

Self-Describing Cryptography through Certified Universal Code

by

Benjamin Adida

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

© Massachusetts Institute of Technology 1999. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by
Ronald L. Rivest
Webster Professor of Electrical Engineering and Computer Science
Thesis Supervisor

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

Self-Describing Cryptography through Certified Universal Code

by

Benjamin Adida

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

There are few end-users today who make use of real security applications. These applications tend to be too complicated, exposing too much detail of the cryptographic process. Users need *simple, inherent* security that doesn't require more of them than simply clicking the "secure" checkbox.

Self-Describing Cryptography is a first step towards this inherent security. It applies the basic programmatic concept of abstraction to separate specific algorithms from generic cryptographic processes in order to eliminate compatibility and upgradability problems. The core idea is to include all algorithms used in a given cryptographic operation within the resulting data itself. These algorithms can then be retrieved by a receiving party to perform decryption, signature verification, and certification, without the receiving party holding any prior knowledge of the specifics of these algorithms. In the process of creating and securing this new abstraction layer, Self-Describing Cryptography explores a new, more flexible security model, where security is defined not as the impossibility to obtain secret information, but as the impossibility to make use of any secret information obtained.

Thesis Supervisor: Ronald L. Rivest

Title: Webster Professor of Electrical Engineering and Computer Science

Acknowledgments

I would first like to thank Professor Ronald L. Rivest, my thesis advisor over the past year. As a freshman, I emailed more than twenty professors about potential UROP (Undergraduate Research) positions. With my lack of experience at the time, I wasn't able to find any such position, but did receive one response pointing to classes I should take to gain this experience, and inviting me to reapply at a later date. The answer was from Professor Rivest. Later, as my fascination with security and cryptography grew, I was able to help out on the secure electronic voting project in Professor Rivest's group. This eventually led to an internship at RSA Data Security, where my interest in combining Java and Cryptography grew even more. As a Master's student, I was given total freedom by Professor Rivest in exploring my idea on self-describing cryptography, which led to this thesis project. Regardless of the amazing advances Professor Rivest has made in the fields of security and cryptography, he has remained extremely approachable and friendly. For all of this I thank him very much.

This research would not have been possible without DARPA's generosity, grant #DABT63-96-C-0018. Dedicated research time, thanks to such grants, leads, in my opinion, to much more useful and interesting research results.

I would also like to thank Professor Abelson, who taught me both when I was his 6.001 student, and 3 years later his 6.001 Teaching Assistant. I have never met anyone so dedicated to students and to the entire education process at MIT, and so talented in his teaching of computer science.

I certainly could not have done this without my friends. Lydia Sandon, my close friend and love since sophomore year, who helped (and continues to help) me in every-day life as well as with reviewing the ideas I put forth in this thesis. Phil Frei and Rodrigo Leroux, my roommates, who helped me test the system and provided wildly entertaining dinner conversations throughout the year. Oliver Roup, who basically made me realize that this idea of self-describing cryptography wasn't so bad after all.

Most importantly, I would like to thank my parents, Pierre and Yvette, and my two sisters, Claire and Juliette, for their unwavering support and encouragement throughout my years at MIT (and earlier). I could not have done any of this without them and the knowledge that I could always turn to them for advice on any topic, at any time. This thesis is dedicated to them, for everything they have done for me.

A person's accomplishments are rarely the product of only him or herself. I owe my sanity and happiness to all of the above mentioned people, and many many more.

Contents

1	Status and Motivation	8
1.1	Security in Today's Applications	8
1.2	Making Cryptography More Transparent	9
1.3	Previous Work	9
2	The Basics of Self-Describing Cryptography	11
2.1	Why It Couldn't Have Been Done Earlier: Java	11
2.2	Standardizing Cryptographic Processes	12
2.2.1	Symmetric Encryption	13
2.2.2	Key Packaging	14
2.2.3	Signing and Verification	16
2.3	Trusting Algorithm Code	17
2.3.1	Simple Certification	18
2.3.2	Bootstrapping Certification	18
2.3.3	Limiting Actions the Algorithm Can Take	20
3	Design of System	21
3.1	Basic Elements	21
3.1.1	The <code>Algorithm</code> Class	21
3.1.2	The <code>Message</code> class	22
3.1.3	Basic Cryptographic Classes	23
3.2	Dynamically Loading Code	24
3.2.1	Name-Spaces	25
3.3	Security Context API	26
3.4	Sandboxing the Code	26
3.5	The Protocols	28
3.5.1	Sending a Message	30
3.5.2	Receiving a Message	33
3.6	Hashed Algorithms	36
3.7	Packaging into a Self-Executable	37
4	Security Model and Attacks	38
4.1	The Trust Model	38
4.1.1	The Entities Involved	38
4.1.2	The Degrees of Trust	39
4.1.3	Entities and Their Level of Trust	39
4.2	Security Attacks on SDC	40

4.2.1	Programmatic Attacks	40
4.2.2	Protocol Attacks	42
4.2.3	Cryptographic Attacks	44
4.3	The No-Way-Out-Sandbox Security Model	45
5	Implementation Details	47
5.1	Java Class Dependencies	48
5.2	Proper Isolation of Algorithms	49
5.2.1	Name Spaces: the <code>ClassLoader</code> object	50
5.2.2	Name-Space Delegation	50
5.2.3	Loading Serialized Objects	51
5.3	Applying Security Constraints to Java 1.2 Code	53
5.3.1	System-Wide Policy	54
5.3.2	Taking <code>PrivilegedAction</code>	55
5.3.3	Assigning the <code>ProtectionDomain</code>	55
6	Conclusion and Future Work	57
6.1	Why Is SDC Important?	58
6.2	Future Work	59
A	Algorithm Implementation Guidelines	60
A.1	Cipher, Plaintext and Ciphertext	60
A.2	Symmetric and Asymmetric Ciphers	61
A.2.1	Symmetric Ciphers	61
A.2.2	Asymmetric Ciphers	62
A.3	Key Packagers	63
A.4	Signing Algorithms	65
A.5	Using the <code>SimpleAlgorithmMaker</code>	66
B	Plaintext Covert Channel Security Fix	67
B.1	The Problem	67
B.2	The Basic Solution	68
B.3	The Consequences for Self-Describing Cryptography	69
C	sde.crypto	70
C.1	Interfaces	70
C.1.1	<i>Interface</i> <code>Signer</code>	70
C.1.2	<i>Interface</i> <code>Ciphertext</code>	71
C.1.3	<i>Interface</i> <code>Cipher</code>	72
C.1.4	<i>Interface</i> <code>Plaintext</code>	72
C.1.5	<i>Interface</i> <code>AsymmetricCipher</code>	73
C.2	Classes	74
C.2.1	<i>Class</i> <code>SymmetricCipher</code>	74
D	sde.protocol	77
D.1	Interfaces	77
D.1.1	<i>Interface</i> <code>SecurityContext</code>	77
D.2	Classes	78
D.2.1	<i>Class</i> <code>KeyPackager</code>	78

D.2.2	<i>Class</i> SDEKeyPair	81
D.2.3	<i>Class</i> EmptyAlgorithm	81
D.2.4	<i>Class</i> Message	85
E	sde.algorithm	89
E.1	Classes	89
E.1.1	<i>Class</i> Algorithm	89
E.1.2	<i>Class</i> AlgorithmClassLoader	92

List of Figures

3-1	An Algorithm Object	22
3-2	A Message Object	23
3-3	Password-Based Key Generation	29
3-4	Public-Key Key Generation	29
3-5	Password-Based Encryption	31
3-6	Public-Key Encryption	31
3-7	Password-Based Key Packaging	32
3-8	Public-Key Key Packaging	33
3-9	Password-Based Unpackaging	34
3-10	Public-Key Unpackaging	34
3-11	Public-Key and Password-Based Decryption	35
3-12	The Initialization of an Empty Algorithm	36
5-1	SDC Class Hierarchy	47
5-2	Object, Class, and ClassLoader Dependencies	49
5-3	The Security Manager regulates access to the Java APIs and to the SDC Security Context	54

Chapter 1

Status and Motivation

1.1 Security in Today's Applications

Achieving acceptable levels of security in software applications is quite complex. It involves meticulous software development using proven cryptographic algorithms and protocols. In general, only software engineers with years of experience in these domains are able to produce software that successfully stands up to most potential security threats. Even then, key sizes become too short as computing power increases, implementation bugs are found in certain algorithms or protocols, and sometimes cryptographic algorithms are discovered to be weak on a purely theoretical level. Because of the all-or-nothing nature of security, any single such breach can compromise the entire application.

Furthermore, there exist serious compatibility issues between various security applications. While S/MIME [3, 4] is emerging as a semi-standard for encrypted email, for example, PGP/MIME [2, 1] is still strong in the hacker community given that the algorithms it uses are not covered by patents. Even within those standards, there are at least 5 or 6 different algorithms that can be used, making it difficult to assume that the other player in a cryptographic exchange will have the ability to decrypt a particular type of encryption. This problem has been greatly aggravated by export restrictions the US government places on cryptography: other countries use different, sometimes incompatible implementations of algorithms, or even completely different algorithms altogether. This compatibility problem has greatly hampered the setup of a world-wide cryptographic architecture for all security needs.

1.2 Making Cryptography More Transparent

The root of these problems lies in that most security software lays bare its internals: it is almost impossible to find a piece of software that deals with security issues in a transparent way, such that the user need only know that the system has been secured. Users generally need to understand certificates, signatures, encryption, hashing, algorithm types, and key sizes to accomplish any level of security in their system. Because all of these tend to change with time (as described above), the user is forced to stay as up to date with security news as are security experts. This is clearly a bad solution. The user shouldn't need to know the details of every cryptographic process to achieve real security. A cryptographer's job needs to go beyond the development of new ciphers and protocols, into the realm of practical, usable security.

An initial solution is to apply a very simple and very basic software engineering principle to security applications: *abstraction*. The user only needs to know that a system has been secured, or that a security breach has been detected. If a cryptographic algorithm becomes weak, a security application should automatically stop using it and start using a new algorithm. If an encrypted message is received, but the application doesn't know the algorithm needed to decrypt it, it should automatically be able to locate this algorithm (from a service on the network), and transparently extend its own functionality.

All of this is possible through a new approach that I will call *Self-Describing Cryptography*. This technique consists of abstracting out the specific algorithmic computations from the cryptographic process in order to achieve better usability, upgradability, compatibility, and overall practicality in security applications.

1.3 Previous Work

Data containing enabling executable code is not a new concept. The idea of a self-extracting archive has existed for several years, with self-extracting archives on the Macintosh and self-unzipping zip files on Windows. These ideas are the inspiration for this concept of Self-Describing Cryptography.

In fact, the idea of self-describing cryptographic processes has been attempted before. RSA Data Security's SecurPC product performs a kind of self-describing trick by allowing the user to encrypt any file, using a password, into an executable. Upon runtime, this

executable prompts the user for the password again, and then proceeds to decrypt the ciphertext automatically. The executable is platform-specific (Windows-only), and does not provide any guarantee to the recipient that it does not contain malicious code. Although it is a decent “quick fix” solution for simple encryption between people who inherently trust each other (and believe that no one is corrupting their line of communication), SecurPC does not provide the level of security Self-Describing Cryptography aims to achieve.

The goal of Self-Describing Cryptography is to take these embryonic concepts and push them to the level of a complete, secure, cryptographic library. Self-Describing Cryptography isn't new as a concept, but it has not yet been successfully implemented as a general approach to cryptographic exchanges.

Chapter 2

The Basics of Self-Describing Cryptography

The basic idea of Self-Describing Cryptography is that any algorithm used in the process of cryptographically modifying a piece of data should be sent along with the processed data, so that methods for decryption and signature verification are included within the received message. Given the simplicity of this idea, it is somewhat surprising that it has not been put to use. There are, in fact, several reasons why Self-Describing Cryptography is not as straight-forward a concept as it initially seems:

- Different users work with different, incompatible computing platforms that make universal algorithmic descriptions difficult to define. There is a need for a universal code for algorithmic operations: a universal programming language.
- More importantly, there is an enormous security risk to letting a foreign algorithm “run loose” on one’s computer.

These two important problems are almost entirely solved by Java, a now well-known computer programming language and platform designed to tackle the problems of untrusted, mobile code.

2.1 Why It Couldn’t Have Been Done Earlier: Java

The Java platform builds a “virtual machine” on top of almost every existing computing platform on the market. This virtual machine, like any other computing platform, could

conceivably be implemented in silicon. Currently, though, it exists only as an emulated machine running on other processors. This means that Java bytecode, the assembly language of the Java Virtual Machine, can be executed on practically any computing platform without recompilation. Java thus provides a universal platform for code execution, a key necessity when describing algorithmic computations in a platform-independent manner.

Java's second key advantage is its highly flexible linking model. A piece of Java bytecode can be dynamically loaded from any source (disk, network, ..) and linked with the running codebase in a transparent fashion. Java provides fine-grained control over how this code is loaded, what name space it occupies, and what actions this code is allowed to take. This allows a running Java program to unexpectedly receive untrusted code and execute it in a "sandboxed" fashion that prevents any potential damage. A cryptographic algorithm, for example, can be allowed to compute cryptographic operations on data it is provided with, but not to make a socket connection to another host or open a file on the local system.

Finally, the Java Virtual Machine bytecode:

- has a typed stack,
- does not support direct memory pointers, and
- provides Garbage-Collection as the only way to clear unused memory.

These features provide the support for the secure computing environment in Java. Such strict control over how memory is created, handled and destroyed is required to prevent untrusted code from overstepping its designated bounds.

Thus, Java is the enabling technology for Self-Describing Cryptography. By allowing dynamic and secure code execution, the Java platform disables potential attacks presented by the mobile-code issues, which are at the very center of Self-Describing Cryptography. One should note that there is no claim made that Java solves every piece of this problem, but only that it enables a new set of programmatic ideas by providing primitives for fine-grained, flexible control of untrusted code.

2.2 Standardizing Cryptographic Processes

The first theoretical step in achieving Self-Describing Cryptography is a generalization of all cryptographic processes in a manner that is independent of the particular algorithms

used in the actual execution of these processes. While an in-depth analysis of any and all cryptographic processes is possible, this dissertation will limit itself to the most common practical protocols used today, without attempting to completely capture every cryptographic protocol imaginable.

More specifically, this dissertation will focus solely on connection-less protocols, like email or medium-stored data, which allow for clear analysis of the sending and receiving mechanisms. This is not to say that these techniques cannot be used in protocols assuming constant, live connections between the encrypting and decrypting parties. It is simply that Self-Describing Cryptography proves its advantages more strongly in a connection-less situation.

2.2.1 Symmetric Encryption

Symmetric Encryption is performed using one key. Usually this key is the same for both encryption and decryption; sometimes the decryption key is different, but easily derived, from the encryption key. This key can always be represented as one sequence of bytes.

This symmetric key is usually a perfectly random series of bytes, with a handful of “weak” keys that must be avoided for each algorithm, and sometimes some internal consistency within the construct of the key. DES (the Data Encryption Standard) [13], for example, uses keys with 56 bits of “real” key data, but represented using 8 bytes, each with 7 bits of data and one bit of parity. Most algorithms can accept key sizes of many different lengths, but there are lower and upper bounds for each algorithm.

In terms of input formatting, all symmetric ciphers can be tuned to accept input one byte at a time (either because they are stream ciphers, or because padding is performed to complete the last block). Thus, the interface to any symmetric cipher can be generalized to the use of streams of bytes for input and output.

In general, symmetric algorithms can be defined with the following properties and abilities:

- a symmetric key can be *generated and validated* using a source of randomness and a requested key size. Validation involves checking that the key is appropriately structured for the given cipher and that it isn’t a “weak” key for that cipher.
- a symmetric cipher can be *initialized for encryption* or decryption with a validated

symmetric key (of the corresponding type for that cipher).

- a symmetric cipher *performs encryption* using the plaintext to encrypt and a source of randomness that might be used in the encryption process (for padding...).
- a symmetric cipher *performs decryption* using only the ciphertext to decrypt.

2.2.2 Key Packaging

In certain situations (mostly military), it is conceivable to exchange a symmetric key in an offline manner. This can be performed, for example, by a secret agent transporting a suitcase of one-time pad CD-ROMs. In most practical applications, however, the symmetric key needs to be exchanged in an online fashion. There are many different methods to perform this exchange, and generalizing this part of the process is the crux of making self-describing cryptography fully functional.

In Self-Describing Cryptography, this key exchange process can be generalized under the umbrella concept of *key packaging*. The idea is that the session key is somehow “packaged” so that only the recipient can “unpackage” it. This packaging is relatively abstract: it might be implemented as true, physical packaging, where a session key is encrypted with the recipient’s public key, or as symbolic packaging, where the “package” is a shared secret knowledge of a passphrase between the sender and recipient.

The most obvious form of key packaging is through public-key cryptography [7, 5] to encrypt the symmetric key such that only the recipient can decrypt it. The encryption part constitutes the “packaging,” the decryption the “unpackaging.” The specific algorithm for public-key encryption also needs to be self-described. Since the public key is advertised by the recipient, the associated public-key encryption algorithm should be included along with this public key. This new public key is, in a way, an *augmented public key*.

A public-key algorithm needs an interface very similar to that defined for the symmetric cipher algorithms. The difference lies mainly in the keypair definition, given that most asymmetric keypairs require very specific internal structure (often confined to specific algebraic fields). The keypair generation is thus specific to each algorithm. All key pairs can be consulted for their private or public key components.

Public-key algorithms are used to encrypt symmetric keys. While these algorithms can often operate on inputs of all sizes, there remains a maximum size beyond which the

encryption must be done in blocks. Furthermore, inputs of small sizes may result in weak encryptions, in cases where the security relies on the circularity of an algebraic field (e.g. modular operations, etc...). This problem is often solved by some form of padding, which generally requires a source of randomness to generate the padding data.

The interface to an asymmetric cipher should be such that:

- the asymmetric cipher can be *initialized for encryption* with a public key.
- the asymmetric cipher can *perform encryption*, once initialized, using the plaintext to be encrypted and a source of randomness.
- the asymmetric cipher can be *initialized for decryption* with a private key.
- the asymmetric cipher can *perform decryption*, once initialized, using only the ciphertext to decrypt.

The asymmetric keypair data structure exists primarily to contain corresponding public and private keys. This grouping is also important because the generation of a keypair is performed as a whole (i.e. the public and private keys are intrinsically linked and must be generated together). The asymmetric keypair data structure must thus provide an interface that allows this consistent generation to be performed using only a source of randomness, and, of course, an interface to extract the public and private keys.

Key packaging, however, isn't necessarily performed through asymmetric encryption of the session key. One alternate method is password-based encryption, where the key is a hashed password known only to the sender and recipient of an encrypted message. While public-key encryption only requires the key packager to literally package and unpackage a session key, password-based encryption's packaging method requires a shared secret password. *Thus, the password-based encryption key packager has an important role to play during the key generation step.* The following can be outlined:

- a key packager can generate a symmetric key using the symmetric-cipher algorithm to help it pick the key. This key generation step involves the key packager so that it may bypass the normal random generation and impose its own method of key generation (via a hashed password, for example).
- a key packager can package a key using the key data, a source of randomness, and possibly a public key. It returns a packaged key.

- a key packager can unpackage this packaged key back into a symmetric key of the right type for the original symmetric algorithm. This may require a private key operation. Again, this process will use the symmetric-key algorithm to properly recover the right symmetric-key format.

2.2.3 Signing and Verification

Signing is quite similar to asymmetric encryption in general, where the actions are reversed: the private key operation is performed on the sender side, and the public key operation on the receiver side. Unlike the key packager, signatures are always done through asymmetric operations. This greatly simplifies the role of the signature algorithm and its interface:

- the signing algorithm can sign a piece of plaintext using a private key and a source of randomness (for potential padding).
- the signing algorithm can verify a signature using the plaintext, signature bytes, and the public key purporting to belong to the signer.

While it is clear that signature algorithms may use hashing first on the plaintext they are signing, this part of the computation is not taken into account, as it does not constitute an inherently necessary part of every signing algorithm. It is clearly possible, at the algorithm-implementation level, to modularize the signing and hashing processes for certain families of signing algorithms. At the level of Self-Describing Cryptography, however, this modularization is unnecessary.

It should also be noted that authentication can be performed in other ways, using Message Authentication Codes (MACs) for example. While this type of authentication is important, it will not be included in this prototype, given that it is used significantly less often in everyday security applications.

As with asymmetric keys used for encryption, it behooves each user to generate their own private key for signing purposes (which could be the same as for decryption purposes). Thus, it is each user's responsibility to choose their signing algorithm, and to generate the appropriate keypair for this signing algorithm. It follows logically that the sender's signing algorithm and public key (e.g. another example of an *augmented public key*) should be sent along with the encrypted message so that the recipient is able to verify the signature

properly. There is also a need for a certificate of the sender’s public key, so that the recipient might be able to trust the binding of the public key to the sender’s claimed identity.

This discussion begs the issue of how much to trust random algorithms communicated to a user via SDC-enabled messages. After all, how is the recipient to trust that a signature algorithm is indeed valid and truly verifies the sender’s identity?

2.3 Trusting Algorithm Code

With interfaces to each type of algorithm defined as above, the framework for Self-Describing Cryptography must clearly define what permissions each piece of code is granted. These issues, brought about by the use of mobile code, have been explored before. The main issue, the assigning of various degrees of trust and permissions to different pieces of a system, was first explored in the Multics Protection Rings [11], an extension of the hardware “supervisor bit”, which selectively grants access to different pieces of code depending on the “protection ring” each one belongs to. Another important aspect of dealing with mobile code, the confinement problem [9], addresses the issue of information leaks from mobile code back to its author, via unexpected, *covert channels*. Finally, there is the intractable problem of verifying that mobile code correctly performs the actions it was meant to perform. Because it would be tantamount to solving the Halting Problem [12], there is no process to automatically verify that a piece of code performs the correct actions. Code can be certified by a trusted entity, however. This certification can be taken as a guarantee of functionality.

In general, there are two solutions that are available to limit algorithm actions:

- *Code Sandboxing*: this technique is only possible under certain conditions relating to runtime system and structure of code. Java provides a valid context for proper code sandboxing [14], which consists of rules added to the runtime system which allow it to block certain code from performing certain operations.
- *Code Certification*: this has always been possible (since public-key cryptography), using any platform and any programming language. The particular algorithms used in a cryptographic operation are certified by some authority the user trusts. No specific restrictions are placed on the way the code runs.

Both of these techniques will be part of the Self-Describing Cryptography framework.

The first will provide basic safety-net security to ensure that entire categories of actions cannot be performed (disk access, network access, etc...). The second will serve to create algorithms that can be trusted at the functionality level, such that ciphers can be trusted to perform truly secure encryptions, and signing algorithms can be trusted to sign and verify data correctly.

The combination of these two techniques will provide much greater security than current off-the-shelf cryptographic software. The certification of code will guarantee proper functionality much like a reputable company's name certifies the functionality of off-the-shelf software. In addition to this, the sandboxing will prevent erroneous code from damaging the user's system, or malicious authorities from accessing users' data.

2.3.1 Simple Certification

An algorithm can be certified using a particular certification algorithm. The certification can be taken to imply that the algorithm is both safe, and performs correctly according to its intended specification (e.g. it really does encrypt the data).

Certification can be done by a certification algorithm, similar to other cryptographic algorithms used in Self-Describing Cryptography. A certification algorithm for other algorithms should certify the following characteristics:

- the algorithm code
- the algorithm type (e.g. Symmetric Cipher, Key Packager, or another Certification algorithm)
- the validity period, meaning a start date and end date.

There is also the possibility of certifying *augmented public keys*. This is very similar to current methods of certifying simple public keys. However, in addition to certifying the public key, the augmented public-key certification would implicitly certify the public-key algorithm as well as the validity of the public key for that public-key algorithm.

2.3.2 Bootstrapping Certification

There is, of course, a serious bootstrapping issue if methods for algorithm certification verification are also algorithms themselves. The user might not trust these algorithms,

meaning that these certification algorithms must themselves be certified! This apparently infinite recursion needs to be bootstrapped with at least one inherently trusted algorithm for certification (which implies a signature) which can then certify any number of other algorithms.

This bootstrapping algorithm doesn't need to be efficient in terms of space or time, but it does need to be as secure as possible. The idea is that this bootstrap can easily be used to certify other, faster, possibly less secure algorithms for a certain period of time. These newly-certified algorithms can then be used for the daily cryptographic needs of the user. The bootstrapping algorithm is only invoked when a major new algorithm needs to be checked, and thus can be highly inefficient, as long as it is secure.

A good bootstrapping algorithm to provide signatures is one that is provably secure under some relatively weak assumption, one that makes as few hardness assumptions as possible. An example of such an assumption is that factoring is hard. This claim has often been used, and has never come close to being proven wrong. If such an algorithm can be used as a bootstrap, the system will probably be secure for decades if not more.

Unfortunately, the bootstrapping algorithm cannot be expected to be forever secure. This means the system needs to allow for a change in the bootstrapping algorithm(s). This should be performed before the bootstrapping algorithm is actually found to be weak (hopefully). The old bootstrapping algorithm can then be used to certify the new bootstrapping algorithm. It should be noted that this operation should happen extremely rarely, and is a sad result of the fact that there is currently no practical, provably secure signature scheme without any assumption of hardness (and there probably never will be such an algorithm).

In addition to this secure certification algorithm, the bootstrap must include a public key that is the root of all SDC certification hierarchies. This public-key is also meant to be inherently trusted, and should be used with the bootstrapping certification algorithm to verify the validity of other algorithms.

Of course, the trick of securely delivering this bootstrap (or of delivering a new bootstrap *after* the old one has been compromised) to every SDC-enabled machine is relatively difficult. Like all security software, the source of the original software has to be highly trusted. There is, however, no way to cryptographically certify this root certification node. An offline delivery (maybe on CD-ROM) is preferable.

2.3.3 Limiting Actions the Algorithm Can Take

Algorithms of all types in the context of Self-Describing Cryptography are not fully trusted.

Thus, they should never be able to:

- access private cryptographic data directly
- read or manipulate local files
- make network connections of any kind

If indirect privileged actions need to be performed by the algorithmic code (e.g. private key operations, network connections for the purpose of downloading a public key from a key server, etc...), they will be delegated to a trusted framework: the *Self-Describing Cryptography framework*. This framework can then perform these privileged actions depending on whether or not the calling code is trusted to request such a privileged action.

Chapter 3

Design of System

The design of this prototype of Self-Describing Cryptography will be very closely linked to the Java platform, language and recently updated security model [8], as Java is the enabling and necessary technology. While the protocols could be made independent from the Java platform, the prototypical aspect of the project leaves this standardization work for a later date.

3.1 Basic Elements

The basic pieces of the system deal with:

- The abstracted algorithm: the `Algorithm` class,
- The self-describing encrypted entity: the `Message` class,
- The basic cryptographic parent classes: symmetric ciphers, asymmetric ciphers, signers, and key packagers.

3.1.1 The Algorithm Class

A Java `Algorithm` object is one that will contain Java code related to one particular algorithm. Since all Java code is contained in classes, an algorithm will contain a set of classes, one of which is the main class that needs to be instantiated to implement the particular algorithm. Thus, the `Algorithm` class requires the following interface:

- ability to add and remove classes from the algorithm

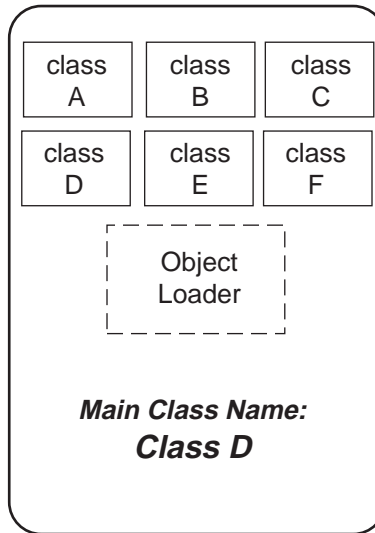


Figure 3-1: An Algorithm Object

- ability to instantiate the main class contained by the algorithm
- ability to convert algorithm to array of bytes and back

The DES Algorithm [13], for example, will be represented by two classes, `DES.class` and `DESKey.class`. The first is the DES algorithm, while the second is obviously the data type of a DES key. The `DES Algorithm` object will thus contain both of these classes, with `DES.class` as its main class.

The full API for the `Algorithm` class is available in the API Appendix.

3.1.2 The Message class

A piece of encrypted data, or data about to be encrypted, needs to be packaged into a data structure that contains all of the relevant, self-describing cryptographic algorithms and information. This data structure will be the `Message` class. The `Message` class should allow:

- signing of the contained plaintext using a signing algorithm, a private key, and a source of randomness,

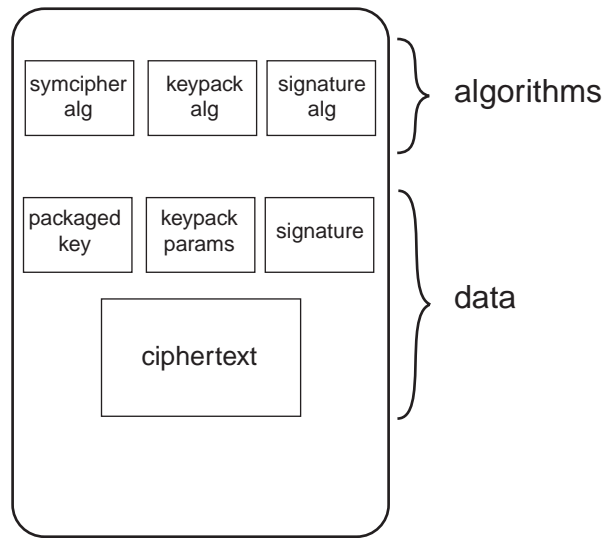


Figure 3-2: A Message Object

- encryption of the contained plaintext and signature using a symmetric cipher, a key packager, and a source of randomness,
- decryption of the contained ciphertext and signature using no additional input (all of the algorithms are still contained from the encryption process), and
- verification of the contained signature on the contained plaintext for the contained public key. Again, this needs no inputs as all of this data is still contained within the `Message` object from the encryption process.

The full API for the `Message` class is available in the API Appendix.

3.1.3 Basic Cryptographic Classes

As defined earlier, there are a few elements that form the backbone of Self-Describing Cryptography by generalizing the different types of cryptographic algorithms. In object-oriented programmatic design, these generalized backbones can be implemented as parent classes that are expected to be subclassed by specific algorithm implementations. The framework code then deals with objects cast to these generic, parent types, while the actual

instances of these objects may override the precise algorithm operations.

The precise APIs of the following classes and interfaces can be found in the API Appendix:

- `Cipher`
 - `SymmetricCipher`
 - `AsymmetricCipher`
- `Signer`
- `KeyPackager`

3.2 Dynamically Loading Code

The key programmatic issue that allows for Self-Describing Cryptography is the ability to have a security application dynamically load code and execute it in a context that allows for direct programmatic interaction with this newly instantiated code. This capability is built into Java with the concept of the `ClassLoader`, a black-boxed way of loading up Java classes.

The technique consists of defining parent classes that define all of the necessary APIs that the framework code needs. For example, the `SymmetricCipher` class is defined with APIs allowing for initialization, encryption, and decryption. The exact method for encrypting and decrypting is not built into the parent class, but the parent class does contain an abstract method that allows the framework code to know about the method without knowing how the method does what it does. Thus, additional classes that will be loaded dynamically during program execution simply need to subclass these existing parent classes such that the compile-time code of the framework can make the proper method calls. These method calls will be overridden at runtime by the specific subclass implementations.

In the case of Self-Describing Cryptography, the atomic element of code is an *Algorithm*, which might, for code modularity reasons, consist of more than just one Java class. Thus, we need to package a number of Java classes into an `Algorithm` object. The classes contained within such an `Algorithm` will be in raw byte array format, ready to be built into fully-formed Java *Class* objects, the building block of the Java language. The SDC framework will pass around these `Algorithm` objects, on disk and over the network. Anytime a

piece of this framework needs to actually instantiate an `Algorithm`, the classes contained within an `Algorithm` will be loaded up into the Java runtime system so that they may be used programmatically to perform cryptographic operations. Java performs this dynamic code loading via a `ClassLoader`, an object with special access to the Java byte-code verifier that builds `Class` objects from raw byte-array class representations. For the purpose of loading the classes from an `Algorithm` object, the SDC framework will define an `AlgorithmClassLoader`, a subclass of Java's built-in `ClassLoader` mechanism. This new class loader's job will be to load an `Algorithm` object and extract, build, and eventually instantiate its contained classes.

3.2.1 Name-Spaces

It becomes immediately clear in this situation that there may be name collision issues. Authors of various algorithms may name their algorithms (or, more probably, their helper classes) similarly. Since most cryptographic processes involve more than one algorithm, the possibility of name collisions and the potential security risks incurred have to be dealt with.

Fortunately, Java defines a separate name-space for each `ClassLoader`. Each algorithm should thus be loaded with its own instance of an `AlgorithmClassLoader`, thereby allowing its classes to take on whatever name they desire. Each algorithm lives in its own name-space, completely isolated from other algorithms the SDC framework might have loaded.

Of course, it can be a relatively bad idea to completely isolate algorithms from one another. Many of these algorithms depend on SDC framework code for base classes, which means that completely separating algorithm name spaces forces the framework code to be reloaded for every single algorithm. Much more dramatic is the resulting incompatibility between *identical* classes loaded by different class loaders. Thus, what is needed is a class loading model that mirrors the class dependency diagram. This can be implemented using Java 1.2's delegation model for class-loading [10]: each classloader can be given a single parent classloader at instantiation, such that the new classloader will check with its parent before loading a class. This allows algorithms to be as compatible with one another as they need to be (e.g. given the class dependencies), while still separating the name spaces for each particular algorithm.

3.3 Security Context API

There are certain operations that algorithms need to take that require disk access, network access, or sometimes even access to private cryptographic information. This type of access cannot be granted directly to the algorithm: access must be regulated by a number of security checks. This is where the Security Context comes in.

The Security Context is an interface that will allow the algorithm to call back to the main security application and request:

- a private key operation,
- a way to retrieve an algorithm's implementation using the algorithm's hash, usually by accessing an algorithm server or checking the local cache,
- a way to retrieve a user's public key for sending an encrypted message to that user, usually by accessing a key server or checking the local cache,
- direct user-interaction

The handling of these operations via a specific API (in this case, a Java object) serves both as a secure guard of these functions, as well as a very useful abstraction so that an algorithm need not know whether the user is interacting via a text interface or a GUI, which algorithms might have been cached, how public keys are looked up, etc... All in all a very useful callback interface.

In Java, such a callback interface will be implemented as a Java Interface, `SecurityContext`, which the main security application which uses SDC should implement. Implementing this interface will involve implementing the 4 or 5 relevant methods. Anytime a SDC operation is then requested, the security application will pass a self-reference to the SDC framework, allowing the algorithms the ability to perform these callbacks correctly.

3.4 Sandboxing the Code

With the callback interface defined to specifically allow the algorithm code to perform specific operations, it is important to shut off all avenues for these potentially malicious algorithms to step outside of their boundaries. Fortunately, the Java 1.2 platform is well equipped to enact this kind of control.

The first important property of the Java runtime system is that potentially dangerous operations can be undertaken only by the Java APIs or Native code (interfaced via the Java Native Interface). Only the Java API (or native code) can access hardware directly. To enable sandboxing, all potentially dangerous Java API calls (filesystem access, network access, and more) perform a check with the Security Manager. The Security Manager, in collaboration with the Access Controller (the apparent redundancy is due to compatibility issues with older Java platforms), then proceeds to check the calling stack and the origin of all the classes present on it. With each class is associated a Protection Domain that indicates to the Access Controller what permissions this class has. If any class on the stack does not possess the proper permission to perform the requested protected action, the Access Controller denies the operation by throwing a Security Exception. This prevents untrusted classes from performing illegal actions by simply calling methods of trusted classes. Only in the case that all classes on the stack are permitted to perform the checked action does the Access Controller allow continued code execution.

By default, classes are not assigned to protection domains, and the security manager is left as a null value: all code has every permission. In the case of Java applets inside a web browser, however, a very constraining security manager is put in place. This is very much the kind of security manager that needs to be implemented in the case of SDC untrusted classes.

Thus, the SDC framework will do a number of things to ensure that untrusted code cannot take risky actions, while the trusted framework is allowed to take any action it needs to:

- the `Algorithm` class will create a Security Manager at load-time, and install it in the Java runtime environment. Having this piece of code inside the initialization of the `Algorithm` class ensures that no `Algorithm` object can be loaded up before the security manager is installed (an object cannot be instantiated until its class has been fully loaded, resolved, and initialized). This piece of code will also set the security policy, ensuring that all local SDC framework code is granted full trust.
- the `AlgorithmClassLoader` can assign a specific `ProtectionDomain` with exactly the right permissions to each class it loads up from the `Algorithm` object it is given. This `ProtectionDomain` will ensure that algorithm classes *cannot* take any of the following

actions:

- access the filesystem
 - access the network
 - access the UI directly
 - create class loaders or access native code (which would indirectly allow the algorithm to bypass its restrictions).
- the `SecurityContext` defined in the previous section needs special permissions, even though it will be called by untrusted code. This is performed by using the Java Security Model's `PrivilegedAction` construct, whereby a class can undertake an action in a privileged way, such that the origin and permissions of the calling classes before it do not impede its actions. This construct comes into play during the Access Controller's stack check described above. As the Access Controller checks up the calling stack (in reverse calling order), it stops its class permission check at the first class whose calling code is contained within a `PrivilegedAction`. Note that this does not allow a class to overstep its set of permissions; it only allows a particular class to confer its set of permissions onto its calling classes. In the Security Context, for example, the algorithm lookup method contains privileged-action code to make a network connection to an algorithm server. The calling code (e.g. the Algorithm) does not have permission to make this network connection by itself, but the action still succeeds because the Security Context guarantees, through the Privileged Action construct, that the action performed is safely limited to a particular functionality, and does not grant a more general, dangerous permission to the calling code.

3.5 The Protocols

With all the basic elements covered, the SDC protocols can be defined. The `SecurityContext` object in SDC-based applications is usually an application-level Java class which makes calls to the SDC library. A `SecurityContext` can be graphical, text-based, or in any other form, as long as it implements the callback methods that the SDC framework needs.

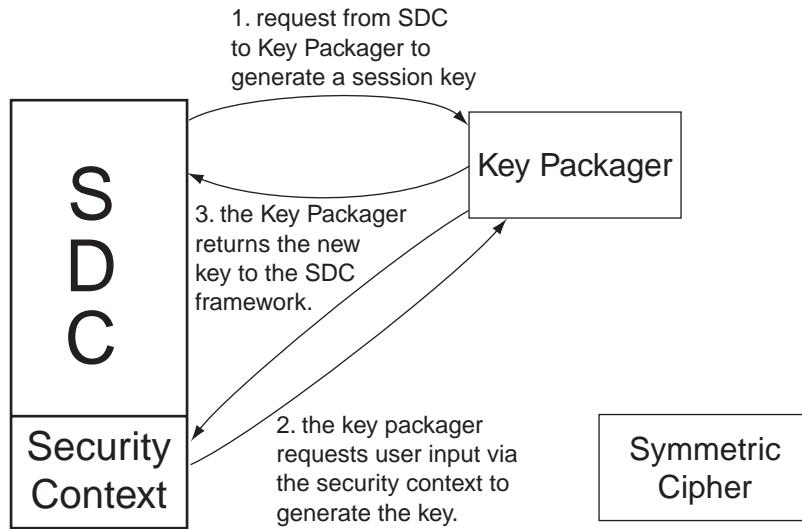


Figure 3-3: Password-Based Key Generation

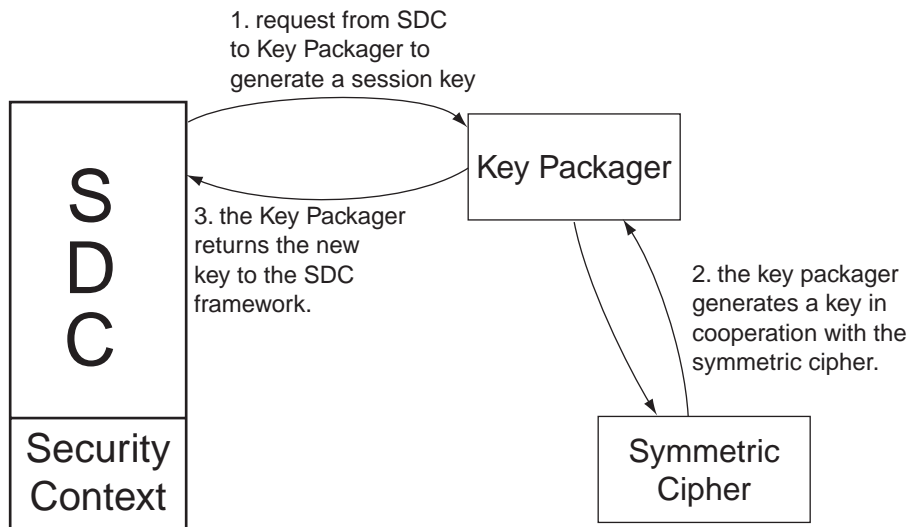


Figure 3-4: Public-Key Key Generation

3.5.1 Sending a Message

- The sender application picks a `Plaintext` object, which can be a string, a file, or anything else that can be represented by a series of bytes.
- The sender application picks a Symmetric Cipher algorithm and a Key Packager algorithm. These algorithms are usually stored on disk, each in a file.
- The sender application creates a new `Message` object in the SDC framework, feeding it the algorithms, plaintext, a source of randomness, and a reference to the application-level `SecurityContext`. The sender then calls the `encrypt()` method on this new `Message`.
- The symmetric cipher and key-packager algorithms are loaded and instantiated by the SDC framework code.
- The SDC framework initializes the key-packaging algorithm with a reference to the symmetric cipher algorithm, a reference to the `SecurityContext`, and a source of randomness. The SDC framework then requests that the key packager generate a symmetric key.
- The key-packaging algorithm uses the resources it gained at initialization time to generate a key. There are two main ways of accomplishing this:
 - *Password-Based Key Packager*
 - * The key packager requests user input via the security context.
 - * This user input is hashed to obtain a series of bytes that can be used to construct a symmetric key.
 - * The symmetric cipher is consulted to check for weak keys, key length, and key type construction. The cipher constructs the key object.
 - * The key packager obtains the empty key object, sets the key data, and returns the newly-created key to the SDC framework.
 - *Public-Key Key Packager*
 - * Since the key data doesn't need to depend on user-known data, the key packager generates random data using the source of randomness provided.

- * As in the password-based case, the key packager interacts with the symmetric cipher object to construct and initialize a new symmetric key.
- * The key packager returns this new key to the SDC framework.

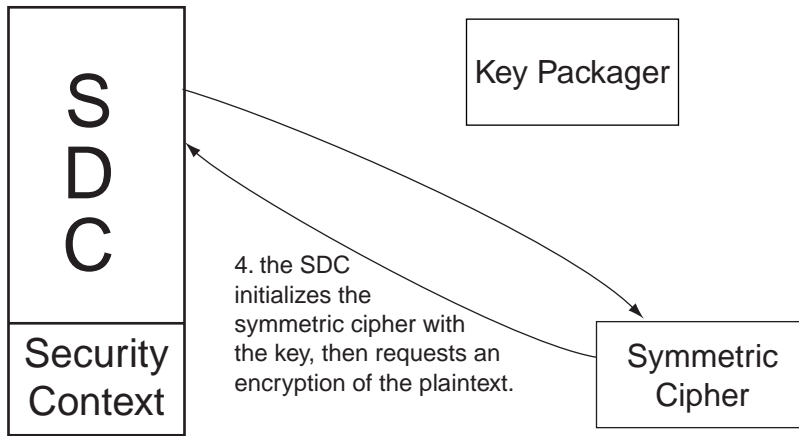


Figure 3-5: Password-Based Encryption

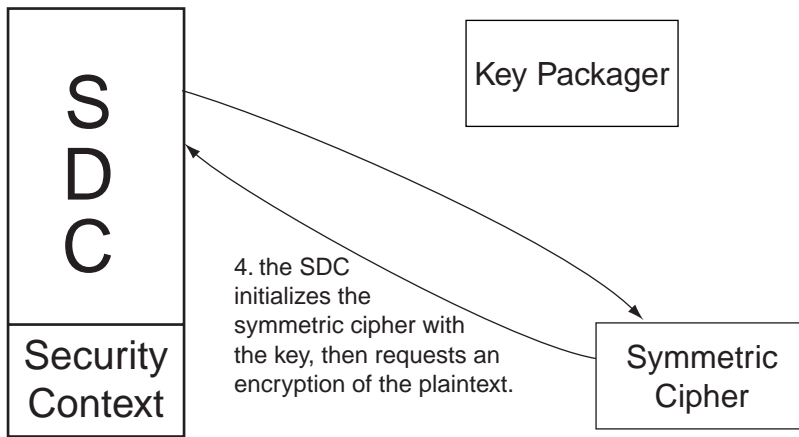


Figure 3-6: Public-Key Encryption

- The SDC framework then requests an encryption of the Plaintext through the symmetric cipher, using the newly generated key and the source of randomness.
- The symmetric cipher returns a **Ciphertext** object.
- The SDC framework stores this **Ciphertext** inside the **Message** object.
- The SDC framework asks the key packager to package the symmetric key.

- The Key Packager uses its resources to perform the key packaging. Looking again at the two main types of key packagers:
 - *Password-Based Key Packager*
 - * There is no information to store since the key will be entirely re-created on the recipient side using only user-input. Thus, the password-based key packager does nothing, and returns a `null` object.
 - *Public-Key Key Packager*
 - * the key packager requests the recipient’s public key from the security context
 - * the security context fetches the public key, either from its cache of known public keys, or from a public-key directory, and returns it to the key packager.
 - * the key packager obtains the appropriate public-key algorithm from the public-key itself, and instantiates this algorithm.
 - * the key packager calls on this new asymmetric algorithm to encrypt the symmetric key for this session.
 - * the key packager takes the encrypted data returned by the asymmetric cipher, and creates a `PackagedKey` from it.
 - * the key packager returns this `PackagedKey` to the SDC framework

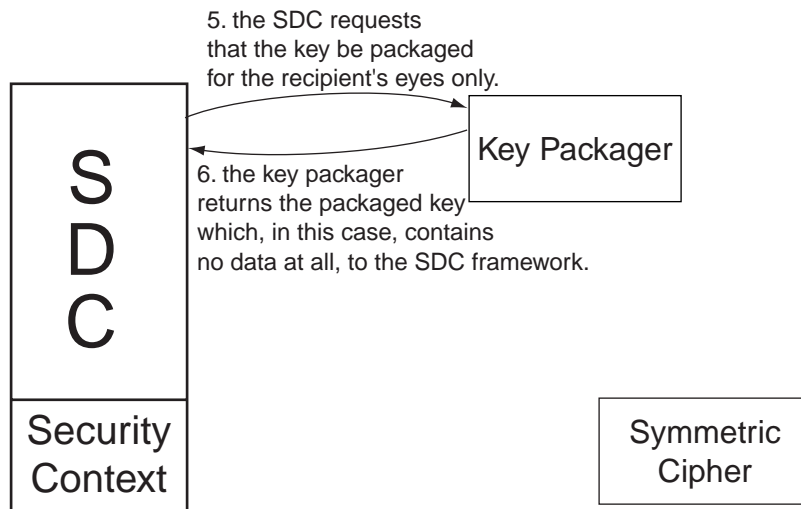


Figure 3-7: Password-Based Key Packaging

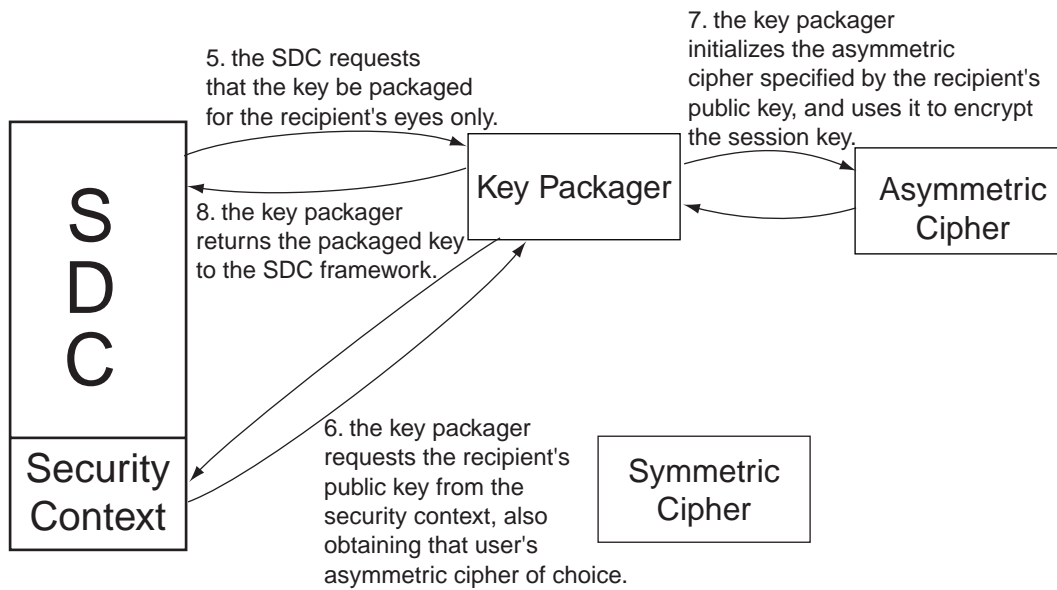


Figure 3-8: Public-Key Key Packaging

- The SDC framework stores the returned `PackagedKey` (which could be null) inside the `Message` object.
- The SDC framework clears all sensitive information from the `Message` object, including the `Plaintext` and the `SymmetricKey`. The only data left is the `Ciphertext` and the `PackagedKey`.
- The SDC framework returns the `Message` to the security application, which can then send it along to the recipient.

3.5.2 Receiving a Message

When an SDC-based application receives a message, all of the necessary information can to be obtained from that message (otherwise, it's not completely self-describing).

- The security application receives the `Message` object, and passes it to the SDC framework, along with a reference to the application-level `SecurityContext`.
- The SDC framework loads the `KeyPackager` algorithm from the `Message` object, and instantiates it.

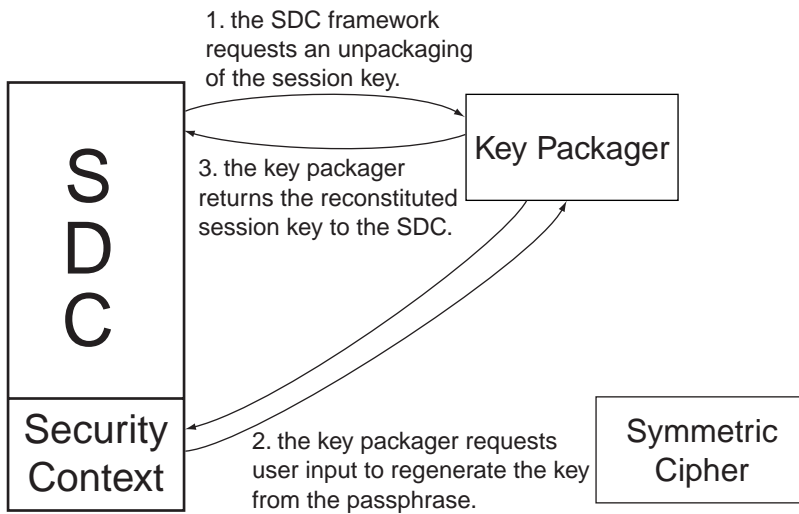


Figure 3-9: Password-Based Unpackaging

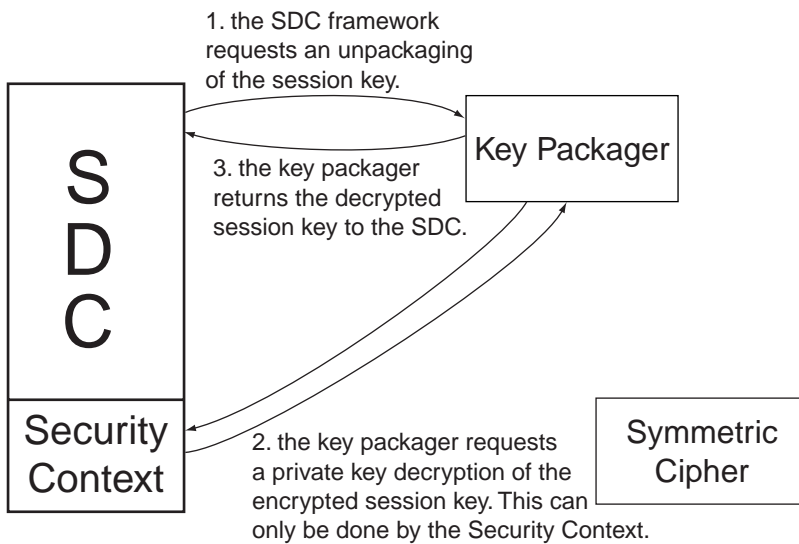


Figure 3-10: Public-Key Unpackaging

- The SDC framework finds the `PackagedKey`, and asks the `KeyPackager` to unpackage it back into a `SymmetricKey`. There are, again, two major ways of approaching this.
 - *Password-Based*
 - * the key packager runs through the same steps it ran through for key generation, by asking for user input, hashing the result to obtain the symmetric key.
 - * the key packager creates the `SymmetricKey` and returns it to the SDC framework.
 - *Public-Key*
 - * the key packager requests a private-key decryption of the packaged key through the security context.
 - * the key packager obtains the decrypted data, creates a `SymmetricKey` from it, and returns it to the SDC framework.
- The SDC framework loads up the `SymmetricCipher` algorithm and instantiates it.

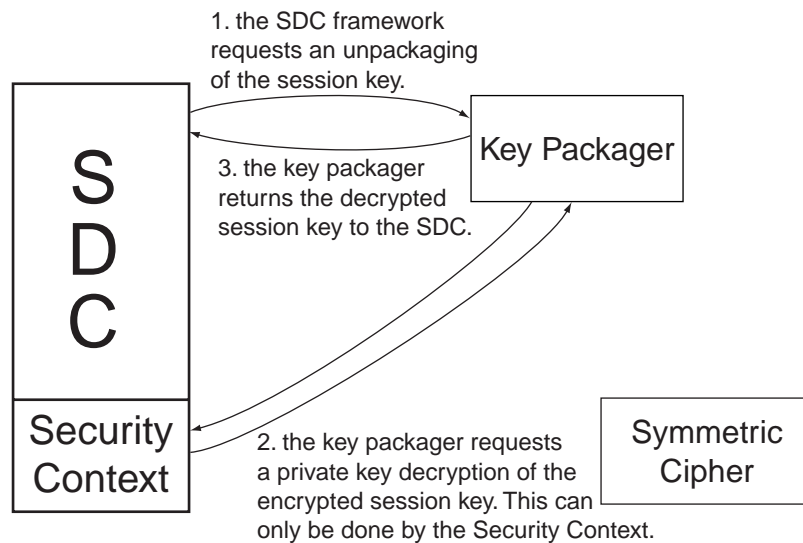


Figure 3-11: Public-Key and Password-Based Decryption

- The SDC framework asks the `SymmetricCipher` to decrypt the `Ciphertext` using the newly-obtained `SymmetricKey`.
- The SDC framework returns the resulting `Plaintext` to the security application.

3.6 Hashed Algorithms

It is clear that sending all algorithms used in an SDC message, every single time, is not a very efficient mechanism. For the purposes of optimizing the system in a significant way (especially for small plaintexts), it is important to send, in place of the actual algorithms, a “fingerprint” of an algorithm. This fingerprint allows the recipient to check her local cache of algorithms, and use the designated algorithm if she already has it.

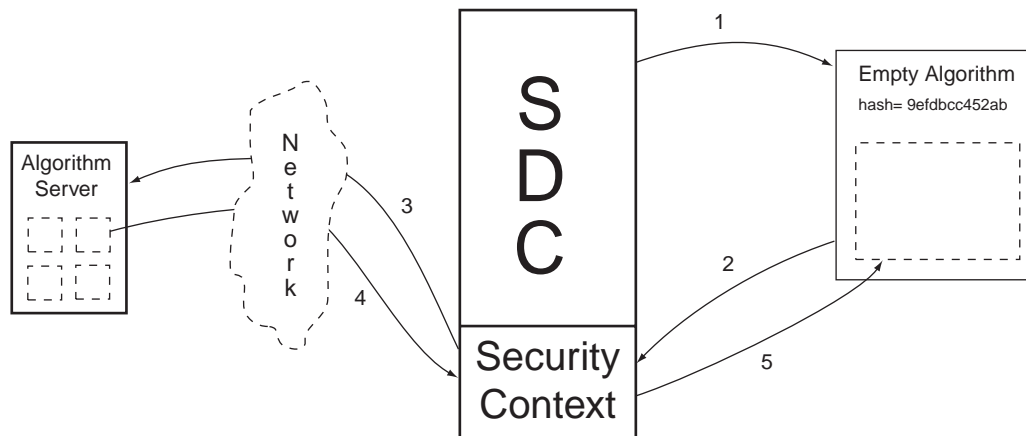


Figure 3-12: The Initialization of an Empty Algorithm

The protocol would then be modified as follows:

- The sender picks algorithms, and performs the encryption, key packaging, and signing of the message.
- Just before the message is sent, its algorithms are “hashed”, meaning that the `Algorithm` object is replaced by an `EmptyAlgorithm`, a placeholder that only contains a SHA-1 hash of the original `Algorithm` object.
- The recipient identifies the `EmptyAlgorithm` objects, and initializes them by replacing them with the proper `Algorithm` objects. This is done by the SDC framework either by checking a local cache of algorithms, or querying an SDC Algorithm Server, a web server whose sole role is to return `Algorithm` objects by SHA1-hash queries.

3.7 Packaging into a Self-Executable

A somewhat tangential piece of Self-Describing Cryptography is the ability to assume that the recipient does not have any piece of the SDC framework. Since SDC already implies sending pieces of executable code, there is nothing that prevents sending the *entire* framework code along. There are a few constraints:

- The recipient must at least have a compatible Java runtime environment. This is assumed to be trivial.
- The received framework code, unlike the base framework code a well-equipped user might have, cannot be trusted to perform filesystem or network access.
- The recipient cannot be expected to have an SDC-enabled private key.

With these restrictions, the only method of sending an SDC message to a recipient who does not possess the SDC framework is to send a message with a password key-packager (which only requires user input), and to send non-hashed algorithms. The recipient will need a Java runtime environment, and will be able to apply any security restriction it wishes to place on the received code.

This added functionality, even with the above-mentioned restrictions, can prove to be extremely useful, because it allows any two users to bootstrap encryption of any piece of data using simply a common secret passphrase.

Chapter 4

Security Model and Attacks

Self-Describing Cryptography relies on mobile code. In any mobile code situation, security is a critical concern. With Self-Describing Cryptography, the mobile code is meant to implement a means for security, which implies that the security concerns are even more important than usual.

In this chapter, the security and trust models of the system are explored to ensure that no security risks have been introduced by the self-describing aspect of the cryptographic library. In addition, a new type of security model is defined, one where mobile code is given more local-access freedom, but less communication freedom.

4.1 The Trust Model

4.1.1 The Entities Involved

There are a few distinct entities to consider, all of which are “active” in the sense that they can execute instructions and thus create potential risk:

- *The SDC Framework code:* This is the code that is the basis for Self-Describing Cryptography. Any user of the system must have this code, and it doesn’t change with time.
- *The bootstrapping algorithms:* These algorithms are stored on disk in a designated “trusted” location. They are not signed, because they are the primordial bootstrap which need to be trusted before any other signing algorithm.

- *The daily-use algorithms:* These algorithms come from an algorithm server, or from an incoming message. They can then be cached locally. They might be signed by someone whom the user trusts to some degree.

4.1.2 The Degrees of Trust

Different pieces of the SDC system can be trusted at different levels. These different levels can be thought of as roles the code is allowed to take on. The following roles are defined:

- *No Trust:* the code is allowed to perform operations on data that is local to it, or data that is explicitly provided to it via a method call. The code is not allowed to create threads, make network connections, read local files, or make any privileged method calls into the framework. This is highly sandboxed code.
- *Minimal, Application-Specific Trust:* Based on the purpose a piece of code is meant to have, this code can, in addition to what fully untrusted code can do, make certain privileged calls to the SDC framework. These calls may be limited to certain methods based on the purpose of the code, and to a certain number of times these methods can be called (calling a method numerous times with different inputs can be used to cryptanalyze private key data). Note that this type of code may, in some ways, indirectly access the filesystem, the network, or other restricted resources, but only via restricted and controlled means of the SDC framework.
- *Full Access Trust:* This code can perform any action it desires, with direct access to filesystem, network, private cryptographic data, and all other restricted resources. This is the code that is sometimes accessed by the minimally-trusted code for privileged actions.

4.1.3 Entities and Their Level of Trust

The only piece of code in the system which is given full-access trust is the SDC framework code, the code that never changes and that should be trusted by the user as the basic enabling skeleton of Self-Describing Cryptography. This is the code through which all privileged actions will pass, and which will ensure that proper trust constraints are placed on other pieces of code.

The minimally-trusted code is a much more subtle area to consider. Many algorithms of the SDC system fit in this category, where, depending on the algorithm's purpose, certain permissions are granted and others are withheld:

- Any algorithm might be in hashed format, whereby it needs to call back to the SDC framework to request the unhashed version of the algorithm.
- Symmetric-Cipher algorithms never need any additional permissions beyond this empty-algorithm resolution need.
- Key-Packaging algorithms may need to request a recipient's public key at encryption time, and request a private key operation at decryption time.
- Signing algorithms never need any access beyond what is provided to them as parameters.

Thus, in general, most algorithms need very few permissions.

4.2 Security Attacks on SDC

The attacks on Self-Describing Cryptography can happen at a number of levels, given that the potentially offending code runs within the same programmatic framework as the rest of the system. For that reason, it is best to first analyze attacks at the programmatic level, and work towards the more classic protocol and cryptographic attacks.

4.2.1 Programmatic Attacks

A programmatic attack is one where a programmatic trick is used by malicious code to gain additional permissions, access, or generally perform actions it shouldn't normally be allowed to perform.

For the purposes of this research, Java language implementation bugs will not be considered. The Java platform may have its bugs, but they will be fixed over time and eventually produce a fairly secure computing platform. In a normally-functioning Java environment, there is perfect type safety, and no memory-related issues that plague C programs (e.g. buffer overflows). These types of attacks can thus be ignored.

The avenues for programmatic attacks come from the fact that some parts of the code are privileged, while others are not. Attacks at this level will attempt to cross the limit from unprivileged to privileged:

- *Calling Static Methods In Privileged Code:* This attack consists in calling static methods in classes loaded with privileged access. Given that static methods can be called directly by fully classified name, without needing any object reference to instances of the class (or to the class object), performing this attack is quite easy. For this attack to succeed in performing actions beyond the original code's ability, however, there must exist static methods that include Java `PrivilegedAction` code. Without this, the malicious code can never overstep its designated permissions because it is intimately tied to its `ClassLoader`, a connection the `AccessController` will detect in its calling-stack check.

In the SDC framework, there are no static methods that use the `PrivilegedAction` construct. Thus, there is no way for malicious code to overstep its bounds in this manner.

- *Creating Privileged Objects:* This attack consists in instantiating objects whose classes are privileged in order to perform privileged actions through them. Because Java performs permission checks on the calling stack at runtime, this attack is, again, only valid if a method of the privileged object uses the `PrivilegedAction` construct, which grants calling classes temporary permissions equal to the privileged class.

Given that there are a few methods in the SDC framework which perform Privileged Actions, these attacks are serious potential threats. Any SDC-framework method that performs a `PrivilegedAction` is expecting to be called by a particular untrusted piece of code at some point, but it may not be expecting to be called by *any* untrusted piece of code: it may be expecting to be called only by untrusted code which is explicitly given a reference to the privileged object. However, unless the constructors of these privileged objects are themselves protected (which would be onerous to include in every privileged constructor), this assumption cannot be made.

Therefore, any method inside a privileged piece of code must explicitly check the calling code for a particular application-specific `Permission` before it performs a `PrivilegedAction`.

- *Adding New Permissions:* If calling privileged code fails, the next attack is clearly to become privileged by gaining additional permissions. This attack inherently fails under the constraints of the Java platform, because the set of permissions that a class is given is set when the Class is instantiated, and cannot be changed at any later point in the execution of the code.
- *Loading New Code:* The last resort in this realm of attacks is then to load entirely new classes of code that may have different, more empowering permissions. The first method for malicious code to accomplish this is to use the ClassLoader used to load the malicious code itself, and ask it to load an additional class with a different `ProtectionDomain`. In the SDC framework, this kind of attack is thwarted by the fact that specifically limited Class Loaders are used to load up untrusted code. These Classloaders impose their `ProtectionDomain` on any class they load up.

Another attack then obviously consists of creating an entirely new `ClassLoader` with less limiting policies on Class permissions, and then using this new `ClassLoader` to execute malicious code suddenly granted with additional permissions. To prevent this latest attack, the SDC framework does not grant `ClassLoader` creation permissions to code that is not fully trusted, given that this would immediately enable it to gain any other set of permissions. Only fully trusted code should be able to create new class loaders.

Thus, through a combination of Java built-in security and carefully-crafted privileged code sections, the SDC framework patches the potential security holes at the programmatic level.

4.2.2 Protocol Attacks

Protocol attacks on Self-Describing Cryptography are performed within the limits of the designated protocols, using the information and interfaces provided by the framework. The malicious code tricks the system into revealing secret information, compromising private cryptographic keys, or performing any other such clearly unwanted action, all via perfectly valid interactions with the system.

This set of attacks is directly related to the inherent SDC setup where semi-trusted code has access to sensitive data. During encryption or decryption, the symmetric cipher has

access to the session key and to the plaintext. At key packaging or unpackaging, the key packager has access to the session key. Thanks to the tight constraints placed on the actions this semi-trusted code can take, there is no direct way for this sensitive information to be communicated to the malicious party in the case these algorithms are indeed malicious. Filesystem and network access are forbidden.

However, one must consider the potential *covert channels of communication* that may exist in the SDC framework. These covert channels present a plausible avenue for attacks, where a malicious algorithm can communicate the sensitive data it encounters back to a malicious party in a way that the SDC framework didn't expect would be used as a means of communication.

Given that the potentially malicious code is heavily regulated in its actions, the only way to covertly communicate information out of the system is through the regular programmatic interface, where a method belonging to the malicious code has the opportunity to return a piece of data that will, in some mischievous manner, find its way back to the malicious party.

The main way to perform this communication is through the `Plaintext` that the symmetric cipher returns to the SDC framework when it is asked to decrypt the `Ciphertext`. This plaintext will usually be shown directly to the recipient, often as a simple piece of text (e.g. an email). This piece of text could very easily be used to trick the recipient into taking an action that compromises the private information previously locked within the symmetric cipher object. The following attack could prove dramatically dangerous:

- Alice sends an SDC-formatted email to Bob using a public-key key packager.
- Eve captures the encrypted email off the network, and copies the packaged session key.
- Eve sends an SDC-formatted email to Bob with a malicious symmetric cipher, and the same packaged key data and key packager as was present in the message from Alice to Bob
- When Bob receives Eve's message, the malicious symmetric cipher captures the decrypted session key information at initialization time. It then performs no actual decryption, but instead simply returns a claimed `Plaintext` that reads "point your

browser to the following URL: `http://hacker.com/XgcVVDDF`” where the URL contains an encoding of the session key.

- While Bob might refuse to follow the advertised URL, it’s highly likely that he will (having no reason not to trust the message). Following the URL will then release to the malicious party the session key of the message Bob received from Alice.

This type of attack will succeed against the SDC framework, under the assumption that the malicious symmetric cipher algorithm is somehow certified by a trusted authority. While it is somewhat difficult to believe that a trusted authority would certify such clearly malicious code, it is certainly not a possibility that can be ignored.

This security attack, as well as all other attacks relying on the `Plaintext` as a covert channel, can be countered by the plaintext-commitment defense. Another possible defense against this attack is to link the symmetric key to its proper symmetric key algorithm, in the same way that public keys are linked to their public key algorithm. These defenses are detailed in Appendix D.

4.2.3 Cryptographic Attacks

Cryptographic attacks on Self-Describing Cryptography are performed within the limits of available protocols, but with intimate knowledge about the cryptographic properties of the algorithms. Valid cryptographic operations can be used to cryptanalyze private key data, or to fake other cryptographic operations that should not have been performed. All of these cryptographic attacks center on the ability for untrusted code to request from the trusted SDC framework a private key operation.

The first such attack is one where the requested decryption is used as a means to perform a chosen-ciphertext attack with one chance to decrypt. Certain algorithms might see their private key weakened by performing the decryption of a particular ciphertext. The classic fix to this type of problem is to require that all valid plaintexts encrypted with a particular public key be formatted in a discernable way, usually using OAEP padding [6]. This allows the algorithm to check this format validity when the requested decryption is performed, and to return the operation’s result only when it is a valid plaintext. While the SDC framework doesn’t mandate the use of encryption schemes that perform OAEP padding, it is quite easy to include this added security in the algorithm itself. To ensure this added security, the

SDC Algorithm Implementation Guidelines should strongly recommend OAEP padding (or any other “plaintext-aware” encryption scheme) for public-key algorithms (See Appendix for the SDC Algorithm Implementation Guidelines).

Another important attack, using the ability of malicious code to request private-key decryptions, is to request numerous private-key decryptions in order to perform cryptanalysis of the recipient’s private key. While OAEP padding does prevent this “Lunchtime Attack” from succeeding extremely well, it is not a foolproof defense. The SDC framework must take precautions to never allow a particular algorithm to request more than one private-key operation in its lifetime. If a message needs to be decrypted again for whatever purpose, the algorithm should be expected to have cached its result. If for whatever reason it fails to do so, a new, raw version of the algorithm should be instantiated so that the first instance of the algorithm cannot accumulate cryptanalytic information about the recipient’s private key by performing adaptive attacks at each decryption it is allowed to request.

4.3 The No-Way-Out-Sandbox Security Model

The above analysis approaches the security model problem from a very classical angle. It becomes apparent, though, throughout this detailed security probing, that any attack that involves discovering secret information (rather than just blindly corrupting local data) is in fact a serial process:

- The malicious party somehow obtains some secret information.
- The malicious party makes use of this secret information, either by storing it, or using it to perform other cryptographic operations, the result of which is stored.

The typical approach is to completely block off the first step in this serial process, assuming that the second step is trivial for a malicious party to perform. Yet, with a secure computing platform like Java, this second step in the attack is no longer necessarily trivial. It becomes possible to focus security efforts on preventing malicious code from making any use of secret information obtained, instead of preventing malicious code from ever obtaining this information.

This approach allows mobile code to be more tightly integrated with local data, possibly even with private cryptographic data. There is a new, stronger guarantee that this now-

integrated code will not be able to make any further use of the private data it obtains, because it cannot communicate the data back to a malicious party or even make direct use of this data in a way that has any effect on the outside environment. As soon as this mobile code is done with the actions it was meant to perform, the trusted framework can simply destroy the sandboxed environment within which the code was confined.

There is no reason, however, to take chances with this approach and therefore relax all protections against malicious code possibly obtaining secret information. If secret information can be completely protected, then it should remain fully-protected. The only point to consider from this new approach is that the situation is not desperate if the malicious sandboxed code cannot be completely prevented from learning private information. The idea of this *No-Way-Out Sandbox* approach is that by considering the system as a whole, more flexible solutions can be established without compromising the overall security of the system.

Chapter 5

Implementation Details

The Self-Describing Cryptography code libraries were implemented using Java 1.2 (otherwise known as Java 2). While this is currently a relatively new technology, with some lesser known features and APIs, it provides much more finely-grained sandboxing than previous Java versions, a feature that is critical to a fully-functional SDC implementation.

The entire code-base for the SDC prototype is somewhat too large to include in its entirety (10,000 lines of Java code). The important, non-trivial pieces of the implementation are detailed below.

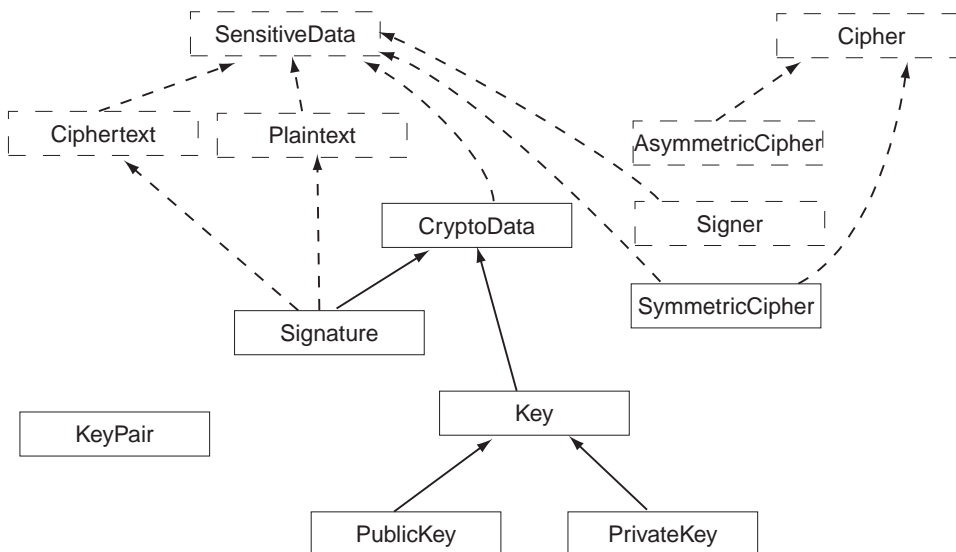


Figure 5-1: SDC Class Hierarchy

5.1 Java Class Dependencies

The Java platform is built on the principle of late binding. Late binding implies that various modules of code are linked at runtime, on an execution-need basis. If class A is first needed twenty minutes into program execution, it is conceivable, in certain Java programs, that it will only be loaded precisely when it is needed, and at no point in the preceding twenty minutes.

This is a great advantage in terms of development and program execution flexibility. It is, in fact, this very ability to load code at any time during program execution that enables a clean and simple implementation of Self-Describing Cryptography. Algorithms can be loaded when a message is received, and bound into the already-running code seamlessly.

This late binding isn't without certain complications, however. One particular algorithm, for example, might be composed of more than just one Java class (in fact, it is highly likely, if the algorithm is well written and modular, that this is the case). All `Algorithm` objects are specified as containing all of the classes that are needed to run a particular algorithm. Actually performing the packaging of the classes into an `Algorithm` object, however, is relatively complicated, given that the linking of the necessary Java classes is only done at runtime.

This problem is also apparent when needing to create a fully-contained SDC self-decrypting Java archive file: figuring out which of the SDC classes are needed to make a functioning self-decrypting system also implies finding the dependencies between various classes.

The way Self-Describing Cryptography solves this problem is by forcefully determining the dependency tree of a Java class. This can only be accomplished by a piece of code that has direct access to the byte-array representation of all Java classes in the system, which can be done if filesystem access to all the classes is available. Given a class for which the system needs to find all the necessary support classes, the dependency-tree walker functions as follows:

- The root class is loaded and parsed to obtain its constant pool, method signatures, and fields.
- Any time a new Java class is referenced in the parsed data, it is added to the pool of “support classes.”

- Any support class that hasn't been parsed is loaded and parsed, exactly like the root class, and all of its support classes are also added to the global list.

At the end of this tree-traversal of the dependencies, the global list contains all of the classes necessary to execute any method in any of the classes listed, and especially any method of the root class.

The astute reader will notice that Java has alternate ways of loading classes that may trick this dependency tree-traversal system. These ways include using the `Class.forName()` method call, where a `String` is used to load up a class whose fully-qualified name is the one specified in the `String`. The interesting counterpoint to this is that no compile-time system can figure out this kind of class dependency, which explains why `Class.forName()` throws a checked exception. This means that any code which uses this runtime-dependent way of loading a class must contain error-code in the case that class is not found. Because this error handling must exist within the code itself, the tree-traversal dependency finder need not bother with this kind of complication. It is also extremely unlikely that a cryptographic algorithm would find a use for this technique of loading a class.

5.2 Proper Isolation of Algorithms

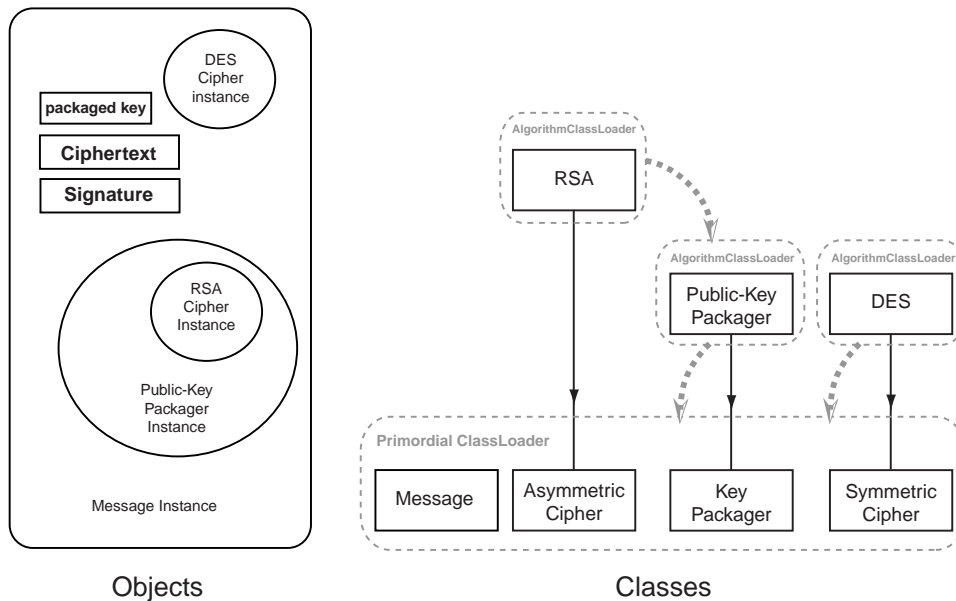


Figure 5-2: Object, Class, and ClassLoader Dependencies

As noted in the design section of this research, different algorithms involved in one

cryptographic exchange may very well be programmed by completely separate entities, each simply conforming to the published Algorithm APIs. Because of this distributed development, different algorithms may very well have identical names, especially their helper classes. Furthermore, different algorithms need to be assigned different permissions. In general, it is highly desirable that code specific to an algorithm be isolated from other pieces of the system. The only method of interaction should be through the algorithms' parent classes and the APIs these parent classes define.

5.2.1 Name Spaces: the `ClassLoader` object

Thankfully, Java allows class code to be clearly separated into name spaces. This name-space separation is performed using a `ClassLoader`, a Java object with very specific properties that allow it to:

- run a byte-array through the Java byte-code verifier to check for legal Java classes.
- create a valid `Class` object from a verified byte-array.
- associate a fully-qualified name with a `Class` object in such a way that no two `Class` objects within the same `ClassLoader` can have the same name.
- assign a `ProtectionDomain` to a newly loaded `Class` in a way that cannot be changed after the class has been loaded.

One `ClassLoader` object therefore defines its Java-class name space, within which one fully-qualified class name corresponds to one class exactly. It seems logical, then, that each Algorithm will be loaded using its own `ClassLoader`, which must be able to load classes specifically from an `Algorithm` object. This is, as described in the Design section, the `AlgorithmClassLoader`.

5.2.2 Name-Space Delegation

The SDC design chapter covered the issue of creating an individual name space for each algorithm in order to prevent name collisions. The resulting complication of incompatible SDC base types was also discussed, as was the idea of creating a hierarchy of name spaces.

The code solution to creating this name-space hierarchy is Java 1.2's `ClassLoader` delegation architecture. Each `ClassLoader` can designate, at creation time, a parent

`ClassLoader` object, which is consulted before any new class is loaded by the child class loader. What this means is that a class already loaded by the parent class loader will not be loaded again by the child class loader. This allows a number of children class loaders to share access to certain base classes in their parent class loader. It is precisely this architecture that is needed for SDC: each algorithm lives in its own `AlgorithmClassLoader`, which is set to initially delegate class searching to the primordial class loader (Java's built-in, bootstrap `ClassLoader` used to load up all of the basic and necessary Java language classes).

In addition to simply using Java 1.2's class loading delegation mechanism, the SDC system must ensure that SDC-framework classes are present in the primordial `ClassLoader` name space. Even with the delegation model, there is a possibility that an SDC-framework class is loaded at a late point in the game, specifically by algorithm code located inside an `AlgorithmClassLoader` name space. Thus, the `AlgorithmClassLoader` needs to perform even more delegation than is mandated by the Java 1.2 architecture: any time a class is loaded by an `AlgorithmClassLoader`, the loading of the class must be delegated to the parent class loader whenever the class requested is not specifically included in the original `Algorithm` object that the `AlgorithmClassLoader` was initialized with. This allows all SDC-framework classes to trickle down to the primordial name space, while maintaining that any algorithm-specific classes are isolated from the rest of the system.

5.2.3 Loading Serialized Objects

This isolation of algorithms into their own `AlgorithmClassLoader` creates an additional, tricky complication when it comes to reconstituting Java objects from byte-array format.

Starting with Java 1.1, there is a built-in way in the Java language to serialize a Java object into a sequence of bytes, provided that the Java object's class implements the `Serializable` interface (serializability is not available by default on all objects for obvious security reasons related to accessing an object's private fields). The tricky and often-glossed-over piece of this technology is the reconstitution step, where the Java system takes a byte-array and creates an Object from it:

- the serialization system checks for valid serialized format and throws an exception if the format is in any way incorrect.

- the object's class name is found, and, if this class isn't yet available in the current Java name space, it is loaded up.
- the local class is hashed, and the hash is compared to the serialized object's class version ID. If they don't match, the reconstitution fails. If they do match, a new instance of the object is created, and the data from the byte-array is used to populate the object's fields.
- All of the main object's fields are reconstituted in the same way. If any one of them fails reconstitution, the overall object's reconstitution fails, too.

The complication here lies in two parts of this protocol:

- The piece of code that loads the serialized object must immediately have access, in the same name space, to the reconstituted object's class, as well as to all of the classes for the object's fields.
- The newly-loaded classes will be loaded by the current `ClassLoader`, which may be completely wrong for the Java program's purpose.

This difficulty presents itself because the SDC framework code, which lives inside the primordial name space, performs the network and filesystem operations instead of granting the algorithms direct access to these resources. The SDC framework will, for example, load up the user's private key to perform private key operations, or load up a recipient's public key for packaging the session key of a message to that recipient. In both of these cases, the SDC framework may very well not know about the particular key types of these keys, as the type may be specific to the algorithm the key is meant for.

The solution is to delay the reconstitution of the specific key type until the specific algorithm has been loaded up and is ready to add classes to its own name space. Two constructs will be needed to make this happen:

- *the ObjectLoader*: sometimes, code needs to load up an object into a different name space. This happens when the public-key key-packager algorithm loads up the packaged key, but needs to have this packaged key created within the public-key algorithm name space, and not within the key-packager name space. Thus, it is useful for every `AlgorithmClassLoader` name space to contain a utility meant only for loading

classes: the `ObjectLoader` class. This class will be artificially inserted into every `AlgorithmClassLoader` at instantiation time, so that the `AlgorithmClassLoader` can easily reconstitute an object with new classes loaded up using the `AlgorithmClassLoader` itself. The `ObjectLoader` is a name-space hack, and the `ObjectLoader` class is loaded individually within each name space.

- *the SDEKeyPair*: the reconstitution of certain key types needs to be delayed beyond the point when the keypair is read from the filesystem or network. This delay is implemented in the `SDEKeyPair` class, whose sole role is to encapsulate a byte-array representation of a `KeyPair` object, and the `Algorithm` object which is meant to load up that `KeyPair`. Thus, when an `SDEKeyPair` is reconstituted, only the byte-array representation of the `KeyPair` object is loaded, without any attempt to find that serialized object's class. Then, when an `Algorithm` wants to make use of this `SDEKeyPair`, it forces the reconstitution of this object by using the `ByteArrayInputStream` construct, and layering an `ObjectInputStream` on top of it. The serialized bytes are then reconstituted into an `Object` only when the code explicitly requests it, allowing for proper deserialization in the correct name space, where the necessary classes will be available.

These two constructs allow the SDC system to delay object reconstitution, as well as to assign a name space into which an object should be reconstituted in. With this added flexibility of control over object reconstitution, the complete isolation of algorithms remains fully workable within the constraints of the Java language.

5.3 Applying Security Constraints to Java 1.2 Code

The last SDC-specific problem that needs to be solved is the application of the Java 1.2 fine-grained security model. Each algorithm that is loaded into the system must be constrained to very few permissions, including only particular application-specific permissions. The SDC-framework code, however, must maintain the ability to perform any programmatic action.

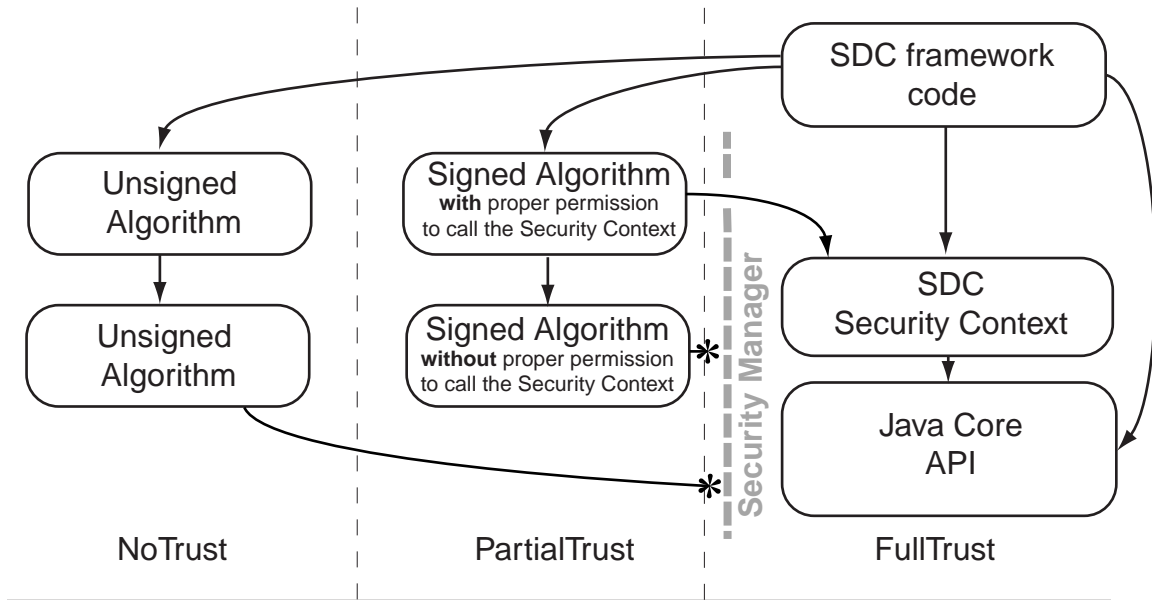


Figure 5-3: The Security Manager regulates access to the Java APIs and to the SDC Security Context

5.3.1 System-Wide Policy

The system must set up a `Policy`, an object whose sole role is to associate a `PermissionCollection` object (i.e. a set of `Permission` objects) to each `ProtectionDomain`.

The central idea is that a `ProtectionDomain` defines an abstract notion of a level of trust. Given the design which includes three levels of trust, the SDC application will defined three different protection domains: `FullTrust`, `PartialTrust`, and `NoTrust`. The `Policy` is responsible for associating the following permissions:

- **FullTrust:** code is granted the `AllPermission` permission, which allows it to do anything it wants.
- **PartialTrust:** code is granted a combination of `PrivateKeyPermission`, `UserInputRequestPermission`, `PublicKeyRequestPermission`. The particular permissions granted to the code depend on whether the type of algorithm in question ever needs to perform the kind of operation protected by the permission in question.
- **NoTrust:** code is granted no permission at all.

5.3.2 Taking PrivilegedAction

When a piece of SDC framework code is used indirectly by less than fully-trusted code, it is usually so that the fully-trusted SDC framework code can confer its fully-trusted status onto the less-trusted code for the duration of a particular action. These particular actions include, for example, fetching a recipient's public key by making a network connection to a public-key server.

The way fully-trusted code can confer its permissions on calling code is by using the `PrivilegedAction` Java construct. A `PrivilegedAction` allows the code to package a method's worth of code and run it with the permissions associated with the `Class` that makes the `PrivilegedAction` call.

Before the trusted-code method takes this `PrivilegedAction`, though, it must check that the calling code has the proper permission. All of the methods that perform these actions will be included within the `SecurityContext` interface. For each of these methods, there needs to be an application-specific permission (`PrivateKeyPermission`, `UserInputRequestPermission`) that regulates access to the particular privileged action, acting as a guard to allow only certain less-trusted code to indirectly perform the privileged action.

5.3.3 Assigning the ProtectionDomain

With all of the basic permission constructs defined as above, the last step is to assign the proper `ProtectionDomain` to each class, and to set up the proper system-wide `Policy`.

First, it is important to ensure that the proper `Policy` is set up before any `Algorithm` object is loaded and instantiated. Instead of requiring any SDC-enabled executable to set up this policy, the SDC library will set up the `Policy` by itself in the static initializer of the `Algorithm` class. Because a static initializer is guaranteed to be run before any instances of the class's object (as per the Java spec), the `Policy` is guaranteed to be installed successfully before any risk of untrusted code grabbing permissions it shouldn't gain (as described in the Design section).

The other important idea is to ensure that all `Algorithm` classes are assigned to the right `ProtectionDomain`. Signed algorithms are assigned to the `PartialTrust` domain, while unsigned algorithms are assigned to the `NoTrust` domain. Signed algorithms, being part of

the `PartialTrust` domain, are granted some application-specific permissions depending on whether they are symmetric ciphers, asymmetric ciphers, signers, or key packagers. All of these assignments are simply controlled by the `AlgorithmClassLoader`, which checks the signature on the algorithm, and, if it is considered valid, then instantiates the `Algorithm` only to check which basic SDC type it subclasses, thus granting it the appropriate permissions.

The `AlgorithmClassLoader` can easily impose the proper `ProtectionDomain`, because it controls all loading of new classes belonging to `Algorithm` objects.

Chapter 6

Conclusion and Future Work

There's no question that a working implementation of Self-Describing Cryptography, an algorithm-independent approach to cryptographic processes, is useful to both developers and users of security-enabled applications.

By clearly defining the programmatic abstraction layer between the cryptographic process framework and the specific algorithms, SDC allows developers to focus on a much narrower, more precise region of security software development. For the end-user, the amount of security knowledge needed to attain a reasonable level of security and privacy in SDC-enabled applications is greatly minimized: upgradability and compatibility issues are significantly addressed.

In short, Self-Describing Cryptography creates a framework for more secure cryptographic applications.

The more important point this research has made, however, is that Self-Describing Cryptography can be built! Simply citing the advantages of a potential technology is perfectly futile unless this technology is indeed viable and possible to implement. This document shows that, thanks to the various safety and platform-independence features of Java, truly-secure mobile code can be a reality.

Self-Describing Cryptography is obviously useful. This document has shown that it is also feasible.

Furthermore, this research introduces a somewhat new approach to analyzing a security model. Classic security models tend to focus on perfect cryptographic schemes that rarely reflect practical complications, leaving that piece of the puzzle to the software engineer. To

obtain truly practical security, however, a cryptographic system design needs to take into account these practical complications. Not only do such considerations make the system more secure, but they allow for increased flexibility in the secure software by integrating all elements of a potential attack, and analyzing all the potential defenses that can be set up to prevent these attacks. The “No-Way-Out Sandbox” approach is an example of just such an integrated security model approach. Practically preventing malicious code from using obtained private data is just as important and useful as preventing the malicious code from obtaining the private data in the first place.

The design and implementation of Self-Describing Cryptography demonstrates the advantages of taking a more global, less segmented approach to defining the security requirements of a piece of software.

6.1 Why Is SDC Important?

Making Self-Describing Cryptography work has a greater importance than simply improving certain modules of cryptographic software. The world of software, in general, is quickly moving towards online delivery, with mobile pieces of code downloaded and uploaded as often as data is.

With software progressively blurring the line between executable code and passive data, it is of the utmost importance to adapt security software to this trend. Regardless of what security experts would prefer, data documents now include executable code: Microsoft Word’s macro language is a very good example of this, while the numerous resulting macro-viruses clearly reveal the resulting complications. Unfortunately, the current world of security protection works only at preventing harmful software from ever running at all. It is of the utmost importance to accept that executable code, some trusted, some not, will be passed around from machine to machine with little if any security measures taken in the process. Operating systems and all application software need to take this into account. Measures need to be taken such that this sharing of executable code can be done safely and easily.

Self-Describing Cryptography provides an example system that addresses this difficult issue of safe execution of untrusted code by attacking the most complicated piece of the puzzle: executing possibly untrusted code whose intended purpose is precisely to protect

the user's privacy. If Self-Describing Cryptography can be accomplished, then other mobile code issues can be resolved, too.

6.2 Future Work

This research can expand into two directions. The first will be to focus on the original goal which motivated Self-Describing Cryptography: ease-of-use of security applications. There are numerous complicated aspects of security applications, each of which pushes users away from using security, simply because few users are truly concerned with making security work until they are the victim of a security breach themselves. One of these issues is rethinking the way public-keys are available for users to send encrypted email, and the way these public-keys are certified. The user should generally be expected to answer only simple questions, such as "do you trust RSA Data Security to tell you who is who?" Security is not only about algorithms and protocols, but also about usability. It's important to explore the usability issues in conjunction with all of the classic security considerations.

Another important direction to take after this research is towards a generally more secure computing platform. Almost all application data is now being outfitted with executable data, and internet data is mixing in with personal data on users' machines. There needs to be a generic way for a user to know what to trust, for applications to sandbox their files, and generally for creating a more secure, organized, trusted computing platform. This would constitute further research into more integrated security models, where practical and theoretical issues alike are considered in defining potential attacks and defenses.

Appendix A

Algorithm Implementation Guidelines

The Self-Describing Cryptography framework relies on many individuals implementing algorithms in their own way. Algorithms can be loaded on-the-fly, sent across the network, or saved as files. In order to work within the SDC framework, however, they need to follow certain APIs and guidelines.

A.1 Cipher, Plaintext and Ciphertext

In order to abstract out the very basic concepts of encryption, the base classes `Cipher`, `Plaintext`, and `Ciphertext` are provided.

The `Cipher` class is provided to simply define a generic way of encrypting and decrypting things. The `Cipher` construct makes no distinction between public-key or symmetric-key cryptography: it assumes that there is a way to encrypt something, and a way to decrypt the result back into the original thing.

The next issue consists in defining these *things* that are encrypted and decrypted. A plaintext will be represented by a subclass of the basic `Plaintext` class, while the corresponding ciphertext will be represented by a subclass of the basic `Ciphertext`. These base classes exist to abstract out the format of what is being encrypted and decrypted. Different subclasses of `Plaintext` and `Ciphertext` can implement file plaintexts, where the file is simply referenced instead of actually contained (which is a requirement for large files). Similarly, one could imagine network streams abstracted out as `Plaintext` or `Ciphertext`.

This allows any type of data to be encrypted properly, with subclasses of `Plaintext` and `Ciphertext` implementing the specific mapping between the data and the raw bytes to encrypt and decrypt.

When implementing an algorithm to perform encryption or decryption, it may be necessary to implement associated version of the `Plaintext` and `Ciphertext`. However, it is highly likely that the built-in plaintext and ciphertext objects will already provide the necessary functionality.

A.2 Symmetric and Asymmetric Ciphers

Creating an actual cipher is the crux of the problem. Doing this is quite simple with the current version of the SDC APIs, which are meant to remain as simple as possible.

A.2.1 Symmetric Ciphers

At least two classes need to be implemented to build a Symmetric Cipher:

- a class that represents the cipher and that extends `sde.crypto.SymmetricCipher`.
- a class that represents the key and that extends `sde.crypto.SymmetricKey`.

The class that is the symmetric cipher can have parameters, stored in the `CryptoParameters` class, so that one class can be used to implement variants of an algorithm. Currently, this facility is unused, and isn't completely necessary to get a fully working system. The methods that need to be implemented are as follows:

- `getKeyObject()`: return an empty instance of the proper key object. This key object is the second class to implement.
- `init(SymmetricKey key, CryptoParameters params)`: set up the algorithm with the symmetric key and parameters. The parameters can be safely ignored for now. The `SymmetricKey` should be checked for the right type, and an exception can be thrown if there is a casting problem.
- `encrypt(byte[] plaintext, SecureRandom srand)`: perform encryption. This will be called immediately after the `init()` method. The `SecureRandom` parameter is

there in case the cipher needs a source of randomness for any level of probabilistic encryption.

- `decrypt(byte[] ciphertext)`: perform decryption. This will be called immediately after the `init()` method.
- `clear()`: clear out any sensitive data that is contained within the `SymmetricCipher` object.

The `SymmetricKey` class is relatively simple, and simply needs to define how it can be created at random, with the following methods that need to be implemented:

- `generate(SecureRandom srand)`: generate itself from the random data generator given to it.
- `clear()`: destroy any sensitive information that was stored.

A.2.2 Asymmetric Ciphers

While asymmetric ciphers are never directly used by the SDC framework, they are common enough that they should be standardized. They are somewhat similar to symmetric ciphers, except for the initialization step.

Implementing an Asymmetric Cipher requires implementing four different classes:

- a class that represents the asymmetric cipher and that subclasses `sde.crypto.AsymmetricCipher`.
- a class that represents the keypair and that subclasses `sde.crypto.KeyPair`.
- a class that represents the public key and that subclasses `sde.crypto.PublicKey`.
- a class that represents the private key and that subclasses `sde.crypto.PrivateKey`.

At the base, the `PublicKey` and `PrivateKey` classes are only wrappers for the `javax.security` versions of those types. Subclasses need not do much, generally, but can extend them in any way desired depending on the algorithm. In general, though, these types are only data containers, with no methods except selectors and constructors.

The `KeyPair` class, on the other hand, must implement at least one key method:

- `generate(SecureRandom srand, CryptoParameters params)`: create the internal `PrivateKey` and `PublicKey` in a consistent way, using the given source of randomness and any parameters (like key length) which would be contained in the `CryptoParameters` object.

Finally, the crux of the asymmetric cipher is, of course, the class which implements the `AsymmetricCipher` interface, and which must therefore include the following methods:

- `getKeyPairObject()`: return a `KeyPair` object which is of the correct type for this algorithm, but which contains no key data.
- `init(PublicKey publicKey)`: initialize the algorithm for encryption with the given public key.
- `init(PrivateKey privateKey)`: initialize the algorithm for encryption with the given private key.
- `encrypt(byte[] plaintext, SecureRandom srand)`: perform encryption of the given plaintext, with the given source of randomness for any probabilistic needs. This method is called immediately after the public-key initialization method.
- `decrypt(byte[] ciphertext)`: perform decryption of the given ciphertext. This method is called immediately after the private-key initialization method.
- `clear()`: clear out any sensitive data that is contained within the `AsymmetricCipher` object.

Given that an `AsymmetricCipher` will be called upon to perform private-key decryption for semi-trusted code, it is important to prevent ciphertext-only attacks. To help in preventing these attacks, it is highly recommend to implement an asymmetric cipher with OAEP padding, or with some other type of plaintext-awareness. This will allow the asymmetric cipher to detect invalid decryptions, in which case a security exception can be raised.

A.3 Key Packers

Making a key packager is possibly the most flexible part of the system, given that the key packager's actions range from prompting the user for data to simply performing a public-key encryption.

In general, a key packager should know about the `SecurityContext` API, which can be found in the API Appendix. If it makes use of public key algorithms, it should know about the `Algorithm` API, and about the `AsymmetricCipher` API as defined above.

Implementing a Key Packager can be done with as little as one class, which must extend the `sde.protocol.KeyPackager` base class. There are a few methods which must be implemented to make this abstract base class real. None of these methods are called directly from other classes: they are all internal to the `KeyPackager` class. Furthermore, they can all reference a number of protected instance fields which are necessary to performing the operations the methods were meant to provide:

- `symmetricCipher`: the symmetric cipher instance that is to be used for performing the plaintext encryption.
- `myParams`: the parameters for key packaging. This is an object of type `KeyPackagingParameters`, and may include, for example, the recipient's email address, or username, or other fingerprint enabling a proper public-key search.
- `securityContext`: the API to the security application for privileged actions, such as performing a private key operation, requesting user input, etc...
- `key`: the symmetric key to be used for plaintext encryption.
- `packagedKey`: the packaged symmetric key, storing the data processed by the key packager, and used for key unpackaging.

The methods that must be implemented are as follows:

- `doKeyGeneration(SecureRandom srand)`: performs the key generation. This method may assume that the `key` field has been instantiated to the right type, but does not contain any actual key data. This may very well ask for user input to generate the key, or call upon the key object itself to generate itself, using the `SecureRandom` object it was given.
- `doKeyPackaging()`: performs the key packaging. This method may assume that the `key` object is fully instantiated and set, and that the `packagedKey` object has been instantiated, but is waiting to be filled. This method may also assume that the `key`

field will be appropriately cleared of its data once the packaged key has been set correctly.

- `doKeyUnpackaging()`: performs the key unpackaging. This method may assume that the `packagedKey` object is fully instantiated and set to the right packaged-key data. It may also assume that the `key` field is instantiated to the right type, but doesn't contain any useful data. Finally, this method may also assume that the `packagedKey` data will be cleared after the `key` has been fully reconstituted.

A.4 Signing Algorithms

Signing algorithms are somewhat similar to asymmetric ciphers, but not entirely the same. Note also that `AsymmetricCipher` and `Signer` are both interfaces, so that a programmer might use the same class to implement both signing and asymmetric cipher abilities (see `sde.basic.RSA` for an example of this).

To build a signing algorithm, one needs the `PublicKey`, `PrivateKey`, and `KeyPair` classes, similar to the `AsymmetricCipher` requirement. On top of that, a class that implements the `Signer` interface must be written, with the following required methods:

- `init(PrivateKey privateKey)`: initialize for signing with the given private key.
- `init(PublicKey publicKey)`: initialize for verification with the given public key.
- `sign(byte[] plaintext, SecureRandom srand)`: sign the given plaintext, using possibly the source of randomness given if there is a probabilistic need. This method returns a `Signature` object. It is guaranteed to be called immediately after the private-key initialization method.
- `verify(byte[] plaintext, Signature signature)`: verify the given signature on the given plaintext. This method returns a boolean indicating the status of the verification. It is guaranteed to be called immediately after the public-key initialization method.

A.5 Using the SimpleAlgorithmMaker

Once an algorithm has been written by a developer, it needs to be packaged into an `Algorithm` object, which can then be written to disk as a single file, ready to be used by an SDC-enabled system.

This operation is currently performed by the command-line `SimpleAlgorithmMaker`, which simply prompts the user for a number of class files to load up. When the developer is done adding class files, the `SimpleAlgorithmMaker` prompts her to choose the *main* class for this algorithm. The main class for an algorithm is the class that implements that algorithm's main functionality. For example, a symmetric cipher's main class is the one that extends the `SymmetricCipher` base class. Once this is picked, the system will create an `Algorithm` object containing the designated classes, and will write it out to a file designated by the developer.

Appendix B

Plaintext Covert Channel Security

Fix

B.1 The Problem

The most significant problem by far with the current prototype of Self-Describing Cryptography has to do with the covert channel available through the `Plaintext` return object. A `SymmetricCipher` could return a bogus `Plaintext` object to the SDC framework which would somehow trick the user into taking a certain action that would indirectly (and unbeknownst to the user) reveal some information about secret cryptographic data to a malicious party.

One example of this is specifically described in the Security section of this document, where a message is sent to a user in order to trick her into decrypting a previously used session key. The malicious `SymmetricCipher`, having access both to this session key and control over the behavior of the `decrypt()` method returning the `Plaintext` object to the user, can then return a `Plaintext` to the user prompting her to visit a particular web site, the URL of which contains the encoded session key.

In general, the problem can be described as follows:

- a malicious algorithm learns private information through some cryptographic process on the recipient's side, usually by reusing data that has been sent to the recipient before in a different message, from a different sender.
- this algorithm then needs to somehow communicate this information, and thus ar-

ranges for the `Plaintext` to contain some data that represents the newly-learned information.

- the user is tricked by this `Plaintext` to perform some action that reveals the private data to the malicious party.

B.2 The Basic Solution

The basic inconsistency in the above scheme is that the `SymmetricCipher` is able to adjust the `Plaintext` object *after* the private information has been learned. It is almost impossible to prevent the `SymmetricCipher` from learning some amount of information about the recipient's private cryptographic data. However, it is critical to prevent this same `SymmetricCipher` from adjusting any of its return values to leak any of this newly-learned information.

The key idea to solving this problem is to realize that, in a non-malicious situation, the `Plaintext` object is known to the sending party. This difference between the malicious and non-malicious case allows a tweak in the protocol that forbids a malicious algorithm from adapting the `Plaintext` to include newly-learned data. The incoming `Message` should present, before any cryptographic operation is performed, a *cryptographic commitment* to what the `Plaintext` object will be. This commitment can be sent in the clear, because the properties of cryptographic bit-commitments are such that the commitment itself does not reveal anything about the data being committed to. However, the commitment is such that the `SymmetricCipher` can only present the real `Plaintext` that was meant to be shown, and not one that is created after the commitment has been communicated to the SDC framework. In the case that the SDC framework obtains a `Plaintext` that does not match the commitment it was given, it simply raises a security exception, and refuses to show the obtained `Plaintext` to the user.

Forcing a bit-commitment on the `Plaintext` prevents the `SymmetricCipher` from modifying this `Plaintext` to communicate any newly-gained information. This solution completely closes the door on using the `Plaintext` as a covert channel, under the assumption that the bit-commitment scheme is cryptographically strong.

B.3 The Consequences for Self-Describing Cryptography

The main complication obtained from modifying the protocol in this way is that this bit-commitment algorithm must come from somewhere! It cannot be communicated by the sender, otherwise the problem remains when this bit-commitment scheme simply cooperates with the symmetric cipher to trick the SDC framework into thinking the `Plaintext` is valid.

Thus, it seems that this bit-commitment algorithm should be an additional bootstrapping algorithm in the SDC framework, on top of the one signature algorithm used to certify all others. This bit-commitment scheme should be directly certified by the bootstrapping signature scheme, by a highly-trusted authority. While highly secure, this commitment scheme should also be relatively efficient, as it will be used in every single transaction of the SDC.

Appendix C

sde.crypto

<i>Package Contents</i>	<i>Page</i>
Interfaces	
Signer	70
Ciphertext	71
Cipher	72
Plaintext	72
AsymmetricCipher	73
Classes	
SymmetricCipher	74

C.1 Interfaces

C.1.1 *Interface* Signer

The parent class for all algorithms that perform signing operations

Declaration

```
public abstract interface Signer
```

Methods

- *init*
public void init(sde.crypto.PrivateKey **privkey**)
 - Usage
 - * initialize the algorithm for signing, using the given private key
 - Parameters
 - * **privkey** - the private key to perform signing
 - Exceptions
 - * sde.crypto.BadKeyException - possible cryptographic problem
- *init*
public void init(sde.crypto.PublicKey **pubkey**)

- Usage
 - * initialize the algorithm for verification
- Parameters
 - * **pubkey** - the public key to use for verification
- Exceptions
 - * `sde.crypto.BadKeyException` - possible cryptographic problem
- *sign*

```
public Signature sign( sde.crypto.Plaintext plaintext,
  java.security.SecureRandom srand )
```

 - Usage
 - * perform a signing operation on a plaintext
 - Parameters
 - * **plaintext** - the plaintext to sign
 - * **srand** - a source of randomness
 - Returns - the signature
 - Exceptions
 - * `sde.crypto.CryptoException` - possible cryptographic problem
- *verify*

```
public boolean verify( sde.crypto.Plaintext plaintext,
  sde.crypto.Signature s )
```

 - Usage
 - * perform verification operation on the given plaintext and signature
 - Parameters
 - * **s** - the Signature
 - * **plaintext** - the plaintext
 - * **pk** - the public key to check
 - Returns - whether or not the signature is correct
 - Exceptions
 - * `sde.crypto.CryptoException` - possible cryptographic problem

C.1.2 Interface Ciphertext

A way to generalize on all types of ciphertext.

Declaration

```
public abstract interface Ciphertext
implements java.io.Serializable
```

Methods

- *decrypt*

```
public Plaintext decrypt( sde.crypto.Cipher cipher )
```

 - Usage

- * This decrypts the Ciphertext. It allows for complete abstraction of how the ciphertext chooses to be represented as bytes (multiple arrays of bytes... or one, etc...)
- Parameters
 - * `cipher` - the cipher algorithm to use for decryption. This must be initialized properly with the right key
- Returns - the decrypted, corresponding plaintext
- Exceptions
 - * `sde.crypto.CryptoException` - if an error occurs in the cryptographic process

C.1.3 Interface Cipher

This is the base class for all algorithms that perform cipher-like operations, meaning encryption and decryption

Declaration

```
public abstract interface Cipher
```

Methods

- *encrypt*

```
public byte encrypt( byte [] plaintext, java.security.SecureRandom
srand )
```

 - Usage
 - * this performs encryption of a plaintext byte array.
 - Parameters
 - * `plaintext` - the plaintext to encrypt
 - * `srand` - the source of randomness
 - Returns - the byte array that is the ciphertext
 - Exceptions
 - * `sde.crypto.CryptoException` - crypto error
- *decrypt*

```
public byte decrypt( byte [] ciphertext )
```

 - Usage
 - * decrypt the given ciphertext bytes
 - Parameters
 - * `ciphertext` - the ciphertext to decrypt
 - Returns - the plaintext byte array
 - Exceptions
 - * `sde.crypto.CryptoException` - error

C.1.4 Interface Plaintext

A way to generalize on all types of plaintext, so that the internal representation of the plaintext never comes into play. (Currently a small exception to that rule with the "toByteArray()" method

Declaration

```
public abstract interface Plaintext
implements java.io.Serializable
```

Methods

- *encrypt*

```
public Ciphertext encrypt( sde.crypto.Cipher cipher,
    java.security.SecureRandom srand )
```

 - Usage
 - * perform the encryption of this plaintext.
 - Parameters
 - * **cipher** - the cipher to use for encryption. This should be initialized with the proper key
 - * **srand** - the source of randomness to use for encryption
 - Returns - the corresponding ciphertext
 - Exceptions
 - * **sde.crypto.CryptoException** - possible cryptographic errors
- *toByteArray*

```
public byte toByteArray( )
```

 - Usage
 - * converts the Plaintext to one byte array. This is used only for specific purposes, and ideally this will be deprecated soon.
 - Returns - the single byte array rep of the Plaintext

C.1.5 Interface AsymmetricCipher

The parent class for all asymmetric ciphers, meaning ciphers that support public-key encryption

Declaration

```
public abstract interface AsymmetricCipher
implements Cipher
```

Methods

- *init*

```
public void init( sde.crypto.PublicKey pubk )
```

 - Usage
 - * initialize the cipher for encryption, with the appropriate public key
 - Parameters
 - * **pubk** - the public key to use for encryption
 - Exceptions
 - * **sde.crypto.BadKeyException** - if the provided key is bad

- *init*

```
public void init( sde.crypto.PrivateKey privk )
```

 - Usage
 - * initialize the cipher for decryption, with the appropriate private key
 - Parameters
 - * **privk** - the private key to use for decryption
 - Exceptions
 - * `sde.crypto.BadKeyException` - in case the key is bad
- *getKeyPairObject*

```
public KeyPair getKeyPairObject( )
```

 - Usage
 - * In order to get true abstraction even when multiple independent classes form an algorithm, there must be a way to get the rightly-typed keypair from an asymmetric cipher. That function is fulfilled by this method.
 - Returns - the keypair of the right type for this cipher
- *encrypt*

```
public byte encrypt( byte [] plaintext, java.security.SecureRandom srand )
```

 - Usage
 - * This can be invoked once the cipher is initialized for encryption, and it performs the complete encryption of a series of bytes.
 - Parameters
 - * **plaintext** - the plaintext
 - * **srand** - a source of randomness
 - Returns - the ciphertext
 - Exceptions
 - * `sde.crypto.CryptoException` - whatever error might happen
- *decrypt*

```
public byte decrypt( byte [] ciphertext )
```

 - Usage
 - * This can be invoked once the cipher has been initialized for decryption. It performs the full decryption process on the provided ciphertext
 - Parameters
 - * **ciphertext** - the ciphertext to decrypt
 - Returns - the plaintext
 - Exceptions
 - * `sde.crypto.CryptoException` - usually when the ciphertext does not decrypt correctly, because it is not a valid ciphertext

C.2 Classes

C.2.1 Class SymmetricCipher

The parent class for all symmetric ciphers, meaning all ciphers where the key is the same for encryption and decryption.

Declaration

```
public abstract class SymmetricCipher
implements Cipher
```

Constructors

- *SymmetricCipher*
public SymmetricCipher()

Methods

- *generateParameters*
public CryptoParameters generateParameters(java.security.SecureRandom sr)
– Parameters
* sr -
- *init*
public abstract void init(java.security.Key key, sde.crypto.CryptoParameters params)
– Usage
* initialize the algorithm for encryption or decryption
– Parameters
* key - the key to use
* params - the parameters
– Exceptions
* sde.crypto.CryptoException - possible cryptographic problem
- *init*
public void init(java.security.Key key)
– Usage
* initialize the algorithm for encryption or decryption. No parameters given, they're assumed to be null.
– Parameters
* key - the key to use
– Exceptions
* sde.crypto.CryptoException - possible cryptographic problem
- *getKeyObject*
public abstract SymmetricKey getKeyObject()
– Usage
* In a fully abstracted system, the framework cannot know how to create the right type of key for a given algorithm. This method allows the rightly-typed key to be created, knowing only the cipher object to be used.
– Returns - a rightly-typed symmetric key, usable for this cipher, but empty.

- *encrypt*

```
public abstract byte encrypt( byte [] plaintext )
```

- Usage

- * perform encryption. Can be done only after initialization

- Parameters

- * **plaintext** - the plaintext bytes

- Returns - the ciphertext bytes

- Exceptions

- * `sde.crypto.CryptoException` - possible cryptographic problem

- *decrypt*

```
public abstract byte decrypt( byte [] ciphertext )
```

- Usage

- * perform decryption. Can only be done after initialization

- Parameters

- * **ciphertext** - the ciphertext bytes

- Returns - the plaintext bytes

- Exceptions

- * `sde.crypto.CryptoException` - possible cryptographic problem

- *clear*

```
public abstract void clear( )
```

- Usage

- * clear out sensitive information

Appendix D

sde.protocol

<i>Package Contents</i>	<i>Page</i>
Interfaces	
SecurityContext	77
Classes	
KeyPackager	78
SDEKeyPair	81
EmptyAlgorithm	81
Message	85

D.1 Interfaces

D.1.1 *Interface* SecurityContext

The Security Context API which gives a privileged action interface to algorithm code that needs to perform specific actions they cannot accomplish by themselves, for abstraction and security reasons.

This is a callback interface.

Declaration

```
public abstract interface SecurityContext
```

Methods

- *doPrivateKeyDecrypt*
public Plaintext doPrivateKeyDecrypt(sde.crypto.Ciphertext c)
 - Usage
 - * requests a private key decryption of some data
 - Parameters
 - * `c` - the ciphertext to decrypt
 - Returns - the plaintext returned
 - Exceptions
 - * `sde.protocol.NoPrivateKeyException` - if the callback interface does not have access to a private key, this action cannot take place.

- *requestUserInput*

```
public String requestUserInput( java.lang.String message )
```

 - Usage
 - * requests user input to continue
 - Parameters
 - * **message** - the prompt message
 - Returns - the user input
 - Exceptions
 - * `java.lang.Exception` - if a problem occurs during prompting
- *getAlgorithm*

```
public Algorithm getAlgorithm( java.lang.String hexAlgorithmHash )
```

 - Usage
 - * perform an algorithm lookup, either from the cache, or from an algorithm server.
 - Parameters
 - * **hexAlgorithmHash** - the hash used as a query.
 - Returns - the Algorithm from the lookup
 - Exceptions
 - * `sde.algorithm.AlgorithmNotFoundException` - if the algorithm cannot be found, this exception is thrown.
- *getPublicKey*

```
public SDEKeyPair getPublicKey( java.lang.String emailAddress )
```

 - Usage
 - * perform a public-key lookup, either from cache or from public-key server
 - Parameters
 - * **emailAddress** - the email address for whom we want a public key
 - Returns - the augmented public key
- *getSecureRandom*

```
public SecureRandom getSecureRandom( )
```

 - Usage
 - * get a source of randomness
 - Returns - the source of randomness.

D.2 Classes

D.2.1 Class KeyPackager

The class that performs generic Key Packaging of the session key. This class intervenes at 3 steps in the SDC process: - key generation - key packaging - key unpackaging

Declaration

```
public abstract class KeyPackager
```

Constructors

- *KeyPackager*

```
public KeyPackager( )
```

- Usage

- * create a simple key packager. This constructor exists so that subclasses with no-arg constructors can exist, too, to enable simple `.forClass().newInstance()` calls.

- *KeyPackager*

```
public KeyPackager( sde.crypto.SymmetricCipher scipher,  
sde.protocol.SecurityContext scontext,  
sde.protocol.KeyPackagingParameters params )
```

- Usage

- * create a key packager with the proper symmetric cipher, security context and parameters for initialization.

- Parameters

- * `scontext` - the security context
- * `scipher` - the symmetric cipher
- * `params` - the key packaging parameters

Methods

- *init*

```
public void init( sde.crypto.SymmetricCipher scipher,  
sde.protocol.SecurityContext scontext,  
sde.protocol.KeyPackagingParameters params )
```

- Usage

- * perform the initialization

- Parameters

- * `scontext` - the security context
- * `scipher` - the symmetric cipher
- * `params` - the params

- *generateKey*

```
public synchronized SymmetricKey generateKey(  
java.security.SecureRandom srand )
```

- Usage

- * generate a symmetric key. This is only called after initialization

- Parameters

- * `srand` - a source of randomness

- Returns - the symmetric key of the right type for the symmetric cipher

- Exceptions

- * `sde.protocol.KeyPackagingException` -

- *doKeyGeneration*
`protected abstract void doKeyGeneration(java.security.SecureRandom
srand)`
 - Usage
 - * perform internal key generation. This is the method that subclasses override.
 - Parameters
 - * `srand` - a source of randomness
 - Exceptions
 - * `sde.protocol.KeyPackagingException` - if there is a problem that prevents packaging.
- *packageKey*
`public PackagedKey packageKey()`
 - Usage
 - * package the key into a `PackagedKey`.
 - Returns - the packaged key
 - Exceptions
 - * `sde.protocol.KeyPackagingException` -
- *doKeyPackaging*
`protected abstract void doKeyPackaging()`
 - Usage
 - * The internal method that performs the raw key packaging. This is the method implemented by subclasses.
 - Exceptions
 - * `sde.protocol.KeyPackagingException` - if the packaging goes wrong
- *unpackKey*
`public SymmetricKey unpackKey(sde.protocol.PackagedKey pk)`
 - Usage
 - * unpack the key. This can only be done after initialization.
 - Parameters
 - * `pk` - the packaged key
 - Returns - the unpacked key.
 - Exceptions
 - * `sde.protocol.KeyPackagingException` -
- *doKeyUnpackaging*
`protected abstract void doKeyUnpackaging()`
 - Usage
 - * perform the raw key unpacking. This is implemented by subclasses.
 - Exceptions
 - * `sde.protocol.KeyPackagingException` - if unpacking goes wrong.
- *clear*
`public void clear()`
 - Usage
 - * destroy any sensitive data contained as state.

D.2.2 Class SDEKeyPair

This is the Augmented Public Key, the key that is packaged with the algorithm. For object loading reasons, the key object is in the form of a byte-array. This prevents immediate deserialization when an augmented public-key is loaded, as described in the Implementation description.

Declaration

```
public class SDEKeyPair
extends java.lang.Object
implements java.io.Serializable
```

Constructors

- *SDEKeyPair*

```
public SDEKeyPair( sde.protocol.KeyPair kp, sde.algorithm.Algorithm
alg )
```

 - Usage
 - * create an augmented public key using an algorithm and a keypair object. The keypair object will be serialized to a byte array and stored in that form.
 - Parameters
 - * **alg** - the asymmetric algorithm
 - * **kp** - the keypair

Methods

- *getAlgorithm*

```
public Algorithm getAlgorithm( )
```

 - Usage
 - * get the asymmetric-cipher algorithm from the augmented public key.
 - Returns - the asymmetric algorithm
- *getSerializedKeyPair*

```
public byte getSerializedKeyPair( )
```

 - Usage
 - * get the serialized keypair. This allows the calling code to now reconstitute the key type as it should be.
 - Returns - the byte-array representation of the augmented keypair object.

D.2.3 Class EmptyAlgorithm

An algorithm which contains only a hash, and which, when initialized, will request the unhashed version it references to be loaded. The Empty Algorithm then takes on that newly-loaded algorithm's identity

Declaration

```
public class EmptyAlgorithm
extends sde.algorithm.Algorithm
```

Constructors

- *EmptyAlgorithm*
`public EmptyAlgorithm(java.lang.String hexHash)`
 - Usage
 - * Create an empty algorithm with the given hash
 - Parameters
 - * `hexHash` - the hash of the algorithm this empty algorithm references

Methods

- *init*
`public void init(sde.protocol.SecurityContext scontext)`
 - Usage
 - * Initialization for an empty algorithm implies requesting from the Security Context the non-hashed algorithm identified by the hash contained within the empty algorithm. No operations can be called on an empty algorithm before initialization.
 - Parameters
 - * `scontext` - the security context to request algs from
 - Exceptions
 - * `sde.algorithm.AlgorithmNotFoundException` - if the security context cant find the alg, this exception is thrown
- *getName*
`public String getName()`
 - Usage
 - * get name from contained algorithm. This method just forwards the call to the contained algorithm.
 - Returns - the name of the algorithm
- *getAuthor*
`public String getAuthor()`
 - Usage
 - * return the author of the contained algorithm. This just forwards the method call to the contained Algorithm.
 - Returns - the author
- *addClass*
`public void addClass(java.lang.String name, byte [] bytes)`
 - Usage

- * add a class to the contained algorithm. This just forwards the method call to the contained Algorithm.
- Parameters
 - * **bytes** - the class bytes
 - * **name** - the class name
- *getClasses*

```
public byte getClasses( )
```

 - Usage
 - * gets the classes from the contained Algorithm. This just forwards the method call to the contained Algorithm.
 - Returns - the classes in byte-array format
- *getClassNames*

```
public String getClassNames( )
```

 - Usage
 - * returns the contained Algorithm's class names. This just forwards the method call to the contained Algorithm.
 - Returns - the class names
- *setMainClassName*

```
public void setMainClassName( java.lang.String mcn )
```

 - Usage
 - * sets the contained Algorithm's main class name. This just forwards the method call to the contained Algorithm.
 - Parameters
 - * **mcn** - the main class name
- *getMainClassName*

```
public String getMainClassName( )
```

 - Usage
 - * returns the main class name from the contained Algorithm. This just forwards the method call to the contained Algorithm.
 - Returns - the main class name
- *lock*

```
public void lock( )
```

Methods inherited from class sde.algorithm.Algorithm

(in E.1.1, page 89)

- *init*

```
public void init( sde.protocol.SecurityContext scontext )
```

 - Usage
 - * initialize the algorithm such that if anything needs to be done before the algorithm is instantiable, do it. The security context is used in case the real algorithm needs to be fetched from the network. This happens especially with EmptyAlgorithm, or if an algorithm needs a helper alg.
 - Parameters

- * `scontext` - the security context to help getting algorithms
 - Exceptions
 - * `sde.algorithm.AlgorithmNotFoundException` - if initialization fails, the algorithm is unavailable
- *getName*

```
public String getName( )
```

 - Usage
 - * get the algorithm's name
 - Returns - algorithm's name
- *getAuthor*

```
public String getAuthor( )
```

 - Usage
 - * get the algorithm's author
 - Returns - algorithm's author
- *instantiateMainClass*

```
public Object instantiateMainClass( )
```

 - Usage
 - * instantiate the algorithm with no specified parent class loader. An `AlgorithmClassLoader` is created, and used to load up the `Algorithm`. The `Algorithm` class's own `ClassLoader` is used as the parent `ClassLoader`. The important idea here is that an `Algorithm` may need to instantiate another algorithm, yet does not have the Security permission to create its own `ClassLoader`, so this method call will contain a privileged section of code.
 - Returns - the newly instantiated algorithm
- *instantiateMainClass*

```
public Object instantiateMainClass( java.lang.ClassLoader parentClassLoader )
```

 - Usage
 - * instantiate the algorithm with a given parent class loader. Does the same thing as the other overloaded method of the same name, but sets the parent classloader of the new `AlgorithmClassLoader`
 - Parameters
 - * `parentClassLoader` - the parent class loader to use
 - Returns - the newly instantiated `Algorithm`
 - Exceptions
 - * `java.lang.ClassNotFoundException` -
- *addClass*

```
public void addClass( java.lang.String name, byte [] bytes )
```

 - Usage
 - * This adds a Java class to the `Algorithm`. This is used only when an `Algorithm` is being constructed from raw Java classes.
 - Parameters
 - * `bytes` - the Java class bytes
 - * `name` - the Java class name
- *getClasses*

```
public byte [] getClasses( )
```

 - Usage
 - * This method returns all of the `Algorithm`'s java classes.
 - Returns - the java classes as byte arrays

- *getClassNames*
`public String getClassNames()`
 - Usage
 - * this returns the names of the Java classes that make up this algorithm.
 - Returns - the names in an array
- *setMainClassName*
`public void setMainClassName(java.lang.String mcn)`
 - Usage
 - * This is used at construction time to set the name of the main class that represents the Class to be instantiated when the algorithm is instantiated
 - Parameters
 - * mcn - the main class name
- *getMainClassName*
`public String getMainClassName()`
 - Usage
 - * this returns the name of the class to instantiate when this algorithm is instantiated
 - Returns - the main class name
- *getHash*
`public String getHash()`
 - Usage
 - * returns the hash of the Algorithm, for use when optimizing and using algorithm "fingerprints"
- *lock*
`public void lock()`
 - Usage
 - * locks the algorithm down and prevents anything from changing

D.2.4 Class Message

The class which represents the generalized self-describing element of data. This class centralizes the generic protocols for connection-less, email-like exchanges. A different kind of interface could implement SDC in different ways.

Declaration

```
public final class Message
extends java.lang.Object
implements java.io.Serializable, java.lang.Cloneable
```

Constructors

- *Message*
`public Message(sde.crypto.Plaintext p)`
 - Usage
 - * create a message with the given plaintext.
 - Parameters
 - * p - the plaintext

- *Message*

```
public Message( sde.crypto.Ciphertext c )
```

 - Usage
 - * create a message with the given ciphertext. This is rarely used, given that usually a full SDC ciphertext is the Message itself.
 - Parameters
 - * *c* - ciphertext

Methods

- *getPlaintext*

```
public Plaintext getPlaintext( )
```

 - Usage
 - * return the plaintext
 - Returns - the plaintext
- *getCiphertext*

```
public Ciphertext getCiphertext( )
```

 - Usage
 - * return the ciphertext
 - Returns - the ciphertext
- *encrypt*

```
public void encrypt( sde.algorithm.Algorithm symAlg,  
sde.algorithm.Algorithm keyPack, sde.protocol.KeyPackagingParameters  
kpp, sde.protocol.SecurityContext scontext, java.security.SecureRandom  
srand )
```

 - Usage
 - * perform encryption of the contained plaintext. This modified the Message data, but does not return anything. It is a self-contained operation. The symmetric cipher and key packager are instantiated and used in the SDC protocol for sending encrypted messages.
 - Parameters
 - * **kpp** - key packaging parameters (might include recipients username).
 - * **scontext** - the security context for performing privileged actions
 - * **keyPack** - the key packaging algorithm
 - * **symAlg** - the symmetric cipher algorithm
 - * **srand** -
 - Exceptions
 - * `sde.crypto.CryptoException` - a cryptographic problem occurred
 - * `sde.protocol.ProtocolException` - a protocol-based error occurred
- *decrypt*

```
public void decrypt( sde.protocol.SecurityContext scontext )
```

 - Usage

- * performs decryption of the contained data. This decryption is self-contained, with the exception of the security context which might be needed for certain privileged actions (like a private key decryption). This follows the SDC protocol for decryption.
- Parameters
 - * `scontext` - the security context
- Exceptions
 - * `sde.crypto.CryptoException` - a cryptographic problem
 - * `sde.protocol.ProtocolException` - a protocol-based problem
- *sign*

```
public void sign( sde.algorithm.Algorithm signAlg,
sde.protocol.SDEKeyPair pk, java.security.SecureRandom srand )
```

 - Usage
 - * sign the contained data, using the appropriate signing parameters. This should be done before encryption.
 - Parameters
 - * `signAlg` - the signature algorithm
 - * `pk` - the keypair to perform the signature
 - * `srand` - a source of randomness
 - Exceptions
 - * `sde.crypto.CryptoException` - a cryptographic problem occurred.
- *verify*

```
public boolean verify( )
```

 - Usage
 - * This is fully self-contained, and performs a signature verification on the contained data.
 - Returns - whether or not the signature verifies.
 - Exceptions
 - * `sde.crypto.CryptoException` - a cryptographic error
- *prepareForSerialization*

```
public void prepareForSerialization( )
```

 - Usage
 - * prepare the algorithm for serialization. This implies that instantiated algorithms will be destroyed, leaving only the Algorithm objects for recreation on the other end of the serialization pipe.
- *hashAlgorithms*

```
public void hashAlgorithms( )
```

 - Usage
 - * hash the algorithms into Empty Algorithms. This turns all of the contained algorithms into empty algorithms containing only a hash of the original algorithm.
 - See Also
 - * `sde.protocol.EmptyAlgorithm` (in D.2.3, page 81)
- *serialize*

```
public byte serialize( )
```

- Usage
 - * convert to a Java-serialization representation: a byte array.
- Returns - the serialized Message object.
- Exceptions
 - * `java.io.IOException` - an error occurred during serialization

- *deserialize*

```
public static Message deserialize( byte [] bytes )
```

- Usage
 - * reconstitute a Message object from a byte array representing a serialized Message.
- Parameters
 - * `bytes` - the byte array - the serialized Message before reconstitution.
- Returns - the reconstituted Message.
- Exceptions
 - * `java.io.IOException` - error occurred during deserialization

Appendix E

sde.algorithm

<i>Package Contents</i>	<i>Page</i>
Classes	
Algorithm.....	89
AlgorithmClassLoader.....	92

E.1 Classes

E.1.1 *Class* Algorithm

An algorithm contains the proper Java classes that allow an actual cryptographic algorithm to run.

Declaration

```
public class Algorithm
extends java.lang.Object
implements java.io.Serializable
```

Constructors

- *Algorithm*
public Algorithm()
 - Usage
 - * create a new algorithm
- *Algorithm*
public Algorithm(java.lang.String **name**, java.lang.String **author**)
 - Usage
 - * create a new algorithm with algorithm name and author
 - Parameters
 - * **author** - the author name
 - * **name** - the algorithm name

Methods

- *init*
`public void init(sde.protocol.SecurityContext scontext)`
 - Usage
 - * initialize the algorithm such that if anything needs to be done before the algorithm is instantiable, do it. The security context is used in case the real algorithm needs to be fetched from the network. This happens especially with `EmptyAlgorithm`, or if an algorithm needs a helper alg.
 - Parameters
 - * `scontext` - the security context to help getting algorithms
 - Exceptions
 - * `sde.algorithm.AlgorithmNotFoundException` - if initialization fails, the algorithm is unavailable
- *getName*
`public String getName()`
 - Usage
 - * get the algorithm's name
 - Returns - algorithm's name
- *getAuthor*
`public String getAuthor()`
 - Usage
 - * get the algorithm's author
 - Returns - algorithm's author
- *instantiateMainClass*
`public Object instantiateMainClass()`
 - Usage
 - * instantiate the algorithm with no specified parent class loader. An `AlgorithmClassLoader` is created, and used to load up the Algorithm. The Algorithm class's own `ClassLoader` is used as the parent `ClassLoader`. The important idea here is that an Algorithm may need to instantiate another algorithm, yet does not have the Security permission to create its own `ClassLoader`, so this method call will contain a privileged section of code.
 - Returns - the newly instantiated algorithm
- *instantiateMainClass*
`public Object instantiateMainClass(java.lang.ClassLoader parentClassLoader)`
 - Usage
 - * instantiate the algorithm with a given parent class loader. Does the same thing as the other overloaded method of the same name, but sets the parent classloader of the new `AlgorithmClassLoader`
 - Parameters
 - * `parentClassLoader` - the parent class loader to use
 - Returns - the newly instantiated Algorithm

- Exceptions
 - * `java.lang.ClassNotFoundException` -
- *addClass*
`public void addClass(java.lang.String name, byte [] bytes)`
 - Usage
 - * This adds a Java class to the Algorithm. This is used only when an Algorithm is being constructed from raw Java classes.
 - Parameters
 - * `bytes` - the Java class bytes
 - * `name` - the Java class name
- *getClasses*
`public byte [] getClasses()`
 - Usage
 - * This method returns all of the Algorithm's java classes.
 - Returns - the java classes as byte arrays
- *getClassNames*
`public String [] getClassNames()`
 - Usage
 - * this returns the names of the Java classes that make up this algorithm.
 - Returns - the names in an array
- *setMainClassName*
`public void setMainClassName(java.lang.String mcn)`
 - Usage
 - * This is used at construction time to set the name of the main class that represents the Class to be instantiated when the algorithm is instantiated
 - Parameters
 - * `mcn` - the main class name
- *getMainClassName*
`public String getMainClassName()`
 - Usage
 - * this returns the name of the class to instantiate when this algorithm is instantiated
 - Returns - the main class name
- *getHash*
`public String getHash()`
 - Usage
 - * returns the hash of the Algorithm, for use when optimizing and using algorithm "fingerprints"
- *lock*
`public void lock()`
 - Usage
 - * locks the algorithm down and prevents anything from changing

E.1.2 Class AlgorithmClassLoader

A class loader meant solely to load up classes from Algorithm objects.

Declaration

```
public class AlgorithmClassLoader
extends java.security.SecureClassLoader
```

Constructors

- *AlgorithmClassLoader*
`public AlgorithmClassLoader()`
 - Usage
 - * create a new Algorithm Class Loader, with no parent class loader
- *AlgorithmClassLoader*
`public AlgorithmClassLoader(java.io.File dir, java.lang.ClassLoader parent)`
 - Usage
 - * create a new Algorithm class loader loading up all the classes in a given directory, with the given parent class loader.
 - Parameters
 - * **dir** - the directory containing the classes
 - * **parent** - the parent class loader
- *AlgorithmClassLoader*
`public AlgorithmClassLoader(sde.algorithm.Algorithm alg, java.lang.ClassLoader parent)`
 - Usage
 - * create a new class loader using the given Algorithm object, and the given ClassLoader as parent class loader.
 - Parameters
 - * **parent** - the parent class loader
 - * **alg** - the algorithm
- *AlgorithmClassLoader*
`public AlgorithmClassLoader(sde.algorithm.Algorithm alg)`
 - Usage
 - * create a new algorithm class loader using the given algorithm object, with no parent class loader
 - Parameters
 - * **alg** -

Methods

- *findClass*
`public Class findClass(java.lang.String name)`
 - Usage
 - * this class is overridden here only for debugging purposes to catch all calls to this method. It only calls the parent class's way of finding a class.
 - Parameters
 - * **name** -
 - Exceptions
 - * `java.lang.ClassNotFoundException` -
- *loadObject*
`public Object loadObject(java.io.InputStream is)`
 - Usage
 - * this allows a class loader to load up an object in such a way that if loading up that object's class hasn't been done yet, it is done via the `AlgorithmClassLoader` in question
 - Parameters
 - * **is** - the input stream to load the object from
 - Returns - the loaded object
- *loadClass*
`public Class loadClass(java.lang.String name)`
 - Usage
 - * this overrides the default way of loading a class. The only reason it is present is to prevent `AlgorithmClassLoaders` from reloading the entire SDE framework code, keeping that framework code in the primordial class loader, available to all.
 - Parameters
 - * **name** -
 - Exceptions
 - * `java.lang.ClassNotFoundException` -
- *getAlgorithmObject*
`public Algorithm getAlgorithmObject(java.lang.String name, java.lang.String author)`
 - Usage
 - * this returns the `Algorithm` object composed of all the classes loaded by this `Algorithm Class Loader`
 - Parameters
 - * **author** - the new algs author
 - * **name** - the new algs name
 - Returns - the new `Algorithm` object

Bibliography

- [1] OpenPGP RFC Specifications, May 1999. <http://www.imc.org/rfc2440>
- [2] PGP/MIME RFC Specification, May 1999. <http://www.imc.org/rfc1991>
- [3] S/MIME RFC Specifications, 1999. <http://www.imc.org/rfc2311>
- [4] S/MIME Version 3 RFC Draft, May 1999. <http://www.imc.org/draft-ietf-smime-msg>
- [5] L. Adleman, R. L. Rivest, and A. Shamir. A Method for Obtaining Digital Signature and Public-Key Cryptosystems. *Communication of the ACM*, 21(2), 1978.
- [6] Mihir Bellare and Phillip Rogaway. Optimal Asymmetric Encryption. In Alfredo De Santis, editor, *Advances in Cryptology—EUROCRYPT 94*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer-Verlag, 1995, 9–12 May 1994.
- [7] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.
- [8] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, December 1997. <http://java.sun.com/people/gong/papers/jdk12arch.ps.gz>
- [9] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the Association for Computing Machinery*, 16(10):613–615, October 1973. <http://www.cl.cam.ac.uk/fapp2/steganography/bibliography/1014.html>
- [10] Scott Oaks. *Java Security*. O’Reilly & Associates, inc., Sebastopol, CA 95472, May 1998.
- [11] Michael D. Schroeder and Jerome H. Saltzer. A Hardware Architecture for Implementing Protection Rings. *Communications of the ACM*, 15(3):157–170, March 1972.
- [12] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings, London Mathematical Society*,, pages 230–265, 1936. Published as *Proceedings, London Mathematical Society*,, volume 2, number 42.
- [13] United States. National Bureau of Standards. *Data Encryption Standard*, volume 46 of *Federal Information Processing Standards publication*. U.S. National Bureau of Standards, Gaithersburg, MD, USA, 1977.
- [14] David J. Wetherall. Safety Mechanisms for Mobile Code. Area Examination Paper, November 1995. <http://www.tns.lcs.mit.edu/djw/area.html>