Some Words About Cryptographic Key Recognition In Data Streams

Alexey Chilikov   Evgeny Alekseev[*]

November 29, 2011

Abstract

Search for cryptographic keys in RAM is a new and prospective technology which can be used, primarily, in the computer forensics. In order to use it, a cryptanalyst must solve, at least, two problems: to create a memory dump from target machine and to distinguish target cryptographic keys from other data. The latter leads to a new mathematical task: «recognition of cryptographic keys in the (random) data stream». The complexity of this task significantly depends on target cryptoalgorithm. For some algorithms (i.e. AES or Serpent) this task is trivial but for other ones it may be very hard. In this work we present effective algorithms of expanded key recognition for Blowfish and Twofish. As far as we know this task for these algorithms has never been considered before.

1 Introduction

Decryption of encrypted data is a classical problem of cryptanalysis. Now this problem is very practical in the digital forensics. There are many computer applications that use strong cryptographic algorithms with strong keys. Generic cryptanalytic methods (like an exhaustive key search) are infeasible in this case and the strength of used algorithms resists other attacks. As a result, forensic experts need other ways to solve this problem which may be not «purely mathematical» but use mathematics, physics and organize aspects «in complex».

One of the prospective ways to solve this problem is a search for encryption keys in the target machine's RAM (i.e. «live-memory analysis»). This problem has been discussed in many recent works ([2, 3, 4, 5]). Most of them describe different ways to get physical access to RAM. There are many methods: preinstalled software, using DMA via connection protocols (e.g. Firewire, [6]), restoring RAM from the hybernation file, «ColdBoot», etc.

Anyway an analyst faces the problem of «encryption key recognition» in the extracted RAM image. Typically, an analyst knows some information for verification of target encryption keys. It may be known signatures of decrypted

[*]Passware, Research Department, {chilikov,alekseev}@passware.com

file system (for «whole-drive encryption») or known patterns in the target file format. But the exhaustive search for keys in RAM is a very expensive process. Fast recognition of tiny numbers of «suspicious» candidiates in a large data stream could significantly reduce the time and cost of the expertise.

One of the interesting methods for key recognition is the «expanded key method». It was discussed in [1] in respect to search for Bitlocker keys in RAM. The main idea of this method is the following: typically, symmetric encryption algorithms expand the small «encryption key» to the array of «round keys» which are used in the sequential encryption rounds. This array is much larger than the original key. As a result there is a redundancy that can be used to distinguish «expanded keys» from other data blocks.

Often this way works much faster than an exhaustive search. Another advantage is that this way doesn't depend on the implementation of the algorithm in the target application but depends on the algorithm only. As a result an expert can use this method for a wide class of applications without expensive «reverse-engineering process».

But this way significantly depends on the target algorithm. From this point of view, there are different classes of algorithms: «simple», «hard» and «impossibly hard». The latter type is not interesting in practice but we believe that it could be designed. The two other classes are more interesting. For «simple» algorithms the original key is a part (typically, start values) of the expanded key. In this case the «recognition problem» is trivial – an analyst must expand start keys to a few rounds and check if the calculated values are equal to the values from the checked array (from RAM). The example of «simple» algorithm is AES.

The algorithm is «hard» in our classification if expanded keys are derived from original key through complex nonlinear scheme and the the array of expanded keys does not contain the original key. Typically, dependencies between different round keys are non-trivial too. Examples of «hard» algorithms are Twofish ([7]) and Blowfish ([8]).

In this work we present the solution to the problem of expanded key recognition in (random) data stream for Twofish and Blowfish. Some previous works discussed this task for concrete implementations but as far as we know there is no proposed «purely algorithmic» solution to this problem which could use only expanded keys properties. Proposed results have been published before only in russian ([9, 10]).

The structure of our work is the following: for each target algorithm we will start from its brief description and after that we will describe the proposed attack and provide its complexity estimations.

2 Blowfish Keys Recognition

2.1 Blowfish (448 bits) Key Expansion

Let's consider the key expansion only for maximal key length – 448 bits. Block size of Blowfish is 64 bits. The «word» is an array of 32 bits (which represents 32-bit unsigned integer). As a result «word» contains 4 bytes. «Encryption Key» K is an array of 14 words.

Expanded keys contain two parts: array P (18 words) and array S (1024 words). The total size of the expanded key is $(18 + 1024) \cdot 4 = 4168$ bytes.

Initially, both arrays contain constants ($\hat{P}$ and $\hat{S}$) which are determined in the algorithm specification. For specified encryption key these arrays are updated The update procedure is described by the following pseudocode:

```
// Stage 0: Key Addition
for( i = 0; i < 18; i++ )
  P[ i ] = P[ i ] ^ K[ i mod 14 ]
// Stage 1: Generating of P
word block[ 2 ] = { 0, 0 }
( P[ 0 ], P[ 1 ] ) = encryptBlock( block )
for( i = 2; i < 18; i+=2 )
  ( P[ i ], P[ i+1 ] ) = encryptBlock( P[ i-2 ], P[ i-1 ] )
// Stage 2: Generating of S
( S[ 0 ], S[ 1 ] ) = encryptBlock( P[ 16 ], P[ 17 ] )
for( i = 2; i < 1024; i+=2 )
  ( S[ i ], S[ i+1 ] ) = encryptBlock( S[ i-2 ], S[ i-1 ] )
```

The function *encryptBlock(. . .)* performs Blowfish-like encryption of data block. More precisely, all operations are the same to Blowfish but use the current state of P and S. In other words, each call of *encryptBlock(. . .)* is slightly different from the previous one and the next one.

2.2 Search of Blowfish Expanded Keys

For different techniques of physical access to RAM some restrictions on the gained image could occur. For example, it may be represented as an unordered set of physical pages. The problem of mapping of physical addresses to logical addresses space may be hard. In this case expanded keys may be fragmented, i.e. lie on two or more non-adjacent pages. The size of Blowfish expanded keys is more than the size of the physical memory page (4096 bytes). As a result the fragmentation of expanded keys is very likely.

However, all expanded keys can be recovered from array P only. Really, for recovering S array we can «encrypt» predefined array $\hat{S}$ via *encryptBlock(. . .)* function with known P. As a result we need to find in data stream only 18 words of P. Hence the following problem occurs: to recognize if fixed array of 18 words is a correct array P (for some original Blowfish key).

The size of checked array (18 words or 72 bytes) is more than the original key size (448 bits or 56 bytes). As a result P array has a redundancy. The probability of «false alarm» is 2^{-128} approximately (if data is random). But we need an efficient algorithm of checking «correctness». We are going to propose such an algorithm below.

Let us know some pairs of plaintext and ciphertext. Then «naive» algorithm of correctness checking is the following:

1. Calculate full expanded key (P, S) from checked array P
2. Try to decrypt ciphertext and compare the result with the known plaintext

Note that this way can work (with smaller probability) if we know only a small part of plaintext or only some relations between plaintext bits. For this algorithm we need to call *encryptBlock(...)* at least $4 \cdot 4 \cdot 256/8 = 512$ times for generating S. Then for an exhaustive search in 4 GiB of RAM we need to perform approximately 2^{41} blocks encryptions. It seems too hard and we are going to propose a better way.

For optimization we can use two steps:

1. Firstly use some «fast correctness checking» for 18 word arrays
2. Use full verification procedure (by knowledge about plaintext and ciphertext pairs) only for «possibly correct» arrays

This way is better than «naive» approach if and only if there is «fast correctness checking» algorithm (faster than 512 block encryptions per key). We will consider these algorithms further.

Note that there is another reason to use a two-step approach. Correctness checking uses only memory image and doesn't use encrypted data. It may be an advantage if we can't have an access to RAM and data storage (e.g. HDD) simultaneously.

2.3 Fast Correctness Checking for P

Let X be a checked 18-word array. We want to recognize if it's a P-part of the expanded key for an original Blowfish key.

For many software implementations of Blowfish parts P and S of the expanded key lie consequently. In this (simplest) case the checking procedure is the following:

1. Call *encryptBlock(...)* of $\hat{S}$ with checked values of X as P
2. Find resulted block in the same RAM page with X

It works successfully if S-part of expanded keys lies after (may be nonconsequently) P-part on one physical page of RAM. In this case we need to perform only 1 block encryption (instead of 512). The probability of «false alarm» (for random data) is approximately $2^{12}/2^{64} = 2^{-52}$ per checked key or totally 2^{-20} for 4 GiB of random RAM data.

If S-part of the expended key doesn't lie after P-part on the same page we must use more complex checking. It is described in the next section.

2.4 Original Key Recovery and Full Correctness Checking for P

When we know P-part of expanded key we can recover the original key K. During this process we can precisely check a «correctness» of P-part.

Consider the block encryption in detail. It works as follows:

```
$E_{P,S}(X)$ :
$(L_0, R_0) = (X[0] \oplus P[0], X[1])$
$for(i = 1; i < 17; i++)$ :
    $(L_i, R_i) = (R_{i-1} \oplus P[i] \oplus F(L_{i-1}, S), L_{i-1})$
$(L_{17}, R_{17}) = (R_{16} \oplus P[17], L_{16})$
$return(L_{17}, R_{17})$
```

Function F depends on S but doesn't depend on P. But during stage 1 $S = \hat{S}$ and is known.

Note that known $P[16]$ and $P[17]$ are the result of encryption of known $P[14]$ and $P[15]$. And all other $P[i]$ for the last encryption are known ($P[0], \ldots, P[15]$ as a part of checked P-array and $P[16]$ and $P[17]$ as a part of initial array $\hat{P}$. As a result we can calculate all intermediate values L_i and R_i. Now we have the relations:

$$\begin{aligned} P[16] &= L_{17} \oplus K[17 \mod 14] \oplus \hat{P}[17] \\ P[17] &= F(L_{17}) \oplus K[16 \mod 14] \oplus \hat{P}[16] \oplus R_{17} \end{aligned}$$

and we can easily reconstruct appropriate $K[2]$ and $K[3]$.

Other $K[i]$ can be reconstructed in a similar way. First R and L are calculated «forward» by known $P[i]$ from the checked array and last R and L are calculated «backward» by known $\hat{P}[i]$ from the initial array. Complexity of «forward» and «backward» rounds are the same because it's Feistel network. Total complexity of recovering two key words is equal to one block encryption.

We need to recover totally 18 words of the key. But for the last two rounds we can check «correctness». Really, $K[3]$ is reconstructed two times: at first round as $K[17 \mod 14]$ and at last-but-one round as $K[3 \mod 14]$. Both values must be the same (for «correct» P-part). The situation is the same for $K[2]$. As a result after 8 «pseudo-encryptions» we have 64 control bits for checking. For a stronger check we can use $K[1]$ and $K[0]$. The probability of a false alarm is approximately 2^{-128}. The total complexity is approximately 8 block encryptions (9-th «pseudo-encryption» is required very rare). It's easy to see that this process is $\approx 512/8 = 64$ times as fast as «naive» approach (but 8 times as slow as «fast check» from previous section).

3 Twofish Keys Recognition

3.1 Twofish (256 bits) Key Expansion

Let's consider Twofish key expansion for maximal key length – 256 bits (32 bytes). As before the «word» is an array of 32 bits (which represents 32-bit unsigned integer). «Encryption key» M is and array of 32 bytes. The expanded key contains 40 words $K_0, \dots, K_{39}$ and 4 S-boxes which depend on the encryption key.

In order to expand key M, firstly two 4-word vectors are derived: $M_e = (M_0, M_2, M_4, M_6)$ and $M_o = (M_1, M_3, M_5, M_7)$, where $M_i = (m_{4i+3}, m_{4i+2}, m_{4i+1}, m_{4i})$.

After that S-boxes are derived but it is not important for further analysis and we don't provide it here.

The core of Twofish key expansion is non-linear function $h(X, (L_0, L_1, L_2, L_3))$ which handles 5 words X, L_0, L_1, L_2, L_3 and returns the word Z. The first step of h is calculation of intermediate vector $Y = (y_0, y_1, y_2, y_3)$ by the following formulae:

$$\begin{aligned}
y_0 &= q_0[q_1[q_1[q_0[q_1(x_0) \oplus l_{3,0}] \oplus l_{2,0}] \oplus l_{1,0}] \oplus l_{0,0}] \\
y_1 &= q_1[q_1[q_0[q_0[q_0(x_1) \oplus l_{3,1}] \oplus l_{2,1}] \oplus l_{1,1}] \oplus l_{0,1}] \\
y_2 &= q_0[q_0[q_1[q_1[q_0(x_2) \oplus l_{3,2}] \oplus l_{2,2}] \oplus l_{1,2}] \oplus l_{0,2}] \\
y_3 &= q_1[q_0[q_0[q_1[q_1(x_3) \oplus l_{3,3}] \oplus l_{2,3}] \oplus l_{1,3}] \oplus l_{0,3}]
\end{aligned}$$

Here x_i is i-th byte of X, $l_{i,j}$ is j-th byte of L_i. q_0 and q_1 are non-linear invertible byte permutations.

The second step of h is calculating Z as a result of multiplication of Y on fixed invertible MDS-matrix M over field $\mathbb{F}_{256}$.

The expanded key words $K_0, \dots, K_{39}$ are derived from M_e and M_o by the following formulae:

$$\begin{aligned}
\varrho &= 2^{24} + 2^{16} + 2^8 + 1 \\
A_i &= h(2i\varrho, M_e) \\
B_i &= h((2i+1)\varrho, M_o) <<< 8 \\
K_{2i} &= (A_i + B_i) \mod 2^{32} \\
K_{2i+1} &= (A_i + 2B_i) \mod 2^{32} <<< 9
\end{aligned}$$

where $<<<$ means cyclic rotation of 32-bit word.

3.2 Original Key Recovery

The expanded key for Twofish is relatively small (160 bytes) and it's very likely to find all K_i on the same physical page. We are going to consider an efficient algorithm of recovering the original key M by known array $K = (K_0, \dots, K_{39})$.

Firstly note that we can easily recover all values of $h(2i\varrho), M_e)$ and $h((2i +$

$1)\varrho, M_o)$. Really

$$\begin{array}{lcl} T & = & K_{2i+1} <<< 23 \\ h(2i\varrho, M_e) & = & (2K_{2i} - T) \mod 2^{32} \\ h((2i+1)\varrho, M_e) & = & (T - K_{2i}) \mod 2^{32} <<< 24 \end{array}$$

for each $i = 0, \dots, 19$. Furthermore, matrix M and permutations q_0, q_1 are easily invertible. Then for each $i = 0, \dots, 39$ we can calculate:

$$\begin{array}{l} k_{i,0} = q_1[q_1[q_0[q_1(x_0) \oplus l_{3,0}] \oplus l_{2,0}] \oplus l_{1,0}] \oplus l_{0,0} \\ k_{i,1} = q_1[q_0[q_0[q_0(x_1) \oplus l_{3,1}] \oplus l_{2,1}] \oplus l_{1,1}] \oplus l_{0,1} \\ k_{i,2} = q_0[q_1[q_1[q_0(x_2) \oplus l_{3,2}] \oplus l_{2,2}] \oplus l_{1,2}] \oplus l_{0,2} \\ k_{i,3} = q_0[q_0[q_1[q_1(x_3) \oplus l_{3,3}] \oplus l_{2,3}] \oplus l_{1,3}] \oplus l_{0,3} \end{array}$$

For even i $l_{i,j}$ are bytes of M_e and for odd i $l_{i,j}$ are bytes of M_o.

How can we recover $l_{0,0}, \dots l_{3,0}$ (i.e. least significant bytes of 4 words of M_e)? We must use relations for even i. There are 20 known bytes $k_{0,0}, k_{2,0}, \dots, k_{38,0}$ and 20 relations

$$k_{2i,0} \oplus f(2i, (l_{3,0}, l_{2,0}, l_{1,0})) = l_{0,0}$$

where

$$f(2i, (l_{3,0}, l_{2,0}, l_{1,0})) = q_1[q_1[q_0[q_1(x_0) \oplus l_{3,0}] \oplus l_{2,0}] \oplus l_{1,0}]$$

Consequently, for each (i, j) we have

$$k_{2i,0} \oplus k_{2j,0} = f(2i, (l_{3,0}, l_{2,0}, l_{1,0})) \oplus f(2j, (l_{3,0}, l_{2,0}, l_{1,0}))$$

Note that values of $f(2i, (l_{3,0}, l_{2,0}, l_{1,0}))$ for fixed i depend on 24 bits only and can be precalculated fast and stored in a small amount of memory (near 16 MiB for each index i). But a more convenient way is to precalculate values of $k_{ij} = f(2i, (l_{3,0}, l_{2,0}, l_{1,0})) \oplus f(2j, (l_{3,0}, l_{2,0}, l_{1,0}))$ for each $(l_{3,0}, l_{2,0}, l_{1,0})$ and store them in the table which is ordered by k_{ij}. For any k there are 2^{16} appreciate vectors $(l_{3,0}, l_{2,0}, l_{1,0})$ on average (or 1/256 part of all vectors). If the checked array is «correct» and if $k_{ij} = k_i \oplus k_j$ for calculated $k_0, \dots, k_{19}$ then for corresponding tables for all k_{ij} contain at least one common value (true $(l_{3,0}, l_{2,0}, l_{1,0})$). Otherwise, for «incorrect» array it is highly unlikely that the tables for four values $k_{01}, k_{02}, k_{03}, k_{04}$ contain a common element (probability of «false alarm» is approximately $2^{24}/2^{4\cdot 8} = 2^{-8}$). It's a rather good way to check an array fast. For a stronger checking we can use another k_{0j}. Final probability of «false alarm» is approximately $2^{24-8\cdot 19} = 2^{-128}$.

Checking is much faster if we precalculate all compatible values of $(k_{01}, k_{02}, k_{03}, k_{04})$ and store them with corresponding tuples $(l_{3,0}, l_{2,0}, l_{1,0})$ in the sorted table. The total size of this table is $7 \cdot 2^{24}$ bytes (112 MiB). Checking is equivalent to search for an element in a table (24 comparisons). If correct tuple $(l_{3,0}, l_{2,0}, l_{1,0})$ is found $l_{0,0}$ is easily calculated as $f(0, (l_{3,0}, l_{2,0}, l_{1,0})) \oplus k_{0,0}$.

For M_o and $k_{2i+1,j}$ the situation is the same.

4 Conclusions

Recognition of cryptographic keys in the data stream has been considered very rarely in previous works. In common case this problem seems to be very hard. However in some practically important cases it can be solved efficiently. It's very actual, firstly, in the digital forensic practice. Methods which are proposed in this work, are successfully applied for analysis of wide-spread cryptographic systems (e.g TrueCrypt, PGP, Bestcrypt, etc). Our results are implemented in special forensic tools (Passware Kit Forensic). We believe that similar methods can be used for reverse engineering and security analysis of computer systems.

Further research of key recognition may be used to increase resistance of computer systems against side-channel attacks.

References

[1] *J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten.*
Lest We Remember: Cold Boot Attacks on Encryption Keys.
http://citpsite.s3-website-us-east-1.amazonaws.com/oldsite-htdocs/pub/coldboot.pdf.

[2] *Brian Kaplan.*
RAM is Key. Extracting Disk Encryption Keys From Volatile Memory.
Carnegie Mellon University. May 2007. Thesis Report.

[3] *Tobias Klein.*
All your private keys are belong to us. Extracting RSA private keys and certificates from process memory.
http://www.trapkit.de/research/sslkeyfinder/keyfinder_v1.0_20060205.pdf.

[4] *Adi Shamir and Nicko van Someren.*
Playing hide and seek with stored keys.
Lecture Notes in Computer Science, Vol. 1648, p. 118–124, 1998.

[5] *Carsten Maartmann-Moe, Steffen E. Thorkildsen, André Årnes.*
The persistence of memory: Forensic identification and extraction of cryptographic keys.
DFRWS 2009, http://www.dfrws.org/2009/proceedings/p132-moe.pdf.

[6] *Adam Boileau.*
Hit by a Bus: Physical Access Attacks with Firewire.
Ruxcon 2006.

[7] *Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson.*
Twofish: A 128-Bit Block Cipher.
http://www.schneier.com/paper-twofish-paper.pdf.

[8] *Bruce Schneier.*
Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993), Springer-Verlag, 1994, pp. 191-204 .

[9] *А. Чиликов, Е. Алексеев.*
Поиск криптографических ключей в RAM.
http://www.ruscrypto.ru/netcat_files/File/ruscrypto.2011.042.zip.

[10] *Чиликов А. А., Алексеев Е. К.*.
Распознавание криптографических ключей в RAM.
Системы высокой доступности. 2011. Т. 7, No.2. С. 42 - 46.