The Design of A Component-Based Encryption Scheme

Thomas Dowling
Department of Computer Science
National University of Ireland, Maynooth
Kildare, Ireland
Thomas.Dowling@may.ie

Brian A. Malloy*
Department of Computer Science
Clemson University
Clemson, SC 29634
malloy@cs.clemson.edu

Abstract

The compelling attraction of the Internet is that it has made gigabytes of information ubiquitous to people of all walks of life. The impact of this pervasive information is that the protection of sensitive data is a stronger mandate than ever before. The mechanisms to defend information are available but are all too frequently ignored, possibly because of the inconvenience of their application. In this paper, we present the design and implementation of a software component for encrypting data that is easily incorporated into existing software applications. We use the adapter *or* wrapper *pattern to present a meaningful interface to clients of the component. We use the Unified Modeling Language, UML, to describe our class design.*

Keywords: Cryptosystem, public key, finite field, elliptic curve, design patterns, class diagram, wrapper pattern, standard C++ library.

1 Introduction

The compelling attraction of the Internet is that it has made gigabytes of information ubiquitous to people of all walks of life. The impact of this pervasive information is that the protection of sensitive data is a stronger mandate than ever before. It is now possible for an Internet shopper to browse for items and purchase them without leaving the computer. This convenience requires that the shopper place sensitive data, such as credit card information, *on the net* where it is prey to detection and theft. The mechanisms to defend information are available but are all too frequently ignored, possibly because of the inconvenience of their application[9].

In this paper, we present the design and implementation of a software component for encrypting data that is easily incorporated into existing software applications. We use the *adapter* or *wrapper* pattern to present a meaningful interface to clients of the component[6]. We use the Unified Modeling Language, UML, to describe our class design[1, 8], including the data attributes and operations. Our current implementation uses finite fields but our component development is on-going, including the development of an elliptic curve class that we can plug into the existing component[6]. We expect that the use of elliptic curves will provide greater efficiency by permitting smaller public keys to ensure the same protection that large keys provide in conjunction with finite fields. To verify ease of use, we are incorporating our component into an existing application[7].

The remainder of this paper is organized as follows. In the next section we provide background information about encryption, finite fields and elliptic curves over finite fields. In Section 3 we describe our design using UML and in Section 4 we draw conclusions.

2 Background and Related Work

In this section, we provide background about encryption schemes that use a *public key*, where the encryption and decryption keys are distinct. It's possible to determine the decryption key from the encryption key but, by using sufficiently large keys, this determination is computationally infeasible. This protection is achieved by the use of a *trapdoor process*. A trapdoor process is computationally trivial in one direction

*Brian completed part of this work on sabbatical at the National University of Ireland, Maynooth, Ireland.

but computationally infeasible in the opposite direction without some auxiliary information. There are many examples of trapdoor processes in mathematics but we concentrate on two specific areas: Finite Fields and Elliptic Curves over finite fields. We identify the trapdoor process and use it to implement the corresponding Public Key Cryptosystem.

2.1 Finite Fields

A field is a set $\mathbf{F}$ with multiplication and addition operations satisfying certain rules; a detailed account appears in reference [6]. Examples of fields include the real numbers $\mathbf{R}$ and the set $\mathbf{Z}_p^*$ of non-zero elements of the integers modulo a prime number p. A *Finite Field* $\mathbf{F}_p$ is a field with a finite number of elements. Examples include $\mathbf{Z}_p^*$. A *generator* g of a finite field $\mathbf{F}_p$ is an element of $\mathbf{F}_p$ whose powers run through all the elements of $\mathbf{F}_p$. Every finite field has a generator. The trapdoor process for Finite Fields is raising to a power in a large finite field, i.e. given a generator g it is computationally infeasible to compute g^{ab} knowing only g^a and g^b. The interested reader may refer to reference [6] for more detail. It is this property of finite fields that is exploited in the encryption scheme below.

2.2 Elliptic Curves over Finite Fields

There are many types of elliptic curves but for the purposes of the present paper we restrict our discussion to curves of the Weierstrass type. Again let $\mathbf{F}_p$ be a finite field. An *Elliptic Curve* $\mathbf{E}$ over $\mathbf{F}_p$ is the set of points (x, y) in $\mathbf{F}_p$ which satisfy the equation
$y^2 = x^3 + ax + b$, where a and b are elements of $\mathbf{F}_p$

A full presentation of elliptic curves is beyond the scope of this paper but the interested reader may consult reference [6] and [10]. Elliptic curves are natural analogues of finite fields and are amenable to computation because of their rich structure. The analogue of raising to a power in a finite field is multiplying a point on an elliptic curve by an integer. This is the trapdoor process required for an encryption scheme.

2.3 Implementing the Public Key Cryptosystem

The implementation that we use is based on the El Gamal encryption scheme[4]. The finite field version works as follows: Let the integer equivalent of the message to be transmitted be denoted by P. Users A and B start by deciding upon a very large finite field $\mathbf{F}_p$ and a generator g of that field. User A randomly chooses an integer a in the range $0 < a < p - 1$. This is the secret deciphering key. A then computes and publishes g^a. This is the public key. User B does the same thing. To send a message to user A, an integer k is chosen at random and A is sent the following pair of elements of $\mathbf{F}_p$,

$$(g^k, Pg^{ak})$$

Recall that g^a is publicly known but a is known only to the user A. With this knowledge the user A can strip off the g^{ak} and retrieve P, but without the knowledge of a no one else can retrieve P.
The Elliptic curve version works as follows: Decide upon a finite field $\mathbf{F}_p$, an elliptic curve $\mathbf{E}$, defined over it, and a base point B on $\mathbf{E}$. Each user chooses a random integer a (the secret key) and publishes the point aB (the public key). The original message is embedded into a set of points Pm on $\mathbf{E}$. To send a message Pm to user S, user A chooses a random integer k and sends the pair of points,
$(kB, Pm + k(a_S B))$, where $a_S B$ is the public key of S.

Since S knows the secret key a_S the embedded message Pm can be retrieved by multiplying the first point by a_S then subtracting the result from the second point. Note that the arithmetic operations for elliptic curves over finite fields are not the same as those over the real numbers. Again the trapdoor process prevents anyone else from obtaining Pm, i.e., given the point $a_S B$ on $\mathbf{E}$, it is computationally infeasible to compute the integer a_S.

3 The Design of the Component

In this section, we present the design of our component, including the *adapter* pattern, or *wrapper* that we use to present a meaningful interface to clients of the component[5]. We use the Unified Modeling Language, UML, to describe our design[1, 8]. In the next

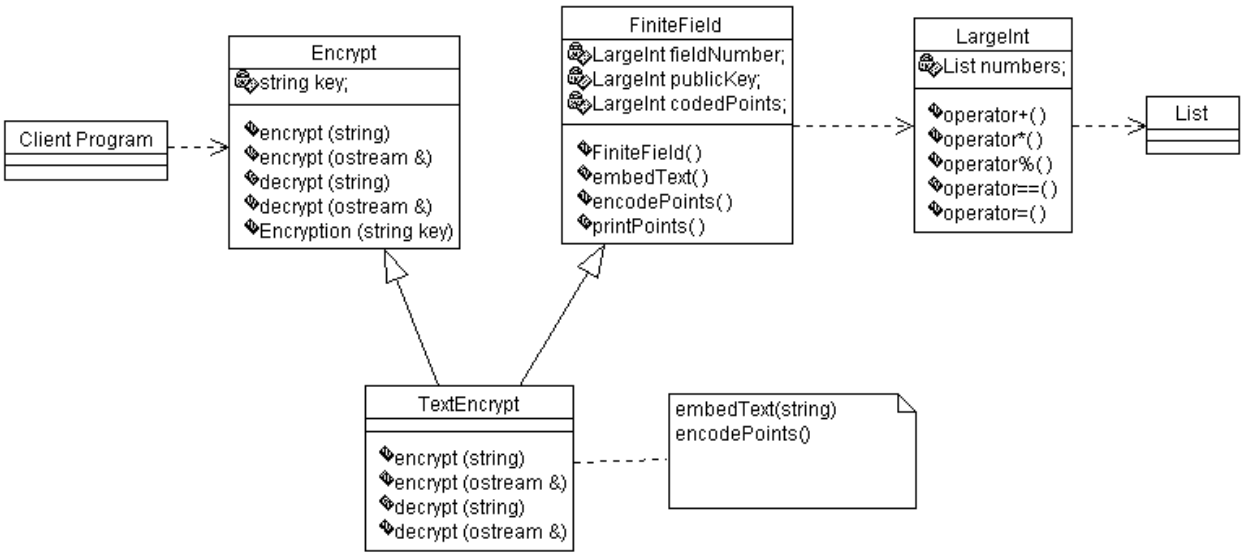

Figure 1: The *wrapper* pattern.

section we discuss the *wrapper* and in Section 3.2 we discuss components and the play-and-play capability of our encryption component.

3.1 The Use of the Wrapper Pattern

Figure 1 illustrates the design of our component, including the *wrapper pattern* that we use to present an easy-to-use interface to clients of the component. The client program is illustrated on the left side of the figure and the inheritance hierarchy, Encrypt and TextEncrypt, present the interface to the client program. The methods for encryption and decryption are symmetric interfaces; in particular, the client may pass either a *string* or an *ostream* to Encrypt, for either encryption or decryption. The *string* type is included in the *standard* C++ *library*, and an *ostream* is a mechanism for organizing and maintaining a sequence of characters[11]. We have found the string and ostream data types to be versatile structures for storing data. Moreover, the ostream structure has proven useful for incorporating a graphical user interface, GUI, into a command-driven application[7].

In addition to the encryption class hierarchy, illustrated in Figure 1, we also show our base class FiniteField, which contains the functionality to actually perform the encryption and decryption of the string or ostream. This functionality is inherited by TextEncode from FiniteField; thus, the *wrapped* class is FiniteField. Our FiniteField class also includes functionality to embed text and includes a *generator*, as discussed in Section 2. To ensure security, the FiniteField class uses extended precision numbers, thus, the association with LargeInt, also illustrated in Figure 1. LargeInt includes functionality to perform extended precision addition, multiplication and modulo arithmetic. The extended integers are stored in a list, shown on the right side of the figure.

3.2 The Plug-And-Play Component

A software component is a unit of software, intended for reuse, that incorporates a clearly defined interface and an easily understood abstraction. The use of software components offers substantial benefits to software engineering by enabling the practical reuse of software. this reuse may serve to amortize the investment in software over multiple applications. We have facilitated the reuse of our encryption component through the use of the wrapper pattern, which hides the implementation details of the FiniteField class behind the wrapper, and allows the functionality of the FiniteField to be exploited by the Encrypt and TextEncrypt hierarchy, opaquely to the client of the compo-

nent. Moreover, the abstraction of encrypt and decrypt is easily understood by clients of the component, who are not encumbered by the functionality of the FiniteField class.

We intend to experiment with our encryption component by investigating the use of elliptic curves over finite fields, as an alternative to the use of finite fields. We expect that the use of elliptic curves will provide greater efficiency to our encryption component by permitting smaller keys to ensure the same protection afforded by larger keys.

4 Concluding Remarks

We have presented the design of an encryption component that uses the *adapter* or *wrapper* pattern to present an easily understood abstraction to clients of the component. Preliminary experiments indicate that we can encrypt or decrypt a file containing 100K bytes of data in a fraction of a second when executed on a 200 MHz Pentium processor running the Linux operating system. We intend to continue our component development by investigating the use of elliptic curves over finite fields, as an alternative to the use of finite fields. We expect that the use of elliptic curves will provide greater efficiency to our encryption component by permitting smaller keys to ensure the same protection afforded by larger keys. We are in the process of investigating the reuse of the wrapper pattern to exploit the standard C++ library list features, and we are investigating the special need for both unit and system testing of our component when incorporated into an existing system[7].

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Object Technology Series. Addison-Wesley, 1999.

[2] D. M. Bressoud. *Factorization and Primality Testing.* Springer-Verlag, 1989.

[3] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on information Theory IT-22*, pages 644–654, 1976.

[4] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on information Theory IT-31*, pages 469–472, 1985.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[6] N. Koblitz. *A Course in Number Theory and Cryptography.* Springer-Verlag, 1994.

[7] B. A. Malloy, J. D. McGregor, and S. Hughes. Integrating a gui into a command driven application. *Proceedings of the IASTED International Conference on Software Engineering and Applications*, Oct 1999. to appear.

[8] J. Rumbaugh, G. Booch, and I. Jacobson. *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison-Wesley, 1999.

[9] M. Scott. Web site. *www.compapp.dcu.ie/m̃ike/*, September 1999.

[10] J. Silverman. *The arithmetic of Elliptic Curves.* Springer-Verlag, 1986.

[11] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, third edition, 1997.