Fortinet-1013

An Undetectable Computer Virus

David M. Chess and Steve R. White
IBM Thomas J. Watson Research Center
Hawthorne, New York, USA
chess@us.ibm.com*, *srwhite@us.ibm.com

One of the few solid theoretical results in the study of computer viruses is Cohen's 1987 demonstration that there is no algorithm that can perfectly detect all possible viruses [1]. This brief paper adds to the bad news, by pointing out that there are computer viruses which no algorithm can detect, even under a somewhat more liberal definition of detection. We also comment on the senses of "detect" used in these results, and note that the immediate impact of these results on computer virus detection in the real world is small.

Computer Viruses

Consider the set of programs which produce one or more programs as output. For any pair of programs **p** and **q**, **p** *eventually produces* **q** if and only if **p** produces **q** either directly or through a series of steps (the "eventually produces" relation is the transitive closure of the "produces" relation.) A *viral set* is a maximal set of programs **V** such that for every pair of programs **p** and **q** in **V**, **p** eventually produces **q**, and **q** eventually produces **p**. ("Maximal" here means that there is no program **r** not in the set that could be added to the set and have the set still satisfy the conditions.) For the purposes of this paper, a *computer virus* is a viral set; a program **p** is said to be an instance of, or to be infected with, a virus **V** precisely when **p** is a member of the viral set **V**. A program is said to be *infected* simpliciter when there is some viral set **V** of which it is a member. A program which is an instance of some virus is said to *spread* whenever it produces another instance of that virus. The simplest virus is a viral set that contains exactly one program, where that program simply produces itself. Larger sets represent polymorphic viruses, which have a number of different possible forms, all of which eventually produce all the others.

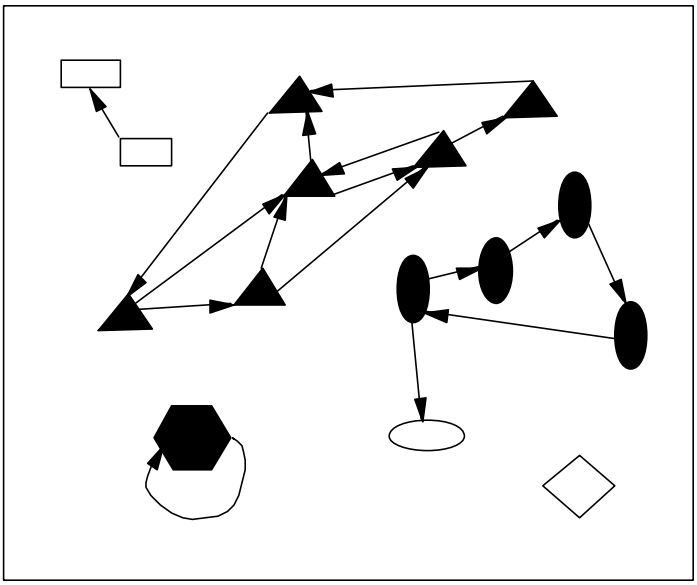

Figure 1. The shapes represent programs, and the arrows show which programs produce which as output. The filled shapes are members of viral sets, the empty shapes are not. The filled hexagon represents a simple non-polymorphic virus, whose sole member produces only itself.

In practical terms, this notion of computer virus encompasses overwriting viruses (which replace existing programs with copies of themselves) and some kinds of worms (which spread as standalone programs by creating new copies of themselves). A more complex notion of computer virus would incorporate "parasitic" viruses, which infect other programs by inserting themselves in such a way that both the viral code and the original program are executed when the infected program is executed. (The classic informal definition of "computer virus" is "a program that can 'infect' other programs by modifying them to include a possibly evolved copy of itself." [1]. A more formal definition in terms of regions of a Turing Machine tape can be found in [2].) In a subsequent paper, we will extend the current results to that richer notion of computer virus; essentially all the results we obtain here still hold.

Detecting a Virus

For the purposes of this paper, an algorithm A detects a virus **V** if and only if for every program **p**, A(**p**) terminates, and returns "true" if and only if **p** is infected with **V**. Similarly, an algorithm A detects a set of viruses ***S*** if and only if for every program **p**, A(**p**) terminates, and returns "true" if and only if **p** is infected with some virus **V** which is a member of ***S***. This is essentially Cohen's definition in [1], and it is the only formal definition of detection that has proven theoretically fruitful. It also captures (at least to a first approximation) our intuitive notion of computer virus detection.

Cohen's Result

In [1], Fred Cohen demonstrates that there is no algorithm that can detect the set of all possible computer viruses (returning "true" if and only if its input is an object infected with some computer virus). The proof is a simple diagonal argument, like Cantor's proof of the uncountability [3] of the real numbers, or Turing's proof of the undecidability of the Halting Problem [4]. For any candidate computer virus detection algorithm A, there is a program **p**, which reads:

if A(**p**), then exit; else spread

Clearly A does not return the correct result when called on **p**, since if it returns "true" (i.e. it says that **p** is infected), then **p** just exits (and is therefore not infected), whereas if A returns anything else (i.e. it says that **p** is not infected), then **p** spreads (and is therefore infected).[1] So there is no algorithm which detects all viruses without error; any program that attempts to detect all viruses will either miss some infected files (a false negative), accuse some non-infected files of being infected (a false positive) or fail to return anything (a bug).

An Undetectable Virus

A very similar example demonstrates that there are viruses for which no error-free detection algorithm exists. 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.[2]

As noted above, a virus is said to be "polymorphic" if the size of the viral set is greater than one; that is, if the code of the virus is different in different infected objects. Consider a virus which is sufficiently polymorphic that for any implementable algorithm X the program **p:**

```
if X(p) then exit, else spread
```

is an instance of the virus (provided of course that **p** actually spreads). There is no algorithm B that correctly detects this virus, by an argument directly analogous to that above: for any algorithm B that claims to detect this virus, there is a program **q:**

```
if B(q) then exit, else spread
```

for which B does not return the correct result. If B(**q**) returns true, then **q** does not spread, and is therefore not an instance of this or any other virus; whereas if B(**q**) returns false, then **q** does spread, and is an instance of the virus.

Is any possible actual virus sufficiently polymorphic to have this property? Clearly yes. Consider a virus **W** one instance of which is **r:**

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

For any candidate **W**-detection algorithm C, there is a program **s:**

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

---

[1] Note that A is not an input to **p** here; every time **p** is run, it calls A on itself, and spreads if and only if A returns false. The program **p** therefore always spreads, or always exits, regardless of any input.

[2] A similar proof, showing that no Turing Machine program can decide if one virus "evolves" into another, can be found in [2], but as far as we are aware the implications of that result for virus detection have never been explored.

```
}
subroutine_one:
    return C(argument);
```

for which C does not return the correct result; if C(**s**) returns true, then **s** just exits (and is therefore not an instance of **W**, or of any other virus), whereas if C(**s**) returns false, then **s** is an instance of **W**. So no algorithm can detect **W** without error.[3]

A Looser Notion of Detection

There is a looser notion of detection under which our result still holds. We may be willing to forgive a candidate **V**-detection algorithm for claiming to find **V** in some program **p** which is not infected with **V**, provided that **p** is infected with *some* virus. Let us say, then, that an algorithm A loosely-detects a virus **V** if and only if for every program **p**, A(**p**) terminates, returning "true" if **p** is infected with **V**, and returning something other than "true" if **p** is not infected with any virus. The algorithm may return any result at all for programs infected with some virus other than V (although it must still terminate).

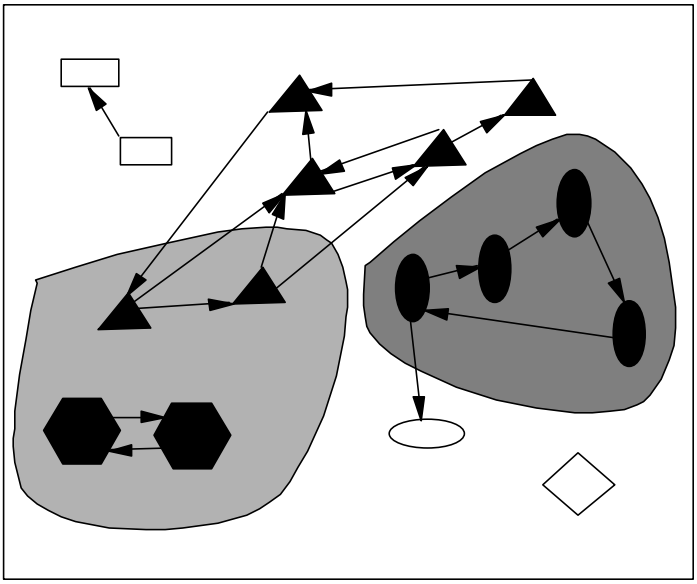

Figure 2. The slanted lines show the (perfect) detection of the viral set of filled ovals; the algorithm picks out exactly those programs infected with that virus. The vertical lines show loose-detection of the viral set consisting of the filled hexagons; the algorithm picks out all the programs in that viral set, as well as some other infected programs.

It is clear that our result still applies under this looser notion of detection. Since every algorithm either returns true for a program which simply exits, or fails to return true for some program infected with **W**, no algorithm even loosely-detects **W**.

Comparison with Cohen

Our result is clearly complementary to Cohen's result in [1] that no algorithm can detect all

[3] This example assumes that P has access to an arbitrarily-long stream of random bits; some formalizations of the notion of algorithm do not allow this. See the appendix for a somewhat more complex example that does not require any random bits.

viruses. That result may be expressed as

" A, $ **V** s.t. A does not detect **V** (for every algorithm, there is some virus that it does not detect)

whereas our results are

$ **V** s.t. " A, A does not detect **V** (there exists a virus which no algorithm perfectly detects)

$ **V** s.t. " A, A does not loosely-detect **V** (there exists a virus which no algorithm loosely-detects)

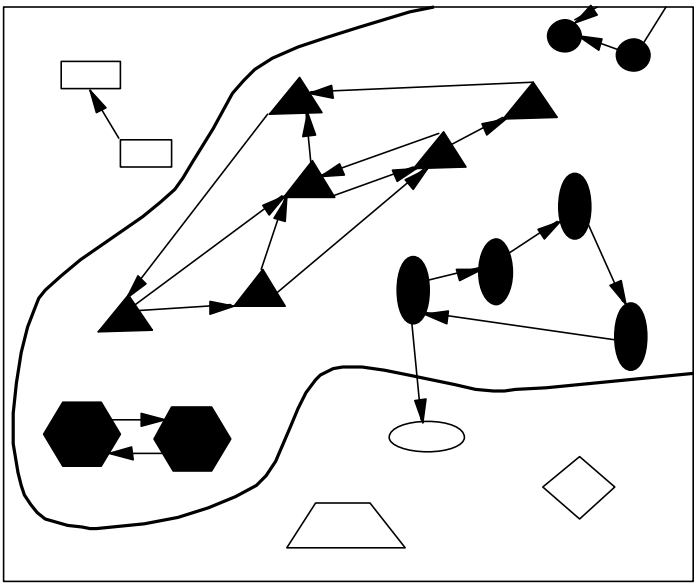

Figure 3. Cohen's result says that it is impossible for a program to perfectly draw the solid line suggested above, enclosing all and only those programs that are infected with some virus. For every program that attempts to draw that line, there will be some infected object that the program says is uninfected, or some uninfected object that it says is infected.

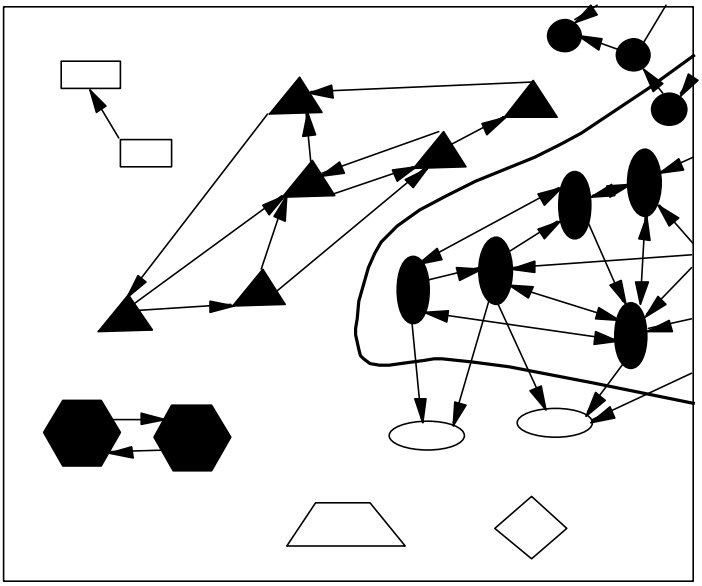

Figure 4. Our result says that for some viruses it is impossible for a program to correctly draw the solid line suggested above, enclosing all those programs that are infected with that virus, and enclosing no programs that are not infected with any virus.

Practical Consequences

The virus **W** above is clearly not a remarkably viable virus, nor is the failure of detection a particularly serious one. In particular, no user of a real virus detection program D would object if it were to say that it found a virus in **t**:

```
if D(t), then exit; else spread
```

or in **u**:

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

since, although these do not in fact spread, they are closely related to programs that do spread. The main immediate impact of Cohen's undecidability result on the day-to-day activities of those who study computer viruses is that we can dismiss without detailed study any claim that some method correctly detects "all possible viruses known and unknown". The main practical impact of the current result is to dispel the notion that it is always possible to create a detector for a given virus that has no false positives, even if you have a copy of the virus in hand.

When we say in common speech that a given program detects a given virus, we mean something rather different from the formal senses above. Every widely-deployed virus detection program in use today will claim to find a virus in at least some non-viral objects (a false positive), because the methods used for detection are approximate, based on the presence of a particular binary

string in a certain place, on the calculation of the finite-size checksum of a macro, on a certain pattern of changes to a file, and so on. Producers of anti-virus software of course try to minimize the number of actual non-viral programs that are falsely detected. But no one worries about the fact that the algorithms used to detect viruses produce false positives on an enormous number of non-viral objects that have never been, and will never be, present on any actual user's computer. This paper's title, then, is deliberately somewhat provocative: while the viruses that we present here are undetectable in the strict formal sense of the term, there is no reason to think that it is impossible to write a program that would detect them sufficiently well for all practical purposes.

Acceptable virus detection, in the real world, involves detecting all viable instances of the virus in question, and preferably some number of minor variants of it, while falsely detecting the virus in only a vanishingly small number of innocent programs that are actually present on a computer somewhere. It would be helpful to have a formal characterization of this more realistic notion of detection; theorists in the area of computer virus protection might usefully work toward such a characterization.

References

[1] Fred Cohen, "Computer Viruses: Theory and Experiments", Computers and Security 6 (1987) 22-35.
[2] Fred Cohen, "Computational Aspects of Computer Viruses", Computers and Security 8 (1989) 325-244.
[3] Y. N. Moschovakis, “Notes on Set Theory”, Springer-Verlag, NY, 1994, p. 11 (or any similar text).
[4] Alan Turing, "On Computable Numbers, with an application to the *Entscheidungsproblem*", Proceedings London Mathematical Society (series 2)
vol 42, 1936-7, pp.230-265.

Appendix

The example undetectable virus in the body of the paper requires access to an arbitrarily-large stream of random bits; many formalizations of the notion of algorithm do not allow this. This Appendix gives a slightly more complex example that does not require any random bits. Understanding the main results of this paper does not require understanding this Appendix.

Consider the virus **W'** one instance of which is **x**:

```
state = 0;
serial_number = 0;
if subroutine_one(x) then exit;
if state==0 then {
     spread once, incrementing the constant in the second line of the new copy,
     spread again, making these changes in the new copy:
          change the constant in the first line to 1,
          replace the body of subroutine_one() with the subroutine corresponding to
          serial_number under mapping M
} else {
     spread, making these changes in the new copy:
          change the constant in the first line to 0,
          change the constant in the second line to 0,
          replace the body of subroutine_one() with the code "return false;"
}
exit;
subroutine_one():
     return false;
end;
```

where M is a mapping from the integers to the set of programs for the relevant machine, such that for every implementable algorithm A, there is some integer n such that M maps n to A. Many such mappings of course exist for the typical machine. (Note that the variable serial_number must be stored as an unlimited-size integer (a "bignum").)

All instances of the above which begin "state = 0;" are viral, and trivial to detect; they differ only in the second line, where serial_number is set. Every program which differs from the above only in having a different non-negative integer in the second line is an element of **W'**. Instances of the above which begin "state = 1;", on the other hand, are only sometimes viral, since the content of subroutine_one() is different, and will often not terminate at all, or will return true (causing the program to exit before spreading). Those that are viral create, in the next generation, exactly the above code again, so every element of **W'** is an ancestor, eventually, of every other (and **W'** is therefore a viral set). For any candidate **W'**-detection algorithm E, consider program **y**:

```
state = 1;
serial_number = Q;
if subroutine_one(y) then exit;
if state==0 then {
     spread once, incrementing the constant in the second line of the new copy,
     spread again, making these changes in the new copy:
          change the constant in line 1 to 1,
          replace the body of subroutine_one() with the subroutine corresponding to
          serial_number under mapping M
} else {
     spread, making these changes in the new copy:
          change the constant in line 1 to 0,
          change the constant in line 2 to 0,
          replace the body of subroutine_one() with the code "return false;"
}
exit;
subroutine_one():
     return E(argument);
end;
```

where M maps Q to an implementation of E. Just as described above, E does not return the correct result when called on **y**, so E does not detect **W'**. And because E either returns "true" on a program that simply exits, or returns something other than "true" on a program infected with **W'**, A also fails to loosely-detect **W'** in the sense defined above.