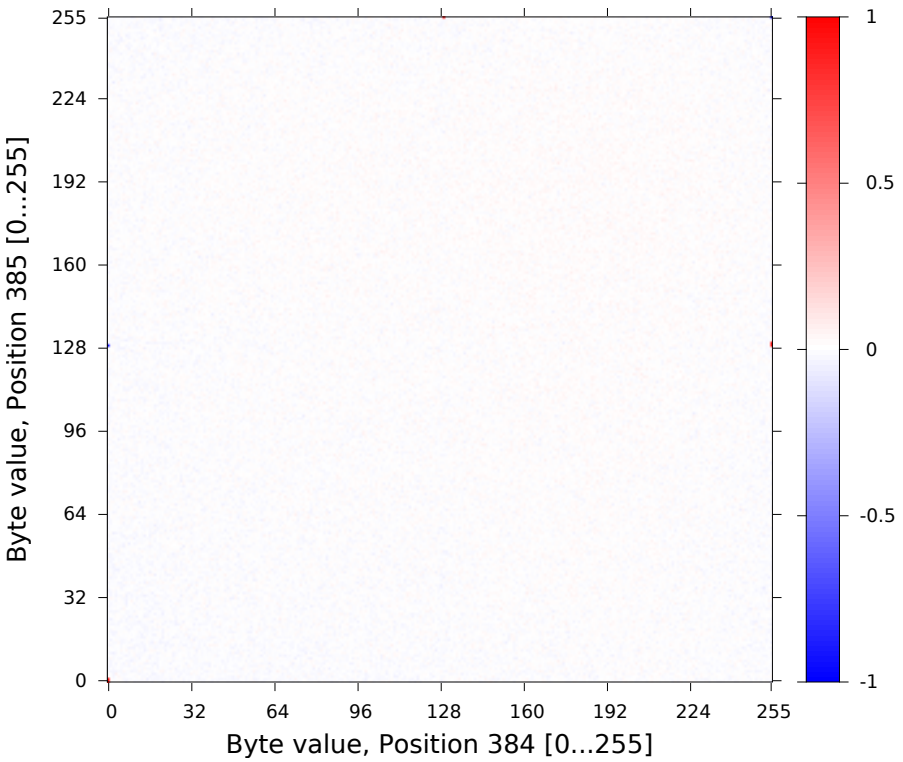
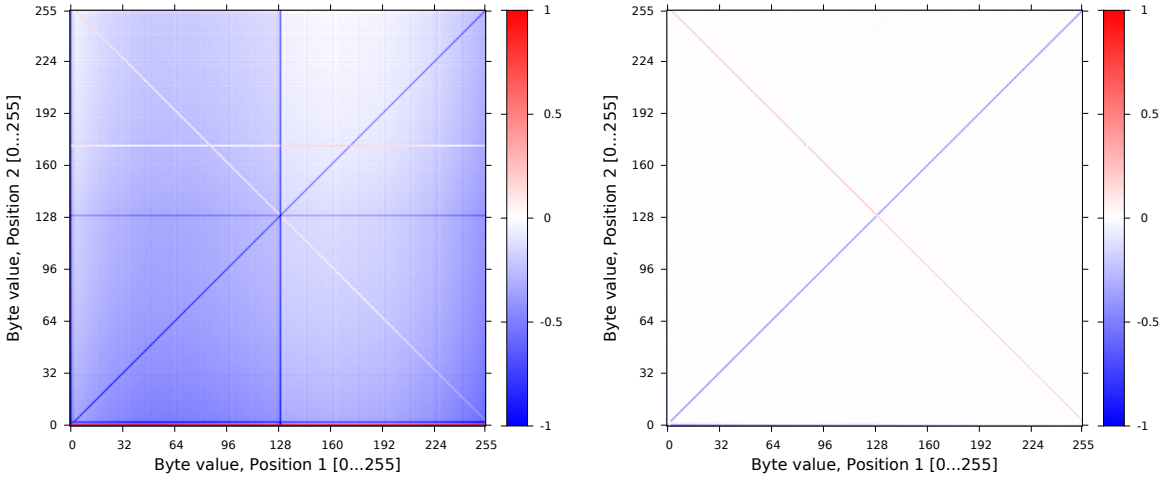


Figure A.2: Measured biases for RC4 keystream byte pair (Z_{384}, Z_{385}). The colouring scheme encodes the strength of the bias, i.e., the deviation from the expected probability of $1/2^{16}$, scaled by a factor of 2^{24}, capped at a maximum of 1.

bytes behave largely independently of each other. In other words, there are very few strong conditional biases in the first 256 positions of the RC4 keystream. For later positions in the keystream, Figure A.2 depicts what is typical in terms of bias behaviour: the presence of Fluhrer-McGrew biases only. These are visible in Figure A.2 at (Z_{384}, Z_{385})= $(\texttt{0x00}, \texttt{0x01})$ and $(\texttt{0x81}, \texttt{0xFF})$ for example.

Finally, of particular interest is the distribution of (Z_1, Z_2). Figure A.3a shows the raw distribution for this position pair, while Figure A.3b shows the residual biases when the product distribution of Z_1 and Z_2 is removed. Note that the raw distribution is predominately negatively biased; this is because of the effect of the



(a) Colouring scheme encodes the strength of the bias, scaled by a factor of 2^{22}, capped at a maximum of 1.

(b) Colouring scheme encodes the strength of the bias after the product of single-byte biases for positions Z_1 and Z_2 is removed, scaled by a factor of 2^{22}, capped at a maximum of 1.

Figure A.3: Measured biases for RC4 keystream byte pair (Z_1, Z_2).

large Mantin-Shamir positive bias towards `0x00` in position Z_2, and the compensating negative single byte biases for all other values of Z_2. Note also the two diagonal lines in Figure A.3b. The "positive" (blue-coloured) diagonal here represents a negative bias in (Z_1, Z_2) for all byte pairs (z, z) where $z \in \mathcal{B} \setminus \{\texttt{0x00}\}$; this bias is also evident in the raw distribution in Figure A.3a. The "negative diagonal" in Figure A.3b shows that there is a systematic difference between the raw double-byte distribution and the product distribution. It manifests itself as a white-coloured negative diagonal in the raw double-byte distribution shown in Figure A.3a; thus, in the raw distribution, it forms a structured set of unbiased pairs against a largely negatively-biased background.

The only other previously known bias of this nature in this portion of the keystream

is due to Isobe *et al.* [23], who showed that:

$$\Pr(Z_1 = \texttt{0x00} \wedge Z_2 = \texttt{0x00}) = 2^{-16} \cdot (1 + 2^{0.996}).$$

This bias is also evident in Figure A.3. By contrast, the new diagonal biases are negative, sporting magnitudes in the region of 2^{-22}. For example, we empirically observe:

$$\Pr(Z_1 = \texttt{0x14} \wedge Z_2 = \texttt{0x14}) = 2^{-16} \cdot (1 - 2^{-6.097}).$$

Let us now formally define a *large* double-byte bias to be one whose magnitude is at least 2^{-24}. We observed 103,031 such large biases in total. Note that with 2^{44} keystreams, all such biases are statistically significant and highly unlikely to arise from random fluctuations in our empirical analysis. For, in each position pair $(r, r+1)$ we have 2^{16} counters, one for each possible pair (Z_r, Z_{r+1}), so, in the absence of any biases, each counter would be (roughly) normally distributed with mean $2^{44} \cdot 2^{-16} = 2^{28}$ and standard deviation σ of approximately $\sqrt{2^{28}} = 2^{14}$. Then a bias of size 2^{-24} would lead to a counter value of around

$$2^{44}(2^{-16} + 2^{-24}) = 2^{28} + 2^6 \cdot 2^{14}$$

which is a 64σ event. Using the standard tail bound for the normal distribution,

even with 2^{25} counters in total (across 512 positions), we would expect to see only $2^{18} \cdot e^{-2048}/\pi \ll 1$ such events.

We found that 643 (less than 1%) of the large biases that we observed were at least twice the size (in absolute value) of biases resulting from the products of single-byte biases or of the expected Fluhrer-McGrew bias in the same positions. In other words, most of the large biases that we observed arise from the product distribution or are explained by Fluhrer and McGrew's results. We also note that we did find double-byte biases in all the positions predicted by Fluhrer and McGrew [25] starting from byte pair (Z_4, Z_5) onwards. This is not surprising given that the idealized assumption concerning the internal state of the RC4 algorithm that was used in the analysis of [25] is well approximated after a few invocations of the RC4 keystream generator. However, in many such cases, the magnitude of the bias we observed is greater than is predicted by the Fluhrer-McGrew analysis. For example, in byte pair (Z_6, Z_7) we observed

$$\Pr(Z_6 = \texttt{0x07} \wedge Z_7 = \texttt{0xFF}) = 2^{-16} \cdot (1 - 2^{-6.487}),$$

whereas the corresponding specified Fluhrer-McGrew probability for this byte pair, namely the $(i+1, \texttt{0xFF})$ byte pair where i is the internal variable of the RC4 keystream generator, is $2^{16}(1 + 2^{-8})$.

We do, however, note a transition to the regular Fluhrer-McGrew double-byte biases from position 257 onwards. We also note the disappearance of the single-byte

biases from roughly this point onwards. This is illustrated in Figure A.4, which shows the absolute value of the largest single-byte bias observed in our data as a function of keystream position r.

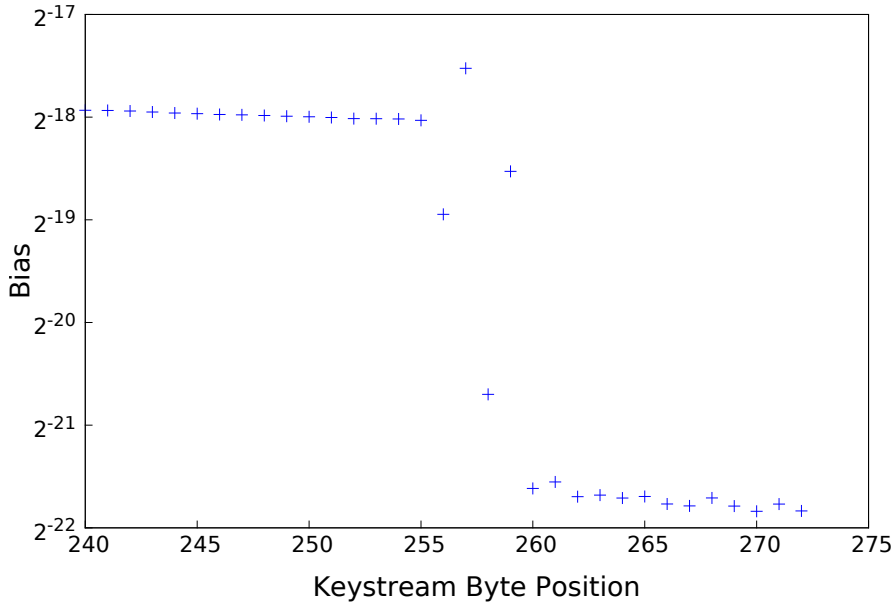


Figure A.4: Absolute value of the largest single-byte bias for keystream bytes Z_{240} to Z_{272}.

Appendix B

Details of IMAP Proof of Concept

In this section we describe the proof-of-concept implementation of our attacks against IMAP in more detail.

B.1 Introducing IMAP

The Internet Message Access Protocol, currently at version 4: revision 1 (IMAP4rev1), facilitates the retrieval and manipulation of e-mail messages stored on a server. We provide a brief description of the protocol, focusing only on the client/server commands and responses that are relevant to this work. Further details can be found in [116].

An IMAP session commences with the establishment of a client/server connection, followed by an initial greeting message from the server and the subsequent exchange of messages between the client and the server. All IMAP messages are text-based

with client messages taking the form of commands and server messages typically taking the form of responses. Client commands cover a broad spectrum of functions including the creation, searching and deletion of e-mails but of interest here are the `AUTHENTICATE` and `LOGIN` commands. These commands are only valid when an IMAP server is in what is known as the *not authenticated* state. This state is entered into when a client/server connection is established, and the client must supply legitimate authentication credentials so as to enable the server to move to the *authenticated* state. The `AUTHENTICATE` command specifies an authentication mechanism to be used by the server to identify and authenticate the user. The various mechanisms available are specified in [117, 118, 119]. We will target the `PLAIN` Simple Authentication and Security Layer (SASL) mechanism [118]. The arguments for this mechanism include an authorization identity string, a username and a password. The authorization identity string is a sequence of zero or more Unicode characters that represent the identity the client wishes to assume. It is possible for the authorization identity string to be empty, in which case, the server will derive an authorization identity from the other credentials provided. As with BasicAuth, the authorization credentials are not directly protected by IMAP.

The format of the `AUTHENTICATE` command is specified in [116] and involves the transmission of username and password in Base64 encoded form, with these fields being separated by a NULL character. The `LOGIN` command identifies and authenticates a user to the server by providing a username and a password. The command has the

following format:

```
A001 LOGIN "username" "password".
```

These credentials are again transmitted in the clear. We note that the IMAP specification [116] recommends that the `AUTHENTICATE PLAIN` and `LOGIN` commands only be used when a secure channel is available, such as provided by TLS. In fact, the `LOGIN` command is recommended as a last resort only. The establishment of a TLS session is achieved by the `STARTTLS` command. Once a client issues this command, it must wait for the server to acknowledge this request and the subsequent completion of the TLS negotiation before issuing any further commands. Also, the client must discard any information about server capabilities received prior to the issuance of the `STARTTLS` command. This is to protect against Man-In-The-Middle (MITM) attacks which alter the capability list prior to the establishment of the TLS session. According to [116], "IMAP client and server implementations MUST implement the TLS_RSA_WITH_RC4_128_MD5 cipher suite, and SHOULD implement the TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA cipher suite." The use of RC4 in IMAP can therefore be expected to be prevalent.

B.2 Attacking IMAP

As with BasicAuth, we require the password to be located sufficiently early in the plaintext streams of IMAP connections, and we need to find a means of forcing an IMAP to repeatedly send the `AUTHENTICATE` or `LOGIN` commands.

We have verified the former condition to hold for specific clients such as Mozilla Thunderbird. For example, with that client, we saw the password vary between positions 102 and 128 for `AUTHENTICATE`, depending on the server that the client connected to; however this number was consistent on a per server basis. We speculate that this is because of different server configurations or presented client capabilities. Moreover, clients are typically configured to connect to the server and check for new mail on a regular basis, typically every 10 minutes, but often much more frequently.

We built a proof-of-concept demonstration of an attack on IMAP, with the setup being as follows. A client machine ran Mozilla Thunderbird, set to check for new mail every 10 minutes. The client can be setup to either connect to the IMAP server via `STARTTLS` on port 143 or `IMAPS` on port 993. The latter is an alternative means of establishing a TLS connection for IMAP data transfers. The server was running the Dovecot[1] open source IMAP email server, configured with TLS and set to prefer RC4 ciphersuites. The MITM was set up between the client and the server. All three (client, server, MITM) were separate physical machines, though they all resided on the same network. We again used Scapy to do all the packet handling at the MITM.

[1] `http://www.dovecot.org/`

The MITM capability was achieved by configuring the Thunderbird client to connect directly to the MITM machine, which then forwarded all traffic through to the IMAP server (via an iptables rule). The MITM then also has the ability to delay or block packets, though it did not exercise that capability for the most part in our attack. This way of configuring our network replaced the ARP spoofing step used in our BasicAuth attack.

The attack then proceeded as follows. The client and server were allowed to establish a TLS-protected IMAP connection. After the password was sent (an event that we can identify because of the rigid format of IMAP messages), the MITM issued TCP `RST` messages terminating the connection. The next time the client tries to poll the server, it is forced to redo the TCP handshake and the TLS handshake, thus opening a new session and allowing the MITM to collect another encryption of the password (since every time the client opens a new connection to the server, it must reauthenticate).

In our proof-of-concept, we successfully executed all of the above steps and collected encrypted passwords. However, the rate at which the encrypted passwords could be gathered was slow, because of its dependence on the frequency with which the IMAP client polls the server for new mail. There are several different ways in which the rate can be increased in practice:

- Many mail clients (though not Thunderbird) actively reconnect and perform client authentication whenever the TLS connection between client and server is

broken. An active MITM attacker could then simply break the TLS connection between client and server at as high a speed as the client and server can handle, thereby increasing the rate at which encrypted passwords are sent.

- The widely implemented IMAP `IDLE` command [120] enables a server to notify a client that an event has occurred on the server of relevance to the user, for example that new mail has arrived at the server. It works by the client regularly sending an `IDLE` command to the server, and the server responding with alerts concerning new messages.[2] When used, the rate of `IDLE` commands can be expected to be every few seconds to a minute (so that the user has the illusion of receiving instant updates). Each `IDLE` command can be expected to be sent on a fresh TLS connection and involve client authentication.

- An alternative mechanism, widely used on cell phones, are PUSH notifications. Here, the user is notified directly by the server when new mail arrives (or another event of significance occurs), rather than in response to a poll. By sending a large amount of spam e-mail to an account at a constant rate, and having the MITM break the TLS connection at a similar rate, an attacker could ensure that the client connects to the server at that rate, with each connection involving reauthentication and therefore retransmission of the user password. Of course, it is unreasonable to send 2^{24} e-mails to a single account in order to mount our attack. However, note that for a PUSH system to be effective, the PUSH

[2] See `http://www.isode.com/whitepapers/imap-idle.html` for a good overview.

notifications would need to be capable of being sent over an unprotected channel (since such a channel may not be in place when the notification needs to be sent). This makes them spoofable by an active MITM, which may be exploited as a means to trigger the establishment of the required TLS connections.

The detailed exploration of these different methods for speeding up our proof of concept against IMAP is left for future work.

Appendix C

Current Efficiency Numbers for Type-I and Type-III Pairings

Sym. vs. Asym. Setting	Size (in bits) $\mathbb{G}_1$	$\mathbb{G}_2$	$\mathbb{G}_T$	Exp. Time (in milliseconds) $\mathbb{G}_1$	$\mathbb{G}_2$	$\mathbb{G}_T$	**Pairing Time**
SS1536 (or Type-I)	1536	1536	3072	5.3 ms	5.3 ms	1.0 ms	14.9 ms
BN256 (or Type-III)	256	1024	3072	0.2 ms	1.2 ms	2.1 ms	2.2 ms

Figure C.1: Comparing Size and Efficiency of Pairing-based Curves.

We include current efficiency numbers for Type-I and Type-III groups as implemented in the highly efficient RELIC cryptographic toolkit version 0.4 [62] (using the GMP library [121] for big number operations and the default configuration options for prime field arithmetic) measured on a standard workstation.[1] In Figure C.1, we show the differences between Type-I and Type-III pairings at the same security level in terms of group representation and efficiency. [2] A typical candidate for Type-I

[1] 2.4 GHz Intel Core i5 processor and 8GB of RAM (1067 MHz DDR3) running Mac OS X Lion version 10.7.5

[2] A careful reader may observe that the exponentiation time for $\mathbb{G}_T$ in SS1536 appears surprisingly

are supersingular elliptic curves (or SS) [122, 123] in which the embedding degree is typically small (*i.e.*, $k \leq 6$). One such example is a supersingular curve at the 128-bit security level where the prime order of the group is large, $|p|=$ 1536-bits, and the embedding degree is $k = 2$. Conversely, one common Type-III candidate at the same security level are Barreto-Naehrig (BN) [124] curves in which the embedding degree is much larger (*e.g.*, $k = 12$) and the prime order can be as small as $|p|=$ 256-bits. As reflected in Table C.1, group operations and pairing times in the Type-III setting can be drastically more efficient and have shorter representations than the Type-I setting.

We remark on hashing into Type-I and Type-III pairing groups. In the Type-I setting, it is feasible to hash arbitrary strings into $\mathbb{G}$, e.g., for the SS curve, hashing arbitrary strings to $\mathbb{G}$ takes on average 36.8 ms. In the Type-III setting (*e.g.*, over ordinary elliptic curves), it is feasible to hash arbitrary strings into both $\mathbb{G}_1$ and $\mathbb{G}_2$ independently with different costs, e.g., for the BN curve, hashing to $\mathbb{G}_1$ takes 0.04 ms and to $\mathbb{G}_2$ takes 0.37 ms on average (a ratio of roughly 9 to 1 from $\mathbb{G}_2$ to $\mathbb{G}_1$). See [125] for more details.

small. We reassure the reader that this is not a typo. With the SS1536, $\mathbb{G}_T = {F_p}^2$ is a lower extension of a larger field, whereas with BN256, $\mathbb{G}_T = {F_p}^{12}$, which is a higher extension of a smaller field. Thus, even though the elliptic curve points are larger with SS1536, the field multiplication operation in $\mathbb{G}_T$ is quite efficient. This does not apply to $\mathbb{G}_1, \mathbb{G}_2$ as those are doing scalar multiplication.

Appendix D

SDL Descriptions for Section 3.4

We now provide examples of the input and output Scheme Description Language (SDL) for AutoGroup+.

D.1 SDL as Input

First we will show our SDL transcription of the DBDH assumption:

```
name := DBDH

setting := symmetric


BEGIN :: types

a := ZR

b := ZR
```

```
c := ZR

z := ZR

END :: types


BEGIN :: func:setup

input := None

 a := random(ZR)

 b := random(ZR)

 c := random(ZR)

 z := random(ZR)

 g := random(G1)


 assumpKey := list{g, a, b, c, z}


output := assumpKey

END :: func:setup


BEGIN :: func:assump

input := assumpKey

 assumpKey := expand{g, a, b, c, z}

 A := g ^ a
```

```
 B := g ^ b

 C := g ^ c


 coinflip := random(bin)

 BEGIN :: if

 if { coinflip == 0 }

   Z := e(g, g) ^ (a * b * c)

 else

   Z := e(g, g) ^ z

 END :: if


 assumpVar := list{g, A, B, C, Z}

output := assumpVar

END :: func:assump
```

Then, the full SDL transcription for the symmetric BB HIBE scheme [57]:

```
name := BB04HIBE

setting := symmetric


BEGIN :: types

M := GT
```

```
ID1 := ZR

ID2 := ZR

END :: types



BEGIN :: func:setup

input := None


g := random(G1)

alpha := random(ZR)

g1 := g ^ alpha

h1 := random(G1)

h2 := random(G1)

g2 := random(G1)

g2alpha := g2 ^ alpha


msk := list{g2alpha}

pk := list{g, g1, g2, h1, h2}

output := list{msk, pk}

END :: func:setup
```

```
BEGIN :: func:keygen

input := list{pk, msk, ID1, ID2}

pk := expand{g, g1, g2, h1, h2}

msk := expand{g2alpha}


r1 := random(ZR)

r2 := random(ZR)

d1 := g2alpha * \

   (((g1^ID1)*h1)^r1) * (((g1^ID2)*h2)^r2)

d2 := g ^ r1

d3 := g ^ r2


sk := list{d1, d2, d3}

output := sk

END :: func:keygen



BEGIN :: func:encrypt

input := list{pk, M, ID1, ID2}

pk := expand{g, g1, g2, h1, h2}
```

```
s := random(ZR)

C1 := (e(g1,g2)^s) * M

C2 := g ^ s

C3 := ((g1^ID1) * h1)^s

C4 := ((g1^ID2) * h2)^s


ct := list{C1, C2, C3, C4}

output := ct

END :: func:encrypt




BEGIN :: func:decrypt

input := list{pk, sk, ct}

pk := expand{g, g1, g2, h1, h2}

ct := expand{C1, C2, C3, C4}

sk := expand{d1, d2, d3}


M := C1*((e(C3,d2) * e(C4,d3))/(e(C2,d1)))


output := M
```

```
END :: func:decrypt
```

Finally, the reduction from [57] for the BB HIBE scheme:

```
name := BB04

setting := symmetric

l := 2

k := 2


BEGIN :: types

l := Int

j := Int

k := Int

M := list{GT}

ID := list{ZR}

IDstar := list{ZR}

alphai := list{ZR}

h := list{G1}

r := list{ZR}

di := list{G1}

Ci := list{G1}

msk := G1

END :: types
```

```
BEGIN :: func:setup

input := list{IDstar}


 a := random(ZR)

 b := random(ZR)

 c := random(ZR)

 z := random(ZR)

 g := random(G1)

 A := g^a

 B := g^b

 C := g^c


 coinflip := random(bin)

BEGIN :: if

 if { coinflip == 0 }

    Z := e(g, g)^(a * b * c)

 else

    Z := e(g, g)^z

END :: if
```

```
 g1 := A
 g2 := B
 g3 := C

BEGIN :: for
 for{i := 1, l}
   alphai#i := random(ZR)
   h#i := (g1^-IDstar#i) * (g^alphai#i)
END :: for

 pk := list{g, g1, g2, h}
 assumpVar := list{A, B, C, Z}
 reductionParams := list{g3, alphai, IDstar}
output := list{msk, pk, reductionParams, assumpVar}
END :: func:setup

BEGIN :: func:queries
input := list{j, pk, ID, reductionParams}
pk := expand{g, g1, g2, h}
```

```
reductionParams := expand{g3, alphai, IDstar}


BEGIN :: for
 for{i := 1, j}
    r#i := random(ZR)
END :: for


dotProd1 := init(G1)
BEGIN :: for
 for{v := 1, j}
    dotProd1 := dotProd1 * (((g1^(ID#v - IDstar#v)) * \
          (g^alphai#v))^r#v)
END :: for


 d1 := (g2^((-alphai#j) / (ID#j - IDstar#j))) * dotProd1


BEGIN :: for
 for{i := 1, j}
 BEGIN :: if
   if {i == j }
      di#j := (g2^(-1/(ID#j - IDstar#j))) * (g^r#j)
```

```
   else
      di#i := g^r#i
 END :: if
END :: for


sk := list{d1, di}
output := sk
END :: func:queries



BEGIN :: func:challenge
input := list{M, ID, reductionParams, assumpVar}
pk := expand{g, g1, g2, h}
assumpVar := expand{A, B, C, Z}
reductionParams := expand{g3, alphai, IDstar}

 b := random(bin)


 C1 := M#b * Z
 C2 := g3
```

```
BEGIN :: for
 for{i := 1, k}
    Ci#k := g3 ^ alphai#i
END :: for


 ct := list{C1, C2, Ci}
output := ct
END :: func:challenge
```

We provide the configuration file that embeds the metadata required by Auto-Group+ to perform the translation:

```
schemeType = "PKENC"
assumption = ["DBDH"]
reduction = ["reductionBB04HIBE"]
short = "public-keys"


masterPubVars = ["pk"]
masterSecVars = ["msk"]


keygenPubVar = "pk"
```

```
keygenSecVar = "sk"

ciphertextVar = "ct"


reducCiphertextVar = "ct"

reducQueriesSecVar = "d"
```

D.2 Translated Scheme and Assumption SDL Descriptions

We now show the SDL outputs of AutoGroup+. The first is the SDL output of the co-DBDH assumption:

```
name := DBDH

setting := asymmetric


BEGIN :: types

a := ZR

b := ZR

c := ZR

z := ZR

END :: types
```

```
BEGIN :: func:setup

input := None

a := random(ZR)

b := random(ZR)

c := random(ZR)

z := random(ZR)

gG1 := random(G1)

gG2 := random(G2)

assumpKey := \

   list{gG1, gG2, a, b, c, z}

output := assumpKey

END :: func:setup


BEGIN :: func:assump

input := assumpKey

assumpKey := \

   expand{gG1, gG2, a, b, c, z}

A := (gG1^a)

BG1 := (gG1^b)

BG2 := (gG2^b)
```

```
CG1 := (gG1^c)

CG2 := (gG2^c)

coinflip := random(bin)

BEGIN :: if

if {coinflip == 0}

Z := (e(gG1,gG2)^((a * b) * c))

else

Z := (e(gG1,gG2)^z)

END :: if

assumpVar := list{gG1, gG2, A,\

       BG1, BG2, CG1, CG2, Z}

output := assumpVar

END :: func:assump
```

The second SDL output is the asymmetric BB HIBE scheme [57] that optimally minimizes the public key parameters:

```
name := BB04HIBE

setting := asymmetric


BEGIN :: types

M := GT
```

```
ID1 := ZR

ID2 := ZR

END :: types


BEGIN :: func:setup

input := None


gG1 := random(G1)

gG2 := random(G2)

alpha := random(ZR)

g1 := (gG1^alpha)

h1 := random(ZR)

h1G1 := (gG1^h1)

h2 := random(ZR)

h2G1 := (gG1^h2)

g2 := random(ZR)

g2G1 := (gG1^g2)

g2G2 := (gG2^g2)

g2alpha := (g2G1^alpha)

msk := list{g2alpha}

pk := list{gG1, gG2, g1, g2G1, g2G2, h1G1, h2G1}
```

```
output := list{msk, pk}

END :: func:setup


BEGIN :: func:keygen

input := list{pk, msk, ID1, ID2}

pk := expand{gG1, gG2, g1, g2G1, g2G2, h1G1, h2G1}

msk := expand{g2alpha}


r1 := random(ZR)

r2 := random(ZR)

d1 := ((g2alpha * \

      (((g1^ID1) * h1G1)^r1)) * (((g1^ID2) * h2G1)^r2))

d2 := (gG2^r1)

d3 := (gG2^r2)

sk := list{d1, d2, d3}


output := sk

END :: func:keygen


BEGIN :: func:encrypt
```

```
input := list{pk, M, ID1, ID2}

pk := expand{gG1, gG2, g1, g2G1, g2G2, h1G1, h2G1}


s := random(ZR)

C1 := ((e(g1,g2G2)^s) * M)

C2 := (gG2^s)

C3 := (((g1^ID1) * h1G1)^s)

C4 := (((g1^ID2) * h2G1)^s)

ct := list{C1, C2, C3, C4}


output := ct

END :: func:encrypt



BEGIN :: func:decrypt

input := list{pk, sk, ct}

pk := expand{gG1, gG2, g1, g2G1, g2G2, h1G1, h2G1}

ct := expand{C1, C2, C3, C4}

sk := expand{d1, d2, d3}


M := (C1*((e(C3,d2) * e(C4,d3))/e(d1,C2)))
```

```
output := M

END :: func:decrypt
```

Appendix E

Proof Sketch of Security for Our Basic System

We now provide a sketch of the proof of security for our basic distributed anonymous credentials system.

Our basic approach is to show that for every real-world adversary $\mathcal{A}$ against the credential system, we can construct an ideal-world adversary $\mathcal{S}$ against the ideal-world system such that the transcript of $\mathcal{A}$ interacting with the real system is computationally indistinguishable from the transcript produced by $\mathcal{A}$ interacting with $\mathcal{S}$. We assume a static corruption model in which the adversary controls some set of users and leave a proof in the adaptive corruption model for future work. For this sketch we also assume that our zero-knowledge signatures of knowledge include an efficient extractor and simulator and that the *params* are created using a trusted setup process. Note

that in the random oracle model this assumption holds for the Fiat-Shamir proofs we employ, provided we conduct the proofs sequentially.

Our proof assumes the existence of a global, trusted transaction ledger, which we use as a black box. We leave a complete proof that considers this construction and models it to future work.

We begin by sketching the simulator $\mathcal{S}$ for our system.

E.1 Description of the Simulator

Minting a credential. When a user controlled by the adversary with nym $Nym_{\mathcal{U}}^{O}$ wants a credential, the user first generates $(c, \pi_M, attrs)$. When the simulator receives notification of this, it first verifies that the credential and proof are valid and meet the organization's policy. If so it employs the knowledge extractor for the signature of knowledge on π_M to obtain (sk, aux).

The simulator then checks if it has a record of $(\mathcal{U}, sk, Nym_{\mathcal{U}}^{O})$ on its list of users. If the user with key sk and nym $Nym_{\mathcal{U}}^{O}$ exists, then $\mathcal{S}$ retrieves $sk_{\mathcal{U}}$ associated with $(\mathcal{U}, sk, Nym_{\mathcal{U}}^{O})$ and proceeds. If it is not on the list, the simulator checks if it has previously seen a user with key sk. If the user with key sk is not present, then the simulator creates a user $\mathcal{U}$ and runs $RegNym(Nym_{\mathcal{U}}^{O}, \mathcal{U}, O)$ to register $Nym_{\mathcal{U}}^{O}$ and obtain $sk_{\mathcal{U}}$ for further interactions with T_P. $\mathcal{S}$ then stores $(\mathcal{U}, sk, sk_{\mathcal{U}}, Nym_{\mathcal{U}}^{O})$ in its list of users controlled by the adversary. If a user $\mathcal{U}$ with key sk exists, then it runs

$RegNym(Nym_{\mathcal{U}}^{O},\mathcal{U},O)$ to register $Nym_{\mathcal{U}}^{O}$ and adds $Nym_{\mathcal{U}}^{O}$ to $\mathcal{U}$'s record.

Once the simulator has registered the nym or verified it already exists, it runs $MintCred(Nym_{\mathcal{U}}^{O},O,attrs,aux)$. The simulator then transmits the credential information to the trusted store and acknowledges the credential's issuance. $\mathcal{S}$ stores $(sk,Nym_{\mathcal{U}}^{O},attrs,aux,c,\pi_M)$ in its list of granted credentials.

When an honest user, through T_P, wants to establish a credential, the simulator creates a credential c (using the publicly available $attrs$) and uses the simulator for the signature of knowledge π_M to simulate the associated proof. It then transmits the credential information $(c,\pi_M,attrs)$ to the trusted store.

Showing a credential. When a user controlled by the adversary wants to show a credential from organization O to verifier V with which it has nyms $Nym_{\mathcal{U}}^{O}$ and $Nym_{\mathcal{U}}^{V}$ respectively, the user first generates π_S. When the simulator receives notification of this, it verifies the proof as in the real protocol (rejecting if it is invalid). If the show verifies, it runs the knowledge extractor for the proof of knowledge on π_S to get sk.

The simulator then checks if it has a record of $(\mathcal{U},sk,Nym_{\mathcal{U}}^{O},Nym_{\mathcal{U}}^{V})$ on its list of users. If the user with key sk and nyms $Nym_{\mathcal{U}}^{O}$ and $Nym_{\mathcal{U}}^{V}$ exists, then $\mathcal{S}$ retrieves $sk_{\mathcal{U}}$ associated with $(\mathcal{U},sk,Nym_{\mathcal{U}}^{O})$ and proceeds. If the record does not exist, either in part or in full, the simulator checks if it has previously seen a user with key sk. If the user with key sk is not present, then the simulator creates a user $\mathcal{U}$ and runs $RegNym(Nym_{\mathcal{U}}^{O},\mathcal{U},O)$ and $RegNym(Nym_{\mathcal{U}}^{V},\mathcal{U},V)$ to register $Nym_{\mathcal{U}}^{O}$ and $Nym_{\mathcal{U}}^{V}$ and obtain $sk_{\mathcal{U}}$ for further interactions with T_P. $\mathcal{S}$ then stores $(\mathcal{U},sk,sk_{\mathcal{U}},Nym_{\mathcal{U}}^{O},Nym_{\mathcal{U}}^{V})$

in its list of users controlled by the adversary. If a user $\mathcal{U}$ with key sk exists, then it runs $RegNym(Nym_{\mathcal{U}}^{O}, \mathcal{U}, O)$ (resp. $RegNym(Nym_{\mathcal{U}}^{V}, \mathcal{U}, V)$) to register $Nym_{\mathcal{U}}^{O}$ (resp. $Nym_{\mathcal{U}}^{V}$) and adds $Nym_{\mathcal{U}}^{O}$ (resp. $Nym_{\mathcal{U}}^{V}$) to $\mathcal{U}$'s record.

Now, the simulator $\mathcal{S}$ runs $ShowOnNym(Nym_{\mathcal{U}}^{O}, Nym_{\mathcal{U}}^{V}, O, V, \mathbf{C})$ where $\mathbf{C}$ is obtained by the simulator through a call to $GetCredList(O)$.

When an honest user (through T_P) wants to show a credential to a verifier V controlled by the adversary, the simulator generates a random prime commitment and runs the zero-knowledge simulator for π_S to simulate a proof that it then sends to V.

E.1.1 Proof (sketch) of a Successful Simulation

Our simulation is computationally indistinguishable from the real protocol if the Strong RSA and the Discrete Logarithm assumptions hold. While we do not provide a full proof here due to space reasons, we provide an overview of the argument for security.

We first begin by discussing the signatures/proofs π_M and π_S. Under the Discrete Logarithm assumption, π_M is a computational zero-knowledge signature of knowledge on aux of the values sk, r, and r' such that the nym $Nym_{\mathcal{U}}^{O}$ and the credential c both belong to the same master secret sk. The proof is constructed using standard techniques in the random oracle model [94], and the resulting proofs are (at least) computationally zero knowledge. An attacker who forges this proof to spend a new coin would violate the soundness guarantee of the proof system. Alternatively, an

attacker might forge this message by identifying a collision on the commitments, which occurs with negligible probability under the Discrete Logarithm assumption [107]. In the event that this occurs, we can use the extractor for the zero knowledge proof to obtain the collision with all but negligible probability.

Under the Strong RSA and Discrete Logarithm assumptions, π_S is a statistical non-interactive zero-knowledge proof of knowledge of the values sk, ω, c, $Nym_{\mathcal{U}}^{V}$, r, and r' such that ω is a witness that c is in the accumulator A and nym $Nym_{\mathcal{U}}^{V}$ and the credential c both belong to the same master secret sk. This proof is again constructed using standard techniques [94, 69] similar to the proofs used by Miers et al. in [82]. In order to forge such a proof, the adversary would need to either find a collision on the commitments or forge an accumulator membership proof. We previously discussed how the first case occurs with negligible probability. The second case occurs with negligible probability under the Strong RSA assumption due to [69]. See the full version of the paper for a formal treatment/reduction of these statements.

Intuitively, we can now see that the simulator will fail with at most negligible probability because it deals solely with zero-knowledge signatures of knowledge and zero-knowledge proofs of knowledge, which have efficient extractors and simulators. Our proofs π_M and π_S have knowledge extractors that succeed with probability $1-\nu(\lambda)$ for some negligible function $\nu(\cdot)$. Since signatures and proofs are the sole point of failure for our simulator described above, it fails with negligible probability. Because the adversary only sees the simulated zero-knowledge proofs and signatures, and the

simulated signatures and proofs are computationally indistinguishable from legitimate ones, the adversary cannot distinguish a simulated transcript from the real protocol except with negligible advantage. Hence the adversary cannot distinguish between an interaction with the simulator and the real protocol.

We note that the Pedersen commitments we use are non-standard in that we output only commitments that are prime. We stress that these commitments remain information theoretically hiding and computationally binding under the assumption that the Discrete Logarithm assumption holds in $\langle g \rangle$.

Vita

Christina Garman received the Bachelor of Science in Computer Science and Engineering and Bachelor of Arts in Mathematics from Bucknell University in 2011 and completed the Masters of Engineering in Computer Science from Johns Hopkins in 2013. Her research interests focus largely on practical and applied cryptography. More specifically, her work has focused on the security of deployed cryptographic systems from all aspects, including the evaluation of real systems, improving the tools that we have to design and create them, and actually creating real, deployable systems. Some of her recent work has been on the weaknesses of RC4 in TLS, cryptographic automation, decentralized anonymous e-cash, and decentralized anonymous credentials. She is also one of the co-founders of ZCash, a startup building a cryptocurrency based on Zerocash. Her work has been publicized in The Washington Post, Wired, and The Economist, and she received a 2016 ACM CCS Best Paper Award.

Starting in January 2018, Christina will join the faculty at Purdue University as

VITA

an Assistant Professor.