Playing hide and seek with stored keys

Adi Shamir* and Nicko van Someren†

September 22, 1998

Abstract

In this paper we consider the problem of efficiently locating cryptographic keys hidden in gigabytes of data, such as the complete file system of a typical PC. We describe efficient algebraic attacks which can locate secret RSA keys in long bit strings, and more general statistical attacks which can find arbitrary cryptographic keys embedded in large programs. These techniques can be used to apply lunchtime attacks on signature keys used by financial institutes, or to defeat authenticode type mechanisms in software packages.

Keywords: Cryptanalysis, lunchtime attacks, RSA, authenticode, key hiding.

1 Introduction

In this paper we consider the problem of efficiently locating cryptographic keys in large amounts of data. As a motivating example, consider a financial institute which uses the manager's PC to digitally sign wire transfers. In our lunchtime attack scenario, the attacker (who can be a secretary, technician, customer, etc.) can sneak into the manager's office for a few minutes while he or she is away for lunch. We assume that the PC is off line, and cannot be directly used to sign unauthorized wire transfers. The goal of the attacker is to quickly scan the gigabytes of data on the hard disk in order to find the secret signature key. This key may be kept as a separate data file on the PC (due to overconfidence), or permanently embedded in the cryptographic application itself (due to poor design). Even worse, the key may be stored on the PC unintentionally and without the knowledge of its security conscious

*. Applied Math Dept., The Weizmann Institute of Science, Rehovot 76100, Israel.
Email: shamir@wisdom.weizmann.ac.il.

†. nCipher Corporation Limited, Cambridge, England.
Email: nicko@ncipher.com.

user. For example, the key may appear in a Windows swap file which contains the intermediate state of a previous signing session, or it may appear in a backup file created automatically by the operating system at fixed intervals, or it may appear on the disk in a damaged sector which is not considered part of the file system. We assume that the attacker can use a diskette to bring in a short program and to bring out the discovered key, but he does not have enough storage to copy the whole contents of the hard disk, and does not have enough time to try each subsequence of bits from the hard disk as a possible signature generation key.

Another example in which an attacker may wish to locate cryptographic keys in large files is in "authenticode" type applications. In many systems a software producer wishes to exercise some control over what code is run on a user's computer. There are many reasons for wanting to do this. A vendor might want to ensure that files have not been corrupted when being used in a mission critical system or that vendor might want to limit third party add-ons to ones it has authorised. If the application is a security sensitive one then it might be necessary to ensure that none of the security features have been subverted. If the application allows cryptographic extensions to be added, a government might insist that any extensions are authorised before they can be used. Clearly there are a number of reasons, both good and bad, for wanting code authentication.

As well as reasons for authenticating code there are also reasons, both good and bad, for wanting to bypass the authentication. A third party software producer might want to try to break the monopoly of the original author by providing add-ons that have not been authorised, or they may want to develop cryptographic extensions for use when they might not otherwise be available. A hacker might maliciously want to subvert the security of a secure system or damage the code in a safety critical system.

2 Finding Secret RSA Keys

In this section we assume that the attacker knows the public key n and e of an RSA[2] scheme used by his victim, and has temporary access to a long string of u bits (representing the full contents of the hard disk) which is known to contain the corresponding secret key d as a contiguous substring of v bits. A typical value of u can be 10^{10}, while a typical value of v can be 10^3.

The simplest solution to the problem (which is applicable to any cryptosystem) is to obtain a cleartext/ciphertext pair, and then to scan the long bit string and perform trial decryption with each subsequence of length v as a possible key. Rare false alarms can be discarded by trying additional

pairs. Ciphertext only attacks are also possible, but typically require more decryptions with each candidate key to identify the expected cleartext statistics. In public key cryptosystems, it suffices to know the victim s public key, since the attacker can generate by himself the required cleartext/ ciphertext pairs.

The main problem in applying this technique to the RSA scheme is that each modular exponentiation is very expensive, and its time complexity grows cubically with the size v of the modulus. If we have to try about u possible substrings as candidate values for the decryption exponent d, we get a total complexity of $O(uv^3)$, which is polynomial but impractical (about 10^{19} for the typical parameters mentioned above).

A faster algorithm is based on the observation that consecutive candidates for d have a huge overlap. When we move a window of size v over a string of size u, the contents of two consecutive windows can differ only in their first and last bits, and in the fact that their other bits are shifted by one bit position. When the contents of the two windows are interpreted as binary integers d' and d'', we can relate them via:

$$d'' = 2d' + c_1 - c_2 2^v$$

where c_1 and c_2 are either 0 or 1. Given a value of the form $m^{d'} \pmod n$, we can compute the value of $m^{d''} \pmod n$ by performing one modular squaring, and 0, 1, or 2 additional modular multiplications with precomputed numbers. Since the complexity of each modular multiplication is $O(v^2)$, the total complexity drops from $O(uv^3)$ to $O(uv^2)$, or about 10^{16} in our typical scenario.

Our next observation is that when the public exponent e is small, this result can be greatly improved. Small e such as 3 and $2^{16}+1$ are very common in software implementations of RSA, since they make the encryption and signature verification operations 2-3 orders of magnitude faster than full size exponents.

Consider the case of $e=3$. The secret exponent d is known to satisfy $3d=1 \pmod{\phi(n)}$, where $\phi(n)=(p-1)(q-1)=n-(p+q-1)$. We can thus conclude that $3d=1+cn-c(p+q-1)$ where c is either 1 or 2. The value of $(p+q-1)$ is unknown, but it contains only half as many bits as n. We can thus perform approximate division by 3, and get for each one of the two choices of c a candidate value for the top half of d. For the typical parameters, this implies that we can easily compute two candidate values for the top 500 bits of d. Such a large number of random bits makes it extremely unlikely that we will encounter false alarms, and thus we can use a straightforward string matching algorithm to search for the known half of d, and recover the other

half from any successful match. The time complexity of such an attack is just $O(u)$, and for all practical purposes it is only limited by the maximal data transfer rate of the hard disk.

This technique can be used for larger values of e, but its efficiency drops rapidly since the number of candidate values for the top half of d grows exponentially in the size of e. We now describe an alternative technique, which remains reasonably efficient for values of e whose binary size is smaller than half the size of n. The basic idea is to compute for each candidate substring d the value of $d\ e$-1. For the correct value d, the result is zero modulo $\phi(n)$. In other words, it is equal to $c.\phi(n)$ in which the multiplier c is smaller than half the size of n. When we reduce $d\ e$-1 modulo the known n instead of modulo the unknown $\phi(n)$, we get zero minus an error term which is somewhat smaller than n, i.e., a small negative value.

To use this observation, we consider two windows of length v in the given bit string of length u, which are shifted by a single bit position with respect to each other. Denote their numeric values by d' and d'', which are related by $d''=2d+c_1-c_2\ 2^v$. Assume that we have already computed $d'e$-1(mod n), and would like to compute $d''e$-1(mod n). Since c_1 and c_2 are single bit quantities, we need a constant number of additions/subtractions to carry out this computation. The algorithm can thus scan the whole bit string in time $O(vu)$, and announce any location which makes the computed result a small negative number, a candidate value for d. If e is sufficiently small (compared to half the size of n), there are likely to be no false alarms. This technique can be optimized further in a variety of ways, such as updating only the most significant bits of de-1(mod n) during the scan, and recomputing its precise value only infrequently in order to prevent excessive buildup of computational errors.

A completely different approach is to look for the secret primes p and q whose product is the known value of n. The signature generation procedure does not have to know these values in order to compute m^d(mod n), but in almost all the practical implementations of the RSA scheme the signature generation process uses these factors to speed up the computation by a factor of 4 by using the Chinese Remainder Theorem.

We make the reasonable assumption that p and q occur next to each other on the long bit string, and thus the distance between their least significant bits is about $v/2$. We can thus try to multiply any pair of substrings of length $v/2$ in which the second substring is shifted with respect to the first by $v/2+i$ bits for i=0,32,64, and compare the result to n. The total complexity of this approach is $O(uv^2)$. However, it can be reduced to just $O(u)$ by performing the test modulo 2^{32}, i.e., by multiplying the least significant words of p and

q, and comparing the bottom half of the result to the least significant word of n. Since multiplication of 32 bit numbers on a PC is a very fast basic operation, and the probability of false alarm is sufficiently small, the algorithm is quite practical.

3 Finding public keys

In the previous section we looked at finding the secret keys in the context of some sort of "lunchtime attack" . In this section we look at finding public keys (usually signature verification keys) with a view to subverting a public key infrastructure.

Consider the case of an "authenticode" system. While it is usually possible to completely disable all signature checking on code, it is rarely desirable to do so. If all checking is removed it may leave a system wide open to naïve attacks. A better method of bypassing code signature checking is to replace the signature verification key with a key of your own choosing. Of course if you can do this, someone else can too, but it can protect against a less able attacker.

The usual process for locating anything is to try to identify some characteristic of what is being located and then to look for that characteristic. One characteristic of cryptographic keys is that they are usually chosen at random. Most code and data is not chosen at random and it turns out that this differentiation is significant. When data is random it has higher entropy than patterned information that is not random. This means that we should be able to locate cryptographic keys among other data by locating sections with unusually high entropy.

During our work we considered one particular system which we knew to contain an RSA signature verification key. The system is a modular cryptographic application programming interface produced by a major software vendor and it is widely used in commercial applications. The file we suspected of holding the key was approximately 300 kilobytes and we had no information as to where the key might be.

3.1 Visual identification of high entropy regions

The human eye and human brain between them are very good at picking up on patterns. Since the majority of the data in programs have some structure, while we expect to see very little structure in key data we can pick out the location of the keys simply by *looking* at the data in some suitable representation. Figure 1 is a one bit per pixel image from part of the program data in

Figure 1 Key information (in the middle of the figure) looks more noisy than the rest of the data

the code authentication system. The middle section of the image contains the signature verification key and it is visibly more noisy than the surrounding data.

While visual inspection of the program data allows us to locate the keys in a body of data, it is rather slow and labour intensive. We can achieve the same result by more mechanical means.

3.2 Identifying keys by measuring entropy

Since we know that key data has more entropy than non-key data, one way to locate a key is to divide the data into small sections, measure the entropy of each section and display the locations where there is particularly high entropy.

While getting a true measure of entropy is a complex task, in practice the entropy of most program code is so low that a true measure is not needed. In our experiments we found that examining a sliding window of 64 bytes of data and counting how many unique byte values were used gave a good enough measure of entropy. Throughout the first body of code we worked on the average window of data contained just under 30 unique values (with a standard deviation of close to 10). The windows which covered the key data averaged 60 unique byte values; a full 3 deviations from the mean. In a body of 300 kilobytes of data only 23 windows had a 'score' greater than 50 and of these 20 were consecutive and corresponded to the location of the key data.

In the general case, where we are faced with locating a key of length v bits in a body of code made up of u bits we can find the areas of highest entropy with a complexity of order u, since our method does not depend on the key and can be performed using only linear passes of the data. Clearly the success of this statistical method depends on the nature of the program concerned.

4 Better methods of hiding keys

In the specific case of hiding RSA private keys, there are many countermeasures which can be used to make the described attacks less likely to succeed. The most obvious technique is to keep all the cryptographic keys on a detachable device such as a smart card, or to keep them encrypted under a strong memorized password on the hard disk. However, such a key must be used by the application in decrypted form during the signature generation process, and thus may be left in such a state somewhere in the PC's file system, as described in the introduction.

A simple example of one such countermeasure when e is small is to replace the standard decryption exponent d by an equivalent exponent of the form $d' = d + c\phi(n)$ for a moderately large c with several dozen bits. The user can directly use d' instead of d in his modular exponentiation operations, and its complexity grows by a negligible amount. The advantage of such a d' is that the attacker can no longer predict some of the bits of the decryption exponent just from the fact that the encryption exponent is small. However, it does not prevent the other attacks described in this paper.

Entropy based attacks to find key data can be resisted by matching the levels of entropy in non-key and key data. In practice we are concerned with the entropy density, not the total entropy; we must have the same amount of information over all but having the large concentration of entropy in the key data makes it easy to spot. We can achieve this either by trying to lower the entropy density of the key data or by raising the entropy of the other data.

We can lower the entropy density of the key by spreading the key out over more of the program. There are various options here. One way we could do this is to construct a set of values, each with relatively few bit value changes and thus with lower entropy, such that some simple combination of these values results in the key value we require. This works well in spreading the information but it incurs a computation overhead to set the key up. Another way would be to generate some code which when run results in the key value being placed into a buffer. Again, this requires some computational overhead but with luck this can be small compared to the computation to use the key.

There is another option for hiding the key, which can potentially not only do away with the computation overhead but also be more robust than other options. Consider that the key must be known at the time that the program is built. Given suitable tools we can present the key as a constant in the computation which is carried out using that key and then we can *optimise*

the code given that constant. This will cause the key to be intimately intertwined with the code which uses it. Not only will the resulting code look very much like normal code (making the key hard to find), but it may also make the computation run faster than if the key were placed in a separate memory buffer. Furthermore if the optimisation process is thorough it will likely be extremely hard to change the key without replacing the entire section of code which uses that key.

The other class of solutions for hiding the key is to make the entropy of the rest of the data appear higher. One way to do this is to encrypt the program so that it decrypts itself before it runs. Work has been carried out in this field by Intel Corporation[1] and others and it can lead to systems which are very hard to subvert but it does so at a cost. There will always be a computation overhead involved in decrypting the code and data before it is used and this will slow the system down.

5 Conclusions

The problem of efficient identification of stored secret keys in lunchtime attacks (as opposed to efficient computation of unknown secret keys by cryptanalysis) had received almost no attention in the literature so far, even though we believe that it poses a great threat to many enterprises with commercial grade physical security (such as banks, brokers, lawyers, travel agencies, etc.). Such attacks are particularly effective when a company sends its computers to a repair shop or sells them as junk, since it leaves no traces and there is no risk of detection (compared to attacks based on sneaking into the manager's room or installing a virus in his computer).

Our techniques seem to be applicable to a wide variety of other public key schemes, in addition to the RSA scheme. For example, in the Fiat-Shamir signature scheme, the secret key s is the square root of the public key a modulo n. A simple scan of the long bit string which checks for each candidate substring s' whether $s'^2 = a \pmod{n}$ has time complexity $O(uv^2)$. By using the algebraic relationship between any two consecutive candidates s' and s'', we can update the value of $s'^2 \pmod{n}$ into $s''^2 \pmod{n}$ in a constant number of addition/subtraction operations, and thus the total time complexity can be reduced to $O(uv)$. We are now in the process of developing similar attacks on other public key cryptosystems.

The problem of keeping a "public" key secret has also received little attention even though a great many public key infrastructures place huge value on a small number of root public keys. If computer programs must be operated in an hostile environment they need to have some form of

protection. While it is relatively easy to build tamper resistant hardware it is much harder to protect computer software. It should be observed that re-keying a code authentication scheme is an attack on the Public Key Infrastructure rather than an attack on the cryptosystem. Over the years we have seen that attacking the PKI is often by far the most efficient way to break public key cryptosystems and this is no exception.

References

[1] David Aucsmith. Tamper resistant software. *Lecture Notes in Computer Science: Information Hiding*, 1174:317{333, 1996.

[2] R.L. Rivest, A. Shamir, and L.M. Adleman. Cryptographic communications system and method. U.S. Patent, 1983. U.S. Patent no. 4,405,829.