6-3-2004

Greenpass Client Tools for Delegated Authorization in Wireless Networks

Nicholas C. Goffee
Dartmouth College

Dartmouth Computer Science Department Technical Report TR2004-509

Greenpass Client Tools for Delegated Authorization in Wireless Networks

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Master of Science

in

Computer Science

by

Nicholas C. Goffee

DARTMOUTH COLLEGE

Hanover, New Hampshire

June 3, 2004

Abstract

Dartmouth's Greenpass project seeks to provide strong access control to a wireless network while simultaneously providing flexible guest access; to do so, it augments the Wi-Fi Alliance's existing WPA standard, which offers sufficiently strong user authentication and access control, with authorization based on SPKI certificates. SPKI allows certain local users to *delegate* network access to guests by issuing certificates that state, in essence, "he should get access because I said it's okay." The Greenpass RADIUS server described in Kim's thesis [55] performs an authorization check based on such statements so that guests can obtain network access without requiring a busy network administrator to set up new accounts in a centralized database. To our knowledge, Greenpass is the first working delegation-based solution to Wi-Fi access control.

My thesis describes the Greenpass client tools, which allow a guest to introduce himself to a delegator and allow the delegator to issue a new SPKI certificate to the guest. The guest does not need custom client software to introduce himself or to connect to the Wi-Fi network. The guest and delegator communicate using a set of Web applications. The guest obtains a temporary key pair and X.509 certificate if needed, then sends his public key value to a Web server we provide. The delegator looks up her guest's public key and runs a Java applet that lets her verify her guests' identity using visual hashing and issue a new SPKI certificate to him. The guest's new certificate chain is stored as an HTTP cookie to enable him to "push" it to an authorization server at a later time. I also describe how Greenpass can be extended to control access to a virtual private network (VPN) and suggest several interesting future research and development directions that could build on this work.

Acknowledgments

First, I would like to thank Dr. Sean Smith for his support and encouragement throughout my time at Dartmouth College. I deeply appreciate not only his intelligence, knowledge, and willingness to provide guidance for my research, but also his sense of humor and his enthusiasm for both information security research and topics altogether far afield from computer science.

My gratitude goes out to the other members of my thesis committee, Drs. David Kotz and Edward Feustel, as well. I sincerely appreciate the comments they have offered regarding my research and writing as well as the patience they exhibited as I prepared this rather lengthy document.

Many thanks are also due to the other members of the Greenpass team: Sung Hoon Kim, Kimberly Powell, Punch Taylor, Kwang-Hyun Baek, Meiyuan Zhao, and John Marchesini. I would also like to thank other members of the Dartmouth PKI Lab for answering questions and offering suggestions throughout the past year, and other friends here at Dartmouth for the help they have offered.

I would also like to thank past mentors at the College of Wooster: Drs. Denise Byrnes and Amon Seagull in the Department of Computer Science, and Drs. Peter Mowrey and Jack Gallagher in the Department of Music. It is only with their help, particularly in the areas of research and writing, that I have been able to come this far.

Finally, the most sincere thanks to my immediate and extended family, who have always been true believers in higher education and who have supported and encouraged my intellectual journey for many years.

This research has been supported in part by Cisco Corporation, the Mellon Foundation, NSF (CCR-0209144), AT&T/Internet2 and the Office for Domestic Preparedness, Department of Homeland Security (2000-DT-CX-K001). Special thanks are due to Graham Holmes and Krishna Sankar at Cisco Corporation for arranging to provide funding and equipment for the Greenpass project. This paper does not necessarily reflect the views of the sponsors.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Problem statement 5

1.3 My contribution 6

1.4 Organization of this thesis 7

1.5 Style and formatting of this thesis 8

1.6 Related papers 8

2 Background **10**

2.1 Wi-Fi authentication standards 10

2.1.1 Preliminary definitions 11

2.1.2 WEP 12

2.1.3 WPA 13

2.1.4 IEEE 802.1x 13

2.2 Certificate-based authentication: TLS 16

2.2.1 Preliminary definitions 17

2.2.2 X.509 name certificates 18

2.2.3 CA hierarchies and trust models 20

2.2.4 TLS authentication 22
2.2.5 OS/browser keystores 24
2.2.6 EAP-TLS revisited 26
2.3 Certificate-based authorization: SPKI/SDSI 27
2.3.1 Authorization 27
2.3.2 SPKI/SDSI motivation and history 29
2.3.3 SPKI authorization certificates 30
2.3.4 SPKI delegation 31
2.3.5 The SPKI certificate format 32
2.3.6 SPKI certificate chains 34
2.4 The Greenpass RADIUS server 36
2.4.1 SPKI-based access control: motivation 36
2.4.2 “Pure” SPKI-based access control 37
2.4.3 SPKI-based authorization with EAP-TLS authentication 39
2.4.4 VLAN switching 40
2.4.5 Final clarifications 42
2.5 Summary 43

3 Greenpass client tools **45**
3.1 Requirements 45
3.1.1 Guest requirements 46
3.1.2 Delegator requirements 47
3.2 Design choices 48
3.2.1 Key and certificate transmission 48
3.2.2 Delegation 51
3.2.3 Guest introduction and authentication 53

3.2.4 SPKI certificate storage 54
3.3 Design 56
3.4 Implementation 59
3.4.1 Web applications: common functionality 59
3.4.2 Guest introduction 61
3.4.3 Delegation 64
3.4.4 The Greenpass front page 67
3.5 Summary 72

4 Related work **74**
4.1 Wi-Fi guest access 74
4.1.1 Captive portals 75
4.1.2 802.1x-based solutions 77
4.1.3 SPKI-based and related solutions 79
4.2 SPKI/SDSI 83
4.2.1 SPKI/SDSI-based access control 83
4.3 Other authorization and delegation systems 85
4.3.1 X.509 attribute certificates 86
4.3.2 X.509 proxy certificates 87
4.3.3 PERMIS 88
4.3.4 KeyNote 89
4.3.5 XML-based authorization 90
4.4 Introduction and visual hashing 93

5 Future work **97**
5.1 Security considerations 98

5.2 Usability issues . . . 101
5.3 Decentralization . . . 103
5.4 Advanced SPKI/SDSI features . . . 106
5.4.1 Advanced SPKI tags . . . 106
5.4.2 SDSI names . . . 107
5.4.3 Threshold subjects . . . 110
5.5 SPKI-based access control to other resources . . . 111
5.6 Authorization language comparison . . . 112
5.7 Other thoughts . . . 113

6 Summary **118**

A Greenpass and VPNs **123**
A.1 Definitions . . . 123
A.2 Motivation . . . 125
A.3 Experiment . . . 126
A.4 Results . . . 127
A.5 Future work . . . 128

Glossary **130**

Bibliography **141**

List of Figures

2.1 IEEE 802.1x authentication 15
2.2 An X.509 name certificate 19
2.3 Carl Ellison's authorization triangle 31
2.4 A SPKI/SDSI public key and its hash 33
2.5 A SPKI authorization certificate 34
2.6 Purely SPKI-based access control to a Wi-Fi network 38
2.7 X.509-based access control to a Wi-Fi network 40
2.8 Hybrid SPKI- and X.509-based access control to a Wi-Fi network 41

3.1 The Greenpass delegation process 49
3.2 Example of a *Visprint* visual fingerprint 55
3.3 Guest introduction Web application (screenshot) 62
3.4 Delegation applet (screenshot) 66
3.5 Re-authorization 71

Chapter 1

Introduction

1.1 Motivation

Wi-Fi (IEEE 802.11) wireless network access is basically ubiquitous here at Dartmouth College, and more and more organizations are adopting it in order to enjoy its nearly effortless connectivity. As this happens, its potential privacy and security pitfalls have moved into the spotlight, with industry players rushing to provide working security standards. Standards that simply encrypt Wi-Fi traffic and restrict access to authorized users, however, are only part of the solution: an organization also needs a flexible way to control just *who* is an "authorized user" without negating the many benefits that wireless networks can provide, such as hassle-free access for legitimate visitors. Dartmouth's *Greenpass* project seeks to provide strong access control to a Wi-Fi network while simultaneously providing flexible guest access. This thesis describes one major component of Greenpass, namely its client tools.

The rapid and widespread adoption of Wi-Fi technology in recent years, despite an overall slump in the technology market, makes it clear that its hassle-free connectivity is in

high demand. Universities, businesses, and other organizations who provide wireless LAN service, however, are justifiably concerned about the privacy and security implications of "drive-by networking." The accessibility of a Wi-Fi network's physical layer makes it almost trivially easy for an outsider to eavesdrop on unencrypted data in transit on that network or to connect to it without permission. Access to a "tethered" network often depends, implicitly, on the physical boundaries of that network: it would be easier (i.e., less conspicuous) to break into resources on a wired network by walking into a building and using a machine already on the network than by carrying in a new machine, so security systems for wired networks often focus on securing who accesses machines *already* connected to the network. (The NFS distributed filesystem, or subscription-based digital library services that are restricted to users within a certain IP-address range, are classic examples of such systems.) Wireless networks reverse this situation by removing nearly all barriers to physical access, which might explain why the original 802.11 standard defines an encryption method that attempts to achieve the rather hazy goal of "wired equivalent privacy" (WEP).

Current and future wireless security standards such as WPA [111] and 802.11i, therefore, provide cryptography-based solutions to both encryption and user authentication. Authentication can be used to restrict access to a wireless LAN to only known users, which an organization may wish to do for numerous reasons. The most obvious motivation is the need to restrict access to sensitive or licensed resources and data available over the network; an organization may also, however, wish to avoid the extra cost of providing network resources and bandwidth to unauthorized users, to avoid a credibility or liability hit that might be incurred should an outsider use the network to launch a direct attack, virus, or spam, and/or to block users who have not installed critical security patches.[1] (Even free community wireless networks projects feel the need to use access control to protect

[1] Some language in this sentence is derived from a technical report [98] and a conference paper [42] I co-authored.

their networks from abuse; a page on the Personal Telco Web site [96] discusses the risks involved.)

Although basic, proven authentication and encryption capabilities must form the basis of a wireless security solution, restricting access is only part of the solution: organizations must also ask themselves, "to *whom* should access be restricted?" The same effortless accessibility that makes a wireless network attractive to hackers also makes it attractive to an organization's legitimate guests who might find it convenient to access the Internet and other network resources during their visits. Some networks carry extremely sensitive traffic (e.g., that which is critical to national security) that requires a highly regimented, inflexible access-control policy; organizations such as universities and enterprises, however, are under tremendous social pressure to offer at least an Internet gateway to their guests as a basic amenity. One need only attend a conference, visit a hotel lobby, or enter a wireless "hotspot" at a cafe to see that this is the case. An article in the May 2003 *Communications of the ACM* [95] notes this trend and predicts that it will continue to its logical conclusion:

> By the fall of 2002, there was a good chance that anytime you walked onto a university campus[2] or into a large office building, its owner would be offering free Wi-Fi connectivity as a basic amenity (*p. 48*).
>
> Over the coming years as people experience free Wi-Fi in their homes and offices, on college campus, at conference centers, and in public parks, they will inevitably come to expect the same easy and cost-free connectivity everywhere (*p. 51*).
>
> Put simply, wireless networks want to be free. By "free," we mean that homes, offices, and public spaces will increasingly be expected to provide hassle-free

[2] My advisor reports that, as he has traveled to various universities, he has found that many, in fact, do *not* offer painless guest access. Perhaps they have grown more aware of the security issues involved since 2002.

wireless bandwidth (*p. 49*).

Further evidence of pressure in this direction can be found in the recent announcement by the Pittsburgh airport that it will move from a pay-per-use Wi-Fi access model to free access.

Traditional access-control methods—including those used in current Wi-Fi standards—do not adequately accommodate the full potential of wireless networking. To accommodate the way people *want* to (and, knowing people, will) use WLANs, a security solution must be able to support a rapidly-changing user population whose access privileges may be ephemeral and must be granted according to real-life, real-time needs. While a central administrator tool controlled by one or at most a handful of employees may provide an adequate way to grant permissions to a changing client population at, say, a coffee shop or other small business, it quickly becomes a bottleneck in any kind of large network that supports a complex organizational structure.

Dartmouth's *Greenpass* project seeks to remedy this situation by letting authorized users carry collections of SPKI/SDSI *authorization certificates* [29, 31, 32] proving that they have permission to access a particular wireless network. These credentials also grant certain users the right to *delegate* network access permissions to others. Our project has initially focused on a simple delegator/guest model where certain privileged local users can delegate temporary access privileges to visitors; hierarchies of SPKI/SDSI certificates, however, can express a variety of authorization scenarios in order to meet organizations' diverse needs.

This thesis describes the set of software tools that allow the Greenpass delegation process to take place. This process first requires that a guest have or obtain a public key with which to identify himself (authenticate). A local user who is authorized to delegate network access permissions must then bind his mental identification of the guest to that public key,

which serves as his guest's identity in the digital world. The local user can then sign and issue a SPKI/SDSI certificate that propagates his own access permission to the guest. By combining SPKI/SDSI delegation to raw public keys with easy, visual-hash-based methods of introducing users without a trusted third party, this research allows guests to join a protected network without a pre-existing agreement between two organizations' CAs (*certification authorities*) or other authentication systems.

1.2 Problem statement

The goal of the Greenpass client tools is to allow a local user who has permission to act as a *delegator* to temporarily allow a guest onto the network by issuing a SPKI certificate that electronically states, in essence, "I said it's okay." In addition, the tools need to meet the following requirements:

- *No pre-existing PKI*: since PKI (*public-key infrastructure*) technology has not been as widely deployed as its advocates would like, the tools must not rely on a pre-existing digital agreement between the local organization (the owner of the wireless network) and the guest's organization. In particular, we cannot assume the existence of a "bridge CA" to aid in authenticating the guest: we need to provide a secure way for the delegator to assert, in essence, "Yes, that's him (or her)." Note that the goal of Greenpass is to allow users access to the *inside* resources of a Wi-Fi network, not to restrict them to an Internet gateway only.

- *Interoperability with current standards*: the tools must be made guest-friendly by using technologies already installed on platforms we can reasonably expect most visitors to bring. Since our initial focus was a university setting, we had to accommodate a more diverse range of platforms than an enterprise might.

1.3 My contribution

My role in the Greenpass project has been, at the highest level, to design and implement a set of tools that allow a local user (a *delegator*) to issue a credential that grants a guest temporary access to the local wireless network. Specifically, the tools and enabling technologies I have implemented include the following components:

- A set of three ***Greenpass Web applications*** that provide a cross-platform interface to the other components. A front page Web app provides a rendezvous point and set of utilities for all users; a guest introduction Web application allows a guest to transmit his public key value—subsequently used as a unique identifier—to a delegator; and a delegation Web app allows a delegator to look up a guest's public key value and send it to our delegation applet.

- A ***delegation applet*** that allows the delegator to verify that the public key value she receives really belongs to her intended guest, and generate and sign an authorization certificate that delegates Wi-Fi access privileges to that guest.

- A number of daemons: a ***dummy CA*** that issues new X.509 certificates to guests who do not have them, an ***introduction cache*** that stores the public keys of new guests who are waiting for delegators to authorize them; and an ***authorization cache*** that stores newly-created credentials and which Greenpass's modified RADIUS server queries to make its final access-control decision granting or denying network access to a particular user.

The goal of these tools is to demonstrate not only that SPKI/SDSI and similar authorization certificate formats provide a sufficient way to distribute wireless access privileges among a dynamic client population, but that such a system need not require new authentication protocols or custom client software.

1.4 Organization of this thesis

The rest of this thesis is organized as follows:

- Chapter 2 discusses background material related to my work. It explains existing Wi-Fi authentication mechanisms, focusing on how the widely-used TLS handshake has been adapted to Wi-Fi authentication; explains the use of and client platform support for X.509 certificates in TLS; introduces the SPKI/SDSI PKI model and SPKI authorization certificates; and finally, describes how the Greenpass RADIUS server described by Kim [55] enables the use of SPKI authorization on a Wi-Fi network.

- Chapter 3 discusses the core of my work: the set of tools that enable a newly-arrived guest to introduce himself to an authorized local delegator, and the delegator to in turn issue an authorization certificate granting the guest access to the wireless network.

- Chapter 4 discusses work that is related to my own, including other enhancements to wireless security (including guest access), other projects using SPKI/SDSI for authorization, alternative certificate and policy formats that compete with SPKI/SDSI, and work related to secure introductions and visual hashing.

- Chapter 5 suggests future work related to Greenpass, including both project development ideas and basic research opportunities.

- Chapter 6 offers a summary of this thesis and some concluding remarks.

- Appendix A describes a recent experiment that enables SPKI-based access control to a VPN.

1.5 Style and formatting of this thesis

Throughout this thesis, I use the word "I" to refer to myself when discussing work I have done mostly unaided. I use the word "we" to refer to myself and others on the Greenpass project when discussing work that involved a fair amount of collective design or implementation. I occasionally use "we" to refer to myself and the reader in order to maintain a perception of discussion, as in "we have previously seen…". I will make every attempt to make the latter use obvious from context.

I introduce most new terms by writing them in *italics*. Where appropriate, I emphasize terms that refer to components of the client tools I have built by writing them in ***bold italics***.

I sometimes use the term *owner* of a public key to refer to the entity who holds the corresponding private key. The word *owner* accurately distinguishes the rightful user of a key pair from somebody who merely knows its public half, just as the owner of a driver's license is different from somebody who might hold it in his hand but does not possess the corresponding face.

1.6 Related papers

Two other student theses describe work related to Greenpass. Kim's Master's thesis [55] discusses the Greenpass RADIUS tools, which I introduce in Section 2.4, in much greater detail. Powell's undergraduate thesis [87] discusses a pilot test of Greenpass with several users and two different operating systems.

Our research group has also written a number of shorter papers related to Greenpass. The department technical report by Smith et al. [98] was our first public description of Greenpass, and Goffee, Kim, Smith et al. [42] is forthcoming in the proceedings of the 3rd Annual PKI R&D Workshop. Baek, Smith, and Kotz [5], also forthcoming, provides

a detailed survey of Wi-Fi security technologies. Some language from papers on which I was a co-author may appear in this thesis. Chapter 3 in particular is adapted and expanded from earlier work; otherwise, I will try to mention in footnotes when sections of this thesis contain derivative work.

Chapter 2

Background

This chapter provides background material related to my work. Section 2.1 provides a brief history of authentication standards for 802.11 wireless LANs, focusing on how the widely-used TLS (formerly SSL) handshake has been adapted to Wi-Fi authentication. Section 2.2 describes the X.509 name certificate standard and expands on how TLS uses it to authenticate users. Section 2.3 introduces SPKI/SDSI authorization certificates and explains why they provide a suitable model for expressing permissions for the dynamic user base of a Wi-Fi network. Finally, Section 2.4 introduces the Greenpass RADIUS server, which authenticates wireless clients using an unmodified EAP-TLS handshake, but bases its final access-control decisions on SPKI/SDSI certificate chains.

2.1 Wi-Fi authentication standards

Greenpass augments existing Wi-Fi security protocols rather than introducing its own. The original Wi-Fi standard provided authentication and privacy via a shared-secret system called *WEP* (*Wired Equivalent Privacy*). WEP provides no key distribution mechanism, relying instead on manual entry of keys; furthermore, its authentication and privacy mecha-

nisms are fundamentally broken. Newer standards such as the Wi-Fi Alliance's *WPA* (*Wi-Fi Protected Access*) and *IEEE 802.11i* provide stronger encryption and multiple authentication options, including the public-key-based TLS handshake. For a thorough discussion of Wi-Fi security, the reader should consult Edney and Arbaugh [25].

2.1.1 Preliminary definitions

A few preliminary definitions must be clarified before proceeding with this section:

- *Wi-Fi* (*Wireless Fidelity*) is simply an informal name for the IEEE 802.11 wireless networking standard [20]. It also gives the Wi-Fi Alliance, an industry consortium of 802.11 vendors, its name.
- IEEE 802.11 defines two modes of operation:
  - In *ad-hoc mode*, mobile wireless stations (i.e., client devices) communicate directly with one other using radio waves, just as nodes of a tethered LAN communicate directly with one another using electrical signals on a wire.
  - In *infrastructure mode*, on the other hand, a specialized *access point* (*AP*) provides central LAN service to one or more mobile client stations in its vicinity. An AP typically acts as a bridge to a wired network, giving its clients access to resources on a larger LAN or, via a gateway, a WAN or the Internet.
- A client station must *associate* with an AP before exchanging data messages with it. Association does not necessarily allow the client to exchange messages with other clients or with the wired LAN the AP provides access to: the AP may require a more complex authentication handshake after a client associates but before it receives full network access.

- To associate with an AP, a client station must know the *SSID* for that AP. An SSID is simply a human-readable "network name" assigned to an AP or group of cooperating APs to help humans distinguish their service from that of other, nearby Wi-Fi networks. An AP may choose to broadcast its SSID or keep it "secret"; not broadcasting it, however, provides no real protection against unauthorized access.

2.1.2 WEP

A mobile device and an access point using WEP share a secret key which they use to encrypt and checksum their communications. The shared secret can be any 40-bit string, but many APs and client tools allow users to type passwords from which 40-bit keys are derived. (According to Edney and Arbaugh [25], the Wi-Fi Alliance also supports the use of "128-bit" encryption, which uses a 104-bit key plus a 24-bit initialization vector.)

WEP provides two modes of operation: *default key mode* and *key-mapping key mode*. In default key mode, all wireless stations use the same key to communicate with the access point (meaning they can potentially decrypt and eavesdrop on one another's traffic). This mode is sufficient for homes or smaller organizations (or would be if WEP provided stronger encryption; see below). On the other hand, each client can be assigned its own key-mapping key. This mode prevents clients from eavesdropping on one another (and prevents a single key compromise from endangering every client's privacy), but requires every access point to maintain a table that maps each client's MAC address to that client's key.

Unfortunately, WEP has fundamental design flaws that prevent it from providing adequate privacy, authentication, or access control. Borisov, Goldberg, and Wagner [14] discuss these flaws in detail, as do Edney and Arbaugh [25]. In addition, WEP's dependence on shared secrets, without providing any particular method for their distribution, does not offer the same flexibility as a public-key-based system would.

2.1.3 WPA

The IEEE 802.11i draft addendum to the 802.11 standard seeks to rectify the problems with WEP and establish an adequate security standard for Wi-Fi networks. Until 802.11i moves from a draft to a final, ratified IEEE standard, however, wireless vendors need an intermediate security standard; to this end, the Wi-Fi Alliance has extracted a subset of 802.11i and published it as *WPA* (*Wi-Fi Protected Access*) [111].

WPA strengthens WEP's encryption and session authentication using *TKIP* (*temporal key integrity protocol*), which works with existing WEP hardware but cleverly adds session keys and rotating per-packet keys that are derived from, but different than, the client/AP master shared key. TKIP also augments WEP with a stronger MIC (message integrity code) and a longer initialization vector (48 bits). Eventually, the more general *RSN* (*robust security network*) standard defined in 802.11i will supersede TKIP, but the initial authentication and session-key generation procedure, IEEE 802.1x, will remain essentially the same.

2.1.4 IEEE 802.1x

On top of TKIP's stronger session integrity, WPA adds a general authentication and access control mechanism called *IEEE 802.1x* [21]. IEEE 802.1x applies to more than just wireless networks: it defines a general access control mechanism for all LANs in the IEEE 802 (i.e., Ethernet) family. The 802.1x standard identifies three entities, shown in Figure 2.1, that take part in the authentication process:

- A *supplicant* requests access to some network or network resource. In a Wi-Fi environment, the mobile client device plays the role of supplicant.

- To request access, the supplicant connects to an *authenticator*, a "door guard" that has the power to grant or deny access to some network service. A Wi-Fi access point

provides a virtual authenticator for each supplicant that tries to connect to it.

- The authenticator need not store its own database of users and their authentication credentials; rather, it can relay the supplicant's handshake messages to an *authentication server*.[1] The AP need not understand the entire sequence of authentication messages: it need only set up the authentication session, relay intermediate messages, and respond to a "success" or "failure" message sent to it from the authentication server.

Typically, multiple authenticators rely on a shared authentication server in order to avoid the maintenance of multiple authentication databases. It is entirely possible, however, for the authenticator and authentication server to be co-located.

An 802.1x authentication handshake involves three layers, also shown in Figure 2.1:

- The innermost layer consists of the actual authentication handshake that takes place between the supplicant and the authentication server (supported handshake types are discussed below). The authenticator need not understand this handshake; it merely relays messages between the two endpoints.

- The supplicant and authentication server encapsulate their handshake in a packet format called *EAP* (*Extensible Authentication Protocol*) [13], shown as the middle layer in Figure 2.1. EAP was originally designed to allow PPP dialup clients to authenticate to a central authentication server so that ISPs would not need to maintain a duplicate user database at each modem pool. The authenticator must understand enough of the EAP packet format to know what to do with various EAP packets: it relays most packets, but must understand the EAP packet the supplicant uses to initiate

[1] The 802.1x standard calls this entity an "authentication" server, but the name is misleading: nothing stops it from carrying out authorization checks after it authenticates the supplicant. In fact, the Greenpass authentication server does just that, as we shall see.

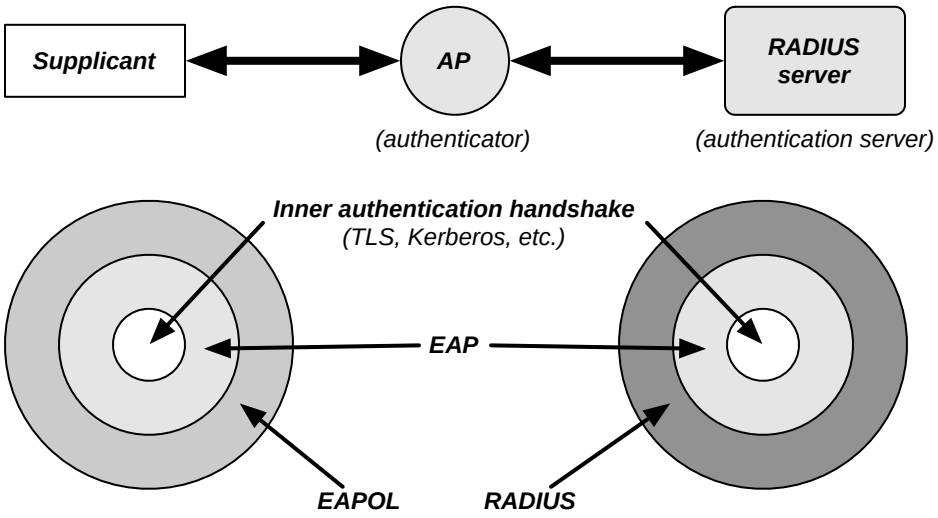

Figure 2.1: An illustration of IEEE 802.1x authentication. In 802.1x, a *supplicant* requests access to a network or network resource protected by an *authenticator*. The authenticator need not store its own database of users and their authentication credentials; rather, it can relay the supplicant's handshake messages to an *authentication server*. In a Wi-Fi setting with 802.1x authentication, the mobile client acts as supplicant, the access point (AP) acts as authenticator, and a *RADIUS server* typically acts as the authentication server. At the bottom of the illustration are the layers of communication between supplicant/AP and AP/RADIUS server. The actual authentication handshake by which the supplicant proves its identity to the RADIUS server (innermost layer) is encapsulated in *EAP* packets, which the AP need understand only well enough to relay them between the other two entities. The supplicant and AP exchange EAP packets via *EAPOL* (*EAP-Over-LAN*), the the AP forwards EAP packets to and receives responses from the RADIUS server via the RADIUS protocol. (Diagram concept courtesy of Kwang-Hyun Baek.)

communication, as well as the EAP "success" or "failure" packet the authentication server will eventually use to tell it whether to grant access to the supplicant.

- Finally, the outer layer transports EAP packets between the entities. The supplicant and authenticator may use a different outer protocol than the authenticator and authentication server do, so long as the inner EAP packets remain intact. The 802.1x standard defines an EAPOL (EAP over LAN) packet format which the supplicant and authenticator use to exchange EAP packets over an Ethernet network (such as 802.11). Often, the authenticator and authentication server exchange EAP packets via the *RADIUS* (*Remote Authentication Dial-In User Service*) protocol [89]; authentication servers that use this protocol are called *RADIUS servers*.

Several inner authentication methods have been adapted for use with EAP. Among them are an MD5-based challenge-response protocol, a one-time password protocol, and an EAP-encapsulated variant of the Kerberos protocol. Of most relevance to our discussion, however, is EAP-TLS [1]. EAP-TLS defines a way to encapsulate a TLS authentication and key-exchange handshake—essentially the same as the SSL handshake between a Web browser and a secure Web server—within EAP. The next section describes TLS authentication in greater detail.

2.2 Certificate-based authentication: TLS

Public-key cryptography allows two parties to authenticate one another if they have prior knowledge of one another's public keys. It also allows entities to learn one another's public keys "on the fly" by means of *certificates*, which are essentially signed, distributed directory entries. This section describes X.509, by far the most common certificate format in use today; it also describes the TLS (formerly SSL) protocol, a proven method by which two

parties can authenticate one another using X.509 certificates and establish an encrypted tunnel around a TCP/IP socket-based connection.

2.2.1 Preliminary definitions

According to Kaufman, Perlman, and Speciner [54]:

> A *public-key infrastructure* (*PKI*) consists of the components necessary to securely distribute public keys. Ideally, it consists of certificates. . . , a repository for retrieving certificates, a method of revoking certificates, and a method of evaluating a chain of certificates from public keys that are known and trusted (*trust anchors*) to the target name.

Although this section, and the following one, do not discuss revocation in much detail, and assume certificates will be stored on each device rather than in a universally-available repository, both sections discuss PKIs. In particular, they contrast two competing certificate formats and explore the differing assumptions about PKI that are inherent in these formats themselves.

This section discusses authentication extensively. In order to avoid confusion when describing authentication procedures, I use the terms *relying party* and *target*,[2] as well as the verb *authenticate* itself, consistently throughout this section and the rest of this thesis. If Alice demands that Bob prove his identity to her, then Alice is the relying party and Bob is the target. Alice *authenticates* Bob, whereas Bob *proves his identity to* Alice.[3]

[2] I borrow this terminology from Kaufman et al. [54].

[3] I discovered that using *authenticate* and *authenticate to* to signify opposite actions was too confusing.

2.2.2 X.509 name certificates

In order for Alice to authenticate Bob using public-key cryptography, she must first know the real Bob's public key value: the party she is communicating with can then prove that he is Bob by demonstrating knowledge of the corresponding private key. If Alice does not already know Bob's public key, then she must obtain it from a third party that she trusts and that she already has some means of authenticating. She might, for example, look up Bob's name in a secure directory that lists various persons' public keys, but a central directory quickly creates a bottleneck in a large, distributed environment such as the Internet. A better alternative is to use *name certificates*, which allow the secure distribution of *name→public key* bindings.

X.509 is the dominant name certificate[4] format on the Internet; an RFC from the IETF's PKIX working group [86] defines the standard profile [46] for its use. An X.509 certificate, such as the one shown in Figure 2.2, is a message generated by the entity named in the certificate's *issuer* field that provides a public key value for the entity named in its *subject* field. The certificate's issuer, called a *certification authority* (*CA*), uses its private key to sign the certificate in order to prove that it was not forged. The certificate can then be distributed freely to anybody who wishes to authenticate its subject; even if the subject himself carries it, the signature prevents him from tampering with the information it contains.

An X.509 certificate identifies both its issuer and its subject using *X.500 distinguished names*. A distinguished name provides the common name (e.g., "Nicholas C. Goffee") of an entity along with additional information (organizational membership, location, email address, etc.) intended to make that name unique. (Making names unique, however, does not necessarily make them meaningful, as we shall discuss in Section 2.3.)

[4]PKIX standards call X.509 name certificates *public-key certificates* (*PKCs*), but I use the term *name certificate* in this thesis for consistency across different certificate formats.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 307 (0x133)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: DC=edu, DC=dartmouth, C=US, O=Dartmouth College,
CN=Dartmouth CertAuth1
        Validity
            Not Before: Jul  3 17:17:37 2003 GMT
            Not After : Jul  2 17:17:37 2004 GMT
        Subject: DC=edu, DC=dartmouth, C=US, O=Dartmouth College,
CN=Nicholas C. Goffee/emailAddress=Nicholas.C.Goffee@Dartmouth.edu
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:e4:23:06:3f:b0:48:6f:c1:94:09:62:af:c5:ee:
                    cf:ad:87:bf:7c:40:7c:0c:49:41:e3:25:d2:47:fd:
                    ...
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
            Digital Signature, Non Repudiation, Key Encipherment
            Netscape Cert Type:
            SSL Client, S/MIME
            X509v3 Subject Alternative Name:
            email:Nicholas.C.Goffee@Dartmouth.edu
            X509v3 Authority Key Identifier:
            keyid:3F:C0:D6:C7:A7:4F:00:7E:EF:06:99:67:6C:BC:96:1E:4D:A3:77:12

            Authority Information Access:
            OCSP - URI:http://collegeca.dartmouth.edu/ocsp

    Signature Algorithm: sha1WithRSAEncryption
        be:36:45:42:b7:75:96:19:82:55:e9:36:c6:48:87:0f:93:31:
        5f:48:82:b6:0e:72:41:3c:12:da:ed:4a:5c:93:d7:16:d9:b6:
        ...
```

Figure 2.2: The information in an X.509 name certificate, as decoded and displayed by the OpenSSL library; the public key modulus and signature value are abbreviated here to save space and improve readability. An X.509 certificate is a digitally-signed message generated by the entity named in the certificate's *issuer* field that provides a public key value for the entity named in its *subject* field. It also contains *version* and *serial number* fields that distinguish it from other certificates issued by the same certification authority (CA), and optionally can contain various *extensions* that give additional information about the certificate, its issuer, or its subject.

Along with the fields discussed above and some administrative fields described in the caption of Figure 2.2, an X.509 certificate may also contain *extension* fields. Often, a CA adds an extension pointing to its *certificate revocation list* (*CRL*) to all certificates it issues; maintaining a CRL allows a CA to revoke certificates that have become invalid before their expiration dates due to loss or compromise of the subject's private key. A CA might also use extensions to list various constraints on a certificate's usage, such as whether the subject is itself a CA (i.e., may issue its own certificates), for example, or whether the certificate should be used to validate its subject's signature on S/MIME emails.

2.2.3 CA hierarchies and trust models

Clearly, X.509-based authentication must involve more than just presenting an X.509 certificate and proving knowledge of the corresponding public key: the relying party must also validate the information in the certificate. The vendor, owner, or user of the relying device or program must configure it to treat one or more CAs as *trust anchors*—i.e., as CAs whose signed certificates the device or program will simply treat as true. Usually, every CA issues a self-signed *root certificate*, containing its own distinguished name and public key, that relying parties can use as a trust anchor.

A CA can certify another CA just as easily as it can certify an end-user, thereby creating a hierarchy of CAs. The primary CA is said to *delegate* to the other, subordinate CA(s). Delegation can decrease the load on the primary CA and also make enrollment (the process of obtaining a certificate) more convenient for end-users. A large corporation's CA, for example, might delegate to subordinate CAs at a number of regional offices, each of which certifies employees in its own region. Regional employees can then visit their local CA, rather than the central CA, to obtain certificates. (CAs that follow stringent policies might require a physical visit, presentation of photo ID, etc. in order to obtain a certificate.

Some organizations operate *registration authorities* (*RAs*) that verify members' identities at various locations, then send those members' verified *name→public key* bindings off to a central CA that issues their final certificates. An RA must sign its certificate requests before sending them to the CA so they cannot be modified in transit.)

In order for a relying party to make use of certificates issued by subordinate CAs in a hierarchy, it must obtain a *certificate chain* for its target rather than a single certificate. Some user Bob, for example, might have a chain of certificates $CA_1 \rightarrow CA_2 \rightarrow CA_3 \rightarrow Bob$, indicating that CA_1 delegated to CA_2, CA_2 delegated to CA_3, and CA_3 certified Bob. A relying party can then authenticate Bob even if CA_1 is the only one of the three CAs that it treats a trust anchor, because it can learn CA_2 and CA_3's public keys (needed to verify their signatures) "on-the-fly" from certificates appearing in Bob's chain. The relying part might construct Bob's certificate chain by searching for the necessary certificates in one or more online repositories (called a "pull" model, since the relying party pulls certificates from a directory), or Bob might carry his entire certificate chain on his own computer so that he can present it when challenged (called a "push" model, because Bob pushes his certificates down the authentication channel).

CA delegation potentially creates a number of hierarchical structures (and sometimes, non-hierarchical structures), which Kaufman, Perlman, and Speciner [54] call *trust models*. For example, if one organization convinced every other organization to trust its CA and let it delegate to their CAs, then the world would rely on the *monopoly* trust model (with delegation). In reality, much of the Internet world relies on an *oligarchy* (informally, a handful) of CAs. This oligarchy consists of two basic types of CA (the following slightly oxymoronic terms are, to my knowledge, my own):

- "Competing monopolies." Companies such as Verisign and Thawte make money by

certifying other organizations.[5] These CAs work best when they bind public keys to globally-known names, such as "IBM" or "Yahoo"; the public can then rely on certificates from such CAs to make sure that online parties who claim to be IBM or Yahoo really are. (In reality, these CAs often issue certificates to businesses with rather obscure names, meaning that they not only bind a name to public key, but also imply that the named entity is a trustworthy business.)

- "Local monopolies." Many organizations run their own CAs that they use to certify their own members or employees. For example, the Dartmouth College CA certifies Dartmouth students, faculty, and employees, who can then use their X.509 certificates to authenticate to Dartmouth-operated Web services such as personalized class enrollment pages. Those services only need rely on the local CA; hence, it has a local monopoly.

2.2.4 TLS authentication

TLS (*Transport Layer Security*) [3] is a standardized version of Netscape's *SSL* (*Secure Sockets Layer*) protocol. Netscape introduced SSL version 2.0 [45] as part of the Netscape Navigator Web browser (SSL version 1.0 was never released to the public), and subsequently updated SSL to version 3.0 [39]; the IETF superseded both with TLS. SSL and TLS augment TCP/IP socket-based connections with the following security features:

- *Endpoint authentication*. The handshake that begins an SSL/TLS session allows either party of a connection—server or client—to demand a proof of identity from the other party. The authentication handshake uses X.509 certificates and is designed to prevent man-in-the-middle and replay attacks.

[5] I call these "competing monopolies" because each such company's marketing department tries to convince us that theirs *is* a monopoly.

- *Secure tunneling*. The SSL/TLS handshake also allows the two connecting parties to negotiate an encrypted tunnel to protect the privacy of their ensuing socket-based communication. To do so, the two parties agree on a cryptographic algorithm (or suite of algorithms) and generate a shared secret with which to encrypt their session.

- *Session authentication*. Throughout an SSL/TLS session, both parties attach MACs (message authentication codes) to messages they send so that intruders cannot tamper with their encrypted session. Just as with the encryption key, the two parties derive a shared MAC key during their initial handshake.

Although SSL and TLS can protect any socket-based connection (and the initial handshake works in other contexts, such as EAP-TLS), their most familiar application is the HTTPS (secure HTTP) protocol used in Web browsers. Most often, a Web browser demands server authentication: it requires that the server it is connecting to first present an X.509 certificate that binds a public key to the server's domain name (e.g., *www.yahoo.com*), then demonstrate knowledge of the corresponding private key. Typically, the server's certificate comes from one of the "competing monopoly" CAs (see Section 2.2.3) such as Verisign or Thawte. SSL/TLS server authentication helps to ensure that the user of a Web browser will transmit sensitive information, such as credit card or bank account numbers, only over an encrypted channel and only to the merchant he intends to receive it.

Many newer Web browsers also support SSL/TLS *client authentication*. In this case, a Web server demands that the client present an X.509 certificate and demonstrate knowledge of the corresponding private key. (SSL/TLS also allows a server to merely request, rather than demand, client authentication.) Currently, many Web-based services that need to authenticate their clients—e.g., online banking or account-based shopping services—do so by demanding a username/password pair. SSL/TLS client authentication provides an

alternative. Client authentication often depends on a "local monopoly" CA (again, see Section 2.2.3), where the owner of the server operates its own CA to issue certificates to its clients. (This model reflects real-world practice: you probably access your bank account using a bank-issued card rather than your driver's license, even though the latter has more global scope as an authentication token.)

2.2.5 OS/browser keystores

Consumer operating system and Web browser platforms typically include *keystores*, which hold the following information for their users:

- *A list of the user's trust anchors*: those CAs the Web browser should trust when authenticating HTTPS servers. In reality, Web browsers come "from the factory" (from the download site, at least) pre-configured with the root certificates of popular (or perhaps merely persuasive) trust anchors. A user might add local CAs, or the CAs for S/MIME email users or other non-SSL uses, to his keystore.

- *A list of non-CA entities the user trusts*: these might include, e.g., Web servers the user has marked as trusted but whose CA certificates he does not have, or end-user certificates of the user's S/MIME email correspondents.

- *A list of the user's own, personal certificate(s) and their corresponding private key(s)*, for use in SSL/TLS client authentication, perhaps, or for signing or decrypting S/MIME email.

Conceptually, the term *keystore* often refers not just to a database of keys and certificates, but also to the API and logic used to manipulate them. For example, rather than treating a keystore as a database from which to read a user's private key before signing some data, a well-written application should instead treat the keystore as an object from

which it can request a "sign this data" service. This view of keystores allows an application to use cryptographic modules—such as USB tokens, smart cards, or trusted modules installed on or added to a computer's motherboard—rather than software keystores. These devices hold a user's private key and perform signing operations with it on behalf of applications: this approach prevents the private key from ever entering the computer's main memory, where a malicious program might steal it. RSA Laboratories' PKCS#11 standard [92] describes one widely-used, abstract API for communicating with "devices which hold cryptographic information and perform cryptographic functions"; Microsoft's CryptoAPI serves the same purpose on Windows machines. On the other hand, most OS and browser keystores support, at minimum, a software cryptographic module that stores certificates and keys on a local disk. The PKCS#12 standard [93] defines a widely-supported format for on-disk keystores; OS and browser keystores can typically import or export this format.

The current state of OS and Web browser keystore and EAP-TLS support is as follows:

- Microsoft Windows provides an OS-wide keystore that any application can use; in particular, both Internet Explorer and Windows' built-in EAP-TLS client can use it for SSL/TLS client authentication. The Windows CryptoAPI, mentioned above, provides support for hardware-based cryptographic modules.

- Apple's Mac OS X has provided an OS-wide keystore (*Keychain*, in their terminology) for some time, but it has only begun to mature with version 10.3 of the operating system. Like Windows, Mac OS X uses its built-in keystore for SSL/TLS client authentication and for EAP-TLS. According to Apple's Mac OS X Web site, they do support smart cards (and presumably similar devices).

- Mozilla, which is evolving into a platform in its own right, supplies a keystore using its own NSS (*Network Security Services*) library. Mozilla (and versions of Netscape

derived from it) supports SSL/TLS client authentication and S/MIME email; however, its keystore is separate from that used by the host OS for EAP-TLS. Mozilla supports cryptographic tokens and similar devices via PKCS#11. The last pre-Mozilla versions of Netscape also supported SSL/TLS client authentication.

- The Java programming language/platform supports a variety of cryptographic operations via Sun's JCE (*Java Cryptography Extension*), a set of cryptographic packages that third parties can extend by building new cryptographic providers. A number of vendors supply PKCS#11 functionality for Java, and the Swiss company Keyon also provides JACAPI [51], a CryptoAPI wrapper for Java. Java also provides its own PKCS#12-like, file-based software keystore format, JKS, which it uses internally to hold the certificates of a user's trusted code signers.

- Linux supports a rich variety of cryptographic functionality via OpenSSL, NSS, and other open-source projects, as well as all the available Java-based products. Linux desktop and system applications, however, do not share any single keystore among themselves.

2.2.6 EAP-TLS revisited

WPA uses EAP-TLS to bring all the characteristics of SSL/TLS authentication to Wi-Fi networks. TLS's PKI-based authentication offers considerably more security and flexibility than WEP's broken shared-secret authentication. In particular, it allows a CA or hierarchy of CAs, configured as the RADIUS server's trust root(s), to add users (by issuing certificates) or remove them (by expiring or revoking certificates) in a distributed fashion—i.e., without directly modifying the RADIUS server's user database. Once a supplicant authenticates using EAP-TLS, the RADIUS server uses extension fields in the RADIUS packet

format to send key material derived during the TLS handshake to the access point for use with TKIP encryption.

2.3 Certificate-based authorization: SPKI/SDSI

SPKI/SDSI [31, 32] defines a name certificate format but also an *authorization certificate* format, both of which depart from assumptions made by X.509. SPKI/SDSI excels at expressing temporary authorizations that reflect person-to-person relationships.

2.3.1 Authorization

TLS specifies an authentication method, but by itself provides no way to determine whether a user ought to be granted access after proving his identity. (As an analogy, consider whether you would let a stranger enter your home simply because he holds a valid driver's license that tells you his name.) In a simple arrangement, a particular CA might issue X.509 certificates only to users who are authorized to access a particular resource; any user who could authenticate could then be granted access. Multiple CAs could even be set up to issue authentication materials for multiple resources. Operating a CA, however, is complex and costly; instead, it might make sense for an organization to operate a single, high-assurance CA so that the administrators of various resources can specify authorized users by name.

Traditional, centralized authorization mechanisms that can be used with any type of authentication include the following:

- *Access-control lists (ACLs),* which simply list who has permission to access a given resource. In a Wi-Fi context, administrators could easily configure a RADIUS server to contain an ACL of users permitted to access the network.

- *Attributes* stored in a database and indexed by user name. An attribute could be nearly anything[6] that might be relevant to its relying party, such as "is more than 21 years old," "is a member of group *professors*," or "has permission to access directory */var/mail/bubba* on host *mail.example.com*." LDAP [114] stores information by X.500 name, and therefore provides an ideal way to retrieve attributes about a user based on the subject name in his X.509 certificate.

- *Policy languages* supply a general syntax for defining a complex set of conditions under which a user should or should not be granted access. (Attributes and policies can interact: an attribute could easily store a policy statement, and a policy might refer to a user's attributes.)

ACLs, attributes, and policies must either be stored in a protected, centralized server or distributed in a secure fashion. Centralized ACLs or attribute or policy databases, however, create bottlenecks, just as centralized authentication directories do. Taking a cue from name certificates, a number of researchers have proposed *attribute certificate* or *authorization certificate* formats with which to distribute authorization information. X.509 attribute certificates [34] are much like signed LDAP directory entries: they bind attributes to X.500 names that will be authenticated using X.509 name certificates. Assertions in a policy language can also be turned into certificates by signing them. SPKI/SDSI, on the other hand, defines a lightweight certificate format designed specifically for conveying permissions from party to party (it can be used for attributes as well). I discuss competing standards in more detail in Section 4.3.

[6] Even a name…

2.3.2 SPKI/SDSI motivation and history

Carl Ellison, one SPKI/SDSI's creators, argues in [30] that the X.509 name and attribute certificate standards are based on a number of flawed assumptions:

- X.500 distinguished names are supposedly global in nature; no global name directory, however, actually exists. In practice, each organization that relies on X.509 certificates operates a CA for its own namespace, and namespaces have not been linked as thoroughly as expected in the original X.500 plan.

- It is possible to make names globally unique (e.g., email address), but not globally meaningful. When someone looks up a name in a phone book, he is likely to find several persons with the same name. There is no guarantee that the seeker will find the additional information in an X.500 name to be meaningful—i.e., useful in distinguishing that particular person from others with the same common name.

- Global names introduce a potential security flaw. A human might generate an ACL entry or attribute certificate, or use a name certificate to make a decision, based only on the common name field of an X.500 name. (Suppose, for example, that a user scrolls through a list of his organization's employees, chooses the first one whose common name matches the person he has in mind, and assigns a permission to that person.) The common name is not necessarily globally unique, and an attacker might take advantage of the resulting gap in verification.

To circumvent these flaws, Ellison instead argues for a PKI with the following characteristics:

- *Public keys as unique identifiers.* In a well-designed cryptosystem, a user's public key value is unique. Computers can identify users by random values such as public keys just as easily as by human-readable names.

- *Local names*. Humans recognize other humans by local names. They may use nicknames or first names merely for convenience, and they may not even know the full names of everyone with whom they communicate. Even corporations and governments tend to identify customers or citizens by local "names" such as employee number, bank account number, or driver's license number. Referring to a name relative to the public key that defined it, Ellison argues, reflects the real-world use of names and avoids the pitfalls that occur when humans become confused by non-local names.

SPKI (pronounced "spooky" or "speaky") and SDSI ("sudsy") were originally separate, but similar, PKIs based on these new assumptions; they have since merged. SPKI (*Simple Public Key Infrastructure*) provides an authorization certificate format that can bind authorization permissions directly to public keys rather than to names, as we shall see. SDSI (*Simple Distributed Security Infrastructure*) [91], designed by Ron Rivest and others at MIT, allows each user to bind names to keyholders in a local namespace identified by his own public key; SDSI also defines a relative name syntax that lets a user refer to names defined by other keyholders. For the rest of this thesis, instead of writing "SPKI/SDSI," I use "SPKI" to refer to authorization certificates and "SDSI" to refer to name certificates.

At present, Greenpass relies entirely on SPKI authorization certificates, so this section does not discuss SDSI in detail. Subsection 5.4.2, however, briefly describes SDSI names and suggests how they might augment Greenpass in useful ways.

2.3.3 SPKI authorization certificates

SPKI authorization certificates bind permissions directly to public keys. Carl Ellison often uses a triangle diagram, such as that shown in Figure 2.3, to illustrate the difference between SPKI authorization and name-based authorization mechanisms.

Authorization mechanisms allow a computer to find out if a connecting user has per-

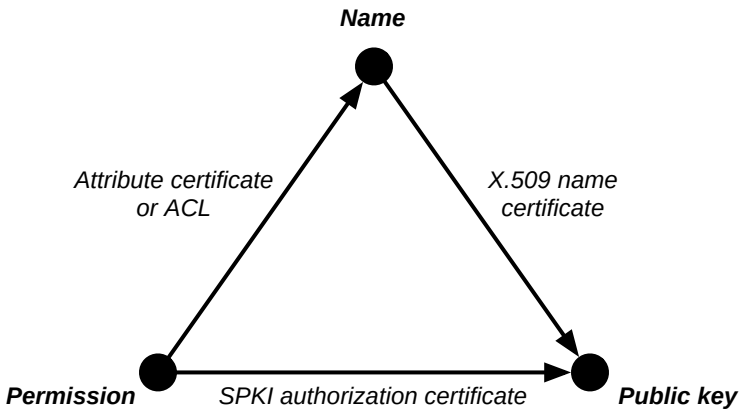

Figure 2.3: Carl Ellison's authorization triangle. X.509 name certificates contain a *name→public key* binding. A computer system that authenticates users using such a certificate must also consult an ACL (access-control list) or attribute certificate that contains a *permission→name* binding to find out what users are allowed to do. SPKI authorization certificates, on the other hand, carry a direct *permission→public key* binding that tells an access controller what a given keyholder may do without relying on an intermediate naming step.

mission to perform a particular action or access a particular resource. To do this using a name-based mechanism, a computer must first authenticate a user to find out his name, then find out if the person with that name is allowed to do what the user is requesting. Therefore, the computer must traverse two edges of Ellison's triangle: it must consult both a *name→public key* binding such as an X.509 certificate, and a *permission→name* binding such as provided by an ACL entry or attribute certificate. SPKI authorization certificates can express a direct *permission→public key* binding.

2.3.4 SPKI delegation

SPKI allows any user, not just a dedicated CA or attribute authority (AA), to issue authorization certificates. In particular, a SPKI certificate might give a keyholder not just the permission to do something, but also the permission to *delegate* that permission, or a subset of it, to others. The motivation behind delegation is simply to allow people to use SPKI

certificates to express things they might want to do in the real world, e.g.:

- Suppose I have a SPKI certificate that grants me access to a particular directory on an FTP server, along with the ability to delegate that permission. If I wish to give a friend access to a certain subdirectory containing shared information, I can simply delegate to him by issuing a SPKI certificate that grants a subset of my own permission to him.

- Suppose I am going on vacation and want to access a number of resources (FTP directory, bank account, etc.) while I am away. However, I do not wish to transfer my master private key to the relatively insecure device—perhaps a PDA—that I plan to carry with me. Instead, I can create a new key pair and delegate all the permissions I will need while I am away to that new key pair. Like X.509 certificates, SPKI certificates have a validity interval, so I could cause this delegation to automatically expire at the end of my vacation.

2.3.5 The SPKI certificate format

All SPKI and SDSI objects are represented as Lisp-like *S-expressions*, as defined by Rivest [90]. S-expressions come in two basic varieties: *canonical* S-expressions, a compact byte-sequence representation used as input to hash and signature functions, and *advanced* S-expressions, used to format and print canonical S-expressions on screen or paper using only printable characters.[7] SPKI/SDSI libraries can typically process advanced S-expressions directly: SPKI/SDSI data can therefore be transferred in a standard format that is simultaneously email-safe and human-readable, an advantage which X.509's ASN.1 encoding does not share.

[7] A third variety, the *transport* S-expression, is simply a Base64-encoded canonical S-expression that can be transmitted, e.g., via email without becoming corrupted.

```
(public-key
 (rsa
  (e #010001#)
  (n
   |AOQjBj+wSG/BlAlir8Xuz62Hv3xAfAxJQeMl0kf93oWFzEcbK03h0kP3ueX4FaMMvsBYEqT
   uCK7h1CQHvuZrsRmjZmoP08zTOfrYYstU9wHW0QrPvTPrWlh52YXygS3NE8fHLOQkjwdCVf1
   CHubDxTnovrO7j7xBOsbeMgJArrvv|)))

(hash md5 |9WgBTLBGk6kIIvJVwZLbAg==|)
```

Figure 2.4: SPKI/SDSI refers to *principals* by directly specifying their public keys or hashes thereof. This figure shows an RSA public key in SPKI/SDSI's *S-expression* syntax (top), along with its hash (bottom).

In both SPKI and SDSI, a keyholder is referred to as a *principal*. A principal can be identified either by its full public key value or by the MD5 or SHA-1 hash thereof. Figure 2.4, for example, shows my public key value and its hash, both as advanced S-expressions. In an advanced S-expression, values surrounded by pound signs, such as `#010001#` in the figure, express a byte or sequence of bytes by its binary value. Values surrounded by vertical bars, such as the public key modulus value in the figure, express a sequence of bytes by its Base64-encoded value. All other S-expression atoms, such as `public-key` or `rsa`, are literal strings to be interpreted by a SPKI/SDSI library or its relying program.

Figure 2.5 shows a complete SPKI authorization certificate, along with its signature. A SPKI certificate contains the following fields:

- An issuer, identified by SPKI principal or SDSI name.
- A subject, identified by SPKI principal, SDSI name, or *threshold subject* (see Section 5.4.3).
- A *(propagate)* flag which, if present, allows the subject to delegate the granted permission to others.
- A *tag*, an S-expression that specifies the permission being granted (or perhaps an

```
(sequence
 (cert
  (issuer (hash md5 |BuFWyi13EpqzJtMff8DcsA==|))
  (subject (hash md5 |9WgBTLBGk6kIIvJVwZLbAg==|))
  (propagate)
  (tag (greenpass-pilot-auth))
  (valid (not-after "2004-07-02_17:43:06")))
 (signature
  (hash md5 |8i/bztbps+DU/bf2U4BtzA==|)
  (public-key
   (rsa
    (e #010001#)
    (n
     |ALgZbWph3tYzwZCjPFmIMNscdMWABEunlgIWJZHOxYAYGjlvFsbQ2Xa5Kht8aq515yENb
     UYeqMggJRmzhrAMS2iFpbPtfbPBgt53iIQ47NwuLUBNj1o7dYXE8xVLNPwMgmJnKdHvbS0
     aQnv09/mAfpkDA0UPt85iMGxQr9BrzlRN|)))
  (rsa-pkcs1-md5
   |cPW9HbmLD17skMI4c9UIuQbS2dxwZl6kwuCgGlRMCmIoFJWDgK5Na8bHNwYUx2C/AzIw0yl
   oL+TiBjTKAI3sgbUi/TNnuCZoJv2C+wVQ+IuwtbBTI0vQVexrVlkmnx2rzNT6vCHXFuZq4hd
   aA52X+4UqhFLXrJtbxl2+aefHjUg=|)))
```

Figure 2.5: A SPKI authorization certificate conveys the authorization given in its *(tag)* field, which must be interpreted on an application-specific basis, from its issuer to its subject. Notice that both the issuer and subject can be identified by hashes of their public key values. The *(propagate)* flag, if present, indicates that the subject is permitted to further *delegate* the stated permission to others. (Note that dates in the SPKI *(validity)* field should always be interpreted as UTC. Furthermore, string comparison functions will correctly compare SPKI dates according to less-than, equal-to, or greater-than relationships.)

attribute of the subject); its meaning depends on the relying party's interpretation.

- Validity information: an interval (as shown) or the URL of an *online test*, such as a CRL.

SPKI certificates may also contain a number of lesser-used optional fields, including a version number and a comment; the SPKI certificate structure document [31] gives full details.

2.3.6 SPKI certificate chains

When various SPKI principals delegate to one another in sequence, they produce a chain of authorization certificates, just as X.509 CAs can produce a chain of name certificates. A

party that relies on SPKI authorization certificates typically trusts one or more *sources-of-authority* (*SOAs*)—perhaps recognized by their public keys only, or hashes thereof, rather than by name—that define permissions for a particular resource. (An SOA is analogous to a trust anchor; I use the terms interchangeably.) The relying party must then find a valid certificate chain from one of its SOAs to the public key of any target whose access permissions it checks. As with X.509 certificate chains, the relying party could look somewhere for the certificates required to build a chain (the "pull" approach), or it could require clients to prove their permissions by presenting complete chains (the "push" approach).

In the SPKI theory document [32], Ellison et al. define the semantics of a SPKI certificate chain by providing an example of how to intersect two certificates in a chain to form a single certificate with the same meaning. (To verify a chain of more than two certificates, one would recursively intersect the certificates until only one remains.) Specifically, they define a 5-tuple representation for SPKI authorization certificates and define a method for performing *tuple reduction* to intersect two certificates.

At a high level, for two certificates to intersect,

- the issuer of the second certificate must match the subject of the first,
- the first certificate must contain a *(propagate)* flag,
- the validity intervals (and online validity tests) of the two certificates must intersect in the obvious manner, and
- the tags—i.e., permissions—must intersect.

At present, all SPKI certificates used by Greenpass contain the same tag, *(greenpass-pilot-auth)*, and two identical tags always intersect to form another copy of that same tag. The reader is directed to the SPKI theory document cited above for information on the semantics of more sophisticated SPKI tags.

2.4 The Greenpass RADIUS server

It is cumbersome to provide guest access using standard authentication and authorization mechanisms, such as EAP-TLS. SPKI provides an easy way to express the real-world relationships that give rise to guest-access scenarios in the first place. Asking guests to install custom wireless networking software that supports a SPKI-based handshake, however, seemed infeasible. Instead, Kim [55] designed a Greenpass RADIUS server that authenticates a supplicant using EAP-TLS, but consults a cache of SPKI certificates to find out if that supplicant is authorized to use the network.

2.4.1 SPKI-based access control: motivation

It is cumbersome to provide guest access using EAP-TLS with a standard RADIUS server for the following reasons:

- It is impossible to authenticate a guest who arrives from an arbitrary home organization. Either the RADIUS server must trust his home organization's CA, or an administrator must set up a new account for him, possibly by having him enroll with the local CA.

- Unless successful authentication implies authorization, an administrator must also set up an ACL entry or policy statement granting access to the guest by name.

In a university or large enterprise, a guest is typically invited to the local premises by a host—either an individual or a department—rather than by the organization as a whole. This host—not a busy network administrator—ultimately vouches for a guest's trustworthiness and grants him permission to access the network. Standard authentication and authorization mechanisms, however, force the host to take her guest to a busy network administrator, who will add both authentication and authorization materials for the guest as

described in the preceding paragraph. Upon closer examination, requiring the administrator to perform either of these actions seems nonsensical:

- The name given to the guest has no meaning to the administrator: it serves only as an identity by which to authorize the guest.
- The administrator asserts that the guest should be granted access by, e.g., adding an ACL entry, but this assertion is not his own: it merely reflects the host's assertion.[8]

SPKI authorization answers these problems directly and eliminates the need for an administrator to act as middleman:

- The guest does not need a name, just a public/private key pair.
- The host can vouch for her guest by delegating to him.

By delegating to her guest's public key, the host tells the access-control mechanism all it needs to know: "the entity who owns this key should be granted access because I said it's okay." (The host's existing certificate chain adds, "and here's why you should listen to me.")

2.4.2 "Pure" SPKI-based access control

Figure 2.6 illustrates purely SPKI-based access control to a wireless network. In this scenario, an access point is configured with a single public key as its source-of-authority. That SOA can then issue one or more authorization certificates to subordinate delegators who propagate network access to those who need it. When a guest or other end-user associates

[8] In a highly sensitive environment where guest access has, nevertheless, been deemed necessary, it might make sense for a security officer, or even a psychologist, to vouch for a guest after performing a background check or interview.

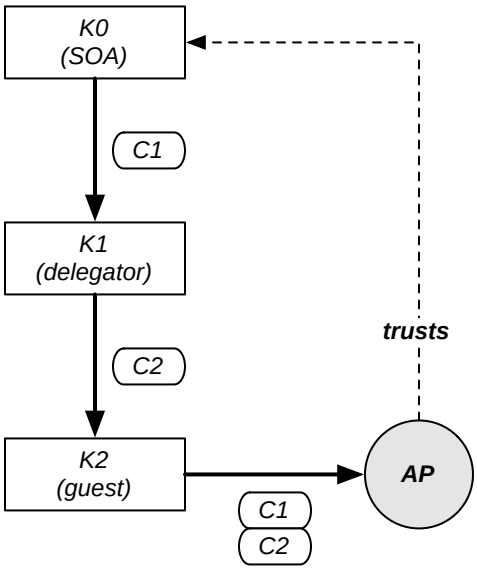

Figure 2.6: Purely SPKI-based access control to a Wi-Fi network. In this example, an access point is configured with a single public key, *K0*, as its source-of-authority (SOA). That SOA issues a SPKI authorization certificate *C1* to a subordinate delegator, identified by her public key *K1*. She in turn issues a certificate *C2* to a guest who holds key *K2*. To obtain network access, the guest presents his entire certificate chain—*C1* and *C2*—to the access point, then proves that he owns the public key that is the final subject of that chain.

with an access point, the AP demands proof of authorization: the user responds by first presenting a chain of certificates originating from the AP's SOA, then proving ownership of the public key that is the final subject of that chain.[9]

This approach does not suit our needs because it would require guests to install custom Wi-Fi drivers to support our custom SPKI-based handshake. Additionally, it would require modifying the access point itself, or perhaps a RADIUS server behind it, to accept the new handshake.

[9]The AP described operates nearly identically to the compact firewall device that Ellison et al. use as a motivating example in the SPKI theory document [32].

2.4.3 SPKI-based authorization with EAP-TLS authentication

The EAP-TLS handshake, as it exists, only allows a supplicant to present X.509 authentication certificates, not SPKI certificates or attribute certificates; as depicted in Figure 2.7, such a handshake is only useful for local users. The Greenpass solution, depicted in Figure 2.8, is to use a hybrid SPKI- and X.509-based access-control mechanism that requires a two-step Wi-Fi login process for guests:

1. the guest's Web browser presents a SPKI certificate chain via HTTP to an *authorization cache* (described in Chapter 3), then

2. authenticates to the RADIUS server itself using an unmodified EAP-TLS handshake.

Kim's modifications [55] to the open-source FreeRADIUS server [37] add an extra step to its EAP-TLS authentication process: after receiving the supplicant's X.509 certificate and obtaining proof that the supplicant owns the public key therein, it consults the authorization cache to see whether the owner of that public key has presented a valid SPKI certificate chain.

Notice that, in this final scenario, the Greenpass RADIUS server "authenticates" a supplicant even if it does not trust the issuer of his X.509 certificate. This works because when a supplicant presents his X.509 certificate, he essentially claims *two* identities: his public key and his name. The EAP-TLS handshake requires the supplicant to prove ownership of the public key he has just claimed: the RADIUS server, therefore, can authenticate him by one identity (his public key) regardless of whether it is able to verify his second identity (his name). This half-authentication is entirely sufficient for our purposes, because SPKI binds permissions directly to public keys.

We have not yet addressed the problem of roaming clients; see Section 5.3 for further discussion.

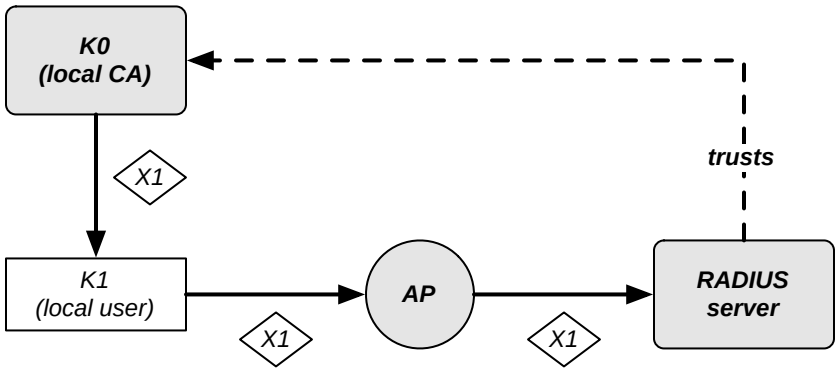

Figure 2.7: This illustration shows the standard X.509 PKI model used for EAP-TLS access control to a Wi-Fi network. Here, the RADIUS server trusts a local CA that certifies all local users. The CA certifies some user by issuing an X.509 certificate *X1*. That user then authenticates using a standard EAP-TLS handshake via which he presents his local X.509 certificate and proves knowledge of the corresponding private key. The RADIUS server makes further authorization decisions, if necessary, based on an internal ACL. Greenpass uses this model for local users.

2.4.4 VLAN switching

Technically, our Wi-Fi authentication system never fully denies access to any user. Instead, we provide two *VLANs* (*virtual local-area networks*): one that provides access only to a Web server hosting Chapter 3's client tools, and one that provides unrestricted access to the local Dartmouth network (as well as its Internet gateway). Our access point is configured with two SSIDs: a broadcast one that does not require authentication but gives access only to our restricted VLAN, and a "secret" SSID that requires authentication/authorization and provides access to the unrestricted VLAN. Without this provision, guests would be caught in a chicken-and-egg problem: they would need credentials in order to access the network, but would only be able to obtain those credentials over the same network.

In theory, all users should be able to connect to a single SSID: the RADIUS protocol includes extensions that can tell an AP to which VLAN a supplicant should obtain access. Kim [55], however, found that the FreeRADIUS server he modified cannot use these ex-

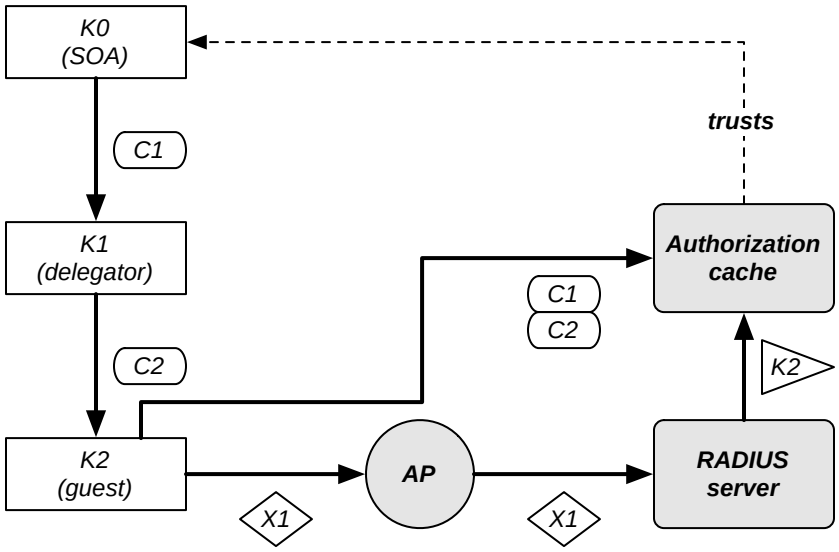

Figure 2.8: This illustration shows the hybrid SPKI- and X.509-based PKI model that we use for access control to a Wi-Fi network in Greenpass. SPKI delegation occurs as in Figure 2.6 above, but the guest must perform a two-step login process: he first presents a SPKI certificate chain via HTTP to an *authorization cache*, then presents his public key—encapsulated in an X.509 certificate—to a RADIUS server via an unmodified EAP-TLS handshake. The RADIUS server checks that he knows the corresponding private key via EAP-TLS, then grants him access if the authorization cache contains a valid certificate chain with his public key as its subject.

tensions in the manner we would prefer without major changes to its structure. A future version of Greenpass could offer more than two VLANs that provide varying access levels to different classes of users, as discussed in Section 5.4.1.

A final detail about our restricted VLAN is worth mentioning. It includes a DNS server that redirects unauthorized users' HTTP requests to the Web server hosting our client tools. That server displays instructions and allows guests to obtain authorization credentials as discussed in Chapter 3. Again, see Kim [55] for further details on our provisions for DNS redirection.

2.4.5 Final clarifications

A few final clarifications are needed for the reader to fully understand the Greenpass RADIUS server and its relation to the client tools described in Chapter 3.

First, local users do not need SPKI certificate chains to obtain access. The Greenpass RADIUS server grants access to any supplicant that the local CA has certified, as illustrated by Figure 2.7. It consults the Greenpass authorization cache only when it encounters a supplicant who was certified by an unknown CA. (In a more elaborate configuration, the local CA might add extensions to its X.509 certificates to indicate which of its subjects should be granted Wi-Fi access. Or, the RADIUS server could be configured to trust no CA and require SPKI credentials from all users who want unrestricted access.)

Second, however, local users *do* require SPKI certificate chains in order to act as delegators. A future implementation could allow mixed certificate chains in which, e.g., the authorization cache would accept X.509 certificates containing a certain "delegator" extension as ancestors to a SPKI certificate. For now, however, a guest must present a full SPKI certificate chain that originates from the authorization cache's SOA, propagates through zero or more delegators, and ends with the guest's own public key.

Third, the delegator/guest scenario is just one scenario Greenpass can handle. Non-local users can easily act as delegators if their own SPKI certificates contain the *(propagate)* flag. Such a use might prove useful if, say, a large conference brought a number of guests to a university campus: the local conference organizers could designate certain non-local conference attendees as trusted delegators, who could then help expedite the process of providing Wi-Fi access to all other attendees. (On the other hand, because SPKI provides only a boolean *(propagate)* flag rather than an integer "delegation depth limit," our current approach cannot *prevent* delegators from giving their guests the power to delegate. This is a limitation inherent in SPKI.)

2.5 Summary

Greenpass uses a hybrid SPKI- and X.509-based access-control mechanism to allow decentralized, delegated authorization of Wi-Fi network users.

IEEE 802.1x forms the basis of our approach by enabling Wi-Fi users to authenticate via *EAP-TLS*, an X.509-based handshake derived from the TLS (formerly SSL) client authentication method supported by most modern Web browsers. While X.509 enables an organization to decentralize its authentication information, standard EAP-TLS still requires a RADIUS server to consult an ACL or other centralized authorization mechanism to find out whether an authenticated user should be granted network access. X.509's particular PKI structure poses two major obstacles to easy guest authorization:

- it is impossible to authenticate a guest who arrives from a home organization whose CA the local RADIUS server does not trust, and
- even if the guest could authenticate, a central administrator would need to add an ACL entry or policy statement to the RADIUS server in order to grant him access.

SPKI authorization answers these problems directly. The person who invited the guest can *delegate* Wi-Fi access privileges directly to him—without the intervention of a network administrator—by issuing him a SPKI certificate. Furthermore, the guest's host can issue this certificate directly to the guest's public key value rather than relying on a CA to provide a name for him. Unfortunately, a purely SPKI-based access-control mechanism would require a custom handshake and custom Wi-Fi drivers to support it, greatly complicating guest access.

Kim's Greenpass RADIUS server [55] allows a hybrid approach:

- local users connect using a standard EAP-TLS handshake with X.509 certificates only, but certain local users might also obtain SPKI certificate chains that designate them as *delegators*;
- a guest can ask a delegator to authorize him directly via a new SPKI certificate;
- the guest presents this SPKI authorization credential to an *authorization cache* by sending it in an HTTP cookie; then
- the guest authenticates to the RADIUS server itself using an unmodified EAP-TLS handshake.

This approach provides the benefits of SPKI delegation without requiring a custom client handshake. Unauthorized users are not rejected entirely, but are put on a restricted VLAN that provides access only to the Greenpass Web application server described in Chapter 3.

Chapter 3

Greenpass client tools

At the end of Chapter 2, we saw how the Greenpass RADIUS server bases its access-control decisions on SPKI/SDSI certificates. This chapter describes a set of Web-based client tools that allow local users to create these certificates by delegating to guests. This chapter is organized as follows: Section 3.1 outlines our requirements for the client tools, Section 3.2 describes the trade-offs we considered and the design decisions we made in order to meet these requirements, Section 3.3 describes the final set of components I created and how information flows between them, and Section 3.4 describes my implementation of these components in greater detail.

3.1 Requirements

At the highest level, the Greenpass delegation process takes place as follows:

1. a guest transmits his public key value (or a hash thereof) to a delegator, and

2. the delegator issues a new SPKI certificate to the guest—referring directly to the guest's public key as the certificate's subject—that grants him Wi-Fi access privi-

leges.

This section analyzes these requirements in greater detail.

3.1.1 Guest requirements

Before the Greenpass RADIUS server will grant a guest access to our unrestricted VLAN, the guest must possess the following:

- *A key pair*: specifically, a key pair associated with an X.509 name certificate. (He needs an X.509 certificate, rather than just a raw key pair, so his WPA client software can authenticate to the Greenpass RADIUS server using EAP-TLS.)

- *A chain of SPKI/SDSI certificates* that proves he is authorized to access the local Wi-Fi network.

Our client tools must enable a guest to obtain these materials. As PKI grows in popularity among various organizations, guests will increasingly have key pairs and X.509 certificates already; we will, however, need to generate them for those who don't. To obtain a SPKI/SDSI certificate via delegation, a guest—we will call him Bob—must interact with a local user, Alice, who is permitted to delegate Wi-Fi access to others. Our client tools, therefore, must allow Bob to complete the following steps:

1. obtain a key pair and associated X.509 certificate, if necessary;

2. transmit the public key value from his X.509 certificate to Alice; and

3. receive a SPKI/SDSI certificate chain granting him Wi-Fi access after Alice has created it, and store it in such a way that his device can present it upon demand.

In addition, these steps should rely only on software that Bob already has installed.

3.1.2 Delegator requirements

In order for Alice to delegate to Bob, our client tools must enable her to do the following:

1. Receive Bob's public key value.

2. Verify that the public key value she has received really belongs to Bob. Alice presumably knows Bob and can identify him by sight or voice, but we cannot assume that she already knows his public key or that she treats Bob's home CA as a trust anchor. Therefore, our client tools must provide a way for Alice to bind her mental "authentication" of Bob to a digital public key value.

3. Construct a SPKI certificate that authorizes Bob to use the Wi-Fi network.

4. Sign the resulting SPKI certificate using her own private key.

5. Append the new SPKI certificate to her own SPKI certificate chain (i.e., the one that gives her authorization to delegate in the first place).

6. Transmit the new certificate chain back to Bob.

Ideally, delegators, like guests, should not need to install custom software. In a straightforward scenario where delegators will invariably be local users at some particular organization, custom software might be acceptable—it could be pre-installed on all machines provided or sold through that organization. Even with this constraint, however, delegators who rarely use their privilege may not wish to have this extra software or deal with the hassle of upgrading it on each of those few occasions they need it. Additionally, avoiding custom software makes our solution more flexible, because it can support non-local delegators such as the trusted conference attendees used as an example in Section 2.4.5.

3.2 Design choices

The most challenging problems posed by the previous section's requirements are as follows:

- The guest needs to transmit his public key value to the delegator, and the delegator needs to transmit a SPKI certificate chain back to him. We would like to avoid custom software (or hardware) on either side.
- We need access to the delegator's private key, again, preferably without requiring her to install custom software.
- The delegator needs to make sure the public key she delegates to is the public key of her intended guest.
- The guest's device needs to store his new SPKI certificate chain and present it upon demand, but we do not wish to modify the guest's Wi-Fi software to support a custom handshake.

This section describes how we solved these problems. Figure 3.1 illustrates the flow of information in our final design, which Section 3.3 describes in full detail. Figure 2.8 illustrates how our client tools' design fits together with a Wi-Fi access point and the Greenpass RADIUS server.

3.2.1 Key and certificate transmission

As we saw in the previous section, the Greenpass client tools must provide a two-way communication path between guest and delegator: a guest needs to transmit his public key value to a delegator, and the delegator needs to transmit a new SPKI certificate back to the guest after signing it.

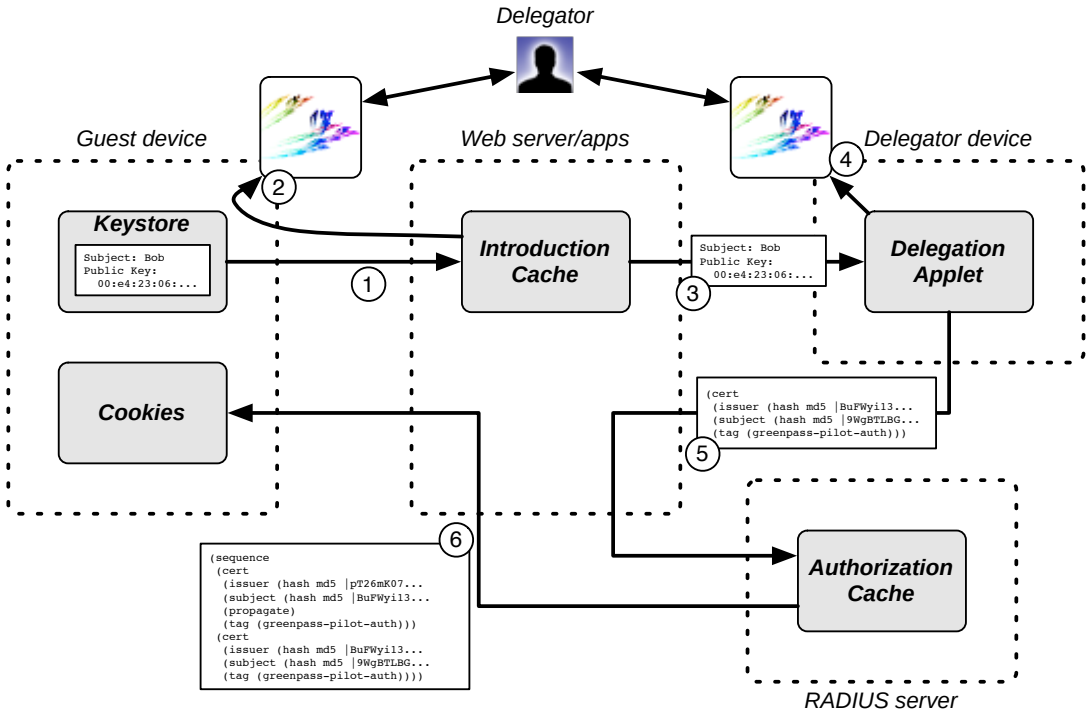

Figure 3.1: The Greenpass delegation process takes place according to the sequence shown in this diagram. Dotted lines in the figure represent the boundaries of different machines or "domains." A line that passes through the "Web server/apps" domain indicates that that information passes through the Web apps en route to its final destination. Referring to the numbers shown above, the delegation process is as follows. (1) A guest presents his public key value—wrapped in an X.509 certificate—to the guest introduction Web app, which then stores it in the introduction cache. (2) The guest Web app generates a visual fingerprint of the guest's public key and sends it back to the guest's Web browser, which displays it on his screen. (3) A delegator connects to the delegation Web app and retrieves the guest's public key value from the introduction cache. The Web app launches the delegation applet in the delegator's Web browser and sends the guest's public key to it. (4) The delegation applet displays the guest's visual fingerprint; the delegator must validate it by comparing it to the one displayed on the guest's screen. (5) If the delegator successfully verifies the guest's identity, the delegation applet will construct and sign a SPKI certificate that propagates Wi-Fi network access privileges from the delegator to the guest. It then submits this certificate (along with the delegator's existing certificate chain) back to the delegation Web app, which forwards it to the authorization cache. (6) When the guest refreshes his Web browser or revisits the Greenpass front page, the Web app recognizes him as someone waiting for authorization credentials, retrieves his fresh certificate chain from the authorization cache, and sends it to his Web browser's cookie store.

We initially considered giving delegators a physical device, such as a USB thumb drive, on which to pass information back and forth. This approach has a number of drawbacks: it does not provide an automated way for guests without key pairs to generate them (unless the device includes custom software that does so), it requires guests to export their certificates to a file on the device, it requires the purchase of a significant number of such devices, and it can be a hassle to work around driver problems and virus scans when using such devices on guest computers.

Direct transmission of keys and certificates over the network or a private channel would have significant advantages. Delegators could, for example, run delegation services from their own machines that guests would discover using, e.g., Apple Computer's Rendezvous protocol [4]. Alternatively, the guest and delegator devices might exchange information directly using an infrared channel. These solutions, unfortunately, would require custom software or hardware on the guest's machine.

In the end, we chose to provide key and certificate transmission using a Web server that provides a rendezvous point between guest and delegator. This approach has the following advantages:

- The Web server can learn the guest's public key value via SSL client authentication. As we observed in Section 2.4.3, an SSL or TLS handshake enables a server to learn a client's public key value and check his knowledge of the corresponding private key, even without trusting the issuer of the client's X.509 certificate.

- In most cases, the guest will not need to export his EAP-TLS certificate to a file. The major client operating systems' (Windows, Mac OS X) default Web browsers (Internet Explorer, Safari) use OS-wide keystores, which they share with EAP-TLS and other clients, for SSL client authentication.

- Web browsers that support SSL client authentication typically support key generation

and certificate enrollment as well, which will provide a convenient way to get these materials to guests who do not already have them.

- Web browsers are ubiquitous on general-purpose networked computers.[1]

In our current design, as shown in Figure 3.1, the guest presents his public key (encapsulated in an X.509 certificate) to a Web application that we provide. This Web application stores his certificate in an *introduction cache* from which the delegator can later retrieve it using a complementary Web application. The delegator, in turn, sends the new SPKI certificate back to her Web application, which stores it in an *authorization cache* from which the guest can retrieve it using his Web application.

3.2.2 Delegation

In order to assist a delegator in signing a SPKI certificate for her guest, our delegation solution needs access to the delegator's private key. Merely constructing the certificate is not a problem: our Web server could build it on the delegator's behalf, then send it to the delegator only for the final signing operation. Accessing the delegator's private key, however, poses a problem. Web browsers do not allow Web servers to access a user's private key, for obvious reasons; this consideration rules out a purely HTML-based approach. We considered a number of possibilities for our delegator tool.

I initially considered using CAPICOM [61], an optional ActiveX control that Microsoft provides for use by Windows developers. CAPICOM allows JavaScript, VBScript, etc. to access a subset of the Windows CryptoAPI. In particular, it allows JavaScript embedded in a Web page to sign text using the browser user's private key. Signing a SPKI certificate using CAPICOM provided difficult, but not impossible: the text to be signed must be passed

[1] Section 5.7 discusses how we might accommodate VoIP phones and other limited networked devices that lack Web browsers and/or 802.1x authentication.

to CAPICOM's *Sign()* function as a Unicode string, but one can provide an arbitrary byte string as an argument by including escape characters in the text (e.g., `"\u35c9"`), so long as the intended byte string has even length. The length of a SPKI certificate in canonical form can easily be adjusted by adding a SPKI *(comment)* field. Still, this approach has numerous drawbacks. First, installing the CAPICOM library on all an organization's machines would create an enormous security risk: a Web page can sign data using a user's private key without the user seeing the data to be signed! Second, the resulting signature is created in PKCS#7 [94] format, rather than the PKCS#1 format normally used by SPKI/SDSI. PKCS#7 signatures are larger, and we would have needed to modify existing SPKI/SDSI libraries to support them. Finally, CAPICOM-based signing would only be available on Microsoft Windows platforms using Internet Explorer, and we wished to keep our tools compatible with multiple platforms.

I also considered writing a standalone application to support delegation. This approach, too, has its problems: it requires rewriting the tool for each platform that delegators might use, and it does not integrate particularly well with the Web-based portion of the client tools. A slight variation on this approach would implement the application as an XPCOM [64] component that runs inside Mozilla (or recent versions of Netscape); only a recompile, rather than a rewrite, would be needed to port such a component to new platforms. I dismissed this idea also: it provides good integration with the Web-based interface we have already chosen for guests, but forces all delegators to install Mozilla. We did not wish to force users to install a particular browser just to delegate.

In the end, I implemented the delegator tool as a signed Java applet. A Web browser will run a signed applet only if its user chooses to mark the signer's X.509 certificate as trusted. Signed applets have greater access privileges than untrusted applets: in particular, they can access the user's local filesystem. The current delegation applet uses this privi-

lege to load the delegator's private key from a password-protected PKCS#12 keystore file. Requiring delegators to pre-install the applet signer's certificate is not unreasonable: an organization could distribute a custom trust anchor certificate for signed Java code more easily than it could distribute custom software.[2] Java also provides highly mature technology for developing full-fledged cross-platform applications, so the delegation applet could easily be transformed into a standalone or Java Web Start [99] application if that option appears more appropriate in the future.

3.2.3 Guest introduction and authentication

Alice receives Bob's public key value via an insecure channel (the Web), and therefore must validate it before using it to identify Bob in a SPKI certificate. The Web server that learns Bob's public key via SSL and transmits it to Alice cannot truly authenticate him, because it does not recognize his home CA. Neither does any software on Alice's computer recognize Bob's home CA. The burden therefore rests on Alice to "authenticate" Bob by some other means. Ellison and Dohrmann [28] call this process *introduction*, which, in their words, consists of "establishing that the key belongs to the person you think it does." (Section 4.4 discusses Ellison and Dohrmann's particular introduction problem in more detail.)

A similar situation might arise if Alice and Bob were users of Phil Zimmermann's PGP secure email program. To encrypt mail for Bob's eyes only, Alice would first need to know his public key value. Although PGP supports the use of CAs, PGP users often exchange public keys values by transmitting them over an insecure channel, then comparing their *public key fingerprints* to ensure that the values were not modified—by accident or ill

[2] Not every Java runtime environment uses the OS-wide keystore to hold trust information regarding code signers. Future work might include discovering all the quirks of various keystores and Java environments, and identifying how best to distribute initial trust anchor certificates for the smoothest Greenpass experience. See Section 5.2 for further discussion.

intent—during transmission. A fingerprint is simply the hexadecimal hash value (computed using, e.g., MD5 or SHA-1) of a user's public key: e.g., *29 6F 4B E2 56 FF 36 2F AB 49 DF DF B9 4C BE E1*. To follow standard PGP culture, Bob would either read Alice his fingerprint (in person or over the phone) or print it on his business card, which he would then hand out to Alice and others.

Average users are unlikely to use this process correctly, however: Ellison and Dohrmann point out that "key verification (e.g., the comparison of hex key fingerprints) is the geekiest, slowest, most painful and most cumbersome part of the introduction process." To speed up the process, they suggest the use of a *visual hash*, also called a *visual fingerprint*. To create a visual hash, a computer simply transforms a hexadecimal fingerprint value into a unique image. A human can compare two visual hashes more much more quickly than two hexadecimal hashes.

We chose to use visual hashing for guest introduction in Greenpass. The client tools generate visual fingerprints using an adaptation [52] of Ian Goldberg's *Visprint* program [43]. Visprint transforms a 128-bit MD5 hash into a colorful fractal image, as shown in Figure 3.2. (Incidentally, Goldberg originally designed Visprint to aid in the exchange of PGP keys.) Our solution is not dependent on visual hashing: any easy method of hash comparison—visual, aural, or otherwise—would suffice.

3.2.4 SPKI certificate storage

After Alice delegates to Bob, Bob's device needs to store his new certificate chain—which consists of one new certificate appended to Alice's existing chain—where it can automatically present it upon demand. Unfortunately, the EAP-TLS handshake only allows a supplicant to present authentication (X.509) certificates, not SPKI certificates (or X.509 attribute certificates, which could serve the same purpose). Bob must, therefore, transmit his autho-

Figure 3.2: A visual fingerprint of the public key shown in Figure 2.4, generated by Johnston's newer adaptation [52] of Ian Goldberg's *Visprint* program [43]. Visprint transforms a 128-bit MD5 hash into a unique image such as the one shown here.

rization certificates via an outside channel. We chose to send Bob's Web browser an HTTP cookie containing his certificate chain. Bob's Web browser will store it without Bob's intervention and present it back to our Web server each time Bob connects to it. We also add Bob's certificate chain to an *authorization cache* (discussed below) of valid certificates. If Bob's credentials expire from this cache, he can revisit our Web server and instantly become re-authorized when his browser presents his certificate chain, instead of having to go through the introduction and delegation process again. Section 3.4 describes our use of HTTP cookies in greater detail.

3.3 Design

I implemented the Greenpass client tools by building five components, used in the pilot described by Powell [87]:

1. A set of three ***Greenpass Web applications***:

   - a *guest introduction Web app*,
   - a *delegation Web app* that wraps and launches the delegation applet described below, and
   - a *front page* that serves as a rendezvous point for both guests and delegators.

   These applications allow users to interact with the remaining four components via a Web browser. I implemented all three Web apps in Python using the Albatross [2] Web application framework, and they are hosted on an Apache Web server.

2. A ***dummy CA*** that the Web applications use to obtain temporary X.509 certificates for guests who need them. The dummy CA is simply an XML-RPC [112] daemon, written in Python, that wraps OpenSSL's basic CA functionality.

3. An ***introduction cache***, another Python-based XML-RPC daemon, that holds guests' X.509 certificates until delegators retrieve them and extract the public keys from them.

4. A ***delegation applet*** that enables delegators to verify their guests' identities and construct and sign SPKI certificates.

5. An ***authorization cache*** that serves two related purposes:

   - it holds freshly-issued SPKI certificates in a place where guests can pick them up, and
   - it caches SPKI certificates that guests have recently presented to our Web server so the Greenpass RADIUS server knows to which guests to grant access.

   The authorization cache is also an XML-RPC daemon, but is written in Java to take advantage of the JSDSI SPKI/SDSI library [53]. At present, the authorization cache also provides validation logic for SPKI certificate chains.[3]

Figure 3.1 illustrates the Greenpass delegation process as a flow of information among the guest's Web browser, the introduction and authorization caches, and the delegation applet. The dotted lines in the figure represent the boundaries of different machines or "domains".[4] A line that passes through the "Web server/apps" domain indicates that that information passes through the Web apps en route to its final destination.

[3] We chose this approach because the JSDSI library proved easier to use than an analogous C library [91] that we originally considered linking to our modified FreeRADIUS server. Section 5.1 discusses the security issues with this approach.

[4] As drawn in Figure 3.1, the domains reflect the setup we used for the Greenpass pilot: i.e., the introduction cache coexists on the same machine with our Web server and its hosted applications, and the authorization cache coexists on the same machine with our Greenpass RADIUS server. The Web apps can be reconfigured to communicate with introduction and authorization caches anywhere.

Referring to the numbers shown in Figure 3.1, the Greenpass delegation process takes place as follows:

1. A guest presents his public key value—wrapped in an X.509 certificate—to the guest introduction Web app, which then stores it in the introduction cache.

2. The guest Web app generates a visual fingerprint of the guest's public key and sends it back to the guest's Web browser, which displays it on his screen.

3. A delegator connects to the delegation Web app and retrieves the guest's public key value from the introduction cache. The Web app launches the delegation applet in the delegator's Web browser and sends the guest's public key to it.

4. The delegation applet displays the guest's visual fingerprint; the delegator must validate it by comparing it to the one displayed on the guest's screen. (Aural hash comparison methods could be added to enable secure guest introduction via telephone.)

5. If the delegator successfully verifies the guest's identity, the delegation applet will construct and sign a SPKI certificate that propagates Wi-Fi network access privileges from the delegator to the guest. It then submits this certificate (along with the delegator's existing certificate chain) back to the delegation Web app, which forwards it to the authorization cache.

6. When the guest refreshes his Web browser or revisits the Greenpass front page, the Web app recognizes him as someone waiting for authorization credentials, retrieves his fresh certificate chain from the authorization cache, and sends it to his Web browser's cookie store.

Figure 2.8 illustrates how our client tools' design fits together with a Wi-Fi access point and the Greenpass RADIUS server.

3.4 Implementation

This section describes various implementation details of the Greenpass client tools. It is organized around the functions of the three Web applications: guest introduction, delegation, and front page/rendezvous point. Subsection 3.4.1 describes the common functionality that supports all three Web applications (and, to some extent, the Python-based daemons), while each of the remaining three sections describes a Web app along with the lower-level components that it uses.

3.4.1 Web applications: common functionality

The Albatross framework [2] provides a lightweight set of classes and utilities which allow developers to write Python-based CGI scripts that are logically organized as Web applications. I originally wrote the Greenpass Web interface using standalone CGI scripts, but decided to re-implement it using Albatross because its templating engine allows a developer to separate logic (Python code) and presentation (HTML code) in a natural manner. It is also simple enough, and its implementation readable enough, that I was able to augment it with extra functionality by extending the provided Albatross classes rather than patching them. This extra functionality includes custom use of cookies, HTTPS support with client authentication, and custom methods for accessing the various Greenpass daemon components.

All three Greenpass Web applications cooperate via cookies placed in their users' Web browsers. One cookie, *greenpass_hash*, holds the MD5 hash of a user's public key as a hexadecimal string: e.g., on my machine this cookie contains the value `f568014cb04693a90822f255c192db02`.[5] A second cookie, *greenpass_waiting*, is only present between the

[5]The Web apps use this cookie for identification only, not as a form of authentication. See, however, Section 5.1 for a discussion of rough edges involving the use of this cookie.

time a guest introduces his public key and the time he receives a certificate chain. Finally, *greenpass_chain* contains a user's SPKI certificate chain itself as a Base64-encoded canonical S-expression.

We wish to use a guest's existing X.509 certificate and key pair, if they exist, for Greenpass identification; I therefore had to configure the Apache server that hosts the Web apps to obtain a client's certificate, if it exists, and pass it on to the CGI script that wraps the guest Web app. Apache's HTTPS module, *mod_ssl*, recognizes a configuration directive, *SSLVerifyClient*, that controls whether Apache should demand client authentication and sets the verification level for client certificate chains. For the Greenpass Web apps, I set the level to *optional_no_ca*, which, fortuitously, requests a client certificate and accepts it even if Apache does not recognize its issuer. (If Apache demanded, rather than requested, client authentication, then guests without certificates could not connect to it; if it accepted only those certificates issued by known CAs, then only guests from organizations with previously-trusted CAs could connect.) I also configured Apache to export various HTTPS session information, including the client's PEM-encoded certificate, to environment variables when running CGI scripts, allowing the Web applications themselves to read and process this information.

The Web applications also share Python modules that allow them to process certificates. One module uses M2Crypto [59], a set of Python wrappers around OpenSSL, to process guests' X.509 certificates—most importantly, to extract public key values from them.[6] The other module is a lightweight canonical S-expression generator that translates public keys into their SPKI/SDSI representation and hashes them. These raw hash values are then converted to hexadecimal strings and used to identify Greenpass users, as discussed earlier.

[6] I had to modify M2Crypto to add support for an OpenSSL function that extracts an RSA public key structure and makes its exponent and modulus available as arbitrary-precision integers; this patch required only three lines of C code, however.

3.4.2 Guest introduction

The guest introduction Web app obtains a guest's X.509 certificate by reading it from an environment variable set by Apache, as described above, then extracts his public key value and generates the visual fingerprint thereof. A guest can also introduce his X.509 certificate by uploading it from a PEM-formatted file on his local hard disk (in case his browser does not support SSL client authentication), or can generate a new X.509 certificate using our dummy CA (described below). To make the guest's Web browser display his fingerprint, our Web app sends the browser a page including an HTML `<IMG>` tag that points to a short, auxiliary CGI script that wraps the command-line *visprint* program; this script generates the actual fingerprint image and sends it to the guest's browser as JPEG data. Since generating a visual fingerprint is fairly expensive, the script in question caches fingerprints as it generates them so it can redisplay them quickly.

In order to help a delegator look up a particular guest, the guest Web app also calculates a four-digit *subject ID* and displays this to the guest. It currently derives this number from the guest's public key fingerprint, modulo 10^4. This "mini-fingerprint" is clearly not large enough to serve any security purpose; it merely provides a convenient way for a guest to communicate his public key identity to a delegator, so the delegator can locate it among many other keys that may have been introduced recently.[7]

After the guest introduces his certificate, he can simply wait at the page that shows his visual fingerprint and refresh his browser after someone has delegated to him. The Web app will then retrieve his SPKI certificate chain from the authorization cache and send it to his browser as a cookie. I fully describe the use of cookies for authorization in Section 3.4.4, below.

Figure 3.3 shows a screenshot of the guest introduction Web app's final page.

[7] Section 5.2 discusses ways to streamline this approach.

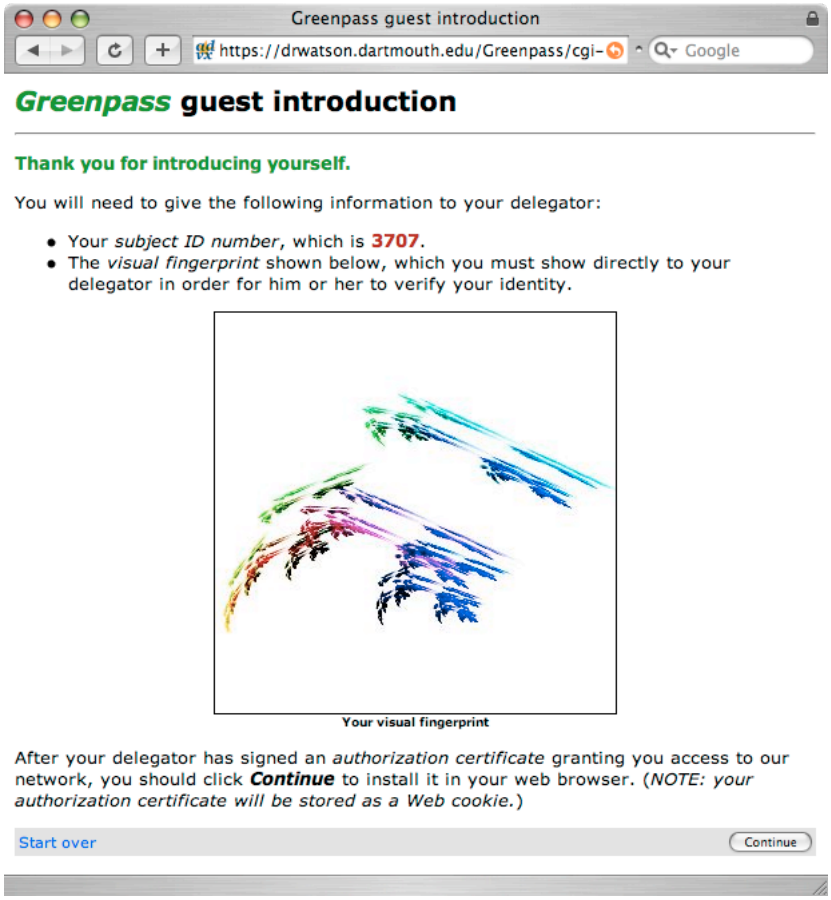

Figure 3.3: A screenshot of the Guest introduction Web app. The guest waits for a delegator after reaching the page shown, which includes a four-digit *subject ID* that helps the delegator find the guest's record in the introduction cache as well as a *visual fingerprint* of the guest's public key.

Dummy CA enrollment

The guest Web app includes an interface to the dummy CA daemon that lets guests obtain a key pair and associated X.509 certificate. The dummy CA does not require a high-assurance client verification process, because no party relies on it as a naming authority and the certificates it issues carry no implicit authorization. Enrollment with the dummy CA is a simple two-step process:

1. The guest visits the enrollment page and enters some basic identifying information in an HTML form, consisting of only three distinguished name (DN) fields: common name (required), organization (optional), and email address (optional). (Nothing prevents the guest from using a nickname as his common name.) When the guest submits the form, his Web browser generates a new key pair, stores the private key locally, and submits the public key to the Web server as a *certificate request*.

2. After the guest submits the enrollment form, the Web app forwards his certificate request to the dummy CA daemon, which responds with a new, signed X.509 certificate. The Web app sends the guest's browser a confirmation page that includes code to install the new certificate in the user's keystore automatically.[8]

The details of certificate enrollment differ between the Netscape/Mozilla family of Web browsers and Internet Explorer; readers interested in the details should refer to the source code of either the Greenpass guest Web app and dummy CA or to the open-source PyCA [88] and OpenCA [80] projects.

[8] Netscape, Mozilla, and Internet Explorer provide for automatic installation of client certificates; Apple's Safari browser requires its user to download the certificate, then double-click on the resulting file. The guest introduction Web app points Safari to an auxiliary CGI script to cause it to download the new certificate.

The introduction cache

The introduction cache is extremely simple. It stores X.509 certificates, indexing them by 4-digit subject ID (see above) for later retrieval by the delegation Web app.

3.4.3 Delegation

The delegation Web app allows a delegator to search for a guest's X.509 certificate by 4-digit subject ID. It then displays a list of matches,[9] letting the delegator choose between them by common name, organization, and email address information (like the subject ID, this information serves no security purpose, but facilitates preliminary identification). After the delegator chooses her desired guest, this Web app launches the delegation applet and sends it the MD5 hash of the guest's public key. The delegation applet, discussed in detail below, takes care of guest fingerprint verification and constructs and signs a SPKI authorization certificate for the guest.

The Web app communicates with the delegation applet using LiveConnect [68]. LiveConnect, developed by Netscape but now supported in Mozilla, Internet Explorer, Safari, and other browsers, allows JavaScript embedded in an HTML page to call public methods of a Java applet included in the same page and, conversely, allows an applet to manipulate components of its HTML page in the same way a JavaScript routine might. When the delegator selects her intended guests from the list of matches and clicks the "Delegate!" button, JavaScript code embedded in the page sends the delegation applet the hash of that guest's public key; it is at this point that the delegation applet displays its main window. After the delegation applet runs and creates a signed SPKI certificate, it in turn uses LiveConnect

[9] Normally, it will only display one match; it displays a list of matches if it finds more than one guest with the same subject ID number. Due to the birthday paradox, we can expect collisions to happen after around 100 guests have introduced themselves.

to place the fresh certificate in a hidden HTML form field and automatically submit the form back to the delegation Web app. In this way, the Web app receives the new certificate server-side and sends it to the authorization cache.

The delegation applet

Figure 3.4 shows a screenshot of the Greenpass delegation applet. This Java applet itself allows a delegator to verify her guest's visual fingerprint and specify a validity interval for the guest's new certificate, as well as specify whether it should contain the SPKI *(propagate)* flag. The first time it runs, it requires her to choose a PKCS#12 file that contains the key pair matching her Greenpass certificate chain.

The applet forces a delegator to verify her guest's identity by choosing his visual fingerprint from among 16 candidates. It generates the 15 incorrect fingerprints by feeding random 128-bit strings (the same length as an MD5 hash) to the Visprint generator. The original delegation applet, on the other hand, showed a single fingerprint and simply asked the delegator if it matched the guest's. The newer approach does not absolutely prevent the delegator from skipping the verification step—she could run the applet twice, for example, and see which image is common to both sets of fingerprint candidates—but it makes it more tedious to skip verification than to not skip it, an important consideration in designing user interfaces for security systems. (As an additional benefit, the use of visual, rather than hex, fingerprints means that *not* skipping verification is *not* tedious. In fact, users at the Greenpass pilot appeared to enjoy comparing one another's visual fingerprints.) In order to create visual fingerprints in Java, I ported an adaptation of the Visprint program [52], written in C, to a Java class.

Having the applet itself create visual fingerprints increases the security of the delegator's private key. If the delegation Web app itself provided fingerprint verification and

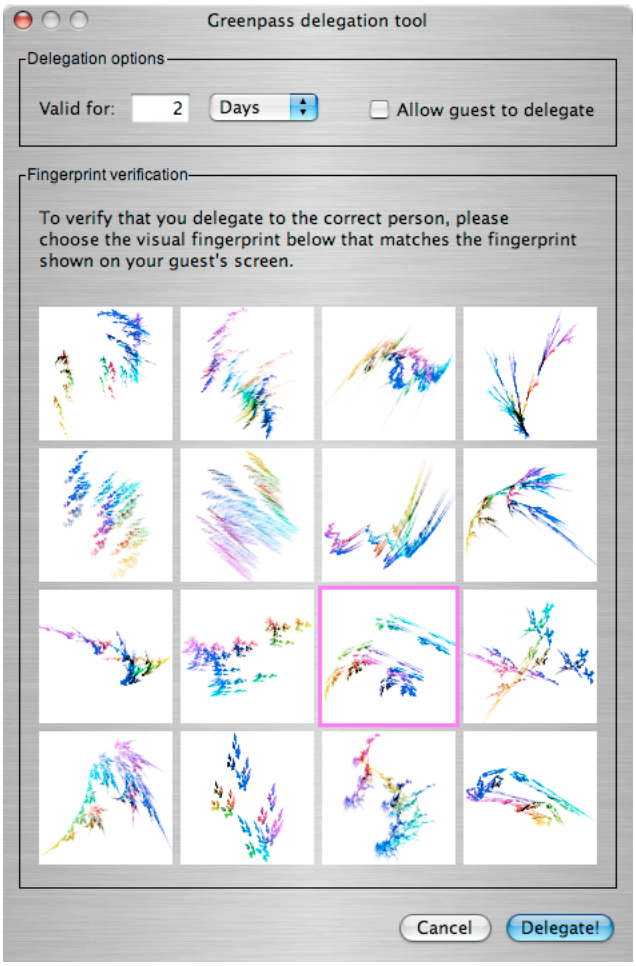

Figure 3.4: A screenshot of the Greenpass delegation applet. This Java applet allows a delegator to verify her guest's visual fingerprint and specify a validity interval for the guest's new certificate, as well as specify whether it should contain the SPKI *(propagate)* flag.

merely sent the verified public key hash to the applet, then an adversary could hijack the applet. Specifically, the adversary could easily create a fake Web application that sends our applet unverified public key values. The delegator's Java environment would trust the applet itself to use her private key—the applet is signed—but the delegator would have no way of verifying to whom the applet was about to issue a certificate.

The delegation applet uses standard Java cryptography functionality for signing operations, and custom S-expression generation routines for certificate generation. The wrapper class it uses to access the delegator's PKCS#12 keystore is general enough that another programmer, given time, might be able to wrap it around platform- or browser-specific keystores, such as provided by Windows or Mozilla, by including a specialized JNI (Java Native Interface) library for each target platform. The applet originally relied on an older Java SPKI/SDSI library from MIT [63], but that library included far more SPKI/SDSI functionality than required, and further depended upon another third-party cryptographic library. Since the applet performs only one limited task—generating and signing a certificate delegating from one hash to another—and uses only canonical S-expressions, I added code to generate the needed S-expressions internally. This step reduced the download size of the applet by a factor of 10.

3.4.4 The Greenpass front page

The Greenpass front page provides a rendezvous point and switchboard of sorts for all Greenpass users. When someone connects to it, it uses the various cookies discussed earlier to identify that user and detect whether he is authorized, and if so, whether as a guest or as a delegator. It then displays the user's status along with appropriate options for that class of user: it displays a short explanation of Greenpass to unauthorized users and offers to redirect them to the guest introduction Web app, and it allows authorized delegators to go to the

delegation Web app. It also provides all authorized users (delegators or guests) two utility options: they can view their SPKI certificate chains as advanced (i.e., human-readable) S-expressions, and they can clear all Greenpass-related cookies from their browsers.

The front page also handles nearly all cookie manipulation for the Greenpass client tools. (As mentioned earlier, however, the guest introduction Web app can retrieve certificate chains for waiting guests, and it is responsible for initially setting the *greenpass_waiting* cookie after guests introduce themselves.) When a user connects, the front page Web app goes through the following decision process:

- If the cookie *greenpass_hash* does not exist in the user's Web browser, the user is unauthorized. Otherwise, use it as a SPKI principal with which to identify the user.
- If the cookie *greenpass_waiting* is present, check the authorization cache to see if a new certificate chain is available for this principal. If so, install it in the *greenpass_chain* cookie in the user's Web browser, then query the authorization cache again to determine if this user has been authorized as a guest or as a delegator. If no chain is available, the user must continue waiting.
- If the cookie *greenpass_waiting* is *not* present, check the authorization cache for the user's status (unauthorized, guest, or delegator).
- If checking the certificate cache fails, look for the cookie *greenpass_chain* in the user's browser. Try to put it back in the authorization cache, which will determine if the chain is valid and whether it authorizes the user as a guest or as a delegator.

If the user is newly authorized (has just received his certificate chain), the front page displays a message to this effect and explains which SSID to connect to for full access. If the user is re-authorized, it explains that his certificate had expired from the authorization cache, but that he should now be able to obtain wireless access as before.

The authorization cache

The authorization cache is written in Java so as to take advantage of Sameer Ajmani's JSDSI library [53]. The term "cache" is perhaps something of a misnomer, because at present it provides not just certificate storage, but all the SPKI verification logic needed by the Greenpass RADIUS server and client tools. It exposes three public methods over XML-RPC:

- *getSubjectStatus()* receives a user's public key fingerprint, and returns the user's current status—unauthorized, authorized guest, or authorized delegator—according to the cache's current contents.

- *getCertChain()* receives a user's fingerprint and, if a certificate chain for that user exists in the cache, returns it.

- There are two (overloaded) versions of *addCertChain()*. The first receives a delegator's certificate chain and a freshly-signed certificate for a guest. It concatenates the two certificate chains to create a new certificate chain for the guest, then checks the validity of the resulting chain by using JSDSI to compose each certificate/signature pair of the chain in order. If the resulting proof sequence propagates Wi-Fi access from the local SOA to the guest, then the cache verifies the signatures on the chain and, if successful, stores the certificate chain (along with whether the subject in question can delegate further) in an internal data structure. The second version of *addCertChain()* does the same, but receives only a single, complete certificate chain. The delegator Web app uses the first version to authorize a new user after a delegator creates a SPKI certificate; the front page uses the second version to re-authorize a user whose certificate chain has expired from the cache.

The most important thing that cookie-based authorization adds to Greenpass is the ability for users to re-authorize themselves by pushing certificate chains to our authorization cache. (This will prove most useful in a decentralized setting with multiple access points, RADIUS servers, and authorization caches; see Section 5.3 for further discussion.) Figure 3.5 illustrates the re-authorization process, which takes place as follows:

1. A user whose certificate chain is not in the authorization cache visits the Greenpass front page. His Web browser automatically presents his public key fingerprint (hash) and certificate chain to the Web server as HTTP cookies.

2. The front page Web app, using the user's fingerprint as his identity, queries the authorization cache to find out what his status is; it returns the status code UNAUTHORIZED.

3. The front page Web app, noticing that the user has presented a certificate chain in an HTTP cookie, responds by sending his certificate chain to the authorization cache, which validates the chain and, if successful, returns the status code AUTHORIZED. (Actually, the status code also contains the user's role—delegator or guest—depending on whether his certificate contains the SPKI *(propagate)* flag.)

The front page then informs the user that he has been re-authorized. Once a user's certificate chain is in the authorization cache, the Greenpass RADIUS server will allow that user onto our unrestricted VLAN after using EAP-TLS to find his public key value.

One problem not sufficiently addressed by our HTTP-based authorization is that of roaming clients; see Section 5.3 for further discussion.

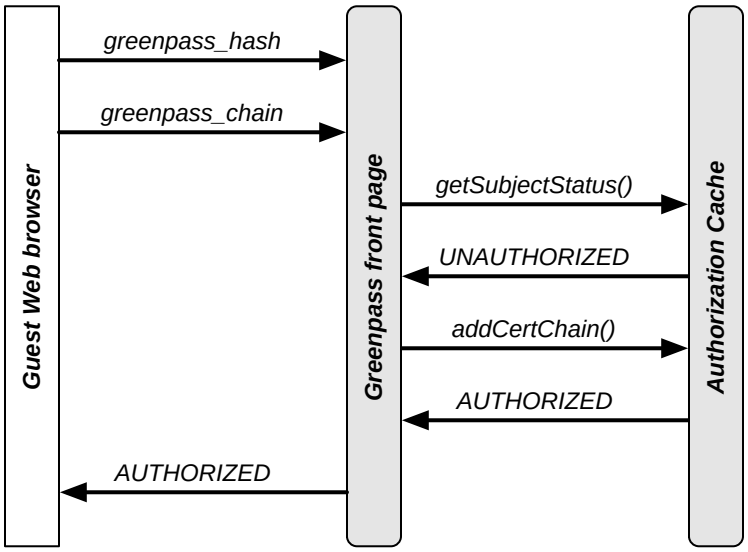

Figure 3.5: If a user has a valid SPKI certificate chain stored in his Web browser, but that chain is not in the authorization cache, then he can visit the Greenpass front page to instantly *re-authorize* himself. The process takes place as follows. (1) The user visits the Greenpass front page; his Web browser automatically presents his public key fingerprint (hash) and certificate chain to the Web server as HTTP cookies. (2) The front page Web app, using the user's fingerprint as his identity, queries the authorization cache to find out what his status is; since it does not contain his certificate chain, it returns the status code UNAUTHORIZED. (3) The front page Web app, noticing that the user has presented a certificate chain in an HTTP cookie, responds by sending his certificate chain to the authorization cache, which validates the chain and, if successful, returns the status code AUTHORIZED.

3.5 Summary

The client tools described in this section are accessible to both delegators and guests via our ***Greenpass Web applications***. These tools enable a guest and a delegator to carry out the following steps:

- A guest can obtain a temporary key pair and X.509 certificate, if he needs them, from our ***dummy CA*** using standard browser-based PKI enrollment functionality.
- A guest can then *introduce* his public key—contained in his existing X.509 certificate or a temporary one—to our ***introduction cache***.
- After a guest introduces himself, the Web application he is connected to displays a *visual fingerprint* of his public key that a delegator can use to verify his identity. He then waits patiently for his delegator to finish issuing a new certificate to him.
- Our Web applications allow a delegator to retrieve her guest's public key value from the introduction cache.
- The Greenpass ***delegation applet***, written in Java, displays the visual fingerprint of the public key she has retrieved so she can compare it to her intended guest's actual public key value. (Other hash comparison methods could be added.) It also performs the cryptographic operations necessary to sign a new SPKI certificate with the delegator's private key.
- The Web application server appends the guest's new certificate to the delegator's existing certificate chain in order to form a complete chain from the local source-of-authority to the guest's public key. It then places the new chain in an ***authorization cache*** where (1) the guest can pick it up and (2) the Greenpass RADIUS server can

"see" the new credential and grant the guest full access to the wireless network the next time he associates to an access point.

- Finally, the Greenpass Web applications can store a guest's credentials directly in his Web browser as an HTTP cookie. This improvement enables "push" authorization of a sort: even if the guest's certificate chain expires from the authorization cache, the Greenpass Web applications can retrieve it from this cookie, process it, and mark the guest as re-authorized.

We have tested our client tools and the Greenpass RADIUS server as described in Powell's pilot report [87]. The client tools do, in fact, allow a complete delegation process at present, and the RADIUS server interprets credentials from the authorization cache as expected: it lets guests with untrusted X.509 certificates, but valid SPKI credentials, onto our unrestricted Wi-Fi network.

Two of the biggest weaknesses of our test setup were as follows:

- Guests must tell their wireless clients to connect to a different SSID before gaining access to our unrestricted VLAN, as discussed in Section 2.4.4. The final page of the guest Web app includes a note to this effect.

- Our test setup includes only one access point, RADIUS server, Web server and authorization cache, so we cannot test the full potential of "push" authorization via HTTP cookies. We have, however, tested re-authorization by flushing the authorization cache of all existing credentials.

Chapter 5 discusses future work related to Greenpass, beginning with a discussion of existing weaknesses in the client tools presented here and suggestions for improvements. Section 5.3 is of particular note, as it describes an example architecture where "push" authorization via HTTP cookies should have a positive impact on scalability.

Chapter 4

Related work

This chapter describes work related to Greenpass and the client tools just described. Section 4.1 describes a number of existing solutions that provide access control to Wi-Fi networks along with guest/visitor access of some form, and also cites two sources that have suggested, but not implemented, delegation-based solutions; Section 4.2 discusses a number of other SPKI/SDSI-based research projects, including some that implement access control to resources other than a Wi-Fi network; Section 4.3 compares and contrasts SPKI with a number of competing authorization certificate formats; and finally, Section 4.4 discusses work related to the problem of public key introduction without a trusted third party, including other visual hashing work.

4.1 Wi-Fi guest access

Many corporations, research groups, and open-source projects recognize the need to offer hassle-free wireless connectivity to guests. This section discusses a number of existing or proposed solutions to the problem and compares them to Greenpass. Most of these solutions provide a different form of "guest access" than Greenpass: guests are usually

allowed in without authentication but given access only to an Internet gateway. Greenpass, on the other hand, can restrict access even to the Internet—and can give users access to internal network resources once they are authorized.

4.1.1 Captive portals

Captive portals let users authenticate to a Wi-Fi network using only their Web browsers. Several commercial vendors sell captive portal solutions to Wi-Fi hotspots such as cafés, bookstores, and airports. Additionally, members of grassroots community network projects have created a number of open-source captive portal implementations; such groups believe that some form of user authentication and monitoring are needed both to protect their open networks from abuse (see Shand [96] for a discussion of the risks), and in some cases to provide greater access privileges to members or founders.

The NoCat Community Wireless Project [70] maintains *NoCatAuth*, a popular set of open-source captive portal tools whose operation is well-documented and fairly representative of captive portals in general. Their toolkit combines two core components: a wireless gateway (*NoCatGW*) that provides access to the Internet or other resources, and an authentication service (*NoCatAuth*) that tells the gateway what filtering rules to apply to which clients. A typical captive portal session might look something like this:[1]

1. A client requests and immediately receives a DHCP lease.

2. The gateway redirects the client's HTTP requests to the authentication service, which displays a secure (HTTPS) login Web page asking for username and password, or perhaps some other form of authentication.

3. Upon successful login, the authentication service sends the gateway a signed message

[1] Parts of this list are quoted directly from the NoCat unofficial RFC [71].

telling it the client's login, current MAC address, and authentication status ("yea" or "nay").

4. The gateway modifies its firewall rules to reflect the user's new status. In NoCatAuth's particular model, "co-op" or "priority" users obtain a higher privilege level than unauthenticated ("public") users.

5. The client must periodically re-authenticate. NoCatAuth opens a small side window in the user's Web browser that periodically refreshes the login page in order to provide automatic re-authentication.

NoCat can be used to enforce fairly complex restrictions and separation of user classes, or simply to make users agree to a terms-of-use license before obtaining full network access.

NoCat and other captive portals share a number of features with Greenpass, such as a Web-based registration service and an authentication/authorization service that tells some other entity (a RADIUS server in Greenpass, a gateway in NoCat) whether a given user should be granted access. The captive portal method of access control—by MAC address—does not provide the same level of security as 802.1x, however, nor does it provide client-to-AP encryption, which can help protect clients' traffic from eavesdroppers. More importantly, existing captive portals do not offer anything analogous to SPKI delegation, instead requiring administrators to create new user accounts manually. Some NoCat setups offer guest access to an Internet gateway—perhaps with limited bandwidth or access to only some ports—in "public" mode without authentication, but Greenpass aims to have guests authenticate and go through an authorization procedure before obtaining *any* access to either the local network or an Internet gateway.

The PersonalTelco Web site offers an extensive list of both open-source and proprietary captive portal implementations [84].

4.1.2 802.1x-based solutions

Quite a few corporations provide large-scale 802.11 packages to enterprises and universities, and some are beginning to offer 802.1x authentication for increased security, along with basic access for guests. Most of these commercial systems are variations on the following theme:

- Guests do not need to authenticate to obtain access to an Internet gateway, but do remain outside a firewall that protects the organization's internal network.
- Local users must authenticate, usually via 802.1x or a VPN connection, in order to obtain access to internal resources.

This section describes two commercial solutions, from Nomadix and Bluesocket, as examples of typical commercial WLAN security systems that provide guest access via variations on the above approach. It also describes a fairly novel guest-access solution used by SURFnet, the Netherlands' national network for higher education and research.

Nomadix

Nomadix sells their Nomadix Service Engine (NSE) software package to various providers of public-access Wi-Fi networks, and also sells a line of its own wireless gateway devices based on NSE. Of their various advertised applications, Enterprise Guest Access shares many common goals with Greenpass: according to a Nomadix whitepaper [72], its stated goal is to let enterprises "provide easy, hassle-free Internet access to people visiting their offices, without decreasing the security of their own Local Area Network." It appears that Nomadix's solution to guest access is to place open, guest-accessible access points in designated "Hot Zones" such as lobbies and meeting rooms, keeping all traffic from these APs outside the enterprise firewall. Nomadix supports 802.1x authentication in their

other products, but it is unclear from the whitepaper whether local users can obtain internal access by authenticating via 802.1x to APs otherwise set aside for for guests.

Bluesocket

Bluesocket's enterprise Wi-Fi configuration with guest access appears very similar to Nomadix's. Their various whitepapers [12, 11] (or, as they call them, "bluepapers") make it clear that they support 802.1x authentication for internal users, but their FAQ Web page [10] also mentions an HTML-based login page for guests, suggesting a captive portal approach. Bluesocket appears to support a number of guest access options, including requiring them to enter their email addresses to log in, or supplying temporary account numbers to guests via pre-paid scratch cards. Bluesocket's gateways can also place guests in a configurable "walled garden" to allow them access only to certain sites (one whitepaper suggests that businesses might want to block guest access to competitors' Web sites).

SURFnet

SURFnet, the Netherlands' national computer network for higher education and research, offers a fairly interesting approach to guest access via *RADIUS proxying*, which they describe on a Web page [101] and in a presentation they have made available online [102]. SURFnet allows wireless users to authenticate via 802.1x using any of a number of EAP inner authentication handshakes, but modifies the RADIUS server to find users' authentication and authorization materials as follows (taken from the SURFnet Web site):

> Locally, all Access Points (APs) are connected to a local RADIUS server that does the lookup of local users. When a guest arrives and tries to log on using a different realm, for instance "john@alfa-ariss.com," the local RADIUS server cannot find the name in its database and forwards the authentication request to

a central RADIUS proxy server. This server contains a table of realms, and the addresses of the corresponding RADIUS servers. It forwards the request to the RADIUS server at Alfa&Ariss, which looks up "John." If John is found and his credentials are correct, the RADIUS server tells the UTwente server (through the proxy server) that John is known and allowed to use WLAN facilities, and the UTwente RADIUS server tells the AP to open up for John. The AP can put John into a Virtual LAN (VLAN) that does not allow him to access protected UTwente resources, but only gives access to the SURFnet backbone.

SURFnet itself provides the RADIUS proxy server(s). Setting up a RADIUS proxy that is trusted by multiple organizations is similar to setting up a bridge CA, except that the RADIUS proxy must be online. Unlike a bridge CA, distributed RADIUS proxying can provide authorization as well as authentication information. Compared to Greenpass, SURFnet's approach (or a bridge CA) has two drawbacks: first, organizations must agree upon and trust a third party to provide user authentication materials; and second, guests from outside the group of affiliated organizations cannot obtain access.

4.1.3 SPKI-based and related solutions

To our knowledge, Greenpass is the first working SPKI/SDSI-based solution to Wi-Fi access control. A number of researchers, however, have proposed and partially implemented similar solutions using either SPKI/SDSI or similar certificate formats.

Koponen et al. [56] describe a simulated system of access control to a café's WLAN based on SPKI delegation. In their example scenario, a cashier issues a SPKI certificate to a paying customer granting her access to the café's network for a predetermined time interval. The customer both introduces her public key value and receives her new SPKI certificate via an infrared link. When she wishes to connect to the network, she must authenticate to

an access controller and present her SPKI certificate. Koponen et al. use XML-encoded SPKI certificates [81] instead of S-expressions.

Koponen et al. appear to have only created a simulation of their system using Java RMI calls, and at the time of their paper had not yet integrated their system with a Wi-Fi network's access control layer. They do not consider how guests are to connect to the network without a custom handshake, nor do they suggest the use of visual hashing when considering non-infrared introduction channels.

Nikander [69] describes an 802.1x-based system for authorization and charging in Wi-Fi LANs, implemented using FreeBSD. At one point in his paper, Nikander suggests extending his system to support delegated authorization of sorts via KeyNote certificates [8] (see Section 4.3.4 below): specifically, he suggests that the founders of a community Wi-Fi network project could issue KeyNote certificates stating that any two existing members can authorize new members. He points out, however, that this system would require custom client software.

UPnP

The UPnP (formerly Universal Plug-and-Play) Forum [105] defines a SPKI-based solution to guest access in home or small-office WLANs, although it does not allow delegation in the same sense that Greenpass does. UPnP seeks to ease the configuration and use of networked devices by defining a uniform interface of SOAP [110] actions for each device class, and allowing users to access those actions using GUI-based software called a *control point*.

The UPnP device security standard [26] "provides the services necessary for strong authentication, authorization, replay prevention and privacy of UPnP SOAP actions." When a factory-fresh device is first powered up, its owner can use a UPnP control point called a

security console to take ownership of it. The owner can then give other users permission to access certain SOAP actions on that device by adding those users (more accurately, their control points) to its internal access-control list (ACL). If the device does not have sufficient storage for an ACL, however, the owner can send authorization privileges to its users by issuing certificates. UPnP ACLs and certificates appear to use an XML-encoded SPKI syntax, right down to the ability to issue both authorization certificates and SDSI-style name certificates.[2] According to the device security standard, devices can optionally support delegation of privileges by non-owners.

The UPnP WLAN access point standard [108] combines with the UPnP *link authentication service* [58] to provide 802.1x authentication with guest access. In the UPnP flavor of wireless guest access, an AP contains its own, internal EAP server whose authentication database can be manipulated via a UPnP control point. Suppose Alice has permission (via the device security model discussed above) to manipulate this authentication database, and Bob visits her home and requests wireless access for his own device. Alice can grant him access as follows:

- Bob connects to the AP via 802.1x, using a standard EAP method: perhaps EAP-TLS, or perhaps a password-based EAP method such as Cisco's LEAP [18]. Bob sends both his EAP identity (a username that EAP always requires, even with EAP-TLS) and a credential (such as an X.509 certificate or password).

- The AP's internal EAP server notes that Bob is an unrecognized user and adds Bob's EAP identity, attempted authentication type, and credential to an entry in its internal database. It marks this entry as "pending."

- The EAP server notifies Alice (via her control point) that a new user is trying to

[2]Carl Ellison, the primary creator of SPKI, has been involved in UPnP, as evidenced by his work on UPnP's device security standard and its *security ceremonies* [27].

enroll.

- Alice decides whether or not to accept the new user. If Bob's credential was a password or similar shared secret, then Alice must enter the actual shared secret she and Bob have agreed upon. If Bob's credential was an X.509 certificate, then she must verify his public key fingerprint.

The UPnP approach resembles Greenpass in a number of ways:

- Local users can authorize new users (or new devices with different credentials) to access the network.
- Authorizing a new user involves making sure his credentials (public key or shared secret) match the credentials the intended user really has.
- Most importantly, it appears that users can authenticate using their existing credentials rather than obtaining new credentials from a local authority. Guests who use passwords will probably not want to take advantage of this capability: it requires giving their existing passwords to the local user. It appears, however, that guests could use existing EAP-TLS certificates.[3]

The UPnP approach, however, also differs in a few important aspects:

- SPKI is used to control who may modify the AP's authentication database, not to control who may access the network itself. To do the latter requires either modifying the access-control handshake, or separating the authentication channel from the authorization channel (as we have done in Greenpass).

[3]This capability is fortunate, since the UPnP Forum has yet to describe an easy administrative interface for certification authorities.

- Since local users control authorization directly at the access point and guests carry only authentication (not authorization) credentials, the UPnP approach does not scale to university- or enterprise-scale environments with many access points.[4] To scale, a guest access solution must provide a way for guests to carry credentials and "push" them to the appropriate access controller.

- It is not clear whether any existing access point implements the UPnP link authentication service.

As with most UPnP services, the link authentication service appears to be aimed at home or small-business users. It is most appropriate for granting temporary access to guests in one's home or for permanently adding new devices to a small, protected WLAN.

4.2 SPKI/SDSI

Although SPKI/SDSI is not as popular as X.509, a considerable number of researchers have built prototype systems that use SPKI/SDSI to handle authorization scenarios that X.509 name certificates cannot. This section describes several that have influenced our ideas regarding Greenpass.

4.2.1 SPKI/SDSI-based access control

MIT's Project Geronimo,[5] described by Maywah [60] and Clarke [19], uses SPKI/SDSI to control access to objects on a Web server. Geronimo includes an extension to the Apache

[4] Freeman et al. [36] define a UPnP interface for configuring a RADIUS server, but it does not appear to support the same functionality as the internal EAP server described above.

[5] MIT's SPKI/SDSI-based Geronimo project is not to be confused with the Apache Foundation's own Geronimo project, an open-source J2EE container.

Web server that processes SPKI/SDSI ACLs in directory-specific *.htaccess* files, only allowing principals listed in such an ACL to connect to the directory to which it applies. (The required SPKI/SDSI verification code is built-in to this Apache module as well.) On the client side, a custom Netscape plugin handles challenges for SPKI/SDSI authorization by building and returning the necessary certificate chain; this plugin also signs a challenge text to authenticate the user.

A number of researchers have designed trust- or credential-management systems for access control based on SPKI/SDSI. Eronen and Nikander [33] describe several SPKI/SDSI-based enhancements to both authorization and authentication in Jini, a Java-based distributed computing environment. Their enhancements include the following:

- Jini services can challenge users for SPKI authorization credentials before granting them access. This approach allows delegation-based access control to arbitrary Jini services.
- Jini users can authenticate the services to which they connect.
- Jini services are typically accessed via *proxies*, downloaded Java bytecode objects. Users can authenticate a proxy—i.e., they can make sure it actually represents the service it says it does.
- Finally, users can delegate permissions to individual applications on their devices. When an application needs to use a proxy to access a remote service, it delegates its own permission to a temporary key and passes a handle to that key to the proxy. In this way, the proxy obtains the user's permission to access a particular service.

Eronen and Nikander use TLS for authentication of Jini clients and servers alongside SPKI/SDSI for authorization, much as Greenpass combines SPKI authorization with EAP-TLS authentication of wireless clients.

Canovas and Gomez [15] describe a distributed credential management system that uses SPKI/SDSI name certificates and authorization certificates. The system contains naming authorities (NAs) and authorization authorities (AAs) from which entities can request name and authorization certificates, including certificates that permit the entity to make requests of further NAs and RAs. The system takes advantage of both name certificates that define groups (i.e., roles) and authorization certificates that grant permissions to either groups or individual entities.[6]

Finally, here at Dartmouth College, Howell and Kotz have explored both formal semantics for SPKI [48] and end-to-end authorization using SPKI [49]. In the latter, Howell and Kotz describe a number of access-control applications built using their SPKI-based *Snowflake* authorization architecture, including a protected Web server similar in concept to MIT's Project Geronimo (see above). Additionally, in the *SPADE* project, Nazareth [66] and Nazareth and Smith [67] have investigated how to define attribute release policies for the Shibboleth federated administration system [97] using SPKI/SDSI.

4.3 Other authorization and delegation systems

Various researchers and working groups have produced other authorization certificate formats that compete with SPKI/SDSI. Each format has its own set of advantages and disadvantages; to truly compete with SPKI/SDSI for our purposes, however, a certificate format must at least provide delegated authorization.

[6] This paragraph is adapted from Goffee, Kim, Smith et al. [42].

4.3.1 X.509 attribute certificates

The IETF's PKIX working group has published an RFC [34] that defines an X.509 *attribute certificate* (*AC*) standard. Attribute certificates are related to X.509 name certificates; an AC, however, includes no public key for its *holder* (i.e., subject), but instead binds an attribute to it. As mentioned in Section 2.3.1, an attribute may contain nearly anything relevant to the relying application; X.509 attributes must conform to X.509's ASN.1 syntax and its OIDs (object identifiers), however, so they are less readable than SPKI tags. The motivation behind separate attribute certificates is that a user's name certificate may be valid for a long time, whereas his attributes and permissions might change quickly. An X.509 attribute certificate can identify its holder by distinguished name, by the name of the issuer of his name certificate and its serial number according to that CA, by the hash of the holder's public key, or by the hash of the holder's identity certificate.

The primary reason we chose SPKI/SDSI over X.509 attribute certificates for Greenpass is the latter's current lack of support for delegation. While the attribute certificate RFC recognizes that AC chains could be used for delegation, it does not recommend doing so at this time, citing the complexity of processing such chains. In particular, the RFC provides no standard way to intersect attributes in the way that SPKI tags can be intersected. We suspect that a well-designed standard for delegation-friendly attributes could provide essentially the same functionality as SPKI/SDSI (albeit with a heavier-weight syntax); we chose SPKI/SDSI because it already meets our needs. Additionally, as mentioned in Section 2.4.3, the EAP-TLS handshake cannot transmit attribute certificates, so had we chosen them, guests still would have needed to carry and present them via HTTP cookies or some other out-of-band means.

The PKIX working group has also released an Internet Draft that proposes two WLAN-related extensions to X.509 [47]. Specifically, their draft specifies the following:

- Two new *KeyPurposeId* values to be used in the *ExtendedKeyUsage* field of X.509 name certificates. The two new values specify that a certificate may be used for EAP authentication over a PPP link or for EAP authentication over a LAN (EAPOL), respectively.

- A certificate extension that specifies to which SSIDs, in a wireless LAN setting, a given certificate can be used to authenticate. This extension allows a user's 802.1x client to automatically choose the correct certificate with which to authenticate to a given access point, based on the latter's SSID. The user may still need to choose a certificate manually if he/she authenticates to multiple access points that rely on different CAs, but happen to have the same SSID.

The draft also specifies an attribute certificate attribute that serves the same purpose as the SSID extension.

These X.509 WLAN extensions *could* be interpreted as authorization fields. A certificate with the appropriate combination of these extensions might be construed as saying, e.g., "the subject is authorized by the issuer to connect to the AP with SSID 'Foobar3'."

4.3.2 X.509 proxy certificates

Welch et al. [109] describe a *proxy certificate* format that allows one entity to delegate some subset of its privileges to another entity (Tuecke, Welch et al. have recently published a PKIX draft profile of proxy certificates [104] as well). Proxy certificates were originally designed for use in the Globus Project's [40] Grid computing architecture, where, for example, long-running processes often need to inherit some of a user's permissions in order to access needed resources. Proxy certificates can also support many of the same actions as SPKI delegation, such as person-to-person delegation, or delegation from a user's master public key (with highly-secure private key) to a short-term key.

A proxy certificate conforms to the same format as a standard X.509 name certificate, with a few exceptions:

- The issuer of a proxy certificate need not be a CA; it might be an *end entity*, i.e., the holder of a standard, non-CA X.509 certificate or even the holder of another proxy certificate.

- To create the subject name of a proxy certificate, the hash of the subject's public key is appended to the issuer's distinguished name as a *relative distinguished name* (*RDN*) component.

- A proxy certificate includes a *proxy certificate information* (*PCI*) extension that defines what subset of the issuer's privileges should be conveyed to the subject. Instead of defining a new syntax expressing delegation policies, the PCI extension allows the issuer to include a statement from one of several existing policy languages.

Proxy certificates resemble X.509 certificates so closely that some authentication libraries may be able to process chains of mixed X.509 and proxy certificates, with two caveats. First, the library must not reject proxy certificates simply because they are issued by end entities instead of CAs. Second, Welch et al. point out that their initial implementation does not use complex policies in the proxy certificates' PCI fields, but merely a special policy type that conveys all the issuer's rights to the subject. Only a modified security library could correctly process chains of proxy certificates that contain complex policies.

4.3.3 PERMIS

The *PERMIS* project defines a *Privilege Management Infrastructure* (*PMI*) that uses a *role-based access control* (*RBAC*) system based on a combination of X.509 attribute certificates

and an XML-based policy language. Roles provide a simple way to define sets of permissions for broad classes of users. PERMIS even supports *hierarchical RBAC,* where superior roles can inherit the attributes of their subordinate roles. Role hierarchies are defined in the root PERMIS policy for a particular domain, while the attributes of a particular role are assigned using *role specification* ACs placed in an LDAP [114] entry for that role. *Role assignment* ACs, in turn, assign roles to users. PERMIS supports delegation (an AC can include an integer control on delegation depth): a user with a particular role assignment AC can assign that role, or a subordinate role, to another user. Although ACs of PERMIS users are stored in an LDAP directory, it appears that PERMIS could easily be adapted to a model where users "push" their role assignment ACs to access controllers.

4.3.4 KeyNote

KeyNote [8] and its predecessor, *PolicyMaker* [9], provide decentralized trust management via signed policy assertions, which are essentially equivalent to authorization certificates. A KeyNote assertion delegates authority from an *authorizer* to one or more *licensees*; as with SPKI, both the authorizer and the licensee may be identified by their public key values. An application that relies on KeyNote must obtain assertions (via either a "push" or "pull" method) and pass them to the KeyNote compliance checker along with a set of name/value pairs called an *action attribute set*. The action attribute set describes the access request a user is making of the application. KeyNote assertions contain a set of conditions; the compliance checker ensures that the action attributes passed by the application match these conditions.

The KeyNote RFC [8] specifies that compliance checking involves building a directed graph from a root assertion called "POLICY" (presumably, this assertion is an unsigned statement internal to the application) to at least one of the principals that requested an

action. Therefore, KeyNote allows delegation chains just as SPKI does. Note, however, that KeyNote does not include a way to limit whether a principal may further delegate its permissions; the application has to decide which principals it will ultimately trust as direct issuers of assertions. The KeyNote RFC does not give any specific guidelines as to how implementations should process certificate chains.

4.3.5 XML-based authorization

Researchers have proposed a number of XML-based approaches to authorization problems. Three fairly popular XML-based languages are SAML, XACML, and XrML.

SAML (*Security Assertion Markup Language*) [73, 76] was designed largely to support *single sign-on* (*SSO*) between Web applications run by different parties. Using SAML, Alice can send Bob an assertion that contains statements about a client, Carla, that Alice and Bob have in common. SAML assertions can contain three types of statement:

- an *authentication statement* tells Bob that Alice authenticated Carla and also includes details about the time and method of the authentication;
- an *attribute statement* tells Bob that Carla has some attribute, such as "Gold" membership status, in Alice's domain.
- an *authorization statement* tells Bob what Carla is allowed to do.

SAML also defines ways for Alice (the *asserting party*) to send Bob (the *relying party*) her assertion. Either Alice can send Bob her assertion behind the scenes using a SOAP-based protocol (a "pull" model), or Alice can send Carla's Web browser a digitally-signed assertion that it later sends to Bob via an HTTP POST request (a "push" model). SAML suggests the use of HTTPS to protect assertions used in the "pull" model from tampering

or replay attacks. (Presumably, Carla's communication with either Alice or Bob must be secured via HTTPS as well, or an attacker could obtain her session token.)

XACML (*eXtensible Access Control Markup Language*) [74, 77, 100] is an XML-based language for defining access-control policies. Of the policy languages and certificate formats already discussed, it is most similar to KeyNote, but XACML is more verbose (a drawback[7]) and contains more constructs with predefined semantics (an advantage). XACML's typical use model involves a *policy enforcement point* (PEP) receiving an access request from some user and sending an XACML description of that request to to a *policy decision point* (PDP). The PDP compares the request to a policy that is structured as follows:

- Each policy contains a number of *rules*.
- Each rule has a *target* that specifies what *resources*, *subjects*, and *actions* must be involved in a request for that rule to apply to it. Each resource, subject, or action is defined by a number of *attributes*.
- Each rule also contains a *condition* that, if met, causes the rule's *effect* (permit or deny) to be fired and combined with the effect of other rules in that particular policy.

Each XACML policy specifies a *combining algorithm* that the PDP should use to combine the results of the enclosed rules. Policies themselves can also have targets, and can be further combined into *policy sets* (which themselves can have targets). A PDP determines whether a particular rule is applicable (based on its target) to the request described by the PEP, then determines whether that rule's condition is true or false and fires the rule's desired effect if its condition is true. Inapplicable rules return a special *NotApplicable*

[7] Being XML-based, however, XACML probably lends itself more readily to the construction of an automated policy editing tool. On the other hand, raw KeyNote assertions are compact, highly readable, and highly writable, none of which are distinctions any XML-based language can claim.

value. XACML policies can also be distributed, in that one policy may refer to a different policy stored in a different location and possibly created by a different entity.

XACML does not define a protocol or assertion format for transporting requests, policies, and/or responses among a PEP, PDP and other entities. The XACML technical committee of OASIS has, therefore, released a draft profile for using XACML policies within SAML 2.0 assertions [79].[8] This proposal augments XACML policies with the following features:

- a way to identify the assertion issuer,
- a way to add a validity period to the assertion, and
- a way to verify an issuer's digital signature on an assertion.

The resulting signed XACML/SAML assertion could be described as an XACML "certificate," since it has features analogous to those found in other authorization certificate formats.

XrML (*eXtensible Rights Markup Language*) [113] might be considered an agreeable middle ground between SPKI/SDSI's simplicity and XACML/SAML's flexibility and standardization. XrML's basic authorization carrier is the *license*, which is analogous to an authorization certificate. An XrML license is structured as follows:

- A license contains an *issuer* and one or more *grants*.
- A *grant* in turn contains four fields which the XrML Technical Overview [22] describes in the following words (italics mine):
  - the *principal* to whom the grant is issued,

[8] Note that I describe SAML 1.1 above because documentation for it is more complete; SAML 2.0 remains a draft at this point.

- the *right* that the grant conveys to the specified principal,
- the *resource* against which the specified principal can exercise or carry out this right, and
- the *condition* that must be met before the right can be exercised.

XrML licenses can identify principals by public key (as in SPKI), by other authentication credentials, or by various properties including X.509 subject name. Like a certificate, a license may be digitally signed by its issuer.

Neither SAML nor XACML currently supports delegation, although both technical committees appear to be considering it for future versions.[9] XrML supports delegation in much the same way as SPKI: a grant may contain a *DelegationControl* element that, when present, permits the holder of that grant to delegate his right to the given resource to other principals. Unlike SPKI, XrML can specify that a licensee may delegate to only those principals that appear in a particular set of principals.

Custom delegation semantics could certainly be added to SAML or XACML; Navarro et al. [65] discuss how to implement a constrained-delegation model in SAML, XACML, and XrML.

4.4 Introduction and visual hashing

Anytime one entity, Alice, needs to authorize an entity from a different domain, Bob, to do something, the problem of key introduction might arise. As discussed in Section 3.2.3, Alice needs a secure way to find out Bob's public value and either authorize him directly via that value or bind a meaningful name to his key. This section describes other research

[9] See the SAML Version 2.0 Scope and Work Items document [75] and the XACML delegation use-cases document [78] for details.

that explores the introduction problem, as well as other work related to the visual hashing approach we chose for Greenpass.

Ellison and Dohrmann [28] describe a situation in which a group leader wants to *invite*—i.e., add—a number of principals to a collaborative group, where membership implies access to resources intended only for that group. (The leader adds principals to his group by issuing SDSI name certificates; see Section 5.4.2 for a discussion of named SDSI groups.) The group in question might be completely ad-hoc in nature, including people from different organizations and people from across departmental boundaries within the same organization: therefore, the group leader cannot rely on a pre-existing naming or authorization infrastructure. As a result, the group leader must securely establish the identity of each invitee by learning his or her public key value. Ellison and Dohrmann choose visual hashing as a quick way for the leader to compare the public key value he receives for each invitee to the value actually stored on that person's machine. In Ellison and Dohrmann's approach, each of the two devices (the leader's and the invitee's) displays a short sequence of colored *flags* in synchronization with the other device; the leader must make sure that each flag displayed on his device matches the one shown on the invitee's device.

Balfanz et al. [6] describe a scenario in which a user wishes to print a sensitive document from his wireless device to a public printer in an airport. The user would like to authenticate the printer (so he doesn't beam his document to the wrong entity) and send his document to it via an encrypted channel (so nobody can eavesdrop on the transmission); he does not, however, already have a public key value for the printer, nor do he and the printer share an existing PKI that would allow him to authenticate it by name. Instead, Balfanz et al. propose a *pre-authentication* step in which the user's device learns the printer's public key (or a hash thereof) via a *location-limited channel* such as an infrared link. By Balfanz et al.'s definition, "location-limited channels have the property that human operators

can precisely control which devices are communicating with each other"; e.g., it would be quite apparent if an intruder tried to introduce his own key, rather than the printer's, to the user by holding his own device between the user's device and the printer during infrared pre-authentication. Pre-authentication, therefore, allows the user to bind a public key value to a particular printer he has identified as the one to which he wants to print. Balfanz et al. also discuss group pre-authentication and key-exchange protocols, which might be useful if, e.g., a group of people in a conference room wish to share confidential resources among themselves.

Perrig and Song [83] propose visual hashing as a possible solution for a number of problems in computer security. They argue that many security weaknesses result from failure to account for human factors, and focus their paper on two human weaknesses in particular:

- humans are slow and unreliable when comparing meaningless strings (such as hash values), and
- humans have difficulty remembering strong passwords.

Perrig and Song go on to formalize some properties that a visual hash algorithm must exhibit to be useful for security purposes, and suggest Andrej Bauer's *Random Art* [7] algorithm as an implementation worth considering. They briefly explore the use of visual hashing for CA root key validation (a specific case of public-key fingerprint comparison), and also suggest that a user could be authenticated by being asked to recognize a *portfolio* of visual hash images that only he knows. Dhamija and Perrig [24] expand on the portfolio concept in their *Déjà Vu* prototype, which uses Random Art images for user authentication.

Due to the ad-hoc nature of PGP introductions, a number of programmers have created alternative methods of fingerprint comparison for this purpose; all these systems transform

hexadecimal hash values into forms that humans can compare more easily. One such program is Visprint [43, 52], which we incorporated into Greenpass for guest key verification (see Section 3.2.3). Another is Raph Levien's PGP Snowflake generator [57]. Additionally, the PGPfone package [85] provides a procedure for transforming hash values into lists of words for the purpose of key comparison. This approach is designed to ease the comparison of hash values via telephone or other voice channels.

Chapter 5

Future work

Researchers and product developers have much exploration left to do in the field of decentralized, delegated authorization, SPKI-based or otherwise. I hope that Greenpass, by beginning to solve the real and widely-recognized problem of Wi-Fi guest access, might spur other researchers to investigate the field of delegated authorization and might influence various bodies to consider what standards are needed to better support it.

This chapter investigates a number of ideas for future research and development. First, it investigates current limitations of the Greenpass client tools: i.e., system aspects that must be changed or tested before Greenpass becomes a fully secure, scalable system for controlling Wi-Fi access. Section 5.1 discusses security improvements to the design of my client tools, Section 5.2 discusses the usability quirks we encountered in our pilot and suggests solutions, and Section 5.3 discusses how Greenpass could be decentralized on a network with multiple access points and, possibly, multiple RADIUS servers. Second, this chapter suggest ideas for future research: Section 5.4 explains how SPKI/SDSI's more advanced features could enable interesting new approaches to Wi-Fi authorization, Section 5.5 discusses how SPKI/SDSI might be used to control access to resources other than the network itself, and Section 5.6 suggests a "bake-off" between competing authorization

certificate formats and policy languages and offers my own speculations on the results of such an exercise.

5.1 Security considerations

Because my own security experience is limited to the Master's research described in this thesis, and because we intend our current Greenpass tools to be a demonstration of SPKI-based delegation rather than a fully deployable system, there are several potential security issues that should not be overlooked.

The XML-RPC communication between the Greenpass RADIUS server and my authorization cache needs to be secured in some manner. Otherwise, an intruder could inject a fake AUTHORIZED response code from the authorization cache's *getSubjectStatus()* method, causing the RADIUS server to accept a user without valid SPKI credentials. For our pilot, the authorization cache ran on the same machine as the RADIUS server, so that the two communicated via a socket on the 127.0.0.1 loopback interface. Loopback security, however, may be dependent on the networking stack that hosts the two components, or even on the host machine's particular networking hardware.

A more general solution would simply secure communication between the RADIUS server and authorization cache using a shared secret or via an authenticated HTTPS session (it is more important for the RADIUS server to authenticate the authorization cache than vice versa). Flanagan [35] proposes a system that allows two XML-RPC parties to authenticate each other using a shared secret, establish a session key, and add MACs to their ensuing messages using that session key. (Flanagan's proposal would provide message integrity only, not encryption.) I was unable to find an implementation of Flanagan's protocol, however. Time constraints also prevented me from learning how to tunnel XML-RPC sessions over an HTTPS connection.

Another solution would simply co-locate the authorization cache and the Greenpass RADIUS server without using socket-based communication. This approach could probably be most quickly implemented by wrapping a SPKI/SDSI library with a custom shared library with the same functionality as the current authorization cache, then linking the RADIUS server code with this custom library. One could use a C-based SPKI/SDSI library such as MIT's SDSI 2.0 [91] or Intel's CDSA [16] or wrap JSDSI—which may be the most active SPKI/SDSI library project—in a C interface using JNI and perhaps gcj [41].

Another consideration is the security of the Greenpass Web applications. A true duplication of Ellison and Dohrmann's [28] public-key introduction process would have both the guest's and the delegator's machines calculate and display a visual hash of the guest's public key. In our approach, the delegator's machine calculates a visual hash on its own since the delegation applet runs client-side, but our Web app generates a visual hash and sends it *over the network* to the guest's Web browser. If an adversary can get the guest's Web browser to display the adversary's own visual hash rather than the guest's, a delegator might delegate to the wrong person. (This issue arises not just with visual hashing, but in any approach where the device that displays or plays a hash representation is not the device that calculates it.) The entire guest introduction process takes place via HTTPS, but there may still be loopholes. A full investigation into these issues would need to consider specifics of the HTTP and HTTPS protocols; Web browsers do not appear to use SSL's built-in session support reliably and consistently enough to depend on it alone for session authentication.

Our Web apps contain a number of what appear at first glance to be security loopholes that do not actually lead to an adversary gaining access to the Wi-Fi network. The Greenpass front page identifies users via public key hashes stored in their *greenpass_hash* cookies. It could use HTTPS client authentication to ensure that their actual public keys

match the value of this cookie, but it does not. This loophole would allow, for example, somebody to modify his cookie store to contain the public key hash of an authorized delegator whose credentials are already in the authorization cache. The front page would then allow the impostor to access the delegation Web app and even run the applet, but he would be stopped when the applet asked for a keystore containing the correct private key for the identity with which the impostor tried to delegate. This "attack" could never create a valid signature without the real delegator's private key.

Another general security issue we need to consider is how vulnerable the Greenpass components are to denial-of-service (DoS) attacks. Other component-to-component communications, besides the critical channel between authorization cache and RADIUS server, might need to be secured to avoid an adversary from mounting a DoS attack by, e.g., performing large numbers of fake introductions or trying to introduce large numbers of invalid certificates to the authorization cache.

A great deal of research remains to be done on the security of visual hashes, particularly regarding which algorithms generate the most secure images. A "collision" of visual hashes might occur when two public keys result in images that look "equal" to a delegator, even if the underlying MD5 hashes are not equal. To discover how easily an intruder could generate a public key value with a visual hash that might fool a delegator, researchers might need to perform studies of human visual perception in addition to mathematical analyses of visual hash algorithms. Human factors must be considered when analyzing the security of any hash-comparison method, even plain hexadecimal key fingerprints.

Two general security issues we have not considered in Greenpass are accounting and revocation. Section 5.4 discusses how SDSI names might aid in accounting. Revocation might not be necessary if guest credentials are short-lived enough, but using short-lived certificates might reduce the flexibility and usability of Greenpass for situations other than

simple guest access.

Finally, the security of Greenpass is dependent on the correctness of the SPKI/SDSI libraries (and other libraries) on which it depends. This factor should not be overlooked, as the JSDSI library [53] is still under development.

5.2 Usability issues

Although SPKI authorization is conceptually quite simple, we have tried to support it using OS and browser functionality designed for X.509 authentication PKIs. This approach adds complexity. We could probably eliminate many of these issues by carefully testing our system and engineering around the various quirks that can arise. There are other elements of the client tools' interface that could be improved as well.

Some of the issues that arise are as follows:

- When a guest connects to our Web applications, he must establish an HTTPS session. His browser asks if he would like to trust our Web server. The solution here is simple: get any production Greenpass Web server certified by a widely-trusted CA such as Thawte or Verisign.

- The guest's HTTPS session also requires client authentication; depending on the guest's configuration settings, his browser may ask for permission to use his private key or, on some platforms, for a password to unlock his private key. This step may be alarming to some users; we need a way to warn users that it might happen and carefully explain that our tools are not trying to do anything malicious.[1]

[1]Many client certificates reveal information, such as name and email address, that the client might like to keep secret. We always offer guests the option of obtaining a temporary certificate, but we might want to include a clear warning before even establishing an HTTPS session with client authentication.

- Delegators need an easier way to look up guests than by four-digit subject ID. My initial implementation of the delegation Web app allowed a delegator to search the distinguished-name fields of recently-introduced X.509 certificates, but guests may not always know their own X.500 distinguished names. One approach might be to let the guest supply a name or nickname to our Web app via an HTML form, which could display the common name from his X.509 certificate as an initial suggestion. Whatever initial identification scheme is used, however, should not produce a false sense of security by implying in any way that the identifier is authoritative.

- It is redundant for a delegator to choose which of the waiting guests matching her query she wants to delegate to *and* choose that guest's visual fingerprint from among several. The delegation applet could instead merge her query results into the pool of random fingerprints it shows her. We would need to consider how to handle the unlikely scenario where her query returns a large number of matches.

- Delegators currently must export PKCS#12 files from their OS or browser keystores in order for the delegation applet to access their private keys. This step adds extra tedium to the delegation process, as it requires delegators who may not be experienced in PKI tools to navigate some of the more obscure configuration dialogs offered by their OS or browser platforms. It also will not work for delegators who keep their private keys on smart cards or USB tokens. The Java *KeystoreWrapper* class included in the delegation applet, however, could wrap native OS or browser keystores using JNI (Java Native Interface) and a native stub library for each platform. Implementing this functionality could clear up a major source of complexity.

- Java runtime environments maintain their own keystores that are not necessarily integrated with OS or browser keystores, and mark various principals as trusted code

signers using Java-specific tools. These quirks call for a thorough study of the behavior of various Java versions; such a study could lead to clear and consistent instructions for installing Java trust anchors. This improvement would smooth the Greenpass experience for delegators, particularly non-local delegators.

- Users still need to configure their EAP-TLS clients outside our client tools. We could create a Web page with detailed instructions on how to do this for each platform, or train some delegators to assist guests with the configuration process.

- Many guests might want to obtain access via telephone rather than through the face-to-face interaction required by visual hashing. PGPfone's [85] word lists, mentioned in Section 4.4, provide a way to compare hash values over a voice channel. Other hash-comparison methods are also worth investigating.

- Finally, the Greenpass client tools could be extended to include more flexible management options, such as the ability to store multiple SPKI certificate chains and the ability for delegators to keep track of to whom they have issued certificates.

5.3 Decentralization

The primary reason we decided to place authorization certificates in HTTP cookies was to provide truly decentralized authorization. Using this approach, a client can present a chain of authorization certificates to an entity who has never seen the client before and who can't communicate directly with the certificate's issuer. This "push" model of authorization takes full advantage of certificates, especially delegated certificates. At present, however, our experimental Greenpass setup uses only a single access point, RADIUS server, and authorization cache.

Further decentralization might be accomplished in many ways, but two representative examples follow:

- Have multiple access points depend on a single RADIUS server and authorization cache. This approach seems to work today with fairly large client populations using standard 802.1x authentication, but cannot scale forever. This approach suggests a single Web application server as well.

- Place several groups of Greenpass tools throughout a network, where each group consists of a single RADIUS server, Web application server, and authorization cache, but several access points. Each group would have its own restricted VLAN. Every group would, however, trust the same source-of-authority for SPKI certificates: guests could gain access within one group, then "push" the resulting credentials to gain access to other groups. Note that a guest and his delegator would need to connect to the same group to carry out the delegation process.

The last scenario might benefit from, e.g., a Linux-based software package that provides a turnkey Web application server, RADIUS server, authorization cache, and perhaps even gateway software to maintain the restricted local VLANs. In general, Greenpass would benefit from a stripped-down RADIUS server that supports only the very basic EAP-TLS handshake and authorization check we require.[2]

Distributed scenarios such as those described above would require careful tuning of DNS or routing entries on each local restricted VLAN. Web browsers present cookies only to the same domain name or domain name suffix that originally set them (the cookie issuer must specify if its cookies should be sent to servers with the same suffix; otherwise they are

[2]Rather than an entire authentication database, the Greenpass RADIUS server only *needs* to store the public key of the local CA and the public key of the local source-of-authority.

sent only to the one original server). The obvious solution is to ensure that all Greenpass Web application servers on a network share the same DNS suffix.

Roaming

Because users can carry their wireless devices with them while moving about an organization's premises, their devices must constantly go through a cycle of disassociating from one AP and associating to another. This process is called *roaming*. Roaming poses a particular problem when the device must also carry out authentication and/or authorization steps. It is essential to consider how Greenpass will impact roaming clients.

Greenpass might impact roaming clients differently depending on which of the two decentralized scenarios listed above is employed. Consider the scenario with a single RADIUS server and set of Greenpass client tools first. Edney and Arbaugh [25] discuss 802.1x *preauthentication*, which allows a device to detect an access point that is coming into range and authenticate to it over the *wired* network without disassociating from the current AP. A preauthenticated device will already have a set of keys in place with which to communicate with the new AP before it drops its connection to the old AP. APs that support preauthentication and rely on a RADIUS server should, in theory, be blind to any extra authorization checks (such as ours) that the RADIUS server performs. The RADIUS server, however, might be bombarded with requests from roaming clients who preauthenticate to a new AP every several seconds. It is also not clear which APs and client devices, if any, support preauthentication at present.

The second scenario—in which a network contains several groups of Greenpass tools—poses a different problem. It is not feasible for a client to carry out an HTTP-based reauthorization step (as described in Section 3.4.4) when roaming between groups. The best solution might be to implement a behind-the-scenes "hand-off" of authorization material

between adjacent groups. Implementing a hand-off between APs themselves might alleviate the load on the single RADIUS server in the first scenario, as well.

As Wi-Fi protocols improve, the IEEE and Wi-Fi Alliance will likely develop their own method of AP-to-AP hand-offs of authentication material. SAML (see Section 4.3.5) might provide an appropriate mechanism for analogous hand-offs between groups of Greenpass client tools, if needed.

5.4 Advanced SPKI/SDSI features

At present, Greenpass uses direct key-to-key SPKI delegation using only a single tag, *(greenpass-pilot-auth)*. SPKI/SDSI provides a number of advanced features that could allow organizations and their members to express more subtle authorizations than "access/no access." These features might, however, require modifications to the SPKI libraries used, and would also require careful consideration of how to retrieve all the certificates necessary to check a particular user's permissions.

5.4.1 Advanced SPKI tags

SPKI tags are simply arbitrary S-expressions to be interpreted by the relying application. As described in the SPKI theory and structure documents [32, 31], tags can include a number of positions, each of which restricts the access granted by the certificate. As a simple example, a tag *(wlan-access)* that grants wireless access might be restricted to only offer access to an SSID called "ssid42" by changing it to *(wlan-access ssid42)*. (Access points and the RADIUS server would be responsible for enforcing such restrictions.) Any position in a SPKI tag can be set to all possible values of that position (a wildcard), a range of values, a specific set or list of values, or the set of all values with a certain prefix. A

SPKI library with full tag intersection capability would allow delegators to propagate all or just a subset of their own privileges to other users.

Two straightforward and clearly useful applications of tags are as follows:

- Users could be given various "classes" of access by providing not just two, but several VLANs and specifying which one(s) a guest should have access to using the SPKI tag in his certificate. Guests might be given access only to an Internet gateway, for example,[3] or given access to resources of one department or another, or to a VLAN that includes digital library materials.

- Users could be given SPKI-based credentials to resources other than the wireless network, as described in Section 5.5 below.

5.4.2 SDSI names

SDSI provides a syntax for *local names*—i.e., names that are meaningful to a particular SPKI/SDSI principal. A brief overview of SDSI names follows:

- Each principal has its own namespace where names meaningful to it are bound to other principals.

- The holder of any key $\langle key \rangle_1$ can issue a SDSI name certificate that binds some name—e.g., *Alice*—to some principal $\langle key \rangle_2$, but only in $\langle key \rangle_1$'s namespace. Such a certificate means "the holder of $\langle key \rangle_1$ calls the holder of $\langle key \rangle_2$ *Alice*."

- SDSI names can be chained. If the person I call "Alice" in my own namespace just introduced me to somebody she calls "Bob," I can immediately refer to that person

[3] One of Greenpass's stated goals is to give authorized guests access to *internal* resources, but advanced tags would allow us to specify *which*, if any, internal resources.

using the SDSI S-expression *(name Alice Bob)*, which means "Alice's Bob." (My computer will need Alice's name certificate for Bob in order to make sense of this name by binding it to a public key.)

- SDSI names can be chained indefinitely: I could, for example, refer to "a friend of a friend of a neighbor's drycleaner" as *(name Alice Bob Carla Dan)*.[4]

- SDSI names can be "globalized" by prefixing a chain of names with a public key value: i.e., anybody can use *(name* $\langle key \rangle_1$ *Alice)* to refer to the person that $\langle key \rangle_1$'s owner calls "Alice."

- Finally, SDSI does not require a one-to-one mapping of names to keyholders. My neighbor, whom I call Alice, could bind the name *drycleaners* in her namespace to the public keys of several drycleaners she knows. I could then, e.g., issue a SPKI certificate with subject *(name Alice drycleaners*) to grant some privilege to all of Alice's drycleaners.

In their most obvious use, SDSI names could make for a more convenient delegation tool. If we wish to provide a more advanced "delegation management tool" that lets delegators keep track of whom they have issued what permissions to, as mentioned in Section 5.2 above, it would make sense to let them specify their guests by name. Specifically, delegation would involve an extra (but very easy) naming step, as follows:

1. the delegator chooses a name for her guest that is meaningful to her;

2. the delegator issues a SDSI certificate to her guest that binds the name she has chosen to his public key (but only in *her* namespace, not some CA's); and finally,

[4]This example assumes, of course, that I call my neighbor Alice, she calls her drycleaner Bob, he calls his friend Carla, and she calls her friend Dan.

3. the delegator issues a SPKI authorization certificate to her guest, using the SDSI name she has assigned him in the *(subject)* field instead of his public key,

If a delegator looks at an authorization certificate she issued months ago, she will probably remember who she issued it to if the subject is a SDSI name that is specifically meaningful to her.

An even more interesting use of SDSI names is as follows. Since SDSI names may be used for groups rather than individuals, and SDSI names are relative, they can naturally express a concept that might best be called "relative roles." This use could allow a convenient form of role-based access control to a Wi-Fi network, while still offering delegated guest access. Whereas standard role-based access control systems use simple role names such as "guests," SDSI can express relative roles such as "Alice's guests," "Bob's guests," or even "any professor's guests," and a source-of-authority can then grant privileges to these roles, as in the following example:

1. A university's SPKI source-of-authority, whose public key we will call $\langle soa \rangle$, issues SDSI name certificates to several trusted delegators to a group in its namespace called *delegators*. Alice, Bob, Carla, and Dan, for example, would all receive the SDSI name *(name* $\langle soa \rangle$ *delegators)*. At present, however, nobody in this group holds any privilege to do anything.

2. Now, the university's SOA issues a SPKI authorization certificate that grants Wi-Fi access to its own *(name delegators guests)*. The SOA states, in effect, "if somebody I call a *delegator* chooses to call you a *guest*, then access points or their RADIUS servers should grant you Wi-Fi access."

3. A visitor shows up on campus and asks Alice for wireless access; she grants his request by adding him to her local *guests* group. (She could still assign a meaningful,

individual name to him, then add that name to her group of guests.)

The approach just described does not offer any limit on delegation depth[5] and would become awkward if used to delegate access to too many types of resource. It might be worth investigating, however, a combination of SDSI relative roles with PERMIS's role-based access control policies (which might be able to enforce delegation depth limits).

A problem with all these naming schemes is that guests are no longer granted authorization via a single chain of certificates: they must present an appropriate combination of name and authorization certificates. Research would need be done into how to provide guests with the needed certificates or provide a reliable lookup protocol for the relying parties.

5.4.3 Threshold subjects

A SPKI *threshold subject* specifies that each of n subjects should receive $1/k$th of a particular authorization. At least k of the n designated subjects must simultaneously try to perform an action to be granted access, or at least k of the n designated subjects must delegate to the same principal for that principal's certificate chain to prove valid. Threshold subjects can increase the security of delegation: for example, by specifying SDSI group or role names as the n members of a threshold subject, a university might require that three delegators from different roles (e.g., student, professor, department chair) must all delegate Wi-Fi access to a guest. Threshold subjects, however, complicate the process of issuing, obtaining, and processing certificate chains.

[5] Nothing stops Alice from issuing a certificate that adds *(name guests guests)* to her group *(name guests)*.

5.5 SPKI-based access control to other resources

If our client tools were extended to provide sufficient generality, we could provide SPKI-based access control to resources other than the network itself. In some cases, of course, this is infeasible because the resources being accessed use their own handshake methods that do not provide the authentication and authorization required for SPKI. In general, however, any system that uses public-key-based authentication could be modified to check a SPKI authorization cache for credentials regarding that public key. For example:

- Many, many resources today are accessed via the Web. Web servers already support public-key based authentication via SSL/TLS, and Greenpass provides a way to present authorization credentials to a Web server as well (HTTP cookies). Someone could, therefore, design a Web server that provides SPKI-based access control along the lines of MIT's Geronimo project [19, 60] (see Section 4.2.1), but without the special client plugin. A client who tried to access a SPKI-protected resource on such a Web server would be redirected to a page where a delegator could grant him access.

- SSH or SFTP could possibly be modified to control access to different directories using SPKI.[6]

- Appendix A describes a short experiment I completed that adapts Greenpass to VPN access control.

- Finally, if IPSEC [50] were able to accept SPKI authorization certificates, we could control access to basically any IP-accessible resource that supports IPSEC.

SPKI-based access control to Web resources would have the most far-reaching implications.

[6] Niels Möller appears to be integrating a SPKI/SDSI library into his *lsh* secure shell implementation [62], but it is not clear how it will be used.

5.6 Authorization language comparison

The various systems described in Section 4.3 might all—some of them with extensions—provide alternative ways to support decentralized, delegated authorization. A "bake-off" study of these systems would prove insightful, and would likely take one of the following two forms:

- several people could look for as many real-world authorization scenarios as possible and figure out how easily they can be expressed in each of the competing languages, or
- a few people could implement a Greenpass system that supports several types of authorization, then see how each system works out as people try to use it in the real world.[7]

An approach that might eliminate the need for out-of-band authorization would use X.509 certificates as both authentication *and* authorization credentials. Delegators would issue guests new X.509 certificates—probably with an anonymous name, or one derived directly from the subject's public key—containing extensions that define the privileges being granted. The extensions themselves might contain policy statements in SPKI or another language. This approach has a few drawbacks:

- The X.509 certificate format, especially its ASN.1 encoding, is more complex than SPKI/SDSI.

[7] The difficulties with this latter approach probably would be (1) linking authorization materials to authentication materials, since each authorization system implicitly assumes certain types of identification, and (2) creating editors for each of the competing languages. The latter step is probably too difficult to attempt until a single target authorization language is settled upon.

- The relying party would need to process these X.509 certificates differently than standard X.509 certificates. In particular, it would have to mostly ignore name fields and make sure the certificates form an unbroken chain from public key to public key. It would also need to interpret the authorization extensions properly.

- Keystores on guest machines might refuse to import and/or store highly non-standard certificates.

- Users would obtain new X.509 certificates with each new privilege granted, and would have to manage all these extra certificates. Cookies automatically get presented only to the entity that issued them.

Proxy certificates [104, 109] (see Section 4.3.2) provide much of the needed functionality. The naming semantics of proxy certificates, are entirely the opposite of SDSI's: whereas SDSI defines names relative to public keys, proxy certificates define public keys relative to names. Proxy certificates are better suited for identifying a temporary key pair acting on behalf of a user than for identifying people that a user has authorized to do something. This consideration would cause problems with the accounting system I suggest in Section 5.7 below.

5.7 Other thoughts

Finally, this section offers some Greenpass-related ideas that do not fit elsewhere.

Accounting

It is important for an organization to log the activities of users on its network, a process called *accounting*. Greenpass poses new accounting challenges for two reasons: guests

will be using the network, and they may not have definitive names. SDSI might allow a distributed accounting method for Greenpass, which would work as follows:

- An organization sets up a "root" accounting entity.
- The root accounting entity issues SPKI certificates that grant some other entities the authorization to act on its behalf. These subordinate accounting entities could be spread throughout an organization's network to prevent bottlenecks.
- Access points, RADIUS servers, or whatever other entities enforce access policies require users to present both an authorization certificate chain from a delegator *and* a certificate from an accounting entity stating that it has seen and recorded that particular certificate chain. It could assert this by issuing a certificate with the SPKI *(object-hash)* of the guest's certificate chain as its subject.

The accounting entity that records a certificate chain could, perhaps, make a note of the SDSI name attached to its subject. The organization would keep on record the public key values of all its local members, so it could trace any delegation chain back to a local user. The organization could hold its local members accountable not so much for the actions of their guests, as for assigning meaningful SDSI names to their guests. Officials who were investigating an incident involving a guest could show his delegator the certificate that granted him access, and because it contained a meaningful SDSI name, the delegator could identify the actual individual responsible.[8]

Virus protection

In a similar manner, guests (or even all users) could be required to obtain credentials from an automated entity that assures their machines do not contain viruses.

[8] I am not a lawyer, however.

A SPKI tag editor

If SPKI controls access to numerous resources, users will need a convenient interface with which to manage their own privileges, as well as assign names to other principals and delegate to them. A usable SPKI "tag editor" might include the following features:

- A "drag-and-drop" authorization editor, which would allow a user to drag his currently-held privileges to other users to whom he has assigned SDSI names.
- A description language for SPKI tags: this could tell a user what the various tags in an application-specific SPKI tag actually mean so he could restrict delegated privileges appropriately. Perhaps the tag editor could even read a tag description and automatically produce a GUI dialog box for editing that particular tag.

Shared-secret authentication

Most computer users understand shared-secret authentication systems, such as passwords, more intuitively than public-key-based systems. Support for password-based EAP methods such as Cisco's LEAP [18] seems more widely supported at present, especially by older systems, than EAP-TLS. Password-based systems are often easier to configure on the client side than PKI-based systems. A Greenpass variation based on password authentication might work as follows:

- A guest connects, without authentication, to a restricted VLAN and connects to a Greenpass Web app.
- The Web app sets up an HTTPS-protected session with the guest and assigns him some random session token (this is a standard method of session protection in many online applications). It then associates some other random number with the guest's

session internally and displays a visual hash of this second token[9] on the guest's screen.

- A delegator connects to the Greenpass Web server and vouches for the guest with that particular visual hash. The Web application lets her do this only after checking the credentials that give her permission to delegate.

- The guest's Web app generates and assigns him a temporary username and password[10] that he types (or copies and pastes) into his Wi-Fi configuration software.

- The access point or RADIUS server would then, somehow, learn that it should grant access to the user with that particular username/password pair.

It is difficult to see how such a system would ever allow PKI-like delegation chains,[11] but it could provide an alternative, and perhaps easier, way for delegators to grant Wi-Fi access to guests who need no further features. It is also not immediately clear how such a system would provide decentralized authorization: the guest might have to carry an HTTP cookie that proves his authorization to use that particular credential, and it is not clear how to distribute such a credential securely among an arbitrarily large population of relying parties. On the other hand, if a roaming system were already in place in which adjacent access points "hand off" information about a client as he moves between them, as discussed in Section 5.3 above, then they could hand off the guest's credentials at the same time.

[9] Using the first number both to authenticate the guest's session and as the source for a visual hash might reveal too much information about it, compromising the guest's session.

[10] Wireless clients typically remember passwords, so there is no reason to use a potentially weak human-generated password.

[11] Crispo, Popescu, and Tanenbaum [23] discuss ways to provide *some* PKI-like functionality using symmetric keys.

Non-compatible devices

Some devices will not be compatible with Greenpass. Particular problem areas include

- devices without WPA's 802.1x support, and
- devices without Web browsers.

A captive portal could be used to control access for such devices—devices without Web browsers might have to "authenticate" using a neighboring machine—but it would negate the security of 802.1x access control if it provided the same level of privilege. An organization could provide one or more limited access levels—implemented using VLANs, as discussed in Section 5.4.1 above—for specialized devices. In particular, it might be possible to support certain devices such as VoIP phones by providing them only the resources needed for their particular usage models. An organization could also provide captive portal access to the Internet, possibly with limited bandwidth and port access, to guests with older, non-WPA operating systems.

Chapter 6

Summary

This thesis has described a set of Web-based client tools that allow users to obtain guest access to a Wi-Fi network from a previously-authorized user of that network, rather than from a central network administrator. These tools form part of Dartmouth's *Greenpass* project. Kim [55] describes the other major component of Greenpass: a RADIUS server that authenticates users with an industry-standard EAP-TLS handshake, but adds an additional authorization step that lets guests with valid SPKI credentials connect even if they do not have trusted X.509 certificates. Because we use EAP-TLS for wireless authentication and standard SSL/TLS for Web-based key introduction, our system does not require any custom software on the guest's machine.

Chapter 1 described our motivation for Greenpass. The physical layer of a Wi-Fi network is accessible to the extreme.[1] Organizations that deploy Wi-Fi networks have to consider two seemingly conflicting demands that this accessibility creates:

- On the one hand, an open physical layer requires strong, cryptography-based privacy

[1] My advisor, using a special antenna, is able to connect to our campus Wi-Fi network from his home that approximately 1.5 miles away, albeit uphill.

and access-control measures at the link layer or higher. Otherwise, organizations might expose sensitive information or find that their networks are being abused.

- On the other hand, Wi-Fi's open physical layer allows an organization to offer cheap—often free—Internet connectivity to visitors. People are beginning to expect this convenience wherever they go, and it seems a shame to deny such universal connectivity to honest, invited visitors because of security concerns about uninvited outsiders.

Current Wi-Fi security standards provide only authentication and access control, however. They typically offer centralized authorization mechanisms such as ACLs that are inconvenient to modify every time a new guest wants access. Public-key cryptography can decentralize authorization to reflect real organizational structures and person-to-person relationships such as the host/guest relationship that typically arises when a person is invited to a large university or enterprise.

Chapter 2 described background material related to my work on the Greenpass client tools. Newer Wi-Fi security standards allow Wi-Fi users to authenticate via *EAP-TLS*, which, like the TLS handshake offered in Web browsers, makes use of X.509 public key certificates. An X.509 PKI by itself, however, does not support guest access easily because it relies on global names: most organizations operate their own, independent certification authorities (CAs), leaving us with no means to authenticate guests from arbitrary home organizations. Instead, Chapter 2 suggests the use of *SPKI/SDSI*, a very lightweight PKI system that can issue authorizations directly to the owner of a particular public key without relying on an intermediate naming step. SPKI/SDSI also allows *delegation*: a local Wi-Fi user can issue a signed SPKI certificate to a guest that states, in essence, "I said it's okay for this keyholder to obtain Wi-Fi access." Kim's Greenpass RADIUS server [55], if it does not recognize the issuer of an X.509 certificate, will extract the public key value from that

certificate and then look in an ***authorization cache*** for a SPKI certificate chain regarding the owner of that key.

Chapter 3 described the client tools I have contributed to the Greenpass project to enable delegation from a host to a guest. Some of the most salient questions answered in that chapter are as follows:

- *How can someone delegate to a user who does not hold a trusted name certificate?* SPKI/SDSI solves part of the problem: it offers certificates that can make statements about the owner of a public key instead of an entity with a certain name. We solve the other part of the problem using *visual hashes* as suggested by Perrig and Song [83] and Ellison and Dorhmann [28]. A delegator can painlessly compare the visual hash of the key she is about to delegate to with the visual hash of her indented guest's key, as displayed on his device's screen.

- *How does the delegation tool access the delegator's private key?* After considering a stand-alone application, we settled on signed ***delegation applet***. A signed Java applet can, if the user marks its signer as trusted, access the user's local filesystem. We use this capability to load the delegator's private key from a password-protected PKCS#12 file, although future versions could support native keystores. Visual fingerprint verification is built into the applet, so its user will always verify who she is about to delegate to before the applet signs a certificate. Other hash-comparison methods would suffice as well.

- *How do the delegator and guest communicate without custom software?* A set of ***Greenpass Web applications*** use the ubiquitous Web browser to provide a cross-platform front end for our client tools and a communication path between guest and delegator. The guest introduction Web app learns the guest's public key via SSL/TLS

client authentication, and can generate new key pairs for guests who need them using standard browser-based PKI enrollment functionality.

- *How does a guest "push" his authorization credentials to the Greenpass RADIUS server if EAP-TLS only supports authentication?* He doesn't. The Greenpass Web apps add his SPKI certificate chain to an HTTP cookie, which his Web browser can later "push" to the Greenpass Web apps (or a clone of them on a different subnet) to prove his authorization. The guest pushes his authorization credentials via a channel that is parallel to, rather than encapsulated in, the EAP-TLS handshake.

Chapter 4 described work related to Greenpass. To our knowledge, ours is the first working solution to Wi-Fi guest access that uses SPKI/SDSI delegation for authorization, though other authors have suggested the idea. A number of researchers have implemented SPKI/SDSI-based or similar access control to resources other than a Wi-Fi network. Also, there are a number of policy languages (such as XACML) and certificate or assertion formats (including SAML, KeyNote, XrML, X.509 attribute certificates, and X.509 proxy certificates) that compete with SPKI/SDSI.

Chapter 5 suggested a number of future research and development ideas related to Greenpass. The client tools still contain a number of problems, but I point them out and suggest possible solutions for them. I also point out that roaming is an essential area of research for any Wi-Fi authentication or authorization system. Delegated authorization is still a largely unexplored field, especially in terms of implemented applications, and offers many opportunities for exciting research. The most fertile grounds for research are probably implementing delegated access control to a variety of resources—particularly Web-based resources—and comparing competing authorization languages and approaches to find one that allows straightforward expression of the policies that people actually need. I expect that the most successful approach might involve static policies at access points or

other access controllers combined with very *simple*, delegated, dynamic assertions made by human end users and automated attribute authorities. PERMIS's combination of hierarchical, delegated roles with a policy language already tends in this direction, but appears to be encumbered by X.509-based naming. Interesting future research might investigate other combinations: e.g., SPKI assertions or SDSI relative name/group certificates combined with more static XACML policies.

Appendix A

Greenpass and VPNs

Section 5.5 suggested a number of additional resources, besides a Wi-Fi network, that might benefit from delegated, SPKI/SDSI-based access control. One class of resources we believe would benefit are *virtual private networks* (*VPNs*), specifically remote-access VPNs. This chapter briefly describes an experiment we have set up that uses our existing Greenpass pilot setup to provide delegated guest access to a network through a *VPN concentrator*.

A.1 Definitions

The VPN Consortium [107] distributes a whitepaper [106] that describes various VPN technologies. By their definition, a VPN is "a private data network that makes use of the public telecommunication infrastructure, maintaining privacy through the use of a tunneling protocol and security procedures." The VPN Consortium considers the following two types of VPN to be entirely distinct:

- *Trusted VPNs* are provided to a company by its communications provider. The company trusts the provider to maintain the privacy of the circuits over which its infor-

mation travels.

- *Secure VPNs* use a cryptographic tunnel to maintain the privacy and integrity of data as it travels from one section of a company's network, across an untrusted network (usually the Internet), and into another section of the company's network.

The rest of this chapter applies only to secure VPNs.

VPNs could protect a network's communications in more than one way:

- *Gateway-to-gateway encryption*. Two or more gateways might communicate via a VPN tunnel, unifying two or more small, private LANs into a single network via the Internet without compromising their original privacy.

- *Remote-access VPNs* provide a private, authenticated tunnel from a single client device into a network that otherwise would not be accessible from the Internet. The client device usually establishes a tunnel to a gateway device on the edge of the target network. This secure gateway device is sometimes called a *VPN concentrator*.

Several protocols can interact in forming a VPN. Some of those more commonly encountered are as follows:

- *L2TP* (*Layer 2 Tunneling Protocol*) [103] can tunnel PPP (Point-to-Point Protocol, as traditionally used for modem connections to the Internet) over an IP network. L2TP inherits PPP authentication mechanisms such as EAP.

- *PPTP* (*Point-to-Point Tunneling Protocol*) is a proprietary Microsoft protocol similar to L2TP.

- *IPSEC* (*IP Security*) [50] is a protocol—or rather, a set of protocols—that allows two IP endpoints to authenticate one another, negotiate a shared session key, then

integrity-protect and/or encrypt their ensuing communications using that session key. IPSEC is rather similar in concept to SSL/TLS (see Section 2.2), but IPSEC protects IP packets themselves, whereas SSL/TLS adds a security layer on top of a TCP/IP socket-based connection.

Many remote-access VPN clients offer L2TP-over-IPSEC [82] as a VPN protocol option: in this case, an L2TP tunnel is established over an IPSEC tunnel. L2TP provides mutual authentication of the endpoints at the beginning of the session while IPSEC protects the privacy and integrity of ensuing L2TP traffic.

A number of vendors offer *SSL VPNs*; the debate is still open as to whether these are true VPNs. SSL VPNs allows users to log in to their home organizations' networks from remote machines using a only a Web browser as their "VPN client," authenticating and tunneling their communication over SSL (or TLS). Given the strength and high interoperability of the SSLv3 and TLS protocols, this approach may well be superior to traditional VPNs for providing easy remote access to Web applications along with strong authentication and privacy mechanisms. SSL VPNs, however, do not serve non-HTTP applications, such as email, telnet, and secure shell, particularly well. Vendors have to invent ad-hoc approaches such as Java-based email and shell programs, or ActiveX- or Java-based proxy servers and port forwarders that run on the client machine and redirect traffic for the duration of the SSL VPN session.

A.2 Motivation

A remote-access VPN allows users to work remotely while still shielding their employer's internal network resources from outside attack. Dartmouth currently uses VPNs to control access to certain highly sensitive resources; delegation (not necessarily guest access) might

provide useful semantics for accessing such resources. In addition, VPNs could provide an alternative method of Wi-Fi access control. Rather than authenticating using 802.1x, a user could connect to an access point that provides access only to a restricted VLAN, then obtain full access to the local network through a VPN concentrator. In either of these cases, we need to enhance the VPN's access control with Greenpass's SPKI-based authorization in order to allow delegated access.

We also wish to demonstrate that we could augment *any* client-server system that uses public-key-based authentication to include a SPKI-based authorization check, without modifying the existing client handshake. All that is needed is to modify the server to extract the public key of any unknown user and use it to check an authorization cache for SPKI certificates regarding that key, just as the Greenpass RADIUS server does.

A.3 Experiment

We added a Cisco VPN 3000-series Concentrator [17] to the test network described in Kim's thesis [55]. We configured it as follows:

- We connected the concentrator's "public" interface—the one that would normally expose the concentrator to the Internet so that remote clients can connect to it—to the restricted VLAN that unauthenticated and unauthorized users connect to, and gave it an IP address on that VLAN.

- We connected the concentrator's "private" interface—the one that the VPN concentrator typically uses to give legitimate users access to the private network it protects—to a subnet on the full, unrestricted Dartmouth network.

- We tested this configuration with simple password-based authentication to confirm that it worked.

- We then configured the VPN concentrator to provide PPTP and L2TP/IPSEC access, both using the authentication type *EAP Proxy*. We configured the VPN concentrator to forward EAP requests to the Greenpass RADIUS server.

Kim's test setup already included DNS and DHCP servers running on the restricted VLAN, so we did not need to configure these. We did have to configure the VPN concentrator to consult the correct Dartmouth DNS server, DHCP server, and gateway on its "private" interface in order to give authenticated and authorized users access to the full Dartmouth network.

A.4 Results

We tested the VPN concentrator configuration using the Microsoft VPN client on a Windows XP laptop computer. We connected to laptop to the "Greenpass Test" SSID on our test network's access point; this SSID does not require authentication but provides access only to the restricted VLAN. We confirmed the following:

- The client authenticated successfully to the VPN concentrator using EAP-TLS with an X.509 certificate issued by the local Greenpass CA.

- The client did *not* authenticate successfully using a temporary certificate created by the Greenpass dummy CA.

- After going through the Greenpass delegation process using the dummy certificate and obtaining a SPKI/SDSI chain, however, the client *was* able to authenticate and connect to the VPN concentrator.

We confirmed that this worked using both PPTP and L2TP, although the latter provides greater security for an established tunnel.

One problem that we ran into was as follows. Windows XP obtains information about DNS servers using DHCP when it first connects to our AP on the "Greenpass Test" SSID. The restricted VLAN includes a DNS server that we use to redirect HTTP requests to the Greenpass front page. After the VPN client establishes a connection to the VPN Concentrator, it obtains a second DHCP lease on the unrestricted network at the remote endpoint of its VPN tunnel. The DNS information it obtains from this second connection, however, does not appear to override the original DNS information, so HTTP requests still get redirected to the Greenpass front page. At present, we can only work around this by manually entering the IP addresses of the correct DNS servers.

A.5 Future work

This experiment was only a preliminary study into using VPN concentrators with Greenpass. It was, however, surprisingly straightforward to set up preliminary guest-access capabilities simply by pointing the VPN concentrator at the existing Greenpass RADIUS server.

Some of the questions and problems that might be considered in the future are as follows:

- How should the enormously complex range of configuration options provided by the VPN concentrator be set up to provide a smooth user experience without the DNS quirk we described, and without other quirks our limited testing did not reveal?
- How would our setup scale to larger networks or different network configurations? In our experiment, we were constrained by the existing configuration of the Greenpass test network, and other configurations might work better.

- Do VPN clients allow a user to connect to one VPN concentrator for the local network he is visiting, and then establish a second VPN tunnel to his own home organization's network? This consideration is essential when providing guest access.

- Currently, the Microsoft VPN client is the only one that provides EAP-TLS authentication. How can we offer access, particularly with certificate-based authentication, to clients on other platforms?

- Finally, how can we provide guest access to a pure IPSEC VPN without relying on L2TP authentication?

This last question is particularly worth considering. The L2TP-over-IPSEC RFC [82] discusses a number of security issues that can arise when authentication is divorced from establishment of lower-level IPSEC tunnel encryption. IPSEC's *IKE* (*Internet Key Exchange*) protocol [44] supports mutual authentication based on X.509 certificates, much as EAP-TLS does. The Cisco VPN concentrator we used supports pure IPSEC authentication using IKE; unfortunately, the only fields of the target's X.509 certificate it will forward to a RADIUS server during this exchange are X.500 distinguished name fields. To carry out a decision based on the target's public key value, we would need to modify the VPN concentrator software itself. Future work might involve modifying an open-source IPSEC VPN system, such as FreeS/WAN [38] running on Linux, to carry out a SPKI-based authorization check during an initial IKE certificate exchange.

Glossary

802.11 The IEEE wireless Ethernet standard that people typically mean when they refer to a "wireless network" or "wireless LAN" (WLAN). Informally called *Wi-Fi*.

802.11a/b/g Addendums to the IEEE 802.11 standard that define different physical layers (with different broadcast spectrums and transmission speeds) for use with wireless Ethernet.

802.11i A draft IEEE addendum to the IEEE 802.11 standard that defines new security features.

802.1x An IEEE access-control mechanism for Ethernet networks that involves three entities—the *supplicant*, the *authenticator*, and the *authentication server*—communicating via the *EAP* authentication protocol. See Section 2.1.4 for a detailed description.

ACL (access-control list) A simple authorization mechanism that lists what entities are allowed to access a particular resource.

ad-hoc mode An 802.11 mode in which wireless client stations communicate directly with one another using radio waves. See also *infrastructure mode*.

associate To establish a connection with a wireless access point. Sometimes—e.g., if the

AP in question uses 802.1x access control—a client station may need to carry out an additional authentication handshake after association before the AP grants it full network access.

attribute certificate A digitally-signed statement that binds an attribute—which could be a privilege, age, role, or nearly anything else—to a name or a public key. Typically refers specifically to an X.509 attribute certificate as described in RFC 3281 [34].

authentication The act of establishing an identity for some party, usually by finding out that party's name.

authentication server In 802.1x, the entity that ultimately processes the supplicant's authentication handshake and tells the authenticator whether to grant the supplicant access.

authenticator In 802.1x, the entity that guards the resource the supplicant is trying to access. The authenticator relays EAP messages between the supplicant and the authentication server.

authorization In the words of Kaufman et al. [54], "permission to access a resource."

authorization cache The Greenpass component that is responsible for receiving SPKI credentials (certificate chains), verifying them, and storing them so it can answer queries from the Greenpass RADIUS server about what principals are authorized.

authorization certificate A digitally-signed statement that binds an authorization (i.e., privilege) either to a name or directly to a public key, giving the owner of that name or public key the right to exercise the privilege.

captive portal A system of authentication and access control to a Wi-Fi network that consists of two entities: (1) a gateway that filters clients based on their MAC addresses

and gives them different levels of access based on their different privilege levels, and (2) a Web-based authentication service that authenticates users and informs the gateway which users/devices should be granted what privilege levels. See Section 4.1.1 or the NoCatAuth project [70] for more information.

certification authority (CA) In X.509 and some other name-based PKIs, a specialized entity that is responsible for verifying *name→public key* bindings and issuing name certificates that tell the rest of the world about those bindings.

certificate Used alone, often refers to a *name certificate*, but could also refer to an *authorization certificate* or an attribute certificate. In general, a certificate might be defined as a digitally-signed statement that binds two independent characteristics of an entity together so that a relying party who trusts the certificate's issuer can infer one characteristic about the entity after learning and verifying the other.

certificate chain A sequence of certificates that conveys trust or privilege from its first certificate's issuer to its last certificate's subject.

CRL (certificate revocation list) A list maintained by an entity that issues certificates listing those certificates it has revoked due to loss or compromise of their subjects' private keys or due to a change of status on the part of the subject. A certificate issuer typically either distributes CRLs periodically to all users of its certificates or makes its CRL available in a location that it specifies in all its certificates.

default key mode In WEP, a mode in which all client stations talking to an AP share the same secret key. See also *key-mapping key mode*.

delegate To grant a subset of one's privileges to another entity or to one's own pseudonym.

dummy CA The component of the Greenpass client tools that issues new X.509 certificates to guests who do not already have them.

EAP (Extensible Authentication Protocol) An authentication standard defined in RFC 2284 [13] and used to authenticate PPP dialup requests and 802.1x Ethernet access requests, among other things. EAP simply provides a general packet format that can be used to encapsulate other authentication handshakes. See Section 2.1.4 for details.

EAP-TLS A standard defined in RFC 2716 [1], used to encapsulate the TLS authentication and key-establishment handshake within EAP.

IETF The *Internet Engineering Task Force*; see http://www.ietf.org/.

infrastructure mode An 802.11 mode in which one wireless station, called an *access point* (*AP*), provides a central LAN service to one or more mobile client stations in its vicinity. Typically, the AP also provides a connection to a larger network backbone. Most organizations provide wireless LAN and Internet connectivity to their members and to guests using infrastructure mode, and it is this mode that Greenpass focuses on. See also *ad-hoc mode*.

introduction The process by which one entity learns another entity's public key in the absence of a trusted third party. See Sections 3.2.3 and 4.4 for more information and related work.

introduction cache The component of the Greenpass client tools that holds X.509 certificates of guests who have recently introduced themselves.

IPSEC (IP SECurity) An IETF standard that lets two IP endpoints authenticate to one another and establish a shared secret for encrypting and integrity-protecting IP packets

they send to one another. See the IPSEC working group's Web page [50] for further information.

key-mapping key mode In WEP, a mode in which each client station has its own, unique secret key that it shares with the access point. The access point has to keep a list of which key goes with which MAC address.

KeyNote A signed assertion (or authorization certificate) language. See Section 4.3.4 or the KeyNote RFC [8] for additional information.

keystore An OS- or browser-provided database that stores a user's trust anchor root certificates, end-entity certificates that the user trusts, and the user's own certificate(s) and corresponding private key(s). See Section 2.2.5 for details.

LDAP (Lightweight Directory Access Protocol) A protocol described in RFC 1777 [114] that provides straightforward read/write access to a hierarchical directory of X.500 names with attached attributes.

name certificate A digitally-signed statement that binds a name to a public key. A name certificates states, in essence, "the entity named *X* knows the private key corresponding to the public key herein."

PEM (Privacy Enhanced Mail) An early method of providing encryption and digital signatures for email or other text messages, described in RFCs 1421–1424. It has been superseded by S/MIME, but many utilities still use the PEM format (simple Base64 encoding with readable header and footer lines) to store X.509 certificates and other cryptographic material in files.

PERMIS A project that provides *role-based access control* (*RBAC*) based on a combination of X.509 attribute certificates and an XML-based policy language. See Sec-

tion 4.3.3 for more information and references.

policy language A computer language that is geared towards specifying complex sets of conditions under which an entity should or should not be granted access to a particular resource. XACML [74, 77] is one example.

PKC (public-key certificate) Another term for a *name certificate*, usually applied specifically to X.509 name certificates.

PKCS#12 A file-based keystore format [93].

PKI (public-key infrastructure) According to Kaufman et al. [54], "the components necessary to securely distribute public keys."

PKIX An IETF working group that develops standards and guidelines for the use of X.509 certificates on the Internet. See their home page [86] for more information.

principal According to Kaufman et al. [54], "a completely generic term used by the security community to include both people and computer systems. Coined because it is more dignified than *thingy* and because *object* and *entity* (which also mean thingy) were already overused." In SPKI/SDSI, a *principal* is the holder of a particular public key.

propagate Used synonymously with "delegate" in SPKI. In particular, the SPKI *(propagate)* tag, if present in a SPKI certificate along with a privilege, gives the subject of that certificate permission to delegate the privilege to others.

proxy certificate A modified X.509 public-key certificate (PKC) that allows the holder of a PKC (or another proxy certificate) to both (1) establish a temporary identity with its own key pair and (2) delegate all or a subset of his privileges to that temporary

identity. See Section 4.3.2, Welch et al. [109], or the X.509 proxy certificate draft RFC [104] for more information.

public-key fingerprint A hash of a public key value that can be written on paper or transmitted via some other trusted channel; used to ensure that a full public key value transmitted via an untrusted network has not been modified in transit.

"pull" authentication/authorization To obtain name or authorization certificates from a directory while authenticating a target entity or checking its access privileges.

"push" authentication/authorization To present authentication or authorization materials directly to a relying party while trying to prove one's identity to that party and/or prove that one is authorized to access a resource it guards.

RADIUS server An authentication server that, in the words of the RADIUS RFC [89], is "responsible for receiving user connection requests, authenticating the user, and then returning all configuration information necessary for the client to deliver service to the user." In the context of 802.1x, a RADIUS server communicates with a supplicant via an EAP tunnel set up by an authenticator.

relying party The entity that is trying to establish the identity of the other party (the *target*) in an authentication handshake, or obtain proof that the target is permitted to access a certain resource in an authorization handshake.

RFC (request for comments) The typical means of publication for IETF standards.

role-based access control (RBAC) An access control system where each user is assigned a *role* or roles, where each role implies a certain set of access privileges. Because there are typically far fewer roles than users, RBAC can reduce complexity by not requiring a separate statement of access privileges for each user.

SAML (Security Assertion Markup Language) An OASIS standard that allows one party to send a second party various XML-based assertions about yet another party. See Section 4.3.5 or SAML's OASIS Web site [73] for more information.

SDSI (Simple Distributed Security Infrastructure) A PKI and name certificate format, since merged with SPKI, whose distinguishing characteristic is its local name syntax. See the documents listed under *SPKI* for more information, or see Section 5.4.2 for a brief introduction. Pronounced "sudsy."

source-of-authority (SOA) A principal and public key that a relying party trusts *a priori* to grant authorizations to other principals. A relying party might trust a particular SOA to grant all types of privileges, or just some. Analogous to *trust anchor*; the latter is more often used to refer to a root naming authority.

SPKI (Simple Public Key Infrastructure) A PKI and authorization certificate format whose distinguishing characteristics are (1) direct binding of privileges to public keys and (2) delegation. See the SPKI theory RFC [32], the SPKI certificate structure draft [31], or Section 2.3 for details. Pronounced "spooky" or "speaky."

SSID A human-readable "network name" that helps client stations determine the AP or group of cooperating APs to which they are associating. An AP can either broadcast its SSID or keep it "secret," but "secret" SSIDs do not add any security.

SSL (Secure Sockets Layer) An Internet protocol created by Netscape that enhances TCP/IP socket-based connections with public-key-based client, server, or mutual authentication, session encryption, and session integrity. SSL has been superseded by the IETF's *TLS* protocol. See Section 2.2 for more details.

supplicant In 802.1x, the entity that attempts to gain access to some network or network

resource.

tag The field of a SPKI certificate that defines (in an application-specific manner) the privilege being granted.

target The party to an authentication or authorization handshake that is trying to prove its identity and/or its permission to access some resource to the party.

TKIP (Temporal Key Integrity Protocol) An enhanced encryption and integrity-protection protocol that replaces WEP in WPA, but remains compatible with WEP hardware. See Chapter 11 of Edney and Arbaugh [25] for an in-depth overview of TKIP.

TLS (Transport Layer Security) An IETF-standardized version of Netscape's SSL protocol. TLS enhances TCP/IP socket-based connections with public-key-based client, server, or mutual authentication, session encryption, and session integrity. See Section 2.2 for more details.

trust anchor An entity that is trusted *a priori* and whose public key is already known.

tuple reduction A method of intersecting SPKI, SDSI, and other certificates to form increasingly simpler structures with the same semantics as the original collection of certificates. Described in the SPKI theory RFC [32].

visual fingerprint A transformation of a public-key fingerprint into a unique image that can be easily recognized or compared with other visual fingerprints.

visual hash See *visual fingerprint*.

VPN (virtual private network) A network that makes use of public communication channels such as the Internet, but uses cryptographic tunneling to maintain the privacy of

its own traffic.

VPN concentrator A secure gateway device that lets clients log in and establish a cryptographic tunnel to a network that otherwise would not be accessible from the Internet.

WEP (Wired Equivalent Privacy) A highly broken security standard defined in the original IEEE 802.11 standard. It uses a shared secret between a client station and access point. See Borisov et al. [14] or Edney and Arbaugh [25] for a discussion of its many weaknesses.

Wi-Fi An informal name for the *IEEE 802.11* wireless Ethernet standard. More accurate than simply "wireless," which could also refer to non-802.11 standards such as Bluetooth.

Wi-Fi Alliance An industry consortium of vendors of IEEE 802.11-based products. The Wi-Fi Alliance also acts an informal standards body that defines extensions to the IEEE 802.11 standard, such as WPA.

WPA (Wi-Fi Protected Access) A subset of 802.11i, released as a transitional standard by the Wi-Fi Alliance, that offers enhanced security but remains compatible with WEP hardware. It uses TKIP for encryption and session integrity and allows the use of 802.1x for authentication and session key establishment.

X.500 distinguished name (DN) A particular name format, used to identify issuers and subjects of X.509 certificates, that includes not just an entity's common name but also other distinguishing information such as organizational membership, email address, location, etc.

X.509 name certificate The most common name certificate format on the Internet. The IETF's PKIX working group [86] defines a standard profile [46] for its use. See

Section 2.2 for details.

XACML (eXtensible Access Control Markup Language) A rather comprehensive XML-based policy language defined by OASIS. See Section 4.3.5 or the XACML technical committee Web site [74] for more information.

XrML (eXtensible Rights Markup Language) A policy (or authorization certificate) language with which a provider of digital resources can grant access privileges to its resources to other parties. See Section 4.3.5 or the XrML Web site [113] for further information.

Bibliography

[1] Bernard Aboba and Dan Simon. PPP EAP TLS Authentication Protocol. RFC 2716, October 1999.

[2] Albatross: a Toolkit for Stateful Web Applications. [http://www.object-craft.com.au/projects/albatross/].

[3] Christopher Allen and Tim Dierks. The TLS Protocol Version 1.0. RFC 2246, January 1999.

[4] Apple Computer Inc. Rendezvous. [http://developer.apple.com/macosx/rendezvous/index.html].

[5] Kwang-Hyun Baek, Sean Smith, and David Kotz. A Survey of WPA and 802.11i RSN Authentication Protocols. Technical report, Dartmouth College, 2004. In preparation.

[6] Dirk Balfanz, D. K. Smetters, Paul Stewart, and H. Chi Wong. Talking to Strangers: Authentication in Ad-Hoc Wireless Networks. In *Network and Distributed System Security Symposium*, 2002.

[7] Andrej Bauer. Gallery of Random Art. [http://gs2.sp.cs.cmu.edu/art/random/].

[8] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The KeyNote Trust-Management System Version 2. RFC 2704, September 1999.

[9] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *IEEE Symposium on Security and Privacy*, May 1996.

[10] Bluesocket Inc. Bluesocket: Frequently Asked Questions. [http://www.bluesocket.com/solutions/faq.html].

[11] Bluesocket Inc. Issues and Requirements for Public Access WLANs. [http://www.bluesocket.com/solutions/Issues-Requirements-for-Pub.Access-WLANs.pdf].

[12] Bluesocket Inc. WLAN Solutions—802.1x and Bluesocket. [http://www.bluesocket.com/solutions/802.1x-Feature-Brief.pdf].

[13] Larry J. Blunk and John R. Vollbrecht. PPP Extensible Authentication Protocol (EAP). RFC 2284, March 1998.

[14] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting Mobile Communications: The Insecurity of 802.11. In *International Conference on Mobile Computing and Networking (MOBICOM)*, 2001.

[15] Oscar Canovas and Antonio F. Gomez. A Distributed Credential Management System for SPKI-based Delegation Scenarios. In *1st Annual PKI Research Workshop*, April 2002.

[16] Common Data Security Architecture (CDSA) project. [http://sourceforge.net/projects/cdsa].

[17] Cisco Systems. VPN 300 Series Concentrator Reference Volume I: Configuration, Release 4.1, January 2004.

[18] Cisco LEAP protocol description. [http://lists.cistron.nl/pipermail/cistron-radius/2001-September/002042.html], September 2001.

[19] Dwaine E. Clarke. SPKI/SDSI HTTP Server / Certificate Chain Discovery in SPKI/SDSI. Master's thesis, Massachusetts Institute of Technology, September 2001.

[20] IEEE Computer Society. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications. IEEE Standard 802.11-1997, June 1997.

[21] IEEE Computer Society. IEEE Standard for Local and metropolitan area networks: Port-Based Network Access Control. IEEE Standard 802.1X-2001, October 2001.

[22] ContentGuard Holdings Inc. XrML 2.0 Technical Overview. [http://www.xrml.org/reference/XrMLTechnicalOverviewV1.pdf], March 2002.

[23] Bruno Crispo, Bogdan C. Popescu, and Andrew S. Tanenbaum. Symmetric Key Authentication Services Revisited. In *9th Australasian Conference on Information Security and Privacy (forthcoming)*, July 2004.

[24] Rachna Dhamija and Adrian Perrig. Déjà Vu: A User Study, Using Images for Authentication. In *9th USENIX Security Symposium*, August 2000.

[25] Jon Edney and William A. Arbaugh. *Real 802.11 Security*. Addison-Wesley, 2003.

[26] Carl Ellison. UPnP DeviceSecurity Service Template. [http://www.upnp.org/standardizeddcps/documents/DeviceSecurity_1.0cc_001.pdf].

[27] Carl Ellison. UPnP Security Ceremonies. [http://www.upnp.org/download/standardizeddcps/UPnPSecurityCeremonies_1_0secure.pdf].

[28] Carl Ellison and Steve Dohrmann. Public-Key Support for Group Collaboration. *ACM Transactions on Information and System Security (TISSEC)*, 6(4):547–565, November 2003.

[29] Carl M. Ellison. SPKI Requirements. RFC 2692, September 1999.

[30] Carl M. Ellison. The nature of a useable PKI. *Computer Networks*, 31:823–830, 1999.

[31] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. Simple public key certificate. IETF Internet-Draft [http://theworld.com/~cme/examples.txt], July 1999.

[32] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI Certificate Theory. RFC 2693, September 1999.

[33] Pasi Eronen and Pekka Nikander. Decentralized Jini Security. In *Network and Distributed System Security Symposium (NDSS)*, pages 161–172, February 2001.

[34] Stephen Farrell and Russell Housley. An Internet Attribute Certificate Profile for Authorization. RFC 3281, April 2002.

[35] Jim Flanagan. Authenticating XML-RPC. [http://twiki.tensegrity.net/bin/view/Main/AuthenticatingXmlRpc], July 2001.

[36] Trevor Freeman and Ajay Garg. UPnP RadiusClient Service Template. [http://www.upnp.org/standardizeddcps/documents/RadiusClientService1.0cc.pdf].

[37] The FreeRADIUS Server Project. [http://www.freeradius.org/].

[38] FreeS/WAN Project. [http://www.freeswan.org/].

[39] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL Protocol Version 3.0. IETF Internet-Draft [http://wp.netscape.com/eng/ssl3/draft302.txt], November 1996.

[40] The Globus Alliance. [http://www.globus.org/].

[41] The GNU Compiler for the Java Programming Language (GCJ). [http://gcc.gnu.org/java/].

[42] Nicholas C. Goffee, Sung Hoon Kim, Sean Smith, Punch Taylor, Meiyuan Zhao, and John Marchesini. Greenpass: Decentralized, PKI-based Authorization for Wireless LANs. In *3rd Annual PKI Research and Development Workshop*, 2004.

[43] Ian Goldberg. Visual Fingerprints. [http://www.cs.berkeley.edu/~iang/visprint.html].

[44] Dan Harkins and Dave Carrel. The Internet Key Exchange (IKE). RFC 2409, November 1998.

[45] Kipp E. B. Hickman. SSL 2.0 Protocol Specification. IETF Internet-Draft [http://wp.netscape.com/eng/security/SSL_2.html], February 1995.

[46] Russell Housley, Warwick Ford, Tim Polk, and David Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3280, April 2002.

[47] Russell Housley and Tim Moore. Certificate Extensions and Attributes Supporting Authentication in PPP and Wireless LAN. IETF Internet-Draft [http://www.ietf.org/internet-drafts/draft-ietf-pkix-wlan-extns-05.txt], March 2004.

[48] Jon Howell and David Kotz. A Formal Semantics for SPKI. Technical Report TR2000-363, Dartmouth College, March 2000.

[49] Jon Howell and David Kotz. End-to-end authorization. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 151–164, 2000.

[50] IP Security Protocol (IPSEC) working group. [http://www.ietf.org/html.charters/ipsec-charter.html].

[51] JACAPI—Java JCE Provider for Microsoft CryptoAPI. [http://www.keyon.ch/de/Produkte/JavaJCE/JACAPI/index.htm].

[52] David Johnston. Visprint, the Visual Fingerprint Generator. [http://www.pinkandaint.com/oldhome/comp/visprint/].

[53] JSDSI: A Java SPKI/SDSI Implementation. [http://jsdsi.sourceforge.net].

[54] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World*. Prentice Hall PTR, second edition, 2002.

[55] Sung Hoon Kim. Greenpass RADIUS Tools for Delegated Authorization in Wireless Networks. Master's thesis, Dartmouth College, June 2004.

[56] Juha P. T. Koponen, Pekka Nikander, Juhana Rasanen, and Juha Paajarvi. Internet access through WLAN with XML encoded SPKI certificates. In *Nordic Workshop on Secure IT Systems*, October 2000.

[57] Raph Levien. PGP Snowflake (Debian GNU/Linux package). [http://packages.debian.org/stable/graphics/snowflake].

[58] Christopher Lord, William Lupton, and Ajay Garg. UPnP LinkAuthentication Service Template. [http://www.upnp.org/standardizeddcps/documents/LinkAuthenticationService1.0cc.pdf].

[59] M2Crypto: A Python crypto and SSL toolkit. [http://sandbox.rulemaker.net/ngps/m2/].

[60] Andrew J. Maywah. An Implementation of a Secure Web Client Using SPKI/SDSI Certificates. Master's thesis, Massachusetts Institute of Technology, May 2000.

[61] Microsoft Corporation. Introducing CAPICOM. [msdn.microsoft.com/library/en-us/dnsecure/html/intcapicom.asp].

[62] Niels Möller. LSH: A GNU implementation of the Secure Shell protocols. [http://www.lysator.liu.se/~nisse/lsh/].

[63] Alexander Morcos. A Java Implementation of Simple Distributed Security Architecture. Master's thesis, Massachusetts Institute of Technology, May 1998.

[64] Mozilla Project. XPCOM. [http://www.mozilla.org/projects/xpcom/].

[65] Guillermo Navarro, Babak Sadighi Firozabadi, Erik Rissanen, and Joan Borrell. Constrained delegation in XML-based Access Control and Digital Rights Management Standards. In *CNIS03, Special Session on Architectures and Languages for Digital Rights Management and Access Control*, 2003.

[66] Sidharth Nazareth. SPADE: SPKI/SDSI for Attribute Release Policies in a Distributed Environment. Master's thesis, Dartmouth College, May 2003. [http://www.cs.dartmouth.edu/~pkilab/theses/sidharth.pdf].

[67] Sidharth Nazareth and Sean W. Smith. Using SPKI/SDSI for Distributed Maintenance of Attribute Release Policies in Shibboleth. Technical Report TR2004-485, Dartmouth College, 2004.

[68] Netscape Communications. JavaScript Guide. [http://wp.netscape.com/eng/mozilla/3.0/handbook/javascript/index.html].

[69] Pekka Nikander. Authorization and charging in public WLANs using FreeBSD and 802.1x. In *Freenix track: 2002 USENIX Annual Technical Conference*, June 2002.

[70] NoCatNet. [http://nocat.net/].

[71] The NoCat Community Wireless Network Project (unofficial RFC). [http://nocat.net/nocatrfc.txt].

[72] Nomadix Service Engine Enterprise Guest Access Application. [http://www.nomadix.com/downloads/applications/Enterprise_Guest_Access.pdf].

[73] OASIS. Security Services (SAML) Technical Committee Web site. [http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security].

[74] OASIS. XACML Technical Committee Web site. [http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml].

[75] OASIS Security Services Technical Committee. SAML Version 2.0 Scope and Work Items (working draft 18). [http://www.oasis-open.org/committees/download.php/6605/sstc-saml-scope-2.0-draft-18-diff.pdf], May 2004.

[76] OASIS Security Services Technical Committee. Technical Overview of the OASIS Security Assertion Markup Language (SAML) V1.1 (technical

committee draft). [http://www.oasis-open.org/committees/download.php/6837/sstc-saml-tech-overview-1.1-cd.pdf], May 2004.

[77] OASIS XACML Technical Committee. eXtensible Access Control Markup Language (XACML) Version 1.1 (committee specification). [http://www.oasis-open.org/committees/xacml/repository/cs-xacml-specification-1.1.pdf], August 2003.

[78] OASIS XACML Technical Committee. XACML delegation use-cases (working draft 01). [http://www.oasis-open.org/committees/download.php/5780/oasis_xacml_delegation_use-cases_wd_01.zip], March 2004.

[79] OASIS XACML Technical Committee. XACML Profile for SAML 2.0 (working draft 02). [http://www.oasis-open.org/committees/download.php/5854/wd-xacml-saml-profile-02.pdf], March 2004.

[80] OpenCA Research and Development Labs. [http://www.openca.org/].

[81] Juha Paarjavi. XML Encoding of SPKI Certificates. IETF Internet-Draft [http://xml.coverpages.org/draft-paajarvi-xml-spki-cert-00.txt], March 2000.

[82] Baiju V. Patel, Bernard Aboba, William Dixon, Glen Zorn, and Skip Booth. Securing L2TP using IPsec. RFC 3193, November 2001.

[83] Adrian Perrig and Dawn Song. Hash Visualization: a New Technique to improve Real-World Security. In *International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC)*, 1999.

[84] Personal Telco Project. PortalSoftware. [http://www.personaltelco.net/index.cgi/PortalSoftware].

[85] PGPfone: Pretty Good Privacy Phone. [http://www.pgpi.org/products/pgpfone/].

[86] Public-Key Infrastructure (X.509) (PKIX) working group. [http://www.ietf.org/html.charters/pkix-charter.html].

[87] Kimberly Powell. Testing the Greenpass Wireless Security System. Senior honors thesis, Dartmouth College, June 2004.

[88] PyCA. [http://www.pyca.de/].

[89] Carl Rigney, Steve Willens, Allan C. Rubens, and William Allen Simpson. Remote Authentication Dial In User Service (RADIUS). RFC 2865, June 2000.

[90] Ronald L. Rivest. SEXP (S-expressions). [http://theory.lcs.mit.edu/~rivest/sexp.html].

[91] Ronald L. Rivest and Butler Lampson. Cryptography and Information Security Group Research Project: A Simple Distributed Security Infrastructure (SDSI). [http://theory.lcs.mit.edu/~cis/sdsi.html].

[92] RSA Laboratories. PKCS #11 v2.11: Cryptographic Token Interface Standard.

[93] RSA Laboratories. PKCS 12 v1.0: Personal Information Exchange Syntax.

[94] RSA Laboratories. PKCS #7: Cryptographic Message Syntax Standard, November 1993. Version 1.5.

[95] Terry Schmidt and Anthony Townsend. Why Wi-Fi Wants to be Free. *Communications of the ACM*, 46(5):47–52, May 2003.

[96] Adam Shand. WhyCaptivePortal. [http://www.personaltelco.net/index.cgi/WhyCaptivePortal].

[97] Shibboleth Project. [http://shibboleth.internet2.edu/].

[98] Sean Smith, Nicholas C. Goffee, Sung Hoon Kim, Punch Taylor, Meiyuan Zhao, and John Marchesini. Greenpass: Flexible and Scalable Authorization for Wireless Networks. Technical Report TR2004-484, Dartmouth College, 2004.

[99] Sun Microsystems. Java Web Start Technology. [http://java.sun.com/products/javawebstart/].

[100] Sun Microsystems. A Brief Introduction to XACML. [http://www.oasis-open.org/committees/download.php/2713/Brief_Introduction_to_XACML.html], March 2003.

[101] SURFnet. SURFnet setup for cross-domain Authentication and Authorisation to WLAN using 802.1X. [http://www.surfnet.nl/innovatie/wlan/crossdomain.shtml]. Web page.

[102] SURFnet. What about 802.1X? An overview of possibilities for safe access to fixed and wireless networks. [http://www.surfnet.nl/innovatie/wlan/802.1Xen.pdf].

[103] W. Mark Townsley, Andrew J. Valencia, Allan Rubens, Gurdeep Singh Pall, Glen Zorn, and Bill Palter. Layer Two Tunneling Protocol "L2TP". RFC 2661, August 1999.

[104] Steven Tuecke, Von Welch, Doug Engert, Laura Pearlman, and Mary Thompson. Internet X.509 Public Key Infrastructure Proxy Certificate Profile. IETF Internet-Draft [http://www.ietf.org/internet-drafts/draft-ietf-pkix-proxy-10.txt], December 2003.

[105] UPnP Forum. [http://www.upnp.org/].

[106] VPN Consortium. VPN Technologies: Definitions and Requirements (white paper). http://www.vpnc.org/vpn-technologies.html, January 2004.

[107] Virtual Private Network Consortium (VPNC). [http://www.vpnc.org/].

[108] Ulhas Warrier, Mark Yoshitake, and Karel Van Doorselaer. UPnP WLANAccess-PointDevice device template. [http://www.upnp.org/standardizeddcps/documents/WLANAccessPointDevice1.0cc.pdf].

[109] Von Welch, Ian Foster, Carl Kesselman, Olle Mulmo, Laura Pearlman, Steven Tuecke, Jarek Gawor, Sam Meder, and Frank Siebenlist. X.509 Proxy Certificates for Dynamic Delegation. In *3rd Annual PKI Research and Development Workshop*, 2004.

[110] World Wide Web Consortium. SOAP Specifications.

[111] Wi-Fi Protected Access: Strong, standards-based, interoperable security for today's Wi-Fi networks. Wi-Fi Alliance whitepaper [http://www.wi-fi.org/OpenSection/pdf/Whitepaper_Wi-Fi_Security4-29-03.pdf], April 2003.

[112] XML-RPC Home Page. [http://www.xmlrpc.com/].

[113] XrML Web site. [http://www.xrml.org/].

[114] Wengyik Yeong, Tim Howes, and Steve Kille. Lightweight Directory Access Protocol. RFC 1777, March 1995.