Undecidable Problems in Malware Analysis

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

Fatih Orhan, Berker Batur
Comodo Security Solutions, Inc.
Clifton, NJ, USA
{fatih.orhan,berker.batur@comodo.com }

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 will give a review of the undecidability results in malware analysis and discuss what can be done in practice.**

Keywords- Computer viruses, malware analysis, virus detection, undecidability.

I. 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 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 [Coh87, Coh89] 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 [CJS+05, RHD+06, BHL+08] 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.

II. 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.

A. 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 [11], 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.

B. Existence of an Undetectable Virus

As summarized above, Cohen [Coh87, Coh89] 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.

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.

C. 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". The problem of deciding whether a given piece of code contains such a template behavior 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.

D. Automatic Unpacking for Malware Detection

The 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. [9] 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:

$\text{UNPACKEX}_{\text{TM}}$ = {<*M*, *w*>: M is a UTM, and *M* simulates a Turing machine on its tape in its computation on *w*}

Royal et al. [9] 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. Hence, it turns out that determining precisely whether a given program contains some unpack-execute behavior in it is impossible.

E. 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."

F. 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 [10] showed that a length-bounded version of Cohen's problem is decidable and NP-complete. Borello and Mé [BM08] showed that detecting whether a given program *P* is a metamorphic variant of another given program *Q* is decidable and NP-complete. 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 not exactly 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.

III. 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.

A. 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.

B. Detecting Unpack-Execute Behavior

Although the general problem of unpack-execute behavior is undecidable, Royal et al. [9] 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. [9] 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 [8]. PolyUnpack performed very well and was able to identify many samples with unpack-execute behavior which PEiD was unable to detect.

C. 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.

IV. 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.

Acknowledgment

We would like to thank Hatice Sakarya for her help during the preparation of this paper.

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

[9] 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.

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

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