A SELF-CHECKING SIGNATURE SCHEME FOR CHECKING BACKDOOR SECURITY ATTACKS IN INTERNET

M.F. Abdulla
Department of Computer Science
University of Yemen
Yemen
al_badwi@hotmail.com

C.P. Ravikumar
Texas Instruments India
Wind Tunnel Road
Murugeshpalya, Bangalore 560017
ravikumar@ti.com

Abstract

It is very common for people across the globe to collaborate on the Internet and intellectual property amongst each other. A serious threat to this form of collaboration can come from "backdoor" attacks from hackers, who can distort the information content. For example, a backdoor attack may replace common operating system functions with malicious ones. A possible precaution against such an attack is to generate a signature database and compare the signature of a system functionality with its golden signature before using the functionality. We present an alternate and novel method to detect Trojan activity. Called time fingerprinting, the method relies on observing a finite number of fingerprints during signature generation and tracing the Trojan fingerprints in system files. We have verified the desired properties using common semi trusted operating system files.

Keywords: Internet Security, Signature, Fingerprint, Trojan, Testing

1. Introduction

Trust is of vital importance in a collaboration. Internet provides a number of means to collaborate on projects, such as e-mail groups and users share files frequently through these means. A serious threat to this form of collaborative computing comes from Trojan attacks, where a seemingly harmless file contains malicious content [5]. For example, attackers are known to replace system-related files that rarely undergo change, such as the kernel image or system daemons. Trojans can result in annoying pop-up messages at the minimum and corruption or deletion of data at the other end of the spectrum. Trojan code in the "login" system functionality can intercept the user's password and forward it to a hacker. Object reconciliation, which compares an object with an earlier version of the same object, has been used as a solution. Checking for properties such as size, date or time of modification is not a foolproof technique since these can be manipulated by a hacker. In fact, a hacker can, through a trial-and-error procedure, insert malicious code and yet generate a binary executable file which matches the original file in these properties that are easy to test.

A checksum or another "signature" can be added to the file to authenticate the file. A database of the golden signatures is maintained on a separate server to minimize the possibility of modifying both the golden and the file signatures. After downloading a file, a user can check if the signature of the file matches the published signature and discard the file if the signatures do not match. Signature checking is also well suited to provide security to the *mobile objects* technology [1], software is divided into individual code blocks. Legal code blocks of application are installed on the client beforehand. When the client application starts, it is first connected to the designated server through the network and additional code blocks are downloaded. Java Applets [4] and ActiveX [2] are examples of mobile objects [4]. For security, each mobile object is associated with a signature. The client must authenticate itself to the server at the starting of the application by transmitting the signature, which, in turn, is compared against the golden signature by the server.

In this paper, we propose a security measure that is faster in detecting the infected illegal files and more efficient in terms of storage used. The signatures of the files are generated using a technique known as Concurrent Intermediate Signature Comparison (CIC). Signature comparisons are done at predefined time instants called legal time fingerprints, which have the property that the original signature becomes equal to a function of the message to be tested. Our technique is applicable for all file types.

2 Basic Idea

A file is an ordered sequence of bytes, and for the purpose of signature computation, we split the file into *n* blocks of b-bits each. We may have to pad the file with at most b-1 zeros. We refer to the file as the "message under test" (MUT) and write

$$MUT = C_1, C_2, C_3 \ldots C_n ,$$

where C_i is the i^{th} code block of the *MUT*. The notation $C(i,j)$ is used to denote the j-th bit of block C_i. Let $C^*_1, C^*_2, C^*_3 \ldots C^*_n$ be the blocks of the original Trojan-free message. The MUT is compacted into a b-bit signature using a scheme such as the *multiple-input signature register or MISR* [3]. Such a method uses a register S called signature register and initializes it to a seed value. For block j in the MUT the following computation is repeated. Let S(k) denote the k-th bit of the signature register. Let P = [P(0), P(1), …, P(b-1)] be a b-bit 0/1 sequence chosen apriori. The polynomial (1+P(1).x + P(2). x^2 + … + P(b-1). x^{b-1}) is known as the *characteristic polynomial* of the signature register.

$$S(k) = S(k-1) \oplus \boldsymbol{C}(i,j) \quad \text{for } k = 1, 2, \ldots, b-1$$

$$S(0) = \Sigma\, [S(k).P(k) \oplus C(i,k)]$$

Let S_j be the content of the signature register after the input code blocks $\boldsymbol{C_1, C_2, C_3 \ldots C_j}$ have been applied. We refer to S_n as the signature of *MUT*. Conventionally, S_n is compared with the "golden" signature G_n of the original message and the MUT is declared *infected* if S_n is different from the golden signature. We shall refer to S_j, $1 \le j < n$, as partial signatures of MUT; we can also talk of a golden partial signature G_j, $1 \le j < n$ in the same spirit, which can be obtained by compacting C^*_1, C^*_2, C^*_3 ... C^*_j into the signature register. When the original message is altered, one or more code blocks C_j will be different from that of the original block C^*_j, and one or more partial signatures of the MUT is *likely* to be different from the corresponding golden signatures. Unlike conventional schemes, where only the final signature S_n is compared with G_n, comparing multiple intermediate signatures with the respective partial golden signatures will improve the confidence of the testing scheme. Conventional signature schemes suffer from aliasing, which occurs when the signature of the infected message matches the golden signature. The concept of partial signature matching reduces the probability of aliasing.

Storing the golden partial signatures G_j in the database is expensive. We present a solution to this problem, based on the notion of *time fingerprints* of the file. A legal time finger print (LFTP) is a time instant $\boldsymbol{i}$ at which the golden partial signature $\boldsymbol{G_i}$ can be indirectly derived from the blocks of the MUT. For example, consider the time instants where $G_i = C^*_{i+1}$ i.e. the (i+1)st data block can be used in place of G_i in comparing the signature S_i with G_i, eliminating the need for storing G_i. The reader may wonder whether such time fingerprints will always exist for a message; our experiments indicate that they do, and in fact, we can prove the existence of time fingerprints (Lemma 2). A generalization of the above scheme is to define a legal time finger print as an instant i where $G_i = g(C^*_{i+1})$ where $\boldsymbol{g}$ is a one-to-one function. The choice of this function is made such that the signature register can be trivially modified to perform the additional function of comparing the current signature with $g(C^*_{i+1})$.

Figure 1 illustrates the idea behind a legal fingerprint. The error bit Y at the time instant i is defined as

$$Y_i = \begin{cases} 0 & \text{if} \quad g(C_i) = S_{i-1} \\ 1 & \text{otherwise} \end{cases}$$

The database consists of the indicator bits X at the time instant i, defined as

$$X_i = \begin{cases} 1 & \text{if} \quad i \text{ is } a \text{ legal fingerprint instant} \\ 0 & \text{otherwise} \end{cases}$$

The MUT is *infected* if the logical AND of bits X and Y is 1 at any time. A fingerprint at instant i becomes *illegal* if $S_{i-1} \neq g(C_i)$. We now present arguments about the effectiveness of our scheme in reducing aliasing. If a time fingerprint falls between two modified code blocks, it is bound to reveal the infection of the message, which may otherwise go undetected. This result is stated as the following lemma.

Lemma 1: *If, at time instant i, the partial signature $S_i \neq G_i$, and C_h is the next modified data code block, then fingerprint LFP_j, $i \le j < i+h$, must be illegal.*

Proof: Assume that a legal time fingerprint $LTFP_{i+j}$ exists at time instant $i+j$ where $0 \le j < h$. This implies $S_{i+j-1} = g(C_{i+j})$. Further C_{i+j} is a correct code block by assumption. Therefore, the partial signature $S_{i+j-1} = G_{i+j-1}$ *for all* $i \le j < i+h$. However, we have a contradiction for $i = j$.

The legal time fingerprints may also be viewed as time instances to observe the generated error bit $\boldsymbol{Y}$ and declare the message under test as infected if the error bit $\boldsymbol{Z}$ is *1*. An efficient way to generate the Trojan-free error bit $\boldsymbol{X}$ is to store only the instances of the legal fingerprints.

3 Code Blocks Functions

We define an auxiliary Boolean function that maps a b-bit input to a b-bit output by flipping j bits of the input, $0 \le j < b$. There can be 2^b such functions and we refer to them as

$H_0, H_1, \ldots, H_{2^b-1}$. The function H_i flips the j^th bit position of the input if and only if the binary representation of i contains a 1 in the j^th position. The function H_0 is the identity function and leaves the input unchanged. Given a binary string *B*, let the function *LeftRotate(B)* refer to the rotation of *B* by 1 bit to the left. For reasons that will become clear later, the signature function g is selected as $g(C^*_{i+1}) = LeftRotate(H_x(C^*_{i+1}))$ where x is determined as explained later; the legal fingerprint $LTFP_{i+1}$ will thus refer to a time instant $i+1$ where $G_i = g(C^*_{i+1})$. We denote by ?$_M$ the set of all legal fingerprints of a message under test *MUT*.

$$\gamma_M = \{LTFP_{i+1} \mid G_i = LeftRotate(C^*_{i+1}); \quad 1 \le i < n\}$$

Figure 2 illustrates the process of generating the error status ***Y*** for the MUT. An MISR is used to compact the blocks, as explained earlier. At the same time, we compare $C_i^{b-1}, C_i^{b-2}, \ldots, C_i^1$ with the signature register bits S_{b-2}, S_{b-3},…,S_0. Another EXOR gate is used to compare S_{b-1} with C_i^0. The values $C_i^{b-j} \oplus S_{b-j-1}, 1 \le j < b$ and $C_i^0 \oplus S_{b-1}$ are *OR-ed* together to generate the bit Y. (Replacing the *OR* process by *NAND* will implement the function H_{2^m-1}). The pseudocode for generating the Legal Time fingerprints LTFPs of a give file is shown in Figure 3.

3.1 Legal Time Fingerprint Space Set

For a given seed, a file can have upto 2^b different sets of legal time fingerprints. Since there are 2^b possible seeds for compacting a *b*-bit code blocks, each file can have also up to $\boldsymbol{2^b}$ legal time fingerprints for a given function H_x. Therefore, each file can have up to $\boldsymbol{2^{2b}}$ different sets of the legal time fingerprints. The following lemma gives the minimum number of LTFPs which can be obtained by implementing any function H_x in our tester application.

Lemma 2: *For a file segmented into **n** b-bit code blocks, there exists a function $\boldsymbol{H_x}$ that leads to at least* $\left\lceil \frac{n}{2^b} \right\rceil$ *legal time fingerprints, irrespective of the characteristic polynomial and the initial seed of the LFSR.*

Proof: Let $\boldsymbol{e_i}$ be the ***b***-bit vector obtained by comparing $g(C_i)$ with the golden partial signature G_{i-1}. A vector required to generate the time fingerprint for the file can assume one of the $\boldsymbol{2^b}$ possible values of $\boldsymbol{e_i}$. In the worst case, $\boldsymbol{2^b}$ successive vectors, corresponding to $\boldsymbol{2^b}$ successive code blocks of the file, are all distinct. Extending this argument, when the file has $\boldsymbol{N.2^b}$ code blocks, a specified vector must repeat at least ***N*** times. Since any vector can be implemented in the program as a legal time fingerprint LTFP for the file with a suitable choice of function $\boldsymbol{H_x}$, it is clear that when the size of the file has ***n*** code blocks, the least number of legal time fingerprints is $\left\lceil \frac{n}{2^b} \right\rceil$.

Based on similar arguments as above, we can prove the following lemma.

Lemma 3: *For a file segmented into **n** b-bit code blocks, there exists an initial seed for the LFSR buffer, which leads to at least* $\left\lceil \frac{n}{2^b} \right\rceil$ *legal time fingerprints irrespective of the characteristic polynomial and the function.*

Files Types	**Size KB**	**Number of LTFPs (b=8, function F_0)**		**Number of LTFPs (b=8, function F_{25})**	
		Bound	**LTFPs**	**Bound**	**LTFPs**
WINSOCK.DLL	22	6	25	6	12
OLE2.DLL	39	10	39	10	8
TCPTSAT.DLL	16	4	12	4	43
COMMON.COM	91	23	35	23	9
FORMAT.COM	49	13	18	13	26
CDPLAYER.EXE	32	8	31	8	3
TELNET.EXE	76	19	9	19	51
REXPROXY.EXE	58	15	37	15	23

Table 1: Number of LTFPs for some system files

3.2 Trojan Time Fingerprints TTFPs

The traditional signature-based antivirus programs protect systems from known Trojans. This approach is vulnerable, as malicious code is becoming more complex to detect. *We have observed that the compaction of the infected files yields many instants similar to that of the legal fingerprints of the original file but occur at different time instants*. These instants belong only to the malicious code. We call these instances as *Trojan time fingerprints (TTFPs)*. A TTF occurs at time instant i if

$$g(C^*_i) \ne G_{i-1} \quad and \quad g(C_i) = S_{i-1}$$

While the set of the LTFPs for a particular file can be determined by simulating the original file, no simulations are required to determine the Trojan Time fingerprints TTFPs. Testing of infected files has shown that even a single malicious instruction added to the document will generate an adequate number of TTFPs. We show that Trojan time fingerprints have many superior properties over legal time fingerprints. A single Trojan time fingerprint is sufficient to declare an infected file. To see how the TTFPs enhance the detection technique, let us consider an infected file with two Legal Time fingerprints as shown in Figure 4. The shaded portions in the figure show the range of the erroneous partial signatures for the infected file. In case A, the Trojan can be detected using illegal time fingerprint of the file. In case B, the Trojan cannot be detected using ITFPs since there are no LTFPs in the shaded portion. However, this Trojan can be detected if it can generate a Trojan Time fingerprint for itself as in case C.

Trojan time fingerprint can be easily detected by a simple modification to the procedure for detecting LTFP, namely, replacing the ***AND*** condition by a logical ***XNOR*** function as shown in Figure 5.

The merits of the Trojan time fingerprint concept are high. Any single TTFP represents a signature of some Trojan. Looking for a single TTFP is similar to running signature-based antivirus program, without the need to actually know all the actual Trojans signatures. The total number of the TTFPs and their time instances can be used for the diagnosis of the actual type of the Trojan that targeted the file.

File under test		*LTFPs*	*ITFPs*	*TTFPs*
Original File	Common.com	9	0	0
Infected File	Com1*	9	9	17
	Com2*	9	5	8
	Com3*	9	2	3

* *Infected Common.com files.*

Table 2: The number of LTFPs, ITFPs, and TTFPS

Table 2 shows the number of the legal and illegal time fingerprint as well as the Trojan time fingerprints for the DOS *Common.com* file. The original file *Common.com* was modified at its beginning (com1), at the middle (com2), and at the last portion of the file (com3). The infection of the file at the initial portion of its content will be easily detected with large number of illegal time fingerprints and Trojan time fingerprints. However, if the infection occurs only at the end of the file, the chance of detection will be less since only the last LFP will be affected and the number of TTFPs is small.

4 Selective Codes Testing

We now present a technique to improve the efficiency of the test scheme for large files. This method relies on "partial" checking of the file based on the following observation. At the LTFP time instances, the partial signature stored in the signature register is all zeros. Therefore, by selecting an initial seed of all zeros, the entire contents of the MUT is divided into a number of *segments* equal to the number of the LTFPs for that message (see Figure 6). Each segment in the document is associated with two LTFPs, the first at the beginning of the segment and the other at its end. The number of the blocks of each segment is equal to the total number of blocks included between these two fingerprints. Based on this observation, we can run the test process on any individual segment alone, or on a selected number of segments, or on the whole contents of the file. Figure 6 illustrates the process of a selective file testing in which only the second and the third segments of the file are tested for Trojan infection.

5 Test Planning

It is too time consuming to test a large number of system files periodically (say upon every login). We can schedule the test planning for each individual file in the system to be done in two separate sessions:

(1) Selective testing, where only a few segments are tested more frequently, say daily. The number of the segments to be tested can be predefined by the user, and the segments themselves can be selected randomly.

(2) Complete testing, where all the files are tested thoroughly, say on a weekly basis. The pseudocode for the selective test planning using both LTFPs and TTFPs is shown in Figure 7. The user can select the segments to be tested by identifying the first LTFP and the last LTFP instances of the selected segments, referred here by *LTFPstart,* and *LTFPend*, respectively.

6 Optimizing the Database Storage

Our simulation results indicate that each file can have a wide range of LTFP sets, including the empty set (one that has zero LTFP). Figure 8 shows an example of LTFP sets for different functions in the simulation of *RexProxy.exe* file. Different functions yield different LTFP sets, (see dashed circles). The appropriate fingerprints set for a file is determined according to the following considerations:

§ A large number of LTFPs allows early detection of the Trojan, but increases the space for the database required to store such a set. The file is segmented into a large number of small segments.

§ With a small number of LTFPs, the size of the database storage is reduced and the Trojan detection will depend on the existence of ITFPs and TTFPs.

§ With an empty set of LTFPs, we eliminate the need of the database storage. The number of segments in the file is this case is one. The testing of the file depends completely on the existence of at least one TTFP.

7 Conclusions

We described an alternate and robust method for the detection of Trojan code in system files based on the notion of time fingerprints. Our approach relies on (1) observing a finite number of fingerprints, which get generated during the course of computing the file signature and (2) tracing the Trojan time fingerprints in the infected files. A benefit to our approach is that because of these two techniques, the detection of the Trojan code in the file becomes more reliable, faster, and requires less space. We have verified the desired properties using common semi -trusted system files for UNIX and DOS operating systems. We have provided a technique to reduce, or even eliminate, the storage of the signature database. Our algorithm however is relatively new, and further analysis is of course justified, as is the case with any new proposal of this sort. We also have presented a test plan mechanism that can support both partial and the complete testing of the file.

References:

[1] Satoru T., Ryoichi S., and Masanori K., "*Seamless Object Authentication in Different Security Policy Domains*," Proceedings of the 33rd Hawaii International Conference on System Sciences – 2000.

[2] Ernst, "*Knowing ActiveX*", Prentice Hall, 1996.

[3] Yarmolik V.N., and Demidenko S. N. "*Generation and Application of Pseudorandom Sequences for Random Testing*", John Wiley and Sons, 1988.

[4] Orfali, and Harkey, "*Client/Java Programming with Java and CORBA*" John Wiley & Son's, 1997.

[5] Harold T., Stuart A. and Paul C., "*A framework for modeling Trojans and Computer Virus Infection* ", Computer Journal, 41(7), pp. 444{458, 1999.

[6] Black J, Halevi S, Krawczyk H, Krovetz T, and Rogaway P, "*UMAC: Fast and Secure Message Authentication,*" In Advances in Cryptology - CRYPTO'99, pp. 216–233, 1999.

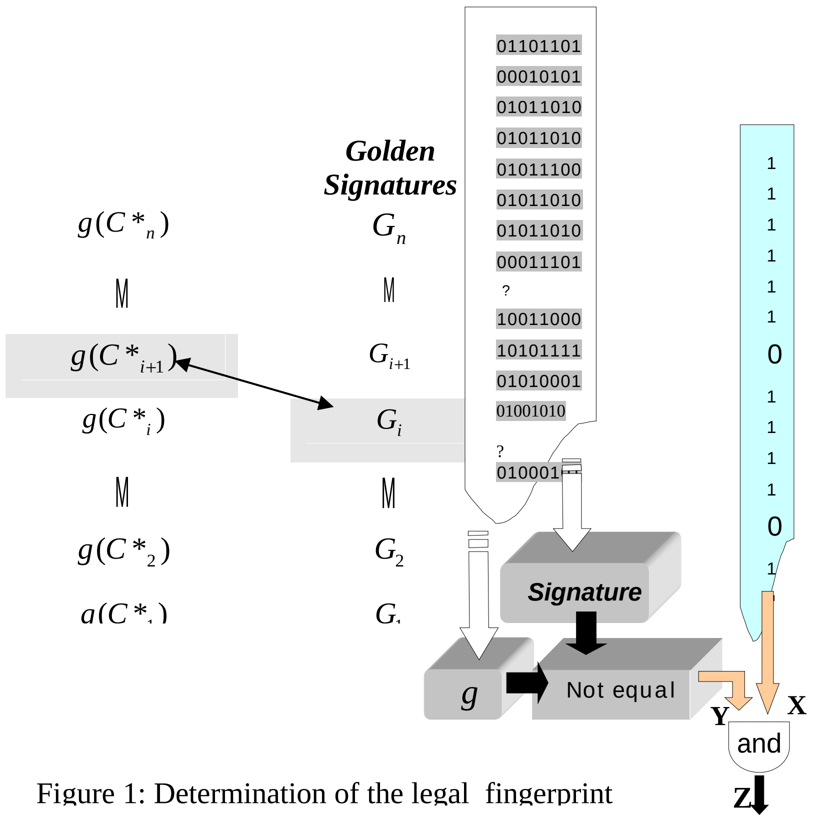

Figure 1: Determination of the legal fingerprint

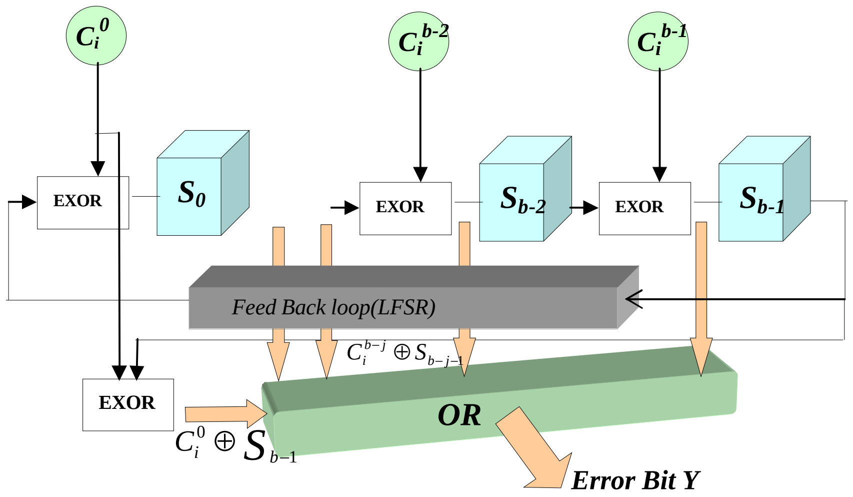

Figure 2: Generating the Error status Y

/ Given the file **msg** to be tested for possible infection, the block size **b**, the file contents are segmented into **n** blocks. The characteristic polynomial of the LFSR is fixed.*/*

```
Algorithm ERROR_STATUS (msg,b,g)
 begin
  Initialize the signature register with seed;
 Set Y to False;
 Pad msg with zeros if necessary and compute n;
  for j = 1 to n   do begin
      $Y = Y \ OR \ (S[0] \oplus g(C_j^0))$;
     for i = 1 to b-1   do begin
          FeedBackSignal = $S(0).P(0) \oplus S(1).P(1) \oplus \ldots \oplus S(b-1).P(b-1)$;
          $S[i-1] = S[i] \oplus C_j^{i-1}$;
          $Y = Y \ OR \ (S[i] \oplus g(C_j^{i-1}))$;
        // Y is the error status signal at time instance j.
        // This signal is sampled at the time instances of the legal time fingerprints.
    endfor
    $S[0] = FeedBackSignal \oplus C_j^0$;
  endfor
end
```

Figure 3: Pseudo code for computing ERROR_STATUS

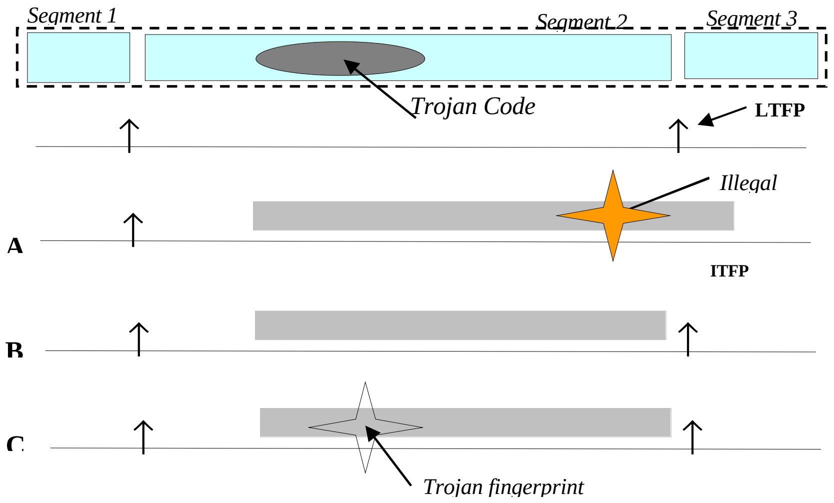

Figure 4: Detection through Legal and Trojan fingerprints

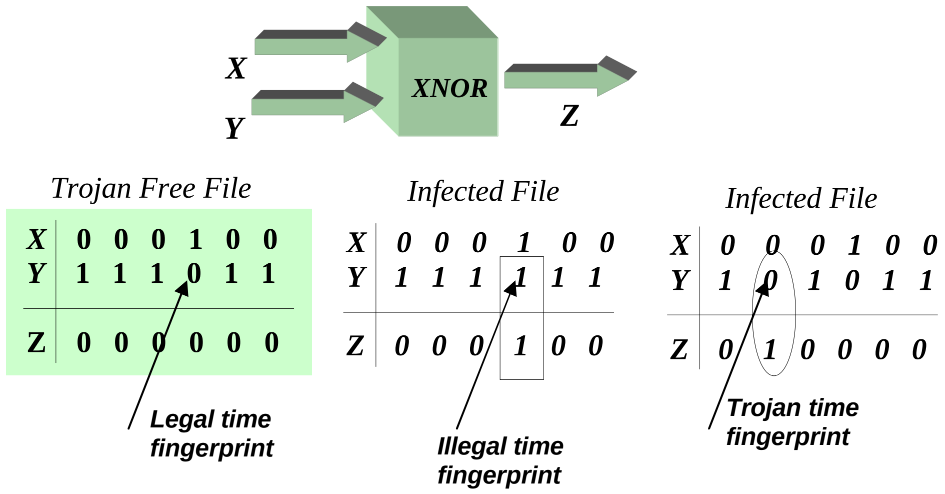

Figure 5: Concept of Trojan Time Fingerprint

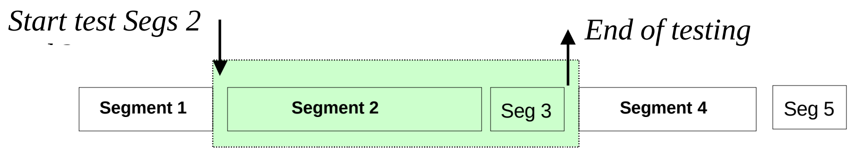

Figure 6: Selective testing to test segments 2 and 3

```
Algorithm Selective_Test (FileArray)
 // FileArray: set of files to be tested for Trojan infection
begin
  for each file € FileArray do begin
    Initialize the signature buffer;
    Select_segment(LTFP_start, LTFP_end); // Select specific segments
    Start with the first block after LTFP_start instant;
    while time $\neq$ LTFP_END do
         Generate the error bit Y; // Check for both illegal or Trojan fingerprints
         if (LTFP instant and Y=False) OR (TTFP instant) then begin
               Declare file to be infected;
               Go to next file;
         end
    end
    Declare file to be Trojan -free;
  end; // testing next file
end .
```

Figure 7: Pseudo code for Selective_Test algorithm

.

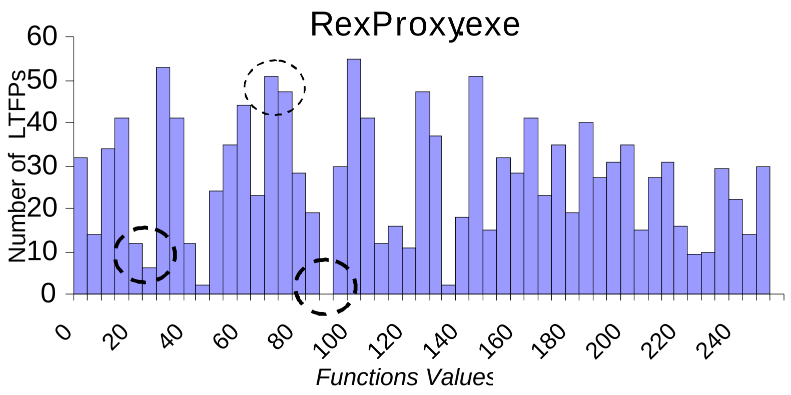

Figure 8: Sets of LTFPs for the RexProxy.exe file for different functions.