Anonymous Authentication of Membership in Dynamic Groups

by

Todd C. Parnell

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

Author ....
Department of Electrical Engineering and Computer Science
May 20, 1999

Certified by ....
Lynn Andrea Stein
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Anonymous Authentication of Membership in Dynamic Groups

by

Todd C. Parnell

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 1999, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents a series of protocols for authenticating an individual's membership in a group without revealing that individual's identity and without restricting how the membership of the group may be changed. These protocols are built on top of a new primitive: the *verifiably common secret encoding.*

This thesis provides a construction for this primitive, the security of which is based on the existence of public-key cryptosystems capable of securely encoding multiple messages containing the same plaintext. Because the size of our construct grows linearly with the number of members in the group, techniques are described for partitioning groups to improve performance.

Client and server software was developed to provide transparent authentication transactions. This software served to explore practical questions associated with the theoritical framework. It was designed as a plug in replacement for use by applications using other authentication protocols.

Thesis Supervisor: Lynn Andrea Stein
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

I would first like to thank Stuart Schechter and Alex Hartemink, who were my co-authors for the original paper on the VCS framework. Working with them in my first and only foray into original crypto research was enlightening, challenging, and ultimately a whole lot of fun. Thanks to Lynn Andrea Stein for supervising this thesis after other paths were closed to me. As both of us attempted to concentrate on the Rethinking CS101 project, I appreciate her support of this additional commitment. Thanks to all those who helped with prior incarnations of the VCS framework: Matt Franklin, Michael Bender, Yanzong Ding, Nailah Robinson, and Michael D. Smith. Thanks to Ron Rivest for encouraging us to turn a half-baked class project into publishable material. And finally, thanks to Jenn, for stability and support throughout the whole thing.

Contents

Chapter 1

Introduction

Authentication systems serve to permit a specific person or groups of people access to a resource while denying access to all others. Because such systems surround us each day, we often don't notice them or their salient features. But when you use your house key, walk past a security guard at work, or log into a computer, you are using an authentication system. Use the wrong key, forget your badge, or mistype your password and access will be denied.

Sometimes when you perform an authentication transaction it is important that you in particular are being authenticated. For instance, when withdrawing money from your bank account. But in other authentication transactions all that is important is that you are a member of some specific group. Authenticating membership in a group is a common task because privileges, such as the right to read a document or enter a building, are often assigned to many individuals. While permission to exercise a privilege requires that members of the group be distinguished from non-members, members need not be distinguished from one another. Indeed, privacy concerns may dictate that authentication be conducted anonymously.

For instance, subscription services such as *The Wall Street Journal Interactive Edition* [34] require subscribers to identify themselves in order to limit service to those who pay, but many subscribers would prefer to keep their reading habits to themselves. Employee feedback programs, which require authentication to ensure that employees can report only on their satisfaction with their own supervisor, also stand to benefit from enhanced privacy. Adding anonymity protects those employees who return negative feedback from being singled out for retaliation.

Most existing systems that authenticate membership in a group do so by identifying an individual, then verifying that the individual is a member in the specified group. The requirement that an individual must identify herself to authenticate her membership can be eliminated by distributing a single group identity key (a shared secret) to be used by all group members. However, this approach makes supporting dynamic groups unwieldy: whenever an individual is removed from the group, a new group identity key must be distributed to all remaining members. Not until every member receives this key can authentication be performed anonymously.

This thesis articulates the requirements for a new type of authentication system, one which authenticates an individual's membership in a group without revealing that individual's identity and without restricting the frequency with which the membership of the group may be changed. It further details an implementation that meets the requirements, using a construct called *verifiably common secret encodings*[1].

Verifiably common secret encodings are a primitive that allow us to cast authentication properties as properties of a restricted type of public-key crypto system. Using this new primitive, this thesis builds anonymous authentication systems for dynamic groups in which a trusted party may add and remove members of the group in a single message to the authenticator. It also shows how group members may replace their authentication keys if these keys should become compromised. These protocols ensure that even if a key does become compromised, all previous and future transactions remain anonymous and unlinkable. This property is called *perfect forward anonymity.*

In addition to a theoretical framework, I have done an implementation to explore the practical questions that arise when using verifiably common secret encodings for authentication. This is the first implementation of the system, and as such serves as a reduction to practice and proof of concept. The implementation provides client and server software to support transparent authentication transactions designed to be used as a primitive by higher level protocols. After authenticating, the client and server make available a private, authenticated, bidirectional communication channel for further communications.

[1]The original work on verifiably common secret encodings [30] was done jointly with Stuart Schechter (*stuart@post.harvard.edu*) and Alex Hartemink (*amink@mit.edu*). Chapter 2 reflects the joint work.

Chapter 2

Theory[1]

This chapter introduces the verifibly common secret encoding and presents theoritical results using the new primitive. Section 2.1 introduces some notation and conventions. Section 2.2 presents a set of requirements for anonymous authentication protocols. Section 2.3 defines a verifiably common secret encoding and lists the operations supported by this primitive. Section 2.4 uses these encodings to create an elementary anonymous authentication protocol. Section 2.5 extends this elementary system to provide key replacement. Section 2.6 gives a trusted third party the ability to add and remove group members by communicating only with the authenticator. Section 2.7 shows how to encode, decode, and verify VCS vectors, an implementation of verifiably common secret encodings. Finally, section 2.8 describes how to scale anonymous authentication for very large groups. This chapter and the next assume the reader is familiar with basic cryptographic primitives[2]

2.1 Conventions

Throughout this paper, any individual requesting authentication is refered to as *Alice*. The authentication process exists to prove to the authenticator, *Bob*, that *Alice* is a member of a group, without revealing *Alice's* name or any other aspect of her identity. When a trusted third party is needed, he is called *Trent*.

All parties are assumed to have a public-key pair used for identification. Public keys are represented using the letter **p** and secret (or private) keys using the letter **s**. For any message m and key **p**, let $\{m\}_{\mathbf{p}}$ represent public-key encryption or the opening of a signature. For any message m and key **s**, let $\{m\}_{\mathbf{s}}$ represent public-key decryption or signing. Symmetric encryption of message m with key k is represented as $E_k[m]$. When necessary, messages to be signed are appended with a

[1]**This chapter was originally published [30] as joint work with Stuart Schecter of Harvard and Alex Hartemink of MIT.**

[2]Readers not familiar with cryptographic primitives such as public key cryptosystems or certificates are encouraged to read a general introduction to the subject before proceeding. [31] is an excellent starting point.

known string to differentiate them from random strings. Messages sent by either *Bob* or *Trent* are also assumed to include a timestamp.

The set **P** is a set of public keys associated with a group. An individual whose public key is in **P** is called a *member* of **P**. More precisely, a member of **P** is an individual possessing a secret key **s** corresponding to a public key $\mathbf{p} \in \mathbf{P}$, such that for the set M of messages that may be encoded using **p**, $\forall m \in M$, $m = \{\{m\}_{\mathbf{p}}\}_{\mathbf{s}}$. To be authenticated anonymously is to reveal only that one is a member of **P**. This definition of *anonymity* provides privacy only if there are other members of **P**. We thus assume that the set **P** is public knowledge and that one can verify that the public keys in **P** are associated with real individuals.

Finally, assume that all communication takes place over an anonymous communication channel [5, 1, 25, 26]. This prevents an individual's anonymity from being compromised by the channel itself.

2.2 Requirements for Anonymous Authentication Protocols

The following three requirements are essential to anonymously authenticate membership in **P**.

> SECURITY: *Only members of* **P** *can be authenticated.*
>
> ANONYMITY: *If an individual is authenticated, she reveals only that she is a member of* **P**. *If she is not authenticated, she reveals nothing.*
>
> UNLINKABILITY: *Separate authentication transactions cannot be shown to have been made by a single individual.*

Note that the above definition of *anonymity* is the broadest possible, since *security* requires that only members of **P** can be authenticated.

The authenticator may choose to compromise *security* by authenticating an individual who is not a member of **P**. Similarly, an individual may choose to forfeit her *anonymity* by revealing her identity. Therefore, it is safe to assume that authenticators act to maintain security and that individuals act to preserve their own anonymity.

The above requirements do not account for the fact that membership in **P** is likely to change. Moreover, people are prone to lose their keys or fail to keep them secret. For a system to be able to address these concerns, the following requirements are also needed:

> KEY REPLACEMENT: *A member of* **P** *may replace her authentication key with a new one and need only confer with the authenticator to do so.*
>
> DYNAMIC GROUP MEMBERSHIP: *A trusted third party may add and remove members of* **P** *and need only confer with the authenticator do so.*

To make membership in **P** dynamic, a third party is trusted to add and remove members. If this third party is not trustworthy, he can manipulate the set **P** to reduce *anonymity*. For instance, if he shrinks **P** so that the group contains only one member, that member's identity will be revealed during her next authentication transaction.[3]

2.3 Verifiably Common Secret Encodings

Begin with a set of public keys, **P**. Recall the definition of *member* of **P** to be an individual possessing a secret key **s** corresponding to a public key $\mathbf{p} \in \mathbf{P}$. A *verifiably common secret encoding* e, of a value x, has the following properties:

SECRECY: *Only members of* **P** *can decode* e *to learn* x.

COMMONALITY: *Any member of* **P** *can decode* e *and will learn the same value* x *that any other member of* **P** *would learn by decoding* e.

VERIFIABILITY: *Any member of* **P** *can determine whether commonality holds for a given value* e, *regardless of whether* e *is properly constructed.*

This primitive can be manupulated using the following three operations:

$$e \leftarrow \text{ENCODE}(x, \mathbf{P})$$

$$x \leftarrow \text{DECODE}(e, \mathbf{s}, \mathbf{P})$$

$$\textit{isCommon} \leftarrow \text{VERIFY}(e, \mathbf{s}, \mathbf{P})$$

In the next three sections, these three functions are used to build anonymous authentication protocols. Section 2.7, provides a concrete algorithmic implementation for these functions.

2.4 Anonymous Authentication

This section presents a simple anonymous authentication protocol that satisfies the requirements of *security*, *anonymity*, and *unlinkability*. It establishes a session key y between *Alice* and *Bob* if and only if *Alice* is a member of **P**. The protocol will serve as a foundation for more powerful systems providing *key replacement* and *dynamic group membership* to be described in Sections 2.5 and 2.6.

This protocol requires that *Bob* be a member of **P**. If he is not, both *Alice* and *Bob* add $\mathbf{p}_{bob}$ to **P** for the duration of the authentication transaction.

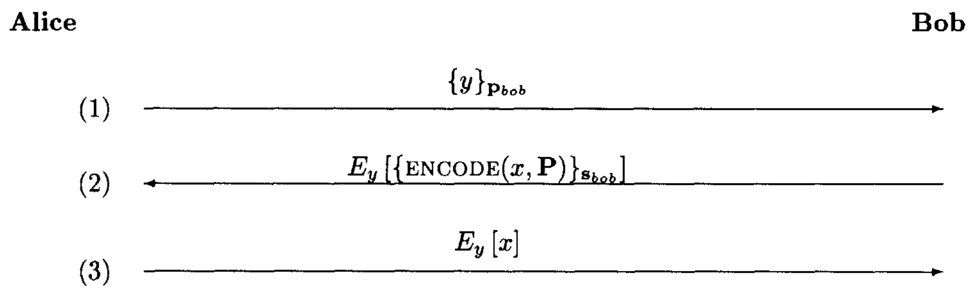

Figure 2-1: An Elementary Anonymous Authentication Transaction

2.4.1 The Authentication Protocol

Before the authentication transaction in Figure 2-1 commences, *Alice* randomly selects a session key y. She then encrypts y with *Bob's* public key to form message (1). This message, which represents a request for authentication, may also be augmented to specify the group in which *Alice's* membership is to be authenticated.

In response, *Bob* randomly picks x. He creates a message containing a verifiably common secret encoding of x, signs it, and then encrypts with the session key y. He sends this to *Alice* as message (2).

Alice decrypts the message and verifies *Bob's* signature to reveal a value e. If $\text{VERIFY}(e, \mathbf{s}_{alice}, \mathbf{P})$ returns true, *Alice* is assured that e is an encoding that satisfies *commonality*. Only then does she use $\text{DECODE}(e, \mathbf{s}_{alice}, \mathbf{P})$ to learn x. If $\text{VERIFY}(e, \mathbf{s}_{alice}, \mathbf{P})$ returns false, *Alice* cannot be assured that e satisfies *commonality* and halts the transaction.

In message (3), *Alice* proves her membership in $\mathbf{P}$ by encrypting x with the session key y. Upon decrypting message (3) to reveal x, *Bob* concludes that *Alice* is a member of $\mathbf{P}$. Authenticated, private communications between *Alice* and *Bob* may now begin.

Alice may later wish to prove that it was she who was authenticated in this transaction. Appendix A shows how *Alice* may request a receipt for this transaction. With such a receipt in hand, *Alice* may, at any point in the future, prove the transaction was hers.

2.4.2 Satisfying the Requirements

Secrecy ensures that only members of $\mathbf{P}$ can decode e to learn x. *Security* is therefore maintained because an individual is authenticated only when she can prove knowledge of x. By requiring that *Bob* be a member of $\mathbf{P}$ we prevent *Bob* from staging a man in the middle attack in which he uses *Alice* to decode a verifiably common secret encoding that he would not otherwise be able to decode.

[3] In the case that a trusted third party cannot be agreed upon, *anonymity* can still be protected by imposing rules governing the ways in which $\mathbf{P}$ can be modified. These rules should be designed to prevent any excessive modification of $\mathbf{P}$ that might compromise *anonymity*. Violations of the rules must be immediately detectable by an individual when she receives changes to the membership of $\mathbf{P}$ during authentication.

Commonality guarantees that any member of $\mathbf{P}$ can decode e and will learn the same value x that any other member would learn by decoding e. If *Alice* is certain that e exhibits *commonality*, it follows that by using x to authenticate her membership, she reveals nothing more than that she is a member of $\mathbf{P}$.

Verifiability is required so that *Alice* may prove for herself that the encoding e exhibits *commonality*, even though she did not create this encoding. Thus, by sending message (3) only when VERIFY() returns true, *Alice* ensures that her authentication will be both anonymous and unlinkable. If *Bob* should be malicious and attempt to construct e in a way that would allow him to discover *Alice's* identity from her decoding of e, verification will fail. *Alice* will halt the transaction before she decodes e. Since message (2) must be signed by *Bob*, *Alice* can use the signed invalid encoding as proof of *Bob's* failure to follow the protocol.

The authentication transaction appears the same regardless of which member of $\mathbf{P}$ was authenticated. As a result, even an otherwise omniscient adversary cannot learn which member of $\mathbf{P}$ was authenticated by inspecting the transaction. Thus, even if *Alice's* key is compromised before authentication, the transaction remains anonymous and unlinkable. We call this property *perfect forward anonymity.*

2.5 Key Replacement

In the protocol above, *Alice* uses a single key pair $(\mathbf{p}, \mathbf{s})$ to represent both her identity and her membership in the group. Because she uses the same key pair for both functions, an adversary who compromises her secret key $\mathbf{s}$ can not only authenticate himself as a member of $\mathbf{P}$, but can also pose as *Alice* in any other protocol that uses $\mathbf{s}$. Ideally, compromising the key used in the authentication process should not compromise *Alice's* identity. By using two key pairs, one to represent her identity and one for authentication, *Alice* significantly reduces the potential for damage should she lose her authentication key. Using two key pairs for the two separate functions also enables *Alice* to replace a lost authentication key.

The pair $(\mathbf{p}, \mathbf{s})$ continues to identify an individual. Each member of $\mathbf{P}$ now generates an authentication key pair $(\mathbf{p}', \mathbf{s}')$ for each group in which she is a member. Because of the severe consequences of losing $\mathbf{s}$, it is safe to assume that $\mathbf{s}$ is kept well guarded. Because only $\mathbf{s}'$ will be needed during the authentication transaction, only the case where an authentication key $\mathbf{s}'$, not an identity key $\mathbf{s}$, is lost or compromised is considered. When $\mathbf{s}'$ is lost or compromised, the individual can disable the key and obtain a replacement by conferring only with the authenticator.

In order to validate her public authentication key $\mathbf{p}'$, each member uses her secret identity key $\mathbf{s}$ to sign a certificate $c = \{\mathbf{p}'\}_{\mathbf{s}}$. This certificate can be opened to reveal the public authentication key as follows: $\{c\}_{\mathbf{p}} = \{\{\mathbf{p}'\}_{\mathbf{s}}\}_{\mathbf{p}} = \mathbf{p}'$.

To initialize the system, all members of $\mathbf{P}$ send their certificates to *Bob*. *Bob* collects all the certificates to form the set $\mathbf{C}$. The set of public authentication keys, $\mathbf{P}'$, can then be generated by opening each certificate in $\mathbf{C}$: $\mathbf{P}' = \{\{c_i\}_{\mathbf{p}_i} : c_i \in \mathbf{C}\}$.

2.5.1 Modifications to the Authentication Protocol

The only modification to the authentication protocol is to require *Bob* to add the set of certificates $\mathbf{C}$ to message (2). The augmented message will be labeled (2a):

Alice **Bob**

$$(2a) \quad \xleftarrow{\quad E_y\left[\{\mathbf{C}, \text{ENCODE}(x, \mathbf{P}')\}_{\mathbf{s}_{bob}}\right] \quad}$$

From the set of certificates $\mathbf{C}$ and public identity keys $\mathbf{P}$, *Alice* computes $\mathbf{P}'$ using the technique shown above. She then verifies e using $\text{VERIFY}(e, \mathbf{s}'_{alice}, \mathbf{P}')$. If the encoding exhibits *commonality*, *Alice* learns x from $\text{DECODE}(e, \mathbf{s}'_{alice}, \mathbf{P}')$.

2.5.2 The Key Replacement Transaction

If *Alice* believes her secret authentication key has been compromised, she simply generates a new authentication key pair, creates a certificate for the new public authentication key, and sends that certificate to *Bob*. *Bob* returns a signed receipt to *Alice* acknowledging the new certificate. Since we assume that *Bob* acts to maintain security, we expect him to use *Alice's* new certificate and authentication key.[4]

2.6 Dynamic Group Membership

This section describes how a trusted third party, *Trent*, may be given sole responsibility for maintaining the set of certificates $\mathbf{C}$. To this end, *Alice* requires that any $\mathbf{C}$ used by *Bob* be signed by *Trent*. During the authentication transaction, message (2a) is replaced by message (2b):

Alice **Bob**

$$(2b) \quad \xleftarrow{\quad E_y\left[\{\{\mathbf{C}\}_{\mathbf{s}_{trent}}, \text{ENCODE}(x, \mathbf{P}')\}_{\mathbf{s}_{bob}}\right] \quad}$$

If *Alice* is to be granted membership in $\mathbf{P}$, she generates an authentication key pair, creates the certificate c_{alice}, and sends it to *Trent* who updates $\mathbf{C}$ and distributes a signed copy to *Bob*.

[4]Even if *Bob* fails to use the new certificate, *Alice* can either proceed using her old key (in the case that it was compromised and not lost) or can use the signed message (2a) as proof of *Bob's* failure to use the new certificate.

To remove *Alice* from $\mathbf{P}$, and thereby prevent her from being authenticated, *Trent* simply removes *Alice's* certificate c_{alice} from $\mathbf{C}$ and distributes a signed copy to *Bob*. In both cases, *Bob* and other members of $\mathbf{P}$ can compute the new $\mathbf{P}'$ using $\mathbf{P}$ and the new set of certificates $\mathbf{C}$.

2.7 Constructing Verifiably Common Secret Encodings

This section shows how to use public-key cryptography to construct verifiably common secret encodings that we call VCS vectors. Assuming that M_i represents the set of messages that can be encrypted by a public key $\mathbf{p}_i \in \mathbf{P}$, the set of messages that can be encoded as a VCS vector for group $\mathbf{P}$ is $\mathbf{M} = \bigcap M_i$.

A *VCS vector* encodes a value x as follows:

$$\vec{e} \leftarrow [\{x\}_{\mathbf{p}_1}, \{x\}_{\mathbf{p}_2}, \cdots, \{x\}_{\mathbf{p}_n}] \text{ where } n = |\mathbf{P}|$$

Encoding, decoding, and verifying VCS vectors can be performed by the following three functions:

$$\text{ENCODE}(x, \mathbf{P}): \qquad \vec{e} \leftarrow \begin{cases} [\{x\}_{\mathbf{p}_1}, \{x\}_{\mathbf{p}_2}, \cdots, \{x\}_{\mathbf{p}_n}] & x \in \mathbf{M} \\ [] & x \notin \mathbf{M} \end{cases}$$

$$\text{DECODE}(\vec{e}, \mathbf{s}_i, \mathbf{P}): \qquad x \leftarrow \{\vec{e}[i]\}_{\mathbf{s}_i}$$

$$\text{VERIFY}(\vec{e}, \mathbf{s}_i, \mathbf{P}): \qquad \textit{isCommon} \leftarrow \vec{e} = \text{ENCODE}(\text{DECODE}(\vec{e}, \mathbf{s}_i, \mathbf{P}), \mathbf{P})$$

When using VCS vectors, *secrecy* holds only if x is not revealed when encrypted multiple times with different public keys. This is not true of RSA with small exponents or Rabin [16, 15, 9]. For this reason, caution must be exercised when selecting a public-key encryption technique.

Commonality holds because any secret key corresponding to a key in $\mathbf{P}$ can be used to decode $\vec{e}$ to learn x. Decrypting $\vec{e}[i]$ with $\mathbf{s}_i$ yields the same secret x for all i.

Any member of $\mathbf{P}$ can use DECODE() to learn x from $\vec{e}$ and then re-encode x using ENCODE() to obtain a valid encoding of x. Because ENCODE() generates a valid encoding, *commonality* will hold for this re-encoded vector. If the re-encoded vector equals the original vector $\vec{e}$, then $\vec{e}$ must also satisfy *commonality*. Hence, as long as ENCODE() is deterministic,[5] any member can verify the commonality of any encoding $\vec{e}$. Consequently, *verifiability* is satisfied.

That the VERIFY() operation can be expressed as a simple composition of the ENCODE() and DECODE() operations is a general statement, independent of how a verifiably common secret encoding is constructed. For this reason, if ENCODE() and DECODE() operations can be devised for which

[5] Probabilistic encryption [14, 3] may still be used under the random oracle model. In this case, make the ENCODE() function deterministic by using its first input parameter, the secret x, to seed the pseudo-random number generator with $\mathcal{O}(x)$.

commonality holds, *verifiability* becomes automatic. Thus, the implementation-specific definition of VERIFY() can be replaced with a general definition:

$$\text{VERIFY}(e, \mathbf{s}, \mathbf{P}): \qquad \textit{isCommon} \leftarrow e = \text{ENCODE}(\text{DECODE}(e, \mathbf{s}, \mathbf{P}), \mathbf{P})$$

2.8 Making Anonymous Authentication Scalable

The number of entries in a VCS vector grows linearly with the number of members of **P**, as does the time required to generate, transmit, and verify the entries. This growth could make anonymous authentication impractical for very large dynamic groups.

This issue can be addressed by authenticating using subsets of **P**. Individuals will now remain anonymous and unlinkable only among the members of their subset rather than among all members of **P**. Because membership in a subset of **P** implies membership in **P**, *security* is not affected. Two ways of assigning subsets are: random generation of single-use subsets during each authentication transaction and the use of a static assignment algorithm.

2.8.1 Single-Use Subsets

During each authentication transaction, *Alice* selects a subset of **P** at random. To ensure her membership, *Alice* augments the subset to include herself. She sends this subset to *Bob* when requesting authentication. *Alice* and *Bob* then use this subset in place of **P** for the remainder of the protocol.

Alice picks her subset of **P** at the time she initiates the authentication transaction. If she has limited long-term storage, she can select the subset by picking keys in **P** by their indices. She then requests keys in **P** from *Bob* by index at the start of the authentication transaction. To prevent *Bob* from sending fraudulent identity keys, *Alice* maintains a hash tree of the keys or their fingerprints.

Alice must be cautious when using single-use subsets. If external circumstances link two or more transactions, *Alice* is anonymous only among the intersection of the subsets used for authentication.

2.8.2 Statically Assigned Subsets

Subsets may also be assigned by a static algorithm such that each member of **P** is always assigned to the same subset $\mathbf{P}_i \subseteq \mathbf{P}$ where $\bigcup \mathbf{P}_i = \mathbf{P}$. These subsets may change only when members are added or removed from **P**. As above, *Alice* uses $\mathbf{P}_i$ wherever she previously would have used **P**.

Even if *Trent* picks the subsets, he may do so in a way that unwittingly weakens anonymity or unlinkability. Using a one-way hash function, preferably generated randomly before the membership is known, ensures that no party can manipulate the assignment of individuals to subsets.

2.9 Summary

This chapter began by developing simple set of requirements needed to perform anonymous authentication of membership in dynamic groups. We explored a primitive, the verifibly common secret encoding, that has the necessary properties to meet the requirements. Then, using this primitive and its associated operations, we developed a full featured protocol that can performs the necessary authentication transaction. Further, we provided a basic theoritical implementation, the VCS Vector, that implements the methods required by the abstract verifibly common secret encoding. Finally, because VCS Vectors scale linearly with the size of the group they encode for, we discussed some strategies to accomodiate large groups by using two different subgroup techniques.

Chapter 3

Implementation

The verifibly common secret encoding framework is theoretically applicable to many practical authentication problems. Users of the Wall Street Journal Interactive Edition, riders on local mass transit, or MIT community members accessing a campus building would all benefit from a VCS style authentication mechanism. The benefits of anonymity and unlinkability are strong motivation for such a system from an individual's perspective, and the key replacement and dynamic membership make the system attractive from the authenticatior's perspective also. The question is, are verifibly common secret encodings as attractive *in practice* as they are in theory? This chapter presents client and server software written to implement verifibly common secret encodings and VCS vectors. The software is an attempt to answer the above question and to explore the problems any implementation will confront.

Section 3.1 discusses the high-level goals of this implementation and its role in other systems. Section 3.2 explores the extent of the implementation, including difficulties encountered with the various tools used. Section 3.3 details the various module implementations. Section 3.4 answers the questions posed here regarding practicality. This chapter concludes in Section 3.5 with a look into possible future works based on this implementation.

3.1 Goals

As noted above, the overarching goal of this implementation is to begin to answer the question, is VCS style authentication practical for real world systems? To answer this question, I identified 3 key sub-goals:

Proof of Concept The VCS framework is firmly grounded in sound theoretical principals, but the difference between theory and practice is much wider in practice than in theory. This implementation serves as initial proof of concept or reduction to practice of the ideas presented in Chapter 2.

Identify Problem Areas Security products are notoriously difficult to use. The initial implementation forces the identification of the difficulties any implementation will face. An important goal was to provide a framework that not only identified the problem areas, but provided model solutions as well.

Effective API Authentication software is never an end in and of itself. Instead, this type of software is built into a browser, smart card or some other system for use whenever authentication is required. Therefore, an authentication package should allow a variety of client software to use the authentication code. This meant designing an effective API on both the client and server side for use in many different environments.

These three goals are necessary, but not sufficient, to determine whether VCS style authentication is practical for real world systems. This implementation is *not* designed to show the absolute speed of VCS in an optimimized design. To perform that evaluation would require many assumptions about the ultimate client and server hardware, as well as the number and frequency of authentication transactions. Further, the tools used here have shortcomings that preclude their use in most systems. Instead, this implementation provides some broad data about the type of hardware necessary, and the types of systems others should consider good candidates. Also, this implementation was not designed to be production quality. Some key features are missing, and a more demanding threat model would likely necessitate changes. See Section 3.5 for a discussion of these changes.

3.2 Design Decisions

3.2.1 Java and the Java Cryptography Extensions

The Java programming language is used for my implementation, a choice which has pros and cons. On the good side, Java runs on a variety of platforms. I wanted this implementation to be useful in many environments, and the easiest way to achieve this goal was to use a cross-platform language. Other laundry-list Java benefits include ease of implementation, OO design, networking support, and maintainability.

Aside from the normal benefits of Java, security applications in particular benefit from Java's library support for cryptography. The Java Cryptography Extension (JCE) is now a standard extension for the Java 2 platform and provides an implementation independent way to access cryptographic primitives [19, 22]. Using the JCE allows high level abstraction and permits the primitive cryptographic work to be done by different modules depending on need. Thus if 100% Pure Java is a goal, you may choose a cryptographic provider that uses only Java in its implementation. If raw speed is required, simply plug in a different provider that uses C or assembler. Switching between the two requires no source code changes[1].

The major technical complaint about Java is speed. The garbage collection and runtime array bounds checking performance penalties are legimite problems in certain applications. However, in this case the speed issue is mitigated by the ability to use a very fast native implementation for the core cryptography functions. In addition, Sun's newly announced Java HotSpot Performance Engine [18] allows server side code to run nearly as fast as C. Section 3.3.2 discusses the speed impact in relation to the server.

The JCE has its own shortcomings. Most notably, the JCE has no programatic way to create a certificate. The implications of this are discussed is Section 3.2.3.

3.2.2 RSA Library

The main JCE provider I used for this implementation was the RSA BSAFE Crypto-J library [27], Version 2.1. This is a commercial library available only inside the US due to export controls. There were two main reasons for this choice. First, the RSA product is compatible with the Java 2 platform JCE (version 1.2). The Java 2 platform is relatively new and the updated JCE API is incompatible with previous versions. Crypto-J was the first third-party vendor to support the new standards. Second, Crypto-J contains public key cryptography primitives, in the form of RSA. Most other popular JCE providers [19, 20, 11] either support JCE v1.1 or do not contain public key cryptography primitives, or both. Recently other vendors developed JCE 1.2 products [2, 17, 13], but their release dates were too late to be included in my implementation. Hopefully in the near future more providers will support the new JCE API and add RSA primitives when the RSA patent expires[2].

Crypto-J contains both 100% Pure Java and C versions of the core cryptographic functions. This makes it an ideal candidate for implementations where speed is important. In fact, should an implementation choose to forgo the JCE framework and use the Crypto-J libraries directly, the choice between Java and C implementations can be made at runtime. However, this choice ties the implementation to Crypto-J. Using the JCE provider methods, only the Java version of the Crypto-J libraries can be accessed.

I attempted an implementation of VCS vectors using the C implementation of Crypto-J. This might have allowed meaningful performance evaluations of the VCS framework. However, in order to maintain compatibility with the client and server (implemented in Java), I was forced to store the RSA keys in a format readable by Java. This caused the C libraries to perform a conversion

[1]This is only strictly true in cases where the cryptographic providers are statically specified. For dynamically specified providers, a single line would need to be changed.

[2]August 2000.

between the Java formatted key and the native format every time an RSA key was used. Profiling results show that the C implementation of RSA was 5–10 times faster than the Java implementation. However, the conversion between formats was very costly and negated all benefits from using the native code. If performance is a goal, it is imperative to avoid conversion of keys.

There is currently a major bug in the Crypto-J library that makes its use for VCS authentication unattractive. Crypto-J uses pointer comparison to determine whether two RSA keys are equal or not: two RSA public keys with equal modulus and public exponent are only equal if they are stored in the same memory location. This makes the Java method `equals(Object o)` essentially useless for RSA keys. Naturally, this bug impacts performance, since Java uses the equals method for things like hash table lookup. Crypto-J is therefore not currently a good choice for VCS authentication in performance critical applications[3].

3.2.3 Key Management

This section looks at how the various key management functions are implemented. The theoretical framework pushes questions of key management entirely to the implementation, and resolving those questions is probably one of the hardest problems any implementation will face. One question is how can an implementation ensure that **P** is public? This question will be faced by all implementations.

The second major question this implementation faced was how to perform authentications without certificates. As noted above, Java has no programatic way to create certificates. This makes the authentication key pair mechanism (Section 2.5) impossible to implement. Java *can* verify standard certificates created by outside sources and the JDK provides an auxiliary program to create DSA [23] certificates[4]. Still, RSA certificates cannot be created by either the JDK or the Crypto-J library. Another mechanism is needed to achieve dynamic membership and key replacement.

Making P Public Knowledge

An assumption of the theoretical system presented in Section 2.1 that the set **P** is public knowledge. The problem of implementing this requirement is a difficult key management issue. Solutions such as a web of trust or hierarchical certificates are extremely powerful, but are overkill for the implementation at hand. This implementation uses a trusted shared repository model. In this model, every member has access to a single place where all public keys are kept. The repository is trusted in the sense that if a public key exists in the repository, it is assumed that the key is valid. A repository model can be implemented as anything from a simple hash table to a full database.

My implementation uses a single shared AFS directory that all clients and the server have read access to as the repository. The file `<principal>.pub` is assumed to contain the public key for

[3] RSA has been notified of this problem. I expect a bug-fix by the next release.

[4] Unfortunately, DSA certificates are useless here since we need the ability to encrypt messages, not just sign them.

the associated principal. The client and server software are not given modify access to the AFS directory at runtime, but can freely use any certificate already present. To add a new principal, an AFS principal with insert access to the directory runs a special program that generates a new key pair. This design in is line with the idea that identity keys are never lost or compromised (Section 2.5).

Performing Trent's Job Without Certificates

Trent performs two functions. First, he maintains a list of all current public authentication keys. He stores these keys as a set of certificates he received from the members of **P**. Second, he signs the set of certificates for use during the authentication transaction. Part of this second function is updating the set of certificates whenever **P** changes. Since we cannot create certificates, some changes to the protocol are in order.

Instead of signed certificates, my implementation uses a signed list of all principals currently in **P**. That is, in the repository there exists a file (`VCS.defaults`[5]), that tells both the server and any clients the current membership of **P**. This list is signed by Trent. Thus, if the current repository has `user1.pub, user2.pub, user3.pub,` and `user4.pub`, the `VCS.defaults` file might contain `user1, user2` to indicate that only half of the principals are currently in **P**. Both the client and server software check the current list during the authentication transaction. Note that it is not necessary for any communication between client and server to include the current membership of **P**, as is required using certificates. The signed list accommodates dynamic membership, but fails to provide for key replacement. Trent can easily modify membership by adding and removing users in the `VCS.defaults` file. His changes are immediately visible to both client and server, so we have fulfilled the requirement that at most one message from Trent to the authenticator can occur. However, key replacement is not satisfied. The only way a key can be replaced in the repository is for Trent to delete the old key and insert a new key file with the same principal name. However, there is no authenticated, secure way a member of **P** can communicate the need to replace their key if they have lost their old key. There appears to be no way to provide for true key replacement without certificates. Instead, my implementation forces Trent and the member of **P** requesting key replacement to do so outside of the VCS framework.

3.2.4 Extent of Implementation

Without certificates it is impossible to implement the full protocol as described in in Section 2.6. Instead I use the shared repository described above to dynamically determine the current members

[5]An unfortunate name. `Current_Membership` would be more appropriate. However, the file name can be specified at runtime, so this is not too much of a concern.

of **P**. With this change from certificates to repository, the implementation here models Figure 2-1. Remember that this version of the protocol meets the security, anonymity, and unlinkability requirements but (by itself) fails to provide key replacement and dynamic membership. Dynamic membership *is* provided for since Trent can modify the `VCS.defaults` file at will to affect the current membership. However, as discussed above, key replacement seems to be impossible without the ability to create certificates.

The current version of the implementation does not provide the receipt mechanism described in Appendix A.

3.2.5 Threat Model

The threat model for this implementation is for a passive adversary. The adversary is assumed to be computationally limited and is restricted to passive listening on the communication channel. The adversary can initiate communications with the authenticator (server), just as any member of **P** can. I assume the adversary cannot break the cryptography (RSA and RC4), nor can the attacker monitor the client and server processes internally as they run.

As noted in Section 2.1, all communication must take place over anonymous channels for anonymity to be maintained. This implementation does not make use of anonymous channels. Thus an additional assumption in the threat model is that the server can neither compromise the client's anonymity nor link access attempts by examining the routing information from the underlying channel. This means the client must set up an anonymous channel before beginning the authentication transaction.

This threat model is appropriate for authentications occurring over the internet where the adversary can listen to packets between client and server at another network node. It would not be adequate if the adversary was allowed access to either the client or server machines. In those cases care would need to be taken to avoid leaving keys either in virtual memory or on disk. Java could not be used in such cases, since it does not allow control over paging[6].

3.3 Module Overviews

This section presents detailed information about the important modules in this implementation.

3.3.1 VCS Vector Implementation

VCS Vectors were introduced in Section 2.7 as the necessary primitive to implement anonymous authentication transactions in dynamic groups. This section presents changes made to VCS vectors

[6]Crypto-J includes an obfuscation mode to keep unencrypted keys from being paged to disk, but this feature ties the implementation to the RSA code.

necessary to efficiently implement the authentication transaction. Recall that A VCS vector encodes a value x as follows:

$$\vec{e} \leftarrow [\{x\}_{\mathbf{p}_1}, \{x\}_{\mathbf{p}_2}, \cdots, \{x\}_{\mathbf{p}_n}] \text{ where } n = |\mathbf{P}|$$

The first problem we encounter with the theoretical construct is the decode operation:

$$\text{DECODE}(\vec{e}, \mathbf{s}_i, \mathbf{P}): \quad x \leftarrow \{\vec{e}[i]\}_{\mathbf{s}_i}$$

The decode operation assumes the recipient of a VCS vector knows *a priori* the correct index i. This may be a reasonable assumption if we were not trying to support dynamic group membership, but with the ability to revoke members comes some difficulties. The recipient of a VCS vector cannot be expected to maintain an absolute index relative to an ever changing group.

There are two possible solutions. First, we can impose an ordering on all keys in $\mathbf{P}$ and $\mathbf{P}'$, then require that a VCS vector is encoded using that ordering. The ordering might be as complicated as the natural ordering of the keys based upon their encoded representations, or as simple as the order in which $\mathbf{P}$ or $\mathbf{P}'$ is transmitted during the protocol. In either case, the recipient of the VCS vector will be able to determine the ordering and index into the array of encoded values. A second solution changes the fundamental representation from an array to a map between key and encoded secret. The recipient of this encoding merely performs a hashtable lookup on their key to find the correct encoded value.

Generally speaking, the first solution will be superior to the second. Keys do have a natural ordering, and the burden associated with maintaining that order on the server is quite small. The second solution will require that the set $\mathbf{P}'$ be transmitted *twice* in message 2. One time will be *Trent's* signed set, and the second will be part of the hashtable. The second solution further requires additional processing on the client side, since the hashtable must be placed into memory. In general, the first solution will be superior in most implementations.

Despite this, the second solution is used here. Since we are using the shared repository model, both the client and server know the current membership of $\mathbf{P}$; there is no particular reason to send $\mathbf{P}$ seperately. However, if we are to remain faithful the the amount of communication bandwidth required, we should send the set in some form. The current implementation only sends $\mathbf{P}$ once, but does so as part of the hashtable.

But why bother with the hashtable at all? For reasons explained in the next section, two useful additions to the VCS vector API are:

$$\vec{e}\,' \leftarrow \text{ADD } (\vec{e}, \mathbf{p}_{bob}, x)$$

$$\vec{e}\,' \leftarrow \text{REMOVE } (\vec{e}, \mathbf{p}_{bob})$$

Add takes a VCS vector, a new public key to encode for, and the secret x and returns a new VCS vector $\vec{e}'$. $\vec{e}'$ is equivalent to $\text{ENCODE}(x, \mathbf{P} \cup \mathbf{p}_{bob})$. Remove takes a VCS vector, a public key the vector was encoded for, and returns a new VCS vector $\vec{e}'$. This $\vec{e}'$ is equivalent to $\text{ENCODE}(x, \mathbf{P} \backslash \{\mathbf{p}_{bob}\})$. The implementation of VCS vector here supports these additional operations, which makes the use of a hashtable desirable. Rather than use hashtables internally to support add and remove, then linearize into an array for transmission (and include **P** seperately) I chose to send the hashtable between client and server[7].

3.3.2 Server Implementation

The server module corresponds to *Bob* in Figure 2-1. Its main purpose is to accept connections from clients and differentiate between authorized users (members of **P**) and others attempting to gain access. This module would normally be run by content providers as a gatekeeper to content such as web pages. When the server authenticates a connection, it returns a private, secure, and authenticated I/O stream to web server or other content-providing service.

Public Interface

The most important public methods are[8]:

`Server(String passphrase)` This is the constructor for the `Server` class. This method constructs a server using typical values for the encryption algorithm, keysize, and TCP/IP parameters. These values (and others) can all be set after construction using one of the public setter methods. See the source code for all available options.

`static void main(String[] args)` The command line version of the server allows detailed debugging of all aspects of the server. When run in this mode, clients are authenticated, but the private, secure, authenticated I/O stream is discarded after authentication. This mode is useful for understanding the protocol in a hands-on manner.

`void start()` The start method tells the server to begin listening for TCP/IP connections from clients. This method also starts a background thread that populates a cache of VCS vectors for improved performance when clients connect. The start method is *not* called during construction by default.

`void stop()` The stop method tells the server to stop listening for TCP/IP connections from clients. All current connections are allowed to complete: no connection is terminated by calling stop. This method also kills the background cache-populating thread and flushes the cache.

[7]The time to linearize the hashtable and the time to reconstruct the hashtable on the client side are about the same. There is no speed advantage to linearizing.

[8]Full source code for this and all other classes is in Appendix B.

`Server.Connection getConnection()` The getConnection method retrieves an authenticated connection from the server. The `Server.Connection` returned contains the private, secure, authenticated I/O stream associated for a single client. This method blocks until a client is authenticated. If more than one connection is ready to be serviced, the oldest outstanding connection is returned.

`addPrincipal(String principal)` addPrincipal adds the given principal to **P**. Connections established by clients after this method is called will allow this principal to authenticate. Important note: the shared repository model requires that the file `principal.pub` exist in the repository for this method to succeed.

`revokePrincipal(String principal)` The inverse of addPrincipal; removes the given principal from **P**. Connections established by clients after this method is called will disallow this principal to authenticate. This method has no effect if the given principal is not a member of **P**.

Life Cycle

Typically, a server goes through the following stages:

1. A server is created.
2. *(optional)* Any non-default parameters are set with a setter method.
3. The `start` method is called.
4. Connections are authenticated by the server and handled via calls to `getConnection`. During this time users may be added or revoked with calls to `addPrincipal` or `revokePrincipal`.
5. The `stop` method is called.

Optimizations

This server was designed to provide low latency responses both to clients wishing to authenticate and to higher level code that needs to add or revoke users. From the life cycle above, we expect the server to spend the majority of its time answering connections and adding and revoking users. To provide low latency, two different optimizations are needed.

The first optimization is to use a queue of pre-constructed VCS vectors to allow instant responses to client requests. In the theoretical protocol, *Bob*, the server, constructs a new VCS vector only after a client connects. Unfortunately, this leads to high latency from the client's point of view since the server must do many public key operations to create a VCS vector. Instead, this implementation uses a seperate thread to populate a cache of VCS vectors for instant use in a connection. This optimization allows the expensive public key operations to be performed when load on the server is low.

Populating the cache in the background is an excellent idea to aid client response time, but necessitates extending the theoretical VCS vector API with add and remove operations (see previous section). Performance suffers if we attempt background queueing in conjunction with dynamic membership without these extensions. In such a case, the cache must be flushed for every add and remove since the cached vectors will authenticate the old set **P**. This can lead to performance worse than not caching at all. We thus optimize addition and revocation by including add and remove in the VCS vector API.

Other Implementation Notes

Since the server does nothing beyond the bare authentication protocol, it is useful for a wide variety of applications. It would probably be possible to create an implementation that was more tightly bound to a particular application (such as web serving) to improve performance and eliminate complexity. Such a server would be useful in situations where a pre-existing communication channel was open and needed to be authenticated. This server assumes the connections it establishes are the start of communication.

The server is optimized for applications where multiple clients will attempt authentication. This server does a good bit of work at startup to hide latencies during connections. This work is assumed to be amortized over many connections. Another design should be used if few connections are expected. Also, because we are designing for many connections, we can achieve near-C like speed using a compiling Java VM such as Hotspot. These VMs compile often executed code segments (hotspots) into native code on the fly as execution progresses. Such compilation is claimed to be performance competitive with statically compiled code. Unfortunately, the Hotspot VM was released too late to be tested with this implementation.

3.3.3 Client Implementation

The client module corresponds to *Alice* in Figure 2-1. Its main purpose is to authenticate to the server on behalf of a user program. This module would normally be run by subscribers or authorized individuals to gain access to content such as web pages. Unauthorized users can still run this module, but will be unable to authenticate successfully. After the client is authenticated, it returns a private, secure, and authenticated I/O stream to the user program.

Public Interface

The most important public methods are:

`Client(String principal, String passphrase)` This is the constructor for the `Client` class. This method constructs a client using typical values for the server's DNS information and algorithm parameters. These values (and others) can all be set after construction using one of the public setter methods. See the source code for all available options.

`static void main(String[] args)` The command line version of the client allows detailed debugging of all aspects of the client. When run in this mode the client will attempt to authenticate, then discard the private, secure, authenticated I/O stream immediately after authentication. This mode is useful for understanding the protocol in a hands-on manner.

`void authenticate()` The authenticate method tells the client to initiate an authentication session to the server. This method will block either until the client is successfully authenticated or until the server refuses authentication.

`CipherInputStream getCipherInputStream()` This method gets the encrypted input channel to the server, which can be used to receive private, authenticated data from the server after the authenticate method finished. This method will block until the authentication transaction has finished.

`CipherOutputStream getCipherOutputStream()` This method gets the encrypted output channel to the server, which can be used to send private, authenticated data to the server after the authenticate method finished. This method will block until the authentication transaction has finished.

Life Cycle

Typically, a client goes through the following stages:

1. A client is created.
2. *(optional)* Any non-default parameters are set with a setter method.
3. The `authenticate` method is called.
4. The private, secure, authenticated I/O streams are used to communicate with the server.
5. The streams are closed and the connection to the server ended.

Other Implementation Notes

Like the server module, the client module does little beyond the bare authentication protocol, which makes it useful for a wide variety of applications. In the previous section we noted the benefits a specialized server might have over a generic one. For the client, a generic client is more flexible, and specialized versions produced for a specific application (such as web browsing) will offer fewer benefits than a specialized server.

Unlike the server module, the client module is designed to perform a single authentication transaction. This explains the difference in how the client and server return the encrypted communication channel after authenticating. The server treats the input and output streams as a single connection and returns both in a single object. The client permits the user to get only the input or output channel, should they desire.

The client does less work at startup than the server, bit there is a significant one-time penality associated with the Crypto-J RSA libraries. On the server this cost is amortized over many connections, but here the startup cost can be a significant portion of the runtime. Because of this. future implementations need to be aware of the startup cost or to design clients that can perform multiple, independent authentication transactions.

3.4 Practical Implications

At the start of this chapter we asked the question, are verifibly common secret encodings as attractive in practice as they are in theory? This section gives some answers to that question using the lessions learned from the current implementation.

In Section 3.1 we identified three major goals for this implementation: Proof of Concept, Identify Problem Areas, and develop an Effective API. As to the first, we have seen the ideas developed in Chapter 2 can be implemented and do work in certain applications. While this implementation is not an endpoint for development, it does act as an effective guidepost for future implementations.

We have seen some problems that would need to be addressed in future implementations. Among the more serious concerns is the youth of cryptography libraries from vendors supporting the JCE. Crypto-J was the first to market with JCE1.2 compliance and public key cryptography, but has its own drawbacks. It is not yet ready for use in VCS style authentication transactions. The JCE itself lacks native support for certificates, which is particurally damaging to the key replacement requirement introduced in Section 2.2.

The final goal, development of an effective API, has been met on both the client and server side. The inclusion of add and remove methods to the VCS Vector API allows high performance, low latency server applications. The APIs described in Section 3.3 are generic, clear and simple, and should provide a starting point for other implementations. As noted above, the client and server modules developed here could be used for a variety of applications, so the need for another API might be eliminated.

Looking back to the larger question, what types of systems can perform VCS style authentications and for what applications? Such systems are most easily differentiated by the hardware the client will run on, since we can safely assume server hardware can scale to meet most reasonable loads. It will be the clients, not the servers, that limit the use of VCS. That said, when VCS authentication was originally concieved, we saw two major applications: web based access control using PCs and door/building access using smartcards. Here we add a third client platform, the palmtop computer, to the discussion.

This implementation was a PC based solution. VCS authentication can be used today on typical end-user hardware, but only if used in conjunction with subgroups. For group sizes less than approximately 50, the client software spent almost all of its time doing the initial startup associated with the RSA code. For groups over 50, informal testing showed a 5 second time increase per 100 users on a SparcStation5 80MHz, which would make subgroups of one or two hundred practical in many applications. Surprisingly, bandwidth is the most pressing concern. On local ethernet, the time to transmit the messages between client and server was approximately the same as the time to do the encryptions. This may be due to conversions taking place in the Java `net.io` package, but is still concerning. Users using a modem to access the web attempting VCS authentication will likely spend more time receiving the encrypted messages than performing encryptions. Still, VCS authentication could be a viable solution if we assume users will authenticate once per session with a server.

The palm computing niche is a significant step down from a PC, but still much better than a smartcard. [12] provides an introduction to the general problem of doing electronic commerce on these types of devices. Summarizing their relevant results, palm computers can be used to do electronic commerce in specially tailored environments, but palm computers are not yet capable of performing demanding public key operations in acceptable times[9]. Since VCS transactions make extensive use of public key operations, palm computers are not yet ready for VCS. However, Moore's Law applies to palm computers too, so they will be powerful enough to perform VCS authentication in the future.

Smartcard solutions are another story. A typical smartcard today has less than 4K of available RAM and severely limited processing and bandwidth capabilities. The lack of storage can be worked around using techniques like hash trees, but these techniques come at the cost of additional bandwidth requirements. Smartcards have neither the processing power nor bandwidth necessary to do VCS authentication, and seem unlikely to be powerful enough for the next 10-15 years. The dream of walking up to a door with a smartcard and performing an anonmyous authentication transaction will sadly remain a dream for a good long while.

[9]For instance, a single 512 but RSA key generation takes 3.4 minutes on average on a PalmPilot. A 512 bit RSA signature verify takes 1.4s.

3.5 Future Work

This implementation leaves plenty of room for future work. This section details some of the directions other implementations or derivative implementations may want to go.

Add Missing Pieces

Java's lack of a certificate mechanism lead the current implementation to use a modified protocol that does not fully meet the key replacement requirement. Future implementations should investigate products and libraries that provide for this important primitive. Alternatively, developing a library that can create standard X.509 certificates shouldn't prove too difficult a task. Once a certificate mechanism is in place, the seperation between $\mathbf{P}$ and $\mathbf{P}'$ should be implemented.

The current implementation does not implement the receipt mechanism described in Appendix A. Applications that would find this useful will want to provide this functionality.

Integration With Existing Products

The current implementation made no attempt to integrate with existing products such as Apache [33] or PGP [24]. However, the simple API should allow integration with these and other products.

Apache integration would be the first step towards a web browsing VCS application. An interesting project would be to create a module for Apache that performs the server side VCS authentication transactions. Part of this project would be to specify in detail the format of the various messages between client and server. If both these tasks were done, web browsing with VCS authentication would be much closer to reality.

Integration with PGP might be one way to get certificates into the VCS framework. PGP is an international standard and widely supported, so it makes sense to use PGP certificates in VCS. PGP also solves the difficult problem of making $\mathbf{P}$ public. PGP keys are easily obtained from public servers, and the web of trust model employed by PGP addresses the needs of individuals very well.

Chapter 4

Related Work

The balance between anonymity and security has been a motivating factor in the delevopment of cryptography for many years. Chaum's paper [6] is usually considered to be a seminal work on the subject. It that paper, Chaum assumes that institutions collect information about individuals who use those institutions' systems. He therefore proposes that individuals use different pseudonyms when conducting transactions with different institutions to prevent those institutions from sharing information and linking user profiles together. This fails to protect those whose right to use a system comes from a pre-existing relationship in which their identity is already known. Moreover, Chaum's approach does not provide unlinkability, leaving open the possibility an individual might reveal her identity through behaviors that can be profiled.

Syverson *et al.* [32] introduce a protocol for unlinkable serial transactions using Chaum's notion of blinding [8]. The protocol is designed for commercial pay-per-use services and relies upon the possibility that any particular service request may be forcibly audited. An audit requires the individual to reveal her identity or risk losing future service. After passing an audit, the individual must make another request before receiving the service originally requested. If requests are infrequent, she may have to wait a significant amount of time before making the second request lest the two requests become linked. This system does not provide adequate anonymity if the timing of any request indicates its nature, as audits can be made at any time. The system also cannot guarantee that a revoked individual does not receive service, as that individual may still make a request that is not audited.

Anonymous identification was first addressed as an application of witness hiding in zero knowledge proof systems [29, 10]. The most efficient such scheme, recently presented by De Santis *et al.* [28] in their paper on anonymous group identification, relies on the assumption that factoring Blum integers is hard[1]. While the extension of the protocol into a full system that supports key

[1]The De Santis *et al.* work was done independently at the same time as the original work on VCS [30].

replacement and dynamic groups is not explicitely addressed by the authors, such an extension is trivial.

For a group of n individuals and an m bit Blum integer, an instance of the De Santis *et al.* proof requires communication complexity $(2m+n)$, and rejects a non-member with probability $\frac{1}{2}$. Thus, to authenticate an individual's membership with certainty $1-\left(\frac{1}{2}\right)^d$, $(2m+n)\cdot d$ bits of communication are required. This would appear to approach a lower bound for such a zero knowledge proof system.

When implementing our current protocol using VCS vectors with k bit encryptions, identification requires $n \cdot m$ bits of communication. The security of the protocol relies on the existence of a public-key function that may securely encode the same plaintext in multiple messages with distinct keys. If the group size n exceeds $\frac{2md}{m-d}$, then the proof system of De Santis *et al.* requires less communication.

It is not clear that VCS vectors approach the lower bound for the size of a verifiably common secret encoding. A better encoding would require a change in cryptographic assumptions, but would have the potential of improving the efficiency of anonymous authentication protocols beyond that which is possible using zero knowledge proof systems.

Group signatures schemes [4, 7] give an individual the ability to anonymously sign messages on behalf of a group. Kilian and Petrank [21] exploit these signatures to create a scheme for identity escrow. Identity escrow provides anonymous authentication, though an individual's anonymity can be revoked by a trusted third party. While individuals may be added to the signature groups, no provision is made for removing members from these groups. Thus, group signatures in their current form are not a sufficient primitive for anonymously authenticating membership in dynamic groups.

Chapter 5

Conclusions

Anonymous authentication is an essential ingredient in a new domain of services in the field of electronic commerce and communication. Real world systems require dynamic group membership and key replacement.

In this paper we have shown how verifiably common secret encodings may be used to anonymously authenticate membership in dynamic groups. We have also shown how to replace keys in these authentication systems. We presented VCS vectors as an example of how verifiably common secret encodings can be constructed. Because the size of of our construct grows linearly with the size of the group $\mathbf{P}$, we described how to authenticate membership using subsets of $\mathbf{P}$.

The implementation described here provides client and server software that performs VCS authentication transactions. The software helped answer the question, "are verifibly common secret encodings as attractive in practice as they are in theory?" We saw that VCS authentication can be used when the client hardware is as powerful as a typical PC. In addition, the implementation explored questions about the current state of Java cryptography and produced an API that allows high-performance implementations.

Bibliography

[1] Anonymizer Inc. `http://www.anonymizer.com/`.

[2] Australian Business Access. `http://www.aba.net.au/solutions/crypto/jce.html`.

[3] Manuel Blum and Shafi Goldwasser. An *efficient* probabilistic public-key encryption scheme which hides all partial information. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology: Proceedings of CRYPTO 84*, volume 196 of *Lecture Notes in Computer Science*, pages 289–299. Springer-Verlag, 1985, 19–22 August 1984.

[4] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups (extended abstract). In Burton S. Kaliski Jr., editor, *Advances in Cryptology—CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 410–424. Springer-Verlag, 17–21 August 1997.

[5] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the Association for Computing Machinery*, 24(2):84–88, February 1981.

[6] D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, October 1985.

[7] D. Chaum and E. van Heyst. Group signatures. In Donald W. Davies, editor, *Proceedings of Advances in Cryptology (EUROCRYPT '91)*, volume 547 of *LNCS*, pages 257–265, Berlin, Germany, April 1991. Springer.

[8] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash (extended abstract). In S. Goldwasser, editor, *Advances in Cryptology—CRYPTO '88*, volume 403 of *Lecture Notes in Computer Science*, pages 319–327. Springer-Verlag, 1990, 21–25 August 1988.

[9] Don Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 10(4):233–260, Fall 1997.

[10] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Yvo G. Desmedt, editor, *Advances in Cryptology—CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer-Verlag, 21–25 August 1994.

[11] Cryptix Development Team. `http://www.cryptix.org/`.

[12] N. Daswani and D. Boneh. Experimenting with electronic commerce on the palmpilot. In *Financial Cryptography: Third International Conference, FC '99*, Lecture Notes in Computer Science, Anguilla, British West Indies, 22–25 February 1999. Springer-Verlag.

[13] Forge Research. `http://www.forge.com.au/`.

[14] S. Goldwasser and S. Macali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984. This paper introduces a new probabilistic encryption technique. It also contains an excellent introduction to other public key cryptosystems with discussion on objections to cryptosystems based on trapdoor functions.

[15] J. Hastad. Solving simultaneous modular equations of low degree. *SIAM Journal on Computing*, 17(2):336–341, apr 1988.

[16] Johan Hastad. On using RSA with low exponent in a public key network. In Hugh C. Williams, editor, *Advances in Cryptology—CRYPTO '85*, volume 218 of *Lecture Notes in Computer Science*, pages 403–408. Springer-Verlag, 1986, 18–22 August 1985.

[17] IAIK. `http://jcewww.iaik.tu-graz.ac.at/`.

[18] Java HotSpot(TM) Performance Engine. `http://java.sun.com/products/hotspot/`.

[19] Java(TM) Cryptography Extension. `http://java.sun.com/products/jce/index.html`.

[20] JCP Computer Services. `http://www.jcp.co.uk/secProduct/security_cdk_index.htm`.

[21] J. Kilian and E. Petrank. Identity escrow. In *Advances in Cryptology—CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 167–185, 1998.

[22] Jonathan Knudsen. *JAVA Cryptography.* The JAVA Series. O'Reilly & Associates, Inc., 101 Morris Street, Sebatopol, CA 95472, 1st edition, May 1998.

[23] National Institute of Standards and Technology (NIST). *FIPS Publication 186: Digital Signature Standard.* National Institute for Standards and Technology, Gaithersburg, MD, USA, May 1994.

[24] Network Associates. `http://www.pgpinternational.com/`.

[25] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998.

[26] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. Technical Report 97-15, DIMACS, April 15 1997. Thu, 16 Jun 1997 19:20:23 GMT.

[27] RSA Data Security, Inc. `http://www.rsa.com/rsa/products/cryptoj/index.html`.

[28] A. De Santis, G. Di Crescenzo, and G. Persiano. Communication-efficient anonymous group identification. In *5th ACM Conference on Computer and Communications Security*, pages 73–82, nov 1998.

[29] A. De Santis, G. Di Crescenzo, G. Persiano, and M. Yung. On monotone formula closure of SZK. In Shafi Goldwasser, editor, *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 454–465, Los Alamitos, CA, USA, November 1994. IEEE Computer Society Press.

[30] S. Schechter, T. Parnell, and A. Hartemink. Anonymous authentication of membership in dynamic groups. In *Financial Cryptography: Third International Conference, FC '99*, Lecture Notes in Computer Science, Anguilla, British West Indies, 22–25 February 1999. Springer-Verlag.

[31] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, Inc., New York, NY, USA, second edition, 1996.

[32] Paul F. Syverson, Stuart G. Stubblebine, and David M. Goldschlag. Unlinkable serial transactions. In Rafael Hirschfeld, editor, *Financial Cryptography: First International Conference, FC '97*, volume 1318 of *Lecture Notes in Computer Science*, pages 39–55, Anguilla, British West Indies, 24–28 February 1997. Springer-Verlag.

[33] The Apache HTTP Server Project. `http://www.apache.org/`.

[34] The Wall Street Journal Online. `http://www.wsj.com/`.

Appendix A

Obtaining Proof of Authentication

Alice may obtain a receipt from *Bob* proving that she was authenticated at time t. To obtain such a receipt, *Alice* chooses a random z and uses a one-way hash function h to generate $Q \leftarrow h\left(\{z\}_{\mathbf{s}_{alice}}\right)$ and $R \leftarrow h(z)$. *Alice* includes Q and R in message (3a):

Alice		**Bob**
(3a)	$\xrightarrow{E_y[x, Q, R]}$	

Bob can issue a receipt when he authenticates *Alice*. The receipt he sends is:

$$\{\text{“}Q \text{ and } R \text{ reveal whom I authenticated at time } t\text{”}\}_{\mathbf{s}_{bob}}$$

If she chooses, *Alice* can at any later time prove she was authenticated by *Bob* by revealing the receipt and the value $\{z\}_{\mathbf{s}_{alice}}$. Anyone can verify the receipt by checking that $Q = h(\{z\}_{\mathbf{s}_{alice}})$ and $R = h\left(\{\{z\}_{\mathbf{s}_{alice}}\}_{\mathbf{p}_{alice}}\right)$.

Appendix B

Source Code

The following pages contain the full source code for the implementation discussed in Chapter 3. The most important files are VCS.java, Server.java, Client.java, and VCSVector.java. VCS.java is the interface which VCSVector.java implements. The other files here include alternative implementations of the VCS interface (NativeVCS and InsecureVCS), wrapper classes (RSAEnvelope and BytesWrapper), and a classe that handles RSA key creation (RSAKeyTool).

B.1 VCS.java

```
/*
 * VCS.java
 */

package vcs;
import java.io.Serializable;
import java.security.KeyPair;
import java.util.Collection;

/**
 * This is the superclass for developing Verifibly Common Secrets
 * (VCS). A VCS is the primitive that makes anonymous authentication
 * of membership in dynamic groups possible.  All VCS implementations
 * implement this interface, typically by directly subclassing this
 * class.
 *
 * <p>The VCS class defines methods to encode, decode, and verify a
 * VCS.  Typically, a VCS goes throught the following stages:
 *
 * <ol>
 * <li> VCS is created by a call to <tt>VCS.getInstance</tt>.
 * <li> A secret is <b>encode</b>ed for a set of public keys .
 * <li> The VCS is transmitted (via Serialization or some other
 * mechanism) to be decoded.
 * <li> <i>(optional)</i> The VCS is <b>verify</b>ed.
 * <li> The secret is retreved by <b>decode</b>ing with a private key.
 * </ol>
 *
 * All general-purpose VCS implementation classes (classes which
 * directly or indirectly inherit from VCS) should provide a
 * "standard" void constructor which creates an empty VCS, ready to be
 * encoded.
 *
 * @author Todd C. Parnell, tparnell@ai.mit.edu
 * @version $Id: VCS.java,v 1.7 1999/04/16 23:35:30 tparnell Exp $
 */
public abstract class VCS implements Cloneable, Serializable {

  /**
   * Get a new VCS object of the specified type.  The VCS returned
   * will be ready for encoding.
   *
   * @param className The class name of VCS you wish to create.
   * @throws VCSException if the specified class cannot
   * be loaded and initilized
   */
  public static VCS getInstance(String className)
      throws VCSException {
    try {
      Class c = Class.forName(className);
      return  (VCS) c.newInstance();
    } catch (Exception e) {
      throw new VCSException(e.getMessage());
    }
  }

  /**
   * Get a new VCS of the default type.
   */
  public static VCS getInstance() {
    return new VCSVector();
  }

  /**
   * Encode the VCS for the given secret and public keys.  In general,
   * encoding a secret in a VCS ensures that the only way to extract
   * the secret is to have a private key corresponding to a public key
   * in the encoded set.  Note that it is entirely possible to encode
   * a VCS that the encoder cannot itself decode.
   *
   * @param secret The secret you wish to encode.
   * @param keys The public keys used to encode.
   * @throws VCSException if the secret cannot be encoded with the
   * given keys
   */
  public abstract void encode(byte[] secret, Collection keys)
    throws VCSException;

  /**
   * Decode the VCS to learn the secret it encodes.  This method will
   * only work if the supplied private key corresponds to a public key
   * used for encoding.
   *
   * @param pair The Keypair to use for decoding.
   * @param keys The public keys used to encode.
   * @return The secret this VCS encoded.
   * @throws VCSException if the VCS cannot be decoded with the
   * provided keypair
   */
  public abstract byte[] decode(KeyPair pair, Collection keys)
    throws VCSException;

  /**
   * Check the integrity of an encoding.  Upon receiving an encoded
   * VCS, the recipitant is not certain the encoding is for the group
   * of people he thinks it is for, or that the encoding is valid.
   * This method verifies both conditions.
   *
   * @param vcs  The VCS to verify.
   * @param pair The keys used to verify the VCS.
   * @param keys The public keys used to encode.
   * @throws VCSException if keys is not able to determine whether the
   * VCS is valid.
   * @return true iff the VCS is a valid encoding.
   */
  public static boolean verify(VCS vcs, KeyPair pair, Collection keys)
    throws VCSException {
   try {
     // create a temp VCS of the same type as vcs
     VCS tempVCS = (VCS) vcs.getClass().newInstance();
     // encode using the decoded secret & the keyset
     tempVCS.encode(vcs.decode(pair, keys), keys);
     // compare for equality
     return tempVCS.equals(vcs);
   } catch (Exception e) {
     throw new VCSException(e.getMessage());
   }
  }
}
```

B.2 Server.java

```
/*
 * Server.java
 */

package vcs;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.security.*;
import java.security.cert.*;
import java.security.spec.*;
import java.security.interfaces.*;
import java.io.*;
import java.net.*;
import vcs.util.*;
import java.util.*;

/**
 * The server side main program.  Servers accept connections from
 * Clients and estabilish a secure communication channel.  A single
 * server listens on a single port and can accept multiple clients.
 *
 * <p> Typically, a Server goes through the following stages:
 *
 * <ol>
 * <li> The Server is created with a given passphrase.
 * <li> <i>(optional)</i> Any non-default parameters are set with a
 * setter method.
 * <li> The <b>start</b> method is called.
 * <li> Connections are authenticated and handled via calls to
 *      <b>getConnection</b>.
 * <li> The <b>stop</b> method is called.
 * </ol>
 *
 * Note that the server will disallow further connections if exitsting
 * connections are not handled by an external controller.  To maintain
 * availability, any code that creates a Server must call
 * <b>getConnection</b> and close the input and output streams
 * contained therein.
 *
 * @author Todd C. Parnell, tparnell@ai.mit.edu
 * @version $Id: Server.java,v 1.19 1999/04/21 22:39:29 tparnell Exp $
 */
public class Server {

  //
  // Class Data & Methods
  //

  /** Size, in bytes, of random data in VCSs */
  private static String NONCE_SIZE = "16";
  /** Unique ID counter for connections */
  private static int COUNTER = 0;

  /** String to tell user how to use the program. */
  private static final String usageStr =
  "usage: vcs.Server [options]\n" +
  "    -h --help        : Print this message.\n" +
  "    -config file     : Use file to configure server.\n" +
  "    -v               : Operate verbosely.\n" +
  "    -principal name : Use name for authentication. (default: " +
                       Globals.SERVER_NAME + ")\n" +
  "    -noncesize n     : Use n byte nonces. (default: " +
                       Server.NONCE_SIZE + ")\n" +
  "    -d dir           : Use dir to find keys. (default: " +
                       Globals.PUB_KEY_DIR + ")\n" +
  "    -keypass pass    : Use pass to unlock private key.\n" +
  "    -port port       : Listen on port. (default: " +
                       Globals.SERVER_PORT + ")\n" +
  "    -vcsclass class : Use VCS of type class. (default: " +
                       Globals.DEFAULT_VCS_CLASS + ")\n" +
  "\nNote: ordering between -config and other flags is important";

  /**
   * Demo program to show server VCS functionality.  Extensive command
   * line arguments.  Run with -h to see options.  Prompts user for
   * missing input.
   */
  public static void main(String[] args) throws Exception {
    try {
      Security.addProvider(new com.sun.crypto.provider.SunJCE());
      Security.addProvider(new COM.rsa.jsafe.provider.JsafeJCE());
    } catch (Exception e) {
      System.err.println("Unable to add crypto providers.  Exiting.");
    }
    final Server me = new Server();
    Server.parseArgs(args, me);
    Server.getUserInput(me);
    me.start();
    // deal with closing the streams as they are created
    Runnable streamCloser = new Runnable() {
      public void run() {
        while (true) {
          Connection c = me.getConnection();
          try {
            c.getCipherInputStream().close();
            c.getCipherOutputStream().close();
          } catch (Exception e) {
          }
        } // while
      }
    };
    (new Thread(streamCloser)).start();
    // run the control program
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    while (true) {
      System.out.print("Select an option:\n1. Add a user.\n2. Remove a user\n3. Toggle Verbosity.\n4. Exit.\n
      String temp = br.readLine();
      if (temp.length() == 0) continue;
      char c = temp.charAt(0);
      if (c == '1') {
        System.out.print("Name: ");
        String name = br.readLine();
        me.addPrincipal(name);
      } else if (c == '2') {
        System.out.print("Name: ");
        String name = br.readLine();
        me.revokePrincipal(name);
      } else if (c == '3') {
        if (me.logStream == null) me.setLogStream(System.out);
        else me.setLogStream(null);
      } else if (c == '4') {
        System.exit(0);
      }

    } // while
  } // main

  /**
   * Set up server with info from command line.
   * Moved to seperate method since it's messy and boring.
   *
   * @param args command line arguments
   * @param server the server to set up
   */
  private static void parseArgs(String[] args, Server server) {
    for (int i = 0; i < args.length; ++i) {
      String option = args[i];
      if (option.equals("-port")) {
```

```java
        try {
          server.props.setProperty("SERVER_PORT", args[++i]);
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.err.println("Must provide size when specifing keysize.  Exiting.");
          System.exit(0);
        }
      } else if (option.equals("-v")) {
        server.setLogStream(System.out);
      } else if (option.equals("-h") || option.equals("--help")) {
        System.out.println(Server.usageStr);
        System.exit(0);
      } else if (option.equals("-noncesize")) {
        try {
          server.props.setProperty("noncesize", args[++i]);
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.err.println("Must provide size when specifing noncesize.  Exiting.");
          System.exit(0);
        }
      } else if (option.equals("-principal")) {
        try {
          server.props.setProperty("SERVER_NAME", args[++i]);
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.err.println("Must provide name when specifing principal.  Exiting.");
          System.exit(0);
        }
      } else if (option.equals("-d")) {
        try {
          server.props.setProperty("PUB_KEY_DIR", args[++i]);
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.err.println("Must provide dir when specifing -d.  Exiting.");
          System.exit(0);
        }
      } else if (option.equals("-keypass")) {
        try {
          server.props.setProperty("keypass", args[++i]);
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.err.println("Must provide pass when specifing keypass.  Exiting.");
          System.exit(0);
        }
      } else if (option.equals("-config")) {
        try {
          File f = new File(args[++i]);
          FileInputStream fis = new FileInputStream(f);
          server.props.load(fis);
          fis.close();
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.err.println("Must provide file when specifing config.  Exiting.");
          System.exit(0);
        } catch (IOException ioe) {
          System.err.println("An error occured while loading config file.  Exiting.");
          System.exit(0);
        }
      } else if (option.equals("-vcsclass")) {
        try {
          server.props.setProperty("DEFAULT_VCS_CLASS", args[++i]);
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.err.println("Must provide class when specifing -vcsclass.  Exiting.");
          System.exit(0);
        }
      } else /* error */ {
        System.err.println("Unknown option: '" + option + "'.  Exiting.");
        System.err.println(Server.usageStr);
        System.exit(0);
      }
    }
  } // parseArgs

  /**
   * Prompts user for passphrase, if not provided on command line.
   *
   * @param server Server to get passphrase for
   */
  private static void getUserInput(Server server) {
    while (server.props.getProperty("keypass") == null ||
           server.props.getProperty("keypass").equals("")) {
      System.out.print("Enter passphrase to unlock private key: ");
      try {
        BufferedReader br =
          new BufferedReader(new InputStreamReader(System.in));
        server.props.setProperty("keypass", br.readLine());
      } catch (IOException ioe) {
        System.err.println("IOException reading passphrase!  Exiting.");
        System.exit(0);
      }
    }
  }

  //
  // Instance fields & methods
  //

  /** Where we send our logging output to */
  private PrintWriter logStream = null;
  /** Storage for all the switches */
  private Properties props = null;
  /** Manager for all current sessions */
  private ConnectionManager manager;
  /** Object to watch for connections from clients */
  private Listener listener;
  /** Thread to create VCSs in background */
  private VCSGenerator vcsGen;
  /** Shared random number generator */
  private SecureRandom sr;
  /** ThreadGroup for all subthreads to share */
  private ThreadGroup tg = new ThreadGroup("Server Thread Group");
  /** Collection of all authenticated connections */
  private LinkedList connections = new LinkedList();
  /** Set of all principals in authorization group */
  private HashMap principals = new HashMap();

  /**
   * Construct a new server.
   *
   * @param passphrase passphrase assocated with this server's private key
   */
  public Server(String passphrase) {
    this(passphrase, true);
  }

  /**
   * Private constructor to allow Server.main to get passphrase
   * from command line.
   */
  private Server() {
    this(null, false);
  }

  /**
   * Constructor that actually does the work.
   *
   * @param passphrase passphrase associated with this server's private key
   * @param checkNull if true, complain if passphrase is null
   */
  private Server(String passphrase, boolean checkNull) {
    if (checkNull && (passphrase == null))
      throw new NullPointerException();

    this.props = new Properties(Globals.DEFAULT_PROPERTIES);
    if (passphrase != null) {
      this.props.setProperty("keypass", passphrase);
    }
    this.props.setProperty("noncesize", Server.NONCE_SIZE);

    // Set up the manager now.  It will begin a new thread, then
```

```
    // block until a listener wakes it.  Don't create the listener
    // here, do that in start().
    this.manager = new ConnectionManager();
  }

  /**
   * Begin listening on the port for connections.  Before calling this
   * method, be certain to fully configure the Server.
   */
  public synchronized void start() throws IOException, VCSException {
    this.log("Intilizing keys for authorized users.");
    try {
      this.principals =
        RSAKeyTool.getAllFromDir(Server.this.props.getProperty("PUB_KEY_DIR"), true);
    } catch (Exception e) {
      throw new VCSException(e.getMessage());
    }

    if (this.listener == null)
      this.listener =
        new Listener(Integer.parseInt(this.props.getProperty("SERVER_PORT")));
    if (this.vcsGen == null)
      this.vcsGen =
        new VCSGenerator(this.props.getProperty("DEFAULT_VCS_CLASS"),
                         Integer.parseInt(this.props.getProperty("noncesize")));
  }

  /**
   * Stop providing service to clients.
   */
  public synchronized void stop() {
    this.listener.stop();
    this.listener = null;
    this.vcsGen.pleaseStop();
    this.vcsGen.flush();
  }


  /**
   * Add the provided connection to the list of available connections.
   * Called by Server.Connection.run() when authenticated.
   */
  private synchronized void addAuthenticatedConnection(Connection c) {
    this.connections.addFirst(c);
  }

  /**
   * Get the oldest authenticated connection from the server.  This
   * method blocks until a client is available.  The callee of this
   * method is responsible for closing both streams of the connection.
   */
  public Connection getConnection() {
    while (this.connections.size() == 0) {
      try { Thread.sleep(2000); }
      catch (InterruptedException e) {}
    }
    synchronized (this) {
      return (Connection) this.connections.removeLast();
    }
  }

  /**
   * Adds principal to the authorization group.  Further VCSs
   * generated by this server will allow princical to authenticate.
   * Note that principal's key must exist in the current key directory.
   * If principal is already authorized, this method has no effect.
   *
   * @param principal the newly authorized user
   */
  public synchronized void addPrincipal(String principal) {
    RSAPublicKey key;
    String f = Server.this.props.getProperty("PUB_KEY_DIR") + File.separator + principal + ".pub";
```

```
    try {
      key = RSAKeyTool.stringToPubKey(f);
    } catch (Exception e) {
      this.log(e);
      return;
    }
    this.principals.put(f, key);
    this.vcsGen.addPrincipal(key);
    this.writeDefaultFile();
  }

  /**
   * Remove principal's ability to authenticate.  Further VCSs
   * generated by this server will not include include principal's
   * public key.  If principal is not an authorized user, this
   * method has no effect.
   *
   * @param principal the revoked user
   */
  public synchronized void revokePrincipal(String principal) {
    String f = Server.this.props.getProperty("PUB_KEY_DIR") + File.separator + principal + ".pub";
    RSAPublicKey key = (RSAPublicKey) this.principals.get(f);
    this.principals.remove(f);
    this.vcsGen.revokePrincipal(key);
    this.writeDefaultFile();
  }

  /**
   * Write the current set of authorized users to the VCS.defaults
   * file.
   */
  private void writeDefaultFile() {
    try {
      File f = new File(Server.this.props.getProperty("PUB_KEY_DIR") + File.separator + "VCS.defaults");
      FileOutputStream fos = new FileOutputStream(f);
      BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fos));
      Iterator it = this.principals.keySet().iterator();
      while (it.hasNext()) {
        String line = (String) it.next();
        int start = line.lastIndexOf(File.separator);
        int end   = line.lastIndexOf(".pub");
        bw.write(line.substring(start+1, end).toCharArray());
        bw.newLine();
      }
      bw.flush();
      bw.close();
    } catch (Exception e) {
      this.log(e);
    }
  }

  /**
   * Set the port to listen on.
   *
   * @param port port to listen on
   */
  public void setPort(int port) {
    if (port < 1)
      throw new IllegalArgumentException("port must be greater than 0");

    this.props.setProperty("SERVER_PORT", (new Integer(port)).toString());
  }

  /**
   * Sets the directory to look for keys in.
   *
   * @param dir the directory to look in for keys
   */
  public void setKeyDir(String dir) {
    this.props.setProperty("PUB_KEY_DIR", dir);
  }
```

```java
/**
 * Sets the type of VCS to encode.
 *
 * @param vcsClass the class of VCS to encode
 */
public void setVCSClass(String vcsClass) {
  this.props.setProperty("DEFAULT_VCS_CLASS", vcsClass);
}

/**
 * Set the current logging stream.  Pass null to turn logging off.
 *
 * @param stream the new stream to log to, or null to end logging
 */
public void setLogStream(OutputStream out) {
  if (out != null) this.logStream =
                  new PrintWriter(new OutputStreamWriter(out));
  else this.logStream = null;
}

/**
 * Write the specified string to the log
 *
 * @param s the string to write
 */
private synchronized void log(String s) {
  if (logStream != null) {
    logStream.println("[" + new Date() + "] " + s);
    logStream.flush();
  }
}

/**
 * Write the specified object to the log
 *
 * @param o the object to write
 */
private void log(Object o) { this.log(o.toString()); }

/**
 * Utility class to monitor a port and report back to the
 * ConnectionManager with all new Sockets.
 */
private class Listener implements Runnable {
  /** Socket we're listening on. */
  private ServerSocket socket;
  /** Flag to indicate we should exit run method. */
  private boolean stopped;
  /** The worker thread that animates us. */
  private Thread spirit;

  /** Private server key */
  RSAPrivateKey privateKey;
  /** Source of randomness */
  SecureRandom sr;

  /**
   * Construct a new Listener.  Reads and saves all current Server
   * configuration parameters.
   *
   * @param port the port to listen on
   * @throws IOException if the socket cannot be bound
   */
  public Listener(int port) throws IOException, VCSException  {
    Server.this.log("Reading private key.");
    try {
      // grab the encrypted private key
      this.privateKey = RSAKeyTool.stringToPriKey (
        Server.this.props.getProperty("PUB_KEY_DIR") + File.separator +
        Server.this.props.getProperty("SERVER_NAME") + ".pri",
        Server.this.props.getProperty("keypass").toCharArray()
                                                    );
    } catch (Exception e) {
      throw new IOException("Unable to load key from disk.");
    }

    Server.this.log("Initilizing PRNG");
    try {
      this.sr = SecureRandom.getInstance("SHA1PRNG");
      /**
       * HACK: we should ask the user for input, to get better
       * randomness.  However, let's use the system clock for demo
       * purposes.
       */
      this.sr.setSeed((new Long(System.currentTimeMillis())).toString().getBytes());

    // the "right" way
//      System.out.println("Enter some keystrokes to seed the random number generator.");
//      System.out.println("Press return after a line or two of text.");
//      BufferedReader br =
//        new BufferedReader(new InputStreamReader(System.in));
//      String temp = br.readLine();
//      sr.setSeed(temp.getBytes());
    } catch (Exception e) {
      throw new VCSException("Error initilizing random number generator");
    }

    this.socket = new ServerSocket(port);
    // give non-zero timeout, to support interruption
    this.socket.setSoTimeout(600000);
    this.spirit = new Thread(Server.this.tg, this, "Listener: " + port);
    this.spirit.start();
  }

  /**
   * Stop listening on the port.
   */
  public void stop() {
    this.stopped = true;
    this.spirit.interrupt();
  }

  /**
   * Body for the animating thread.  Waits for connections, accepts
   * them, and passes the socket back to the main program.
   */
  public void run() {
    while ( ! this.stopped ) {
      try {
        Socket client = socket.accept();
        Server.this.manager.addConnection(client);
      } catch (InterruptedIOException e) {
        // do nothing
      } catch (IOException e) {
        Server.this.log(e);
      }
    } // while
  } // run
} // class Listener

/**
 * Manages the list of all current sessions.
 */
private class ConnectionManager extends Thread {
  /** Current list of connections. */
  private ArrayList connections;

  /**
   * Create a ConnectionManager.
   */
  public ConnectionManager() {
    super(Server.this.tg, "ConnectionManager");
    this.connections = new ArrayList();
    this.setDaemon(true);
```

```
      Server.this.log("Starting connection manager.");
      this.start();
    }

    /**
     * Listener objects call this method when they accept a new
     * connection.  Here we simply note the new connection and start a
     * session.
     *
     * @param s the socket the client is connected to
     */
    synchronized void addConnection(Socket s) {
      // Create Connection thread to handle it
      Connection c = new Connection(s);
      connections.add(c);
      Server.this.log("(" + c.count + ") " +
                      "Connected to " + s.getInetAddress().getHostAddress() +
                      ":" + s.getPort() + " on port " + s.getLocalPort());
      c.start();
    }

    /**
     * A Connection calls thes method just before it exits.
     */
    public synchronized void endConnection() { this.notify(); }

    /**
     * Keep the list of connections up to date by removing connections
     * that are no longer alive.
     */
    public void run() {
      while (true) {
        for (int i=0; i < this.connections.size(); ++i) {
          Connection c = (Connection) this.connections.get(i);
          if ( !c.isAlive() ) {
            this.connections.remove(i);
            Server.this.log("(" + c.count + ") " + "Authentication of " +
                            c.client.getInetAddress().getHostAddress() +
                            ":" + c.client.getPort() + " finished.");
          }
        }
        try { synchronized (this) { this.wait(); }
        } catch (InterruptedException ie) {
        }
      }
    } // run
  } // class ConnectionManager

  /**
   * Represents a single authentication session to one client.
   */
  public class Connection implements Runnable {

    /** Socket to talk to the client over. */
    private Socket client;
    /** Animating thread. */
    private Thread spirit;
    /** Connection number */
    private int count = Server.COUNTER++;
    /** Indicator for when the run method has exited */
    private boolean runDone = false;
    /** Secure outgoing stream */
    private CipherOutputStream cipherOut;
    /** Secure incoming stream */
    private CipherInputStream cipherIn;

    /**
     * Create a new connection.  Does not start the new thread.
     */
    private Connection(Socket s) {
      this.client = s;
      this.spirit =
        new Thread(Server.this.tg,
                   this, "Server.Connection(" + this.count + "):" +
                   client.getInetAddress().getHostAddress() +
                   ":" + client.getPort());
    }

    /**
     * Begin the thread.
     */
    private void start() {
      this.spirit.start();
    }

    /**
     * is the Thread alive?
     */
    private boolean isAlive() {
      return this.spirit.isAlive();
    }

    /**
     * Run a single instance of the protocol.
     */
    public void run() {

      try {
        // begin by using encrypted objects over an insecure stream
        ObjectInputStream in =
          new ObjectInputStream(this.client.getInputStream());
        ObjectOutputStream out =
          new ObjectOutputStream(this.client.getOutputStream());;

        RSAEnvelope envelope = (RSAEnvelope) in.readObject();

        this.log("Decrypting session key.");
        SecretKey sessionKey =
          (SecretKey) envelope.open(Server.this.listener.privateKey);

        this.log("Getting a new VCS");
        VCSGenerator.Entry entry = Server.this.vcsGen.getEntry();
        VCS vcs = entry.vcs;
        byte[] secret = entry.secret;
        this.log("The secret is " + Base64.encode(secret));
        // set up the encrypt/decrypt ciphers
        Cipher decryptCipher = Cipher.getInstance(Globals.SYMMETRIC_ALG);
        decryptCipher.init(Cipher.DECRYPT_MODE, sessionKey,
                           Server.this.listener.sr);
        Cipher encryptCipher = Cipher.getInstance(Globals.SYMMETRIC_ALG);
        encryptCipher.init(Cipher.ENCRYPT_MODE, sessionKey,
                           Server.this.listener.sr);
        // send the VCS over the private line
        out.writeObject(new SealedObject(vcs, encryptCipher));

        // wait for authentication
        this.log("Waiting for reply...");
        String userString =
          (String) ((SealedObject)in.readObject()).getObject(sessionKey);

        // the end result!
        if (userString.equals(Base64.encode(secret))) {
          this.log("User Successfully Authenticated");
        } else {
          this.log("ERROR -- User Not Authinticated");
        }

        // set up the CipherStreams
        this.cipherIn =
          new CipherInputStream(this.client.getInputStream(), decryptCipher);
        this.cipherOut =
          new CipherOutputStream(this.client.getOutputStream(), encryptCipher);

        // tell the Server about the new authenticated connection
```

```
      Server.this.addAuthenticatedConnection(this);
    } catch (Exception e) {
      this.log(e);
    } finally {
      this.runDone = true;
      Server.this.manager.endConnection();
    }
  } // run

  /**
   * Get the encrypted input channel to the authenticated client.
   * Will block until the authentication transaction has finished.
   * <i>Warning: do not attempt to wrap an ObjectInputStream around
   * the returned CipherInputStream.  There appears to be a bug in the
   * 1.2.1 JDK implementation.</i>
   *
   * @throws VCSException if the client did not successfully
   * authenticate
   */
  public CipherInputStream getCipherInputStream()
       throws VCSException {
    while ( !this.runDone ) {
      try { Thread.sleep(5000); }
      catch (InterruptedException e) {}
    }
    if (this.cipherIn == null) {
      throw new VCSException("Client did not successfully authenticate");
    } else return this.cipherIn;
  }

  /**
   * Get the encrypted output channel to the authenticated client.
   * Will block until the authentication transaction has finished.
   * <i>Warning: do not attempt to wrap an ObjectOutputStream around
   * the returned CipherOutputStream.  There appears to be a bug in the
   * 1.2.1 JDK implementation.</i>
   *
   * @throws VCSException if the client did not successfully
   * authenticate
   */
  public CipherOutputStream getCipherOutputStream()
       throws VCSException {
    while ( !this.runDone ) {
      try { Thread.sleep(5000); }
      catch (InterruptedException e) {}
    }
    if (this.cipherOut == null) {
      throw new VCSException("Client did not successfully authenticate");
    } else return this.cipherOut;
  }

  /** Perform logging */
  private void log(Object o) {
    Server.this.log("(" + this.count + ") " + o);
  }

  /*
   * Because we're relying on users of the class to close the
   * streams, it's best to try to help them out where we can.  This
   * doesn't eliminate their need, but doesn't hurt.
   */
  protected void finalize() throws IOException {
    if (this.cipherOut != null) cipherOut.close();
    if (this.cipherIn  != null) cipherIn.close();
  }

} // class Connection

/**
 * This class generates VCSs in a seperate thread, so that we can
 * service client requests as quickly as possible.
 */
```

```
private class VCSGenerator extends Thread {
  /** Flag to tell us to stop */
  private boolean stopped;
  /** Storage of all the VCSs */
  private LinkedList store;
  /** Maximum number of VCSs to store */
  public int maxStored = 10;
  /** Type of VCSs to create */
  private String type;
  /** Nonce size to encode */
  private int size;

  /**
   * Construct a new VCSGenerator and set it to work.
   *
   * @param type the fully qualified type of VCS to create
   * @param size nonce size to encode
   */
  public VCSGenerator(String type, int size) throws VCSException {
    super(Server.this.tg, "Generator");
    // figure out if type is valid
    VCS test = VCS.getInstance(type); // throws exception if invalid
    this.type = type;
    this.size = size;
    this.store = new LinkedList();
    this.start();
  }

  /**
   * Body of thread execution.  Creates a bunch of VCSs.
   */
  public void run() {
    while ( ! this.stopped ) {
      if (this.store.size() < this.maxStored) {
        try {
          VCS temp = VCS.getInstance(this.type);
          byte[] secret = new byte[this.size];
          Server.this.listener.sr.nextBytes(secret);
          temp.encode(secret, Server.this.principals.values());
          /*
           * This may be a minor synchronization problem, since a
           * flush could have occured after we generated a VCS but
           * before we store it.  However, in the name of
           * performance I have chosen this option.
           */
          synchronized (this) {
            this.store.addFirst(new Entry(secret, temp));
          }
        } catch (VCSException e) {
          Server.this.log(e);
        }
      } else {
        Server.this.log("VCS cache filled.");
        try { synchronized (this) { this.wait(); }
        } catch (InterruptedException ie) {
        }
      }
    }
  } // run

  /**
   * Get one of the generated Entries.
   */
  public synchronized Entry getEntry() {
    while (this.store.size() == 0) {
      // wait a bit
      try { Thread.sleep(2000); }
      catch (InterruptedException ie) {}
    }
    Entry e = (Entry) this.store.removeLast();
    this.notify(); // create some more
    return e;
```

```
    }

    /** The polite way to stop */
    public void pleaseStop() {
      this.stopped = true;
      this.notify();
    }

    /**
     * Discard all currently cached VCS entries.
     *
     * @return the number of entires removed
     */
    public synchronized int flush() {
      int size = this.store.size();
      this.store.clear();
      this.notify();
      return size;
    }

    /**
     * Adds a new principal to all currently queued VCSs.  The new
     * principal will be able to authenticate for all subsequent VCSs.
     *
     * @param key new principal to authorize
     */
    public synchronized void addPrincipal(RSAPublicKey key) {
      try {
        Iterator it = this.store.iterator();
        while (it.hasNext()) {
          Entry e = (Entry) it.next();
          if (e.vcs instanceof VCSVector)
            ((VCSVector)e.vcs).addPrincipal(e.secret, key);
        }
      } catch (VCSException e) {
        // something's amiss...
        Server.this.log(e);
      }
    }

    /**
     * Remove a principal to all currently queued VCSs.  The new
     * principal will be unable to authenticate for all subsequent VCSs.
     *
     * @param key principal to revoke
     */
    public synchronized void revokePrincipal(RSAPublicKey key) {
      try {
        Iterator it = this.store.iterator();
        while (it.hasNext()) {
          Entry e = (Entry) it.next();
          if (e.vcs instanceof VCSVector)
            ((VCSVector)e.vcs).revokePrincipal(e.secret, key);
        }
      } catch (VCSException e) {
        Server.this.log(e);
      }
    }

    /** Storage for secret, vcs pairs */
    public class Entry {
      public final byte[] secret;
      public final VCS vcs;
      public Entry(byte[] secret, VCS vcs) {
        this.secret = secret;
        this.vcs = vcs;
      }
    }

  } // VCSGenerator

} // class Server
```

B.3 Client.java

```java
/*
 * Client.java
 */

package vcs;
import java.math.BigInteger;
import java.util.HashSet;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.security.*;
import java.security.cert.*;
import java.security.spec.*;
import java.security.interfaces.*;
import java.io.*;
import java.net.*;
import java.util.*;
import vcs.util.Base64;

/**
 * The client side main program.  Clients authenticate to a
 * Server and establish a secure communication channel.
 * A single client must be created for every connection needed.
 *
 * <p> Typically, a Client goes through the following stages:
 *
 * <ol>
 * <li> The Client is created for a given principal with a passphrase.
 * <li> <i>(optional)<i> Any not-default parameters are set with a
 * setter method.
 * <li> The <b>authenticate</b> method is called.
 * <li> The authenticated, encrypted streams are used to communicate
 * with the server.
 * <li> The streams are closed.
 * </ol>
 *
 * @author Todd C. Parnell, tparnell@ai.mit.edu
 * @version $Id: Client.java,v 1.19 1999/04/21 15:34:54 tparnell Exp $
 */
public class Client {

  //
  // Class Data & Methods
  //

  /** String to tell user how to use the program. */
  private static final String usageStr =
  "usage: vcs.Client [options]\n" +
  "   -h --help       : Print this message.\n" +
  "   -config file    : Use file to configure client.\n" +
  "   -v              : Operate verbosely.\n" +
  "   -principal name : Use name for authentication.\n" +
  "   -alg alg        : Use alg as the symmetric algorithm (default: " +
                        Globals.SYMMETRIC_ALG + ")\n" +
  "   -keysize n      : Use n bit session key (default: " +
                        Globals.SESSION_KEY_SIZE + ")\n" +
  "   -d dir          : Use dir to find keys (default: " +
                        Globals.PUB_KEY_DIR + ")\n" +
  "   -keypass pass   : Use pass to unlock private key.\n" +
  "   -srvname name   : Server hostname (default: " +
                        Globals.SERVER_NAME + ")\n" +
  "   -srvport port   : Server port (default: " +
                        Globals.SERVER_PORT + ")\n" +
  "\nNote: ordering between -config and other flags is important";

  /**
   * Demo program to show client VCS functionality.  Extensive command
   * line arguments.  Run with -h to see options.  Prompts user for
   * missing input.
   */
  public static void main(String[] args) throws VCSException {
    try {
      Security.addProvider(new com.sun.crypto.provider.SunJCE());
      Security.addProvider(new COM.rsa.jsafe.provider.JsafeJCE());
    } catch (Exception e) {
      System.err.println("Unable to add crypto providers.  Exiting.");
    }
    Client client = new Client();
    Client.parseArgs(args, client);
    Client.getUserInput(client);
    client.authenticate();
    try {
      client.getCipherInputStream().close();
      client.getCipherOutputStream().close();
    } catch (IOException e) {
    }
  }

  /**
   * Set up client with info from command line arguments.
   * Moved to seperate method since it's messy and boring.
   *
   * @param args command line arguments
   * @param client the client to set up
   */
  private static void parseArgs(String[] args, Client client) {
    for (int i = 0; i < args.length; ++i) {
      String option = args[i];
      if (option.equals("-keysize")) {
        try {
          client.props.setProperty("SESSION_KEY_SIZE", args[++i]);
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.err.println("Must provide size when specifing keysize.  Exiting.");
          System.exit(0);
        }
      } else if (option.equals("-principal")) {
        try {
          client.props.setProperty("principal", args[++i]);
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.err.println("Must provide name with -principal.  Exiting.");
          System.exit(0);
        }
      } else if (option.equals("-d")) {
        try {
          client.props.setProperty("PUB_KEY_DIR", args[++i]);
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.err.println("Must provide dir with -d.  Exiting.");
          System.exit(0);
        }
      } else if (option.equals("-keypass")) {
        try {
          client.props.setProperty("keypass", args[++i]);
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.out.println("Must give passphrase when specifing keypass.  Exiting.");
          System.exit(0);
        }
      } else if (option.equals("-v")) {
        client.setLogStream(System.out);
      } else if (option.equals("-h") || option.equals("--help")) {
        System.out.println(Client.usageStr);
        System.exit(0);
      } else if (option.equals("-srvport")) {
        try {
          client.props.setProperty("SERVER_PORT", args[++i]);
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.err.println("Must provide port when specifing srvport option.  Exiting.");
          System.exit(0);
        }
      } else if (option.equals("-srvname")) {
```

```
        try {
          client.props.setProperty("SERVER_NAME", args[++i]);
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.err.println("Must provide host when specifing srvname option.  Exiting.");
          System.exit(0);
        }
      } else if (option.equals("-alg")) {
        try {
          client.props.setProperty("SYMMETRIC_ALG", args[++i]);
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.err.println("Must provide alg when specifing alg option.  Exiting.");
          System.exit(0);
        }
      } else if (option.equals("-config")) {
        try {
          File f = new File(args[++i]);
          FileInputStream fis = new FileInputStream(f);
          client.props.load(fis);
          fis.close();
        } catch (ArrayIndexOutOfBoundsException obe) {
          System.err.println("Must provide file when specifing config.  Exiting.");
          System.exit(0);
        } catch (IOException ioe) {
          System.err.println("An error occured while loading config file.  Exiting.");
          System.exit(0);
        }
      } else /* error */ {
        System.err.println("Unknown option: '" + option + "'.  Exiting.");
        System.err.println(Client.usageStr);
        System.exit(0);
      }
    }
  } // parseArgs

  /**
   * Prompts user for principal & passphrase, if not provided on
   * command line.
   *
   * @param client Client to populate
   */
  private static void getUserInput(Client client) {
    while (client.props.getProperty("principal") == null ||
           client.props.getProperty("principal").equals("")) {
      System.out.print("Enter principal: ");
      try {
        BufferedReader br =
          new BufferedReader(new InputStreamReader(System.in));
        client.props.setProperty("principal", br.readLine());
      } catch (IOException ioe) {
        System.err.println("IOException reading passphrase!  Exiting.");
        System.exit(0);
      }
    }

    while ( client.props.getProperty("keypass") == null ||
            client.props.getProperty("keypass").equals("") ) {
      System.out.print("Enter passphrase to unlock private key: ");
      try {
        BufferedReader br =
          new BufferedReader(new InputStreamReader(System.in));
        client.props.setProperty("keypass", br.readLine());
      } catch (IOException ioe) {
        System.err.println("IOException reading passphrase!  Exiting.");
        System.exit(0);
      }
    }
  } // getUserInput


  //
  // Instance Data & Methods
  //

  /** Where we send our logging output to */
  private PrintWriter logStream = null;
  /** Storage for all the switches */
  private Properties props = null;
  /** Secure incoming stream */
  private CipherInputStream cipherIn;
  /** Secure outgoing stream */
  private CipherOutputStream cipherOut;
  /** Flag to indicate the authentication is done */
  private boolean runDone = false;

  /**
   * Construct a new client, ready to authenticate to the default
   * server.
   *
   * @param principal name to authenticate as (use principal's public key)
   * @param passphrase passphrase associated with principal
   */
  public Client(String principal, String passphrase) {
    this(principal, passphrase, true);
  }

  /**
   * Private construtor to allow Client.main to get principal and
   * passphrase after construction.
   */
  private Client() {
    this(null, null, false);
  }

  /**
   * Constructor that actually does the work.
   *
   * @param principal name to authenticate as (use principal's public key)
   * @param passphrase passphrase associated with principal
   * @param checkNulls if true, complain if principal or passphrase in null
   */
  private Client(String principal, String passphrase, boolean checkNulls) {
    if (checkNulls && (principal == null || passphrase == null))
      throw new NullPointerException();

    this.props = new Properties(Globals.DEFAULT_PROPERTIES);
    if (principal != null)
      this.props.setProperty("principal", principal);
    if (passphrase != null)
      this.props.setProperty("keypass", passphrase);
  } // constructor

  /**
   * Run the VCS authentication protocol.
   *
   * @throws VCSException if any problems occur
   */
  public void authenticate() throws VCSException {

    this.log("Getting keys from disk.");
    RSAPublicKey srvPubKey = null;
    KeyPair myKeys = null;

    try {
      // grab the encoded server key from disk
      srvPubKey =
        RSAKeyTool.stringToPubKey(this.props.getProperty("PUB_KEY_DIR") +
                                  File.separator +
                                  this.props.getProperty("SERVER_NAME") +
                                  ".pub");

      // grab my keys
      RSAPrivateKey myPriKey =
        RSAKeyTool.stringToPriKey(this.props.getProperty("PUB_KEY_DIR") +
                                  File.separator +
                                  this.props.getProperty("principal") +
```

```
                              ".pri",
                              this.props.getProperty("keypass").toCharArray());
      RSAPublicKey myPubKey =
        RSAKeyTool.stringToPubKey(this.props.getProperty("PUB_KEY_DIR") +
                              File.separator +
                              this.props.getProperty("principal") +
                              ".pub");
      myKeys = new KeyPair(myPubKey, myPriKey);
    } catch (IOException ioe) {
      throw new VCSException("Could not load keys from disk.  Check path and try again.");
    } catch (Exception e) {
      // crypto related problem
      throw new VCSException("Error loading keys from disk.  " +
                         "Check passphrase and principal.");
    }

    this.log("Initilizing PRNG");
    SecureRandom sr = null;

    try {
      sr = SecureRandom.getInstance("SHA1PRNG");
      /**
       * HACK: we should ask the user for input, to get better
       * randomness.  However, let's use the system clock for demo
       * purposes.
       */
      sr.setSeed((new Long(System.currentTimeMillis())).toString().getBytes());

      // the "right" way
//      System.out.println("Enter some keystrokes to seed the random number generator.");
//      System.out.println("Press return after a line or two of text.");
//      BufferedReader br =
//        new BufferedReader(new InputStreamReader(System.in));
//      String temp = br.readLine();
//      sr.setSeed(temp.getBytes());
    } catch (Exception e) {
      throw new VCSException("Error initilizing random number generator");
    }

    this.log("Generating session key.");
    SecretKey sessionKey = null;

    try {
      KeyGenerator sessionGen =
        KeyGenerator.getInstance(this.props.getProperty("SYMMETRIC_ALG"));
      sessionGen.init(Integer.parseInt(this.props.getProperty("SESSION_KEY_SIZE")), sr);
      sessionKey = sessionGen.generateKey();
    } catch (Exception e) {
      throw new VCSException("Error creating session key.  Verify symmetric algorithm and keyize.");
    }

    this.log("Encrypting session key.");
    RSAEnvelope msg1 = null;

    try {
      msg1 = new RSAEnvelope(sessionKey, sr, srvPubKey);
    } catch (Exception e) {
      throw new VCSException("Error encrypting session key.");
    }

    this.log("Establishing connection to server & sending session key.");
    try {
      // establish the socket connection
      Socket sock = new Socket(this.props.getProperty("SERVER_NAME"),
                     Integer.parseInt(this.props.getProperty("SERVER_PORT")));
      OutputStream out = sock.getOutputStream();
      InputStream  in  = sock.getInputStream();
      ObjectOutputStream objOut = new ObjectOutputStream(out);
      ObjectInputStream  objIn  = new ObjectInputStream(in);

      objOut.writeObject(msg1);

      // set up symmetric ciphers
      Cipher decryptCipher = Cipher.getInstance(Globals.SYMMETRIC_ALG);
      decryptCipher.init(Cipher.DECRYPT_MODE, sessionKey, sr);
      Cipher encryptCipher = Cipher.getInstance(Globals.SYMMETRIC_ALG);
      encryptCipher.init(Cipher.ENCRYPT_MODE, sessionKey, sr);

      this.log("Waiting for reply...");
      SealedObject msg2 = (SealedObject) objIn.readObject();
      this.log("Got vcs, verifying.");
      VCS vcs = (VCS) msg2.getObject(decryptCipher);

      Collection pubKeys =
        RSAKeyTool.getAllFromDir(this.props.getProperty("PUB_KEY_DIR"), true).values();
      boolean okay = VCS.verify(vcs, myKeys, pubKeys);
      if (!okay) this.log("ERROR: verify failed");
      this.log("Decrypting.");
      byte[] vcsSecret = vcs.decode(myKeys, pubKeys);
      this.log("The vcs secret is: " +
               Base64.encode(vcsSecret));
      this.log("Replying to server.");
      objOut.writeObject(new SealedObject(Base64.encode(vcsSecret),
                                          encryptCipher));

      // set up the CipherStreams
      this.cipherIn = new CipherInputStream(in, decryptCipher);
      this.cipherOut = new CipherOutputStream(out, encryptCipher);
    } catch (Exception e) {
      this.log(e);
    } finally {
      this.runDone = true;
    }
  } // authenticate

  /**
   * Get the encrypted input channel to the server.
   * Will block until the authentication transaction has finished.
   * <i>Warning: do not attempt to wrap an ObjectInputStream around
   * the returned CipherInputStream.  There appears to be a bug in the
   * 1.2.1 JDK implementation.</i>
   *
   * @throws VCSException if the client did not successfully
   * authenticate
   */
  public CipherInputStream getCipherInputStream()
      throws VCSException {
    while ( !this.runDone ) {
      try { Thread.sleep(5000); }
      catch (InterruptedException e) {}
    }
    if (this.cipherIn == null) {
      throw new VCSException("Client did not successfully authenticate");
    } else return this.cipherIn;
  }

  /**
   * Get the encrypted output channel to the server.
   * Will block until the authentication transaction has finished.
   * <i>Warning: do not attempt to wrap an ObjectOutputStream around
   * the returned CipherOutputStream.  There appears to be a bug in the
   * 1.2.1 JDK implementation.</i>
   *
   * @throws VCSException if the client did not successfully
   * authenticate
   */
  public CipherOutputStream getCipherOutputStream()
      throws VCSException {
    while ( !this.runDone ) {
      try { Thread.sleep(5000); }
      catch (InterruptedException e) {}
    }
    if (this.cipherOut == null) {
      throw new VCSException("Client did not successfully authenticate");
```

```
    } else return this.cipherOut;
  }

/**
 * Configure server information.
 *
 * @param host server to connect to
 * @param port port to connect to
 */
public void setServer(String host, int port) {
  if (host == null)
     throw new IllegalArgumentException("host cannot be null");
  if (host.equals(""))
     throw new IllegalArgumentException("host cannot be empty");
  if (port < 1)
     throw new IllegalArgumentException("port must be greater than 0");

  this.props.setProperty("SERVER_NAME", host);;
  this.props.setProperty("SERVER_PORT", (new Integer(port)).toString());
}

/**
 * Sets the directory to look for keys in.
 *
 * @param dir the directory to look in for keys
 */
public void setKeyDir(String dir) {
  this.props.setProperty("PUB_KEY_DIR", dir);
}

/**
 * Set the symmetric algorithm information.
 *
 * @param alg algorithm to use
 * @param keysize key size, in bits, to use
 */
public void setAlg(String alg, int keysize) {
  if (keysize < 1)
    throw new IllegalArgumentException("need positive keysize");
  this.props.setProperty("SYMMETRIC_KEY_SIZE",
                        (new Integer(keysize)).toString());
  this.props.setProperty("SYMMETRIC_ALG", alg);
}

/**
 * Set the current logging stream.  Pass null to turn logging off.
 *
 * @param stream the new stream to log to, or null to end logging
 */
public void setLogStream(OutputStream out) {
  if (out != null) this.logStream =
                  new PrintWriter(new OutputStreamWriter(out));
  else this.logStream = null;
}

/**
 * Write the specified string to the log
 *
 * @param s the string to write
 */
private synchronized void log(String s) {
  if (logStream != null) {
    logStream.println("[" + new Date() + "] " + s);
    logStream.flush();
  }
}

/**
 * Write the specified object to the log
 *
 * @param o the object to write
 */
private void log(Object o) { this.log(o.toString()); }

  /*
   * Because we're relying on users of the class to close the
   * streams, it's best to try to help them out where we can.  This
   * doesn't eliminate their need, but doesn't hurt.
   */
  protected void finalize() throws IOException {
    if (this.cipherOut != null) cipherOut.close();
    if (this.cipherIn  != null) cipherIn.close();
  }
}
```

B.4 VCSVector.java

```
/*
 * VCSVector.java
 */

package vcs;

import java.util.*;
import java.io.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.security.*;
import java.security.cert.*;
import java.security.interfaces.*;
import java.security.spec.*;
import COM.rsa.jsafe.*;
import COM.rsa.jsafe.provider.*;
import java.math.BigInteger;

/**
 * Vector-based implementation of Verifibly Common Secrets.  Linear in
 * time and space in the size of the group.
 *
 * @author Todd C. Parnell, tparnell@ai.mit.edu
 * @version $Id: VCSVector.java,v 1.9 1999/04/21 22:39:32 tparnell Exp $
 */
public class VCSVector extends VCS {

  /** Mapping from RSAPublicKeys to encrypted table entries. */
  private HashMap map = new HashMap();

  /**
   * Hash of the secret this VCS encodes for.  Used to verify update
   * (via addPrincipal) are using the same secret.
   */
  private BytesWrapper secretHash;

  /**
   * Should only be instantiated via VCS.getInstance.  */
  protected VCSVector() {}

  /**
   * Encode a VCSVector.  Each entry in keys will be used to encrypt
   * (using RSA) the secret.
   *
   * @param secret The secret to encode
   * @param keys The public keys to encode for
   * @throws VCSException if the VCS cannot be encoded with the
   * given keys
   */
  public void encode(byte[] secret, Collection keys) throws VCSException {
    if ( ! this.secretCheck(secret) ) {
      throw new VCSException("secret does not match with previous encoding");
    }
    try {
      Cipher cipher = Cipher.getInstance("RSA");
      Iterator it = keys.iterator();
      while (it.hasNext()) {
        RSAPublicKey key = (RSAPublicKey) it.next();
        cipher.init(Cipher.ENCRYPT_MODE, key);
        byte[] bytes = cipher.doFinal(secret);
        this.map.put(key, new BytesWrapper(bytes));
      }
    } catch (Exception e) {
      throw new VCSException(e.getMessage());
    }
  }

  /**
   * Add the given key to the set of authorized keys this VCS encodes
   * for.  Note that the secret given here must be the same as the
   * secret given for all other principals.
   *
   * @param secret The secret to encode
   * @param key The principal to encode for
   */
public void addPrincipal(byte[] secret, RSAPublicKey key)
     throws VCSException {
  if ( ! this.secretCheck(secret) ) {
    throw new VCSException("secret does not match with previous encoding");
  }
  try {
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.ENCRYPT_MODE, key);
    byte[] bytes = cipher.doFinal(secret);
    this.map.put(key, new BytesWrapper(bytes));
  } catch (Exception e) {
    throw new VCSException(e.getMessage());
  }
}


/**
 * Remove a prinicpal from the set of principals encoded for by this
 * VCS.  Requires the secret originally used to encode for to remove
 * the principal.
 *
 * @throws VCSException if secret does not match with the secret
 * originally encoded for.
 */
public void revokePrincipal(byte[] secret, RSAPublicKey key)
     throws VCSException {
  if ( ! this.secretCheck(secret) ) {
    throw new VCSException("secret does not match with previous encoding");
  }
  try {
    Iterator it = this.map.entrySet().iterator();
    while (it.hasNext()) {
      Map.Entry entry = (Map.Entry) it.next();
      RSAPublicKey testKey = (RSAPublicKey) entry.getKey();
      if (testKey.getModulus().equals(key.getModulus()) &&
          testKey.getPublicExponent().equals(key.getPublicExponent())) {
        // match!
        it.remove();
        return;
      }
    } // while
    // didn't find that principal
  } catch (Exception e) {
    throw new VCSException(e.getMessage());
  }
  throw new VCSException("Requested to remove a principal not encoded for.");
}

/**
 * Decode the VCS to learn the secret it encodes.  This method will
 * only work if the supplied private key corresponds to a public key
 * used for encoding.
 *
 * @param pair The Keypair to use for decoding.
 * @param keys The public keys used to encode.
 * @return The secret this VCS encoded.
 * @throws VCSException if the VCS cannot be decoded with the
 * provided keypair
 */
public byte[] decode(KeyPair pair, Collection keys) throws VCSException {
  RSAPublicKey pubKey = (RSAPublicKey) pair.getPublic();

  /* HACK: RSAPublicKey's equal method is broken.  We really want to
   * do this:
```

```java
   * BytesWrapper wrapper = (BytesWrapper) this.map.get(pubKey);
   * but instead we need to use an iterator and compare modulus
   * and exponent.
   */
  BytesWrapper wrapper = null;
  Set entries = this.map.entrySet();
  Iterator it = entries.iterator();
  while (it.hasNext()) {
    Map.Entry entry = (Map.Entry) it.next();
    RSAPublicKey testKey = (RSAPublicKey) entry.getKey();
    if (testKey.getModulus().equals(pubKey.getModulus()) &&
        testKey.getPublicExponent().equals(pubKey.getPublicExponent())) {
      // match!
      wrapper = (BytesWrapper) entry.getValue();
    }
  } // while
  // END HACK

  if (wrapper == null)
    throw new VCSException("The private key provided was not encoded for by this VCS.");

  try {
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.DECRYPT_MODE,
                (RSAPrivateKey) pair.getPrivate());
    return cipher.doFinal(wrapper.bytes);
  } catch (Exception e) {
    throw new VCSException(e.getMessage());
  }
}

public String toString() {
  return "A VCSVector";
}

/**
 * Make sure we encode for a single secret.  The magic here is that
 * we keep a hash of the secret between calls to encode.
 *
 * @return true iff secret matches with the secret used to encode
 */
private boolean secretCheck(byte[] secret) throws VCSException {
  try {
    MessageDigest md = MessageDigest.getInstance("SHA");
    BytesWrapper mdBytes = new BytesWrapper(md.digest(secret));
    if (this.secretHash != null) {
      if ( !this.secretHash.equals(mdBytes) )
        return false;
    } else
      this.secretHash = mdBytes;
  } catch (NoSuchAlgorithmException e) {
    throw new VCSException(e.getMessage());
  }
  return true;
}

/**
 * Determines if two VCSVectors are equal.  Warning, this is an
 * O(n^2) operation in the size of the set, due to the bug in
 * RSALabs' equal code.  Given a bugfix, it should be at worst O(n).
 */
private boolean equals(VCSVector o) {
  /* HACK: we want to do this:
   *    return this.map.equals(o.map);
   * but we can't.
   */
  // if not the same size, clearly not equal
  if (this.map.size() != o.map.size()) {
    System.out.println("DEBUG: size not equal");
    return false;
  }
  // find matching pairs, one at a time
  Set entries = this.map.entrySet();
  Iterator it = entries.iterator();
  // create a new set so we can remove entries as we go along
  Set oEntries = new HashSet(o.map.entrySet());
OUTER:
  while (it.hasNext()) {
    Map.Entry entry = (Map.Entry) it.next();
    RSAPublicKey testKey = (RSAPublicKey) entry.getKey();
    BigInteger modulus = testKey.getModulus();
    BigInteger exponent = testKey.getPublicExponent();
    Iterator it2 = oEntries.iterator();
    while (it2.hasNext()) {
      Map.Entry oEntry = (Map.Entry) it2.next();
      RSAPublicKey key = (RSAPublicKey) oEntry.getKey();
      if (modulus.equals(key.getModulus()) &&
          exponent.equals(key.getPublicExponent())) {
        // match -- remove the current entry and continue
        oEntries.remove(oEntry);
        continue OUTER;
      }
    } // inner loop
    // no match
    return false;
  } // while
  return true;
}

/**
 * Compares the specified Object with this VCSVector for equality.
 * Returns true if the given Object is also a VCSVector and both
 * have encoded the same secret for the same set of public keys.
 *
 * @param o object to be compared for equality with this VCSVector
 * @return true if the specified object is equal to this VCSVector
 */
public boolean equals(Object o) {
  if (o instanceof VCSVector)
    return this.equals( (VCSVector) o );
  else return false;
}

/**
 * Returns the hash code value for this VCS.  The hash code of a
 * VCSVector is determined by the secret and the public keys
 * encoding the secret.
 *
 * @return the hash code value for this VCSVector
 */
public int hashCode() {
  return this.map.hashCode();
}
}
```

B.5 RSAEnvelope.java

```java
/*
 * RSAEnvelope.java
 */

package vcs;

import java.io.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.security.*;
import java.security.cert.*;
import java.security.interfaces.*;
import java.security.spec.*;
import COM.rsa.jsafe.*;
import COM.rsa.jsafe.provider.*;

/**
 * Digital envelope for transmitting arbitratry data using RSA.
 * Encrypt a one-time symetric key using RSA, then encrypt the data
 * using your favorite symetric algorithm.
 *
 * @author Todd C. Parnell, tparnell@ai.mit.edu
 * @version $Id: RSAEnvelope.java,v 1.3 1999/03/31 20:49:20 tparnell Exp $
 */
public class RSAEnvelope implements Serializable {

  /** The encrypted symetric key. */
  private byte[] sessionKey;
  /** The encrypted data. */
  private SealedObject encryptedData;

  /**
   * Creates a digital envelope.  Uses DES as the symetric algorithm.
   *
   * @param data   The data to be put into the envelope.
   * @param random Source of randomness.
   * @param key    Recipitant's public key
   */
  public RSAEnvelope(Serializable data,
                     SecureRandom random,
                     RSAPublicKey RSAkey) {
    try {
      // first, create the symetric key
      KeyGenerator generator =
        KeyGenerator.getInstance("DES");
      generator.init(random);
      SecretKey sKey = generator.generateKey();

      // encrypt the SecretKey with the RSAPublicKey
      Cipher rsaCipher = Cipher.getInstance("RSA");
      rsaCipher.init(Cipher.ENCRYPT_MODE, RSAkey);
      this.sessionKey = rsaCipher.doFinal(sKey.getEncoded());

      // encrypt the data with the SecretKey
      Cipher desCipher = Cipher.getInstance("DES");
      desCipher.init(Cipher.ENCRYPT_MODE, sKey);
      this.encryptedData = new SealedObject(data, desCipher);

    } catch (Exception e) {
      e.printStackTrace();
      System.exit(0);
    }
  }

  /**
   * Returns the data in the envelope.
   */
  public Object open(RSAPrivateKey key) throws Exception {
    // decrypt the SecretKey with the RSAPrivateKey
    Cipher rsaCipher = Cipher.getInstance("RSA");
    rsaCipher.init(Cipher.DECRYPT_MODE, key);
    byte[] bytes = rsaCipher.doFinal(this.sessionKey);
    SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
    KeySpec spec = new DESKeySpec(bytes);
    SecretKey sKey = skf.generateSecret(spec);

    // decrypt the data with the SecretKey
    Cipher desCipher = Cipher.getInstance("DES");
    desCipher.init(Cipher.DECRYPT_MODE, sKey);
    return this.encryptedData.getObject(desCipher);
  }

}
```

B.6 NativeVCS.java

```
/*
 * NativeVCS.java
 */

package vcs;
import java.util.*;
import java.io.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.security.*;
import java.security.cert.*;
import java.security.interfaces.*;
import java.security.spec.*;
import COM.rsa.jsafe.*;

/**
 * Vector-based implementation of Verifibly Common Secrets.  Linear in
 * time and space in the size of the group.  Uses RSA labs native
 * libraries for encryption and decryption.
 *
 * @author Todd C. Parnell, tparnell@ai.mit.edu
 * @version $Id: NativeVCS.java,v 1.3 1999/04/16 23:35:32 tparnell Exp $
 */
public class NativeVCS extends VCS {

  /** Mapping from RSAPublicKeys to encrypted table entries. */
  private HashMap map = new HashMap();

  /**
   * Should only be instantiated via VCS.getInstance.
   */
  protected NativeVCS() {}

  /**
   * Encode the VCS for the given secret and public keys.
   *
   * @param secret The secret you wish to encode.
   * @param keys The public keys used to encode.
   * @throws VCSException if the secret cannot be encoded with the
   * given keys
   */
  public void encode(byte[] secret, Collection keys)
       throws VCSException {
    try {
      SecureRandom sr = new SecureRandom();
      JSAFE_AsymmetricCipher cipher =
        JSAFE_AsymmetricCipher.getInstance("RSA", "Native");
      Iterator it = keys.iterator();
      while (it.hasNext()) {
        RSAPublicKey defaultKey = (RSAPublicKey) it.next();
        byte[] modulus = defaultKey.getModulus().toByteArray();
        byte[] exponent = defaultKey.getPublicExponent().toByteArray();
        JSAFE_PublicKey key =
          JSAFE_PublicKey.getInstance("RSA", "Native");
        key.setKeyData( "RSAPublicKey", new byte[][] {modulus, exponent} );
        cipher.encryptInit(key, sr);
        cipher.encryptUpdate(secret, 0, secret.length);
        byte[] bytes = cipher.encryptFinal();
        this.map.put(key, new BytesWrapper(bytes));
      }
    } catch (Exception e) {
      System.out.println(e);
      throw new VCSException(e.getMessage());
    }
  }

  /**
   * Decode the VCS to learn the secret it encodes.  This method will
   * only work if the supplied private key corresponds to a public key
   * used for encoding.
   *
   * @param pair The Keypair to use for decoding.
   * @param keys The public keys used to encode.
   * @return The secret this VCS encoded.
   * @throws VCSException if the VCS cannot be decoded with the
   * provided keypair
   */
  public byte[] decode(KeyPair pair, Collection keys)
       throws VCSException {

    JSAFE_PublicKey pubKey = null;
    try {
      RSAPublicKey defaultKey = (RSAPublicKey) pair.getPublic();
      byte[] modulus = defaultKey.getModulus().toByteArray();
      byte[] exponent = defaultKey.getPublicExponent().toByteArray();
      pubKey = JSAFE_PublicKey.getInstance("RSA", "Native");
      pubKey.setKeyData( "RSAPublicKey", new byte[][] {modulus, exponent} );
    } catch (Exception e) {
      throw new VCSException(e.getMessage());
    }

    /* HACK: JSAFE_PublicKey's equal method is broken.  We really
     * want to do this:
     * BytesWrapper wrapper = (BytesWrapper) this.map.get(pubKey);
     * but instead we need to use an iterator and compare modulus
     * and exponent.
     */
    BytesWrapper wrapper = null;
    byte[][] keyData = pubKey.getKeyData();
    Set entries = this.map.entrySet();
    Iterator it = entries.iterator();

       OUTER_LOOP:
    while (it.hasNext()) {
      Map.Entry entry = (Map.Entry) it.next();
      JSAFE_PublicKey testKey = (JSAFE_PublicKey) entry.getKey();
      byte[][] testKeyData = testKey.getKeyData();
      for (int i = 0; i < keyData.length; ++i) {
        for (int j = 0; j < keyData[i].length; ++j) {
          if (keyData[i][j] != testKeyData[i][j]) continue OUTER_LOOP;
        }
      }
      // match!
      wrapper = (BytesWrapper) entry.getValue();
    } // while
    // END HACK

    if (wrapper == null) System.out.println("DEBUG: wrapper is null");

    try {
      RSAPrivateKey defaultKey = (RSAPrivateKey) pair.getPrivate();
      byte[] modulus = defaultKey.getModulus().toByteArray();
      byte[] exponent = defaultKey.getPrivateExponent().toByteArray();
      JSAFE_PrivateKey key =
        JSAFE_PrivateKey.getInstance("RSA", "Native");
      key.setKeyData( "RSAPrivateKey", new byte[][] {modulus, exponent} );

      JSAFE_AsymmetricCipher cipher =
        JSAFE_AsymmetricCipher.getInstance("RSA", "Native");
      cipher.decryptInit(key);
      cipher.decryptUpdate(wrapper.bytes, 0, wrapper.bytes.length);
      return cipher.decryptFinal();
    } catch (Exception e) {
      throw new VCSException(e.getMessage());
    }
  }
}
```

B.7 InsecureVCS.java

```java
/*
 * InsecureVCS.java
 */

package vcs;
import java.math.BigInteger;
import java.security.KeyPair;
import java.util.Collection;

/**
 * A trivial, insecure VCS implementation.  Keeps the secret in
 * unencrypted format.  Decoding an InsecureVCS does <b>not</b> have
 * the requirement that a matching private key be provided.
 *
 * <p>This class should be used for <i>testing purposes only.</i> It
 * provides no security and does not fulfill the contract of
 * <b>decode</b>.
 *
 * @author Todd C. Parnell, tparnell@ai.mit.edu
 * @version $Id: InsecureVCS.java,v 1.6 1999/04/16 23:35:33 tparnell Exp $
 */
public class InsecureVCS extends VCS {

  /** Brain dead storage of the secret */
  private byte[] secret;

  /**
   * Should only be instantiated via VCS.getInstance.
   */
  protected InsecureVCS() {}

  /**
   * Encode a InsecureVCS.  <i>Note: no cryptographic operations are
   * performed, and the encoding uses is the idenity function.</i>
   *
   * @param secret The secret to encode
   * @param keys The public keys to encode for.  (Ignored)
   */
  public void encode(byte[] secret, Collection keys) {
    this.secret = secret;
  }

  /**
   * Determine the secret encoded.  <i>Note: no cryptographic
   * operations are performed, and this method will return the secret
   * regardless of the parameters passed to it.</i>
   *
   * @param pair KeyPair to use for decoding.  (Ignored)
   * @param keys The public keys used to encode.  (Ignored)
   */
  public byte[] decode(KeyPair pair, Collection keys) {
    return this.secret;
  }

  public boolean equals(Object o) {
    if (o instanceof InsecureVCS) return this.equals( (InsecureVCS)o );
    return false;
  }

  public boolean equals(InsecureVCS vcs) {
    return (vcs.secret == this.secret);
  }

  public int hashCode() {
    if (this.secret == null) return 0;
    return (new BigInteger(this.secret)).hashCode();
  }

  public Object clone() {
    InsecureVCS vcs = new InsecureVCS();
    vcs.encode(this.secret, null);
    return vcs;
  }

  public String toString() {
    return "An Insecure VCS.  Encoded secret: " +
      vcs.util.Base64.encode(this.secret);
  }
}
```

B.8 RSAKeyTool.java

```
/*
 * RSAKeyTool.java
 */

// ToDo:
// Add command line flags for Iteration and PBEAlg
// Clean up some error handling

package vcs;

import javax.crypto.*;
import javax.crypto.spec.*;
import java.security.*;
import java.security.cert.*;
import java.security.interfaces.*;
import java.security.spec.*;
import java.io.*;
import vcs.util.Base64;
import COM.rsa.jsafe.*;
import COM.rsa.jsafe.provider.*;
import java.util.*;
import java.math.BigInteger;

/**
 * RSA Key creation and management utility class.
 *
 * @author Todd C. Parnell, tparnell@ai.mit.edu
 * @version $Id: RSAKeyTool.java,v 1.11 1999/04/16 23:37:06 tparnell Exp $
 */
public class RSAKeyTool {

  //
  // Class fields and methods
  //

  private static KeyFactory keyFactory;

  /**
   * Recover RSA public key encoded with this tool in files back into a
   * RSAPublicKey.
   *
   * @param keyFile string file name where the encoded key resides
   * @return the RSA public key corresponding to the file
   * @throws IOException if keyFile doesn't exits or cannot be read
   * @throws NoSuchAlgorithmException if RSA cannot be found
   * @throws InvalidKeySpecException if keyFile doesn't specify a valid RSAPublicKey
   */
  public static RSAPublicKey stringToPubKey(String keyFile)
       throws IOException, NoSuchAlgorithmException, InvalidKeySpecException {
    if (RSAKeyTool.keyFactory == null)
      RSAKeyTool.keyFactory = KeyFactory.getInstance("RSA");
    File file = new File(keyFile);
    FileInputStream fis = new FileInputStream(file);
    byte[] keyBytes = new byte[(int)file.length()];
    fis.read(keyBytes);
    fis.close();

    EncodedKeySpec encKeySpec = new X509EncodedKeySpec(keyBytes);
    return (RSAPublicKey)RSAKeyTool.keyFactory.generatePublic(encKeySpec);
  }

  /**
   * Recover RSA private key encoded with this tool in files back into a
   * RSAPrivateKey.
   *
   * @param keyFile string file name where the endoded resides
   * @param pass password to unlock the key
   * @return the RSA public key corresponding to the file
   * @throws IOException if keyFile doesn't exits or cannot be read
   * @throws NoSuchAlgorithmException if RSA cannot be found
   * @throws InvalidKeySpecException if keyFile doesn't specify a valid RSAPrivateKey
   */
public static RSAPrivateKey stringToPriKey(String keyFile,
                                           char[] pass)
     throws IOException, NoSuchAlgorithmException, InvalidKeySpecException,
            NoSuchPaddingException, InvalidAlgorithmParameterException,
            InvalidKeyException, IllegalBlockSizeException,
            BadPaddingException {

  if (RSAKeyTool.keyFactory == null)
    RSAKeyTool.keyFactory = KeyFactory.getInstance("RSA");

  File file = new File(keyFile);
  FileInputStream fis = new FileInputStream(file);
  byte[] salt = new byte[8];
  byte[] fileBytes = new byte[(int)file.length() - 8];
  fis.read(salt);
  fis.read(fileBytes);
  fis.close();
  // decrypt to get PKCS8 encoded private key
  KeySpec ks = new PBEKeySpec(pass);
  SecretKeyFactory skf =
    SecretKeyFactory.getInstance(Globals.PBE_ALG);
  SecretKey key = skf.generateSecret(ks);
  AlgorithmParameterSpec aps =
    new PBEParameterSpec(salt, Globals.PBE_ITERATIONS);
  Cipher pbeCipher = Cipher.getInstance(Globals.PBE_ALG);
  pbeCipher.init(Cipher.DECRYPT_MODE, key, aps);
  byte[] RSAKeyBytes = pbeCipher.doFinal(fileBytes);
  // decode to get a PrivateKey
  EncodedKeySpec encKeySpec =
      new PKCS8EncodedKeySpec(RSAKeyBytes);
  return (RSAPrivateKey) RSAKeyTool.keyFactory.generatePrivate(encKeySpec);
}

/**
 * Add all public keys from dir to the HashSet.
 *
 * @param dir directory to retreive keys from
 * @return all the public keys the the directory
 */
public static HashMap getAllFromDir(String dir) throws Exception {
  return RSAKeyTool.getAllFromDir(dir, false);
}

/**
 * Add public keys from dir to the HashSet.  If onlyDefaults is
 * false, adds all public keys in directory.  If true, only
 * principals in <i>VCS.defaults</i> will be added.
 *
 * @param dir directory to retreive keys from
 * @param onlyDefaults controls whether to igrone VCS.defaults or not
 * @return public keys from the directory
 */
public static HashMap getAllFromDir(String dir, boolean onlyDefaults)
     throws Exception {
  // get directory
  File dirFile = new File(dir);
  if (dirFile.isFile())
    throw new IOException(dir + " is not a directory");

  // temp storage for files
  HashSet files = new HashSet();

  if (onlyDefaults) {
    File control = new File(dir + File.separator + "VCS.defaults");
    if (control.exists()) {
      BufferedReader br = new BufferedReader(new InputStreamReader(new FileInputStream(control)));
```

```java
      String line;
      while ( (line = br.readLine()) != null ) {
        String temp = dir + File.separator + line + ".pub";
        File f = new File(temp);
        if ( !f.exists() ) continue;
        files.add(temp);
      }
      br.close();
    }
  } else /* all files */ {
    String[] allFiles = dirFile.list();
    int length = allFiles.length;
    for (int i = 0 ; i < length ; ++i) {
      String temp = dir + File.separator + allFiles[i];
      if (temp.regionMatches(temp.length()-4, ".pub", 0, 4)) {
        files.add(temp);
      }
    }
  }
  // convert String filename into public keys
  HashMap keys = new HashMap();
  Iterator it = files.iterator();
  while (it.hasNext()) {
    String next = (String) it.next();
    keys.put(next, RSAKeyTool.stringToPubKey(next));
  }
  return keys;
}


private static final String usageStr =
"usage: vcs.RSAKeyTool [options]\n" +
"   -h --help       : Print this message.\n" +
"   -v              : Operate verbosely.\n" +
"   -principal name : Create a keypair for name.\n" +
"   -keypass pass   : Use pass to lock private key.\n" +
"   -d dir          : Use dir for key storage.\n" +
"                     (Default = " +
                      Globals.PUB_KEY_DIR + ")\n" +
"   -keysize size   : Set keysize.  (Default = " +
                      Globals.ASSYMETRIC_KEY_SIZE + ")\n" +
"   -modulus size   : Set modulus.  (Default = " +
                      Globals.RSA_MODULUS_SIZE + ")\n";

public static void main(String[] args) {
  try {
    Security.addProvider(new com.sun.crypto.provider.SunJCE());
    Security.addProvider(new COM.rsa.jsafe.provider.JsafeJCE());
  } catch (Exception e) {
    System.out.println("Unable to add crypto providers.  Exiting.");
    System.exit(0);
  }
  RSAKeyTool me = new RSAKeyTool(args);
}

//
// Instance fields and methods
//

/** Operate verbosely? */
private boolean verbose;
/** Password for the private key. */
private String passwd;
/** Principle we're manipulating. */
private String principal;
/** Directory to put keys. */
private String keyDir = Globals.PUB_KEY_DIR;
/** A source of randomness */
private SecureRandom random;
/** RSA Key Size, default = 512 */
private int keySize = Globals.ASSYMETRIC_KEY_SIZE;
/** RSA Modulus Size, default = 17 */
private int modulus = Globals.RSA_MODULUS_SIZE;

/**
 * The Constructor
 */
private RSAKeyTool(String[] args) {
  // populate instance fields
  this.parseArgs(args);
  // ask user about any missing information
  this.getUserInput();
  // seed the PRNG
  this.setupPRNG();
  // do it!
  this.create();
} // constructor

/**
 * Creates the keypair & save to the given locations.
 */
private void create() {
  if (this.verbose) {
    System.out.println("Beginning key creation.");
  }
  try {
    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
    BigInteger bigint =
      new BigInteger( (new Integer(this.modulus)).toString() );
    RSAGenParameterSpec spec =
      new RSAGenParameterSpec(this.keySize, bigint);
    keyGen.initialize(spec, this.random);
    if (this.verbose) {
      System.out.println("Generating keypair.");
    }
    KeyPair pair = keyGen.genKeyPair();
    RSAPublicKey pubKey = (RSAPublicKey) pair.getPublic();
    RSAPrivateKey priKey = (RSAPrivateKey) pair.getPrivate();
    if (this.verbose) {
      System.out.print("Public key : ");
      System.out.println(Base64.encode(pubKey.getEncoded()));
      System.out.print("Private key : ");
      System.out.println(Base64.encode(priKey.getEncoded()));
    }

    if (this.verbose) {
      System.out.println("Writing keypair to files.");
    }
    // the public key is easy, since it doesn't need to be protected
    FileOutputStream fos =
      new FileOutputStream(this.keyDir +
                           "/" +
                           this.principal +
                           ".pub");
    fos.write(pubKey.getEncoded());
    fos.close();

    // the private key will be protected with a passphrase
    // first, create some salt...
    byte[] salt = new byte[8];
    MessageDigest md = MessageDigest.getInstance("MD5");
    md.update(this.passwd.getBytes());
    md.update(priKey.getEncoded());
    System.arraycopy(md.digest(), 0, salt, 0, 8);

    // set up the encryption
    KeySpec ks = new PBEKeySpec(this.passwd.toCharArray());
    SecretKeyFactory skf =
      SecretKeyFactory.getInstance(Globals.PBE_ALG);
    SecretKey pbeKey = skf.generateSecret(ks);
    AlgorithmParameterSpec aps =
      new PBEParameterSpec(salt, Globals.PBE_ITERATIONS);
    Cipher pbeCipher = Cipher.getInstance(Globals.PBE_ALG);
    // do the encryption
```

```java
    pbeCipher.init(Cipher.ENCRYPT_MODE, pbeKey, aps);
    // write to file
    fos = new FileOutputStream(this.keyDir +
                               "/" +
                               this.principal +
                               ".pri");
    fos.write(salt);
    fos.write(pbeCipher.doFinal(priKey.getEncoded()));
    fos.close();
  } catch (Exception e) {
    System.err.println("Error.  Aborting.");
    e.printStackTrace();
    System.exit(0);
  }
} // create

/**
 * Populate instance fields with info from command line arguments.
 * Moved to seperate method since it's messy and boring.
 */
private void parseArgs(String[] args) {
  for (int i = 0; i < args.length; ++i) {
    String option = args[i];
    if (option.equals("-v")) {
      this.verbose = true;
    } else if (option.equals("-h") || option.equals("--help")) {
      System.out.println(RSAKeyTool.usageStr);
      System.exit(0);
    } else if (option.equals("-principal")) {
      try {
        this.principal = args[++i];
      } catch (ArrayIndexOutOfBoundsException obe) {
        System.err.println("Must provide name with -prinicpal option.  Exiting.");
        System.exit(0);
      }
    } else if (option.equals("-d")) {
      try {
        this.keyDir = args[++i];
      } catch (ArrayIndexOutOfBoundsException obe) {
        System.err.println("Must provide dir with -d option.  Exiting.");
        System.exit(0);
      }
    } else if (option.equals("-keypass")) {
      try {
        this.passwd = args[++i];
      } catch (ArrayIndexOutOfBoundsException obe) {
        System.err.println("Must provide file with -keypass option.  Exiting.");
        System.exit(0);
      }
    } else if (option.equals("-keysize")) {
      try {
        this.keySize =
          Integer.valueOf(args[++i]).intValue();
      } catch (ArrayIndexOutOfBoundsException obe) {
        System.err.println("Must provide number with -keysize option.  Exiting.");
        System.exit(0);
      } catch (NumberFormatException nfe) {
        System.err.println("Couldn't parse keysize.  Exiting.");
        System.exit(0);
      }
    } else if (option.equals("-modulus")) {
      try {
        this.modulus =
          Integer.valueOf(args[++i]).intValue();
      } catch (ArrayIndexOutOfBoundsException obe) {
        System.err.println("Must provide number with -modulus option.  Exiting.");
        System.exit(0);
      } catch (NumberFormatException nfe) {
        System.err.println("Couldn't parse modulus.  Exiting.");
        System.exit(0);
      }
    } else /* error */ {
      System.err.println("Unknown option: '" + option + "'.  Exiting.");
      System.err.println(RSAKeyTool.usageStr);
      System.exit(0);
    }
  }
} // parseArgs

/**
 * Prompts user for any data not currently in fields.
 */
private void getUserInput() {
  while (this.principal == null || this.principal.equals("")) {
    System.out.print("Enter principle to operate on: ");
    try {
      BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
      this.principal = br.readLine();
    } catch (IOException ioe) {
      System.err.println("IOException!  Exiting.");
      System.exit(0);
    } // try/catch
  } // while

  while (this.passwd == null || this.passwd.equals("")) {
    System.out.print("Enter passphrase to lock private key: ");
    try {
      BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
      this.passwd = br.readLine();
    } catch (IOException ioe) {
      System.err.println("IOException!  Exiting.");
      System.exit(0);
    } // try/catch
  } // while
} // getUserInput

private void setupPRNG() {
  if (this.verbose) {
    System.out.println("Initilizing random number generator");
  }
  try {
    this.random = SecureRandom.getInstance("SHA1PRNG", "JsafeJCE");
    System.out.println("Enter some keystrokes to seed the random number generator.");
    System.out.println("Press return after a line or two of text.");
    BufferedReader br =
      new BufferedReader(new InputStreamReader(System.in));
    String temp = br.readLine();
    this.random.setSeed(temp.getBytes());
  } catch (IOException ioe) {
    ioe.printStackTrace();
    System.exit(0);
  } catch (NoSuchAlgorithmException alg) {
    alg.printStackTrace();
    System.exit(0);
  } catch (NoSuchProviderException prov) {
    prov.printStackTrace();
    System.exit(0);
  }
} // setupPRNG
```

B.9 BytesWrapper.java

```
/*
 * BytesWrapper.java
 */

package vcs;
import java.io.Serializable;
import java.math.BigInteger;

/**
 * Wrapper object to for byte arrays.
 *
 * @author Todd C. Parnell, tparnell@ai.mit.edu
 * @version $Id: BytesWrapper.java,v 1.1 1999/04/11 17:44:10 tparnell Exp $
 */
public class BytesWrapper implements Serializable {
  /** What we're wrapping. */
  public final byte[] bytes;
  /** Support for fast hashing */
  private transient int hash;

  /**
   * Construct a new wrapper for the given bytes.
   *
   * @param bytes the bytes to wrap
   */
  public BytesWrapper(byte[] bytes) {
    this.bytes = bytes;
  }

  /**
   * Extract the bytes from the wrapper.
   *
   * @return the wrapped bytes
   */
  public byte[] getBytes() {
    return this.bytes;
  }

  public boolean equals(Object o) {
    if (o instanceof BytesWrapper) {
      BytesWrapper bw = (BytesWrapper) o;
      int mylength = this.bytes.length;
      if (mylength != bw.bytes.length) return false;
      for (int i=0; i<mylength; ++i) {
        if (this.bytes[i] != bw.bytes[i]) return false;
      }
      return true;
    } else return false;
  }

  public int hashCode() {
    // Q&D way to get a hash.  Cache the value.
    if (this.hash != 0) return this.hash;
    this.hash = (new java.math.BigInteger(this.bytes)).hashCode();
    return this.hash;
  }
}
```

B.10 Globals.java

```
/**
 * Globals.java
 */

package vcs;

/**
 * Holds some globals.
 *
 * @author Todd C. Parnell, tparnell@ai.mit.edu
 * @version $Id: Globals.java,v 1.9 1999/04/16 23:36:15 tparnell Exp $
 */
public class Globals {

  /** Session key size, in bits.  Default is 56. */
  public static final int SESSION_KEY_SIZE = 128;
  /** Asymetric key size.  Default is 512. */
  public static final int ASSYMETRIC_KEY_SIZE = 512;
  /** RSA modulus size.  Default is 17. */
  public static final int RSA_MODULUS_SIZE = 17;
  /** Default public key directory. */
  public static final String PUB_KEY_DIR = "/mit/tparnell/thesis/keys/";
  /** Default server to authenticate to. */
  public static final String SERVER_NAME = "fop.mit.edu";
  /** Default server port. */
  public static final int SERVER_PORT = 4321;
  /** Phassphrase Based Encryption alg. */
  public static final String PBE_ALG = "PBEWithMD5AndDES";
  /** PBE Iteration Count */
  public static final int PBE_ITERATIONS = 20;
  /** Default Symmetric Algorithm */
  public static final String SYMMETRIC_ALG = "RC4";
  /** Default VCS class. */
  public static final String DEFAULT_VCS_CLASS = "vcs.VCSVector";
  /** Default properties file */
  public static final java.util.Properties DEFAULT_PROPERTIES =
    new java.util.Properties();

  static {
    try {
      DEFAULT_PROPERTIES.setProperty(
        "SESSION_KEY_SIZE",
        (new Integer(SESSION_KEY_SIZE)).toString());
      DEFAULT_PROPERTIES.setProperty(
        "ASSYMETRIC_KEY_SIZE",
        (new Integer(ASSYMETRIC_KEY_SIZE)).toString());
      DEFAULT_PROPERTIES.setProperty(
        "RSA_MODULUS_SIZE",
        (new Integer(RSA_MODULUS_SIZE)).toString());
      DEFAULT_PROPERTIES.setProperty("PUB_KEY_DIR",
                                     PUB_KEY_DIR);
      DEFAULT_PROPERTIES.setProperty("SERVER_NAME", SERVER_NAME);
      DEFAULT_PROPERTIES.setProperty(
        "SERVER_PORT",
        (new Integer(SERVER_PORT)).toString());
      DEFAULT_PROPERTIES.setProperty("PBE_ALG" , PBE_ALG);
      DEFAULT_PROPERTIES.setProperty(
        "PBE_ITERATIONS",
        (new Integer(PBE_ITERATIONS)).toString());
      DEFAULT_PROPERTIES.setProperty("SYMMETRIC_ALG", SYMMETRIC_ALG);
      DEFAULT_PROPERTIES.setProperty("DEFAULT_VCS_CLASS", DEFAULT_VCS_CLASS);
    } catch (Exception e) {
      // do nothing
    }
  }

  /** Prevent instantiation. */
  private Globals() {}
}
```

B.11 VCSException.java

```
/*
 * VCSException.java
 */

package vcs;

/**
 * Generic exception thrown by classes in package vcs.
 *
 * @author Todd C. Parnell, tparnell@ai.mit.edu
 * @version $Id: VCSException.java,v 1.1 1999/04/06 20:32:04 tparnell Exp $
 */
public class VCSException extends Exception {
  public VCSException() {}
  public VCSException(String s) { super(s); }
}
```