6.857 Computer and Network Security	November 08, 2001

Lecture Notes 18 : Viruses and Worms

Lecturer: Ron Rivest — *Scribe: Chiou/Chou/Ma/Vo*

1 Topics

- Overview
- Virus Theory (Undetectable viruses)
- Obfuscation

2 Overview

This lecture covers two major topics. The first topic concerns whether it is possible to construct a program to determine whether another program (passed as input) has a given property.

The second major topic is code obfuscation, the concept of transforming an input program into a "black box" which the user can not determine the implementation details of.

3 Virus Theory (Undetectable Viruses)

We are used to working with programs that take programs as input, or produce programs as output. For example, compilers and interpreters deal with code that a user has written, and virus scanners search for particular virus programs. A few examples of these types of programs include:

- compilers and interpreters
- optimizers
- decompilers
- byte-code verifiers
- virus scanners

Today we'll be looking at programs that operate on their own code. These types of programs are known as "Self referential programs."

3.1 Self referential programs

One example of a self-referential program is a program that prints itself out:

```
char *s = ‘‘char *s = \%c\%s\%c; main() \{printf(s, 34, s, 34);\}’’;\\
main() \{ printf(s, 34, s, 34); \}\\
```

Note that 34 is decimal code for the double-quote character.

Note that the string s in the program is used both as a “format” string and as a “data” string.

The following exercise can be implemented using the same principles (and the C function sprintf):

Exercise: Write a program P that applies some function A to the text of program P.

This exercise follows the same model as the self-referential program listed above; the program applies the function A to the text of the program P. The text assigned to the string “s” includes the results of applying A to the text of program P.

$$P \equiv \left\{ \begin{array}{l} \text{char *s;} \\ \text{s = } \langle \text{ text for program P} \rangle \text{;} \\ \langle \text{ define subroutine A} \rangle \\ \text{A(s)} \end{array} \right\} P \equiv A(P) \text{ P does whatever A does on the text for program P}$$

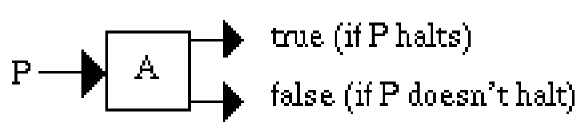
3.2 Halting problem

The halting problem is the problem of determining if a given input represents a program P that halts. (Assume P has no inputs.) One type of non-halting program is a program that enters an infinite loop of some sort.

Theorem

The halting problem in general is undecidable. This means that there is no program A that will always correctly determine whether or not a given input program P halts.

A program A that correctly solved the halting problem would behave as follows:



It would return “true” if the program P halts and “false” if the program P doesn't halt.

Proof by Contradiction

This proof demonstrates that the halting problem is undecidable.

1. Assume A exists.

2. Consider the following input P:

$$P \equiv \left\{ \begin{array}{l} \text{char *s;} \\ \text{s} = \langle \text{ text for program P} \rangle; \\ \langle \text{ define subroutine A} \rangle \\ \text{if A(s) then loop else halt;} \end{array} \right\}$$

   The program P can be displayed in simplified form as follows:

$$P \equiv \{ \text{ if A(P), then loop else halt; } \}$$

3. The program A that decides the halting problem can not correctly determine whether or not P halts. For example, if A(P) is true then P should loop. But that would make A(P) false (since P loops instead of halting). But if A(P) is false then P will halt, which would make A(s) true...

4. Therefore A either doesn't halt on input P, or else A(P) is wrong. Since we assumed that A could correctly determine whether or not P halted, our assumption must be incorrect.

Cohen used arguments similar to the above to show: There is no perfect virus scanner [1]

For the following, we define a "virus" as a program that "spreads" (e.g., reproduces, as in a worm, or infects other programs...)

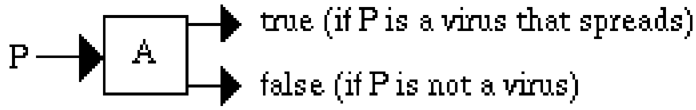
Theorem

There is no program that detects all viruses. (Virus detection is undecidable.)

Proof by contradiction

1. Assume there exists some A that correctly detects all viruses. A would take in a program "P" and determine whether or not it was a virus and would return true if it was.

[1] Cohen '87



2. Consider the following (simplified) input to A:

$$P \equiv \{ \text{ if A(P), then halt else spread } \}$$

A either loops on P or it produces some output. If it produces some output then that output is wrong, no matter whether it is true or false. Thus, there is no A that decides whether or not P is a virus.

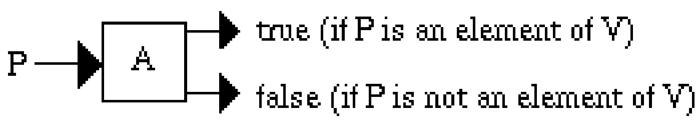
3. Every virus scanner is going to make some mistakes, either false positives or false negatives (or it may hang...)

Note: The generic definition of virus detection is the ability to detect any specific virus, rather than detect whether any given piece of code might be a virus.

Question: Can we catch any specific virus we want to?
Answer: Not if virus may be polymorphic! [2] Polymorphic viruses are viruses that change form each time they are replicated.

Theorem

There is a non-detectable polymorphic virus. A detector A that successfully caught this virus would have the following behavior:



Virus V is a maximal set of programs which will spread other copies of V. This may include spreading an altered version of the original program.

The "spread" function takes one argument (new form)
For example, spread (P') will spread an instance of P', not P.
"virus" = maximal set of programs that can spread as each other.
"detect" = detect all variations of virus.

Another way of writing this is as follows:

[2] Chess/White 2000

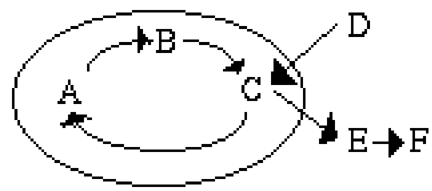
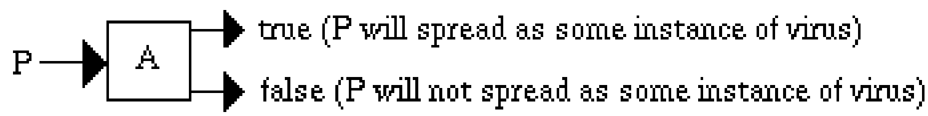


Figure 1: The set of programs in virus V that produce altered originals.



Proof

1. Consider the following on input P:

$$P \equiv \left\{ \begin{array}{l} (int)s(x)\{ \\ \langle \text{body of s} \rangle \\ \} \\ \text{if s(P) then halt} \\ \text{else spread(P' = P with body of s replaced by random program\&} \\ \text{references to P replaced by P')} \end{array} \right\}$$

2. This proof follows the same model as the previous proofs in this lecture. Assume that some function "s" exists which determines whether or not P is a virus. The program P applies the detector "s" to the input program P. If s(P) is true (i.e., the detector determines that P is a virus) then P halts. If the detector determines that P is not a virus then P spreads.

Question: Why can't you just write a program that detects whether a program writes a variable?
Answer: Whether a program writes a variable is also undecidable for similar reasons. Imagine a detector Q that determines whether a particular program writes the variable s. Now imagine a program P with the following behavior:

$$P \equiv \{\text{if Q(s) == false, then s} = \langle \text{ text for program P} \rangle ;\}$$

The variable s is only written if Q determines that the variable is not written. Thus, Q must either loop or return an incorrect value.

Question: Theoretically, doesn't that virus not exist because it has no random number generator?
Answer: No, the random number generator can be eliminated through counters.

Question: Does the set "virus" have to be cyclic?
Answer: No, the set does not have to cyclic, however it must be strongly connected.

Question: Is it true that the set of "virus" cannot be infinite, since there are limited resources?
Answer: In practice, that is true. However, such limitations are usually ignored when doing "theory".

3.3 Generalization: Rice's Theorem

Let X be some nontrivial property of the behavior of programs.
Then X is undecidable.

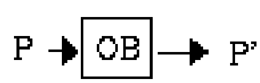
Proof

The proof of this theorem follows the same model as all of the other proofs in this lecture. Imagine some property P and some detector A which determines whether the input program I has property P. Construct I such that if A(I) is true then I does not exhibit property P, and if A(I) is false the I does exhibit property P.

Examples of some properties that can not always be determined: stack overflow, halt, divide by zero, buffer overflow, type error, equivalence to another program...

4 Code Obfuscation

Obfuscation is the technique of hiding the underlying implementation details (source code, etc.) of a particular program. For example, running an input program P through an obfuscator OB might produce output P'.



P' has exactly the same functionality as P but having P' is no better than having oracle access to P in terms of determining P's underlying implementation. P' is a "black box"; all you can learn from it is the I/O behavior.

If we had perfect obfuscators, then we could have:

1. Software protection: User has access to functionality, but not intellectual property behind it...
2. PK encryption (obfuscate symmetric key encryption program)
   (Note: original D-H paper suggested this...)
3. Homomorphic encryption (with many operations)

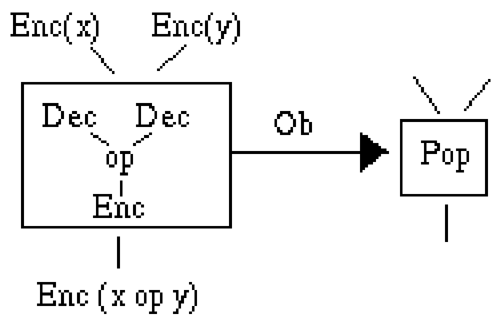


Figure 2: Obfuscators used for homomorphic encryption.

BUT Obfuscation is impossible

For any obfuscator, ∃ a family of functions for which obfuscation is impossible. If you know the details of the obfuscator, there is always a family of functions whose underlying details can be determined even after being obfuscated.

Question: Is the presumption that you know the obfuscator?
Answer: Yes, the obfuscator is a public program.

Question: Will obfuscating something twice be different from obfuscating it once?
Answer: I don't know. I haven't thought about it before.

Question: How can you prove that obfuscation is impossible? Is it complicated?
Answer: Yes, the proof is complicated. The proof is also somewhat like the proof that we just did.