Malicious Cryptography Techniques for Unreversable (malicious or not) binaries

Eric Filiol
filiol@esiea.fr

ESIEA - Laval
Operational Cryptology and Virology Lab $(C+V)^O$

H2HC 2010 - Sao Paulo & Cancun November 27-28th, 2010

Outline

Introduction

- Preventing code (binaries) analysis is a critical issue:
  - Protection of industrial secrets.
  - DRM and copyright enforcement.
  - Fighting against software piracy.
  - Hinder code undertanding as long as possible (malware context).
- Two main approaches known:
  - Code encryption.
  - Code obfuscation.

Introduction (2)

Code Obfuscation

Make machine code difficult to understand. Used to conceal code logic, to prevent tampering, deter reverse engineering (security through obscurity principle).

- May induce anti-debugging, anti-decompilation and anti-disassembly mechanisms.
- Known as theoretically impossible techniques
  - Case of black-box obfuscation (Barak et al. - 2001).
- In practice... well it is not so obvious.
  - On Best-possible Obfuscation (Goldwasser - Rothblum 2007)
  - What about white-box model (Josse - Eicar 2008)?
- Does not necessarily increase the data entropy.

Introduction (3)

Code encryption

Code encryption transforms native binary code (plaintext) into random data (ciphertext) by means of an encryption algorithm and a secret key. The process must be reverseable to come back to the native code upon execution.

- Increase data entropy (close to random data).
  - Very easy to identify and detect, even locally.
- The secret key is somewhere in the code: just find it and decipher.
- The encryption routine, even protected, can be used as an oracle.
- In practice... well it is not so obvious to enforce strong code encryption.

Our aim

- Imagine new methods to protect code from analysis and reverse-engineering.
- Use the power of malicious cryptography.
- We want the analyst to be unable
  - to distinguish encrypted data from non encrypted data (TRANSEC aspect).
  - to interpret data (COMSEC issue).
- Provide a high level of code mutation (poly/metamorphism) while preserving the previous unabilities (add more confusion).
- Protect against both static (disassembly/decompilation) and dynamic (debugging, functional analysis, sanboxing...) analysis at the same time.

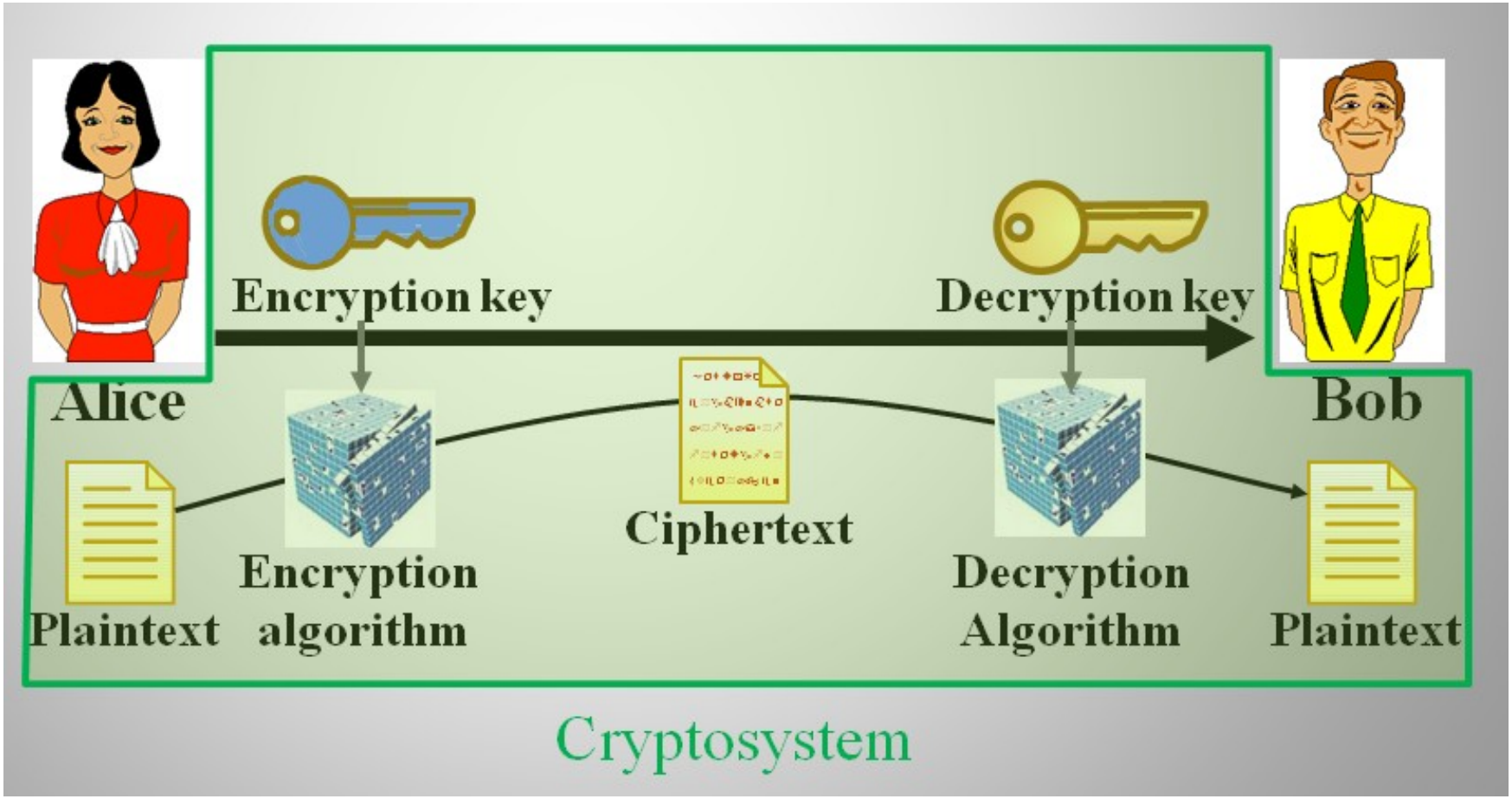
Project Status

- Our techniques are about to be implemented in the AndroGuard project (http://code.google.com/p/androguard/).
- Already implemented in the LibThor library (available soon on http://libthor.avcave.org)
- Source code and data presented in this talk are provided upon request
- Technical paper available in `http://arxiv.org/abs/1009.4000`

Agenda

Outline

What a cryptosystem really is?



Code Mutation: polymorphism/metamorphism

Code mutation

Ability for a binary code to change wholly (metamorphism) or partly (polymorphism) in order to remove as much as possible code invariants. Polymorphism aims at bypassing static analysis while metamorphism aims additionally at preventing behavior-based detection.

- To achieve efficient code mutation, critical instructions must be changed to prevent the analyst to rely on code invariants.
- CFG instructions are primarily concerned (change the course of execution and the way to change it).
- Invariants we want to get rid of:
  - Critical sequences of bytes (contiguous or not).
  - Behavior (time-indexed meta-patterns), functional traces...

Malicious Cryptography

Emerging domain initiated in (Filiol & Josse, 2007; Filiol & Raynal, 2008; Filiol, 2010).
Covers different fields:

- Use cryptography to build totally undetectable and invisible malware (*Über-malware*).
- Use malware to perform cryptanalysis operations:
  - steal secret keys or passwords,
  - manipulate encryption algorithms on-the-fly to weaken them dynamically and temporarily,
  - **modify the cryptographic environment in the target computer**.
- Design of encryption algorithms with hidden trapdoors.

This is the interconnection of computer virology with cryptology and mathematics for mutual benefit.

Cryptographic Environment Manipulation

Let us consider an arbitrary encryption algorithm E. Three main techniques can be used (Filiol, 2010):

1. Choose an arbitrary pair (P, C) and design a suitable pair (E', K') such that $C = E'(K', P)$ (resp. $P = E'(K', C)$), where K' is purposedly weak.
   - $\Rightarrow$ use a malware to replace E with E'.
2. Choose an arbitrary (E, C, P) and compute K such that $C = E(K, P)$ (Filiol, 2006).
3. Modify an arbitrary algorithm on-the-fly (e.g. with a malware)
   - modify E in E' to add some arbitrary 3-tuple (P', C', K') in the working domains of E. Thus we have $C' = E'(K', P')$ (resp. $P' = E'(K', C')$) while still having $C_i = E'(K, P_i)$ (resp. $P_i = E'(K, C_i)$). for almost all legitimate pairs (C_i, P_i).

Entropy Profile

Shannon entropy

Shannon entropy is a measure of information disorder or more precisely of information unpredictability.

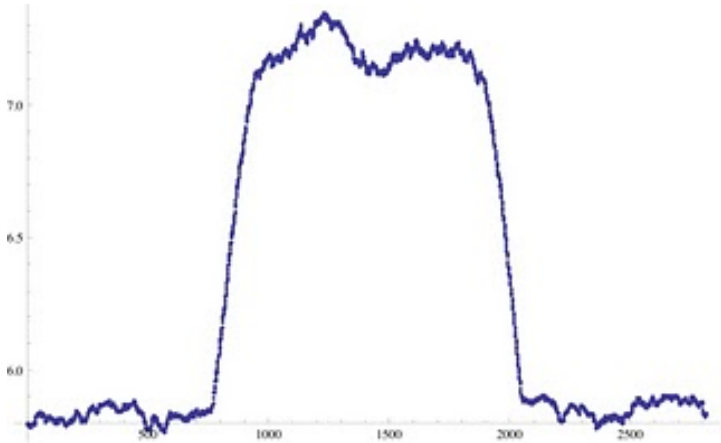
- Let us consider an information source X. When parsed, the source outputs characters $x_i \quad (i = 0, \ldots, 255)$ with probability $p_i = P[X = x_i]$. The source entropy is given by

$$H(X) = \sum_{i=0}^{255} -p_i \log_2(p_i)$$

- Random, compressed or encrypted data will exhibit a high entropy value.

Entropy Profile (2)

- Native executable (unprotected): average entropy $H(X) = 5.099$
- Packed executable: average entropy $H(X) = 6.801$
- Encrypted executable: $H(X) = 7.175$
- Detecting local entropy is straighforward (E. Carrera's tools)



- COMSEC vs TRANSEC.
- Existing solutions: steganography or Perseus technology (iAWACS 2010).

Key Management

- Using encryption requires to manage secret quantities (keys).
- The knowledge of the key and of the encryption algorithm gives a total access to the code.
- The main issue lies in the fact that the key is always somewhere hidden in the code.
- Just find it and decipher.

Key Management (2)

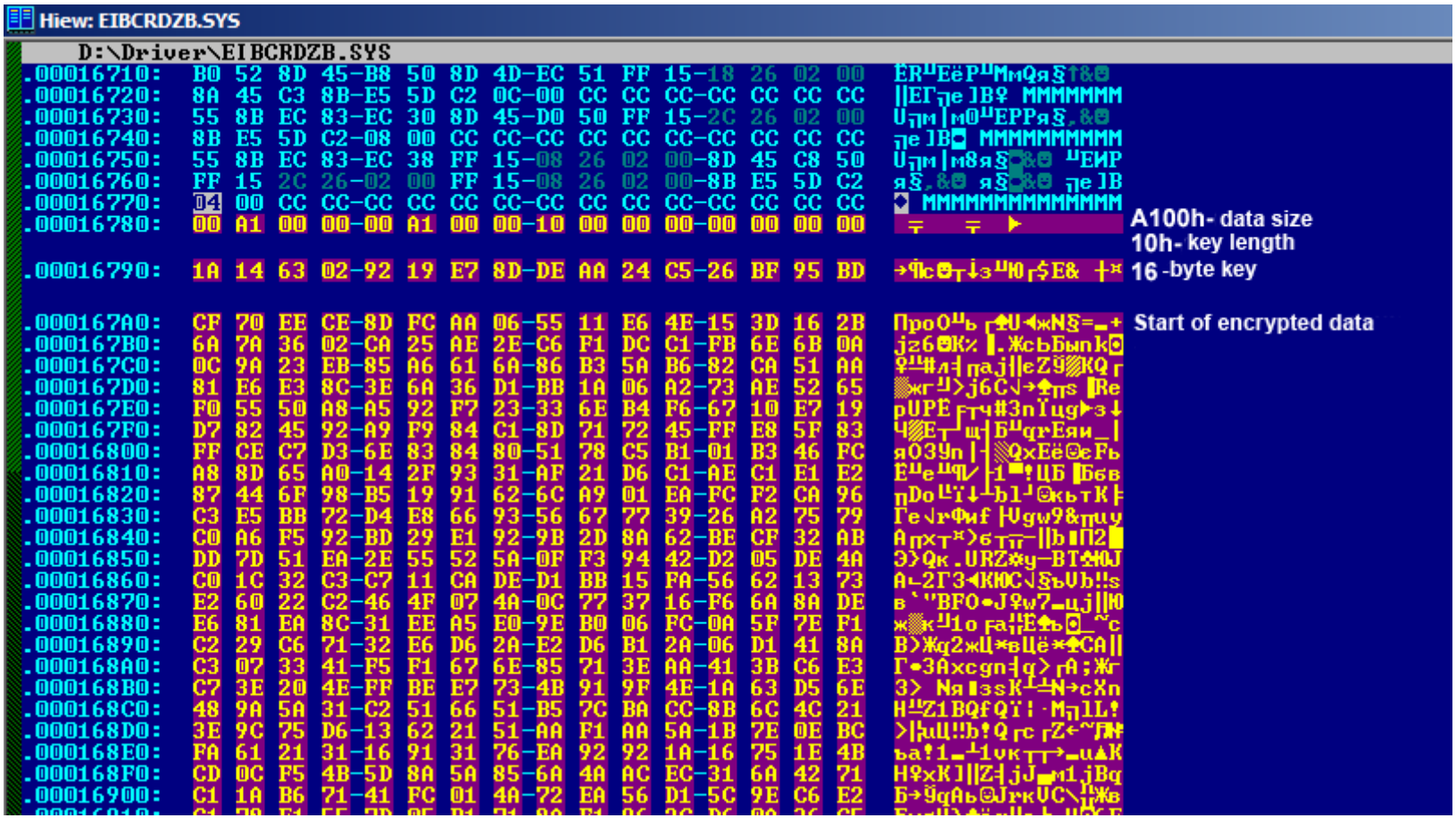
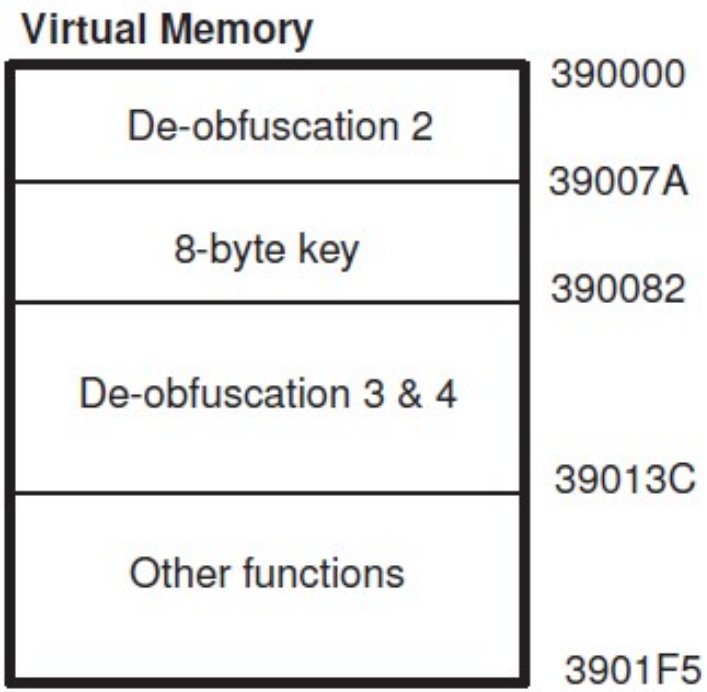


Figure: Black Energy Malware (Tarakanov - 2010)

Key Management (3)



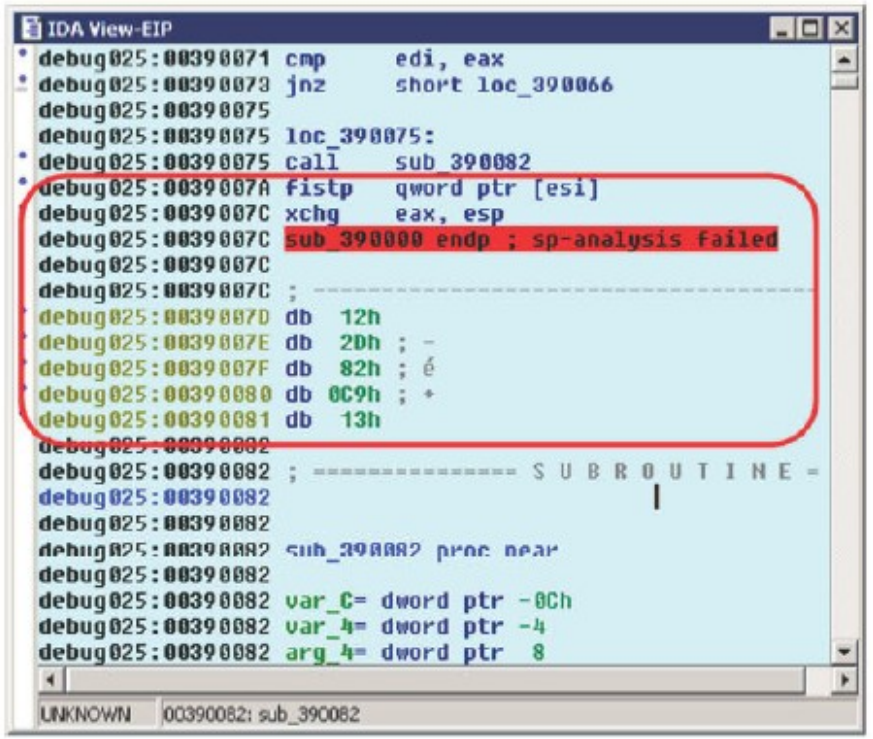


Figure: Zeus botnet (Binsalleeh et al., 2010)

New Trends as Solutions

- Change entropy profile while preserving security
  - Use of malicious random generator/encryption algorithm.
- Change the way to generate/manage keys
  - Environmental key generation and management (Riordan & Schneier, 1998; Filiol, 2005/2007)
  - K-ary viruses (Filiol 2007 - Desnos 2009)

Outline

Malicious PRNG

- Sophisticated polymorphic/metamorphic and obfuscation techniques must rely on PRNG (Pseudo-Random Number Generator).
  - Our aim is to generate sequences of random numbers (here bytecode values) on-the-fly while hiding the code behavior.
- Sequences are precomputed and we have to design a generator which will afterwards output those data.
- Three cases are to be considered:
  1. the code is built from any arbitrary random sequence (refer to the technical paper);
  2. the sequence is given by a (non yet protected) instance of bytecode and we have to design an instance of PNRG accordingly (non published yet);
  3. **Produce random data that can be somehow interpreted by a fixed PRNG as meaningful instructions (like `jump 0x89`) directly.**

Malicious PRNG (2)

- We fix some arbitrary sequence of code (bytecode, opcode, operands...)

$$X_0, X_1, \ldots, X_n$$

- we want to hardcode them under a protected (obfuscated) form given by

$$K_0, K_1, \ldots, K_n$$

  - The K_i and X_i values must have the same, low entropy profile
- Whenever the code wants to execute (protected) instruction K_i, it inputs it to the dFSM which outputs the value X_i.

Malicious PRNG (3)

- We then have to find a deterministic Finite State Machine (dFSM) acting as a malicious PRNG such that we have

$$X_0 = dFSM(K_0), X_1 = dFSM(K_1) \dots X_i = dFSM(K_i) \dots$$

- either fix value X_i and find out the "key" K_i for an arbitrary dFSM,
- or for an arbitrary set of pairs (X_i, K_i) design a unique suitable dFSM for those pairs.
- We are going to present the first case

Malicious PRNG (4)

Security requirements

- The analyst can only access to the malware code which contains unprotected data U_i and protected code (K_i values) but he must not be able to distinguish U_i from the K_i values.
- He has no access to the dFSM and thus cannot use it as an oracle (see further, implementation issues).
- The dFSMs we design must be "malicious" enough to be used for code mutation purposes at the same time,
  - a single X_i value must be produced from many different possible K_i values
- Generalization of obfuscation through obscure predicate (the dFSM is such a predicate in itself).

Case Study

- Without loss of generality, case drawn from Desnos' Libthor and Androguard library (obfuscation through polymorphic VMs).
  - $[X86ASM] \rightarrow [REILIR] \rightarrow [BYTECODES]$
- Let us consider the following piece of (critical) code we want to protect from reversing (extract).

```
[X86 ASM] MOV EAX, 0x3 [B803000000]
[REIL IR] STR (0x3, B4, 1, 0), (EAX, B4, 0, 0)
[BYTECODES] 0xF1010000 0x40004 0x3 0x0 0x6A
```

- Values at 0x00 contributes directely to the TRANSEC aspect (very low entropy profile).

Case Study (2)

Five fields in the bytecode:

- 0xF1010000
  - 0xF1: the opcode of the instruction (STR),
  - 0x01: specifies that it is an integer value,
  - 0x00: useless with respect to this instruction,
  - 0x00: specifies that it is a register.
- 0x40004
  - 0x04: the size of the first operand,
  - 0x00: useless with respect this instruction,
  - 0x04: the size of the third operand,
- 0x3: direct value of the integer,
- 0x0: useless with respect to this instruction,
- 0x6A: value of the register.

Case Study (3)

- In the rest of the talk, we take the following (critical) bytecodes we intend to protect from the reversing and analysis.

```
0x2F010000 0x040004 0x3 0x0 0x89 (1)
0x3D010000 0x040004 0x3 0x0 0x50 (2)
0x5010000 0x040004 0x3 0x0 0x8D (3)
```

- In real cases, we would consider far more instructions (as an example, all the CFG instructions).

Security Model

Our technique works under two specific requirements:

- The encryption algorithm code we use, must never be accessible to the analyst.
  - he can analyze it and understand its internals/principle!
  - So obfuscate it or better, use k-ary codes: the algorithm is deported into a different file, out of access for the analyst.
- The analyst cannot access to the encryption algorithm as an oracle
  - Black-box ability denied. He cannot send inputs and observe corresponding outputs without being detected by the encryption algorithm.
  - The encryption algorithm then behave differently and send wrong output to fool the analyst.

Security Model (2)

Security assumption

Our techniques work efficiently provided that these two conditions are ALWAYS fullfilled!

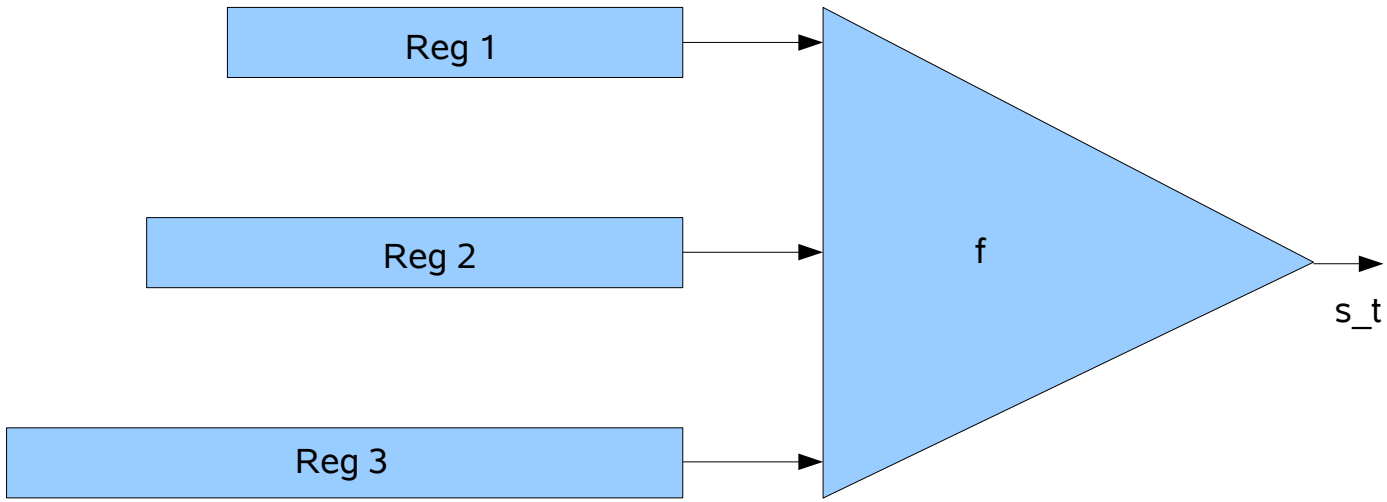
- We will assume that they will be (see further, implementation issues)!
- Remark: if we can prevent human analysis we will put any detection software in check!

Outline

Our Malicious PRNG

Without loss of generality, let us consider a 59-bit key stream cipher.

- Three linear feedback shift-register R_1, R_2 and R_3 of length 17, 19 and 23
- One combination function



- We have considered other kind of encryption system (stream and block) with larger key size (available upon request).

Implementation issues

- Implemented as

```
void sco(unsigned long long int * X,
unsigned long long int K)
{
/* K obfuscated value (input),
X unobfuscated value (output) */
...
}
```

- Two major cases:
  - Either the dFSM outputs critical data under a concatenated form
  - Or data segmentation is preserved (32-bits values).

Concatenated form

- The five 32-bit integer sequence (1) is processed as a unique 160-bit quantity

$$0x2F010000\ 0x040004\ 0x3\ 0x0\ 0x89$$

$$\rightarrow 0x2F010000040004000000030000000000000089$$

- At the implementation level we break this 160-bit quantity into three 59-bit integers (note that 59 is the entropy of our dFSM; see further) M_1, M_2 and M_3:

$$M_1 = 0x0BC04000000LL$$

$$M_2 = 0x08000800000060LL$$

$$M_3 = 0x00000000000089LL$$

Concatenated form (2)

- To transform the K_i values (obfuscated form in the code) into (unobfuscated values) X_i

```
/* Generate the M_i values */
sco(&M_1, K_1);
sco(&M_2, K_2);
sco(&M_3, K_3);
X_1 = M_1 >> 10; /* X_1 = 0x2F010000L */
X_2 = ((M_2 >> 37) | (M_1 << 22)) & 0xFFFFFFFFL
/* X_2 = 0x00040004L */
X_3 = (M_2 >> 5) & 0xFFFFFFFFL; /* X_3 = 0x3 */
X_4 = ((M_3 >> 32) | (M_2 << 27)) & 0xFFFFFFFFL;
/* X_4 = 0x0 */
X_5 = M_3 & 0xFFFFFFFFL; /* X_5 = 0x89 */
```

Concatenated form (3)

- We must (pre-)compute the K_i values outputing the M_i values.
- Cryptanalytic step (not addressed here; too much mathematics)
  - This step cannot be achieved in less than 30 minutes!
- For sequence (1), we get for instance

$$K_1 = 0x6AA006000000099LL$$

$$K_2 = 0x500403000015DC8LL$$

$$K_3 = 0x0E045100001EB8ALL$$

Concatenated form (4)

- For sequence (2), we get for instance

$$M_1 = 0x0F4040000000LL \quad K_1 = 0x751436000053C0LL$$

$$M_2 = 0x0800080000000060LL \quad K_2 = 0x4C07A200000A414LL$$

$$M_3 = 0x0000000000000050LL \quad K_3 = 0x60409500001884ALL$$

- For sequence (3), we get for instance

$$M_1 = 0x01404000000LL \quad K_1 = 0x76050E00001F0B1LL$$

$$M_2 = 0x0800080000000060LL \quad K_2 = 0x00000010C80C460LL$$

$$M_3 = 0x000000000000008DLL \quad K_3 = 0x0000000750980 31LL$$

Results Analysis

- The code interpretation is not straightforward since code/data are no longer aligned
  - The dFSM entropy must not be a power of 2!
- The entropy of quantities K_i is close to that of values X_i (no local entropy pick).

Unicity distance

The unicity distance UD is the minimal size for a dFSM output to be produced by a single secret key

- The design of our dFSM has been carefully designed to have $UD > H(K) = 59$.
- Consequently a large number of 59-bit keys can output an arbitrary output sequence
- Enable a huge poly/metamorphic power.

Results Analysis (2)

Serie	M_i values	Number of secret keys K_i
(1)	M_1	314 (file res11)
(1)	M_2	2,755 (file res12)
(1)	M_3	8,177 (file res13)
(2)	M_1	319 (file res21)
(2)	M_2	2,755 (file res22)
(2)	M_3	26,511 (file res23)
(3)	M_1	9,863 (file res31)
(3)	M_2	2,755 (file res32)
(3)	M_3	3,009 (file res33)

Table: Number of possible keys for a given output value

Results Analysis (3)

- This implies that we can randomly select our 9 M_i values and thus we have

$$\begin{aligned} & 314 \times (2,755)^3 \times 8,177 \times 319 \times 26,511 \times 9,863 \times 3,009 \\ = \quad & 13,475,238,762,538,894,122,655,502,879,250 \end{aligned}$$

  different possible code variants ($\approx 2^{103}$ variants).
- Files ($resij$ with $i, j \in \{1, 2, 3\}$) are freely available upon request.
- The mutation engine in this case has size less than 400 Kb.

Results Analysis: code mutation

- C code of mutation:

```
f1 = fopen("res11","r");
f2 = fopen("res12","r");
f3 = fopen("res13","r");
randval = (314.0*(rand()/(1 + RAND_MAX));
for(i = 0; i < randval; i++)
fscanf(f1,
K_1 = y1 | (y2 << 17) | (y3 << 36);
/* do the same for values M_2 and M_3 of serie (1) */
....
/* repeat the same for series (2) and (3) */
....
/* Generate M_1 value for series(1) */
sco(&M_1, K_1);
```

Non concatenated form

- In this case, the dFSM outputs 59-bit chunks of data whole only the 32 least significant bits are useful.
- We consider five 32-bit values M_1, M_2, M_3, M_4 and M_5 instead of three.
- For sequence (1), for instance we get

$$M_1 = 0x???????2F010000LL$$

$$M_2 = 0x??????00040004LL$$

$$M_3 = 0x??????00000003LL$$

$$M_4 = 0x??????00000000LL$$

$$M_5 = 0x??????00000089LL$$

where? describes any random nibble.

- Recover the X_i values with $X_i = M_i \quad \& \quad 0xFFFFFFFFL;$

Non concatenated form (2)

- Provides better TRANSEC features.
- Increased code mutation capability
  - Around 2^{140} 5-tuples $(K_1, K_2, K_3, K_4, K_5)$ whenever input in our dFSM produces the same set of three 5-tuples $(X_1, X_2, X_3, X_4, X_5)$ (sequences (1) to (3)).
  - With only three 160-bit sequences of bytecode, it is possible to have a huge poly/metamorphic power.
- When considering more complex structures (a 100-instruction CFG for instance) we obtain more than 2^{4000} obfuscated variants.

Outline

Operational Implementation: Constraints

The previous scheme indeed enables code unreversibility provided that

- The analyst has no access to the encryption algorithm code which remains unknown to him.
- He must not be able to use it as an oracle (otherwise it could brute-force the code and submit K_i values repeatedly).
- In order to fullfill all constraints we use k-ary codes (Filiol 2007; Desnos 2009)

K-ary Malware (Filiol 2007)

The viral information is no longer contained in a single code as usual malware do, but it is split into k different innocent-looking (not all executables eventually) files whose combined action - serially or in parallel - results in the actual malware behavior. Three possible classes A, B and C.

Operational Implementation: k-ary Codes

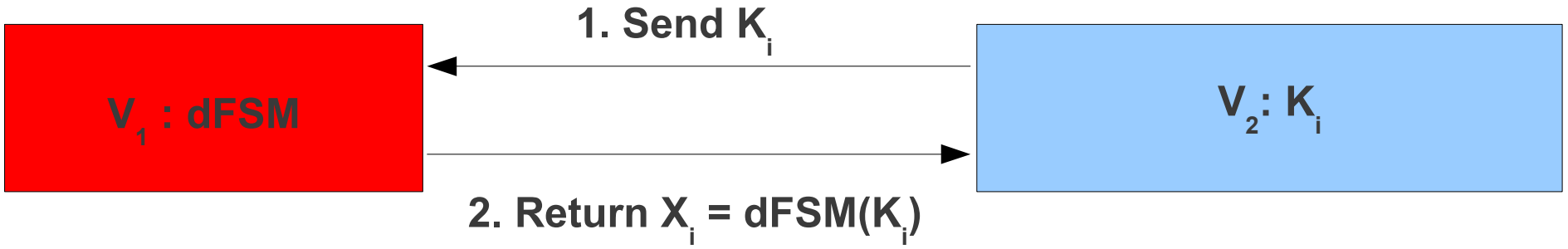


Figure: 2-ary implementation of malicious PRNG

Different implementations considered (among many others possible).

Communication pipes Only parallel class A or C k-ary codes can be implemented. Not the most optimal solution.

Named communication pipes K-ary parallel class B codes can be efficiently implemented (the most powerful class: no reference in any part to other any part).

System V IPC This is the most powerful method since everything is located into shared memory.

Operational Implementation: security

To prevent the analyst to use the V_1 part as an oracle, this latter must be able to detect that V_2 is currently in a sandbox or any other virtual environment.
We combine

- Cryptographic tricks (combinatorial integrity, environmental keys, zero-knowledge authentication protocols...)
- τ-obfuscation techniques (Beaucamps - Filiol, 2008).
  - Any virtualization always induces time delay that are statistically detectable (hence the use of zero-knowledge challenges between V_1 and V_2.
- Virtual environment detection (Filiol, 2007; Desnos - Filiol - IvanleF0u, 2009).

Whenever V_1 suspects V_2 to be executed in a virtual environment, fake data are sent to fool the analyst.

Operational Implementation: applications

Aside malware, this scheme is currently implemented for software protection, software watermarking anf fingerprinting.

- Part V_1 can be external and accessible only through a network application.
- Part V_1 can be buried deep inside in kernel-land while V_2 remains in user-land.
- Part V_1 can be installed in a smart card.

Industry support and development for this technology provided by DFT Technologies Inc.

DFT Technologies

Secure data, everywhere.

Outline

Conclusion and Future Works

- This scheme, under the implementation constraints we have defined ensures actual unreversability of protected binaries.
- Many applications are possible for program protection purposes.
- Current research and experiments consider
  - Variable (polymorphic) dFSM: a given sequence of bytecode can be protected by a lot of different dFSM which are computed by V_1
  - Non determinism: the dFSM behaves differently according to its input (from V_2).
- The power of malicious cryptography is unlimited.
- To be continued...

Many thanks for your attention.

Questions and answers!