Malicious Cryptology and Mathematics

Eric Filiol
Laboratoire de Cryptologie et De Virologie Opérationnelles
ESIEA
France

1. Introduction

Malicious cryptology and malicious mathematics is an emerging domain initiated in Filiol & Josse (2007); Filiol & Raynal (2008;b). It draws its inspiration from crypto virology Young & Yung (2004). However this latter domain has a very limited approach of how cryptography can be perverted by malware. Indeed, their authors consider the case of extortion malware in which asymmetric cryptography is only used inside a malware payload to extort money in exchange of the secret key necessary to recover the file encrypted by the malware (e.g. a computer virus).

Malicious cryptology and malicious mathematics make in fact explode Young and Yung's narrow vision. This results in an unlimited, fascinating yet disturbing field of research and experimentation. This new domain covers several fields and topics (non-exhaustive list):

- Use of cryptography and mathematics to develop "*super malware*" (*über-malware*) which evade any kind of detection by implementing:
  - Optimized propagation and attack techniques (e.g. by using biased or specific random number generator) Filiol et al. (2007).
  - Sophisticated self-protection techniques. The malware code protects itself and its own functional activity by using strong cryptography-based tools Filiol (2005b).
  - Sophisticated auto-protection and code armouring techniques. Malware protect their own code and activity by using strong cryptography.
  - Partial or total invisibility features. The programmer intends to make his code to become invisible by using statistical simulability Filiol & Josse (2007).
- Use of complexity theory or computability theory to design undetectable malware.
- Use of malware to perform cryptanalysis operations (steal secret keys or passwords), manipulate encryption algorithms to weaken them on the fly in the target computer memory. The resulting encryption process will be easier to be broken Filiol (2011).
- Design and implementation of encryption systems with hidden mathematical trapdoors. The knowledge of the trap (by the system designer only) enables to break the system very efficiently. Despite the fact that the system is open and public, the trapdoor must remain undetectable. This can also apply to the keys themselves in the case of asymmetric cryptography Erra & Grenier (2009).

One could define malicious cryptology/mathematics as the interconnection of computer virology with cryptology and mathematics for their mutual benefit. The number of potential applications is almost infinite. In the context of this chapter, we could also define it – or a part of it – as the different mathematical techniques enabling to modify or manipulate reality and to reflect a suitable but false image of reality to the observer (may it be a human being or an automated system).

In this chapter we intend to present in more details a few of these techniques that are very illustrative of what malicious cryptography and malicious mathematics are. Section 2 first recalls a few definition and basic concepts in computer virology and in cryptology to make this chaper self-contained. In Section 3, we expose a detailed state-of-the-art of malicious cryptology and malicious mathematics. We then detail two of the most illustrative techniques in the two next sections. Section 4 addresses how mathematical reality can be perverted to design processor-dependent malware. Section 5 then exposes how malicious cryptosystems can be used to protect malware code against detection and analysis.

2. Basic definitions and concepts

2.1 Computer virology

A rather large definition of what malware (shortened for of *Malicious Software*) are, here follows.

Definition 1. *A* malware *is a malicious code or unwated piece of software like a virus, a worm, a spyware, a Trojan horse... whose aim is to undermine systems' confidentiality, integrity or availability.*

In a more formal approach, malware are programs that take data from the environment (computer, system, users..) as input argument and output one or more malicious actions: file erasing, data eavesdropping, denial of services... A detailed and technical presentation of what malware are, is availble in Filiol (2005).

We will address the problematic of anti-antiviral techniques that are used by malware. Indeed, most of the malicious cryptology and malicious mathematics techniques aims at providing such capabilities to malware. It is logical that the latter enforce techniques to prevent or disable functionalities installed by antiviral software or firewalls. Two main techniques can be put forward:

- *Stealth techniques*.- a set of techniques aiming at convincing the user, the operating system and security programs that there is no malicious code. Malware then aim to escape monitoring and detection
- *Polymorphism/metamorphism*.- As antiviral programs are mainly based on the search for viral signatures (scanning techniques), polymorphic techniques aim at making the analysis of files – only by their appearance as sequence of bytes – far more difficult. The basic principle is to keep the code vary constantly from viral copy to viral copy in order to avoid any fixed components that could be exploited by antiviral programs to identify the virus (a set of instructions, specific character strings).

  Polymorphic techniques are rather difficult to implement and manage and this is precisely where lies the critical aspect of designing powerful malicious techniques drawn from both mathematics and cryptology. Two following main techniques (a number of complex variants exist however) are to be considered:

- Code rewriting into an equivalent code. From a formal point of view any rewriting technique lies on one or more formal grammar. According to the class of the grammar considered, then the malware protection is more or less stronger.
- Applying encryption techniques to all or part of malware code. Generally, those encryption techniques consist in masking every code byte with a constant byte value (by means of XOR). Any valid encryption technique implies the use of a static key that eventually constitutes a true signature (or infection marker) when ill-implemented. Moreover any skilled reverse-engineer will always succeed in extracting this static key and hence will easily unprotect the malware source code.

- *Code armouring* Filiol (2005b) consists in writing a code so as to delay, complicate or even prevent its analysis. While polymorphism/metamorphism aims at limited/preventing automated (e.g. by an antivirus software) analysis, code armouring techniques's purposes is to limit or bar the reverse engineer (a human being) analysis.

2.2 Cryptology

2.2.1 Cryptosystems

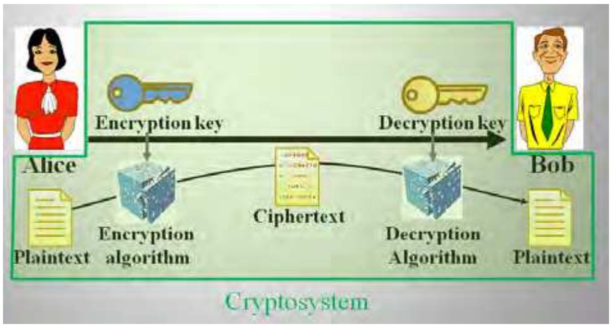

Fig. 1. General structure of a cryptosystem

A cryptosystem S (symmetric case) is defined as the set of an encryption/decryption algorithm E, a secret K, a plaintext message P and a ciphertext C. Let us recall that in the case of asymmetric cryptography (also well-known as public-key cryptography), the decryption and encryption algorithms are different, as well as the encryption and decryption keys. Asymmetric cryptography is mainly used for authentication and digital signature purposes but it can also be used to encrypt small quantities of information (a few dozen bytes). On the contrary, in the symmetric case, the key K and the algorithm E are the same. Symmetric cryptography is considered for encryption purposes. The plaintext is supposed to be secret while the cipher text is supposed to be accessed by any one. So with those notations, we have

$$C = E(K, P) \text{ and } P = E(K, C)$$

From a technical point of view, the internal operations (especially with respect to the key) may slightly differ according to the different classes of cryptosystems. But to summarize, any cryptosystem can be defined as a complex combination of substitutions and transpositions of bytes or string of bytes.

Cryptanalysis is the art of breaking cryptosystems; in other words, the attacker wants to access the plaintext P without the *a priori* knowledge of the key. This access can be performed directly through plaintext recovery or indirectly through key recovery which then enables the attacker to decipher the cipher text.

2.2.2 Entropy profile

Most of the problem regarding the use of cryptography for malicious purposes lies in the fact that code armouring and code mutation involve random data. These must be generated on-the-fly. In the context of metamorphism, the generator itself must be random too. For sake of simplicity, we shall speak of *Pseudo-Random Number Generator* (PRNG) to describe both a random number generator and an encryption system. The difference lies in the fact that in the latter case either random data produced from the expansion of the key are combined with the plaintext (stream ciphers) or they are the result of the combination of the key with the plaintext (block ciphers).

The whole issue lies in the generation of a so-called "good" randomness. Except that in the context of malicious cryptography Filiol (2007), the term "good" does not necessarily correspond to what cryptographers usually mean. In fact, it is better – yet a simplified but sufficient reduction as a first approximation – to use the concept of entropy Filiol & Raynal (2008). In the same way, the term of random data will indifferently describe the random data themselves or the result of encryption.

Consider a (malicious) code as an information source X. When parsed, the source outputs characters taking the possible values $x_i \quad (i = 0, \ldots, 255)$, each with a probability $p_i = P[X = x_i]$. Then the entropy $H(X)$ of the source is the following sum[1]:

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

Random data, by nature will exhibit a high entropy value thus meaning that the uncertainty is maximal whenever trying to predict the next value output by the source X. On the contrary, non random data exhibit a low entropy profile (they are easier or less difficult to predict).

From the attacker's point of view the presence of random data means that something is hidden but he has to make the difference between legitimate data (e.g. use of packers to protect code against piracy) and illegitimate data (e.g. malware code). In the NATO terminology – at the present time it is the most precise and accurate one as far as InfoSec is concerned– random data relate to a COMSEC (*COMmunication SECurity*) aspect only.

For the attacker (automated software or human expert), the problem is twofold: first detect random data parts inside a code and then decrypt them. In this respect, any code area exhibiting a high entropy profile must be considered as suspicious. To prevent attention to be

[1] Let us note that here the entropy considers single characters or 1-grams only. A more accurate value would consider all the possible n-grams and would compute entropy when $n \to \infty$.

focused on those random parts, is it possible to add some TRANSEC (*TRANSmission SECurity*) aspect. The most famous one is steganography but for malware or program protection purposes it is not directly usable (data cannot be directly executed) and we have to find different ways. The other solution is to use malicious statistics as defined and exposed in Filiol & Raynal (2008). It is also possible to break randomness by using noisy encoding techniques like in Perseus technology.

Breaking randomness applies well on any data used for code mutation (e.g. junk code insertion), including specific subsets of code as CFGs (*Comtrol Flow Graphs*): randomly mutated CFG must exhibit the same profile as any normal CFG would. Otherwise, considering the COMSEC aspect only is bound to make the code detection very easy.

2.2.3 Key management

Encrypting a code or a piece of code implies its preliminary deciphering whenever it is executed. But in all of the cases – except those involving money extortion introduced Young and Yung Young & Yung (2004) – the key must be accessible to the code itself to decipher. Consequently in a way or another it is contained in a more or less obfuscated form inside the code. Therefore is it accessible to the analyst who will always succeed in finding and accessing it. Instead of performing cryptanalysis, a simple decoding/deciphering operation is sufficient.

It is therefore necessary to consider keys that are external to the encrypted code. Two cases are possible Filiol (2007):

- Environmental key management. The code gathers information in its execution environment and calculates the key repeatedly. The correct key will be computed when and only when the suitable conditions will be realized in the code environment – which is usually under the control of the code designer. The security model should prohibit dictionary attacks or environment reduction attacks (enabling reduced exhaustive search) by the code analyst. Consequently the analyst must examine the code in an controlled dynamic area (sandbox or virtual machine) and wait until suitable conditions are met without knowing when they will be. However it is possible to build more operational scenarii for this case and to detect that the code is being analyzed and controlled.
- Use of k-ary codes (Filiol (2007c)) in which a program is no longer a single monolithic binary entity but a set of binaries and non executable files (working in a serial mode or in a parallel mode) to produce a desired final (malicious or not) action. Then the analyst has a reduced view on the whole code only since generally he can access a limited subset of this k-set. In the context of (legitimate) code protection, one of the files will be a kernel-land module communicating with a userland code to protect. The code without the appropriate operating environment – with a user-specific configuration by the administrator – will never work. This solution has the great advantage of hiding (by outsourcing it), the encryption system itself. It is one particular instance with respect to this last solution that we present in this paper.

3. The state-of-the-art of malicious cryptology and malicious mathematics

As any recently emerging field, the history of malicious cryptology and malicious mathematics is rather short. Young and Yung seminal work Young & Yung (2004) has stirred up almost no further research in this area. The main reason is that this new field relates to techniques generally considered by the “bad guys”, in other word the attackers (which include militaries, spies...).

Publishing such results is never easy since it addresses the critical issues of result reproducibility at least to enable the process of peer-review: any result whose reality must be verified and confirmed is likely to leak enough data and information allowing thus a misuse by the attacker. This situation hence does not ease the emergence of such a new research field. The first consequence is that the actual corpus of knowledge regarding malicious mathematics and malicious cryptography is likely to be far more extended that has been already published. Most of those techniques are known and used by the military and the intelligence domains. As a result, only a very few seminal papers (both theoretical and technical) are know at the present time. The second consequence is that most of the results are published in the most famous international hacking conferences which are far more reluctant at publishing theoretical and practical results in this area. In this respect, academics are losing interesting opportunities.

3.1 From mathematics and cryptology to computer virology

3.1.1 The origins

The mid 1930s could be reasonably considered as the starting point of the history of malicious mathematics. Indeed, the masterpiece Kleene’s recursion theorem in essence contains much of the computer virology concepts Filiol (2005). Of course neither Kleene nor followers in recursion theory, calculability, complexity theory have ever imagined that their results could be used and interpreted in a malicous way.

During World War II then many studies have been performed that could be considered as first premises of malicious mathematics. Von Neuman’s work, for exemple, dedicated to *Self-reproducing automata* – in other words viruses – is undoubtly the best case. As many such studies were at that time – due to the war[2] and then due to cold war era – it is very likely that many results were classified and still are nowadays. For instance, NSA released in 2010 only research works performed in computer virology. The case of cryptology – a subfield of mathematics and computer science – is probably even more emblematic since it has been considered as weapons, strictly controlled by governments and military until the early 2000s and is still under a strict control regarding for instance exports and knowledge dissemination (see the different national regulations and the Wassenaar Agreement Wassenaar Agreement (1996)). This the reason why seminal research works in the field of malicious mathematics and malicious cryptology are still very limited at least in the public domain.

Another famous case is the research work of Kraus (1980) which can be considered as the founding work of computer virology in 1980. The German government made suitable and efficient pressures to forbid the publication of this thesis. In this work, Kraus exposes theoretical and practical results in mathematics (recursive functions, computability and calculability theory) that actually has given birth to the modern computer virology Filiol

[2] von Neuman’s results which were established in 1948-1949, have been published in 1966 only, by his student R. Burk.

(2005). The last copy of Krause's manuscript has been by chance discovered in a wet cellar of the University of Dortmund, translated into English and published in 2009 only Kraus (1980).

3.1.2 Early stages of modern malicious cryptology

From an academic point of view, malicious cryptology begins with Young & Yung (2004) but with a very narrow vision. In their approach, a malware encrypts user's data with asymmetric cryptology public keys and extort money from the user who wants the key to access his data again. In this approach the malware itself does not deploy any malicious mathematics or malicious cryptology techniques. Only the payload uses cryptology for money extorsion. Young & Yung contribution, while having the merit to initiate the issue does not propose more than a cryptography book.

In 2005, the first academic description of the use of cryptology in the viral mechanism itself is described in Filiol (2005b). In this case the symmetric cryptology combined with environmental key management enables, for specific operational conditions, to protect malware code against reverse engineering. The concept of total code armouring is defined and experimented.

3.1.3 Formalization results

3.1.3.1 Testing simulability

When dealing with malicious attacks, the main problem that any has to face is detection by security software. The issue is then first to understand how security software are working in order to infer or to identify ways to bypass detection efficiently. In Filiol & Josse (2007), a theoretical study – backed by numerous experiments – describes in a unified way any detection technique by means of statistical testing. From this model, the concept of *statitical simulability* is defined. Here the aim is to reverse the use of statistics to design powerful attack that mimic normal behaviour. In this respect, we can speak of malicious statistics.

Definition 2. *Simulating a statistical testing consists for an adversary, to introduce, in a given population $\mathcal{P}$, a statistical bias that cannot be detected by an analyst by means of this test.*

There exist two different kinds of simulability:

- the first one does not depend on the testings (and their parameters) the defender usually considers. It is called *strong testing simulability*.
- on the contrary, the second one does depend on those testings that the attackers aims at simulating. It is called *weak testing simulability*.

Here we call "tester" the one who uses statistical testing in order to decide whether there is an attack or not, or whether a code is malicious or not...

Definition 3. *(Strong Testing Simulability) Let P be a property and T a (statistical) testing whose role is to decide whether P holds for given population $\mathcal{P}$ or not. Strongly simulating this testing consists in modifying or building a biased population P in such a way that T systematically decides that P holds on $\mathcal{P}$, up to the statistical risks. But there exists a statistical testing T' which is able to detect that bias in $\mathcal{P}$. In the same way, we say that t testings $(T_1, T_2, \ldots, T_t)$ are strongly simulated, if applying them* *results in deciding that P holds on $\mathcal{P}$ but does no longer hold when considering a $(t+1)$-th testing T_{t+1}.*

In terms of security, strong simulability is a critical aspect in security analysis. In an antiviral context, strong simulability exists when the malware writer, who has identified any of the techniques used by one or more antivirus, writes malware that cannot be detected by the target antivirus but only by the malware writer. As a typical example, a viral database which contains t signatures is equivalent to t testings (see previous section) and any new malware corresponds to the testing T_{t+1}.

Definition 4. *(Weak Testing Simulability) Let P be a property and T a testing whose role is to decide whether P is valid for a given population $\mathcal{P}$ or not. To weakly simulate this testing means introducing in $\mathcal{P}$ a new property P' which partially modifies the property P, in such a way that T systematically decides that P holds on $\mathcal{P}$, up to the error risks.*

Weak simulability differs from strong simulability since the attacker considers the same testings as the tester does. The attacker thus introduces a bias that the tester is not be able to detect.

The property P' of Definition 4 is generally opposite to the property P. It precisely represents a flaw that the attacker aims at exploiting. Bringing weak simulability into play is somehow tricky. It requires to get a deep knowledge of the testings to be simulated.

The central approach consists in introducing the property P' in such a way that the estimators E_i in use remain in the acceptance region of the testing (generally that of the null hypothesis). Let us recall that during the decision step, the tester checks whether $E < S$ or not. Thus weak simulability consists in changing the value $S - E$ while keeping it positive. For that purpose, we use the intrinsic properties of the relevant sampling distribution.

3.1.3.2 Reversing calculability and complexity theory

There are two key issues as far as computer science is concerned:

- *Calculability*. Here the central concept is Turing machines. The aim is to decide whether there exists a Turing machine (e.g. a program) which can compute a given problem or not. Some problems are not computable (the corresponding Turing machine never stops). Consequently the problem has no solution! So calculability theory aims at determining which problems are computable and which are not (undecidable problems).
- *Complexity*. When dealing with computable programs, another issue arises: how efficiently a problem can be computed. Here the central tool is the number of operations to solve a problem. From that number, problems are split into complexity classes. To describe things simply Papadimitriou (1993), the *Polynomial class* (P) corresponds to problems that are "computationally easy"Âăto solve, *Non deterministic polynomial class* (NP) to "computationally hard" to solve problems while *NP-complete class* contains the hardest problems from the NP class ("computationally very hard" problems). In practice, only the P class is computable (from seconds to a few hours however!).

So, as exposed in Filiol (2008c), a clever attacker will consider problems that are either impossible to solve (undecidable problems) or computationally hard to solve in order to design his attack or his malware. In other words, he opposes these problems to the defender which must solve them (whenever possible).

In Filiol (2007b); Zbitskiy (2009) the formalization based on formal grammars is considered. Code mutation (polymorphism, metamorphism) is formally described and the authors demonstrate that the choice of formal grammar class (according to the Chomsky classification) can yield powerful attacks. This work has been later extended by Gueguen (2011) by using van Wijngaarden grammars.

Another field of mathematics provides a lot of complex problems: combinatorics and discrete mathematics. Powerful approaches also considers those parts of mathematics to design or model powerful attacks based on malicious mathematics. For instance, the concept of cover set from the graph theory has enabled to understand worms or botnets spreading mechanisms and hence to design powerful versions of those malicious codes Filiol et al. (2007).

The concept of k-ary malware Filiol (2007c) is directly inspired from combinatorics.

Definition 5. *A* k-ary malware *is a family of k files (some of them may be not executable) whose union constitues a computer malware and performs an offensive action that is equivalent to that of a true malware. Such a code is said* sequential *(*serial mode*) if the k constituent parts are acting strictly one after the another. It is said* parallel *if the k parts executes simultaneously (parallel mode).*

Detecting such codes becomes almost impossible due to the number of combinations that should be explored. It has been proved Filiol (2007c) that, provided that the program interactions are determistic, the detection of k-ary malware is NP-complete.

3.1.4 Miscellanous

Other techniques have been recently considered which cannot clearly be related to the previous approaches. They can be considered as exotic approaches and we have chosen to expose them in a detailed way. For instance, the attacker can exploit to his own benefit the difference that exists between theorerical reality and practical reality in mathematics. This different is a critical issue in computer science. In Section 4 we explain how processor-dependent malware can be designed.

Other approaches can also rely on malicious cryptanalysis techniques. These techniques have been initiated in Filiol (2006) with the concept of *Zero-knowledge-like proof of cryptanalysis*.

Definition 6. *(Zero-knowledge-like proof of cryptanalysis) Let be a cryptosystem S_K and a property $\mathcal{P}$ about the output sequence of length n produced by S denoted σ_K^n. No known method other than exhaustive search or random search can obtain property $\mathcal{P}$ for σ_K^n. Then, a* zero-knowledge-like proof of cryptanalysis *of S consists in exhibiting secret keys $K_1, K_2, \ldots, ...K_m$ such that the output sequences $(\sigma_{K_i}^n)_{1 \leq i \leq m}$ verify $\mathcal{P}$ and such that, checking it requires polynomial time complexity. Moreover, the property $\mathcal{P}$ does not give any information on the way it was obtained.*

It worth considering that the reader/verifier can bring up against the author/prover that some random keys has been taken, the keystream has been computed and afterwards been claimed that the keystreams properties have been desired. In other words, the author/prover tries to fool the verifier/reader by using exhaustive search to produce the properties that have been considered for the zero-knowledge-like proof protocol. Thus the relevant properties must be carefully chosen such that:

- the probability to obtain them by random search over the key space makes such a search untractable. On the contrary the verifier/reader would be able to himself exhibit secret keys producing keystream having the same properties by a simple exhaustive search;
- the known attacks cannot be applied to retrieve secret keys from a fixed keystream having the properties considered by the author/prover.
- to really convince the reader/verifier, a large number of secret keys must be produced by the author/prover, showing that "he was not lucky".

Since there do not exist any known method other than exhaustive search or random search to produce output sequences σ_K^n having property $\mathcal{P}$, and since the complexity of a successful search is too high in practice, anybody who is effectively able to exhibit a secret K producing such output sequences obviously has found some unknown weaknesses he used to obtain this result. The probability of realizing property $\mathcal{P}$ through an exhaustive search gives directly the upper bound complexity of the zero-knowledge-like proved cryptanalysis.

This technique has been used to design an efficient code armouring technique. We will detail it in Section 5.

3.2 From computer virology to cryptology: malware-based operationel cryptanalysis

While in the previous section we have exposed how cryptography and mathematics can be used for malicious purposes, the present section deals with the opposite view: how malware can help to solve difficult (in other words computationally hard) problems. As far as cryptology is concerned, it mainly relates to "how get secret quantities (e.g. cryptographic keys) in illegitimate way?"

In this respect, malware can help to solve this problem very efficiently in a technical field called *Applied cryptanalysis*. As an example, let us consider the case of the AES (Advanced Encryption Standard). Let us assume that we use a *Deep-crack*-like computer that can perform an exhaustive key search of 2^{56} keys per second (in real life, this computer does not exist; the best cryptanalysis allows exhaustive key search of 56-bit keys in roughly 3 hours). Then, any brute force attack on the different AES versions will require with such a machine:

- 1.5×10^{12} centuries for a 128-bit key,
- 2.76×10^{31} centuries for a 192-bit key,
- 5.1×10^{50} centuries for a 256 bit-key.

It is obvious that this approach which has been used for a long time, is no longer valid for modern systems. Moreover, the mathematical analysis of AES did not revealed exploitable weaknesses. Consequently, other techniques, called "*applied cryptanalysis*" must be considered. The purpose of these techniques is not to attack the system directly (via the algorithm) but rather to act at implementation or management levels. By way of illustration, it is as if you wanted to go into a room by making a hole in the bombproof door, when you need only walk through the paper-thin walls. One of these approaches consists in using computer viruses or other malware.

3.2.1 Circumventing IPSec-like network encryption

IPSec-based protocols are often presented by IT-experts as an efficient solution to prevent attacks against data exchange. More generally, the use of encryption to protect communication channels or to seclude sensitive networks is seen as the ultimate defence. Unfortunately, this confidence is illusory since such “armoured” protocols can be manipulated or corrupted by an attacker to leak information as soon as an access is managed with simple userâĂŹs permission. In Delaunay et al. (2008), the authors demonstrate how an attacker and/or a malware can subvert and bypass IPsec-like protocols (IPSec, WiFi, IP encryptors...) to make data evade from the accessed system illegitimately. By using a covert channel, they show how a malware can encode the information to be stolen, can insert it into the legitimate encrypted traffic and finally collect/decode the information on the attackerâĂŹs side. This attack is summarized in Figure 2 The malware uses efficient techniques from the error-correcting

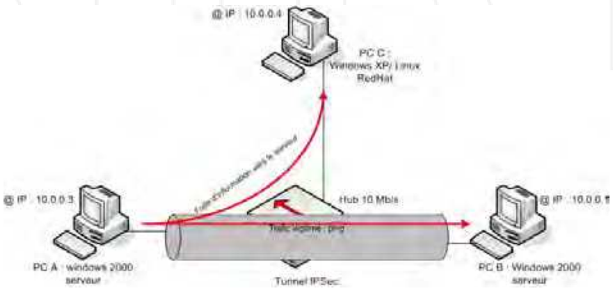

Fig. 2. General Scheme of IPSec-like protocol attacks by malware

theory to perform its attack. Moreover it adapts its behaviour to the encrypted traffic load in order to make its own activity totally invisible.

3.2.2 Dynamic cryptographic backdoors

A more efficient approach consists in exploiting implementation weaknesses in the environment (by means of a malware) or to change this latter as exposed in Filiol (2011). The first technique exploits the fact that many encryption algorithms rely on the operating system primitives to generate secret keys (e.g. Microsoft cryptographic API). The second solution is to modify the cryptographic algorithm on-the-fly in memory, and especially its mode of operation and/or its mathematical design. In fact the malware installs a dynamic trapdoor. The algorithm is not modified on the hard disk (no static forensics evidence). The trapdoor has a limited period of time and can be replayed more than once. In both cases, the encryption has been weakened in such a way that the attacker has just to intercept the ciphertext and perform the cryptanalysis. This technique has been efficiently applied to take control over the anonimity TOR network Filiol (2011b).

3.2.3 Anti-forensics cryptography

In the field of forensic analysis, encrypted data are particularly interesting when discovered on a hard disk. They are supposed to be a clear proof that a person has intended to protect data against analysis. When the forensics expert succeeds in one way or another to decrypt those encrypted data, the underlying plaintext clearly becomes very strong evidence for the judge. This means that in the process of digital proof, cryptography has a status of extremely high confidence.

In Filiol (2010a) it demonstrated how an attacker can use (malicious) cryptography in order to manipulate both the forensic analyst and the judge and thus fool them to incriminate an innocent people wrongly. The approach mainly lies in malicious cryptography techniques. The aim is to undermine a judgeâĂŹs blind faith in the value of cryptographic evidence and fool all those who rely heavily upon it. The paper shows with a fictional scenario how such an attack and manipulation can be performed. Hence it shows that the concept of (cryptographic) proof must be considered very cautiously and has no absolute value.

When coming back to the definition of a cryptosystem given in Section 2.2.1, the key point lies in the fact that any arbitrary cipher text C is defined relatively to a 3-tuple (P, K, E) only. In other words, you can do the following:

- **Technique 1**. Choose an arbitrary plaintext P, an arbitrary cipher text C and design accordingly a suitable pair (K, E) such that we obtain $C = E(K, P)$ and $P = E(K, C)$. This means that if we find random data on a hard disk and we do not have the algorithm E, we can assert almost anything. But worse, using malware we can replace any original pair (K, E) with a "malicious" one (K, E).
- **Technique 2**. Choose an arbitrary 3-tuple (E, C, P) and compute the key K such that we obtain $C = E(K, P)$. This approach has been partially addressed in Filiol (2006).
- **Technique 3**. Consider an arbitrary set of cipher texts $C_1, C_2 ... C_i ...$ and an arbitrary encryption/decryption algorithm E thus relating to (legitimate) plaintexts $P_1, P_2 ... P_i ...$ Thus we have for all i, for a key K i (possibly different for any i):

$$C_i = E(K, P_i) \text{ and } P_i = E(K, C_i)$$

  For any additional arbitrary triplet (P, C, K), it is then possible to modify the cryptosystem E into the system E so that we obtain

$$C = E(K, P) \text{ and } P = E(K, C)$$

  while we still have $C_i = E(K, P_i)$ and $P_i = E(K, C_i)$.

From a technical point of view, either the attacker modifies the system E on-the-fly or he simply replaces it with the modified one (e.g. with a virus). This technique is obviously the trickiest one but also the most convincing one.

Technically speaking, the difficulty depends on the kind of encryption system you intend to consider. While it is relatively easy for stream ciphers, the task becomes far more complex, yet not impossible, with block ciphers. Another aspect to take into consideration is the length (in bits) of the values C and P. The longer they are, the more difficult it will be to perform such rogue manipulation. However in the context of anti-forensics âĂŞ e.g. fooling an forensics expert âĂŞ those lengths can be relatively limited. The attacker will just have to build his attacking scenario carefully.

4. Processor-dependant malware

From the beginning of malware history (circa 1996), malware are:

- either operating system specific (Windows *.*, Unices, Mac, . . .);
- or application specific (e.g. macro viruses);
- or protocol dependent (e.g. *Conficker* versus *Slammer*).

At the present time, there are almost no hardware specific malware, even if some operating system are themselves hardware dependent (e.g. Nokia cell phones with *Symbian*). Recently, GPGPU malware Ioannidis (Polykronakis & Vasiliadis) have been proposed but they just exploit the fact that graphic cards are just distinct devices with almost the same features and capability as the system they are connected to. They do not really condition their action on new computing features. GPGPU malware strongly depend on the graphic card type (CUDA or OpenCL enabled).

We propose here to investigate the following critical issue: *is it possible to design malware – or more generally, any program – that operate beyond operating system and application types and varieties*? More precisely, we want:

- to operate beyond operating system and application types/varieties . . . ;
- while exploiting hardware specificities.

If such an approach would be possible, this would:

- enable far more precise and targeted attacks, at a finer level (surgical strikes) in a large network of heterogeneous machines but with generic malware;
- and represent a significant advantage in a context of cyberwarfare.

The recent case of the*StuxNet* worm shows that targeted attacks towards PLC components are nowadays a major concern in cyberattacks. However, while it can be very difficult to forecast and envisage which kind of applications is likely to be present on the target system (it can be a secret information), the variety in terms of hardware – and especially as far as processors are concerned – is far more reduced due to the very limited number of hardware manufacturers. We propose to consider *processor-dependent malware* and to rely on the onboard processor, which seems a good candidate to design hardware dependent software (a malware is nothing more than a software).

To design such *processor-dependent malware*, we need to identify the processor as precisely as possible. This is possible thanks to a different ways:

- by reversing existing binaries (but this provides a limited information since a given binary can indifferently execute on several processors like Intel x86s or AMDs chips),
- classical intelligence gathering...

There is a large spectrum of possibilities to collect this technical intelligence. But there is a bad news: deriving knowledge about processor internals is tricky and require a lot of work. Instead of analyzing processor logic gates architecture, we propose to work at the higher level: *to exploit mathematical perfection versus processor reality*.

4.1 Exploiting mathematical processor limitations

In order to use processors to discriminate programs' action and execution, we exploit the fact that first there is a huge difference between the mathematical reality and their implementation in computing systems and second that this difference is managed in various ways according to the processor brand, model and type.

4.1.1 Mathematical perfection versus processor reality

Let us begin with a very classical example: the algorithm given in Table 1. *what does this code (really) compute?*

```
Algorithm 1 :  The $\sqrt{\ }$ problem
   Input: — a real $A$;
   Output: — a boolean $B$
   Begin:
      $B = \sqrt{A} * \sqrt{A}$;
      Return[A==B];
   End.
```

Table 1. The Square-root problem

Let us suppose we choose $A = 2.0$ as input for this *Square-root* algorithm, we then have two possible answers, that are *opposite*:

1. *Mathematically*: **True** is returned;
2. *Practically*: **False** is returned!

Let us now explain why we have these different outputs. This comes from the fact that processors:

- have an increasing (architecture) complexity and size,
- have bugs, known and unknown (not published),
- use floating point arithmetic,
- use generally "secret" algorithms for usual arithmetic functions like $1/x$,$\sqrt{x}$, $1/\sqrt{x}$ ... that can be computed:
  1. at the *hardware* level;
  2. and/or at the *software* level.

As an example of a "secret algorithm", let us cite the famous Pentium Bugs *case* in 1994: Intel has never published neither the *correct* algorithm nor its bugged version used for the division but some researchers have tried reverse engineering techniques to understand which algorithm was programmed actually Coe et al. (1995).

Let us now consider the following problem: *can we define a set of (simple) tests to know on which processor we are?* As a practical example: *is it possible to know whether we are on a mobile phone or on a computer?*

The Intel Assembly Language instruction **CPUID** can be used both on Intel and AMD processors, but it has at least two severe drawbacks:

- it is easy to "find" it whenever scanning the file (malware detection issue);
- some other processors cannot recognize and process this instruction.

4.1.2 Processor bugs

Known or unknown bugs are good candidates to design such a set of tests and hence to discriminate processors:

- as an instance of such bug/test, it is easy determine whether we use a 1994 bugged Pentium or not: just use the numerical value that makes appear the *Pentium Division Bug*;
- but a lot of bugs will *freeze* the computer only (this can be used for processor-dependent denial of service [DoS] however);
- and it is not so simple to find a list of bugs, even if there are supposed to be "known".

The most interesting bugs to consider involve a floating point arithmetic operator. However it is worth keeping in mind that the knowledge of some other bugs (by the manufacturer, a Nation State...) can be efficiently used to target processors specifically and hence it represents a critical knowledge not to say a strategic one. Worse, hiding such bugs or managing floating arithmetics in a very specific way is more than interesting.

More generally let us consider some differences that exist event within the same type of processors but produced in two different versions: a national and an "export" version. As an example, we can consider the POPCOUNT function which compute the Hamming weight of an integer (the number of 1s in its binary form). Since it is a critical function in the context of cryptanalysis, the national version of a few processors have this function implemented in hardware while the export version just emulate it at the software level. Consequently a good way to discriminate national version from export version consists in computing Hamming weight a large number of times and then to record the computation time: it will be significantly higher for the export version which hence can be specifically targeted by a malware attack.

4.1.3 Using floating point arithmetics: the IEEE P754 standard

The IEEE P754 standard Overton (2011) has been approved as a norm by the IEEE ANSI in 1985. A lot of processors follow and comply to it but some processors do not (e.g. the CRAY 1 or the DEC VAX 780). Moreover, not all microcontrollers follow this standard either.

This norm does not impose algorithms to compute usual functions like $1/x$,$\sqrt{x}$, $1/\sqrt{x}$ or e^x. It just gives a specification for the four basic operations: addition, substraction, multiplication and division. So, for all other functions, there is very likely to exist differences as far as their implementation as algorithms are concerned. But we have to find them!

For 32-bit, environments, we have (see Table 2):

- 1 bit for the sign;
- 23 bits for the mantissa;

sign(x)	mantissa(x)	exponent(x)
1 bit	23 bits	8 bits

Table 2. Structure of 32-bit float "numbers" in the IEEE P754 Standard

- 8 bits for the exponent (integer).

The floating point arithmetic has a lot of curiosities, let us see some of them. One can find in Kulisch & Miranker (1983) the following questions due to Rump:

- Evaluate the expression

  $$F(X, Y) = \frac{(1682XY^4 + 3X^3 + 29XY^2 - 2X^5 + 832)}{107751}$$

  with $X = 192119201$ and $Y = 35675640$. The "exact" result is 1783 but numerically we can have a very different value like $-7.18056\,10^{20}$ (on a 32-bit IEEE P754 compliant processor).
- Evaluate the expression

  $$P(X) = 8118X^4 - 11482X^3 + X^2 + 5741X - 2030$$

  with $X = 1/\sqrt{2}$ and $X = 0.707$. The "exact" result is 0 but numerically we can have a very different value like $-2.74822\,10^{-8}$ (on a 32-bit IEEE P754 compliant processor).

Let us recall that the numerical value of an algebraic expression depends (generally) on the compiler because a non basic numerical expression result depends strongly on the order of the intermediate computations.

If we want to know on which processor we are working, we need to find, before anything else, some critical information:

1. first the *base* value used to represent numbers;
2. second the *word length*, *i.e.* the number of bits the processor is used to work (with floating point numbers for example).

For the base value, it is easy to conjecture that the base is 2, at least for modern processors. As far as the the word length is concerned, we have not found any numerical algorithm that is able to answer this question but we have found something very close. The algorithm given in Table 3 called the *Gentleman Code* Gentleman & Marovitch (1974) is surprisingly very interesting for both problems. First we can again ask: *what does this code (really) compute?* Well, again, we have two possible answers:

1. *Mathematically*: the two loops are theoretically *infinite loops* so they are looping forever;
2. *Practically* (see Erra & Grenier (2009)):
   - $\log_2(A)$ gives the number of bits used by the mantissa of floating point numbers;
   - B is the base used by the floating point arithmetic of the environment (generally it is equal to 2).

Both values are of course *processor-dependent* constants. So, with a small program, which has a polynomial time complexity, we can compute the number of bits used to represent the *mantissa* of any floating point number and so, we can deduce the word length.

```
Algorithm 2 :  The Gentleman Code
  Input: — A=1.0 ; B=1.0;
  Output: — $A, B$
  Begin:
    A=1.0;
    B=1.0;
    While ((A+1.0)-A)-1.0==0 ;
      A=2*A;
    While ((A+B)-A)-B==0 ;
      B=B+1.0;
    Return[A,B];
  End.
```

Table 3. The Gentleman code

4.2 Implementation and experimental results

Let us present a few tests that enables to discriminate processors operationally (for more details refer to Erra & Grenier (2009)). Table 4 summarizes a first set of tests. So these

Processor	Tests			
	1.2-0.8 == 0.4	0.1+0.1 == 0.2	0.1+0.1+0.1 == 0.3	0.1+…0.1 == 1.0
VAX 750	Yes	Yes	No	No
AMD 32	No	Yes	No	No
AMD 64	No	Yes	No	No
ATOM	No	Yes	No	No
INTEL DC	No	Yes	No	No
MIPS 12000	No	Yes	No	No
dsPIC33FJ21	No	Yes	Yes	No
IPHONE 3G	No	Yes	No	No

Table 4. A few easy computations

tests are interesting but not completely useful, this shows that we can simply know whether the processor follows the IEEE P754 arithmetic norm or not. For these simple expression, all processors that are IEEE P754 compliant will give the same answers..

With the following constant definitions in our test program in C, we obtain the results given in Tables 5 and 6.

- #define Pi1 3.141592653
- #define Pi2 3.141592653589

- #define Pi3 3.141592653589793
- #define Pi4 3.1415926535897932385

These results are more interesting, especially those in the third column (the numerical computation of $\sin(10^{37}\pi_1)$) in Table 5: a simple computation gives four subclasses of the set of processors (emphasized by a double horizontal lign between the subclasses). To

Processor	$\sin(10^{10}\pi_1)$	$\sin(10^{17}\pi_1)$	$\sin(10^{37}\pi_1)$	$\sin(10^{17}\pi_1) == \sin(10^{17}\pi_2)$
IPHONE 3G	0.375...	0.423...	-0.837...	No
AMD 32	0.375...	0.424...	-0.837...	No
AMD 64	0.375..	0.424..	0.837...	No
ATOM	0.375..	0.423..	-0.832..	No
INTEL DC	0.375...	0.423...	-0.832...	No
MIPS 12000	0.375...	0.423...	-0.832...	No
dsPIC33	*0.81...*	*0.62...*	*-0.44...*	*Yes*

Table 5. Computation of $\sin(10^{10}\pi)$ for various numerical values of the constant π

Processor	$\sin(10^{37}\pi_1)$	$\sin(10^{37}\pi_2)$	$\sin(10^{37}\pi_3)$	$\sin(10^{37}\pi_4)$
IPHONE 3G	47257756	9d94ef4d	99f9067	99f9067
AMD 64	af545000	af545000	af545000	af545000
ATOM	47257756	9d94ef4d	99f9067	99f9067
INTEL DC	47257756	9d94ef4d	99f9067	99f9067
MIPS 12000	47257756	9d94ef4d	99f9067	99f9067
dsPIC33	bee5	bee5	bee5	bee5

Table 6. $\sin(10^{37}\pi)$ in hex for various numerical values of the constant π

conclude with this part, it is important to stress on the *Influence* of the Compiler. To illustrate this, let us give a last example. We want to compute the generalized sum

$$s(N) := \sum_{i=1}^{N} 10^N \tag{1}$$

The "exact" value is of course $N * 10^N$, but let us have a look at the Table 7 to see some values we can have when computing $s(N) - N * 10^N$. However we have to point out that the results of the Table 7) heavily depend of course of the processor but *also* of the compiler used, of the options used and so on ....

N	10	21	22	25	30	100
$s - N * 10^N$	0.0	0.0	$-8.05\,10^8$	$-6.71\,10^7$	$-4.50\,10^{15}$	$4.97\,10^{86}$

Table 7. Computation of $s(N) - \sum_{i=1}^{N} 10^N$ for different values of N

4.3 Open problems

More work has to be done to understand these aspects more deeply. Nonetheless, we have here new insights on how design more specific attacks when considering the processor type AND the compiler version/type as the same time.

Floating Point Arithmetic (FPA) looks promising to define a set of tests enabling to identify the processor or, more precisely, a subset of possible processors. In the context of the malicious use of mathematics, a lot of open problems are arising. Let us mention a few of them.

The first open problem relates to the following question: *can we find an numerical algorithm, with a linear complexity in time and space which computes a floating point expression to distinguish a given processor more precisely?* Beyond the examples presented here, a promising algorithm could be based on a variant of the famous *logistic equation*, thoroughly studied in the chaos theory, which is defined by:

$$x_{n+1} = r\, x_n\, (1 - x_n) \tag{2}$$

with $r \in [0, 4]$.

The sequence defined by Equation 2, for a chosen and fixed x_0, can exhibit very different behaviors:

- a *periodic* behavior for example for values of r less than 3.0;
- or a *chaotic* behavior for values of r slightly larger than 3.57.

Another open problem goes deeper into the misuse of mathematics by attackers: *find processor-dependent hash functions*. Generally, hash functions are defined as independent from the processor. But, in some cases (export control of the cryptography, countermeasures...), one can desire to get rid of this view. The aim then consists in taking the *opposite idea*. We consider a hash function that heavily depends of the processor used to compute it. For example, it can be interesting to design a specific hash function for a *smartphone* or a specific processor. The best way to design such a hash function seems to use the properties of the floating point arithmetic operators of the processor; more specifically some of the arithmetic functions implemented on the processor. Consequently, this second open problem evolves slightly towards the following question: *can we define, for a specific processor, hash functions that use the floating point arithmetic of the concerned processor that respect the classical requirements for such functions?*

5. Evading detection and analysis

Fighting against computer malware require a mandatory step of reverse engineering. As soon as the code has been disassemblied/decompiled (including a dynamic analysis step), there is a hope to understand what the malware actually does and to implement a detection algorithm. This also applies to protection of software whenever one wishes to analyze them.

In this section, we show how the techniques of malicious cryptography enable to implement total amoring of programs, thus prohibiting any reverse engineering operation. The main interest of that approach lies in the fact that TRANSEC properties are achieved at the same time. In other words, the protected binaries have the same entropy as any legitimate, unprotected code. This same technique can also achieve a certain level of polymorphism/metamorphism at the same time. For instance, a suitable 59-bit key stream cipher is sufficient to generate up to 2^{140} variants very simply. More interestingly, the old fashioned concept of decryptor which usually constitutes a potential signature and hence a weakness, is totally revisited.

To illustrate this approach, let us consider the case study presented in Filiol (2010b) (among many other similar approaches) in which only a very few instructions are protected against any disassembly attempt. Let us consider a piece of x86 assembly instructions to protect from analysis. These instructions are translated into an intermediate representation (IR) derived from the REIL language before a final translation into bytecode.

To evade analysis and detection, we intend to protect this final bytecode by using a malicious PRNG, e.g. the last line in the following extract of code:

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

Let us now explore the different possible malicious PRNG we can use depending on the various operational conditions.

5.1 Malicious PRNG & protection scenarii

Sophisticated polymorphic/metamorphic or obfuscation techniques must rely on PRNG (Pseudo-Random Number Generator). In our context, the 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 (the malicious PRNG) which will afterwards output those data. The idea is that any data produced by the resulting generator will be first used by the code as a valid address, and then will itself seed the PNRG to produce the next random data.

Three cases are to be considered:

1. the code is built from any arbitrary random sequence;
2. the sequence is given by a (non yet protected) instance of bytecode and we have to design an instance of PNRG accordingly;
3. a more interesting problem lies in producing random data that can be somehow interpreted by a PRNG as meaningful instructions like `jump 0x89` directly.

This relates to interesting problems of PRNG cryptanalysis. We are going to address these three cases.

From a general point of view it is necessary to recall that for both three cases the malware author needs reproducible random sequences. By reproducible (hence the term of pseudo-random), we mean that the malware will replay this sequence to operate its course of execution. The reproducibility condition implies to consider a *deterministic Finite-State Machine* (dFSM). The general scheme of how this dFSM is working is illustrated as follows. Without the dFSM, any instruction data whenever executed produced a data used by the next instruction and so on (e.g. an address, an operand...).

$$I_0 \rightarrow D_0 \rightarrow I_1 \rightarrow D_1 \ldots \rightarrow D_i \rightarrow I_(i+1) \rightarrow \ldots$$

The problem lies in the fact that any analysis of the code easily reveals to the malware analyst all the malware internals since all instructions are hardcoded and unprotected. But if a few data/instructions are kept under an encrypted form, and are deciphered at execution only, the analysis is likely to be far more difficult (up to decryptor and the secret key protection issue). It is denied of *a priori* analysis capabilities. So to summarize, so we intend to have

$$I_0 \rightarrow D'_0 \rightarrow I_1 \rightarrow D_1 \ldots \rightarrow D'_i \rightarrow I_(i+1) \rightarrow \ldots$$

where $dFSM(D'_i) = D_i$ for all i. Upon execution, we just have to input data D'_i into the dFSM which will then output the data D_i.

A few critical points are worth stressing on

1. no key is neither required nor used;
2. instructions can similarly be protected as well.

Of course to be useful as a prevention tool against (static and dynamic) analysis, the dFSM must itself be obfuscated and protected against analysis. But this last point is supposed to be fulfilled Filiol (2010b).

5.2 (Malware) code built from an arbitrary sequence

In this case, the sequence is arbitrary chosen before the design of the code and hence the code is written directly from this arbitrary sequence. This case is the most simple to manage. We just have to choose carefully the dFSM we need. One of the best choice is to take a congruential generator since it implies a very reduced algorithm with simple instructions.

Let us consider X_0 an initial value and the corresponding equation

$$x_(i+1) = a * X_i + b \qquad mod(N)$$

where a is the multiplier, b is the increment and N is the modulus. Since the length of the sequence involved in the malware design is rather very short (up to a few tens of bytes), the choice of those parameters is not as critical as it would be for practical cryptographic applications. In this respect, one can refer to Knuth's reference book to get the best sets of parameters Knuth (1998).

Here are a few such examples among many others:

Standard minimal generator $a = 16,807 - b = 0 - N = 2^{31} - 1$.

VAX-Marsaglia generator $a = 16,645 - b = 0 - N = 2^{32}$.

Lavaux & Jenssens generator $a = 31,167,285 - b = 0 - N = 2^{48}$.

Haynes generator $a = 6,364,136,223,846,793,005 - b = 0 - N = 2^{64}$.

Kuth's generator $a = 22\,695\,477 - b = 1 - N = 2^{32}$ and $X_{n+1} >>= 16$.

Of course the choice of the modulus is directly depending on the data type used in the malware.

Another interesting approach is to consider hash functions and S/key. The principle is almost the same. We take a (m, n) hash function H which produces a n-bit output from a m-bit input with $m > n$. In our case we can build m in the following way

```
 m = <data to protect><padding of random data><size of data>
```

or equivalently

```
 m = D_i <random data> |D_i|
```

Then we choose a m-bit initialization vector (IV) and we compute the random sequence as follows

$$IV \rightarrow D_i = H(IV) \rightarrow x = H^{|D_i|}(D_i) \rightarrow y = H^{|x|}(x) \rightarrow H^{|y|}(y) \rightarrow$$

The iteration value $|D_i|$ can be used to get one or more required arbitrary value thus anticipating the next case. Of course the nature of the hash function is also a key parameter: you can either use existing hash function (e.g MD5, SHA-1, RIPEMD 160, SHA-2...) and keep only a subset of the output bit; or you can design your own hash function as explained in Knuth (1998).

5.3 Random sequence coming from an arbitrary (malware) code

In this slightly different case, the sequence is determined by a (non yet protected) instance of a code. This issue is then to design or use an instance of PRNG accordingly. This is of course a far more difficult issue which implies cryptanalytic techniques. To formalize the problem we have a sequence

$$X_0, X_1, X_2 \ldots x_i \ldots X_n$$

which represents critical data (addresses, ASM instructions, operands...) of a particular instance of a (malware) code. As for example let us consider three series of 32-bit integers describing bytecode values:

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

They are just different instances of the same instruction Filiol (2010b). The aim is to have these data in the code under a non hardcoded but an obfuscated form, e.g.

$$K_0, K_1, K_2, \ldots K_i, \ldots K_n \ldots$$

We then have to find a dFSM such that

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

The notation K_i directly suggests that the quantity input to the dFSM is a key in a cryptographic context but these keys have to exhibit local low entropy profile at the same time. So the malicious PRNG must take this into account as well. In this case, we have to face a two-fold cryptanalytic issue:

- either fix the output value X_i and find out the key K_i which outputs X_i for an arbitrary dFSM,
- or for an arbitrary set of pairs (X_i, K_i) design a unique suitable dFSM for those pairs.

The first case directly relates to a cryptanalytic problem while the second refers more to the problem of designing cryptographic dFSMs with trapdoors. In our context of malicious cryptography, the trapdoors here are precisely the arbitrary pairs of values (X_i, K_i) while the dFSM behaves for any other pair as a strong cryptosystem Filiol (2010a). This second issue is far more complex to address and still is an open problem.

Let us focus on the first case which has been partially addressed for real-life cryptosystem like *Bluetooth* E0 Filiol (2007) in the context of zero knowledge-like proof of cryptanalysis. But in the present case we do not need to consider such systems and much simpler dFSM can be built conveniently for our purposes: sequences of data we use are rather short.

To fullfil all operational constraints Filiol (2010b) those dFSM have to exhibit additional features in order to

- be used for code mutation purposes,
- exhibit TRANSEC properties. In other words, if we have $Y = dFSM(X)$, then X and Y must have the same entropy profile. Replacing X with a Y having a higher entropy profile would focus the analyst's attention (or trigger security software alert by considering local entropy tests).

In Filiol (2010b) a 59-key bit stream cipher has been considered. This encryption system is a combination generator which ismade of three linear feedback shift register and a combining Boolean function. The value K_i initializes the content of registers R_1, R_2 and R_3 at time instant $t = 0$, and the stream cipher (our dFSM) outputs bits s^t which represent the binary version of values X_i.

To describe the general principle of this technique, we will use this dFSM in a procedure whose prototype is given by

```
void sco(unsigned long long int * X, unsigned long long int K)
 {
  /* K obfuscated value (input), X unobfuscated value (output) */
  /* (array of 8 unsigned char) by SCO                         */
  ...
 }
```

Now according to the level of obfuscation we need, different ways exist to protect critical data inside a code (series of integers (1), (2) and (3) above). We are going to detail two of them.

5.3.1 Concatenated bytecodes

The dFSM outputs critical data under a concatenated form to produce chunks of code corresponding to the exact entropy of the input value (K_i). This enables to prevent any local increase of the code entropy. For the dFSM considered, it means that we output series (1), (2) and (3) under the following form

```
1)--> 0x2F010000000400040000000300000000000000089
2)--> 0x3D010000000400040000000300000000000000050
3)--> 0x05010000000400040000000300000000000000008D
```

Let us detail the first output sequence (1). It will be encoded as three 59-bit outputs M_1, M_2 and M_3

```
 M_1 =     0x0BC04000000LL;
 M_2 = 0x080008000000060LL;
 M_3 = 0x000000000000089LL;
```

To transform M_1, M_2 and M_3 back into five 32-bit values X_1, X_2, X_3, X_4 and X_5, we use the following piece of code:

```
/* 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 */
```

Values M_1, M_2 and M_3 will be stored in the code as the values K_1, K_2 and K_3 with $dFSM(K_i) = M_i$:

```
 K_1 = 0x6AA006000000099LL;
 K_2 = 0x500403000015DC8LL;
 K_3 = 0x0E045100001EB8ALL;
```

Similarly we have for sequence (2)

```
M_1 =     0x0F404000000LL;  K_1 = 0x7514360000053C0LL;
M_2 = 0x080008000000060LL;  K_2 = 0x4C07A200000A414LL;
M_3 = 0x000000000000050LL;  K_3 = 0x60409500001884ALL;
```

and for sequence (3)

```
M_1 =     0x01404000000LL;  K_1 = 0x76050E00001F0B1LL;
M_2 = 0x080008000000060LL;  K_2 = 0x00000010C80C460LL;
M_3 = 0x00000000000008DLL;  K_3 = 0x000000075098031LL;
```

The main interest of that method is that the interpretation of code is not straightforward. Code/data alignment does not follow any logic (that is precisely why a 59-bit dFSM has been considered compared to a seemingly more obvious 64-bit dFSM ; any prime value is optimal).

Moreover, as we can notice, the K_i values are themselves sparse as unobfuscated opcodes are (structural aspect). Additionally, their entropy profile (quantitative aspect) is very similar to the M_i values (and hence the X_i ones). This implies that any detection techniques based on local entropy picks is bound to fail.

Due to the careful design of the 59-bit dFSM, the unicity distance obtained is greater than 59 bits (the unicity distance is the minimal size for a dFSM output to be produced by a single secret key). In the present case, a large number of different 59-bit keys can output an arbitrary output sequence. Here are the results for the three series (1), (2) and (3) (Table 8): This implies

Serie	M_i values	Number of K_i	M_i values	Number of K_i	M_i values	Number of K_i
(1)	M_1	314	M_2	2,755	M_3	8,177
(2)	M_1	319	M_2	2,755	M_3	26,511
(3)	M_1	9,863	M_2	2,755	M_3	3,009

Table 8. Number of possible keys for a given output value Filiol (2010b)

that the 9 M_i values can be randomly selected and thus we have

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

different possible code variants. It is approximatively equal to 2^{103} variants. Details of implementation are described in Filiol (2010b).

5.3.2 Non concatenated bytecodes

In this second case, the dFSM outputs 59-bit chunks of data whose only the 32 least significant bits are useful. Then here five 59-bit chunks of data M_1, M_2, M_3, M_4 and M_5 are output. For sequence (1) we have

```
M_1 = 0x*******2F010000LL;
M_2 = 0x*******00040004LL;
```

```
    M_3 = 0x********00000003LL;
    M_4 = 0x********00000000LL;
    M_5 = 0x********00000089LL;
```

where the symbol $\star$ describes any random nibble.

The main interest of that method lies in the fact that it naturally and very simply provides increased polymorphism properties compared to the previous approach. Indeed about 2^{140} 5-tuples $(K_1, K_2, K_3, K_4, K_5)$ whenever they are input in the dFSM, produces 5-tuples $(X_1, X_2, X_3, X_4, X_5)$. Then a huge number of different instances of the same code can be produced by randomly choosing any possible 5-tuples. By increasing size of the memory of the FSM we even can arbitrarily increase the number of possible polymorphic instances.

6. Conclusion

The rise of malicious mathematics and malicious cryptography results in a number of critical issues.

For the first time, the advantage goes definitively to the attacker as soon as he uses malicious mathematics and malicious cryptology techniques. He has just to use most of the results of calculability theory and complexity theory for his own benefit. This shed a new light on the importance to answer to the famous question: *"does P = NP or not?"* But a positive solution would solve only the problem partly. Let us recall that modern cryptology for the first time in Mankind History was giving the advantage to the defender. Unfortunately this period of peace was short. The threat has just evolved, adapted and changed. We now have to face a very unsecure world again.

From that it follows that science can no be neutral and the question of science's dual use – and it corollary the control of knowledge – is more than ever into question.

Aside those society issues, malicious mathematics and malicious cryptology propose a new, stimulating, exciting research universe from a purely academic point of view. A huge number of open problems, both theoretical and practical have been identified. More are likely to come. in this respect, we can hope that meny good things can also arise from the research that will consider those problems.

7. References

Coe, T., Mathissen, T., Moler, C. & Pratt, V. (1995). Computational Aspects of the Pentium Affair. *IEEE Computational Science & Engineering*, 2(1): 18–30.

Delaunay, G., Filiol, E. & Jennequin, F. (2008). Malware-based Information Leakage over IPSEC Tunnels. *Journal of Information Warfare*, 7(3):11–22.

Erra, R. & Grenier, C. (2009). How to choose RSA Keys (The Art of RSA: Past, Present and Future)? *iAWACS 2009*, Laval, France. [online] `http://www.esiea-recherche.eu/Slides09/slides_iAWACS09_Erra-Grenier_How-to-compute-RSA-keys.pdf`

Filiol, E. (2005). *Computer Viruses: from Theory to Applications*, IRIS International Series, Springer Verlag France, ISBN-10: 2287239391.

Filiol. E. (2005). Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis: the BRADLEY virus. *Proceedings of the 14th EICAR Conference*, ESAT Publishing, pp. 201–217.

Filiol, E (2006). Zero Knowledge-like proof of cryptanalysis of Bluetooth Encryption. *International Journal in Information Technology*, 3(4): 285–293.

Filiol, E., Franc, E., Gubbioli, A., Moquet, B. & Roblot, G. (2007). Combinatorial Optimisation of Worm Propagation on an Unknown Network. *International Journal in Computer Science*, 2(2): 124–130.

Filiol, E. & Josse, S. (2007). Statistical model for viral undecidability, *Journal of Computer Virology*, 3(2): 65–74, `http://www.springer.com/computer/journal/11416`

Filiol, E. (2007). Zero knowledge-like Proof of Cryptanalysis of Bluetooth Encryption. *International Journal in Information Theory*, 3(4): 40–51. [online] `http://www.waset.org/journals/ijit/v3/v3-4-40.pdf`

Filiol, E. (2007). *Techniques virales avancées*, IRIS Series, Springer Verlag France, ISBN 978-2-287-33887-8 (An English translation is due at the end of 2012).

Filiol, E. (2007). Metamorphism, Formal Grammars and Undecidable Code Mutation. *International Journal in Computer Science*, 2(1): 70–75.

Filiol, E. (2007). Formalisation and Implementation Aspects of *K*-ary (malicious) Codes. *Journal in Computer Virology*, 3(2): 75–86.

Filiol, E. & Raynal, F. (2008). Malicious Cryptography...reloaded, *CanSecWest Conference*, Vancouver, Canada. [online] `http://cansecwest.com/csw08/csw08-raynal.pdf`

Filiol, E & Raynal, F. (2008) Enciphered communications: a can of worms? *Revue de Défense Nationale*, Special Issue "From Crybercrime to Cyberwarfare", 5:86–97.

Filiol, E. (2008). Malware of the Future: When Mathematics Work for the Dark Side. *Hack.lu 2008*, Keynote Talk, Luxembourg, October 22nd. [Online] `http://hack.lu/archives`.

Filiol, E. (2010). Anti-forensics Techniques Based on Malicious Cryptography. *Proceedings of the 9th European Conference in Information Warfare ECIW 2010*, Thessaloniki, Greece, pp. 63–70, Academic Conferences International Press.

Filiol, E. (2010). Malicious Cryptography Techniques for Unreversable (malicious or not) binaries". *H2HC 2010 Conference* Sao Paulo, Brazil [online] `http://arxiv.org/abs/1009.4000`

Filiol, E (2011). Dynamic cryptographic backdoors, *CanSecWest Conference*, Vancouver, Canada.

Filiol, E (2011). Dynamic cryptographic backdoors - How to Take Control over the TOR Network, *H2HC Conference*, Sao Paulo, Brazil, October 2011.

Gentleman, W. & Marovitch, S. (1974). More on algorithms that reveal properties of floating point arithmetic units. *Communications of the ACM*, 17(5): 276–277.

Gueguen, G. (2011). Van Wijngaarden Grammars and Metamorphism. *6th International Workshop on Frontiers in Availability, Reliability and Security 2011 (ARES/FARES'11)*, Vienna, Austria.

Ioannidis, S., Polykronakis, M. & Vasiliadis, G. (2010). GPU-assisted Malware. *Malware 2010*. [online] `http://dcs.ics.forth.gr/Activities/papers/gpumalware.malware10.pdf`

Knuth, D. E., (1998). *The Art of Computer Programming: Seminumerical Algorithms*, Volume 2, Addison-Wesley.

Kraus, J. (1980). *Selbst Reproduzierende Programme*. Master Thesis in Computer Science, University of Dortmund. Translated from the German and edited by D. Bilar & E. Filiol, *Journal in Computer Virology*, 5(1): 9–87.

Kulisch, U. W. & Miranker, W. L. (1983). Arithmetic of computers, *Siam J. of computing*, 76:54–55.

Overton, M.-L. (2001). *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM Publishing, ISBN-10: 0-89871-571-7.

Papadimitriou, C. H. (1993). *Computational Complexity*. Addison Wesley, ISBN-10 0201530821.

Perseus homepage. `http://code.google.com/p/libperseus/`

The Wassenaar Arrangement in Export Control for Conventional Arms and Dual-Use Goods and Technologies `http://www.wassenaar.org`

Young, A. & Yung, M. (2004). *Malicious Cryptography: Exposing Cryptovirology*, Wiley, ISBN 0-7645-4975-8.

Zbitskiy, P. V. (2009). Code Mutation Techniques by Means of Formal Grammars and Automata. *Journal in Computer Virology*, 5(3): 199–207.