Signature Generation and Detection of Malware Families

V. Sai Sathyanarayan, Pankaj Kohli, and Bezawada Bruhadeshwar

Centre for Security, Theory and Algorithmic Research (C-STAR)
International Institute of Information Technology
Hyderabad - 500032, India
{satya_vs,pankaj_kohli}@research.iiit.ac.in, bezawada@iiit.ac.in

Abstract. Malware detection and prevention is critical for the protection of computing systems across the Internet. The problem in detecting malware is that they *evolve* over a period of time and hence, traditional signature-based malware detectors fail to detect obfuscated and previously unseen malware executables. However, as malware evolves, some semantics of the original malware are preserved as these semantics are necessary for the effectiveness of the malware. Using this observation, we present a novel method for detection of malware using the correlation between the semantics of the malware and its API calls. We construct a base signature for an entire malware class rather than for a single specimen of malware. Such a signature is capable of detecting even unknown and advanced variants that belong to that class. We demonstrate our approach on some well known malware classes and show that any advanced variant of the malware class is detected from the base signature.

Keywords: Malware Detection, Signature Generation, Static Analysis.

1 Introduction

Malware or malicious code refers to the broad class of software threats to computer systems and networks. It includes any code that modifies, destroys or steals data, allows unauthorized access, exploits or damages a system, or does something that the user does not intend to do. Perhaps the most sophisticated types of threats to computer systems are presented by malicious codes that exploit vulnerabilities in applications. Pattern based signatures are the most common technique employed for malware detection. Implicit in a signature-based method is an apriori knowledge of distinctive patterns of malicious code. The advantage of such malware detectors lies in their simplicity and speed. While the signature-based approach is successful in detecting known malware, it does not work for new malware for which signatures have not yet been prepared. There is a need to train the detector often in order to detect new malware.

One of the most common reasons that the signature-based approaches fail is when the malware mutates, making signature based detection difficult. The presence of such a metamorphism has already been witnessed in the past [5, 9].

Y. Mu, W. Susilo, and J. Seberry (Eds.): ACISP 2008, LNCS 5107, pp. 336–349, 2008.

Malware authors often tend to obfuscate the executable so as to make analysis difficult and to evade detection. Four techniques [15] are commonly employed for obfuscating executables. The first approach, *insertion of dead code* involves insertion of code that does not change the malware behavior, such as a sequence of `NOP`s (no operation instructions). The second approach, *register reassignment* involves changing the usage of one register with another such as `eax` with `ebx` to evade detection. The third approach, *instruction substitution* replaces a sequence of instructions with an equivalent instruction sequence. Finally, the fourth approach, *code transposition* involves jumbling the sequences of instructions in such a way that the behavior of the code remains the same. We note that, although all of these approaches change the code pattern in order to evade detection, the behavior of the malware still remains the same.

Past research has focused on modeling program behavior for intrusion and malware detection. Such modeling of program behavior was first studied by Forrest et al [24]. Their approach called N-Grams used short sequences of system calls to model normal program behavior. Sekar et al [25], used system calls to construct a control flow graph of normal program behavior. Peisert et al [26], use sequence of function calls to represent program behavior. Based on such results, in our approach, we have used API calls as measure of the malware program behavior. Specifically, we use only a subset of API calls, called *critical* API calls in our analysis. These critical API calls are the ones that can possibly cause malicious behavior. API calls have been used in the past research for modeling program behavior [20, 22] and for detecting malware [19, 21, 27].

We use static analysis to extract critical API calls from known malicious programs to construct signatures for an entire malware class rather than for a single specimen of malware. In our approach, a malicious program is detected by statistical comparison of its API calls with that of a malware class. The technique presented in this paper aims to detect known and unknown malicious programs, including self-mutating malware. Also, it is capable of detecting malware that use common obfuscations. Our approach relies on the fact that the behavior of the malicious programs in a specific malware class differs considerably from programs in other malware classes and benign programs. The main contributions of this paper include:

- **Detection using API calls.** We extract critical API calls from the binary executable of a program to classify it as malicious or benign. The extracted calls are subjected to a statistical likelihood test to determine the malware class.
- **Effective against common obfuscations.** Common obfuscations such as those explained above change the code pattern but do not affect the behavior of the malware. By generating a signature that reflects the behavior of the malware, our technique is able to defeat such common obfuscations. Also, since we consider only critical API calls, such obfuscations have no effect on our signature generation approach.

– **Effective against new variants.** By constructing a signature for a malware family, our approach is automatically able to detect future variants that belong to that family.

Paper Organization. In Section 2, we present the related work done in the field of malware detection. In Section 3, we describe our approach for malware detection. In Section 4, we describe a prototype implementation of our approach, present experimental results and evaluate the effectiveness of our approach. Finally, we conclude in Section 5.

2 Related Work

Several techniques have been studied in the past for malware detection. Cohen [11] and Chess & White [12] use sandboxing to detect viruses. They proved that in general the problem of virus detection is undecidable. Christodorescu and Jha [15] use static analysis to detect malicious code in executables. Their implementation called *SAFE* handles most common types of obfuscations used by malware writers, such as insertion of `NOP`s between instructions, that are used to evade detection. In [4], Christodorescu et al exploited semantic heuristics to detect obfuscated malware. Although, their approach works well for obfuscated malicious programs, the time taken (over a minute to classify) by their approach makes it impractical for use in commercial antivirus scanners. Kruegel et al [16] use control flow graph information and statistical methods for disassembling obfuscated executables. Bergeron et al [18] consider critical API calls and security policies to test for presence of malicious code. Their approach does not work for obfuscated malicious executables. Zhang et al [19] use fuzzy pattern recognition to detect unknown malicious code. The approach does not handle obfuscated program binaries and gives many false positives. Martignoni et al [7] use real-time program monitoring to detect deobfuscation in memory. Their implementation *OmniUnpack* detects obfuscation for both known and unknown packers. MetaAware [27] identifies patterns of system or library functions called from a malware sample to detect its metamorphic version. Bilar [10] uses statistical structures such as opcode frequency distribution and graph structure fingerprints to detect malicious programs. The approach presented in this paper detects malicious programs including those with common obfuscations as well as previously unknown variants of malware families.

In [17], Krugel et al use dynamic analysis to detect obfuscated malicious code, using mining algorithm. Their approach works well for obfuscated malicious programs but takes several seconds to test a single program. DOME [23] uses static analysis to detect system call locations and run-time monitoring to check all system calls are made from a location identified during static analysis. Min-Sun et al [22] use dynamic monitoring to detect worms and other exploits. Their approach is limited to detection of worms and exploits that use hard-coded addresses of API calls, and does not work for other malware types such as trojans or backdoors. Also, as evident by our experimental results, our approach is much faster than all other approaches described above.

3 Our Approach for Malware Detection

In this section, first, we briefly outline our approach for malware signature generation and classification. Next, we describe our program behavior model used for signature generation and the statistical comparison technique. Then, we present our malware detection algorithm using our program behavior model. Finally, we describe our prototype implementation in detail and show a sample signature of a malware extracted using our approach.

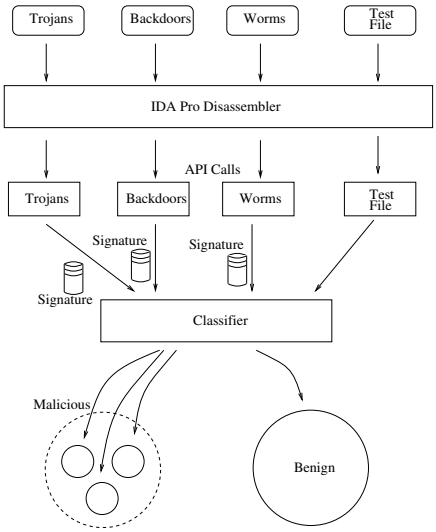

Fig. 1. Architecture of our malware detector

3.1 Malware Signature Generation and Classification Approach

We create signatures based on the characteristics of an entire malware class rather than a single sample of malware. Malware classes are defined based on similar behavior. The behavior of a malware class can be specified based on the API calls that the members of the malware calls use. For instance, a virus trying to search for executable files will typically make use of API calls such as `FindFirstFileA`, `FindNextFileA` and `FindClose` in `KERNEL32.DLL`. The behavior of searching files is captured by the use of these API calls. Rather than considering all API calls, we consider only *critical API calls* [17, 18]. Critical API calls include all API calls that can lead to security compromise such as calls that change the way the operating system behaves or those used for communication, such as Registry API, File I/O API, WinSock etc. We do not consider API calls which can be added or removed in a sample of malware without changing its malicious behavior, such as `MessageBox`, `printf`, `malloc` etc. For each malware class, we extract API calls and their call frequency from several malicious programs. The signature for the malware class is then computed using

several samples that are known to belong to that class. From our results, we have observed that 2 or 3 samples from a malware class are adequate to create a signature. Given any test file, it is classified as malicious or benign by statistical comparison of the frequency of its critical API calls with that of the malware classes. Figure 1 shows the architecture of our malware detector. Next, we describe our strategy for malware behavior profiling and show our method is used to generate signatures and classify programs as benign or malicious. In our classification, we not only differentiate between benign and malicious programs, but also between different malware classes.

Malware Behavior Profiling. Malicious programs exhibit a behavior that can be distinguished from behavior of benign programs. The signature for a malware class is based on the frequency of *critical* API calls. Let the vector $P = (f_1, f_2, \ldots, f_n)$ be a *profile* created from a program by extracting its critical API calls, where f_i represents the frequency of i^{th} critical API call and n being the total number of critical API calls.

We use a statistical measure to differentiate between malware and benign programs. To detect malware, we measure the difference between the proportions of the critical API calls in a signature and that of a test program using Chi-square test [6]. Chi-square test is a likelihood-ratio or maximum likelihood statistical significance test that measures the difference between proportions in two independent samples. The signature S_i for a malware class M_i specifies the frequencies of critical API calls that a sample of malware which belongs to M_i is expected to have. To test the membership of a given test file in a malware class, its API calls are extracted and compared to that in the signature. The Chi-square is then computed as:

$$\chi_i^2 = \frac{(O_i - E_i)^2}{E_i} \qquad ; 1 \leq i \leq n$$

Here, O_i is the observed frequency of the i^{th} critical API call in the test file and E_i is its expected frequency, i.e. frequency in the signature of a malware class. Now, χ^2 is compared against a threshold value ϵ from a standard Chi-square distribution table with one degree of freedom. The degrees of freedom is associated with the number of parameters that can vary in a statistical model. A significance level of 0.05 was selected. This means that 95% of the time we expect χ^2 to be less than or equal to ϵ. For one degree of freedom and significance level 0.05, $\epsilon = 3.84$. Let $U = \{API_i \mid \chi_i^2 \leq \epsilon_i\}$. We define a *degree of membership* λ as

$$\lambda = \frac{|U|}{n}$$

Degree of membership λ is a measure of belongingness of test file to a malware class. The statistical profiling algorithm is shown in Algorithm 1.

Input: API frequency set for a file, $P = \{O_1, O_2, \ldots, O_n\}$, and another API frequency set $M = \{E_1, E_2, \ldots, E_n\}$
Output: Degree of membership, λ
for $i = 1$ *to* n **do**
  $\chi_i^2 = \frac{(O_i - E_i)^2}{E_i}$;
end
$U = \{API_i \mid \chi_i^2 \leq 3.84\}$;
$\lambda = \frac{|U|}{n}$;
return λ

Algorithm 1. STAT(P, M)

Signature Generation. The signature for a malware class is then computed as follows. Let $R_i = \{P_1^i, P_2^i, \ldots, P_m^i\}$ be the set of profiles of samples in malware class M_i. The signature vector S_i for the malware class M_i is then defined as the set of the mean frequency of every critical API call occurring in M_i.

$$S_i = \frac{1}{m} \sum_{j=0}^{m} P_j^i$$

This signature vector is then tested against samples $T = \{T_1, T_2, \ldots, T_k\}$ known to belong to the same malware class M_i using the statistical analysis. We here define a threshold δ as

$$\delta_i = \frac{1}{k} \sum_{j=0}^{k} \lambda_j$$

Here λ is the outcome of a statistical analysis test. This signature S_i and threshold δ_i is computed for every malware class M_i. We note that each individual test sample shows a distinct set of frequencies, which differ noticeably from those shown by benign programs and other malware classes.

Classification Strategy. Let P be the profile obtained from a test file T. Let S_i be a signature for the malware class M_i, and δ_i be the corresponding degree of membership. Let B be the benign set and t be the total number of malware classes.

Then, if

$$\exists\ i,\ 1 \leq i \leq t, \quad \delta_T \geq \delta_i \\ \Rightarrow \quad T \in M_i$$

Otherwise, if

$$\forall\ i,\ 1 \leq i \leq t, \qquad \delta_T < \delta_i \\ \Rightarrow \quad T \ \in \ B$$

Also, if

$$\exists\ i, j,\ 1 \leq i, j \leq t, \qquad \delta_T \geq \delta_i \quad AND \quad \delta_T \geq \delta_j \\ \Rightarrow T \in M_i \cup M_j$$

Note that if $\delta_T \geq \delta_i$ and $\delta_T \geq \delta_j$, then it means the test file T contains features of both malware classes M_i and M_j.

A *false positive* occurs when a benign program is classified as malicious. A false positive for a signature S_i is defined as the probability

$$Pr\left(\delta_T \geq \delta_i \mid T\ \in\ B\right)$$

A *false negative* occurs when a malicious program is classified as benign. For a specific malware class M_i and signature S_i, this is defined as

$$Pr\left(\delta_T < \delta_i \mid T \in M_i\right)$$

This usually happens when the data in the profile is distorted and therefore M_i cannot be detected. We now formally state our malware detection algorithm. The algorithm is composed of two parts: the signature generator *SIGNATURE_GENERATE* (Algorithm 2) and the detector *DETECT* (Algorithm 3).

Input: The set of profiles $R_i = \{P_1^i, P_2^i \ldots, P_m^i\}$ for a malware class M_i
Output: Signature S_i and threshold δ_i

1 Select an arbitrary set $U \subset M_i$. Let $U = \{U_1, U_2, \ldots, U_k\}$.
2 Let $Q = M_i - U$. Let $Q = \{Q_1, Q_2, \ldots, Q_{m-k}\}$.
3 Compute signature as $S_i = \frac{1}{m-k} \sum_{j=1}^{m-k} Q_j$;
4 **for** $j = 1$ *to* k **do**
5 $\quad \lambda_j = STAT(S_i, U_j)$;
6 **end**
7 Compute threshold as $\delta_i = \frac{1}{k} \sum_{j=1}^{k} \lambda_j$.

Algorithm 2. SIGNATURE_GENERATE(M_i)

3.2 Prototype Implementation Details

We have implemented a prototype of the technique. Our implementation is written for malware on Win32 platform and it consists of two components - API call extractor and Classifier.

API Call Extractor. The API Call Extractor component is implemented as a plugin to the IDA Pro Disassembler [8]. It begins by locating the `.idata` segment which is an `EXTERN` segment that contains list of addresses of API functions imported by the PE file. For each address in the `.idata` segment, it retrieves the corresponding API function name and its set of cross-references. The API

Input: A test file T with API frequency set $P = \{f_1, f_2, \ldots, f_n\}$, a signature S_i and corresponding threshold δ_i for a malware class M_i
Output: TRUE if $T \in M_i$, FALSE otherwise
$\delta_T = STAT(P, S_i)$;
if $\delta_T \geq \delta_i$ **then**
  return *TRUE*
end
return *FALSE*

Algorithm 3. DETECT(T, M_i)

```
.idata:0040F2F0 ; int __stdcall send(SOCKET s, const char *buf, int len, int flags)
.idata:0040F2F0                 extrn __imp_send:dword  ; DATA XREF: send
```

Fig. 2. API function `send` in `.idata` segment

```
.text:004019A7 loc_4019A7:
; CODE XREF: sub_401990+31
.text:004019A7                  push    0               ; flags
.text:004019A9                  push    1               ; len
.text:004019AB                  push    esi             ; buf
.text:004019AC                  push    ebx             ; s
.text:004019AD                  call    send
....
....
....
.text:004019C3 loc_4019C3:
; CODE XREF: sub_401990+13
.text:004019C3                  push    0               ; flags
.text:004019C5                  add     edi, ebp
.text:004019C7                  push    1               ; len
.text:004019C9                  push    edi             ; buf
.text:004019CA                  push    ebx             ; s
.text:004019CB                  call    send
```

Fig. 3. Calls to API function `send` that actually transfer control to an intermediate thunk

```
.text:00401FE6
.text:00401FE6 ; Attributes: thunk
.text:00401FE6
.text:00401FE6 ; int __stdcall send(SOCKET s, const char *buf, int len, int flags)
.text:00401FE6 send             proc near
; CODE XREF: sub_401990+1D
.text:00401FE6                                          ; sub_401990+3B
.text:00401FE6                  jmp     ds:__imp_send
.text:00401FE6 send             endp
```

Fig. 4. Thunk for API function `send`

call frequency is given by the number of cross-references in the code region. Note that, in many cases compiler generates code in such a way that a `call` to an API function is made through an intermediate `jmp` instruction, called a *thunk*.

```
...
...
GetWindowsDirectory 1.625000
WriteFile 9.375000
GetFileAttributes 1.125000
CopyFile 3.000000
DeleteFile 6.375000
CreateFile 9.000000
SetFileAttributes 1.125000
GetTempPath 2.375000
GetSystemDirectory 3.250000
GetModuleFileName 6.500000
...
...
```

Fig. 5. Signature for MyDoom worm family

In such a case, if a cross-reference is a thunk, it may lead to an incorrect API call frequency since several API calls will transfer control to the thunk which in turn would jump to the actual API function. Therefore, we check each cross-reference and if it is a thunk, we retrieve all cross-references for this thunk as well to get the correct API call frequency. Such a code taken from the disassembly of Borzella worm [3] is shown in Figures 2,3 and 4.

Classifier. The classifier reads the entire set of profiles produced by the API call extractor for each malware class and produces a signature. When given a file containing API call frequencies of a test file, it uses the above algorithm to classify the test file as benign or as the appropriate malware class. Figure 5 shows a sample signature created for *MyDoom* worm family.

4 Experimental Analysis

The testing environment consisted of a Windows XP Service Pack 2 machine. The hardware configuration included a Pentium 4 3.2 GHz processor and 512 MB of RAM. We used IDA Pro version 5.2.0.

4.1 Effectiveness

Testing on new variants. To test the effectiveness of our malware detector against new variants of malware families, we tested it on eight malware families. The malware families were gathered from VX Heavens [2]. For each malware family, we used two earliest possible variants to construct the signature and the rest for testing the signature. We tested our approach on the following malware families: MyDoom(30 variants), Bifrose (18 variants), Agent (14 variants), Delf (13 variants), InvictusDLL (13 variants), Netsky (10 variants), Bagle (9 variants) and Chiton (19 variants). Our approach was able to detect all variants in the above malware families except one variant in Netsky family. The detailed results are presented in Table 1 and 2.

Although, from Table 1, Netsky.r could not be detected when using the signature created from Netsky.c and Netsky.d, but it was detected when the signature

was generated from Netsky.c and Netsky.p. From these results, we note that our approach is most suited for detecting many variants of a malware family. This implies that if there is a new variant that is not classified by our approach, it is probable that the malware writer has made some significant changes in its behavior. In such a case, that variant can be used for training which will be sufficient for detecting many more advanced variants of the family. We have illustrated this from the Netsky worm example.

Testing on generic malware classes. Above experiments tested specific malware families. We wanted to test our approach for detecting arbitrary and unknown malware classes by using only signatures generated from some known broad classes of malware. So, we constructed signatures for broad classes of malware such as trojans, worms, backdoors and viruses. To test the effectiveness of our detection method and to identify potential false negatives, we gathered 800 malicious programs in Portable Executable (PE) [1] format. To test the false positive rate, we gathered 200 benign programs from a fresh installation of Windows XP service pack 2. Signatures for malware classes were constructed by incrementally chosing higher number of training samples such 10, 20 and so on upto 60 samples from each malware class. 29 benign programs out of 200 were incorrectly classified as malicious. The evaluation results are presented in Table 3.

We found that several benign programs share behavior (for instance, searching files, copying files to network drives etc.) with certain malicious programs. The observed false positive rate is due to such *shared behavior*. Figure 6 shows the plot of detection rate, false negative rate and false positive rate with increasing

Table 1. Effectiveness evaluation to detect malware variants

Malware	Variants in training set	Variants Tested	Detected	Malware	Variants in training set	Variants Tested	Detected
Netsky	c d	e	✓	**Chiton**	a b	c	✓
		gen	✓			d	✓
		l	✓			e	✓
		m	✓			f	✓
		n	✓			h	✓
		p	✓			i	✓
		r	✗			j	✓
		x	✓			k	✓
Bagle	a bb	b	✓			l	✓
		ab	✓			m	✓
		ad	✓			n	✓
		ae	✓			o	✓
		al	✓			p	✓
		as	✓			q	✓
		bi	✓			r	✓
						t	✓
						Chiton	✓

Table 2. Effectiveness evaluation to detect malware variants (contd.)

Malware	Variants in training set	Variants Tested	Detected	Malware	Variants in training set	Variants Tested	Detected
MyDoom	a c	d	✓	**Bifrose**	a ab	ae	✓
		e	✓			ag	✓
		f	✓			aq	✓
		g	✓			at	✓
		h	✓			ax	✓
		l	✓			bb	✓
		o	✓			bc	✓
		q	✓			bf	✓
		r	✓			bg	✓
		u	✓			bh	✓
		v	✓			bk	✓
		y	✓			bl	✓
		aa	✓			bo	✓
		ae	✓			bs	✓
		af	✓			ca	✓
		ag	✓			cc	✓
		ai	✓	**Agent**	a ab	ad	✓
		aj	✓			ae	✓
		ak	✓			ah	✓
		al	✓			aj	✓
		an	✓			bc	✓
		aq	✓			bd	✓
		ar	✓			abz	✓
		as	✓			aci	✓
		at	✓			acx	✓
		av	✓			adr	✓
		ay	✓			ads	✓
		az	✓			aec	✓
Delf	62976 c	d	✓	**InvictusDLL**	101.a 101.b	099	✓
		f	✓			201.b	✓
		g	✓			102	✓
		h	✓			103.a	✓
		j	✓			200.b	✓
		k	✓			201.a	✓
		m	✓			200.a	✓
		n	✓			a	✓
		r	✓			b	✓
		v	✓			c	✓
		w	✓			d	✓

number of training samples. As shown in the figure, the false positive and false negative rate falls and the detection rate increases with an increase in the number of training samples. Hence, it can be inferred from the plot that the accuracy of the signature increases with an increase in the number of training samples. The results show that even in the absence of the base signature, our technique was able to detect a new malware using the signature constructed from broad malware classes with reasonable accuracy. Once the new malware is detected, its base signature can easily be constructed to detect its future variants.

Table 3. Evaluation Results

Class	Tested	Detected	False negatives
Worms	131	121	10
Trojans	362	300	62
Backdoors	161	103	58
Viruses	146	146	0

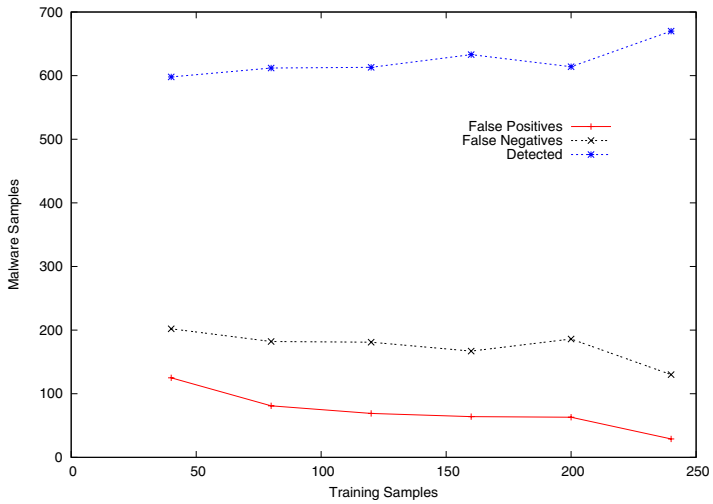

Fig. 6. Change in Detection Rate and False Negative Rate with change in number of training samples

Table 4. Time (in seconds) comparison with SAFE

Malware	Annotator/API Call Extractor		Detector	
	SAFE	Our Approach	SAFE	Our Approach
Chernobyl	1.444	2.172	0.535	0.0138
zombie-6.b	4.600	1.718	1.149	0.0314
f0sf0r0	4.900	1.781	0.923	0.0256
Hare	9.142	1.665	1.604	0.0282

4.2 Performance Testing

We tested the time it requires to classify a given file as malicious or benign. We consider the time taken by our approach to extract the API calls and to classify it as malicious or benign. We compare our approach to SAFE [15]. SAFE creates an abstraction pattern of the malicious code and converts it into an internal representation. Given a test program, it creates a control flow graph (CFG) of the test program, and checks whether the internal representation of malicious

code is present in the CFG. SAFE has been tested only on a very few malware samples. Table 4 compares the time taken by our approach with that of SAFE for four samples of malware. Clearly, our approach is much faster than SAFE.

5 Conclusion and Future Work

We presented a method to generate signatures for malware classes to detect previously unknown malicious programs. Our malware detection approach is space efficient. Rather than creating a new signature for every variant in a malware family, it creates a single signature that reflects the behavior of the entire family. It reduces the human effort required to generate a signature for a new malware. Also, it is able to detect malicious programs with common obfuscations, a problem which the commercial antivirus scanners being used today do not address. Thus, our malware detection approach is most suitable for use in commercial antivirus scanners.

The accuracy of our signature generation method for detecting future variants of a malware family is good. Although the detection error rate for new malware in broad classes such as trojans and backdoors seems high in our experiments but the results are encouraging. Malware authors often tend to *pack* malware in order to evade detection and to make analysis difficult. Such malware use a decompression or decryption routine to extract the compressed or encrypted malicious code in memory. A limitation of our approach is that it does not work for packed malware. The future work involves incorporating a generic unpacking technique to detect even the packed malware and extending the signature generation algorithm to better utilize API calls to reduce the error rate.

References

[1] Pietrek, M.: An In-Depth Look into the Win32 Portable Executable File Format, in MSDN Magazine (March 2002)
[2] VX Heavens, `http://vx.netlux.org`
[3] Viruslist.com - Email-Worm.Win32.Borzella, `http://www.viruslist.com/en/viruses/encyclopedia?virusid=21991`
[4] Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-Aware Malware Detection. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy, May 08-11, 2005, pp. 32–46 (2005)
[5] Marinescu, A.: An Analysis of Simile, `http://www.securityfocus.com/infocus/1671`
[6] Sokal, R.R., Rohlf, F.J.: Biometry: The principles and practice of statistics in biological research, 3rd edn. Freeman, New York (1994)
[7] Martignoni, L., Christodorescu, M., Jha, S.: OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In: Twenty-Third Annual Computer Security Applications Conference (ACSAC), Miami Beach, FL (December 2007)
[8] Guilfanov, I.: An Advanced Interactive Multi-processor Disassembler (2000), `http://www.datarescue.com`
[9] Ferrie, P., Ször, P.: Zmist opportunities. Virus Bullettin (2001)

[10] Bilar, D.: Statistical Structures: Tolerant Fingerprinting for Classification and Analysis given at BH 2006, Las Vegas, NV. Blackhat Briefings USA (August 2006)
[11] Cohen, F.: Computer Virus: Theory and experiments. Computers and Security 6, 22–35 (1987)
[12] Chess, D.M., White, S.R.: An undetectable computer virus. In: Proceedings of Virus Bulletin Conference (2000)
[13] Landi, N.: Undecidability of static analysis. ACM Letters on Programming Language and systems (LOPLAS) 1(4), 323–337 (1992)
[14] Myres, E.M.: A precise interprocedural data flow algorithm. In: Conference Record of the 8th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 1981), pp. 219–230. ACM Press, New York (1981)
[15] Christodorescu, M., Jha, S.: Static Anlaysis of Executables to Detect Malicious Patterns. In: Proceeding of the 12th USENIX Security Symp (Security 2003), pp. 169–186 (August 2003)
[16] Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: Proceedings of USENIX Security, San Diego, CA, pp. 255–270 (August 2004)
[17] Christodorescu, M., Jha, S., Krugel, C.: Mining Specification of Malicious Behavior. In: Proceeding of the 6th joint meeting of the European Software Engineering Conference. ACM SIGSOFT Symp. On ESES/FSE 2007 (2007)
[18] Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M.M., Lavoie, Y., Tawbi, N.: Static Detection of Malicious Code in Executable Programs. In: Symposium on Requirements Engineering for Information Security (SREIS 2001) (2001)
[19] Zhang, B., Yin, J., Hao, J.: Using Fuzzy Pattern Recognition to Detect Unknown Malicious Executables Code. In: Wang, L., Jin, Y. (eds.) Fuzzy Systems and Knowledge Discovery. LNCS (LNAI), vol. 3613, pp. 629–634. Springer, Heidelberg (2005)
[20] Peisert, S., Bishop, M., Karin, S., Marzullo, K.: Analysis of Computer Intrusions Using Sequences of Function Calls. IEEE Transactions on Dependable and Secure Computing (TDSC) 4(2) (April-June, 2007)
[21] Bergeron, J., Debbabi, M., Erhioui, M.M., Ktari, B.: Static Analysis of Binary Code to Isolate Malicious Behaviors. In: Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises, June 16-18, 1999, pp. 184–189 (1999)
[22] Sun, H.-M., Lin, Y.-H., Wu, M.-F.: API Monitoring System for Defeating Worms and Exploits in MS-Windows System. In: Batten, L.M., Safavi-Naini, R. (eds.) ACISP 2006. LNCS, vol. 4058. Springer, Heidelberg (2006)
[23] Jesse, C., Rabek, R., Khazan, I., Scott, M., Robert, L., Cunningham, K.: Detection of Injected, Dynamically Generated,and Obfuscated Malicious Code. In: Proc. of 2003 ACM workshop on Rapid Malcode (October 2003)
[24] Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A Sense of Self for Unix Processes. In: IEEE Symposium on Security and Privacy 1996 (1996)
[25] Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In: IEEE Symposium on Security and Privacy (2001)
[26] Peisert, S., Bishop, M., Karin, S., Marzullo, K.: Analysis of Computer Intrusions Using Sequences of Function Calls. IEEE Transactions On Dependable and Secure Computing 4(2) (April-June, 2007)
[27] Zhang, Q., Reeves, D.S.: MetaAware: Identifying Metamorphic Malware. In: Annual Computer Security Applications Conference (ACSAC) (2007)