A Java implemented key collision attack on the Data Encryption Standard (DES)

John Loughran
NUI Maynooth
Co. Kildare
Ireland

Tom Dowling
NUI Maynooth
Co. Kildare
Ireland
tdowling@cs.may.ie

ABSTRACT

A Java implementation of a key collision attack on DES suggested by Eli Biham, [1], is discussed. Storage space minimization and fast searching techniques to speed up the attack are described. We also demonstrate the suitability of Java for large data cryptographic attacks and illustrate the extensive cryptographic features of the language.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classification—*JAVA*; E.3 [**Data encryption**]

General Terms

Algorithms, Security

Keywords

DES, Key collision attack

1. INTRODUCTION

The paper is organized as follows; We firstly give a brief overview of DES and outline how to implement and use DES with the aid of the Java Cryptographic Extension,(JCE). We then outline Biham's attack on DES. In the next section we describe the practical details of implementing this attack in Java and discuss the issues of storage and searching. We then briefly present the results of our implementations and finally discuss conclusions, possible generalizations, and future work.

2. THE DATA ENCRYPTION STANDARD, (DES)

DES, [6], is the most widely used example of a symmetric block cipher. Symmetric Block ciphers use the same secret information, or key, to encrypt and decrypt messages and messages are processed in blocks of fixed length. DES uses keys of effective length 56 bits which creates a possible 2^{56} distinct keys. Testing each of these keys until you find the right one is called a brute force attack. There are many examples, [2], of brute force attacks on DES but we will implement an attack that does not need to test all the keys. We can think of the DES algorithm as a black box as Java provides an easy to use interface to generate DES keys and encrypt or decrypt using the DES algorithm.

PPPJ '03 Kilkenny City, Ireland

3. JAVA CRYPTOGRAPHY AND DES

In this section we discuss a Java based framework, the Java Cryptography Architecture (JCA),which we will use to implement our DES cryptosystem. The discussion is based on the treatment of JCA presented in [5]. JCA is a framework that specifies design patterns for designing cryptographic concepts and algorithms. For example any mathematical algorithm that performs encryption is called a `Cipher`. The JCA architecture separates concepts from their implementations. These concepts are encapsulated by classes in the `java.security` and `javax.crypto` packages. For example the concept of a Cipher is represented by the `javax.crypto.Cipher` concept class. JCA relies heavily on the factory method design pattern to supply instances of its concept classes. A factory method is basically a special kind of static method that returns an instance of a class. A thorough discussion of the factory method design pattern appears in [3]. The idea here is that a concept class is asked for an instance that implements a particular algorithm. This is accomplished using a `getInstance()` factory method. The following code fragment demonstrates the process by producing an instance of the Cipher concept class that uses the DES algorithm:

```
Cipher cryptObject = Cipher.getInstance("DES");
```

One major advantage of this set up is that to change the cryptographic algorithm used you need only change the argument in the `getInstance()` method. We still have not explained how to implement the algorithms called via these factory methods. This implementation is accomplished by software vendors, or security providers, who write the implementations of the algorithms that plug into the JCA. These implementations are collectively called the Java Cryptographic Extension, JCE. The provider used in this paper was an Austrian implementation called IAIK. Details appear in [4]. The availability of this resource was one of the main reasons for implementing the attack in Java.

4. THE KEY-COLLISION ATTACK

Biham's attack involves the following important concepts:

1. A known plaintext header. Many encrypted messages begin with a known header. For example postscript files begin with `%!PS-Adobe-2.0`.

2. The Birthday Paradox. A full account of the Birthday Paradox appears in [7]. Very simply the Birthday Paradox is a statistical result which shows that given some property which has n possible values and given two sets, each of about $sqrt(n)$ entries selected randomly from the n entities, there is a high probability that some entity in the first set has the same value as some entity in the second set. Applying this to our attack, $n = 2^{56}$ and it is only necessary to have two collections of 2^{28} ciphertexts to have a high probability of finding a matching pair of ciphertexts. It may be instructive to enumerate here. The Birthday Paradox effectively reduces the amount of ciphertexts needed from 72,057,594,037,927,936 to 268,435,456. This makes implementation a whole lot easier and faster.

The Biham attack can be summarized as follows:

1. We encrypt the known plaintext header 2^{28} times, each time with a different randomly chosen DES key. These `(ciphertext,key)` pairs should then be stored in an appropriate data structure.

2. We then compare incoming ciphertexts of the same plaintext header, encrypted with an unknown key, with the ciphertexts above.

3. When a match is found the key value corresponding to the matching ciphertext is returned.

It is expected that a key will be found while it is still in use and that it can be used to decrypt the message or even substitute a message favorable to the attacker.

5. IMPLEMENTATION OF THE ATTACK

The first requirement of the attack is the generation of 2^{28} ciphertexts using randomly generated DES keys. This seems like a lot of work but Java is ideally suited to the task. Using the `Key` and `KeyGenerator` concept classes Java generates random DES keys as follows,

```
KeyGenerator keyGen = KeyGenerator.getInstance("DES");
keyGen.init(new SecureRandom());
Key key = keyGen.generateKey();
```

These keys are then fed into the DES Cipher to encrypt a known plaintext header. The header chosen was the postscript header `%!PS-Ado` which was 64 bits long. These ciphertexts need to be stored efficiently and this is discussed in the next section. Incoming ciphertexts then needed to be compared with the stored ciphertexts. Again this process is discussed in the next section. In order to save space we did not store DES Key objects but simply stored their values as `Long` integer objects. This caused another problem because when the key was eventually found it needed to be converted into a DES Key object and a simple constructor did not exist. This is because DES keys are designed to be generated randomly and not from specific values. The problem was solved by utilizing the lesser known `KeySpec` and `SecretKeyFactory` concept classes. Firstly the `Long` key is converted to a byte array then the concept classes are invoked and finally a cast to a Key object is performed,

```
KeySpec spec =
new DESKeySpec(ByteFormOfKeyValue);
SecretKeyFactory desFactory =
SecretKeyFactory.getInstance("DES");
SecretKey thisSecretKey =
desFactory.generateSecret(spec);
Key thisKey = (Key)thisSecretKey;
```

This `Key` object can now be passed to a DES Cipher object in order to encrypt or decrypt a message.

6. STORAGE, SPACE AND SEARCHING CONSIDERATIONS

In this section we discuss how to overcome the problem of storing and searching efficiently a set of 2^{28} `(ciphertext,key)` pairs. Each ciphertext string was 8 bytes long and each DES key object was 212 bytes. To store this set without any optimizations would require $2^{28} \times (8 + 212) \approx 55$ GB of space. Clearly we needed to reduce this load. The first optimization was to extract the key value from the key object as described above.

The next optimization was to pick an appropriate data structure to store the data. For various reasons, including the searching complexity of , $O(1)$ [8], the data structure chosen was a Hashtable. Java imposes a limit on the number of entries a Hashtable can have so various storage schemes were tested and refined until the following approach was adopted,

1. The `(ciphertext,key)` pairs were stored in 2^{14} different Hashtables.

2. During pair generation a number of Hashtables were kept in memory, and filled up until their combined size caused the system to slow significantly, (2^{22} pairs was the limit).

3. Each ciphertext and key value was stored in a particular Hashtable. The Hashtable chosen was the one with the same name as the `hashCode()` mod 16384 (2^{14}), of the ciphertext. Each Hashtable was in turn stored in an array of Hashtables indexed by its name. This strategy was chosen as it was quick to implement and promised to give an even spread of pairs between the different Hashtables. Initial testing showed this to be the case.

4. When a total of 2^{22} pairs were generated, the partly filled Hashtables were written to disk into files whose filenames contained the index of the Hashtable in its array. The idea was that when examining incoming ciphertexts, the hashCode of each ciphertext would be obtained, and mod 16384 of this would be the index. Then the Hashtable file whose name contained this index was read in from the hard disk, the Hashtable object extracted and searched for the ciphertext in question. This meant that for each incoming ciphertext only one file needed to be read in and searched, keeping the constant time complexity advantage of using Hashtables.

5. Because testing has shown that only 2^{22} pairs could be generated on each run of the program before writing to disk, the process was repeated 64 times storing the Hashtable files to a different directory each time. These mini-Hashtable files were then merged into 2^{14} Hashtable files each containing an average of 2^{14} pairs, giving a total of 2^{28} `(ciphertext,key)` pairs.

6. Testing showed that the Multiple Hashtables Approach was feasible although the generation process took about 36 hours. This does not affect the timing of the attack as this process can be done off-line before the attack commences.

7. This approach also had significant space savings with the space required at the end of the process to store the Hashtables being approximately 7.5 GB. This compares very favorably with the initial estimate of 55 GB.

   In order to test the attack we generated and stored 2^{28} ciphertexts generated with "unknown" keys. These ciphertexts simulated a stream of ciphertexts being captured by an attacker. Again to improve performance the ciphertexts were stored as follows; 2^{14} arrays were set up and named from 000000 to 016383. As each ciphertext was generated its hashCode mod 16384 was calculated and it was stored in the appropriate array. These arrays were then sent to 2^{14} files named after the hashCode value.

We can now describe the searching process.

Each in turn of the 2^{14} simulated unknown ciphertext files were read into memory at random. Each file contained an array of ciphertexts whose hashCode mod 16384 was the same.The name of the ciphertext array file, was then used to open the corresponding Hashtable file. The Hashtable was searched for each ciphertext in turn to the end of the array. If no match was found a new array file was read in from disk and the process repeated until all 2^{14} arrays were searched. Thus the number of times a Hashtable file needed to be read in from memory was reduced from 2^{28} to 2^{14}. This process resulted in an improved estimated time for searching for the key from 6.2 years down to 2.3 hours.

The results of this test are encouraging. The test was performed on a `Pentium 4` machine with clock speed `2.5 GHz`, `1 GB` of RAM, and a `60 GB` hard disk. During testing a key was found in 1.5 hours on average. The whole search of all arrays took an average of 2.3 hours.

7. CONCLUSIONS AND FUTURE WORK

The speed of the program is not earth shattering when compared with existing attacks on DES using dedicated hardware. However, the fact that it was possible in a reasonable timescale is due in part to the Biham attack and in part to strategies and techniques developed during the course of the project, many of which grew out of its inherent difficulties.

The Biham algorithm was successfully implemented in Java, with the help of the JCA. Biham's assertion that a key could be found by a comparison of 2^{28} ciphertexts with a stored table containing 2^{28} `(ciphertext, key)` pairs has been verified in Java . The fact that the algorithm can be implemented in Java and a key found in an expected average time of just over an hour is encouraging for a cryptanalyst. It is worrying for anyone using DES to encrypt sensitive information or data. As Biham explains in his paper, numerous files of the same type are commonly stored on modern computers with large hard disks. If these files are of the same type they are likely to have the same file header. Using DES to encrypt these files with different keys can in fact help an attacker to find a key as shown in this paper, rather than increasing their security as was previously believed. Thus DES cannot be described as a secure way to encrypt data for storage under these circumstances.

One might argue that finding a single key which depends on receiving messages transmitted over a network, encrypted with 2^{28} different keys, will be of limited use due to the time necessary to receive these messages. This is true in the implementation described here. However, by listening over a large enough network like the Internet, and forwarding messages of a certain type to a central location, it should be possible to collect 2^{28} ciphertexts in a short space of time. During this time they can be sorted into arrays as described in this paper ready for comparison with the corresponding Hashtable. Generation of the Hashtables is not a limiting factor as they can be pre-computed. Indeed given a number of attackers cooperating over a network, tables containing a total of more than 2^{28} pairs can be generated easily, with the result that fewer ciphertexts are needed for a successful attack.

This implementation can be applied equally to modern variants of DES including AES, albeit with greater complexity. The architecture of JCA makes this task relatively simple. The portability of Java makes this implementation suitable for a distributed attack over many machines that could considerably speed up the process.

8. REFERENCES

[1] Biham, E. *How to decrypt or even substitute DES-encrypted messages in* 2^{28} *steps*, Information Processing Letters **84**, (2002), 117-124.

[2] Electronic Frontier Foundation *Cracking DES-Secrets of Encryption Research, Wiretap Politics and Chip Design*, O'Reilly, 1998.

[3] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *DesignPatterns: Elements of Reusable Software*, Addision -Wesley, 1995.

[4] IAIK documentation *http://jcewww.iaik.at/products/jce/documentation /javadoc/index.html*

[5] Knudsen, B. *Java Cryptography*, O'Reilly, 1998.

[6] National Bureau of Standards *Data Encryption Standard*, U.S. Department of Commerce, FIPS pub**81**, (1980)

[7] Stallings, W. *Cryptography and Network Security*, Prentice Hall, 2003.

[8] Watt,D., Brown,D. *Java Collections*, Wiley, 2001.