# On the Impossibility of Virus Detection

**David Evans**
*University of Virginia*
https://www.cs.virginia.edu/evans

*Why is computer security so hard? Why do virus scanners not work?*

The question of how to detect malicious programs seems like a practical one, but at its core depends on the fundamental nature of general purpose computers and deep theoretical questions about what computers can and cannot do. This essay starts by formalizing the question of what it means to detect a virus. Then, we discuss theoretical results that show solving this (and most other important computer security problems) is impossible for general purpose computers. But impossible doesn't mean hopeless — it just means we have to redefine our problems and approaches, and aim for solutions that work well in practice, even if nothing can work in theory.

## 1 Virus Detection

Although there are many different kinds of malicious programs, we limit our focus to a narrow and precise type of malware known as a *virus*. The notion of a virus, in computing, is inspired by biological viruses that infiltrate cells of other organisms and hijack their machinery for duplicating themselves. Similarly, a computer virus is a program that spreads itself by infecting other programs. Computer viruses have been studied formally since Fred Cohen's pioneering work in the 1980s [1, 2].

We use a limited, but precise, definition of a virus:

> A *virus* is a computer program that when executed will copy its own code into another program.

Many other definitions would be reasonable, but to talk precisely about virus detection we need a somewhat formal definition of what a virus is. This definition simplifies a lot of the real issues in malware detection, and does not capture many other malicious things a program could do. Actually deciding what program behaviors are malicious is highly subjective. For example, a program that replaces advertisements in web pages may be malware, may be an ad blocker behaving as the machine owner desires, or may be a required trade-off the user knowingly accepts. For our purposes, though, we don't need to resolve the tricky issues that would come from defining malware as a program that does something the user doesn't want. As it turns out, we'll see that the impossibility of virus detection applies to other definitions as well.

The goal of a *virus detector* is to take as input a sample and produce as output a Boolean that indicates if that sample is a virus. If a sample is a virus—that is, it exhibits the virus behavior of copying its code into another program—the virus detector should output **True**; otherwise, the detector should output **False**. To formalize this, we define the Virus Detection Problem:

*Virus Detection Problem*

**Input:** A description of a program.
**Output:** If the input program behaves like a virus (running it can infect other files) output **True**. Otherwise, output **False**.

A perfect virus detector would produce the correct output for all input program descriptions, and would always finish. Unfortunately, even with a simplified and precise definition of a virus, such a virus detector is not just hard to build, it is *theoretically impossible*! To understand why, we need to step back and understand more deeply what a computer is.

## 2 Universal Machines

*What is so special about computers?*

Humans have been building machines for thousands of years. The earliest machines amplified the physical abilities of humans and animals. Levers, pulleys, wheels and axles made it possible for us to lift heavier objects and move them more efficiently. As humanity progressed, we learned to build increasingly complex and adaptable machines, leveraging power from animals, wind, and water, and developing ways to control the function of machines by adjusting levers and switches.

Although these machines were powerful and versatile, none of these machines were *universal*. A universal machine is a machine that can simulate any other machine. If we are talking about machines that transform physical objects, universal physical machines cannot be built. But in the abstract world of information, universal machines are everywhere [4].

In a practical sense, computers differ in the ways they get input (e.g., from a touchscreen, keyboard, buttons, network, or camera) and how they provide output (e.g., signals to a graphical display, network messages, flashing lights). They also differ in how fast they can operate and how much memory they have. Aside from these, though, all computers are deeply equivalent. The processor in your microwave can simulate your smartphone or the world's most powerful supercomputer, and vice versa. The only difference is how long it takes and how large a problem they can solve before running out of memory.

The reason all computers are so universally identical is because they run programs. A program is just a list of instructions, and all programs can be expressed using a very limited and simple set of instructions. This seems very surprising at first, especially because of all the effort that goes into designing processors that can execute programs efficiently and all the bloviating about the advantages and disadvantages of different programming languages. But, all we need to execute any program is memory and control.

For memory, need a way to store a value — this could be any element from a finite set, like a letter or digit. It would be enough to just be able to store one of two possible values. In computing, the two distinguished values are denoted as the bits, **0** and **1**. Then, we can put together bits from many locations to represent larger numbers.

To keep things simple, though, let's assume each location can store an arbitrarily large number and is initialized to hold the value 0. Memory in practical computers is similar to this, except the important difference is that in an idealized abstract model the number of locations is infinite, wheras any computer that can actually be physically constructed has a limited number of memory locations.

The stored values are in locations that can be identified using a natural number. We'll assume two simple instructions for updating the values stores in memory: increment [*location*] and decrement [*location*]. The increment instruction updates the value stored in the *location* parameter by one; the decrement instruction decreases it by one. The [*location*] parameter is an indirection: the location update is the location identified by the value in *location*. For example, if the initial value in all locations is 0, the instruction increment [0], will increment the value stored in the location identified by the value in location 0 (which is currently 0). Executing the same instruction again, would increment the value stored in location 1. This ability to access memory indirectly is important—without it, each instruction in a program could only access a fixed memory location; with it, a single instruction can be used to access any location.

For control, we need a way to execute instructions and to decide which instruction to execute next based on a value in memory. The program is just a list of instructions, so we can number them $1, 2, \ldots, N$. Normally, we just execute the next instruction, starting with instruction 1, then instruction 2, and so on. That's not enough for interesting programs, though, we need a way to make decisions and keep going.

Adding one simple control instruction is sufficient: ifzero [*location*] *target*. If the value stored in location *location* is **0**, it jumps to the *target* instruction as the next instruction to execute instead of continuing normally; otherwise, the instruction does nothing and execution proceeds to the following instruction. As with the store instruction, the *location* value is an indirect reference: the decision to jump is based on the value in the location given by the *location* parameter.

Importantly, the jump *target* can be before the current instruction. This means we can have backward jumps, and execute the same instructions over and over again. This enables short programs to produce infinitely long executions. For example, the program

```
1.  ifzero [0] 1                    LOOPFOREVER
```

runs forever, executing instruction 1 over and over again since the initial value in memory location 0 is 0.

Practical computers have much larger and more complex instruction sets than these three simple instructions, including instructions for performing mathematical operations like floating point division and for moving values around memory in various ways. But, all of these instructions could be simulated with a sequence (possibly a very long one!) of instructions in our simple three-instruction language.[1]

We won't attempt to provide a full proof that our simple instructions are enough to simulate every computer, but to build our intuition let's consider how to simulate a general addition instruction with our two instructions. Our goal is to simulate the instruction, add [*location1*] [*location2*], which stores in *location1* the sum of the values currently in *location1* and *location2*.

Here's a program that simulates an add instruction for locations 0 and 1 using only increment, decrement, and ifzero:

```
1.  increment [2]          Make sure location 2 holds a non-zero value.
2.  ifzero [1] 6           If the value in location 1 is zero, we're done.
3.  increment [0]          Add one to the value in location 0.
4.  decrement [1]          Subtract one from the location in location 1.
5.  ifzero [2] 2           Keep going - jump back to instruction 2.
6.  ...                    Finished addition, continue
```

Once we have a way to simulate an add instruction, we can also simulate multiplication (just repeated addition), and build up instructions for any operation.

We can also write a program that can interpret other programs. A *universal machine* is a machine that can simulate any other machine. With the right sequence of instructions, a machine that can execute our three simple instructions would be sufficient. The universal machine takes as input a description of a machine and its input, and simulates what the described machine would do on the given input. So, to define a universal machine we would need to design a way to represent programs in our instruction language using number values that would be stored in memory. The universal machine would need to interpret those stored instructions in its input, and use the rest of memory to simulate the machine describe in the input.

A universal machine produces as output the same output that the simulated input machine would have produced. With a universal machine, the program that provides the instructions for the machine to execute is just another part of its input.

The notion of a universal machine goes back to Gottfried Wilhelm Leibniz in the 1600s [3], but was formalized and clearly understood by Alan Turing in the 1930s [8]. Turing developed a formal model of a computer and showed how to construct a universal machine using that very simple model. Turing developed his model in the 1930s, before anything resembling modern computers was available, so the *computer* he was modeling was a human doing computation with pencil and paper.

Instead of the infinite random access memory we assumed, Turing's model used a simpler infinite one-dimensional tape for working memory. On each step, the machine could read one square on the tape, and make a decision about what to do next following simple rules based on the symbol that was read from the tape and the current *state of mind*. The state of mind keeps track of what the machine is doing, but is one of a finite set of possible states. It could be represented by a natural number, with some maximum possible value.

Based on the state and symbol read from the tape, the rule determines what the machine does next: it writes a symbol on the tape, moves one square left or right, and updates the state of mind. This model has come to be known as a *Turing Machine*, and is still widely used today.

[1]We don't actually even need three instruction types. Several single-instruction programming languages have been proposed that are equally powerful [6].

Turing showed that it was possible to design a machine that could take as input the description and input for *any* machine, and simulate what the described machine would do on that input. This is a *universal machine* — one machine that can simulate every machine! We could construct a Turing Machine that simulates our three-instruction computer, or construct a three-instruction computer to simulate a Turing Machine.

## 3 Undecidability of Virus Detection

*What can computers do?*

Computers seem so powerful they can do just about anything, and the notion of universal machines means even a simple computer can simulate every other computer. There are some problems no computer can solve, however. These problems are labeled *undecidable*.

The most famous undecidable problem is the Halting Problem, introduced by Turing, which asks if a program will run forever:

*Halting Problem*

**Input:** A description of a program.
**Output:** If the input program halts, output **True**. Otherwise, output **False**.

Some programs (such as the one-instruction LOOPFOREVER program introduced earlier) clearly run forever; others (such as any program without any backwards jump) always eventually finish. But, to solve the Halting Problem, we need a machine that can determine for *any* input program if it will run forever.

One way we might try to solve this is to simulate the input program, and just see if it ever terminates. With a universal machine, we know we can simulate any input program, and if the simulated program terminates output True. But, how long should we keep simulating the program to know that it will never terminate? We could try running it for a million steps, but there is a program that terminates after 1,000,001 steps. This is true for any number of simulated steps; however many steps the simulator runs for, there is a program that terminates after one more step.

Another strategy might be to observe the state of the simulated machine. If it ever repeats a complete configuration, we know the simulated machine will run forever. Repeating a complete configuration means the entire state of the simulated machine is identical to a previous configuration. For the three-instruction the configuration is the contents of every memory location and the current instruction to execute. For the Turing machine, the configuration is the state-of-mind, location of the tape head, and the complete contents of the tape.

Since our machine's rules are deterministic, and depend only on the simulated machine configuration, if the same configuration is reached it will do exactly what it did the previous time. This means it will eventually return to the same configuration again, repeating forever. Thus, if we ever see a configuration repeat in the simulated machine we know it will never terminate.

But, there are machines that run forever without ever repeating a configuration. A simple example is a machine that keeps moving right (on the infinite tape):

$$(S, -) \rightarrow (S, \text{write } \mathbf{0}, \text{move } \textbf{Right})$$

This is analogous to our *Loop-Forever* program for the three-instruction model computer. It describes a Turing Machine that starts in state-of-mind $S$, and has one rule that says to write a **0** on the tape, move right, and transition to stay in state $S$. So, on the simulated infinite tape it will run forever, but never repeat the same configuration since it keeps writing more **0** symbols on the tape.

These examples do not prove that there is no program that solves the Halting Problem, just show that two obvious strategies for doing so both fail. To prove there is no solution, we need to show that all possible ideas for solving it fail. Proving impossibility seems impossible, since there are infinitely many possible programs to consider. But, Turing proved that there is no program that can solve the Halting Problem, because if such a program actually existed it would lead to a logical contradiction.

The key idea is to consider what happens when a program runs on a description that includes itself. Since our machine is universal, we know that it can simulate any program description, including its own. Suppose there were some program, HALTS, that solves the Halting Program. That would mean that for any input, *P*, that describes a program, the result of running HALTS on input *P* is **True** if *P* terminates and **False** if *P* runs forever. Suppose we run HALTS on an input that is its own self-description. The result of running HALTS on its own description should be **True**, since if HALTS solves the Halting Problem, it terminates on every input. This is self-referential, but no contradiction yet since outputting **True** on its own description is perfectly reasonable.

To get a contradiction, we need something a bit more complex, but building on the same idea. The trick is to create a program whose self-described behavior is the opposite of what it should be:

CONTRADICTHALTS: if (running HALTS on description of CONTRADICTHALTS outputs **True**) then LOOPFOREVER else *Halt*

If CONTRADICTHALTS terminates, running HALTS on an input that is the description of CONTRADICTHALTS should output **True**. This means the then-branch should be taken, which loops forever. So, CONTRADICTHALTS does not terminate, and running HALTS on an input that is a description of CONTRADICTHALTS should have output **False**. But if running HALTS on input that is a description of CONTRADICTHALTS outputs **False**, the else-branch should execute, which terminates. This means CONTRADICTHALTS terminates, and HALTS should have output **True**.

Neither result makes sense! Encountering a logical paradox like this can only be resolved by concluding that one of the steps to reach the paradox must not exist. The paradox results from the assumption that HALTS exists, where HALTS is a program that solves the Halting Problem.

In a careful proof, such as the one in Turing's paper [8], it would be necessary to show that everything else is sensible and does exist to reach the conclusion that the one new thing we assumed to reach the contradiction must not exist. With our informal argument, perhaps the thing that does not exist is a logical if that branches on the output of HALTS. To produce a more formal argument we would need to show how to construct CONTRADICTHALTS explicitly. We won't go to that level of detail here, but instead will be satisfied with the argument that everything else used in the argument seems likely to exists, and the only uncertainty concerns the existence of HALTS.

Thus, we can conclude that no program that solves the Halting Problem exists. This means there is no program HALTS that can correctly output **True** for every input that is a description of a program that terminates, and **False** for every input that is a description of a program that runs forever. Any candidate for HALTS must, for some input, either give the wrong output or fail to give any output. Hence, the Halting Problem is *undecidable*.

The undecidability of the Halting Problem doesn't just mean there is no program that can solve the Halting Problem, it also means that is no program to solve almost any interesting question about a program, including the Virus Detection problem we started with.

To see this, we use a technique, used frequently in computer science, known as a *reduction* proof. The idea is to show that if we could construct a *B*, we could use *B* to construct an *A*, but since we already know that *A* does not exist, so *B* cannot exist since it would allow us to build something impossible. In this case, *A* is a program that solves the Halting Problem, HALTS. We know HALTS does not exist because the Halting Problem is undecidable. With the reduction proof, we'll show that if we have a Virus Detection program, DETECTVIRUS, we could use it to construct HALTS, so DETECTVIRUS must not exist either.

The main idea is to construct a machine that first executes any input program *P*, and then behaves like a virus:

MAKEVIRUS(*P*) = *P*; INFECTFILE.

Here, INFECTFILE is a program that exhibits virus behavior. Then if there exists a program, DETECTVIRUS, that solves the Virus Detection problem, we can use it to solve the Halting Problem:

HALTS(*P*) := **not**(DETECTVIRUS(MAKEVIRUS(*P*)))

That is, we could check if any program described by *P* halts but using DETECTVIRUS to check if the program that simulates *P* followed by performing an infection is a virus. If *P* describes a program that halts, then MAKEVIRUS(*P*) exhibits virus behavior since *P* eventually halts and then INFECTFILE will execute. But, if *P* does not halt, INFECTFILE will never execute since *P* never finishes.

There's one wrinkle with this argument, which is that *P* may already include a virus, but also never terminate. In that case, DETECTVIRUS(*P*; INFECTFILE) would output **True** since *P* behaves like a virus, even though *P* never terminates, leading our definition of HALTS to produce the wrong result. Without a fix for this, it could be the case that the virus detection program, DETECTVIRUS, actually exists since we have not shown that it can be used to build a correct Halting Problem program.

The good news (at least for our proof, not for the desire to have virus detectors!), is that there is a way to fix this — we just need to know there is a way to transform any program into one that has the same halting behavior but does not behave like a virus. We can do this by transforming the input program, *P*, into a program that is definitely not a virus by making all instructions that would write to files instead redirect to a shadow file system, which is destroyed after *P* finishes.

This will ensure the transformed *P* will not infect any persistent file. Doing such a redirection is straightforward. We just need to put a wrapper around system calls that would open a file to use a shadow copy of the file instead of the real file. We'll call the function that transforms all the file operations in a program, SHADOWIZEFILES. Then, we can correctly define HALTS, a program that solves the Halting Problem, using a program DETECTVIRUS that solves the Virus Detection problem:

$$\textsc{Halts}(P) := \textbf{not}(\textsc{DetectVirus}(\textsc{MakeVirus}(\textsc{ShadowizeFiles}(P))))$$

This shows the Virus Detection problem is undecidable — there is no algorithm that can correctly determine if an input program description is a virus. The specific argument we used applies for our very simple definition of a virus, but similar arguments would apply to any interesting type of malicious (or non-malicious) behavior done by a program.[2]

## 4 Impossibility is not Hopeless

*When the problem we want to solve is undecidable, what should be do?*

The theoretical result that the Virus Detection problem is undecidable doesn't mean we should give up entirely on protecting computing systems from viruses or other types of malware. But, we should be careful not to overstate the significant of the theoretical result.

First, all of theoretical results are based on idealized abstract models of computing machines. The Turing Machine model is remarkably robust, and captures well many different ways of modeling computers and our intuition about what an unbounded computer is. But, it is still an abstract model. In particular, it assumes infinite working memory (in the form of an infinite tape, in Turing's instantiation). Although modern computers have unfathomably large memories relative to human-scale experience, their memories are not infinite. No physical machine can have infinite memory, so there are Turing Machines that cannot be simulated by actual computers. Indeed, because of the finiteness of all real computers, it is in theory always possible to solve the Halting Problem for real computers by simulating the execution for the number of possible computer configurations number of steps; if the simulated computer has not halted within that number of steps, it never will since it must have repeated a configuration. This is an astronomically large number for even the tiniest computer today. The simplest computers you can buy today have at least 16 kilobytes of memory, so can be in $2^{16*8*2^{10}}$ states. This is an 81-digit number, well exceeding the number of atoms in the known universe. So, although in theory one could simulate a processor for that number of steps, in practice it is totally infeasible.

Even if we accept that the abstract model of a computer is adequate, it is important to realize that all the impossibility result proves is that it is impossible to find a virus detection program that gets the right answer for every possible input. The Halting Problem proof depends on construction one particular program for which no correct answer could be given. It does not mean that there do not exist programs which give the correct answer nearly all of the time. It could run forever on some inputs and give the correct answer on all other inputs. This would not be useful for a virus detection since we need it to always finish in a reasonable time.

Instead, it could always finish but give the wrong output for some inputs. For a virus detector, we call these wrong outputs a *false positive* when the detector outputs **True** for an input that is not a virus,

[2]The general theorem that any determining when a program exhibits any non-trivial program behavior is undecidable is *Rice's Theorem* [7].

and *false negatives* when the detector outputs **False** for an input that is a virus. The impossibility result means that any virus detector that always gives an output must be incorrect for some inputs. We could build a virus detector that has no false positives (in the extreme, by outputting **False** for all inputs), or one that has no false negatives (by always outputting **True**), but it is theoretically impossible to build one that is correct for all inputs. In practice, vendors of virus detectors make tradeoffs between having a false positive rate that is too high and annoying to customers, and missing too many real viruses.

Finally, the impossibility result does not preclude strategies that *modify* suspicious programs. Indeed, we already used the SHADOWIZEFILES transformation in our argument to transform a program that might behave like a virus into one that we know does not. This simple transformation wouldn't be a useful anti-virus transformation since it prevents useful programs from doing useful normal behaviors that make any persistent changes to files. But a more complex transformation could be designed that would allow programs to make certain types of changes to files but not others, or to observe program behaviors closely enough to recognize and disrupt known types of malicious behavior. Behavioral malware detectors take this approach, and there is no theoretical reason they could not be perfect. The challenge is defining malicious behavior precisely enough to prevent all malicious activities programs might do, without also preventing useful behaviors from benign programs.

## Acknowledgements

This paper was instigated by Melih Abdulhayoglu, based on my posted lecture slides from my Introduction to Computing course. Some of the ideas here are covered in Chapter 12 of my introductory computing textbook [5].

## References

[1] Fred Cohen. *Computer Viruses*. PhD thesis, University of Southern California, 1985.

[2] Fred Cohen. Computer Viruses: Theory and Experiments. *Computers & Security*, 6(1), 1987.

[3] Martin Davis. *The Universal Computer: the Road from Leibniz to Turing*. WW Norton & Company, 2000.

[4] David Evans. *Dori-Mic and the Universal Machine!* http://dori-mic.org/.

[5] David Evans. *Introduction to Computing: Explorations in Language, Logic, and Machines*. http://computingbook.org/.

[6] William F Gilreath and Phillip A Laplante. *Computer Architecture: A Minimalist Perspective*, volume 730. Springer Science & Business Media, 2003.

[7] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[8] Alan Mathison Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230–265, 1936–7.

Version: February 12, 2017 (see https://www.cs.virginia.edu/evans/virus for latest version)