

Processor-dependent Malwares

Anthony Desnos² & Robert Erra² & Eric Filiol¹
aka the FED Group

ESIEA - $(C + V)^O$
38 rue des Dr Calmette et Guérin, 53 000 Laval, France
filiol@esiea.fr

ESIEA - SI&S
9 rue Vésale, 75 005 Paris, France
{erra, desnos}@esiea.fr

iAWACS'09



Current section

- 1 Introduction
- 2 Theoretical Background
 - Starting from a formal model of malware
 - Exploring the Viral Classes
 - Practical Utility of the Formal Model
- 3 Exploiting Mathematical Processor Limitations
 - Mathematical perfection versus Processor Reality
 - The bugs
 - The standard IEEE p754
- 4 Implementation and Experimental Results
- 5 Conclusion and Future Work

Introduction

- From the beginning of malware history (circa 1996), malwares are
 - either operating system specific (Windows *.* , Unices, Mac, ...),
 - or application specific (e.g. macro viruses),
 - or protocol dependent (e.g. *Conficker* vs *Slammer*)
 - ...
- At the present time, quite no hardware specific malware.
- Even if some operating system are themselves hardware dependent (e.g. *Symbian* malware).

Introduction

- Critical issue: is it possible to design malware that go beyond operating system and application varieties but
 - go beyond operating system and application varieties. . .
 - while exploiting hardware specificities?
 - . . .
- If such an approach is possible, this would enable
 - far more precise attacks, at a finer level (surgical strikes) in a large network of heterogeneous machines but with generic malware,
 - in a context of cyberwarfare, this would represent a significant advantage.
- Good candidate: the onboard processor.

Idea: Malware-dependent Processor

- Identifying the processor is easily possible
 - either reverse existing binaries,
 - or analyze public market offers.
 - ...
- Large spectrum of possibilities to collect this technical intelligence.
- Bad news: deriving knowledge about processor internals is tricky and require a lot of work.
- Instead of analyzing processor logic gates architecture, work at the higher level.

⇒ Exploit Mathematical perfection versus Processor Reality



- 1 Introduction
- 2 Theoretical Background
 - Starting from a formal model of malware
 - Exploring the Viral Classes
 - Practical Utility of the Formal Model
- 3 Exploiting Mathematical Processor Limitations
 - Mathematical perfection versus Processor Reality
 - The bugs
 - The standard IEEE p754
- 4 Implementation and Experimental Results
- 5 Conclusion and Future Work

Current section

- 1 Introduction
- 2 **Theoretical Background**
 - Starting from a formal model of malware
 - Exploring the Viral Classes
 - Practical Utility of the Formal Model
- 3 Exploiting Mathematical Processor Limitations
 - Mathematical perfection versus Processor Reality
 - The bugs
 - The standard IEEE p754
- 4 Implementation and Experimental Results
- 5 Conclusion and Future Work

A Few Basic Notations (Sorry) [Fil05]

- We consider the formal model given by Zuo and Zhou (2003 & 2005).
- Sets \mathbb{N} and S are the set of natural integers and the set of all finite sequences of such integers, respectively.
- Let s_1, s_2, \dots, s_n be elements from S .
 - Let $\langle s_1, s_2, \dots, s_n \rangle$ describe an injective computable function from S^n to \mathbb{N} whose inverse function is computable as well.
 - If we consider a partial computable function $f : \mathbb{N} \rightarrow \mathbb{N}$, then $f(s_1, s_2, \dots, s_n)$ describes $f(\langle s_1, s_2, \dots, s_n \rangle)$ in an abridged way.
 - This notation extends to any n -tuple of integers i_1, i_2, \dots, i_n .

A Few Basic Notations (2) (Sorry Again)

- For a given sequence $p = (i_1, i_2, \dots, i_k, \dots, i_n) \in S$, we denote $p[j_k/i_k]$ the sequence p in which the term i_k has been replaced by j_k , let say $p[j_k/i_k] = (i_1, i_2, \dots, j_k, \dots, i_n)$.
- If the element i_k of sequence p is computed by a computable function v (equivalently compute $p[v(i_k)/i_k]$), let us adopt the equivalent abridged notation $p[v(\underline{i_k})]$ in which the underlined symbol describes the computed element.
- In the general case (compute more than one element at the same time in p), we note $p[v_1(\underline{i_{k_1}}), v_2(\underline{i_{k_2}}), \dots, v_l(\underline{i_{k_l}})]$.

A Few Basic Notations (3) (Don't Give up)

- Finally we describes by $\varphi_P(d, p)$ a function which is computed by a program P in the environment (d, p) .
 - d and p are denoting data in the environment (including clock, mass memories and equivalent structures or devices) and programs (including those of the operating system itself) respectively.
 - That environment corresponds in fact to the operating system which has been extended to the activity of one or more users.
- When considering the Gödel coding e for the program P , we use the notation $\varphi_e(d, p)$. Its definition domain is then denoted by W_e while his image space is denoted E_e .

Zuo & Zhou Formal Model

- Let us give the general formal definition of computer viruses (most complete case).

Definition

(Non Resident Virus) A total recursive function v is a non resident virus if for every program i , we have:

- $$\varphi_{v(i)}(d, p) = \begin{cases} D(d, p), & \text{if } T(d, p) \quad (i) \quad \text{(Added Fonctionnality)} \\ \varphi_i(d, p[v(\underline{S(p)})]) & \text{if } I(d, p) \quad (ii) \quad \text{(Infection)} \\ \varphi_i(d, p), & \text{otherwise} \quad (iii) \quad \text{(Imitation)} \end{cases}$$
- $T(d, p)$ and $I(d, p)$ are two recursive predicates such that there is no value $\langle d, p \rangle$ that satisfies them both at the same time. Moreover both functions $D(d, p)$ et $S(p)$ are recursive.
- The set $\{\langle d, p \rangle : \neg(T(d, p) \vee I(d, p))\}$ is infinite.

Exploring the Model

- The two predicates $T(d, p)$ and $I(d, p)$ represent the payload and the infection trigger conditions respectively.
- Whenever $T(d, p)$ is true, the virus executes the payload $D(d, p)$ while whenever $I(d, p)$ est true, the virus selects a target program by means of the selection function $S(p)$ and then infects it. Finally the original program i is executed (host program).
- Virus kernel: the set of functions $D(d, p)$ and $S(p)$ with predicates $T(d, p)$ and $I(d, p)$.
- The virus kernel describes the malware in a univoqual way.
- This model can be extended to other form of malware (more sophisticated viruses, Trojan. . .).

Polymorphic Viruses

Definition

The pair (v, v') of total recursive functions v and v' is called *Polymorphic virus with two forms* if for every program i we have

$$\varphi_{v(i)}(d, p) = \begin{cases} D(d, p), & \text{if } T(d, p) \\ \varphi_i(d, p[v'(S(p))]), & \text{if } I(d, p) \\ \varphi_i(d, p), & \text{otherwise} \end{cases}$$

and

$$\varphi_{v'(i)}(d, p) = \begin{cases} D(d, p), & \text{if } T(d, p) \\ \varphi_i(d, p[v(S(p))]), & \text{if } I(d, p) \\ \varphi_i(d, p), & \text{otherwise} \end{cases}$$

- Whenever predicate $I(d, p)$ is true the virus selects a target program by means of $S(p)$, infects it then transfers control back to the host program x . $S(p)$ is performing the code mutation as well.

Metamorphic Viruses

Definition

Let v and v' be two different total recursive functions. The pair (v, v') is called metamorphic virus if for every program i , then the pair (v, v') satisfies:

$$\varphi_{v(i)}(d, p) = \begin{cases} D(d, p), & \text{if } T(d, p) \\ \varphi_i(d, p[v'(S(p))]), & \text{if } I(d, p) \\ \varphi_i(d, p), & \text{otherwise} \end{cases}$$

et

$$\varphi_{v'(i)}(d, p) = \begin{cases} D'(d, p), & \text{if } T'(d, p) \\ \varphi_i(d, p[v(S(p))]), & \text{if } I'(d, p) \\ \varphi_i(d, p), & \text{otherwise} \end{cases}$$

where $T(d, p)$ – respectively $I(d, p)$, $D(d, p)$, $S(p)$ – is different from $T'(d, p)$ – respectively $I'(d, p)$, $D'(d, p)$, $S'(p)$.

- Metamorphic viruses are similar to polymorphic viruses except that selection functions $S(p)$ and $S'(p)$ are different. The kernel of metamorphic forms are totally different.



Stealth Viruses

Definition

The pair (v, sys) made of a total recursive function v and a system call sys (a recursive function as well) is a stealth virus with respect to the system call sys , if there exists a recursive function h such that for every program i we have:

$$\varphi_{v(i)}(d, p) = \begin{cases} D(d, p), & \text{if } T(d, p) \\ \varphi_i(d, p[v(\underline{S(p)}), h(\underline{\text{sys}})]) & \text{si } I(d, p) \\ \varphi_i(d, p), & \text{otherwise} \end{cases}$$

et

$$\varphi_{h(\text{sys})}(i) = \begin{cases} \varphi_{\text{sys}}(y), & \text{if } x = v(y) \\ \varphi_{\text{sys}}(i), & \text{otherwise} \end{cases}$$

- Note stealth is a relative concept (with respect to a given set of system calls).

What the Model Show Us

- We must identify and use a feature that will make a virus (in the general case, a malware) operating whether a given type of processor chip is present or not.
- In the previous formal definition, whatever may be the class of virus, the obvious candidates for usable features are predicates $T(d, p)$ and $I(d, p)$ (payload and infection trigger conditions respectively).
- In the optimal case, we are interested in considering two different features to control and manage payload triggering and infection control separately and independently.

What the Model Show Us (2)

- Code mutation and stealth can also be managed with respect to specific processors in the same way.
 - As an example a malware will enforce Hardware Virtual Machine-based rootkit techniques whenever present.
 - Code mutation (e.g metamorphism) will be activated only if a suitable processor instruction set is available.
- This approach, yet formal, gives a powerful insight of how design processor dependent malware.
- This enables to greatly reduce the problem of side effect that may betray the activity of a malware.

Current section

- 1 Introduction
- 2 Theoretical Background
 - Starting from a formal model of malware
 - Exploring the Viral Classes
 - Practical Utility of the Formal Model
- 3 Exploiting Mathematical Processor Limitations**
 - Mathematical perfection versus Processor Reality
 - The bugs
 - The standard IEEE p754
- 4 Implementation and Experimental Results
- 5 Conclusion and Future Work



Mathematical perfection versus Processor Reality

Algorithm 1 : The $\sqrt{}$ problem

Begin:

$A=2.0;$

$B = \sqrt{A} * \sqrt{A};$

Return[$B==2$];

End.

Well, we have two possible answers:

- 1 *Mathematically:* **True** is returned
- 2 *Practically:* **False** is returned!

Mathematical perfection versus Processor Reality

Algorithm 2 : The $\sqrt{}$ problem

Begin:

$A=2.0;$

$B = \sqrt{A} * \sqrt{A};$

Return[$B==2$];

End.

Well, we have two possible answers:

- ① *Mathematically:* **True** is returned
- ② *Practically:* **False** is returned!

Mathematical perfection versus Processor Reality

Processors:

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



Mathematical perfection versus Processor Reality

Problem: can we define a set of (simple) tests to know on which processor we are ?

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,

- it is easy to "find" it
- but for other processors that don't understand **CPUID** ?

Bugs, known and unknown are good candidates to make a set of tests:

- It is easy to design a test to know if we are on a 1994 bugged Pentium: just use the *Pentium Division Bug*
- But a lot of bugs will only *freeze* the computer
- and it is not so simple to find a list of all known bugs

So, we will not use bugs.

The standard IEEE p754 [Ove01]

- Approved by IEEE ANSI in 1985
- Some processors do not follow it (example: CRAY 1, DEC VAX 780)
- A lot of processors follow it, but not all (microcontrollers)
- The norm does not impose the algorithms to compute usual functions $1/x$, \sqrt{x} , $1/\sqrt{x}$ or e^x
- So, we will have some differences . . . with implementation of algorithms
- But, we have to find them!

Floating point numbers in the standard IEEE p754:

For 32 bits, we have:

- 1 bit for the sign;
- 23 bits for the mantissa;
- 8 bits for the exponent (integer).

$$fl(x) = \boxed{\text{sign}(x)} \boxed{\text{mantissa}(x)} \boxed{\text{exponent}(x)}$$

The Gentleman Code [Mul89]

Algorithm 3 : The Gentleman Code

Input: — $A=1.0$; $B=1.0$;

Output: — What does this code (really) compute ?

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.

The Gentleman Code

Well, again, we have two possible answers:

- ① *Mathematically*: the two loops are theoretically *infinite loops* so they are looping forever
- ② *Practically*:
 - $\log_2(A)$ 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 2).

Both values are *processor-dependent* constants.



The Gentleman Code

Well, again, we have two possible answers:

- ① *Mathematically*: the two loops are theoretically *infinite loops* so they are looping forever
- ② *Practically*:
 - $\log_2(A)$ 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 2).

Both values are *processor-dependent* constants.



The Gentleman Code

Well, again, we have two possible answers:

- ① *Mathematically*: the two loops are theoretically *infinite loops* so they are looping forever
- ② *Practically*:
 - $\log_2(A)$ 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 2).

Both values are *processor-dependent* constants.



The Gentleman Code

Well, again, we have two possible answers:

- ① *Mathematically*: the two loops are theoretically *infinite loops* so they are looping forever
- ② *Practically*:
 - $\log_2(A)$ 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 2).

Both values are *processor-dependent* constants.



The Gentleman Code

Well, again, we have two possible answers:

- ① *Mathematically*: the two loops are theoretically *infinite loops* so they are looping forever
- ② *Practically*:
 - $\log_2(A)$ 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 2).

Both values are *processor-dependent* constants.



The Gentleman Code

Well, again, we have two possible answers:

- ① *Mathematically*: the two loops are theoretically *infinite loops* so they are looping forever
- ② *Practically*:
 - $\log_2(A)$ 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 2).

Both values are *processor-dependent* constants.



Current section

- 1 Introduction
- 2 Theoretical Background
 - Starting from a formal model of malware
 - Exploring the Viral Classes
 - Practical Utility of the Formal Model
- 3 Exploiting Mathematical Processor Limitations
 - Mathematical perfection versus Processor Reality
 - The bugs
 - The standard IEEE p754
- 4 Implementation and Experimental Results**
- 5 Conclusion and Future Work

Some basic (too simple) tests

Processor	$1.2-0.8==0.4$	$0.1+0.1==0.2$	$0.1+0.1+0.1==0.3$	$0.1+\dots 0.1==1.0$	
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	

Some less basic tests

With

- `#define Pi1 3.141592653`
- `#define Pi2 3.141592653589`
- `#define Pi3 3.141592653589793`
- `#define Pi4 3.1415926535897932385`

Processor	$\sin(10^{10} \pi_1)$	$\sin(10^{17} \pi_1)$	$\sin(10^{37} \pi_1)$	$\sin(10^{17} \pi_1) == \sin(10^{17} \pi_2)$
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
IPHONE 3G	0.375...	0.423...	-0.837...	No

Some less basic tests

Processor	$\sin(10^{37}\pi_1)$	$\sin(10^{37}\pi_2)$	$\sin(10^{37}\pi_3)$	$\sin(10^{37}\pi_4)$	
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	
IPHONE 3G	47257756	9d94ef4d	99f9067	99f9067	

Some floating point curiosities (Rump, [DM97, KM83])

- Evaluation of

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

with $X = 192119201$ and $Y = 35675640$. Exact result is 1783 but numerically $-7.18056 \cdot 10^{20}$.

- Evaluation of

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

with $X = 1/\sqrt{2}$ and $X = 0.707$. Exact result is 0 but numerically $-2.74822 \cdot 10^{-8}$.

Don't forget the *influence* of the compiler

Let us give a last example, we want to compute

$$s = \sum_{i=1}^N 10^N$$

Exact value is $N * 10^N$. But one can have something like:

N	10	21	22	25	30	100
$s - N * 10^N$	0,0	0,0	$-8.05 \cdot 10^8$	$-6.71 \cdot 10^7$	$-4.50 \cdot 10^{15}$	$4.97 \cdot 10^{86}$

Current section

- 1 Introduction
- 2 Theoretical Background
 - Starting from a formal model of malware
 - Exploring the Viral Classes
 - Practical Utility of the Formal Model
- 3 Exploiting Mathematical Processor Limitations
 - Mathematical perfection versus Processor Reality
 - The bugs
 - The standard IEEE p754
- 4 Implementation and Experimental Results
- 5 Conclusion and Future Work

Conclusion and Future Work

- Floating Point Arithmetic (FPA) looks promising to define a set of tests to identify the processor or, more precisely, a subset of possible processors.
- We propose, *asap*, the Proc_Scope Tool: a software tool.
- Proc_Scope uses carefully chosen *numerical expressions* that give information on the processor.

More results to be published very soon in *Journal of Computer Virology*.





M. Daumas and J.-M. Muller.
Qualité des calculs sur ordinateur.
Masson, 1997.



E. Filiol.
Computer Viruses, from theory to applications, IRIS International Series.
Springer Verlag France, 2005.



U.W. Kulisch and W.L. Miranker.
Arithmetic of computers.
Siam J. of computing, 76:54–55, 1983.



J.-M. Muller.
Arithmétique des ordinateurs.
Masson, 1989.



M. L. Overton.



Numerical Computing with IEEE Floating Point Arithmetic. SIAM, 2001.

