Malware: From Modelling to Practical Detection*

R.K. Shyamasundar, Harshit Shah, and N.V. Narendra Kumar

School of Technology and Computer Science
Tata Institute of Fundamental Research
Homi Bhabha Road, Mumbai 400005, India
{shyam,harshit,naren}@tcs.tifr.res.in

Dedicated to the memory of
Amir Pnueli: 1941-2009
A Computer Science Pioneer and A Great Human being

Abstract. Malicious Software referred to as Malware refers to a software that has infiltrated to a computer without the authorization of the computer (or the owner of the computer). Typical categories of malicious code include Trojan Horses, viruses, worms etc. Malware has been a major cause of concern for information security. With the growth in complexity of computing systems and the ubiquity of information due to WWW, detection of malware has become horrendously complex. In this paper, we shall survey the theory behind malware to provide the challenges behind detection of malware. It is of interest to note that the power of the malware (or for that matter computer warfare) can be seen in the theories proposed by the iconic scientists Alan Turing and John von Neumann. The malicious nature of malware can be broadly categorized as injury and infection analogously in the epidemiological framework. On the same lines, the remedies can also be thought of through analogies with epidemiological notions like disinfection, quarantine, environment control etc. We shall discuss these aspects and relate the above to notions of computability.

Adleman in his seminal paper has extrapolated protection mechanisms such as quarantine, disinfection and certification. It may be noted that most of the remedies in general are undecidable. We shall discuss remedies that are being used and contemplated. One of the well-known restricted kind of remedies is to search for signatures of possible malwares and detect them before getting it through to the computer. Large part of the current remedies rely on signature based approaches that is, heavy reliance on the detection of syntactic patterns. Recent trends in security incidence reports show a huge increase in obfuscated exploits; note that in the majority of obfuscators, the execution behaviour remains the same while it can escape syntactic recognitions. Further, malware writers are using a combination of features from various types of classic malwares such as viruses and worms. Thus, it has become all the more necessary to

* The work was partially supported under Indo-Trento Promotion for Advanced Research.

take a holistic approach and arrive at detection techniques that are based on characterizations of malware behaviour that includes the environment in which it is expected to execute.

In the paper, we shall first survey various approaches of behavioural characterization of malware, difficulties of virus detection, practical virus detection techniques and protection mechanisms from viruses. Towards the end of the paper, we shall briefly discuss our new approach of detecting malware via a new method of validation in a quarantine environment and show our preliminary results for the detection of malware on systems that are expected to carry a priori known set of software.

1 Introduction

Malicious code is any code that has been modified with the intention of harming its' usage/user. Informally, destructive capability of malware is assessed based on how well it can hide itself, how much loss it can inflict on the owner/user of the system, how rapidly it can spread, etc. Malware attacks result in tremendous costs to an organization in terms of cleanup activity, degraded performance, damage to its reputation, etc. Among some of the direct costs incurred are, labour costs to analyze and cleanup infected systems, loss of user productivity, loss of revenue due to direct losses or degraded performance of system, etc. According to a report on financial impact of malware attacks [35], the direct damages incurred in 2006 were USD 13 billion. Although the trend shows a decline in direct damages since 2004 (direct damages in 2004 and 2005 were USD 17.5 billion and USD 14.2 billion respectively), this decline is attributed to a shift in the focus of malware writers from creating damaging malware to creating stealthy, fast-spreading malware so that infected machines can be used for sending spams, stealing credit-card numbers, displaying advertisements or opening a backdoor to an organization's network. This increase in the indirect and secondary damages (that are difficult to quantify) explains why malware threat has worsened in recent years despite a decline in direct damages.

Malware can be primarily categorized [15] as follows:

– Virus - Propagates by infecting a host file.
– Worm - Self-propagates through e-mail, network shares, removable drives, file sharing or instant messaging applications.
– Backdoor - Provides functionality for a remote attacker to log on and/or execute arbitrary commands on the affected system.
– Trojan - Performs a variety of malicious functions such as spying, stealing information, logging key strokes and downloading additional malware - several further sub categories follow such as infostealer, downloader,dropper,rootkit etc.
– Potentially Unwanted Programs (PUP) - Programs which the user may consent on being installed but may affect the security posture of the system or may be used for malicious purposes. Examples are Adware, Dialers and Hacktools/hacker tools (which includes sniffers, port scanners, malware constructor kits, etc.)

– Other - Unclassified malicious programs not falling within the other primary categories.

The breakdown of malware as per the study reported in [15], is summarized in Table 1. It shows that Trojan comprised a major class of malware in 2008. This is indicative of attacker's preference to infect machines so that a host of malicious activities (advantageous to the attacker) can be launched from them. Signature based scanning techniques for malware detection are highly inadequate. With increased sophistication in detection techniques, stealth techniques employed by malware writers have also witnessed huge sophistication.

Table 1. 2008 Malware trends

Malware Class	**Percentage**
Trojan	46
Other	17
Worm	14
Backdoor	12
PUP	6
Virus	5

The theory of computer viruses was developed much before the actual instances of it were seen in wild. The foundations were laid by Cohen's formal study [10] of computer viruses based on the seminal theory of self-reproducing automata invented by John von Neumann [26]. In this paper, we first survey[1] some of the important theoretical results on computer viruses, several detection techniques, and protection mechanisms. Finally, we briefly discuss our recent study for detecting the presence of malwares in systems like embedded systems where we know a priori the software that is loaded in them.

2 Formal Approaches to Virus Characterization

Informally, a virus can be defined as any program that propagates itself by infecting a host file (trusted by the user). The infected host file when executed, in addition to performing its' intended job it also selects another program and infects it thereby spreading the infection.

Cohen [10] was the first to formalize and study the problem of computer viruses. The following definition intuitively captures the essence of a virus as defined by Cohen [10].

Definition 1. *A computer virus is a program that can infect other programs, when executed in a suitable environment, by modifying them to include a possibly evolved copy of itself.*

[1] For an excellent detailed exposition of computer viruses from theory to practice, the reader is referred to [14].

We now present the pseudo-program of a possible structure of a computer virus as presented in [9].

```
program virus:=
{1234567;

subroutine infect-executable:=
{loop: file = get-random-executable-file;
if first-line-of-file = 1234567 then goto loop;
prepend virus to file;
}

subroutine do-damage:=
{whatever damage is to be done}

subroutine trigger-pulled:=
{return true if some condition holds}

main-program:=
{infect-executable;
if trigger-pulled then do-damage;
goto next;}

next:}
```

Although the above pseudo-program of a virus includes a subroutine to perform *damage*, the ability to perform damage is not considered a vital characteristic of a virus by Cohen.

Cohen's formal definition of a virus was based on the Turing machine computing model. The possibility of a virus infection comes from the theory of self-reproducing automata defined by Jon Von Neumann [26]. Every program that gets infected can also act as a virus and thus, the infection can spread throughout a computer system or a network.

Cohen identifies the ability to infect as the key property of a virus, thus allowing it to spread to the transitive closure of information sharing. Given the widespread use of sharing in current computer systems, viruses can potentially damage a large portion of the network. Recovering from such a damage will be extremely hard and perhaps often impossible. Thus, it is of great importance to detect and protect systems from viruses.

Cohen's formalization was remarkable, because it captures the intuition that a program is a virus only when executed in a suitable environment. However, Cohen's formal definition does not fully capture the relationship between a virus and a program infected by that virus. Soon after Cohen's formalization of viruses, Adleman (Cohen's Ph.D advisor) proposed a formalization of viruses using recursive functions computing model. Before providing Adleman's definition of viruses, let us set up some basic notation and concepts as given in Adleman [1].

A virus can be thought of as a program that transforms (infects) other programs.

Definition 2. *If v is a virus and i is any program, $v(i)$ denotes the program i upon infection by virus v*

A system on which a program is executing can be characterized by giving the set of data and programs that are present in the system. A program can be thought of as a state transformer. If i is a program, d is a sequence of numbers that denotes the data in a system and p is a sequence of numbers that denotes the programs in a system then $i(d, p)$ denotes the state resulting when program i executes in the system. Together d and p tell us the state of the system.

Definition 3. *We say that state (d_1, p_1) is v-related to state (d_2, p_2), denoted $(d_1, p_1) \cong_v (d_2, p_2)$ iff*

- $d_1 = d_2$ *and*
- $p_1 \neq p_2$ *i.e.* p_1 *and* p_2 *differ*
- *number of programs in* p_1 *and* p_2 *are the same and*
- *either the ith program in p_1 and the ith program in p_2 are the same or the ith program in p_2 results when the ith program in p_1 is infected by virus v*

Intuitively Adleman's definition of a virus can be stated as follows:

Definition 4. *A program v, that always terminates, is called a virus iff for all states s either*

1. *Injure: all programs infected by v behave the same when executed in state s*
2. *Infect or Imitate: for every program p, the state resulting when p is executed in s is v-related to the state resulting when $v(p)$ is executed in s*

Adleman's definition of a virus characterizes the relationship between a virus and a program infected by it. However, there is no quantification or characterization of injury and infection. Although the notion of injury appears explicitly in his definition of a virus, Adleman considers the ability to infect to be the core of a virus (Remark 2 in Adleman [1]). Based on this definition he arrives at a classification of viruses[2] into the four disjoint classes *benign*, *Epeian*, *disseminating* and *malicious.*

Intuitively, *benign viruses* are those which never injure the system nor infect other programs. Consider a program p, that compresses programs to save disk space, and adds a decompression routine to its binary so that the program gets decompressed during execution. p is an example of a benign virus.

Epeian viruses cause damage in certain conditions but never infect other programs. Boot sector viruses and other programs that delete some key files of the system are examples of Epeian viruses.

[2] For immunological analogies between computer and biological viruses, the reader is referred to[17].

Disseminating viruses spread by infecting other programs but never injure the system. Internet worms like Netsky, Bagle, Sobig, Sober, MyDoom, Conficker etc., are examples of disseminating viruses.

Malicious viruses are those programs that cause injury in certain conditions and propagate themselves by infecting other programs in certain conditions.

3 Techniques for Detecting Viruses

Cohen [9] considers the problem of detecting a computer virus and proves that it is undecidable in general to detect a virus by its appearance (static analysis). We now give an intuitive (informal) account of Cohen's result

Theorem 1. *Detecting a virus by its appearance is undecidable.*

Proof. Proof is by contradiction.
Let us assume to the contrary that there exists a procedure D that can tell whether a program is a virus or not. Let there be a program p which infects other programs if and only if D would have called it benign. Given p as input, if D calls it benign, then p infects – thus leading to a contradiction. If on the other hand D calls it a virus then p does not infect, again leading to a contradiction. All the possibilities lead to contradiction.

Thus, it is not possible to have a procedure such as D.

The above theorem illustrates that if we know what defense mechanism is used by the defender, we can always build a virus that infiltrates into the system i.e., fools the defender. No defense is perfect. Similarly, given a virus, there is always a defense system that defends against that particular virus.

Adleman considers the problem of detecting whether a program is a virus or not and proves it undecidable. We present the theorem as in [1].

Theorem 2. *For all Godel numberings of the partial recursive functions* $\{\phi_i\}$

$$V = \{i|\phi_i \textit{ is a virus}\} \textit{ is } \prod_2 -\textit{complete}$$

What this theorem implies is that it is impossible to always correctly tell whether a given program is a virus or not. However, it is very important to be able to detect viruses and nullify them before they carry out the damage.

Having had a glance of the possibilities, let us look at widely used methods for detecting viruses.

3.1 Signature Based Detection

In signature based detection of viruses, we have a database of known malicious patterns of instructions. Whenever a file is scanned the detection algorithm compares the sequence of symbols present in the file with the database of known malicious patterns. If the algorithm finds a match it declares the file to be a virus.

Note that signature based detection algorithm critically depends on the database of known malicious patterns. This database is created by analyzing known viruses by extracting sequences of instructions present in them and removing any sequences from them that are typical of benign programs. If we remove very little, we are in danger of not being able to identify even minor modifications of the virus. If we remove too much, then we are in danger of marking genuine programs as viruses. Since a lot of discretion is needed to remove benign patterns, this process of building the database is very hard to automate and is typically compiled by human experts. For this reason it takes reasonable time (even 4 to 5 days) for adding the signature of a newly detected virus to the database, and by this time the virus would have spread wide and caused damage.

Thus, signature based detection techniques work well for known viruses but cannot handle new viruses. This inability limits their effectiveness in controlling the spreading of new malware. Because of their syntactic nature, these techniques are susceptible to various program obfuscation techniques that preserve the program behaviour but change the program code in such a way that the original program and the modified program look very different.

In [6], Chrisodorescu et al., reveal gaping holes in signature-based malware detection techniques employed by several popular, commercial anti-virus softwares. In their technique, a large number of obfuscated versions of known viruses are created and tested on several anti-virus softwares. The results demonstrate that these tools are severely lacking in their ability to detect obfuscated versions of known malware. An algorithm to generate signatures used by different anti-virus softwares to detect known malware is also presented. Results show that in many cases, the whole body of the malware is used as a signature for detection. This inability to capture malware behaviour comprehensively explains their failure to detect obfuscated versions of malware.

IBM Internet Security Systems X-Force 2009 Mid-Year Trend and Risk Report [16] states that the level of obfuscation found in Web exploits and in PDF files continues to increase while some of these obfuscation techniques are even being passed to multimedia files. The amount of suspicious obfuscated content has nearly doubled from Q1 to Q2 of 2009. In the following, we discuss obfuscation techniques.

Program Obfuscation Techniques

Program obfuscation techniques modify a program in such a way that its behaviour remains the same but analysis of the obfuscated program becomes difficult. The main objective for such transformations is to prevent/make difficult reverse engineering in order to protect intellectual property or to prevent illegal program modifications. Several popular techniques for program obfuscation are code reordering, instruction substitution, variable renaming, garbage insertion and data and code encapsulation. An exhaustive list of obfuscating transformations and measures to gauge their efficacy are presented in [11]. Obfuscation techniques are heavily used by malware programs to evade detection.

Substantial research has been done on theoretical aspects of program obfuscators. In [2], Barak et al., present impossibility results for program obfuscation with respect to "virtual black box" intuition which states that anything that can be computed efficiently from an obfuscated version of program, can be computed efficiently with just an oracle access to the program. On the positive side, much of the interesting information about programs is hard to compute. In fact, Rice's theorem [29] states that any non-trivial property of partial recursive functions is undecidable. In [11], Collberg et al., use the result that different versions of precise static alias analysis are NP-hard [19] or even undecidable [28] to construct *opaque predicates* – whose values are known to the obfuscator but are hard to compute by static analysis. In [4], authors present a control-flow obfuscation technique that implants an instance of a hard combinatorial problem into a program.

Polymorphism Generators / Mutation Engines
A better technique to evade detection is adopted by polymorphic and metamorphic viruses. Polymorphic viruses exhibit a robust form of self-encryption wherein the encryption and decryption routines are themselves changed. Thus, the virus body is the same in all the versions but the encrypted versions appear different. In metamorphic viruses, the contents of the virus itself are changed rather than the encryption and decryption routines. In order to achieve this objective, metamorphic viruses make use of obfuscation techniques like code reordering, instruction substitution, variable renaming and garbage insertion.

For example, Chameleon, the first polymorphic virus, infects COM files in its directory. Its' signature changes every time it infects a new file. This makes it difficult for anti-virus scanners to detect them. Other polymorphic viruses are Bootache, CivilWar, Crusher, Dudley, Fly, Freddy, etc.

3.2 Static Analysis of Binaries

Static analysis techniques are used to determine important properties like *control flow safety* (i.e., programs jump to and execute valid code), *memory safety* (i.e., program only uses allocated memory) and *abstraction preservation* (i.e., programs use abstract data types only as far as their abstractions allow). Type systems are usually employed to enforce these properties. Although Java Virtual Machine Language (JVML) also has the ability to type-check low-level code, it has several drawbacks like semantic errors between the verifier and its' English language specification and also the difficulty in compiling high-level languages other than Java. In [23], authors present a Typed Assembly Language (TAL) which is a low-level statically-typed target language that supports memory safety even in the presence of advanced structures and optimizations; TALx86 is targeted at Intel architectures. In [34], authors present a technique to instrument well-typed programs with security checks and typing annotations so that the resulting programs obey the policies satisfied by security automata [30] and can be mechanically checked for safety. This technique allows enforcement of virtually all *safety policies* since it uses a security automata for policy specification. In

[24], authors present a technique for protecting privacy and integrity of data by extending Java language with statically-checked information flow annotations.

An interesting approach to establish safety of un-trusted programs is presented in [25]. In this approach referred to as *Proof Carrying Code*, the code producer provides a proof along with the program. The consumer checks the proof along with the program to ensure that his safety requirements are met with. Main objective of PCC was to be able to extend a system with a piece of software that was certified to obey certain memory safety properties. Though proof generation is quite complex and is often done by hand, proof verifier is relatively small and easy to implement. This technique has been applied to ensure safety of network packet filters that are downloaded into operating system kernel. An application to ensure resource usage and data abstraction in addition to memory safety for un-trusted mobile agents was also provided.

An interesting approach to detect variants of a known virus by performing static analysis on virus code and abstracting out its behaviour is presented in [5]. We now describe their architecture for detecting variants of a known virus [5]:

1. Generalize the virus code into a virus automata with uninterpreted symbols to represent data dependencies between variables,
2. Pattern-definitions are internal representations of abstraction patterns used as alphabet by the virus automata,
3. The executable loader transforms the executable into a collection of control flow graphs (CFG's) one for each procedure,
4. `Annotator` takes a CFG from the executable and the set of abstraction patterns and produces an annotated CFG as an abstract representation of a program procedure, and
5. `Detector` computes whether the virus (represented by the virus automata) appears in the abstract representation of the executable (represented as a collection of annotated CFG's) using tools for language containment and unification.

Note that for the above algorithm to work, abstraction patterns have to be provided (manually constructed) for each kind of transformation (code transportation, dead-code insertion etc). Once these are provided the rest of the algorithm is automatic. Thus, such an architecture works well for recognizing those variants of known viruses for which abstraction patterns are provided. In conclusion, this approach works better than signature based matching, but still has the drawback that it cannot recognize new viruses.

3.3 Semantics Based Detection

In [8], authors formalize the problem of determining whether a program exhibits a specified malicious behaviour and present an algorithm for handling a limited set of transformations. Malicious behaviour is described using templates, which are instruction sequences where variables and symbolic constants are used. They abstract away the names of specific registers and symbolic constants in the specification of the malicious behaviour, thus becoming insensitive to simple transformations such as register renaming. They formalized the notion of when an instruction sequence contains a behaviour specified by a template, to match the intuition that they should have the same effect on the memory upon execution. A program is said to satisfy a template iff the program contains an instruction sequence that contains a behaviour specified by the template. The algorithm to check whether a program satisfies a given template proceeds by finding, for each template node, a matching node in the program. Once two matching nodes are found, one needs to check whether the define-use relationships true between template nodes also hold true in corresponding program nodes. If all the nodes in the template have matching counterparts under these conditions, the algorithm is said to have found a program fragment that satisfies the template.

The above approach performs better than the static analysis based approach, since it incorporates semantics of instructions. Templates is a better form of abstraction than abstraction patterns. However, note that still templates have to be constructed by hand by looking at known malicious programs.

In [7] authors present a way of automatically generating malware specifications by comparing the execution behaviour of a known malware against the execution behaviours of a set of benign programs. Algorithm [7] for extracting malicious patterns (malspecs) proceeds as follows

1. **Collect execution traces.** In this step, traces are collected by passively monitoring the execution of each program.
2. **Construct dependence graphs.** In this step, they construct dependence graphs from the traces to include *def-use* dependence and *value-dependence*.
3. **Compute contrast subgraph.** In this step, they extract the minimal connected subgraphs of the malware dependence graphs which are not isomorphic to any subgraph of the benign dependence graph.

For using malspecs for malware detection, they consider the semantics aware malware detector described above and convert malspecs into templates that can then be used by the detection algorithm.

4 Protection Mechanisms

In the preceding section, we have studied the problem of detecting viruses. We have seen theoretical results indicating that the problem is undecidable in general. We have also seen some algorithms that can detect particular viruses. In this section, we will study the possibility of restricting the extent of damage due to an undetected virus.

Cohen [9] identifies three important properties of usable systems *sharing*, *transitivity of information flow* and *generality of interpreting information*. Sharing is necessary if we want others to use the information produced by us. For sharing information, there must be a well-defined path between the users. Once we have a copy of the information, it can be used in any way, particularly we can pass it on to others. Thus, information flow is transitive. By generality of information, we mean that information can be treated as data or program.

If there is no sharing, there can be no information flow across boundaries and hence, viruses cannot spread outside a partition. This is referred to as *isolationism.* Cohen [9] presents partition models that limit the flow of information. These models are based on well studied security properties like integrity and confidentiality/secrecy. These properties when enforced in tandem result in partitioning the set of users into closed subsets under transitivity of information flow. However precise implementations are computationally very hard (NP-complete). Isolationism is unacceptable in this inter-networked world. Without generality of interpreting information viruses cannot spread since infection requires altering the interpreted information. However, for building useful systems we must allow generality of interpreting information.

Adleman [1] studies the conditions under which a virus can be isolated from the computing environment. We present an intuitive account of his study.

Definition 5. *The infected set of a virus v, denoted I_v is defined as the set of those programs which result from v infecting some program.*

Definition 6. *A virus v is said to be absolutely isolable iff I_v is decidable.*

If a virus v is absolutely isolable, then we can detect as soon as a program gets infected by v and we can remove it. Thus, if a virus is absolutely isolable then we can neutralize it. For example, using this method we can neutralize viruses which always increase the size of a target program upon infection. Unfortunately, not all viruses are absolutely isolable as noted in the theorem below [1].

Theorem 3. *There exists a program v which always terminates such that*

1. *v is a malicious virus*
2. *I_v is semi-decidable*

To deal with the viruses of the kind described in the above theorem, Adleman introduces the notion of germ set of a virus.

Definition 7. *The germ set of a virus v, denoted G_v is defined as the set of those programs which behave (functionally) the same as a program infected by v.*

Germs of a virus are functionally the same as infected programs, but are syntactically different. They are not resulted from infections, but they can infect other programs.

Definition 8. *A virus v is said to be isolable within its germ set iff there exists a set of programs S such that:*

1. *$I_v \subseteq S \subseteq G_v$*
2. *S is decidable*

If a virus v is isolable within its germ set by a decidable set S, then not allowing programs in the set S to be written to storage or to be communicated will stop the virus from infecting. Moreover, isolating some uninfected germs is an added benefit. Unfortunately, not all viruses are isolable within their germ set.

Adleman [1] suggested the notion of a *quarantine* as another protection mechanism. In this method, one executes a program in a restricted environment and observe its behaviour under various circumstances. After one gains sufficient confidence in its genuineness, it can be introduced into the real environment. Several techniques are developed based on this idea. Application sandboxing and virtualization are some of the widely studied methods.

Virtualization Techniques and Sandboxing
The most important aspect of malware detection is being able to observe malware behaviour without being detected by malware. Malware detector typically runs in the same machine/environment as the malware. Thus, it is susceptible to subversion by malware. To overcome this limitation, several techniques use a virtual machine for malware detection. In this scenario, the host OS runs a Virtual Machine Monitor (VMM) that provides a virtual machine with guest OS on top of it. Another alternative is to have the VMM run directly on the hardware instead of the host OS. VMM provides strong isolation and protects the host from damage. Therefore, malware-detector can be used outside the virtual machine without the fear of being compromised by malware.

In [20], authors present a technique for detecting malware by running it in a virtual environment and observing its behavoiur from outside. In order to reconstruct the semantic view from outside, the guest OS data structures and functions are cast on VMM state (which is visible to the malware detector). Using this technique, the authors were able to view volatile state (e.g., list of running processes) and persistent state (e.g., files in a directory) of the virtual machine. This allowed them to detect rootkits like FU[3], NTRootkit[4] and Hacker Defender[5] that hide their own files and processes. As compared to this, the malware detection program running inside the virtual machine could not detect the rootkits. Another approach for malware detection via hardware virtualization is presented in [12]. In this approach, the authors present requirements for transparent malware analysis and present a malware detection mechanism that relies on hardware virtualization extensions. The advantage of using hardware virtualization extension is that it provides features like higher privilege, use of shadow page tables and privileged access to sensitive CPU registers. These techniques offer better transparency.

However, virtual machine techniques are not completely foolproof. Malware writers employ sophisticated techniques to detect whether the program is running inside a virtual machine. Whenever a virtual environment is detected, malware can modify its behaviour and go undetected. In [18] and [27], authors outline various anomalies that exist in virtualization techniques that can be exploited

[3] http://www.rootkit.com/board_project_fused.php?did=proj12
[4] http://www.megasecurity.org/Tools/Nt_rootkit_all.html
[5] http://hxdef.czweb.org

by malware to detect the presence of a virtual environment. Among the various anomalies presented are: discrepancies between interfaces of real and virtual hardware, inaccuracies of execution of some non-virtualized instructions, inaccuracy due to difficulty in modeling complex chipsets, discrepancies arising out of shared physical resources between guests and timing discrepancies. Even with hardware virtualization extensions, several timing anomalies can be easily detected. Thus, VM techniques do not offer complete transparency. In [21] an application of virtualization to launch a rootkit is presented. This rootkit installs a virtual machine under the current OS and then hoists the original OS into this virtual machine. These Virtual Machine Based Rootkits (VMBRs) can support other malicious programs in a separate OS that is isolated from the target system. VMBRs are difficult to detect because software running on the target system cannot access their state.

Sandboxes are security mechanisms to separate running programs by providing a restricted environment in which certain functions are prohibited (e.g., `chroot` utility in UNIX that allows one to set the root directory for a process). In this sense, sandboxes can be viewed as a specific example of virtualization (e.g., Norman Sandbox[6] that analyzes programs in secure, emulated environments). Sandboxes restrict the effects of a program within a specific boundary. For example, CWSandbox[7], uses API hooking technique to re-route the system calls through monitoring code. Thus, all relevant system calls made by un-trusted programs are monitored at run-time, their behaviour is analyzed and automated reports are generated. We have developed a similar sandboxing technique for Linux OS [32] wherein we monitor the system calls made by an un-trusted program and restrict its activities at run-time. We provide a guarded command based policy specification language to encode security policies. This language is as expressive as a Security Automata [30] and can express complex policies that depend on temporal aspects of system call trace. Whenever a system call is intercepted, it is allowed to go through only when it is deemed safe with respect to the policy at hand.

Sandboxes also suffer from limitations described above for the virtualization techniques.

5 A New Approach: Validating Behaviours of Programs

In this section, we present a promising future direction [33], for protection from computer viruses. Our idea is based on validating as opposed to verification in the context of compilers [3,22]. The validation approach is based on comparing whether one program is as good or as bad as the other. Such an approach lies on the notion of bisimulation due to David Park and Robin Milner; these concepts are very widely used in the context of process algebra and protocol verification. Recollect that viruses spread by infecting trusted programs. When an infected program is executed, the program in addition to performing its intended job

[6] http://www.norman.com/technology/norman_sandbox/
[7] http://www.cwsandbox.org/

(observable by the user) also results in damage (happens in the background without the users knowledge/consent). Based on this observation we divide the behaviour of a program into two parts: *external behaviour* (behaviour observable from outside) and *internal behaviour* (behaviour observable from inside the system with the help of a monitor). We further note that if the infection caused by the virus modifies the external behaviour then the user will suspect and remove the program. Therefore, it is reasonable to assume that infection modifies only the internal behaviour of programs. We therefore suggest that we must monitor the internal behaviour of trusted programs and validate them against their intended behaviour in an environment.

For the class of transactional reactive programs, we can define the external behaviour as the sequence of interactions that happen between the user and the program. For example, if we consider a vending machine as our system, one possible external behaviour is given by `place-coin ^ choose-item ^ receive-item`. We also observe that during execution of a program p with external behaviour t, the main process may spawn child processes internally (not necessarily observable to the user) for modularly achieving/computing the final result. Thus, the total (internal + external) behaviour can be denoted by a tree with processes, data operations etc denoted as nodes and directed edges. We can now define the internal behaviour of a program as the process tree generated during execution together with the associated system calls made by each process (vertex/node) in the tree. Based on this model of the internal behaviour, we can derive an algorithm for model-checking whether a program behaviour simulates a given behaviour. If the observed behaviour of a program simulates its' intended behaviour, then we say that the program is uninfected else we say that it is infected. We can use this method to detect if and when a trusted program is infected.

We have performed a lot of experiments and obtained encouraging results. We present some of the experiments and our observations in this section. We performed our experiments on a machine with Linux (Ubuntu distribution) OS. We monitored (unobtrusive) the sequence of system calls made (using *strace* tool), when the genuine text editor *nano* is used to edit a file. Note that system calls act as an interface between the application and the underlying hardware devices (can be thought of as services). We have also noted the % of time spent in various system calls, the number of processes created during execution, total running time, CPU and other resources used during this operation. We have collected similar information for the genuine *ssh* program starting from the time the service is started to the time the user logged in and completed the session.

We executed an infected version of *nano* program and collected the observable information during its execution. We then compared it with its intended behaviour and we easily concluded from the observations made that the version of *nano* we executed is infected. Moreover we were also able to identify the instructions added due to infection to the program.

Summary of differences in the system call profiles of the genuine *nano* vs the infected *nano*:

1. original program made 18 different system calls whereas the infected version made 48
2. infected program made network related system calls like `socket`, `connect`, etc. whereas the original program made none
3. infected program spawned 3 processes whereas the original program did not spawn any process
4. there is a huge difference in the number of `read` and `write` system calls
5. we observed a difference in the timing information provided by `strace` summary (when both the versions were run only for a few seconds). Original program spent around 88% on `execve` system call and 12% on `stat64` whereas the infected version spent 74.17% on `waitpid`, 10.98% on `write`, 6.28% on `read`, 4.27% on `execve` and negligible time on `stat64`. This indicates that the infected program spent more time waiting on children than in execution. This increased percentage of time spent on writing and reading by infected program indicates malfunction.

We executed an infected *ssh* program and collected the observable information during its execution. We then compared it with its intended behaviour and found that infected program modified the authentication module of the *ssh* program. Infected *ssh* would enable an attacker to successfully login to our host using a valid username with a magic-pass. In this case the infection has removed certain instructions from the program.

At a high level we can describe the expected behavior of ssh as follows

1. start sshd service
2. wait for a connection and accept a connection
3. authenticate the user
4. prepare and provide a console with appropriate environment
5. manage user interaction and logout
6. stop sshd

Summary of differences in behavior between genuine *ssh* and the infected *ssh*:

1. start sshd service
   - Genuine sshd uses the keys and config files from `/etc/ssh` whereas the infected ssh obtains these from a local installation directory
2. authenticate the user
   - Genuine sshd used *kerberos*, *crypto* utilities and *pam* modules which the infected ssh does not use
   - The infected ssh uses the config and sniff files (local/untrusted resources) which the genuine sshd does not use

To test the resilience of our approach to the simple syntactic transformations, that the virus writers are resorting to evade detection, we have compiled the virus responsible for infection under various levels of optimization. *gcc* compiler performs several simple syntactic transformations like loop unrolling, function inlining, register reassignment etc. We have executed the infected programs *nano* (similarly *ssh*) compiled under different optimization levels and collected the observable information during execution. What we observed was that barring very minor changes these programs produced the same traces of system calls. One difference we observed was the way in which contents of a file were buffered into and out of memory. Optimized program read in chunks of size 4096 whereas lower level of optimization resulted in reading chunks of a smaller size. These experiments demonstrate that the various obfuscators would have little impact on our approach and we will be able to catch infections.

To summarize, we have presented an approach in which we benchmark the intended behaviours of trusted programs in an execution environment and whenever we want to validate whether the installation of the trusted program in a similar environment is tampered, we collect the observable information during runtime and compare it with its intended behaviour. If there is a significant difference between the two, then we say that the program is infected. We have also showed that the method is resilient to obfuscation. We are conducting experiments to see the effect of polymorphic and metamorphic viruses on our approach. Note that the method we presented will also be very useful for validating the embedded systems because typically the software and the hardware configurations of an embedded system are very few.

In our study so far, the above approach seems to be very fruitful for checking un-tampering of devices of network communication and automobile software [31]. In fact, our study shows that our above approach does not need the constraints imposed in [31] or their related works on *Pioneer* and *swatt* protocols. The work is under progress and will be reported elsewhere.

6 Discussion

Malware has been a major cause of concern for information security. Today malicious programs are very widely spread and the losses incurred due to malware is very high (in some cases they result in financial loss while in some other cases they spoil the reputation of an organization). It is argued in [13] that the financial gain from criminal enterprise has lead to large investments of funds in developing tools and operational capabilities for the attackers on net transactions. Thus, it is imperative to develop mechanisms to defend ourselves from malware attacks. To develop sound defense mechanisms it then becomes necessary that we understand the fundamental capabilities of virus. In this paper, we have surveyed various efforts to formalize the notion of a virus/malware and characterize the damage due to them. We presented theoretical results based on the formalizations to show that detection of viruses in general is an undecidable problem. We presented a variety of detection mechanisms which are widely deployed to detect viruses in limited cases. We have also established the shortcomings of these detection mechanisms and discussed different techniques which the virus writers are resorting to evade detection. We then went on to survey some methods to limit the damages due to unidentified viruses. Unfortunately these methods end up being too complex to implement or force us towards unusable systems (systems with no sharing for example). It may be pointed out that the malware can be exploited for criminal activities.

We have also presented a promising new direction for protecting from malware. Our approach is based on the observation that malware infect trusted programs to include a subroutine for damage to be carried out in the background without the users consent. We proposed a framework based on runtime monitoring of trusted programs and the notion of bisimulation to validate their behaviour. We presented various experimental results to demonstrate the efficacy of the approach and showed its resilience to program obfuscation techniques.

Acknowledgement

The authors are grateful to L.M. Adleman, and F Cohen who founded the excellent theory of computer viruses. One of the authors (Harshit Shah) was supported under ITPAR II project from DST, Govt. of India.

References

1. Adleman, L.M.: An abstract theory of computer viruses. In: Goldwasser, S. (ed.) CRYPTO 1988. LNCS, vol. 403, pp. 354–374. Springer, Heidelberg (1990)
2. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S.P., Yang, K.: On the (im)possibility of obfuscating programs. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 1–18. Springer, Heidelberg (2001)
3. Bhattacharjee, A.K., Sen, G., Dhodapkar, S.D., Karunakar, K., Rajan, B., Shyamasundar, R.K.: A system for object code validation. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 152–169. Springer, Heidelberg (2000)
4. Chow, S., Gu, Y., Johnson, H., Zakharov, V.A.: An approach to the obfuscation of control-flow of sequential computer programs. In: Davida, G.I., Frankel, Y. (eds.) ISC 2001. LNCS, vol. 2200, pp. 144–155. Springer, Heidelberg (2001)
5. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: SSYM 2003: Proceedings of the 12th conference on USENIX Security Symposium, Berkeley, CA, USA, pp. 12–12. USENIX Association (2003)
6. Christodorescu, M., Jha, S.: Testing malware detectors. In: ISSTA 2004: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, pp. 34–44. ACM, New York (2004)
7. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: Crnkovic, I., Bertolino, A. (eds.) ESEC/SIGSOFT FSE, pp. 5–14. ACM, New York (2007)
8. Christodorescu, M., Jha, S., Seshia, S.A., Song, D.X., Bryant, R.E.: Semantics-aware malware detection. In: IEEE Symposium on Security and Privacy, pp. 32–46. IEEE Computer Society, Los Alamitos (2005)
9. Cohen, F.: Computer viruses: theory and experiments. Comput. Secur. 6(1), 22–35 (1987)
10. Cohen, F.: Computer Viruses. PhD thesis, University of Southern California (1986)
11. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland (July 1997), `http://www.cs.auckland.ac.nz/ collberg/Research/Publications/CollbergThomborsonLow97a/index.html`

12. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: CCS 2008: Proceedings of the 15th ACM conference on Computer and communications security, pp. 51–62. ACM, New York (2008)
13. Dittrich, D.: Malware to crimeware: How far they gone, and how do we catch up? Login, The USENIX Magazine 34(4), 35–44 (2009)
14. Filiol, E.: Computer Viruses from Theory to Applications. IRIS International Series. Springer, France (2005)
15. IBM X Force Threat Reports. IBM Internet Security Systems X-Force, trend and risk report (2008), `http://www-935.ibm.com/services/us/iss/xforce/trendreports/`
16. IBM X Force Threat Reports. IBM Internet Security Systems X-Force, mid-year trend and risk report (2009), `http://www-935.ibm.com/services/us/iss/xforce/trendreports/`
17. Forrest, S., Hofmeyr, S.A., Somayaji, A.: Computer immunology. CACM 40(10), 88–96 (1997)
18. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is not transparency: Vmm detection myths and realities. In: HOTOS 2007: Proceedings of the 11th USENIX workshop on Hot topics in operating systems, Berkeley, CA, USA, pp. 1–6. USENIX Association (2007)
19. Horwitz, S.: Precise flow-insensitive may-alias analysis is np-hard. ACM Trans. Program. Lang. Syst. 19(1), 1–6 (1997)
20. Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In: CCS 2007: Proceedings of the 14th ACM conference on Computer and communications security, pp. 128–138. ACM, New York (2007)
21. King, S.T., Chen, P.M., Wang, Y.-M., Verbowski, C., Wang, H.J., Lorch, J.R.: Subvirt: Implementing malware with virtual machines. In: SP 2006: Proceedings of the 2006 IEEE Symposium on Security and Privacy, Washington, DC, USA, pp. 314–327. IEEE Computer Society, Los Alamitos (2006)
22. Kundaji, R., Shyamasundar, R.: Refinement calculus: A basis for translation validation, debugging and certification. Theoretical Computer Science 354, 156–168 (2006)
23. Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., Zdancewic, S.: Talx86: A realistic typed assembly language. In: Second Workshop on Compiler Support for System Software, pp. 25–35 (1999)
24. Myers, A.C.: Jflow: practical mostly-static information flow control. In: POPL 1999: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 228–241. ACM, New York (1999)
25. Necula, G.C.: Proof-carrying code. In: POPL 1997: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 106–119. ACM, New York (1997)
26. Von Neumann, J.: Theory of Self-Reproducing Automata. University of Illinois Press, Champaign (1966)
27. Raffetseder, T., Krügel, C., Kirda, E.: Detecting system emulators. In: Garay, J.A., Lenstra, A.K., Mambo, M., Peralta, R. (eds.) ISC 2007. LNCS, vol. 4779, pp. 1–18. Springer, Heidelberg (2007)
28. Ramalingam, G.: The undecidability of aliasing. ACM Trans. Program. Lang. Syst. 16(5), 1467–1471 (1994)
29. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society 74(2), 358–366 (1953)

30. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (2000)
31. Seshadri, A., Luk, M., Perrig, A., van Doorn, L., Khosla, P.: Externally verifiable code execution. CACM 49(9), 45–49 (2006)
32. Shah, H.J., Shyamasundar, R.K.: On run-time enforcement of policies. In: Cervesato, I. (ed.) ASIAN 2007. LNCS, vol. 4846, pp. 268–281. Springer, Heidelberg (2007)
33. Shyamasundar, R., Shah, H., Kumar, N.N.: Checking malware behaviour via quarantining (abstract). In: Int. Conf. on Information Security and Digital Forensics, vol. City University of London, Full manuscript under submission process (September 2009)
34. Walker, D.: A type system for expressive security policies. In: POPL 2000: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 254–267. ACM, New York (2000)
35. `www.computereconomics.com` 2007 malware report: The economic impact of viruses, spyware, adware, botnets, and other malicious code, `http://www.computereconomics.com/page.cfm?name=Malware%20Report`