last time

dynamic linking
- global offset table
- stubs
- lazy binding

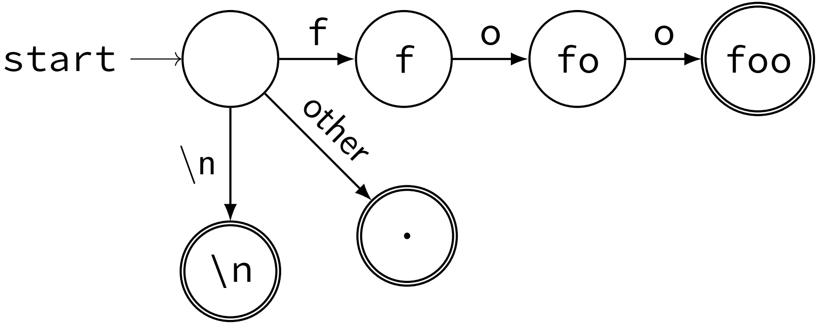
viruses calling standard library

integrity checking files
- tripwire: did it change?
- application signing
- limitations
  - malware in scripts
  - malicious software in "data" files
  - downloaded software people want to run

searching for patterns
- regular expression matching

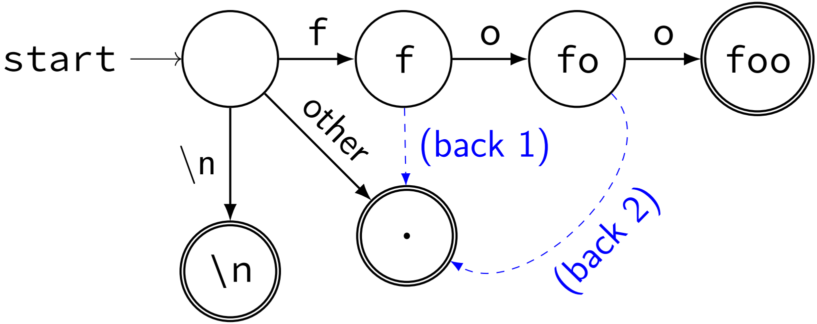
flex: state machines

```
foo      {...}
.        {...}
\n       {...}
```



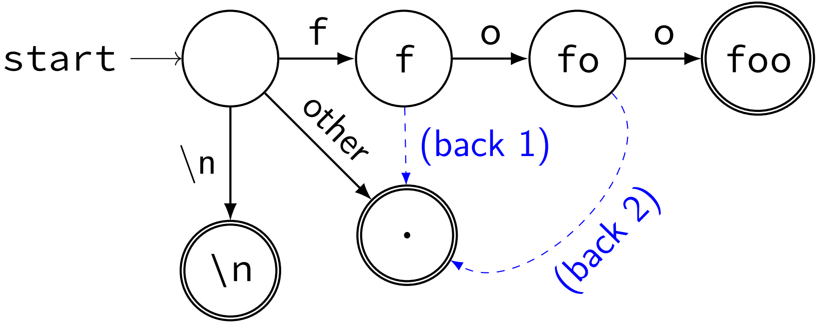
flex: state machines

```
foo     {...}
.       {...}
\n      {...}
```



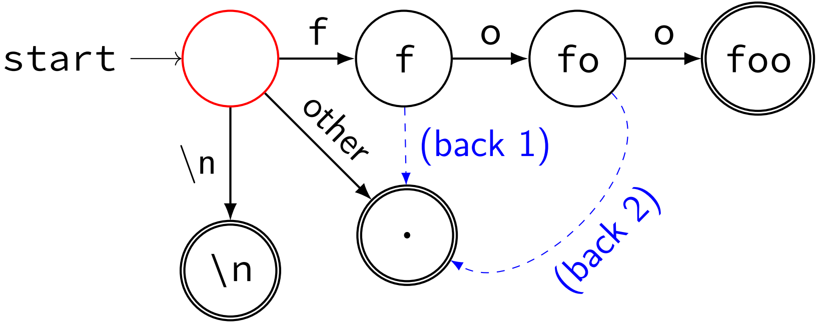
state machine matching

abfoofoabffoo



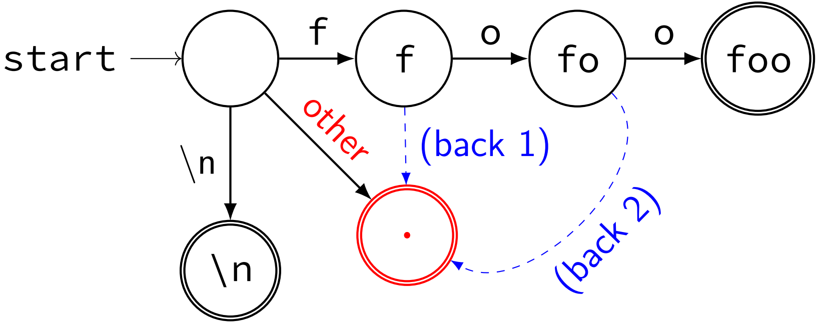
state machine matching

abfoofoabffoo



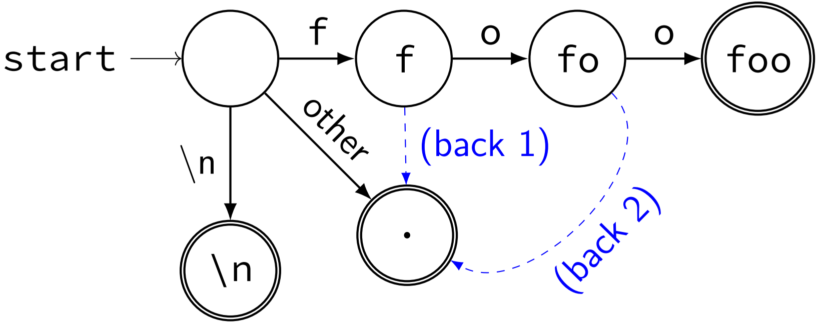
state machine matching

~~a~~bfoofoabffoo



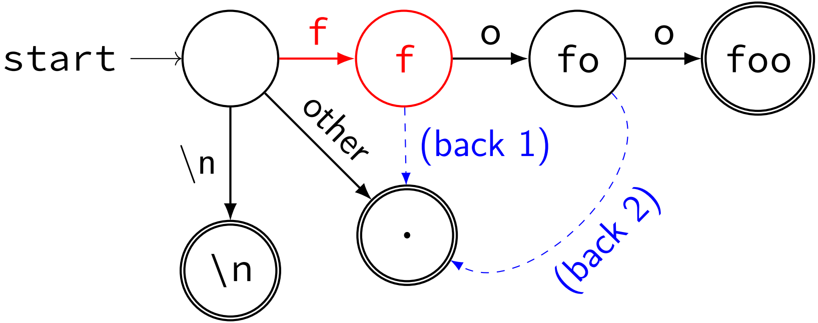
state machine matching

~~ab~~foofoabffoo



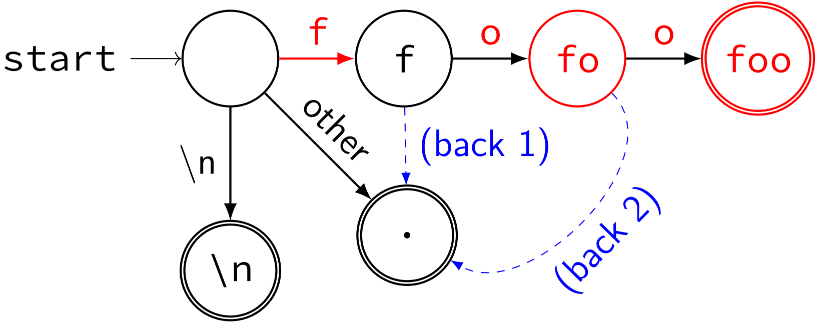
state machine matching

~~ab~~foofoabffoo



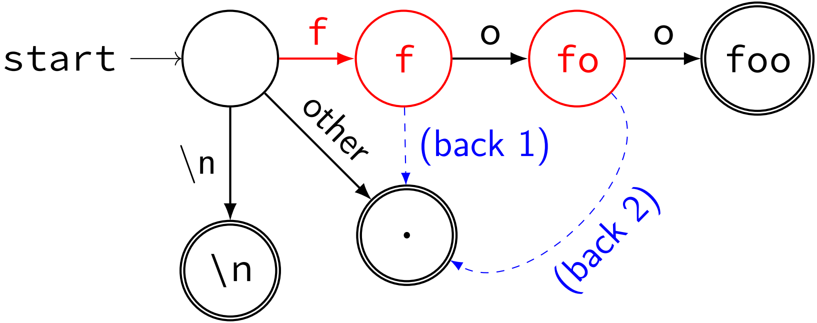
state machine matching

~~ab~~foofoabffoo



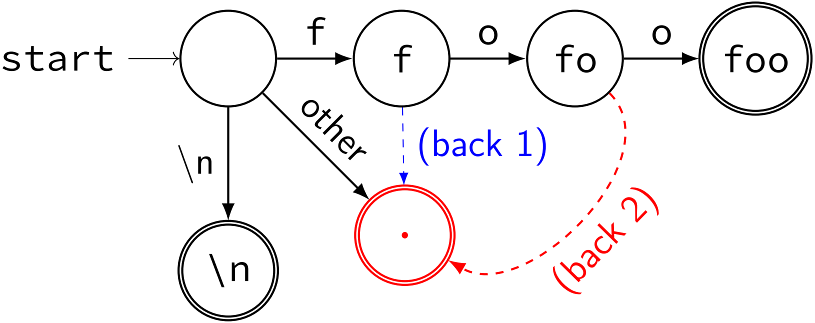
state machine matching

~~abfoo~~foabffoo



state machine matching

~~abfoo~~f~~o~~abffoo



why this?

one pass matching (except for some backtracking)

can make state machine bigger to avoid some backtracking

basically speed of file I/O

handles multiple patterns well

flexible for “special cases”

why this?

one pass matching (except for some backtracking)

can make state machine bigger to avoid some backtracking

basically speed of file I/O

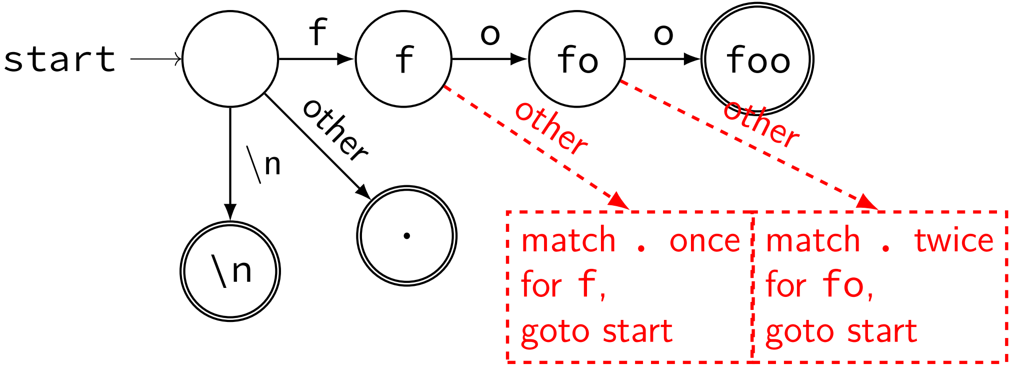
handles multiple patterns well

flexible for “special cases”

real anti-virus: probably custom pattern “engine”

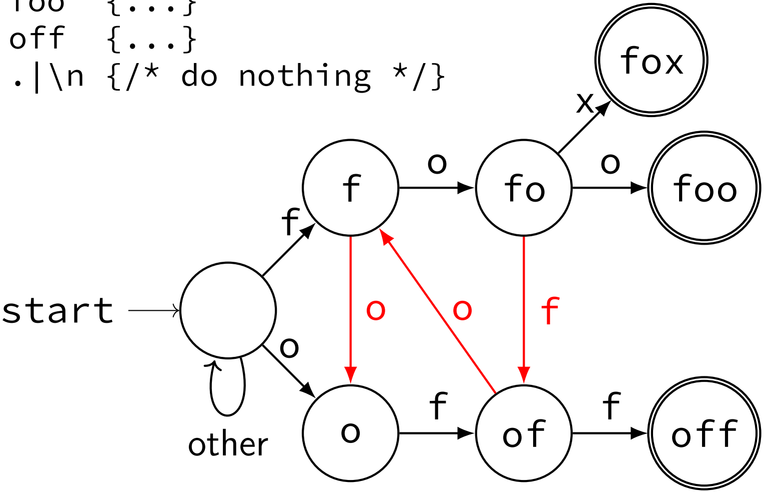
precomputing backtracking

```
foo     {...}
.       {...}
\n      {...}
```



avoiding backtracing?

```
fox   {...}
foo   {...}
off   {...}
.|\n {/* do nothing */}
```



Vienna patterns (1)

simple Vienna patterns:

```
/* bytes of fixed part of Vienna sample */
\xFC\x89\xD6\x83\xC6\x81\xc7\x00\x01\x83(etc) {
        printf("found Vienna code\n");
    }
```

Vienna patterns (2)

simple Vienna patterns:

```
/* Vienna sample with wildcards for
   changing bytes: */
/* push %CX; mov ???, %dx; cld; ... */
\x51\xBA(.|\n)(.|\n)\xFC\x89(etc) {
        printf("found Vienna code w/placeholder\n");
    }
/* mov $0x100, %di; push %di; xor %di, %di; ret */
\xBF\x00\x01\x57\x31\xFF\xC3 {
        printf("found Vienna return code\n");
    }
```

Vienna patterns (2)

simple Vienna patterns:

```
/* Vienna sample with wildcards for
   changing bytes: */
/* push %CX; mov ???, %dx; cld; ... */
\x51\xBA(.|\n)(.|\n)\xFC\x89(etc) {
        printf("found Vienna code w/placeholder\n");
    }
/* mov $0x100, %di; push %di; xor %di, %di; ret */
\xBF\x00\x01\x57\x31\xFF\xC3 {
        printf("found Vienna return code\n");
    }
```

regular expressions are flexible

for Vienna: lots of flex features we didn't need

- things being repeated variable number of times
- one of list of possible characters (bytes)
- ...

but viruses try to make pattern matching hard

good to think about what we can easily match

hard for patterns?

malware makes modificates to evade pattern matching

exercise: suppose we have a pattern for a Vienna-like virus, and a new version makes the following change. Which of the following is going to be easiest/hardest to change the pattern for?

A. inserting random number of nops every 8 non-nop instructions of virus code
B. replacing code at random offset in executable instead of appending
C. registers used for temporaries in virus code chosen at random each time the virus copies itself
D. instead of appending all the virus code, virus code now split between cavities with "loader" appended (the "loader" reforms code from the cavities and jumps to them)

making scanners efficient

lots of viruses!

- huge number of states, tables
- copies of every piece of malware pretty large

reading files is slow!

making scanners efficient

lots of viruses!

- huge number of states, tables
- copies of every piece of malware pretty large

reading files is slow!

handling volume

storing signature strings is non-trivial

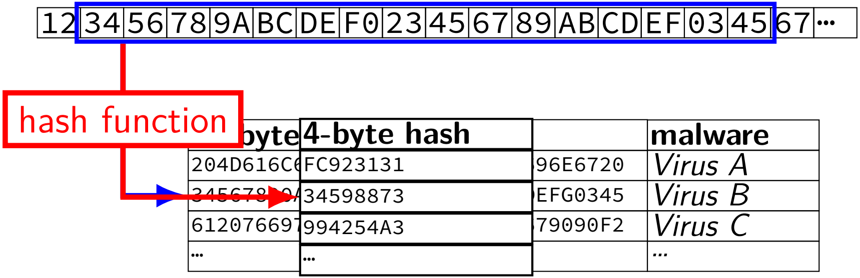
tens of thousands of states???

observation: fixed strings dominate

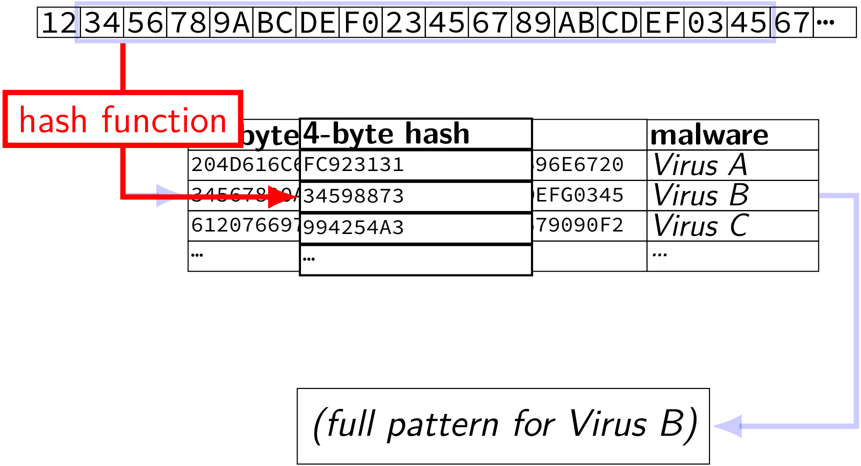
scanning for fixed strings

16-byte "anchor"	**malware**
`204D616C6963696F7573205468696E6720`	*Virus A*
`34567890ABCDEF023456789ABCDEFG0345`	*Virus B*
`612076697275732073747472696E679090F2`	*Virus C*
...	...

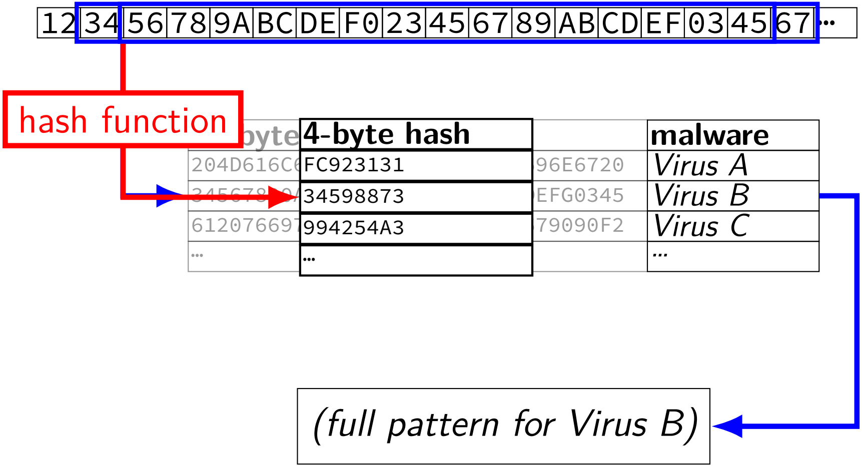
scanning for fixed strings



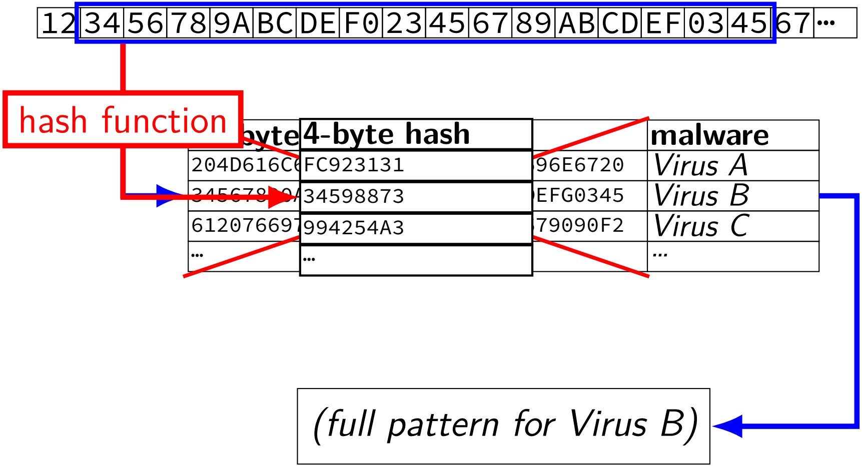
scanning for fixed strings



scanning for fixed strings



scanning for fixed strings



making scanners efficient

lots of viruses!

- huge number of states, tables
- copies of every piece of malware pretty large

reading files is slow!

the I/O problem

scanning still requires reading the whole file

can we do better?

selective scanning

check entry point and end only

- a lot less I/O, maybe

check known offsets from entry point

heuristic: is entry point close to end of file?

real signatures: ClamAV

ClamAV: open source (mostly email) scanning software

signature types:

- hash of file
- hash of contents of segment of executable
  - built-in executable, archive file parser
- fixed string
- basic regular expressions
  - wildcards, character classes, alternatives
- more complete regular expressions
  - including features that need more than state machines
- meta-signatures: match if other signatures match
- icon image fuzzy-matching

example ClamAV signatures (1)

hashes

```
4b3858c8b35e964a5eb0e291ff69ced6:78454:Xls.Exploit.Agent-4323916-1:73
7873be8fc5e052caa70fdb8f76205892:293376:Win.Trojan.Sality-93158:73
f358d77926045cba19131717a7b15dec:293376:Win.Trojan.Sality-93159:73
48d4c5294357e664bac1a07fce82ea22:450024:Win.Trojan.Sality-93160:73
e4b8442638b3948ab0291447affa6790:293376:Win.Trojan.Sality-93161:73
df36dc207b689a73ab9cf45a06fb71b0:232448:Win.Trojan.Sality-93162:73
baaeeabc7f4be3199af3d82d10c6b39f:293376:Win.Trojan.Sality-93163:73
...
```

example ClamAV signatures (2)

simple regular expressions (with hex, different syntax than flex)...

```
Win.Trojan.Vienna-1:0:*:5051e8??00{1-255}5b83eb??fc8d37bf0001b90300f3a48bf3558bec83
Win.Trojan.Vienna-2:0:*:be000356c3*50be????8bd6fcb90500bf0001f3a48bfab430cd21
Win.Trojan.Vienna-3:0:*:50ba????8bf283c60090bf0001b90300fcf3a48bfab430cd213c02
Win.Trojan.Vienna-4:0:*:b440b900048bd681eac102cd21721f3d
Win.Trojan.Vienna-5:0:*:b904048bd681ea130352515350b4
...
Win.Trojan.Vienna-129:0:*:51b89b03cd213d01017503e9????ba6d03fc8bf283c60a90b90300bf0
```

example ClamAV signatures (3)

'logical' signatures: mutliple regexes together:

```
Andr.Trojan.Pjapps-58;Engine:51-255,
    Container:CL_TYPE_ZIP,Target:0;
    (6&0&1&(2|3)&(4|5));  // expected patterns of below
    3a39303333; // pattern 0
    696d6569;   // pattern 1
    616e64726f69642e6c6f67; // pattern 2
    77696e646f772e6c6f67;   // pattern 3
    4e6f6b69614e373631302d31; // pattern 4
    336c676f6167646d66656a656b67666f73397431356368 6f6a6d; // pattern 5
    0:646578 // pattern 6: "0:" means must be found at beginning of file
```

playing cat

harder to fool ways of detecting malware?

goal: small changes to malware preserve detection

ideal: detect new malware

detecting new malware

look for anomalies

- patterns of code that real executables “won’t” have

identify bad behavior

viruses and executable formats

header: machine type, file type, etc.
program header: "segments" to load (also, some other information)
segment 1 data
segment 2 data

viruses and executable formats

header: machine type, file type, etc.
program header: "segments" to load (also, some other information) **length edited by virus**
segment 1 data
segment 2 data **virus code + new entry point?**

viruses and executable formats

header: machine type, file type, etc.
program header: "segments" to load (also, some other information) **length edited by virus**
segment 1 data
segment 2 data **virus code + new entry point?**

heuristic 1: is entry point in last segment?
(segment usually not code)

viruses and executable formats

header: machine type, file type, etc.
program header: "segments" to load (also, some other information) **new segment added by virus**
segment 1 data
segment 2 data
segment 3 data — virus segment

viruses and executable formats

header: machine type, file type, etc.
program header: "segments" to load (also, some other information) **new segment added by virus**
segment 1 data
segment 2 data
segment 3 data — virus segment

heuristic 1: is entry point in last segment?
(segment usually not code)

viruses and executable formats

header: machine type, file type, etc.

program header: "segments" to load
(also, some other information)
new segment added by virus

segment 1 data

segment 2 data

segment 3 data — virus segment

heuristic 2: did virus mess up header?
(e.g. do sizes used by linker but not loader disagree)
section names disagree with usage?

defeating entry point checking

insert jump in normal code section, set as entry-point

add code to first section instead (perhaps insert new section at beginning)

defeating entry point checking

insert jump in normal code section, set as entry-point

add code to first section instead (perhaps insert new section at beginning)

“dynamic” heuristic: run code in VM, see if switches sections

heuristics: library calls

dynamic linking — functions called by name

how do viruses add to dynamic linking tables?

- often don't! — instead dynamically look-up functions
- if do — could mess that up/lots of code

heuristic: look for API function name strings

evading library call checking

modify dynamic linking tables

- probably tricky to add new entry

reimplement library call manually

- Windows system calls not well documented, change

hide names

evading library call checking

modify dynamic linking tables

- probably tricky to add new entry

reimplement library call manually

- Windows system calls not well documented, change

hide names

hiding library call names

common approach: store hash of name

runtime: read library, scan list of functions for name

bonus: makes analysis harder

behavior-based detection

things malware does that other programs don't?

basic idea: run in virtual machine; and/or monitor all programs

behavior-based detection

things malware does that other programs don't?

modify system files

modifying existing executables

open network connections to lots of random places

…

basic idea: run in virtual machine; and/or monitor all programs

hooking

hooking — getting a 'hook' to run on (OS) operations

- e.g. creating new files
- e.g. modifying executable files

ideal mechanism: OS support

less ideal mechanism: change library loading

- e.g. replace 'open', 'fopen', etc. in libraries

less ideal mechanism: replace OS exception (system call) handlers

- very OS version dependent

less ideal mechanism: debugger support

hooking

hooking — getting a 'hook' to run on (OS) operations

- e.g. creating new files
- e.g. modifying executable files

ideal mechanism: OS support

less ideal mechanism: change library loading

- e.g. replace 'open', 'fopen', etc. in libraries

less ideal mechanism: replace OS exception (system call) handlers

- very OS version dependent

less ideal mechanism: debugger support

Microsoft Sign in

Hardware Dev Center

Table of contents

What Is a File System Filter Driver?

Last Updated: 1/24/2017

IN THIS ARTICLE +

A *file system filter driver* is an optional driver that adds value to or modifies the behavior of a file system. A file system filter driver is a kernel-mode component that runs as part of the Windows executive.

A file system filter driver can filter I/O operations for one or more file systems or file system volumes. Depending on the nature of the driver, *filter* can mean *log*, *observe*, *modify*, or even *prevent*. Typical applications for file system filter drivers include antivirus utilities, encryption programs, and hierarchical storage management systems.

hooking

hooking — getting a 'hook' to run on (OS) operations

- e.g. creating new files
- e.g. modifying executable files

ideal mechanism: OS support

less ideal mechanism: change library loading

- e.g. replace 'open', 'fopen', etc. in libraries

less ideal mechanism: replace OS exception (system call) handlers

- very OS version dependent

less ideal mechanism: debugger support

changing library loading

e.g. install new library — or edit loader, but …

not everything uses library functions

what if your wrapper doesn't work exactly the same?

hooking

hooking — getting a 'hook' to run on (OS) operations

- e.g. creating new files
- e.g. modifying executable files

ideal mechanism: OS support

less ideal mechanism: change library loading

- e.g. replace 'open', 'fopen', etc. in libraries

less ideal mechanism: replace OS exception (system call) handlers

- very OS version dependent

less ideal mechanism: debugger support

changing exception call handlers (1)

OS data structure tells hardware where program requests go

simpliest mechanism: edit that data structure

- and save a copy of what was there before

point to your code

- and call what was there before after behavior check

heuristics example: DREBIN paper

from 2014 research paper on Android malware: Arp et al, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket"

primary contribution of paper: big dataset of malware

but tried to detect malware, too...

features from applications (without running):

- hardware requirements
- requested permissions
- whether it runs in background, with pushed notifications, etc.
- what API calls it uses
- network addresses

detect dynamic code generation explicitly

statistics (i.e. machine learning) to determine score

heuristics example: DREBIN paper

advantage: Android uses Dalvik bytecode (Java-like)

high-level "machine code"

much easier/more useful to analyze

accuracy?

tested on 131k apps, 94% of malware, 1% false positives

versus best commercial: 96%, $< 0.3\%$ false positives

(probably has explicit patterns for many known malware samples)

...but

statistics: training set needs to be typical of malware

cat-and-mouse: what would attackers do in response?

machine learning and adversaries

I don't like most ML-based approaches to malware detection

problem: most machine learning not designed to deal with adversaries

attack: find factors used to ID benign programs

- do all of them as much as possible
- inquiry: what might they be in DREBIN case?

attack: provide many malware samples with benign weird behavior

- machine learning uses weird behavior to identify malware
- may lower effectiveness on 'normal' malware

anti-anti-virus

defeating signatures:

avoid things compilers/linkers never do

make analysis harder

- takes longer to produce signatures
- takes longer to produce “repair” program
- may evade attempts to automate analysis

make changing viruses

- make any one signature less effective

some terms

armored viruses

viruses designed to make analysis harder

metamorphic/polymorphic/oligomorphic viruses

viruses that change their code each time

different terms — different types of changes (later)

obfuscation, generally

malware often *obfuscates* (obscures) its code

several reasons for this

- prevent their from being signatures
- make analysis more difficult
- prevent others from modifying+copying

note: many of these technique sometimes employed by commercial software

- intended to prevent copying/reverse-engineering

Tigress as example of obfuscation

Tigress — researcher developer obfuscation tool

`https://tigress.wtf`

includes many *transformations* typical of real-world obfuscation

we'll talk about the ideas behind many of them

future assignment: modify code obfuscated with Tigress

example Tigress transformations

we'll look at some simple ones Tigress provides

I'm showing you the pattern,
not the actual code Tigress generates

Tigress: provided transform: Merge

```
void foo(int a) { code for foo }
void bar(int a) { code for bar }

... foo(x) + bar(y) ...
```

```
void foo_bar(int s, int a) {
    if (s == 0) {
        code for foo
    } else {
        code for bar
    }
}

... foo_bar(0, x) + foo_bar(1, y) ...
```

Tigress: provided transform: Split

```
void foo(int a, int b) {
    int x = ...;
    code for foo part 1
    code for foo part 2
}
```

```
void foo1(int *a, int *b, int *x) {
    code for foo part 1
}
void foo2(int *a, int *b, int *x) {
    code for foo part 2
}
void foo(int a, int b) {
    int x;
    foo1(&a,&b,&x); foo2(&a,&b,&x);
}
```

Tigress: example transform: Flatten

```
void foo() {
    A;
    if (X) {
        B;
    } else {
        C;
    }
    D;
}
```

```
void foo() {
    int s = 0;
    for (;;) {
        switch(s) {
        case 0: A; s = X ? 1 : 2; break;
        case 1: B; s = 3; break;
        case 2: C; s = 3; break;
        case 3: D; return;
        }
    }
}
```

transformations so far?

all can be combined!

annoying for analysis

hard to do without unobfuscated code

- can't easily be redone/changed by self-replicating malware

probably more distinctive than original code for signatures

- (just match the transformed version since it won't change often)

next topic: transformations to avoid signatures

- (Tigress supports those, but not our primary examples)

obfuscation versus analysis

which of these does obfuscation seem most/least likely to hamper doing?

A. determining what remote servers some malware contacts

B. determining a password the malware requires to access extra functionality

C. accessing extra functionality in the malware protected by a password

D. determining whether the malware will behave differently based on the time

recall: library calls in viruses

viruses making library calls

- can't use normal dynamic linker stuff

common solution: search by name:

```
char *names[] = GetFunctionNamesFrom("kernel32.dll");
for (int i = 0; i < numFunctions; ++i) {
    if (strcmp(names[i], "GetFileAttributesA") == 0) {
        return functions[i];
    }
}
```

problem: legit application code won't do this

easy to look for string 'GetFileAttributesA'

searching for hashes

```c
char *functionNames[] = GetFunctionsFromStandardLibrary();
/* 0xd7c9e758 = hash("GetFileAttributesA") */
unsigned hashOfString = 0xd7c9e758;
for (int i = 0; i < num_functions; ++i) {
    unsigned functionHash = 0;
    for (int j = 0; j < strlen(functionNames[i]); ++j) {
        functionHash = (functionHash * 7 +
                        functionNames[i][j]);
    }
    if (functionHash == hashOfString) {
        return functions[i];
    }
}
```

encrypted(?) data

```
char obviousString[] =
    "Please open this 100%"
    " safe attachment";
char lessObviousString[] =
    "oSZ^LZ\037POZQ\037KWVL\037\016\017"
    "\017\032\037L^YZ\037^KK^\\WRZQK";
for (int i = 0; i < sizeof(lessObviousString) - 1; ++i) {
    lessObviousString[i] =
        lessObviousString[i] ^ '?';
}
```

encrypted data and signatures

get rid of some easy signatures

especially if ‘key’ changes or hashes used

but not enough:

decryption code is very distinctive

encrypted data and signatures

get rid of some easy signatures

- especially if ‘key’ changes or hashes used

but not enough:

- decryption code is very distinctive

can we do better with this “encryption” idea?

encrypted(?) viruses

```
char encrypted[] = "\x12\x45...";
char key[] = "...";
virusEntryPoint() {
    decrypt(encrypted, key);
    goto encrypted;
}
decrypt(char *buffer, char *key) {...}
```

choose a new key each time!

not good encryption — key is there

sometimes mixed with compression

encrypted viruses: no signature?

`decrypt` is a pretty good signature

still need to a way to disguise that code

how about analysis? how does one analyze this?

- one way: use a debugger, stop before goto

backup slides

regular expressions

one method of representing patterns like this:
regular expressions (regexes)

restricted language allows very fast implementations

especially when there's a long list of patterns to look for

homework assignment next week

regular expressions: implementations

multiple implementations of regular expressions

we will target: flex, a parser generator

simple patterns

alphanumeric characters match themselves

`foo`:

- matches exactly `foo` only
- does not match `Foo`
- does not match `foo␣`
- does not match `foobar`

backslash might be needed for others

`C\+\+`

- matches exactly `C++` only

metachars (1)

special ways to match characters

`\n`, `\t`, `\x3C`, …— work like in C

`[b-fi]` — `b` or `c` or `d` or `e` or `f` or `i`

`[^b-fi]` — any character but `b` or `c` or …

`.` — any character except newline

`(.|\n)` — any character

metachars (2)

`a*` — zero or more as:

(empty string), a, aa, aaa, ...

`a{3,5}` — three to five as:

aaa, aaaa, aaaaa

`(abc){3,5}` — three to five abcs: (“grouping”)

abcabcabc, abcabcabcabc, abcabcabcabcabc

`ab|cd`

ab, cd

`(ab|cd){2}` — two ab-or-cds:

abab, abcd, cdab, cdcd

metachars (3)

`\xAB` — the byte `0xAB`

`\x00` — the byte `0x00`

flex is designed for text, handles binary fine

`\n` — newline (and other C string escapes)

example regular expressions

match words ending with `ing`:

```
[a-zA-Z]*ing
```

match C `/* ... */` comments:

```
/\*([^*]|\*[^/])*\*/
```

flex

flex is a regular expression matching tool

intended for writing parsers

generates C code

parser function called `yylex`

flex example

```
          int num_bytes = 0, num_lines = 0;
          int num_foos = 0;
%%
foo       {
            num_bytes += 3;
            num_foos += 1;
          }
.         { num_bytes += 1; }
\n        { num_lines += 1; num_bytes += 1; }
%%
int main(void) {
    yylex();
    printf("%d bytes, %d lines, %d foos\n",
           num_bytes, num_lines, num_foos);
}
```

flex example

```
        int num_bytes = 0, num_lines = 0;
        int num_foos = 0;
%%
foo     {
          num_bytes += 3;
          num_foos += 1;
        }
.       { num_bytes += 1; }
\n      { num_lines += 1; num_bytes += 1; }
%%
int main(void) {
    yylex();
    printf("%d bytes, %d lines, %d foos\n",
           num_bytes, num_lines, num_foos);
}
```

three sections

flex example

```
        int num_bytes = 0, num_lines = 0;
        int num_foos = 0;
%%
foo     {
          num_bytes += 3;
          num_foos
        }
.       { num_bytes += 1; }
\n      { num_lines += 1; num_bytes += 1; }
%%
int main(void) {
    yylex();
    printf("%d bytes, %d lines, %d foos\n",
           num_bytes, num_lines, num_foos);
}
```

first — declarations for later C code in output file

flex example

```
        int num_bytes = 0, num_lines = 0;
        int num_fo
%%
foo     {
          num_bytes += 3;
          num_foos += 1;
        }
.       { num_bytes += 1; }
\n      { num_lines += 1; num_bytes += 1; }
%%
int main(void) {
    yylex();
    printf("%d bytes, %d lines, %d foos\n",
           num_bytes, num_lines, num_foos);
}
```

patterns, code to run on match
as parser: return “token” here

flex example

```
        int num_bytes = 0, num_lines = 0;
        int num_foos = 0;
%%
foo     {
          num_bytes += 3;
          num_foos += 1;
        }
.       { num_bytes += 1; }
\n      { num_lines += 1; num_bytes += 1; }
%%
int main(void) {
    yylex();
    printf("%d bytes, %d lines, %d foos\n",
           num_bytes, num_lines, num_foos);
}
```

extra code to include

flex: matched text

```
%%
[aA][a-z]* {
                printf("found a-word '%s'\n",
                       yytext);
           }
(.|\n)     {} /* default rule: would output text */
%%
int main(void) {
    yylex();
}
```

flex: matched text

yytext — text of matched thing

```
%%
[aA][a-z]* {
                printf("found a-word '%s'\n",
                       yytext);
           }
(.|\n)     {} /* default rule: would output text */
%%
int main(void) {
    yylex();
}
```

flex: definitions

```
A           [aA]
LOWERS      [a-z]
ANY         (.|\n)
%%
{A}{LOWERS}* {
                printf("found a-word '%s'\n",
                       yytext);
             }
{ANY}        {} /* default rule would
                   output text */
%%
int main(void) {
    yylex();
}
```

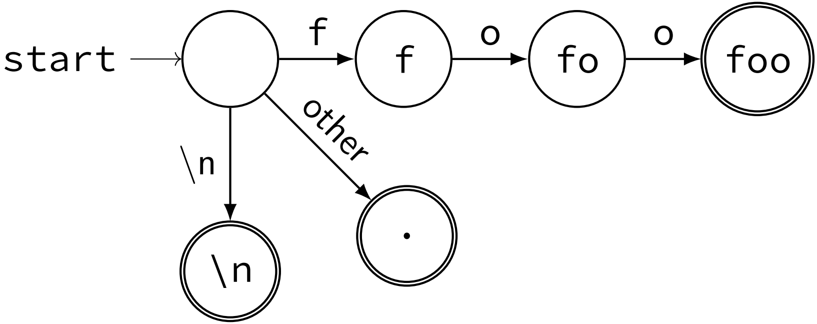
flex: definitions

```
A             [aA]
LOWERS        [a-z]
ANY           (.|\n)
%%
{A}{LOWERS}* {
                printf("found a-word '%s'\n",
                       yytext);
             }
{ANY}        {} /* default rule would
                   output text */
%%
int main(void) {
    yylex();
}
```

definitions of common patterns included later

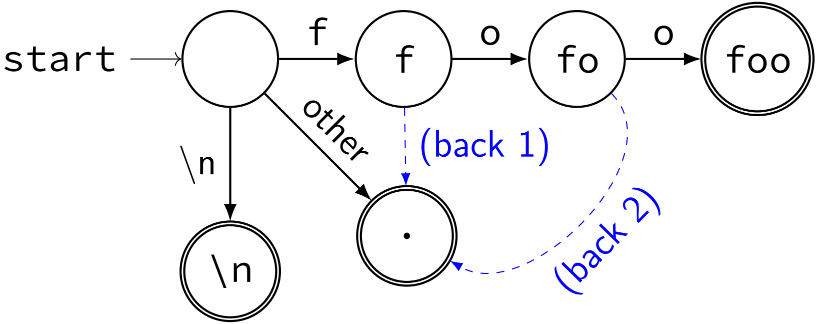
flex: state machines

```
foo        {...}
.          {...}
\n         {...}
```



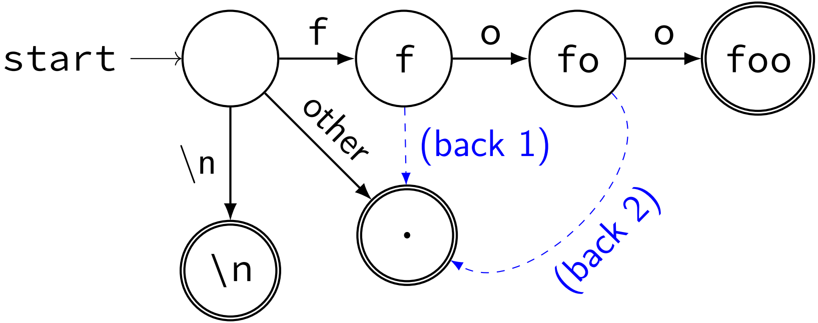
flex: state machines

```
foo     {...}
.       {...}
\n      {...}
```



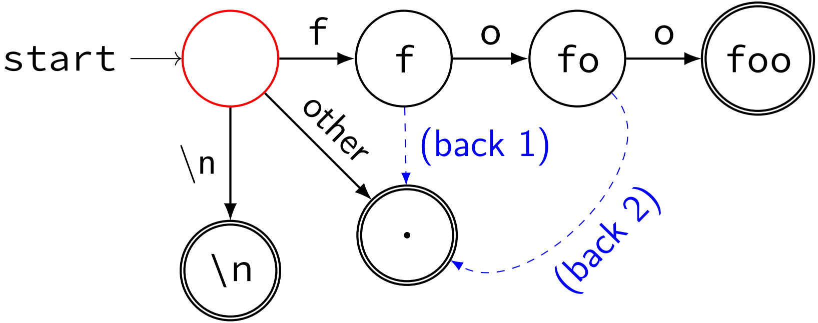
state machine matching

abfoofoabffoo



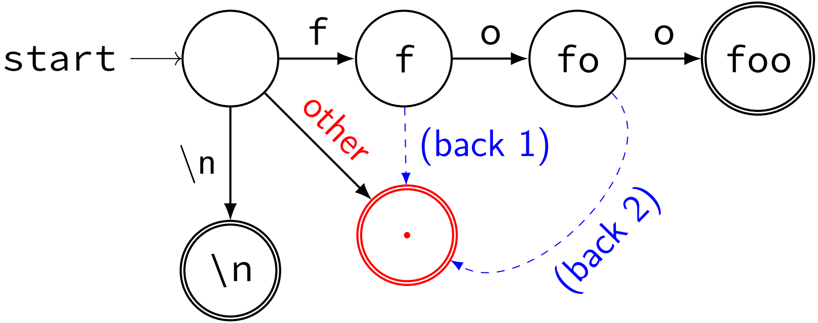
state machine matching

abfoofoabffoo



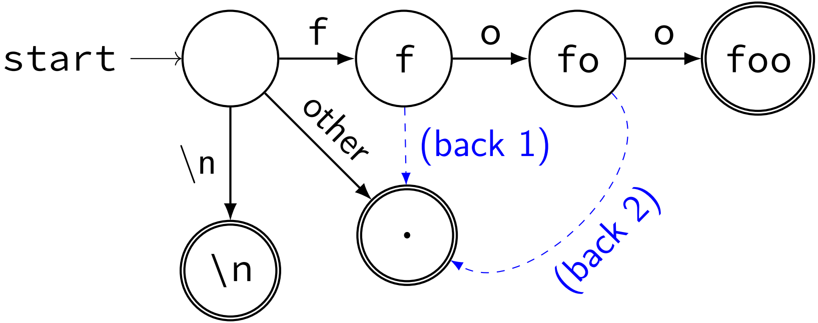
state machine matching

~~a~~bfoofoabffoo



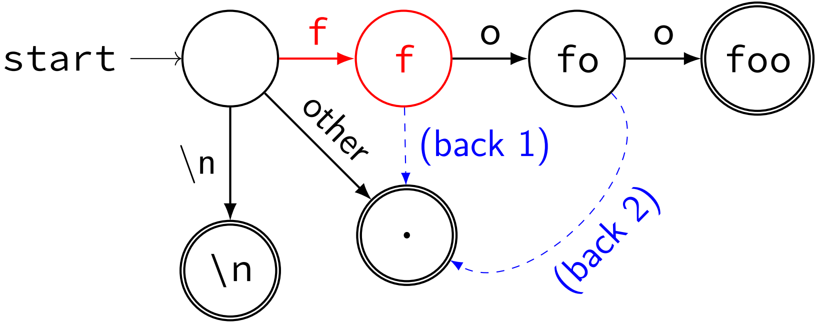
state machine matching

~~ab~~foofoabffoo



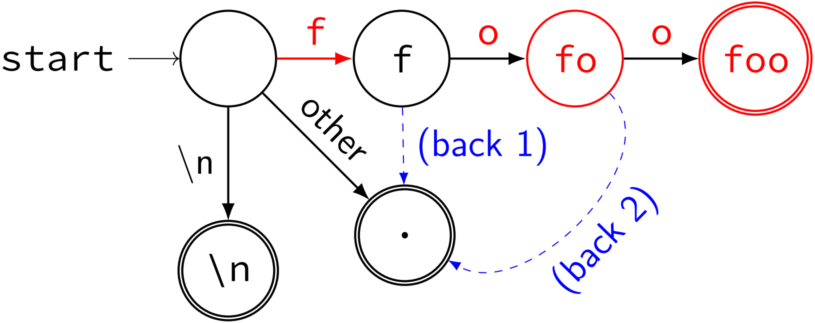
state machine matching

~~ab~~foofoabffoo



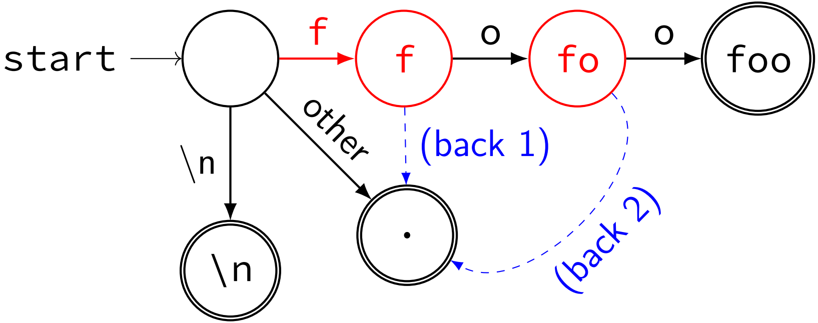
state machine matching

~~ab~~foofoabffoo



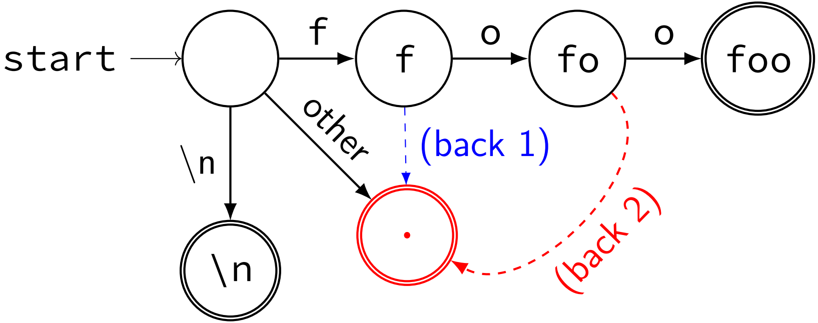
state machine matching

~~abfoo~~foabffoo



state machine matching

~~abfoo~~f~~o~~abffoo



flex states (1)

```
%x str
%%
\"              { BEGIN(str); }
<str>\"         { BEGIN(INITIAL); }
<str>foo        { printf("foo in string\n"); }
foo             { printf("foo out of string\n"); }
<INITIAL,str>(.|\n) {}
%%
int main(void) {
    yylex();
}
```

flex states (1)

```
%x str
%%
\"              { BEGIN(str); }
<str>\"         { BEGIN(INITIAL); }
<str>foo        { printf("foo in string\n"); }
foo             { printf("foo out of string\n"); }
<INITIAL,str>(.|\n) {}
%%
int ma
    yy
}
```

declare "state" to track

which state determines what patterns are active

flex states (1)

```
%x str
%%
\"              { BEGIN(str); }
<str>\"         { BEGIN(INITIAL); }
<str>foo        { printf("foo in string\n"); }
foo             { printf("foo out of string\n"); }
<INITIAL,str>(.|\n) {}
%%
int main(void) {
    yylex();
}
```

flex states (2)

```
%s afterFoo
%%
<afterFoo>foo    { printf("later foo\n"); }
foo              {
                   printf("first foo\n");
                   BEGIN(afterfoo);
                 }
(.|\n)  {}
%%
int main(void) {
    yylex();
}
```

flex states (2)

declare non-exclusive state

```
%s afterFoo
%%
<afterFoo>foo     { printf("later foo\n"); }
foo               {
                    printf("first foo\n");
                    BEGIN(afterfoo);
                  }
(.|\n)  {}
%%
int main(void) {
    yylex();
}
```

finding packers

easiest way to decrypt self-decrypting code — run it!

solution: virtual machine in antivirus software

makes antivirtualization/emulation more important

finding packers with VM

run program in VM for a while

- how long?

then scan memory for known patterns

or detect jumping to written memory

rootkits

rootkit — priviliged malware that hides itself

same ideas as these anti-anti-virus techniques

chkrootkit

`chkrootkit` — Unix program that looks for rootkit signs

how?

- tell-tale strings in system programs
- overwritten entries in system login records
- known suspicious filenames

after scanning — disinfection

antivirus software wants to repair

requires specialized scanning

- no room for errors
- need to identify all
- need to find relocated bits of code