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.

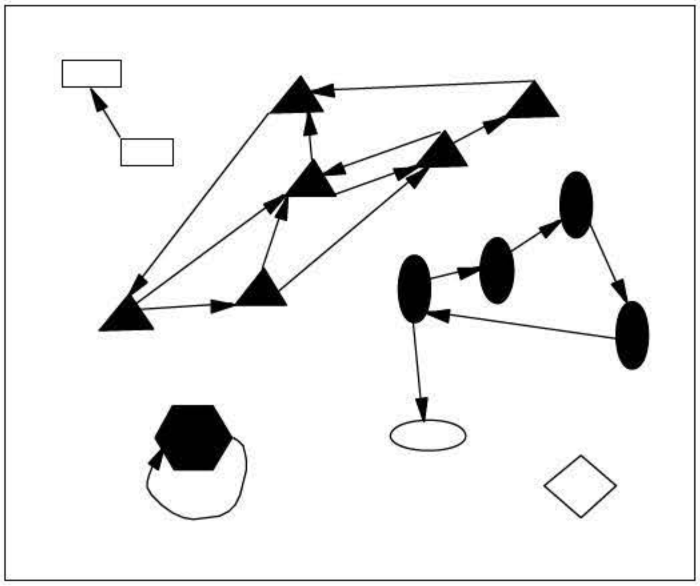

Figure 1. The shapes represent programs, and the arrows show which programs produce which as output. The filled shapes are members of viral sets, the empty shapes are not. The filled hexagon represents a simple non-polymorphic virus, whose sole member produces only itself.

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).

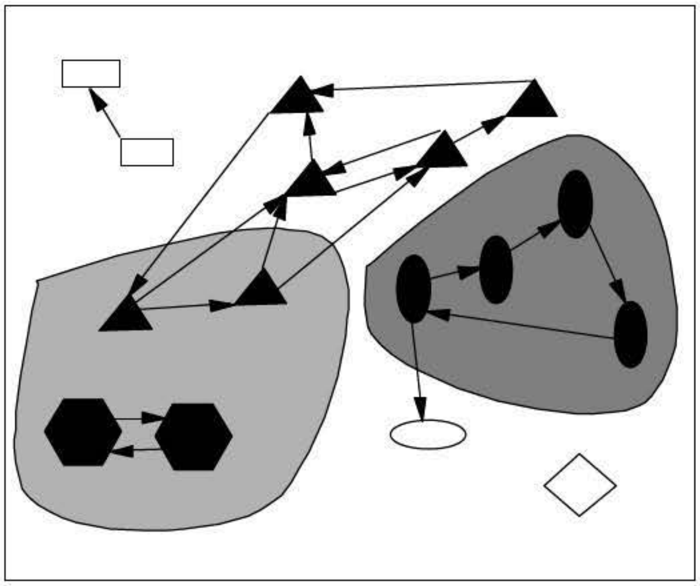

Figure 2. The slanted lines show the (perfect) detection of the viral set of filled ovals; the algorithm picks out exactly those programs infected with that virus. The vertical lines show loose-detection of the viral set consisting of the filled hexagons; the algorithm picks out all the programs in that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

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

whereas our results are

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

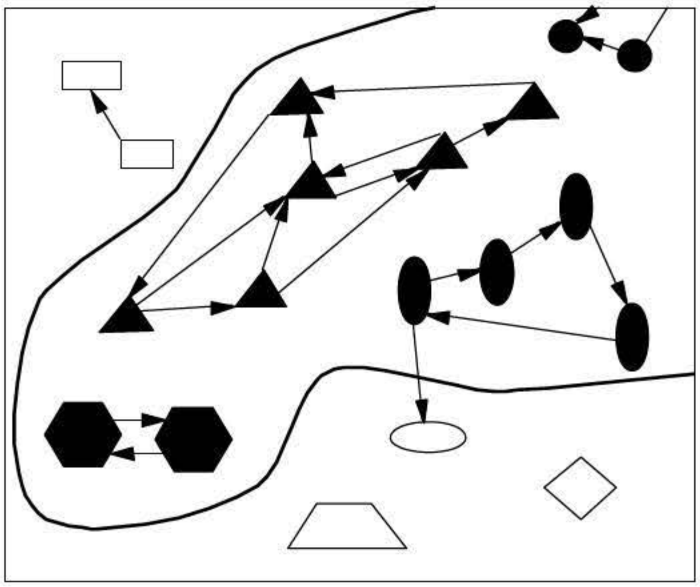

Figure 3. Cohen's result says that it is impossible for a program to perfectly draw the solid line suggested above, enclosing all and only those programs that are infected with some virus. For every program that attempts to draw that line, there will be some infected object that the program says is uninfected,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

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase® Smarter legal research.