D R A F T
(Please do not quote.)

HcryptoJ: A Java Platform for Education and Research in Historical Cryptology

Ralph Morelli
Computer Science Department
Trinity College
Hartford, CT 06106
ralph.morelli@trincoll.edu

March 22, 2004

Abstract

This paper describes HcryptoJ (*H*istorical cryptology in *J*ava), a Java-based programming library for implementing and analyzing historical ciphers. HcryptoJ is designed to support research and teaching in historical cryptography and computer science. It is suitable for use by novice and intermediate level Java programmers as well as by nonprogrammers and has been used in a variety of undergraduate research and teaching projects. HcryptoJ is general enough to implement any symmetric key cipher system that translates plaintext strings into ciphertext strings and vice versa. This paper provides a brief overview of the HcryptoJ project and shows how it is being used to support undergraduate teaching and research. The HcryptoJ library, including source code and documentation, is available for free download.

1 Introduction

HcryptoJ (*H*istorical cryptology in *J*ava) is a Java *application programming interface (API)* for implementing and analyzing historical ciphers. An API is a library or an organized collection of classes and methods that is used to support programming within a certain domain. The domain in this case is *historical cryptology*, by which we mean primarily encryption, decryption, and cryptanalysis of ciphers that were used up through World War II. Such ciphers typically use a shared (*symmetric*) key together with manual or mechanical calculations to encode a block of alphabet symbols.

The primary goal of HcryptoJ is to serve as an easy-to-use programming platform for the research and study of historical ciphers and cryptology. At the same time HcryptoJ serves as an excellent platform for student programming and research projects at both the introductory and advanced levels. For example, with HcryptoJ it is possible for a beginning programmer to implement a full-fledged cipher system by implementing only a handful of methods. HcryptoJ contains a number of built-in utility classes and methods, such as text analysis classes and file I/O methods, which make it a suitable platform for programming assignments in Java-based CS1 and CS2 courses. Because of its extensible object-oriented design, the HcryptoJ system itself is a suitable object of study for courses in data structures and software engineering. Indeed, the first version of HcryptoJ was developed in part as a group project in a software engineering class [5].

HcryptoJ also serves as a good platform for supporting various undergraduate research projects. These projects range from largely experimental work – e.g., designing and running tests for a genetic algorithm that analyzes cryptograms – to program design and implementation projects – e.g., researching and implementing a program to implement a particular cipher system – to artificial intelligence projects – e.g., designing and implementing an expert system or genetic algorithm to analyze ciphers. All of these examples are actual student projects that have been or are being conducted with HcryptoJ.

What makes HcryptoJ suitable for a wide range of educational uses is its standardized and extensible programming interface. HcryptoJ's object-oriented design is closely modeled on that of the *Java Cryptography Extension (JCE)*, a Java library that provides a standardized programming framework for encryption algorithms ([2],[4]). As befits our purpose of supporting undergraduate education and research, HcryptoJ's design is simpler than the JCE.

1.1 HcryptoJ General Features

HcryptoJ provides an extensible platform for implementing and analyzing the full range of historical ciphers:

- It provides a general framework for implementing cipher systems. It includes implementations of the cipher systems shown in Table 1. These can be studied as examples of how to use the library. Once the HcryptoJ framework is understood, beginning and intermediate programmers can easily implement their own cipher systems.

Cipher Engines		
Affine	Caesar	Substitution
Playfair	Polysubstitution	Vigenere
Transposition	Autoclave (plugin)	NullCipher (plugin)

Table 1: Built-in cipher engines in the current version of HcryptoJ.

- It provides a general framework for implementing various type of *analyzers* – that is, programs that perform some type of cryptanalysis of an encrypted message. Table 2 provides an inventory of analyzers that have already been implemented and are provided with the library.

Cryptanalysis Tools		
Frequency Analyzer	Histogram Tool	Index of Coincidence
Caesar Analyzer	Affine Analyzer	Substitution Analyzer
Vigenere Analyzer	Null Analyzer (plugin)	Pattern Word Searcher

Table 2: Built-in analyzers in HcryptoJ.

- It is compatible with both command line and graphical (windows-based) user interfaces. Examples of both types of programs are provided with the library. Some of the programs, such as *CryptoToolJ* can be used as is to perform cryptographic and cryptanalytic tasks using built-in classes and methods.
- It employs an extensible, object-oriented design that makes it relatively easy to build encryption and cryptanalysis applications. New encryption and analysis algorithms can be created by extending existing classes and can be easily plugged into the existing applications, such as CryptoToolJ.
- It is written in Java so it can be used on just about any hardware and software platform. Java programs can run without re-compilation on any system that supports the Java Runtime Environment (JRE), which is supported by Windows, Linux, Macintosh, Unix and most major operating systems.
- It supports the full range of *Unicode* alphabets (character sets), which allows it to handle messages written in Greek, Hebrew, Katakana, and other character sets.
- It is available, along with its source code and documentation, for free download at:

  `http://starbase.trincoll.edu/~crypto`

In the remainder of this paper we describe HcryptoJ's overall design and functionality and provide examples of how it has been used in undergraduate teaching and research.

2 Cryptography Basics

Before describing HcryptoJ it will be helpful to provide some background on cipher algorithms. There are a number of good references available on historical cryptography, including Kahn [3] and Beker and Piper [1].

Cryptography is the art and science of designing systems for transmitting secret messages. The message to be kept secret is called the *plaintext* and the process of hiding its meaning is called *encryption* (or *enciphering*). The secret message is called the *ciphertext* and the process of translating the ciphertext to plaintext is called *decryption* (or *deciphering*).

A *cipher* is an algorithm for translating plaintext to ciphertext and vice versa. Normally the translation will depend on some kind of *cryptographic key*, which is used by the algorithm to encrypt or decrypt the message. There are a wide range of historical ciphers, but generally speaking they fall into two distinct groups: *substitution ciphers* and *transposition ciphers*. For example, a very simple substitution cipher is the *Caesar cipher*, in which each of the letters a through z is represented by the letter which occurs three places after it in the alphabet, with the last three letters, x through z, wrapping around to the beginning of the alphabet. So for the standard plaintext alphabet, we get the following ciphertext alphabet:

```
Plaintext:  a b c d e f g h i j k l m n o p q r s t u v w x y z
Ciphertext: D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
```

To encrypt the message "caesar was a roman" we simply replace each plaintext character with the corresponding ciphertext character: FDHVDU ZDV D URPDQ. Decryption would perform the substitution in the opposite direction.

A *transposition* cipher rearranges the letters in the original message. For example, in one type of transposition cipher, the transposition key *43210* describes how to rearrange the letters in a 5-character block in which the character positions are numbered 0 through 4. Using this key a transposition cipher would encrypt "caesa rwasa roman" into ASEAC ASAWR NAMOR.

Cryptanalysis is the development and study of techniques used to break ciphers. (Generally *cryptology* refers to the combined fields of cryptography and cryptanalysis.) For example, a cryptanalysis of a Caesar cipher would be an attempt to decipher the message FDHVDU ZDV D URPDQ without knowing its key. If it is known that the message was created with a Caesar cipher, one simple analysis would be to try each of the 25 possible shifts until the message is translated. If the type of cipher is not known, the cryptanalysis might proceed by counting the frequencies of each of the letters in the message and comparing those with the frequencies of the English alphabet. In this example there aren't enough letters in the message for this *frequency analysis* to work, but with a long enough message, a simple substitution cipher can easily be broken by this type of analysis.

3 The HcryptoJ Architecture

This section describes the HcryptoJ architecture and provides an example of how it can be used to create a cipher system. We begin with a brief overview of the main classes and methods that make up HcrtypoJ.

3.1 Engines and Keys

In HcryptoJ, every cipher system contains two basic components, a *cipher engine*, which performs the encryption and decryption algorithms, and a cryptographic *key*, which contains the secret data used by the engine.

Figure 1 shows the primary classes and methods in the HcryptoJ architecture. The classes are grouped into *packages*, which are represented by the dotted rectangles. All of the classes in the library are subclasses of the *java.lang.Object* class, as indicated by the closed-headed arrows. The basic design utilized by HcryptoJ is that a *Cipher* is an object that uses a specific *CipherEngine* to encrypt and decrypt *Strings* of text. The HcryptoJ architecture can be extended with minimal effort by incorporating new cipher engines – such as *CaesarEngine* and *TranspositionEngine* – into this hierarchy.

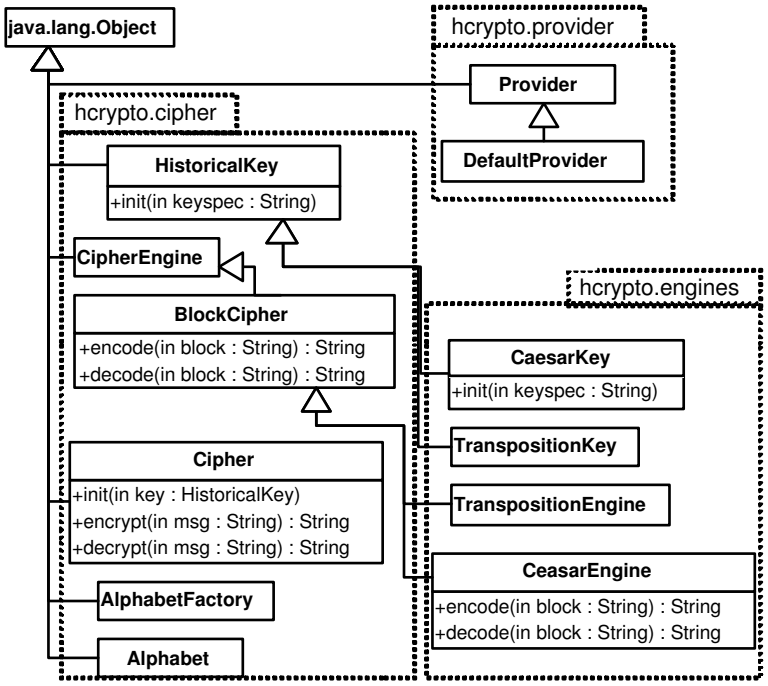

Figure 1: The main classes and methods used for encryption and decryption in the HcryptoJ library.

As Figure 1 shows, a key is any subclass of the built-in `HistoricalKey` class. Similarly, a cipher engine is a subclass of the built-in `BlockCipher` class. A cipher engine must contain implementations of the following three methods (whose names are abbreviated in the figure):

```
engineInit(HistoricalKey key)
```

```
engineEncode(String s): String
engineDecode(String s): String
```

The first method describes how to initialize the cipher engine given an `HistoricalKey` of the appropriate type. The encode and decode methods describe how to translate each *block* of the message, where a block can be as little as 1 character of text. For example, for a Caesar cipher, the `engineEncode()` method must describe how to shift each character by a given amount. So, assuming the Caesar shift is 3, `engineEncode("a")` would return "d". The `engineDecode()` method must describe the converse operation.

3.2 Creating a Cipher System

The following code shows the `engineEncode()` method for HcryptoJ's default Caesar cipher:

```
public String engineEncode(String s) throws Exception {
   if (blocksize != 1)
       throw new Exception("Invalid blocksize for Caesar cipher " + blocksize);
   char ch = s.charAt(0);
   if (alphabet.isInAlphabet(ch)) {
      return "" + encodeShift(ch, shift);
   }
   else
      return s;
}
```

This code first checks that the `blocksize` was set correctly when the engine was initialized. The `encodeShift()` and the `alphabet.isInAlphabet()` methods used here are built-in HcryptoJ methods. If *ch* is in the alphabet, the `encodeShift()` method will replace it with the character *shift* characters to its right.

For this example, a similar amount of coding would be required for the `engineDecode()` method. Because of HcryptoJ's object-oriented design, `engineEncode()` and `engineDecode()` methods are called automatically by the `Cipher.encrypt()` and `Cipher.decrypt()` methods (Figure 1). These are the methods that would be called by an application program to encrypt or decrypt a cipher. An example is provided below. Thus, because of its object-oriented design, HcryptoJ reduces the encryption and decryption tasks to the coding of two methods, *encode()* and *decode()*.

The `engineInit()` method for Caesar cipher is just as easy to code. Its task is basically to get information that is encapsulated in the cryptographic key and store that information in the cipher engine. In this case of the Caesar cipher, this information includes the *shift* used by the Caesar engine, the alphabets used for both encryption and decryption, and the blocksize assumed by the cipher. Each of these items of data are stored in the `CaesarKey`:

```
protected void engineInit(HistoricalKey hKey) throws Exception {
```

```
   if ( !(hKey instanceof CaesarKey))
       throw new Exception("InvalidKey: Caesar requires CaesarKey");
   key = (CaesarKey) hKey;
   alphabet = key.getAlphabet();
   cipherAlphabet = key.getCTAlphabet();
   shift = key.getShift();
   this.blocksize = key.getBlocksize();
}
```

The shift, blocksize, and alphabets set up in this method are used in the `engineEncode()` and `engineDecode()` methods.

Thus, creating one's own cipher engine with HcryptoJ is a matter of writing three methods. That is, it is not necessary to write code for the many other methods involved in the encryption process. Of course, not every cipher engine will have methods that are as easy to code as Caesar cipher. But HcryptoJ's architecture standardizes what must be done: the `engineInit()` method delivers key data to the cipher engine and the `engineEncode()` and `engineDecode()` methods describe exactly how each character is translated.

Now let's look at how `CaesarKey` fits into this scheme. As Figure 1 shows, a valid key for, say, a Caesar cipher, must be defined as a subclass of `HistoricalKey`. It must contain a valid implementation of the `init()` method, which is responsible for translating a *key specification* into data that can be used by the cipher engine. Here's an example of `CaesarKey.init()`:

```
public void init(String keyspec) throws Exception {
   initKey(keyspec);
   this.blocksize = 1;
   this.shift = Integer.parseInt(keyword);
   if (shift < 0)
       throw new Exception ("Invalid shift value for Caesar cipher: " + shift);
}
```

A *keyspec* for a Caesar cipher takes the form *3/az*. The data preceding the slash is called the *keyword* and the data following the slash is called the *alphabet descriptor*. As in the previous methods, much of the work here is done by built-in methods. Thus the `initKey()` method will separate the keyword from the alphabet descriptor and will directly create the alphabet that will be used by the cipher engine. So the only real work done here is to convert the shift from a string into an integer because our Caesar engine expects the shift to be a positive integer.

As this example shows, using the HcryptoJ platform, the process of designing of an encryption engine can focus primarily on the algorithms needed to perform encryption and decryption. One needn't worry about the ancillary details of performing I/O, breaking the ciphertext into blocks, padding the blocks with extra characters, and the several other tasks required to encrypt and decrypt a message. Similarly, the amount of coding required to create a complete cipher system is fairly minimal, requiring just the coding of the three methods

discussed above. Because of these features, the HcryptoJ architecture is well-suited for serving as a platform for undergraduate programming assignments and research projects. (An appendix contains the complete code for the default Caesar cipher.)

3.3 Using a Programmer-Defined Cipher System

This section describes how a programmer-defined cipher system would be incorporated into an application program. The following code segment is a complete application program that uses Caesar cipher to encrypt and decrypt a string:

```
import hcrypto.cipher.*;   // Import library classes and methods
import hcrypto.provider.*;
import hcrypto.engines.*;

public class SimpleTester {
  public static void main(String args[]) throws Exception {
    Provider.addProvider(new MyProvider("MyName"));
    Cipher cipher =
      Cipher.getInstance("Caesar", "MyName");
    HistoricalKey key =
      HistoricalKey.getInstance("Caesar", cipher.getProvider());
    key.init("3/AZ");
    cipher.init(key);
    System.out.println(cipher.encrypt("THIS IS A TEST"));
  }
}
```

The first three statements *import* the names of the classes and methods from the HcryptoJ library. The *SimpleTester* program begins execution in the `main()` method. The first statement installs a provider that contains necessary information about the programmer-defined cipher system:

```
    Provider.addProvider(new MyProvider("MyName"));
```

The provider interface, which is modeled on the *Java Cryptography Extension* ([4]), contributes to HcryptoJ's extensibility. A *provider* is a searchable index of available ciphers. The main provider class, `Provider`, contains methods that add new providers into a program and methods that search for ciphers by name or by name and provider. Writing a provider class for one's own cipher is very simple. Here's the complete implementation of `MyProvider`:

```
package providers;
import hcrypto.provider.*;
public class MyProvider extends Provider {
   public MyProvider(String name) {
      this.name = name;
      put("Caesar", "plugins.CaesarEngine", "plugins.CaesarKey");
   }
}
```

`MyProvider` simply associates the name of the cipher engine, *Caesar*, with the names of the Java classes used to implement its engine (*plugins.CaesarEngine*) and its key (*plugins.CaesarKey*).

After telling the program about the existence of this new cipher, it is now possible to create an instance of the Caesar cipher:

```
Cipher cipher =
  Cipher.getInstance("Caesar", "MyName");
```

The two arguments in the `Cipher.getInstance()` method are the name of the cipher engine and the name of its provider. The `getInstance()` method will search the available providers for an implementation of "Caesar" provided by "MyName." Assuming it is found, the program can then move on to the encryption and decryption steps.

Before a cipher object can be used to encrypt or decrypt messages, it must be initialized with a key of the appropriate type. That is the purpose of the next three statements, which create an instance of the key, initialize the key, and pass the key to the cipher engine:

```
HistoricalKey key =
  HistoricalKey.getInstance("Caesar", cipher.getProvider());
key.init("3/AZ");
cipher.init(key);
```

The last statement in `main()` uses the `encrypt()` method to encrypt a string:

```
System.out.println(cipher.encrypt("THIS IS A TEST"));
```

Once a cipher object has been initialized with an appropriate key, it will use that key in all subsequent encryptions and decryptions. Thus, given the fact that in this example Caesar cipher uses a shift of 3, the encryption of "THIS IS A TEST" would result in "WKLV LV D WHVW."

3.4 Creating and Using Analyzers

Because cryptanalysis can take many and diverse forms, HcryptoJ provides a very general programming interface. As shown in Figure 2, an *Analyzer* is any class that implements the `hcrypto.analyzer.Analyzer` interface, which consists of the following methods:

```
setup(String msg)
run()
getReport(): String
```

Any class that implements these methods can be easily incorporated into application programs that use the HcryptoJ library. The *setup()* method takes the message being analyzed as a parameter and initializes the analyzer. The *run()* method performs the analysis. The *getReport()* method gets the analyzer's report, which must be collected into a string. Aside from these minimal

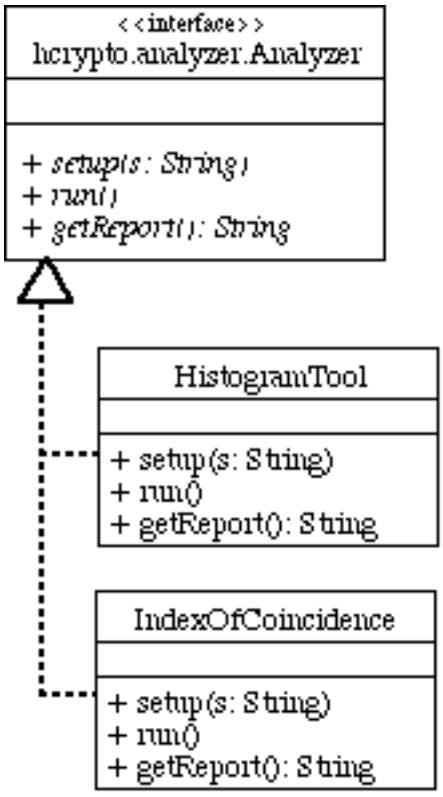

Figure 2: HcryptoJ's analyzer interface.

requirements, the analyzer can perform any kind processing whatsoever on the target message.

The following is an example of a very simple plugin analyzer that just returns the first five characters of the message.

```
package analyzers;     // Analyzer plugins belong to this package
import hcrypto.analyzer.*;

public class TestAnalyzer implements Analyzer {
    private String text;
    private StringBuffer resultSB = new StringBuffer();

    public TestAnalyzer() { }
    public void setup(String text) {this.text = text; }
    public void run() {
      resultSB.append("TestAnalyzer: The fiirst 5 letters are "
              + text.substring(0,5) + "\n");
    }
```

```
    public String getReport() {return resultSB.toString(); }
}
```

Table 2 provides of list of examples that come with the current version of HcryptoJ. *HistogramTool* is an analyzer that provides a frequency histogram of the letters contained in the message. *IndexOfCoincidence* is an analyzer that computes a statistic that is used by cryptanalysts to determine whether a message was encrypted with a simple or polyalphabetic substitution cipher. *CaesarAnalyzer* is an analyzer that automatically breaks messages encrypted with Caesar cipher. As these examples show, HcryptoJ's analyzer interface can serve as a platform for a wide variety student programming and/or research projects.

4 Using HcryptoJ in Teaching and Research

In this section we provide a functional overview of HcryptoJ and show how it can be (and has been) used in a variety of undergraduate projects.

4.1 Using Built-in Applications

Table 3 provides a list of HcryptoJ's built-in application programs. In addition to being useful application programs that can be used in a variety of non-programming projects, they serve as examples of how to design and build an application using HcryptoJ. These programs can be run without recompilation on any platform that supports the Java Runtime Environment.

Application	**Description**
TestCipher	Encrypts/decrypts text entered on the command line.
FileCipher	Encrypts/decrypts files named on the command line.
TestAnalyzer	Command-line cryptanalysis tool.
CryptoToolJ	Menu-driven cryptology tool.
CryptoAppletJ	Applet version (http://starbase.trincoll.edu/ crypto/).

Table 3: Applications available with HcryptoJ.

4.2 Encrypting and Decrypting Short Messages

TestCipher is a command-line program that encrypts and decrypts short messages using any of HcryptoJ's built-in cipher engines. It is mainly useful for quick testing of new cipher engines as they are being developed. For example, the following command will encrypt the message "this is a test" using a Caesar cipher:

```
java TestCipher Caesar 3/az "this is a test"
```

Note the use of the *key specification* "3/az" to communicate the Caesar shift (3) and alphabet (lowercase *a* to *z*) to the cipher engine. This command line will first create a *CaesarKey*, initializing it with the key specification. It will then create and initialize a *CaesarEngine*, which can then be used to encrypt or decrypt messages using that key. In this case, the program would produce the following output:

```
The keyword is: 3, Algorithm=Caesar  Provider=Default
this is a test
wklv lv d whvw
this is a test
```

4.3 Encrypting and Decrypting Files

The *FileCipher* program can be used with any of the built-in cipher engines to encrypt and decrypt text files. For example, the following command will Caesar-encrypt *file1*, an ISO-2022-JP encoded email message that includes a mixture of ASCII characters and Japanese Katakana characters:

```
java  FileCipher Caesar 3/printable+UnicodeBlock.KATAKANA -e file1 file2 ISO-2022-JP
```

This example also illustrates HcryptoJ's internationalization capability. In this command, the complicated looking key specification sets up a Caesar key, with a shift of 3, that will translate all printable ASCII and all Katakana characters in the input file.

Both *FileCipher* and *TestCipher* can serve as platforms for student programming projects. For example, as a senior programming project, a student developed an implementation of *RailFence* cipher, a well-known transposition cipher. (See http://starbase.trincoll.edu/crypto/historical/). During development of RailFence classes, the `TestCipher` program was used as follows to test and debug the algorithms:

```
java  TestCipher RailFence 3/azAZ09 "this is a test"
```

Successful performance of the cipher in this kind of test confirms both that the cipher conforms to the appropriate object-oriented design and that it works correctly. From the student's perspective the project required mastery of both algorithm design and object-oriented design.

4.4 Cryptanalyzing Messages

TestAnalyzer is a command-line program that can be used to test a cryptanalysis program. For example, the following command line runs *CryptogramAnalyzer* – a program that tries to break simple substitution cryptograms – on the encrypted file, producing an analysis either in text or in a pop-up window:

```
java TestAnalyzer CryptogramAnalyzer encryptedfile
```

By taking care of the file I/O and other low-level tasks, the *TestAnalyzer* program serves as a useful platform for a variety of text and cipher analysis projects at the beginning and intermediate level. For example, as long as it is designed as an implementation of the `hcrypto.analyzer.Analyzer` interface, any student-written analyzer can be tested using this program. For example a student is currently working on an expert system program that will attempt to determine the type of cipher that was used to create an encrypted message. When finished, this program will be able to tell whether the message was encrypted with a transposition or substitution cipher, whether it is a simple or poly-alphabetic substitution cipher, and so on.

4.5 Using CryptoToolJ

CryptoToolJ is a menu-driven application that integrates HcryptoJ's built-in ciphers and analyzers into a easy-to-use cryptology program (Figure 3). It can be run either as a standalone program or as an online applet, although the applet version has much-reduced functionality because of Java security restrictions. CryptoToolJ can be used to encrypt and decrypt messages created either with its default cipher engines – Caesar, simple substitution, Vigenere, and others – or with student-written cipher engines that can be incorporated into the program as plugins. Similarly, it can be used to cryptanalyze messages using either built-in or student-written analyzers.

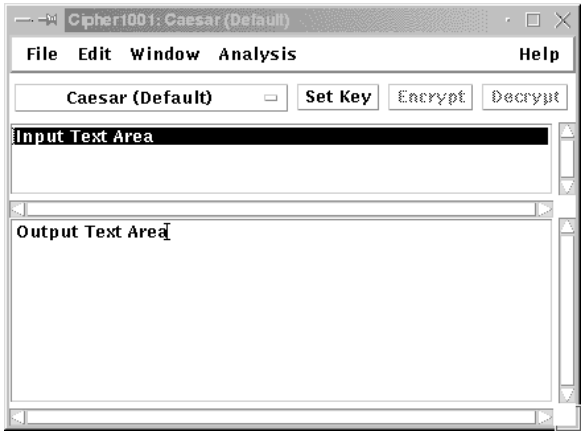

Figure 3: CryptoToolJ is a GUI program that supports encryption, decryption, and cryptanalysis of various kinds of historical ciphers.

As with the other built-in application programs, CryptoToolJ can be used as a platform for testing students programs. By providing an easy-to-use interface, the student can focus on the design of the cryptographic and/or analytic algorithms.

4.6 Extending HcryptoJ and CryptoToolJ

One of our long-range goals is to extend HcryptoJ and CryptoToolJ with implementations of all well-known historical ciphers as well as analyzers that attack them. In addition to built-in cipher engines and analyzers, CryptoToolJ's functionality can be extended through its *plugin* interface. Plugin engines and analyzers (see Tables 1 and 2) that are placed in special plugin directories are loaded into the program at runtime. Of course, the code for these new resources must follow the design specifications that were described previously. But the HcryptoJ interface is quite simple and accessible.

Similary, incorporating a new analyzer resource into CryptoToolJ works in a similar fashion. The Java classes (code) that implement the analyzer must be placed in the *analyzers* subdirectory. When CryptoToolJ is started it searches this directory and adds analyzer plugins to its Analysis menu. In order for the program to be able to run the analyzer, the analyzer must provide an implementation of the *hcrypto.analyzer.Analyzer* interface, which contains the necessary details on how the two programs will communicate with each other.

5 An Accessible and Extensible Research Tool

As the preceding discussion shows, the HcryptoJ library greatly simplifies the process of creating and using cipher engines and cryptanalyzers. Its usefulness as a platform for undergraduate teaching and research derives from its extensible, object-oriented design.

Because the library's architecture manages most of the ancillary tasks involved in writing a cipher system, the essential programming tasks can be left to the student. Even novice Java programmers (CS1 students) can make use of its classes to create their own simple cipher engines and their own applications. More advanced programmers can make use of its analyzer interface to create more ambitious ciphers as well as their own cryptanalysis tools. Students studying software engineering can use the system as an object of study and for further design work.

Because of its accessibility and extensibility, HcryptoJ can also be useful to researchers, who wish to create an implementation of a historical cipher or an analysis program as part of their study, and to students and instructors of historical cryptology, who can use it to gain a very detailed understanding of the various encryption and analysis algorithms.

5.1 Examples of Student Research Projects

In addition to the examples of programming projects mentioned above, HcryptoJ has served as an evolving platform for student and faculty research in a variety of areas, including the following projects:

- HcryptoJ's and CryptoToolJ's original design and implementation were the result of a semester-long project in an Introduction to Software En-

gineering course. They were subsequently revised as part of a student-faculty summer research project ([5]).

- HcryptoJ was used as a platform for the development and testing of simple substitution analyzer based on the *genetic algorithm GA* approach. This project demonstrated that a word-based genetic algorithm could successfully cryptanalyze short cryptograms ([6]). Presently a pair of first-year computer science students are using the GA to design and run various experiments aimed at improving its performance.

- A senior computer science major researched and developed an implementation of the *RailFence* cipher as her senior project. See

  `http://starbase.trincoll.edu/~crypto`

- A senior computer science major is developing a frequency-based genetic algorithm for analyzing simple substitution ciphers. When completed this approach will be compared, experimentally, with the word-based GA.

- As another senior project, a computer science major is presently developing an expert system that will use various cryptanalysis techniques to identify the type of cipher engine that was used to create a message. The goal of this project is to be able to break messages written in any one of a handful of substitution and transposition ciphers.

As these examples show, the availability of a platform such as HcryptoJ makes interesting cryptographic and cryptologic work accessible to undergraduates at all stages of their computer science education.

6 Related Work

There are numerous cryptography and cryptanalysis tools available. The following list provides a brief summary:

- Cipher Clerk – a Java applet that contains implementations of a large collection of historical ciphers (http://members.magnet.at/wilhelm.m.plotz/-Bin/CipherClerk.html).

- Secret Code Breaker – Javascript implementations of Caesar and other substitution ciphers (http://codebreaker.dids.com).

- Advanced Cryptography Tool – A cryptographic tool that uses strong encryption (http://maga.di.unito.it/security/resources/ACT/act.html).

- Cypher – A cryptologic tool for historical ciphers that was developed as a software engineering project at the University of North Carolina (http://www.cs.unc.edu/ stotts/COMP145/homes/crypt/).

- The Cryptology Matrix – A collection of resources, including CryptAid, for creating and analyzing historical ciphers from New Mexico State University (`http://www.math.nmsu.edu/crypto/public_html/`).
- Cryptlib – A security toolkit that provides strong encryption and authentication services (http://www.cs.auckland.ac.nz/ pgut001/cryptlib/).
- Pretty Good Privacy (PGP) – A security toolkit that provides strong encryption and authentication services (http://web.mit.edu/network/pgp.html).

Most of these are commercial products that provide strong encryption and authentication services. These are not suited at all for historical cryptography. But their most important limitation for our purposes is the fact that they could not serve as a programming and development platform for undergraduates in computer science. Among those that are specifically designed for historical ciphers (CryptAid, Cypher), none provides the extensible, integrated platform available in HcryptoJ. And none appears to be designed to support undergraduate research and teaching in computer-based historical cryptography.

Similarly, although one can find examples of cryptanalysis tools online, these are usually designed to solve one specific problem – e.g., to solve simple substitution cryptograms. As far as we know, none of these provides an integrated, extensible programming platform for the study of historical cryptanalysis, such as that provided by HcryptoJ.

HcryptoJ's cryptography and cryptanalysis framework is well-suited to to support computer-based research and education in historical cryptology. It is our hope that amateur cryptographers, researchers, and students of cryptology will use HcryptoJ in their work and contribute to its continued development.

7 Plans for the Future

In addition to the projects described above, we are also working to improve the documentation and support for HcryptoJ, which is available at

`http://starbase.trincoll.edu/~crypto/`

Although HcryptoJ is presently limited to historical cryptography, our eventual goal is to move beyond historical algorithms into asymmetric algorithms and strong encryption, such as public-key encryption, and more sophisticated forms of cryptanalysis. We currently do not offer a cryptography course but our hope is to extend HcryptoJ in ways that would make it a suitable platform for a upper-level computing-based cryptography course. What we've learned from our past experience in this area is that one way to expand our curriculum in that direction would be to enlist the help of students, both in courses such as software engineering and in independent study projects, to design, program, document, and test the system as we develop it further.

8 Acknowledgements

The author would like to acknowledge the work of Emeritus Professor Ralph Walde, who has provided invaluable advice on the overall design and development of HcryptoJ and to the following Trinity students, who have worked on various parts of its design and development: Gregg Marcuccio, Carolyn Rucci, Van Hong Dao, William Servos, Jacob Wegner, Sophia Knight, Brian Hart, and Michelle Lombard.

9 References

1. Beker, H. & Piper, F. *Cipher Systems.* New York: John Wiley & Sons, 1967.

2. Java Cryptography Extension. URL: http://java.sun.com/products/jce/.

3. Kahn, David. *The Codebreakers.* New York: Macmillan, 1967.

4. Knudsen, Jonathan. *Java Cryptography.* Cambridge: O'Reilly & Associates, Inc., 1998.

5. Morelli, R., Walde, R. and Marcuccio, G. A java API for historical ciphers. *Proceedings of the Thirty Second SIGCSE Technical Symposium on Computer Science Education*, pp. 307-311, 2001.

6. Morelli,R. and Walde, R. A word-based genetic algorithm for cryptanalysis of short cryptograms. *Proceedings of the 2003 Florida Artificial Intelligence Research Symposium (FLAIRS)*, forthcoming, 2003.

10 Appendix: Source Code

```
package hcrypto.engines;

import hcrypto.cipher.*;
public class CaesarKey extends HistoricalKey {
    public final static String DEFAULT_KEY_DESCRIPTOR_PROMPT_STRING = "a positive integer";
    public final static String DEFAULT_KEYWORD_STRING = "5";
    private int shift;

    public CaesarKey() { } //  The default constructor is required.

    public void init(String keyspec) throws Exception {
        initKey(keyspec);                    // Inherited from superclass
        this.blocksize = 1;
        this.keyDescriptorPrompt = "positive integer";
        this.shift = Integer.parseInt(keyword);
        if (shift < 0)
            throw new Exception ("Invalid shift value for Caesar cipher: " + shift);
```

```
    }

    public int getShift() {
        return shift;
    }

    public String getAlgorithm() {
        return "Caesar";
    }
}

package hcrypto.engines;

import hcrypto.cipher.*;

public class CaesarEngine extends BlockCipher {

    private int shift;             // Caesar shift
    private CaesarKey key;

    public CaesarEngine() {
        alphabetRangeOptions = "111111";  // Caesar allows all 6 possible alphabet ranges
    }

    protected void engineInit(HistoricalKey hKey) throws Exception {
        if ( !(hKey instanceof CaesarKey))
            throw new Exception("InvalidKey: Caesar requires CaesarKey");
        key = (CaesarKey) hKey;
        alphabet = key.getAlphabet();
        cipherAlphabet = key.getCTAlphabet();
        shift = key.getShift();
        this.blocksize = key.getBlocksize();
    }

    public String engineEncode(String s ) throws Exception {
        if (blocksize != 1)
            throw new Exception("Invalid blocksize for Caesar cipher " + blocksize);
        char ch = s.charAt(0);
        if (alphabet.isInAlphabet(ch)) {
           return "" + encodeShift(ch, shift);
        }
        else
           return s;
    }

    public String engineDecode( String s ) throws Exception {
        if (blocksize != 1)
            throw new Exception("Invalid blocksize for Caesar cipher " + blocksize);
        char ch = s.charAt(0);
      if (cipherAlphabet.isInAlphabet(ch)) {
```

```
            return "" + decodeShift(ch, cipherAlphabet.getSize() - shift);
        }
        else
            return s;
    }
}
```