Intractable Problems in Malware Analysis and Practical Solutions

Ali Aydın Selçuk
Dept. of Computer Engineering
TOBB University of Economics and Technology
Ankara, Turkey

Fatih Orhan, Berker Batur
Comodo Security Solutions, Inc.
Clifton, NJ, USA

Abstract

Malware analysis is a challenging task in the theory as well as the practice of computer science. Many important problems in malware analysis have been shown to be undecidable. These problems include virus detection, detecting unpacking execution, matching malware samples against a set of given templates, and detecting trigger-based behavior. In this paper, we give a review of the undecidability results in malware analysis and discuss what can be done in practice.

1. Introduction

The number of malware programs encountered by security companies multiplies every year. Each of these programs needs to be analyzed by static and dynamic analysis tools. The task of running each program in a controlled environment and analyzing its behavior manually is a tedious and labor-intensive task. Therefore, there is a great need for automation of this process and for tools that will help with the analysis.

One of the most significant theoretical results in malware analysis is from the seminal works of Cohen on computer viruses [6, 7] where he showed that a program that detects all computer viruses precisely is impossible. Later, Chess and White [4] gave an example of a polymorphic virus that cannot be precisely detected by any program. Other results followed [2, 5, 21] which stated the impossibility of certain critical tasks in static and dynamic malware analysis.

In this paper, we give a brief survey of the major undecidability results found in the malware analysis literature. Then we give examples from the positive side showing what can be done on these undecidable problems in practice.

2. Malware Analysis and Undecidability

Since Cohen [6] gave the first formal treatment of computer viruses, many problems in malware analysis have been shown to be undecidable. Many of these results are based on the fact that precisely deciding whether a given program/input satisfies a certain post-condition, for an arbitrary post-condition, is undecidable. The proofs are based on two general techniques: Either they build a self-contradictory program assuming the existence of a decider for the given problem, similar to [6], or they give a reduction from a well-known undecidable problem, such as the Halting Problem, similar to [7]. In this section, we review some of the most significant undecidability results in the field.

2.1. Undecidability of the General Virus Detection Problem

The first result on the undecidability of the general virus detection problem is due to Cohen [6]. Using a well-known proof technique, he argued that:

"In order to determine that a given program '*P*' is a virus, it must be determined that *P* infects other programs. This is undecidable since *P* could invoke any proposed decision procedure '*D*' and infect other programs if and only if *D* determines that *P* is not a virus. We conclude that a program that precisely discerns a virus from any other program by examining its appearance is infeasible."

He gave the following piece of program "contradictory-virus" as an example that cannot be detected by a virus detector *D* in a correct way:

```
program contradictory-virus :=
{....
  main-program :=
  {if ~D(contradictory-virus) then
    {infect-executable;
    if trigger-pulled then
      do-damage;
    }
    goto next;
  }
}
```

As Cohen [6] observed, "… if the decision procedure *D* determines *CV* to be a virus, *CV* will not infect other programs, and thus will not act as a virus. If *D* determines that *CV* is not a virus, *CV* will infect other programs, and thus be a virus. Therefore, the hypothetical decision procedure *D* is self-contradictory, and precise determination of a virus by its appearance is undecidable." A minor flaw in this argument was observed by Steinparz [25], who noted that this argument only shows the impossibility of a virus detector which is not a virus itself. Otherwise, if *D* is a virus itself, it can return "true" on contradictory-virus and be correct.

A more formal proof was again given by Cohen himself [7] by a reduction from the Halting Problem. He showed that the existence of a precise virus detector would imply a decider for the Halting Problem and hence is not possible.

Furthermore, Cohen [8] observed that whether a sequence is a virus or not depends on the environment in which it is run. Thus any given sequence is or is not a virus as a function of the environment in which it is placed.

2.2. Existence of an Undetectable Virus

As summarized above, Cohen [6, 7] showed the impossibility of a virus detector that detects all viruses precisely. Chess and White [4] extended this result by showing that there are viruses, in theory, with no error-free detectors. They explained, "That is, not only can we not write a program that detects all viruses known and unknown with no false positives, but in addition there are some viruses for which, even when we have a sample of the virus in hand and have analyzed it completely, we cannot write a program that detects just *that* particular virus with no false positives."

The result of Chess and White is based on an extension of the contradiction argument in Cohen's first paper [6]: Consider a polymorphic virus *W* that is able to modify its code. This virus modifies its spreading condition such that if some particular subroutine in it returns "false" on *W* itself, it spreads. Furthermore, this subroutine is subject to change as a part of *W*'s polymorphism. Now, if some detector code *C* were to detect *W*, there is at least one instance of this polymorphic virus, where the subroutine is replaced by *C*, that cannot be detected by *C*: Just like Cohen's argument, detection by *C* would result in the virus' not spreading, and hence would imply a false positive.

To illustrate their point, they gave an example pseudocode of such a virus *W*, one instance of which is *r*:

```
if subroutine_one(r) then exit, else
{
  replace the text ofsubroutine_one
with a random program;
  spread;
  exit;
}
subroutine_one:
  return false;
```

They noted that for any algorithm *C* that detects *W*, there is a program *s* for which *C* does not return the correct result:

```
if subroutine_one(s) then exit, else
{
  replace the text of subroutine_one
with a random program;
  spread;
  exit;
}
subroutine_one:
  return C(argument);
```

If *C*(*s*) returns true, then *s* will just exit, but if *C*(*s*) returns false, then s is an instance of the virus W.

The same argument shows the non-existence of a detector for *W* under a looser notion of detection as well: Say a program "detects" a virus *V* if it (i) returns "true" on every program infected with *V*, (ii) returns "false" on every program not infected with any virus, (iii) may return "true" or "false" on a program that is infected with some virus other than *V*. The impossibility argument above applies to this looser notion of detection verbatim. Hence, Chess and White [4] concluded that there exists, in theory, some virus that cannot be detected precisely by any virus detector even under this looser notion of detection.

2.3. Semantic-Aware Malware Detection

A malware detector based on a pattern matching approach is fundamentally limited against obfuscation techniques used by hackers. The goal of malware obfuscation is to morph or modify the malware to evade detection. A piece of malware can modify itself by, for example, encrypting its payload, and then later decrypting it during execution. A polymorphic virus tries to obfuscate its decryption code using several transformations, such as code transposition, nop insertion, and register reassignment. Metamorphic viruses, on the other hand, try to evade detection through obfuscating the entire code. When they replicate, these viruses change their code by techniques such as substitution of equivalent instruction sequences, code transposition, register reassignment, and change of conditional jumps. The fundamental limitation of the pattern-matching approach for malware detection is that it is mainly syntactic and does not consider the semantics of the program flow and the instructions.

Christodorescu et al. [5] studied a method to overcome this limitation by incorporating instruction semantics to detect malicious code traits. In their framework, malicious behavior is defined by hand-constructed "templates". A template *T* is defined as a 3-tuple (I_T, V_T, C_T): I_T is a sequence of instructions,

and V_T is the set of variables and C_T is the set of symbolic constants that appear in I_T. An "execution context" of a template $T = (I_T, V_T, C_T)$ is an assignment of values to the symbolic constants of the set C_T.

For a given program P, they say that P satisfies a template T (denoted by $P \Rightarrow T$) if P contains an instruction sequence I such that I contains a behavior specified by T. The problem of deciding whether a given piece of code contains such a template behavior (i.e., $P \Rightarrow T$) is modeled as the "Template Matching Problem".

The Template Matching Problem turns out to be undecidable. Christodorescu et al. [5] gave a reduction from the Halting Problem to the Template Matching Problem, and stated that a precise solution for the general Template Matching Problem is impossible.

2.4. Automatic Unpacking for Malware Detection

An obfuscation mechanism that is much used by modern malware is to hide the malicious portion of the payload as data at compile time, and then transform it into an executable at run time, a behavior known as "unpack and execute". The unpack transformation can be something simple, such as an XOR by a block of random-looking data, or something more complex, such as decryption by a cipher like AES.

Royal et al. [21] worked on detecting such polymorphic viruses by focusing on the result of the unpack operation. The idea is to compare the executable code during the run time with that before the run time. When a change is detected, it is written out for further analysis.

The code and the data sections of a program are formally modeled as a Turing machine M and its input w. Then the unpack detection problem becomes whether w contains another program in it that will be emulated by M during computation. This problem can be formulated as the following formal language:

$\text{UnpackEx}_{\text{TM}} = \{\langle M, w\rangle$: M is a UTM, and M simulates a Turing machine on its tape in its computation on $w\}$

Royal et al. [21] gave a theorem which stated that the $\text{UnpackEx}_{\text{TM}}$ language is undecidable. They proved this result by a reduction from the Halting Problem. Their proof can be summarized as follows:

A mapping reduction $\text{HALT}_{\text{TM}} \leq \text{UnpackEx}_{\text{TM}}$ will be given to prove that $\text{UnpackEx}_{\text{TM}}$ is undecidable.

Let f be a function that takes $\langle M,w\rangle$ as input and computes $\langle M',w'\rangle$ as output where $\langle M,w\rangle \in \text{HALT}_{\text{TM}}$ if and only if $\langle M',w'\rangle \in \text{UnpackEx}_{\text{TM}}$. The Turing machine F given below computes f:

F = "On input $\langle M,w\rangle$, a valid encoding of a Turing machine M and an input string w:

1. Construct a Turing machine T:
   T="On input x:
      1. Ignore x and halt."
2. Construct the following UTM M' from M:
   M' is the same as M, except:
      for all $q \in Q$, $\gamma \in \Gamma$
         if $\delta(q, \gamma)$ goes to a halting state then
            Replace this transition with a transition that begins simulating T on the input tape. I.e., change the transition to $\delta(q, \gamma) = (q_{start,T}, —, —)$.
3. Output $\langle M', \langle T, w\rangle\rangle$."

The output of the mapping F, a UTM M', will execute a Turing machine T in those cases where M will halt on w. A decider for $\text{UnpackEx}_{\text{TM}}$ could decide whether M' will execute T and so decide HALT_{TM}. But HALT_{TM} does not have a decider. Hence, a decider for $\text{UnpackEx}_{\text{TM}}$ cannot exist. Therefore, $\text{UnpackEx}_{\text{TM}}$ is undecidable

Hence, it turns out that determining precisely whether a given program contains some unpack-execute behavior in it is impossible.

2.5. Automatically Identifying Trigger-Based Behavior

A common feature found in modern malware is to contain some hidden malicious behavior that is activated only when triggered; such behavior is called trigger-based behavior. Various conditions are used for triggering, such as date and time, some system event, or a command received over the network.

Brumley et al. [2] studied how to automatically detect and analyze trigger-based behavior in malware. Their approach employs mixed symbolic and concrete execution to automatically explore different code paths. When a path is explored, a formula is constructed representing the condition that would trigger execution down the path. Then a solver is employed to see whether the condition can be true, and if so, what trigger value would satisfy it.

Like many other problems in malware analysis, an exact, automatic identification of trigger-based behavior turns out to be undecidable by a reduction from the halting problem. Brumley et al. [2] observed that "Identifying trigger-based behaviors in malware is an extremely challenging task. Attackers are free to make code arbitrarily hard to analyze. This follows from the fact that, at a high level, deciding whether a piece of code contains trigger-based behavior is undecidable, e.g., the trigger condition could be anything that halts the program. Thus, a tool that uncovers all trigger-based behavior all the time reduces to the halting problem."

2.6. Self-Modifying Code and Formal Grammars

Filiol [11] studied the complexity of detecting self-modifying code (i.e., polymorphic and metamorphic viruses) using formal grammars. He worked on the formalization of metamorphism by means of formals languages and grammars. He showed how code mutation techniques can be modelled by formal grammars, and how their detection can be converted to the problem of deciding a language.

He modelled a self-modifying program as a grammar G_2 whose language consists of grammars that are produced from a starting grammar G_1 according to the derivation rules specified by G_2. This definition was used to describe the fact that the virus kernel changes from one metamorphic form to the other: It is both the virus code and the set of mutation rules that change. (This view of metamorphic viruses resembles two-level 2VW grammars, as Filiol pointed out.)

In this context, detecting whether a given program is a form of a given metamorphic virus is an instance of the language decision problem for Class 0 (free) grammars. Filiol showed that this problem can be reduced from the Halting Problem and hence is undecidable.

2.7. NP-Complete Problems

Although the general cases of the aforementioned problems are undecidable, it turns out that it is possible to obtain their decidable versions by assuming some bound on the time or memory available to the malware.

Spinellis [24] showed that a length-bounded version of Cohen's problem is decidable and NP-complete, by a reduction from the Boolean Satisfiability Problem (SAT).

Borello and Mé [1] showed that detecting whether a given program *P* is a metamorphic variant of another given program *Q* is decidable and NP-complete.

On a related venue, Fogla and Lee [12] showed that detecting a polymorphic blending attack, which can blend in with normal traffic and can evade an anomaly-based IDS, is also NP-complete, by a reduction from the 3-SAT problem.

Song et al. [23] studied the strengths and weaknesses of polymorphic shellcode. They developed metrics to measure the capabilities of polymorphic engines. In the end, they concluded that polymorphic behavior in general is too greatly varied to be modelled and detected effectively.

Bueno et al. [3] showed that the space- and time-bounded versions of the unpacking problem are decidable, and the time-bounded version is NP-complete.

Of course, a problem's being NP-complete is hardly good news. It is usually interpreted as that no efficient solution exists for the worst case of that problem. However, efficient solutions may exist for the average case, or it can be possible to obtain reasonably good solutions by heuristics or approximation algorithms.

3. Practical Solutions

Despite the negative theoretical results on undecidability of some fundamental questions in malware analysis, practical tools have been in action since the very early days of computer viruses. By tolerating some degree of inaccuracy (i.e., tolerating some degree of false positives or negatives, or allowing inconclusive results), it is possible to build algorithms that are very effective in practice. In this section, we summarize some of the tools developed for the problems reviewed in Section 2.

3.1. Detecting Malware by Template Matching

Despite the fact that the general Template Matching Problem is undecidable, it is possible to detect malware using template matching algorithms that are mostly accurate. Christodorescu et al. [5] developed a toolkit for that purpose. The toolkit works in two phases: First, the binary program to be analyzed is disassembled, a control graph is constructed, one per program function, and an intermediate representation (IR) is generated. The IR is further processed and put into an architecture- and platform-independent form. In the second phase, the IR is compared against a given set of malware templates. Each comparison either returns "yes" or "don't know". Suggested malware templates for comparison include procedures such as a decryption loop or mass mail sending.

Christodorescu et al. [5] tested their tool on a real-world malware sample consisting of seven variants of Netsky (B, C, D, O, P, T, and W), seven variants of Bagle (I, J, N, O, P, R, and Y), and seven variants of Sober (A, C, D, E, F, G, and I), all being email worms with many diverse forms found in the wild. The authors tested the malware against templates capturing the decryption loop and mass mailing functionalities. The tool detected all Netsky and Bagle variants with 100% success. The Sober worm was not detected due to a limitation in the prototype implementation, related to matching calls into the Microsoft Visual Basic runtime library. Nevertheless, their test demonstrated the success of their template matching algorithm on diverse forms of malware.

The tool was tested on a benign sample as well in order to test its false positive rates. 97.78% of the programs in the given sample were detected as

benign after successful disassembly, while 2.22% could not be disassembled.

Kwon et al. [16] developed a technique called BinGraph, in which they leveraged the semantics in API call sequences of different malware families as templates and detected metamorphic malware samples using signatures and template matching. They extracted signatures from subgraphs of API calls and used them to represent semantic behaviors of metamorphic malware. In first phase, using the Import Address Table (IAT) from executable they constructed initial behavior graph with edges representing API call sequences. Followed by subgraph extraction and graph abstraction, they stored the abstracted semantic graphs in a 128x128 adjacency matrix. At the final step of sematic signature extraction, they applied graph mining techniques using a greedy strategy to select candidate signatures.

Kwon et al. [16] used 166 malware programs (randomly selected 20% of their malware collection) and generated 32 semantic signatures representing this set. They compared these signatures against the set of remaining 661 (unseen) malicious and 1,202 safe binaries. They achieved a 98.18% detection rate on the malicious sample and detected 649 malware programs, with no positive matches on the benign sample.

Luh et al. [17] used sentiment analysis technique by collecting kernel level events in order to model malicious and benign behaviors. Timestamped process, thread, image load, file, registry and Network event logs were collected from the Windows kernel and runaway entries were eliminated. Log-likelihood ratio scores for each pre-processed bi-gram event traces were calculated and used in compilation of malicious and safe semantic dictionaries. At the final step, score normalization and adjustment was applied to calculated values using whether monitored trace is a part of some malicious event sequence or not.

Safe event logs are collected from standard Windows users having more than 80 OS session and over 500 processes in each. Different types of malware samples like MyDoom, Zeus, Koobface, etc. were used to infect machines in controlled environments and their respective event traces were used in bi-gram extraction from malicious behaviors. Luh et al. [17] achieved 98.2% accuracy for malicious behavior detection with low false positive rates. Moreover, it is reported that threshold optimization on determined confidence values could increase the performance of related classification technique up to 100.0% for this particular test set used in conducted experiment.

3.2. Detecting Unpack-Execute Behavior

Although the general problem of unpack-execute behavior is undecidable, Royal et al. [21] gave an algorithm for a bounded version of this problem. Let n denote the number of instructions of a given program P to execute before it halts. The program ExtractUnpackedCode(P,n) works in two phases:

- Phase 1: Static Analysis. Program P is disassembled to identify code and data. Blocks of code that are separated by non-instruction data are partitioned into sequences of instructions. These sequences form the set I, which will be queried repeatedly in the next phase to detect if P is executing unpacked code.
- Phase 2: Dynamic Analysis. Program P is executed one instruction at a time. The current instruction sequence is captured by in-memory disassembly starting at the current value of the program counter until some non-instruction data is encountered. The current instruction sequence is compared against each instruction sequence in the set I. If the current sequence is not a subsequence of any instruction sequence in I, then it did not exist in P.

Royal et al. [21] developed this algorithm into a practical tool for MS Windows systems, called PolyUnpack. They tested the tool on the OARC malware suspect repository and compared its performance with that of the Portable Executable Identifier (PEiD), a popular reverse-engineering tool which uses a specific set of signatures to detect unpack-execute behavior [20]. PolyUnpack performed very well and was able to identify many samples with unpack-execute behavior which PEiD was unable to detect.

Another technique developed by Korczynski [15] helped reconstructing packed binaries with self-modifying code blocks up to some accuracy, easing further analysis on them by malware analysts. Both static and dynamic analysis were used in related work and binaries having self-modifying property and IAT destruction are focused for binary reconstruction. Tough it is not directly related with malware detection, Korczynski's [15] technique helps further methods and algorithms to be developed for bounded version of undecidable unpack-execute behavior.

There are mainly two phases in general unpacking technique of developed by Korczynski [15]:

- First phase: Self-modifying code detection, where dynamically loaded modules are being tracked and several snapshots are being captured including exported functions, in-direct references and memory writes
- Second phase: Recovering import address table using dynamically loaded function branches using a heuristic filtering method

Korczynski [15] carried out two experiments to verify that the developed technique improves self-modifying code discovery and IAT reconstruction compared with the clean memory dumps. 117 malware samples belonging to 9 different malware families were used in the first experiment. Initially, 35% of the malware had no IAT in the clean memory dumps. This sample was analyzed and successful IAT reconstructions up to some degree were performed on 66% of them. In the second experiment, a simple so-called hello world application was developed and packed with 21 different packers. IAT reconstruction was partially done on 61% of the packed binaries.

Kim et al. [14] studied distinctive properties of obfuscation techniques applied on safe and malicious samples and developed a technique named as DynODet, first to detect dynamic obfuscation and then use features of present obfuscation to classify unknown sample as either clean of malware.

DynODet uses both static and dynamic analysis results while determining whether any obfuscation is present or not. Static analysis leverages finding the expected path of the program before its execution and using this discovery to compare seen actual paths in dynamic analysis. Six different obfuscation techniques were tracked and used by Kim et al. [14]: self-modification, section mislabel obfuscation, dynamically generated code, unconditional to conditional branch obfuscation, exception-based obfuscation, and overlapping code sequences.

Kim et al. [14] used 6,192 safe Windows programs in their experiment and using distinctive obfuscation features they reduced the false positive rate nearly 70% in terms of one or more false obfuscation detection on these safe samples. Totally 100,208 malware samples used for malicious data set and using distinctive obfuscation features, one or more of the tracked obfuscation techniques were detected on 32.74% of the malware set with a detection rate of 2.5% on the clean set. Findings from the DynODet research showed its usefulness to develop a new detection technique using the presence of distinctive obfuscation techniques in unknown samples.

3.3. Detecting Trigger-Based Behavior

Detection of trigger-based behavior by manual analysis is a virtually impossible task due to the intensive labor required. On the other hand, a precise automatic analysis is not possible either; as explained in Section 2.5, the general problem of automatic identification of trigger-based behavior is undecidable. Nevertheless, a great deal of help can be obtained from automatic analysis to alleviate the burden of manual analysis. Brumley et al. [2] designed a tool for this task. Their approach consisted of several phases: First, the different types of triggers of interest are specified. Then, different code paths are explored using mixed symbolic and concrete execution. For a path explored by this process, a formula is constructed representing the condition that would trigger execution down the path. Then a solver is employed to see whether the condition can be true, and if so, what trigger value would satisfy it.

Brumley et al. [2] developed this approach into a program called MineSweeper. They tested MineSweeper on real-world malware containing trigger-based behavior. On every case, MineSweeper was able to detect the trigger condition and the trigger-based behavior. The analysis time varied depending on the complexity of the malware, from 2 to 28 minutes. In general, MineSweeper is not guaranteed to detect every piece of malware containing trigger-based behavior, but it can definitely be used as a tool of great assistance over the impractical alternative of manual analysis.

Kang et al. [13] researched specifically botnet malware samples and developed a trigger-based behavior detection technique called BotMelt. In an approach different from [2], they used dynamic symbolic execution (also known as symbol propagation) where network packets were used to mark the data flow. This technique allows revealing outbound trigger conditions different from locally-decided trigger conditions. Authors evaluated their technique in terms of validness of executed codes, malicious activity detection rate and behavior detection ratio among all possible behavior branches. four different botnet malware samples, HwDoor, Bisonal, KeyBoy, and Plez were used in evaluation experiments, and BotMelt yielded successful results in all evaluation criteria.

Papp et al. [18] developed a framework to detect trigger-based malicious behavior in source code level using a semi-automated approach. First, automated source code instrumentation technique is being applied to discover function calls and variables that interact with running environment. Then fresh symbolic values are given via replaced dummy functions and mixed concrete and symbolic execution tool is used to generate potentially malicious test cases that could trigger hidden malicious behavior of malware. At the final step, execution traces for each generated test case input were generated and passed to analysts for manual analysis in order to classify them as either malicious or benign.

In the experiment phase, Papp et al. [18] collected five real-world malware samples, all written in C and including some kind of trigger-based behavior in implementation. The developed framework succeeded in revealing trigger-based behaviors in three of them. Moreover, unsuccessful cases of hidden behavior detection were reported as failed due to limitations of the open-source tools used.

3.4. Malware Protection by Whitelisting and Default Deny Approach

Given that some fundamental problems in malware analysis and detection are undecidable, an alternative solution applied by practical security tools (e.g., Comodo's Endpoint Security [9]) is the default deny approach to protect users from malware infections: Rather than blocking only blacklisted malware applications and allowing all other safe and unknown applications, only whitelisted applications [22] are permitted to run on a host's real operating system (OS). Samples blacklisted as malware are blocked by default. And, unknown programs are permitted to run in some virtualized environments such as containments, sandboxes, etc. Creating these shadow file systems, registries, and communication ports helps blocking most damages caused by malicious application, and correctly defining a program for virus detection [10].

Although the default deny approach provides a higher level of protection compared to the default allow approach, one of the current problems using these virtualized environments is encountered in application usability. Purdila and Terzis [19] developed a dynamic browser containment environment to protect users from web-based malware, where they intercept system service requests of processes and limit browser's access to critical system resources to prevent malware damages. Although they managed to provide the desired protection, their proposed technique introduced some overhead; an 11.8% increase in latency and a 13.4% decrease in throughput.

Song et al. [23] studied shellcode decoding routines of polymorphic malware and reported the intractability of modelling this kind of characteristic behavior using known methods. They concluded that modelling and whitelisting safe behaviors and content would be a more promising and viable way to pursue.

4. Conclusion

Malware detection has been a major problem since the early days of computing. Theoretical results have been given on the inexistence of perfect detectors on various problems. Nevertheless, there is a great deal of work to be done using less-than-perfect tools. Bounded versions of the undecidable malware detection problems are in fact decidable. By assuming certain bounds on the time or memory available to the malware, it should be possible to develop detectors that work quite accurately in practice.

5. References

[1] Jean-Marie Borello, Ludovic Mé, "Code obfuscation techniques for metamorphic viruses", *Journal in Computer Virology*, 4(3):211–220, 2008.

[2] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, Heng Yin, "Automatically identifying trigger-based behavior in malware" In *Botnet Detection*, pp. 65–88, Springer, 2008.

[3] Denis Bueno, Kevin J. Compton, Karem A. Sakallah, Michael Bailey, "Detecting Traditional Packers, Decisively", *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2013)*, 2013.

[4] David M. Chess, Steve R. White, "An undetectable computer virus", *Proceedings of Virus Bulletin Conference*, vol. 5, 2000.

[5] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, Randal E. Bryant, "Semantics-aware malware detection", *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)*, 2015.

[6] Fred Cohen, "Computer viruses: theory and experiments", *Computers and Security*, 6(1):22-35, 1987.

[7] Fred Cohen, "Computational aspects of computer viruses", *Computers and Security*, 8(4):325-344, 1989.

[8] Fred Cohen, "Computer Viruses", Doctoral dissertation, University of Southern California, 1986.

[9] Comodo, "End Point Security and the Case For Auto Sandboxing", White Paper. https://containment.comodo.com/resources/white-papers/White-Paper-End-Point-Security-And-The-Case-For-Auto-Sandboxing.pdf

[10] David Evans, "On the Impossibiltiy of Virus Detection", 2017.

[11] Eric Filiol, "Metamorphism, Formal Grammars and Undecidable Code Mutation", *International Journal of Computer Science*, vol. 2, no. 9, pp. 70-75, 2007

[12] Prahlad Fogla, Wenke Lee, "Evading network anomaly detection systems: Formal reasoning and practical techniques.", *In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pp. 59–68, 2006

[13] Byeongho Kang, Jisu Yang, Jaehyun So, Czang Yeob Kim, "Detecting Trigger-based Behaviors in Botnet Malware", *Proceedings of the 2015 Conference on research in adaptive and convergent systems*, pp. 274-279, 2015

[14] Danny Kim, Amir Majlesi-Kupaei, Julien Roy, Kapil Anand, Khaled Elwazeer, Daniel Buettner, Rajeev Barua, "DynODet: Detecting Dynamic Obfuscation in Malware", *14th International Conference on Detection of Intrusion and Malware, and Vulnerability Assesment (DIMVA)*, 2017

[15] David Korczynski, "RePEConstruct: Reconstructing binaries with self-modifiying code and import address table descturction", In *MALWARE*, pp. 31-38 IEEE Computer Society, 2016.

[16] Jonghoon Kwon, Heejo Lee, "BinGraph: Discovering Mutant Malware using Hierarchical Semantic Signatures", In *MALWARE*, pp. 104-111. IEEE Computer Society, 2012.

[17] Robert Luh, Sebastian Schrittwieser, Stefan Marschalek, "LLR-based sentiment analysis for kernel event sequences", *31st IEEE International Conference on Advanced Information Networking and Applications (AINA)*, 2017.

[18] Dorottya Papp, Levente Buttyán, Zhendong Ma, "Towards Semiautomated Detection of Trigger-based Behavior for Software Security Assurance", In *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES)*, no. 14, 2017.

[19] Octavian Purdila, Andreas Terzis, "A Dynamic Browser Containment Environment for Countering Web-based Malware", *Proceedings of the 8th RoEdunet International Conference*, 2009.

[20] Jibz, Qwerton, snaker, xineohP. *PEiD*. peid.has.it, 2005.

[21] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, Wenke Lee, "PolyUnpack: Automating the hidden-code extraction of unpack-executing malware", *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006.

[22] Adam Sedgewick, Murugiah Souppaya, Karen Scarfone, "Guide to Application Whitelisting", *NIST Special Publiction (800-167)*, 2015.

[23] Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis and Salvatore J. Stolfo, "On the Infeasibility of Modeling Polymorphic Shellcode"" *In Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2007

[24] Diomidis Spinellis, "Reliable identification of bounded-length viruses is NP-complete", *IEEE Transactions on Information Theory*, 49(1):280–284, 2003.

[25] Franz X. Steinparz, "A comment on Cohen's theorem about undecidability of viral detection", *Alive*, vol. 1, 1991.

