Reliable Identification of Bounded-length Viruses is NP-complete

Diomidis Spinellis, *Member, IEEE*

***Abstract*—A virus is a program that replicates itself by copying its code into other files. A common virus protection mechanism involves scanning files to detect code patterns of known viruses. We prove that the problem of reliably identifying a bounded-length mutating virus is NP-complete by showing that a virus detector for a certain virus strain can be used to solve the satisfiability problem. The implication of this result is that virus identification methods will be facing increasing strain as virus mutation and hosting strategies mature, and that different protection methods should be developed and employed.** [1] [2]

***Index Terms*—buffer-overflow, complexity, detection, identification, mutation, NP-complete, security, virus**

I. Introduction

ONE often-used defence against computer viruses is the execution of an *anti-virus* program that detects and cleans programs that appear to be infected. Virus writers respond to this defence by trying to thwart anti-virus software through targeted attacks, mutations, or social engineering. Mutating viruses are a particularly insidious threat, because detection algorithms need to be constantly updated and to spend increasing processing time to identify new mutation types. The question of whether complexity theory is on the side of virus writers or the protection vendors could have important practical implications. In this paper we will prove that there exist realistic viruses whose reliable detection is of NP-complete complexity [1] and that therefore the general problem of reliable bounded-length virus identification is NP-complete.

II. Viral Software

Intentionally created malicious software [2]—often termed *malware*—is typically classified into Trojan horses, viruses, and worms [3]. A Trojan horse is a program that exploits the rights of its user to perform an action its user does not intend, a virus is a Trojan horse that replicates itself by copying its code into other program files [4], while a worm is an independently-running program that replicates through a network exploiting security weaknesses to invade other computers.

D. Spinellis is an Assistant Professor in the Department of Management Science and Technology at the Athens University of Economics and Business, Athens, Greece. E-mail: dds@aueb.gr .

[1] *IEEE Transactions on Information Theory*, 49(1):280–284, January 2003.

[2] This is a machine-readable rendering of a working paper draft that led to a publication. The publication should always be cited in preference to this draft using the reference in the previous footnote. This material is presented to ensure timely dissemination of scholarly and technical work.

A number of virus prevention and detection methods have been proposed and are commonly implemented [5], [6]. Reference [7] contains an annotated bibliography of malware analysis and detection papers. Prevention methods involve limiting the flow of information between programs through the use of appropriate hardware and software protection domains, coupled with self-defence mechanisms, instrumentation, and fault-tolerance. Since the above methods will typically interfere with many legitimate operations (such as the installation of new software or the correction of an existing version) they need to be coordinated through carefully designed and executed security procedures. Unfortunately current practice in system administration often renders these methods useless. A large percentage of users typically administer their personal workstations on their own, in most cases exercising the full rights of the system administrator, without sufficient training and diligence.

Therefore, as a secondary line of defence, detection measures are often employed to locate virus instances and infections. Two often used detection measures involve either the comparison of the system's programs against known-good versions (typically condensed in the form of a checksum or a cryptographically secure signature [8]) or the comparison of files against patterns of known viruses. Since the first method depends on a known-clean system and can not be used to check software of unknown origin, the second, scanning, method is the one most commonly employed. A number of software vendors provide virus-scanning software that can search new and existing system files for patterns of all known viruses. The vendors regularly distribute updated versions of the virus patterns to keep the virus detection process up-to-date.

Virus writers however, have developed a series of countermeasures. Even early academic examples of viral code were cleverly engineered to hinder the detection of the virus [9]. Since the actual task of writing a virus is relatively simple [10], [11] modern virus code focuses on employing platform independence, stealth, effective replication, and detection countermeasures. Three pattern-matching detection countermeasures typically employed are the *encryption* of the virus body with a variable cryptographic key, the *polymorphic* generation of the decryption routine using equivalent code instructions, and, more recently, the *metamorphic* generation of the whole virus body through the addition, removal, permutation, and substitution of code sequences. Viruses that employ these techniques, such as W32/Simile [12] can be very difficult to identify. In the following section we establish that reliably detecting instances of such viruses is a problem of NP-complete complexity.

III. Identification Complexity

A virus is formally defined [13] by reference to a Turing Machine [14]

$$M : (\quad S_M, I_M, O_M : S_M \times I_M \to I_M, \\ N_M : S_M \times I_M \to S_M, D_M : S_M \times I_M \to d) \tag{1}$$

with a given set of states S_M, set of input symbols I_M, and maps (O_M, N_M, D_M) that, based on its current state $s \in S_M$ and input symbol $i \in I_M$ coming from a semi-infinite tape, determine: the output symbol $o \in I_M$ to write on the tape, the machine's next state $s' \in S_M$, and the tape's motion $d \in \{-1, 0, 1\}$.

Given the machine M, a sequence of tape symbols $v : v_i \in I_M$ can be considered as a virus *for that machine* iff, processing the sequence v at time (sequence point) t implies that at a future time point t' a sequence v'—not overlapping with v—will exist on the tape, and that the sequence v' will have been written by M at a point t'' lying between t and t':

$$\begin{array}{ll} \forall \square_M \; \forall t \; \forall j : & \\ \quad S_M(t) = S_{M_0} & \wedge \\ \quad P_M(t) = j & \wedge \\ \quad \{\square_M(t,j) \ldots \square_M(t, j + |v| - 1)\} = v & \Rightarrow \\ \exists v' \; \exists j' \; \exists t' \; \exists t'' : & \\ \quad t < t'' < t' & \wedge \\ \quad \{j' \ldots j' + |v'|\} \cap \{j \ldots j + |v|\} = \emptyset & \wedge \\ \quad \{\square_M(t', j') \ldots \square_M(t', j' + |v'| - 1)\} = v' & \wedge \\ \quad P_M(t'') \in \{j' \ldots j' + |v'| - 1\} & \end{array} \tag{2}$$

where

- $t \in \mathbb{N}$ stands for the number of times the machine has performed its basic operation—"move"
- $P_M(t) \in \mathbb{N}$ represents the machine's tape cell position number at time t
- S_{M_0} is the machine's initial state
- $\square_M(t, c) \in I_M$ represents the content of cell c at time t

Note that in the original seminal reference [13] the above virus definition appears in the context of a viral set $VS = (M, V)$: a tuple consisting of a Turing Machine M and a set of symbol sequences $V : v, v' \in V$. From the virus definition it is clear that the notion of a virus is intimately associated with its interpretation in a given context—environment. It has been shown [13] that "any self-replicating tape symbol sequence is a one element VS, that there are countably infinite VSs and non VSs, that machines exist for which all tape sequences are viruses and for which no tape sequences are viruses, and that any finite sequence of tape symbols is a virus with respect to some machine." The same reference also proves that in the general case determining whether a given tuple $(M, X) : X_i \in I_M$ is viral is an undecidable problem (i.e. that there is no algorithm that can reliably detect all viruses) through a reasoning similar to that employed to prove the undecidability of the Halting Problem [14]. Other researchers have shown that there are also virus types (viruses that evolve to contain an instance of the virus detection program) that can not be detected by any error-free algorithm [15].

As is often the case, current practice differs from theory. Typical pattern based virus detection software scans a (relatively) known environment (processor architecture and operating system) to locate one of several (thousands in practice) *a-priori* known viruses. In the following paragraphs we will therefore establish the complexity of the more restricted problem of locating an instance of a known finite length virus in a given execution environment. For instance, the virus programs we provide in the appendices are only viral in the context of compilation and execution following the rules of Haskell and ANSI C/POSIX, respectively.

The complexity of detecting a known fixed virus pattern of length M in a program of length N is harnessed by the Boyer-Moore string-searching algorithm [16] which never uses more than $N + M$ steps and under many circumstances (a small pattern and a large alphabet) can use about N/M steps. Unfortunately, as we saw in the previous section, virus writers are seldom thus accommodating; fixed search patterns are not any more a viable virus detection method. We will prove that the problem of reliably identifying a bounded-length mutating virus is NP-complete. Our proof is based on showing that a virus detector D for a certain virus strain V can be used to solve the satisfiability problem, which is known to be NP-complete [17]. (This approach works in the same way for any similar NP-complete problem; the satisfiability of the problem we are examining is not a special case.)

The virus V is a mutating self-replicating program. We assume that the virus detector D can reliably determine in P-time whether a given candidate program C is a mutation of the virus V. We will use the virus detector as an oracle for determining the satisfiability of an N-term boolean formula S of the following type:

$$\begin{array}{lll} S = & (x_{a_{1,1}} \vee x_{a_{1,2}} \vee \neg x_{a_{1,3}} \vee \ldots) & \wedge \\ & (\neg x_{a_{2,1}} \vee x_{a_{2,2}} \vee \neg x_{a_{2,3}} \vee \ldots) & \wedge \\ & (x_{a_{3,1}} \ldots) & \wedge \\ & \vdots & \end{array} \tag{3}$$

$$0 \leq a_{i,j} < N \tag{4}$$

and thereby show that a P-time reliable virus detector is equivalent to a P-time solution to the satisfiability problem.

We will use the satisfiability formula S to create a virus archetype A and a possible instance of a virus phenotype P. The virus is a triple

$$(f, s, c) \tag{5}$$

where

- f is the virus processing and replication function
- s is a boolean value indicating whether an instance of the virus has found a solution to S
- c is an integer encoding the candidate values for S

The function f maps a triple (f, s, c) into a new triple (f, s', c') and is defined as follows:

$$\lambda(f, s, c).(f, s \vee S, \text{if} c = 2^N \text{ then } c \text{ else } c + 1) \tag{6}$$

Each S term x_n is calculated from c through the expression

$$\left\lfloor \frac{c}{2^n} \right\rfloor \bmod 2 = 1 \tag{7}$$

A new generation of the virus is generated by applying f to the current generation.

Expressed in words, each new virus generation

1) evaluates S by extracting successive boolean value combinations from c
2) increments c until it reaches 2^N
3) passes the result of the S evaluation to the next generation

We can now ask D whether the virus archetype A

$$(f, \mathrm{False}, 0) \tag{8}$$

will ever result in a virus mutation phenotype P

$$(f, \mathrm{True}, 2^N) \tag{9}$$

that is whether one of the virus mutations will satisfy S.

We have thus proven that a reliable virus detector D operating in P-time can be used as a P-time satisfiability oracle and that therefore reliable virus detection is NP-complete.

As an example for the operation of the virus consider the satisfiability of the formula S

$$(x_0 \vee x_1) \wedge \neg x_0 \tag{10}$$

The virus replication function f—after omitting for simplicity of expression the conditional, which only serves to limit the number of virus mutations—will be:

$$\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \neg x_0, c + 1) \tag{11}$$

the corresponding archetype A:

$$(\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \neg x_0, c + 1), \mathrm{F}, 0) \tag{12}$$

and the phenotype P indicating satisfiability:

$$(\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \neg x_0, c + 1), \mathrm{T}, 4) \tag{13}$$

This particular virus will generate a mutation P—and thereby indicate that S is satisfiable—in four generations through the following sequence:

$$\begin{aligned}
&fffffA \rightarrow \\
&fffff\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \neg x_0, c + 1) \\
&\quad (\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F}, 0) \xrightarrow{\beta} \\
&ffff(\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F} \vee (\mathrm{F} \vee \mathrm{F}) \wedge \neg \mathrm{F}, 0 + 1) \xrightarrow{\delta} \\
&ffff(\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F}, 1) \rightarrow \\
&fff\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \neg x_0, c + 1) \\
&\quad (\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F}, 1) \xrightarrow{\beta} \\
&fff(\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F} \vee (\mathrm{T} \vee \mathrm{F}) \wedge \neg \mathrm{T}, 1 + 1) \xrightarrow{\delta} \\
&fff(\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F}, 2) \rightarrow \\
&ff\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \neg x_0, c + 1) \\
&\quad (\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F}, 2) \xrightarrow{\beta} \\
&ff(\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{F} \vee (\mathrm{F} \vee \mathrm{T}) \wedge \neg \mathrm{F}, 2 + 1) \xrightarrow{\delta} \\
&ff(\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{T}, 3) \rightarrow \\
&f\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \neg x_0, c + 1) \\
&\quad (\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{T}, 3) \xrightarrow{\beta} \\
&(\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{T} \vee (\mathrm{T} \vee \mathrm{T}) \wedge \neg \mathrm{T}, 3 + 1) \xrightarrow{\delta} \\
&(\lambda(f, s, c).(f, s \vee S, c + 1), \mathrm{T}, 4) \equiv P
\end{aligned} \tag{14}$$

IV. Implications

The creation of metamorphic viruses is a relatively recent phenomenon that places a considerable threat on our information system infrastructures. From a theoretical point of view, the viruses bear remarkable similarities to the virus we have examined and the examples depicted in this paper's appendices. Virus detection programs however need not be 100% correct. Users can tolerate the (typically remote) possibility of some "noise" (false positives), because in practice it is quite rare for a non-viral program to match the detection pattern of a known virus. As an example, a virus detector that detected this paper's viruses and also detected as a virus all triplets of the form $(f, s, n) : \forall s\ \forall n$ (even cases where f is a non-satisfiable formula and s is true) would probably be tolerated as a functioning "good-enough" virus detector, although strictly speaking it detects some false positives. Such a virus detector can be implemented to terminate in linear time and is not NP-complete.

Thus, given the difference between the theoretically perfect detection (which is in the general case undecidable, and for known viruses, as we demonstrated, NP-complete) and the practically sufficient identification (which is the basis for a number of working virus scanner implementations) two questions arise.

1) How can the notion of "sufficiently good detection" be formalized in information theory terms?
2) Can the increasing ability of metamorphic viruses to mutate move the identification threshold currently used by virus detection programs to the point where either numerous legitimate data sequences are falsely detected as viruses, or real viruses fail to be detected?

An interesting phenomenon affecting the above topics concerns the currently permeable boundary between code and data. *Buffer overflow attacks* [18] are based on data that overwrites a carelessly written program's return stack address lying at the end of a data buffer to cause the program to execute part of that data. This renders all data files (documents, images, music, video—many of them highly compressed) stored on a computer potential carriers of viral code and dramatically increases the data a virus detector has to scan and discriminate. Few viruses currently propagate through buffer overflows; these weaknesses have traditionally been mainly exploited by worms and Trojan horses [19]. However, once such viruses are released, the current virus detection approach will come under increasing strain, faced with the short pattern vectors of mutating viruses and orders of magnitude more data to scan; as an example a 18GB disk filled with MP3 files is likely to contain any 4-byte (virus) pattern. In the medium and long term, hardening our security defences and developing software, procedures, and work practices that will stem the spread of malware seem to be the only reasonable alternatives.

Appendix I
Virus Code in Haskell

The following code defines the virus replication function and the respective archetype and candidate phenotype, for determining the satisfiability of the expression

$$(x_0 \vee x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_5) \wedge (x_2) \tag{15}$$

The satisfiability function candidate values are encoded using Haskell's arbitrary precision integers.

```
module Virus where
 replicate :: (replicate, Bool, Integer)->
   (replicate, Bool, Integer)
 replicate (v, b, i) = (v, b ||
     (((bit 0 i) || (bit 3 i) ||
        not (bit 4 i)) &&
      (not (bit 1 i) || (bit 5 i)) &&
      ((bit 2 i)))
     , if i == 64 then i else i + 1)

 -- Extract bit b out of the Integer n
 bit :: Integer -> Integer -> Bool
 bit b n = n `div` (2 ^ b) `rem` 2 == 1

 virus_archetype = (replicate, False, 0)
 virus_phenotype = (replicate, True, 64)
```

Appendix II
Virus Code in C

The following code is the virus archetype, again for determining the satisfiability of the expression (15). The satisfiability function candidate values are encoded as elements of the array `x`.

```
#include <stdio.h>
#include <ctype.h>

/* Number of variables to satisfy */
#define N 6
int x[N] = {
0, 0, 0, 0, 0, 0,
};
void
advance(void)
{
  int i, j;
  for (i = 0; i < N; i++)
    if (x[i] == 0) {
      for (j = 0; j < i; j++)
        x[j] = 0;
      x[i] = 1;
      return;
    }
}
void
print_vector(FILE *f)
{
  int i;
  for (i = 0; i < N; i++)
    fprintf(f, "%c, ", x[i] ? '1' : '0');
  fputc('\n', f);
}
main()
{
  char buff[1024];
  FILE *fi = fopen(__FILE__, "r");
  FILE *fo = fopen("new" __FILE__, "w");

  if ((x[0] || x[3] || !x[4]) &&
      (!x[1] || x[5]) && (x[2]))
    fprintf(fo, "/* Satisfied */\n");
  advance();
  while (fgets(buff, sizeof(buff), fi))
    if (isdigit(buff[0]))
      print_vector(fo);
    else
      fputs(buff, fo);
  fclose(fi); fclose(fo);
  system("cc new" __FILE__);
  return 0;
}
```

The candidate virus phenotype begins as follows:

```
/* Satisfied */
[...]
int x[N] = {
1, 1, 1, 1, 1, 1,
};
```

Acknowledgments

The author acknowledges the valuable suggestions of the anonymous referees on an earlier version of this paper. The publication of this work was supported by the IST project mEXPRESS (IST-2001-33432), which is funded in part by the European Commission.

References

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[2] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi, "A taxonomy of computer program security flaws," *ACM Computing Surveys*, vol. 26, no. 3, pp. 211–254, Sept. 1994.

[3] Peter J. Denning, "Computer viruses," *American Scientist*, pp. 236–238, May-June 1988.

[4] Fred Cohen, "Computer viruses: Theory and experiments," *Computers & Security*, vol. 6, no. 1, pp. 22–35, Feb. 1987.

[5] Eugene H. Spafford, Kathleen A. Heaphy, and David J. Ferbrache, "A computer virus primer," in *Computers Under Attack: Intruders, Worms, and Viruses*, Peter J. Denning, Ed., chapter 20, pp. 316–355. Addison-Wesley, 1990.

[6] Vassilis Prevelakis and Diomidis Spinellis, "Sandboxing applications," in *USENIX 2001 Technical Conference Proceedings: FreeNIX Track*. Usenix Association, June 2001.

[7] Prabhat K. Singh and Arun Lakhotia, "Analysis and detection of computer viruses and worms: An annotated bibliography," *ACM SIGPLAN Notices*, vol. 37, no. 2, pp. 29–35, Feb. 2002.

[8] R. Rivest, "RFC 1321: The MD5 message-digest algorithm," Apr. 1992, Status: INFORMATIONAL.

[9] Ken L. Thompson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984.

[10] Tom Duff, "Experience with viruses on UNIX systems," *Computing Systems*, vol. 2, no. 2, pp. 155–171, Spring 1989.

[11] M. Douglas McIlroy, "Virology 101," *Computing Systems*, vol. 2, no. 2, pp. 173–184, Spring 1989.

[12] Frédéric Perriot, Peter Ferrie, and Péter Ször, "W32/Simile," Online http://www.virusbtn.com/resources/viruses/indepth/simile.xml. Current June 2002, 2002.

[13] Fred Cohen, "Computational aspects of computer viruses," *Computers & Security*, vol. 8, no. 4, pp. 325–344, June 1989.

[14] Alan M. Turing, "On computable numbers, with an application to the Entscheidungs Problem," *Proceedings of the London Mathematical Society*, vol. 2, no. 42, pp. 230–265, 1936, Corrections in 2(43):544–546.

[15] David M. Chess and Steve R. White, "An undetectable computer virus," in *Virus Bulletin Conference*, Sept. 2000, Online http://www.research.ibm.com/antivirus/SciPapers/VB2000DC.pdf. Current June 2002.

[16] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 262–272, Oct. 1977.

[17] S. A. Cook, "The complexity of theorem prooving procedures," in *Proceeding of the 3rd ACM Symposium on Theory of Computing*. 1971, pp. 151–158, ACM.

[18] Crispan Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *Proceedings of the DARPA Information Survivability Conference and Exposition*, Hilton Head, SC, USA, Jan. 2000, DARPA, pp. 119–129.

[19] Mark W. Eichlin and Jon A. Rochlis, "With microscope and tweezers: An analysis of the internet virus of November 1988," in *IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 1989, pp. 326–345.

PLACE PHOTO HERE

Diomidis Spinellis holds an MEng in Software Engineering and a PhD in Computer Science both from Imperial College (University of London, UK). Currently he is an Assistant Professor at the Department of Management Science and Technology at the Athens University of Economics and Business, Greece. He is the author of the book "Code Reading: The Open Source Perspective" (Addison Wesley, 2003) and more than 60 journal papers and conference presentations. He has contributed software to the BSD Unix distribution, the X-Windows system, and is the author of a number of open-source software packages, libraries, and tools. His research interests include Information Security, Software Engineering, and Ubiquitous Computing.

Dr. Spinellis is a member of the ACM, the IEEE Computer Society, the Greek Computer Society, the Technical Chamber of Greece, and a founding member of the Greek Internet User's Society. He is a co-recipient of the Usenix Association 1993 Lifetime Achievement Award.