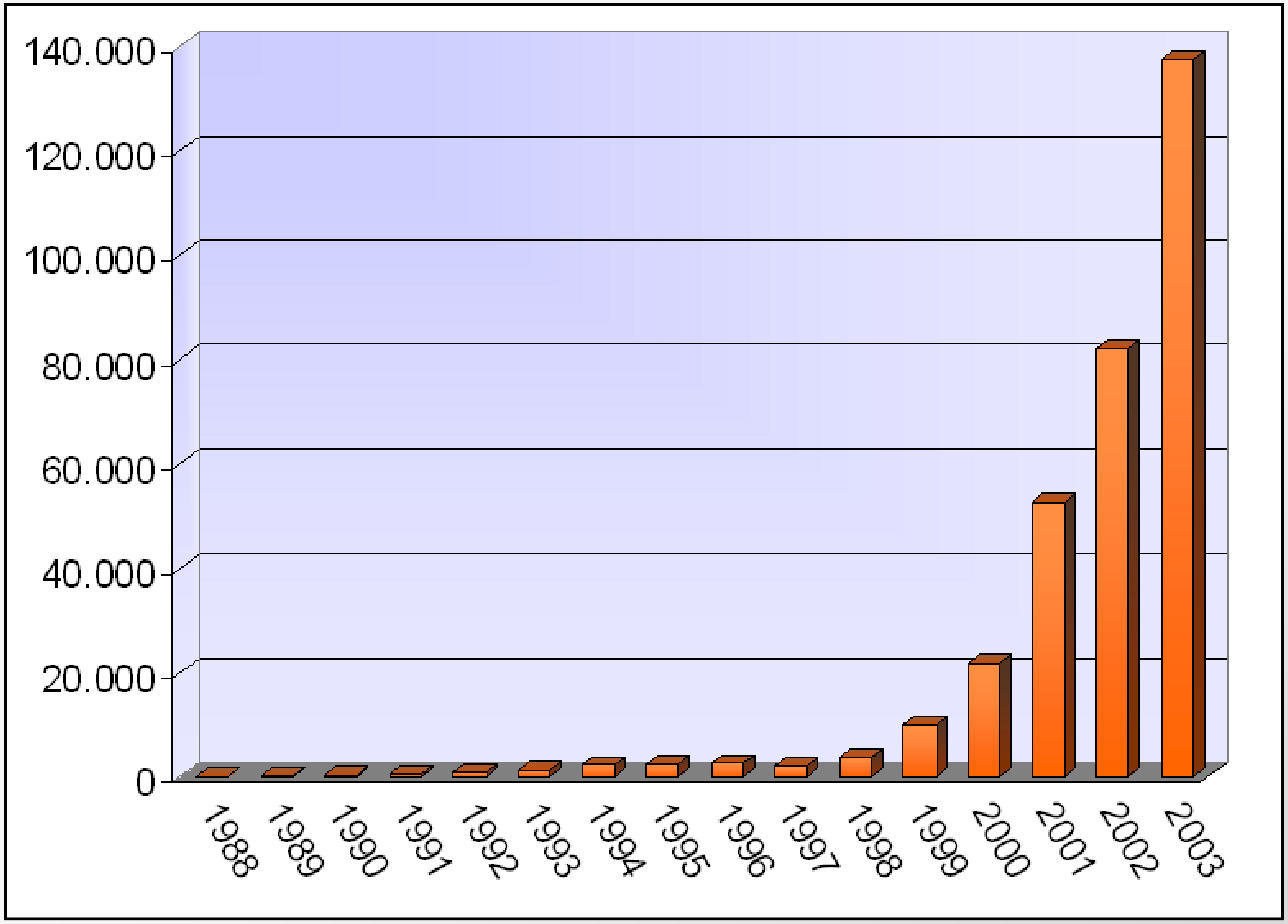
Detecting Malicious Code by Model Checking

Johannes Kinder, Stefan Katzenbeisser,
Christian Schallhart, Helmut Veith.

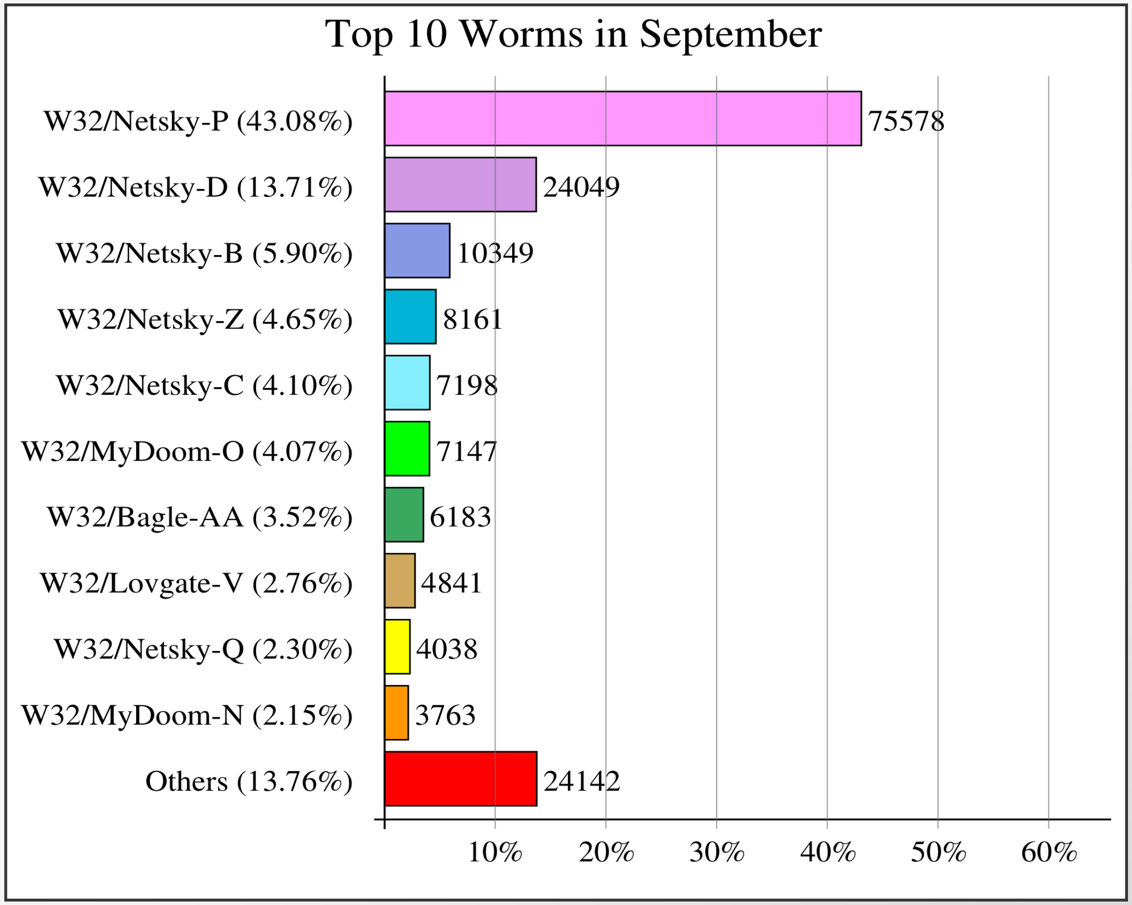
Conference on Detection of Intrusions and Malware
& Vulnerability Assessment, DIMVA 2005

Computer Security Incidents



Computer Security Incidents from 1988-2003 (Source: CERT)

E-Mail Worms – Prevalence

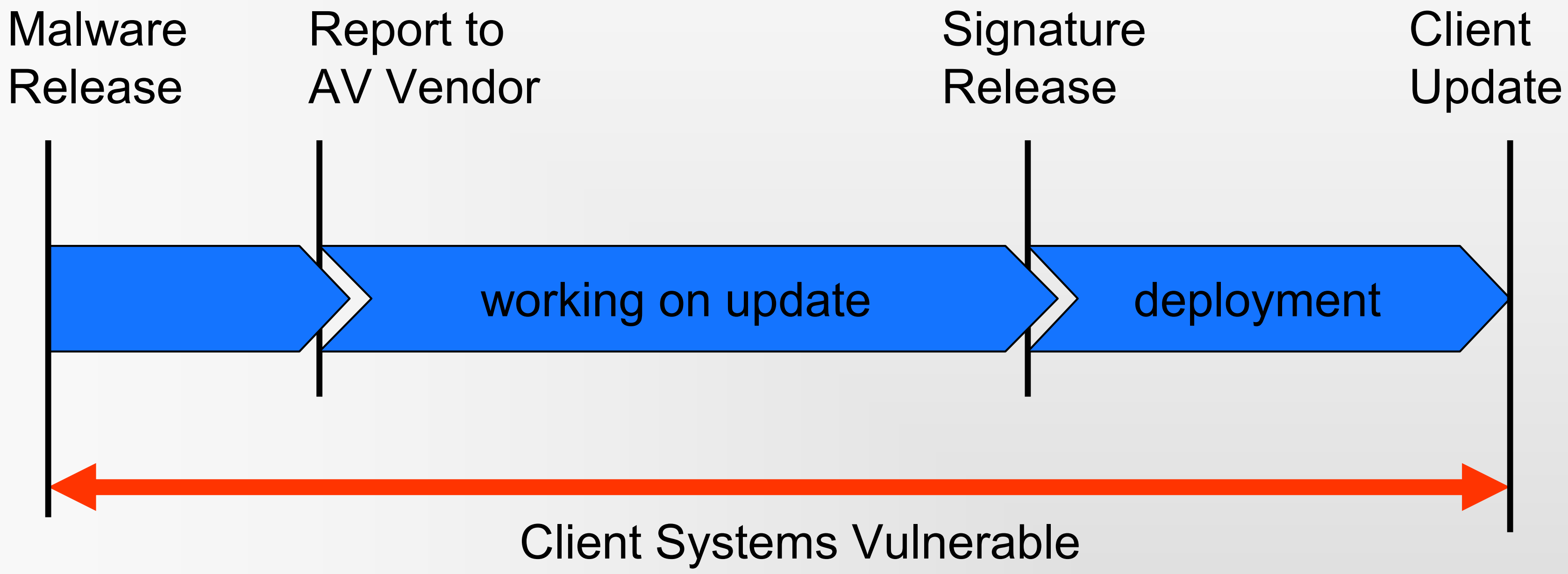

Top 10 Worms in September

Computer worms in incoming e-mails at the Department of Computer Science of the TUM in September 2004.

E-Mail Worms – Facts

- Predominantly variants of existing worms
  - Currently 200 new threats per month (Symantec)
  - More than 30 variants of NetSky, up to 3 in one day
  - Source code often widely distributed
  - ‘Script-Kiddies’
  - Variants differ only slightly in terms of functionality
  - Binary worm code can be highly different (compiler settings, executable packers)
- Timely updates to virus detectors are critical

Window of Vulnerability



In case of the Sober.C worm, this timespan ranged from 10 hours up to 4 days! (Source: Virus Bulletin, 02/04)

Detection Methods

- Signature Matching
  - Regular expressions
  - Fast and reliable
  - Not mutation tolerant (Christodorescu, Jha 2003)
- Dynamic Analysis
  - Limited timespan, not all execution paths
  - Useful for monitoring (IDS)
- Static Analysis
  - Verification of possible behavior
  - Relies on disassembly

Model Checking

- Well proven verification method
- Classically used for verifying properties such as Fairness and Liveness in distributed systems
- Verifies whether a model obeys a specification
  - Models are given as labeled transition systems
  - Specifications are given in temporal logics (e.g. CTL or LTL)

Example for Fairness:

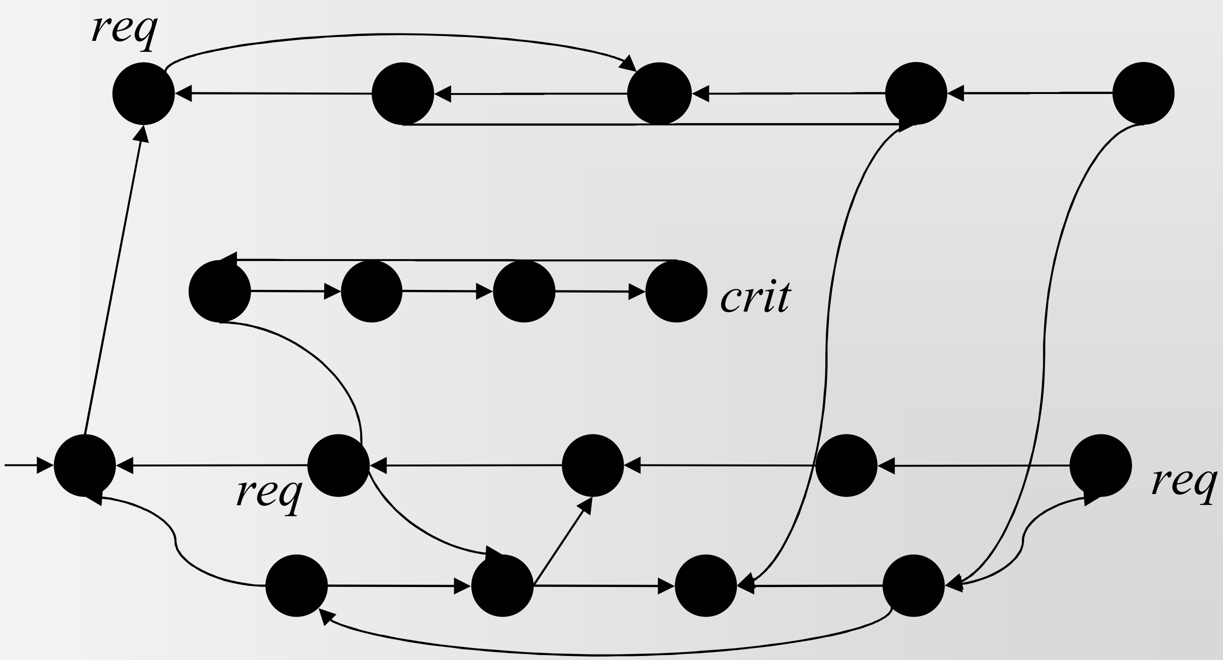
“Whenever a process requests to enter its critical area, it is eventually allowed to do so”

Model Checking – Example

- CTL specification of Fairness:

$$\boxed{\mathbf{AG}\ (req \rightarrow \mathbf{AF}\ crit)}$$

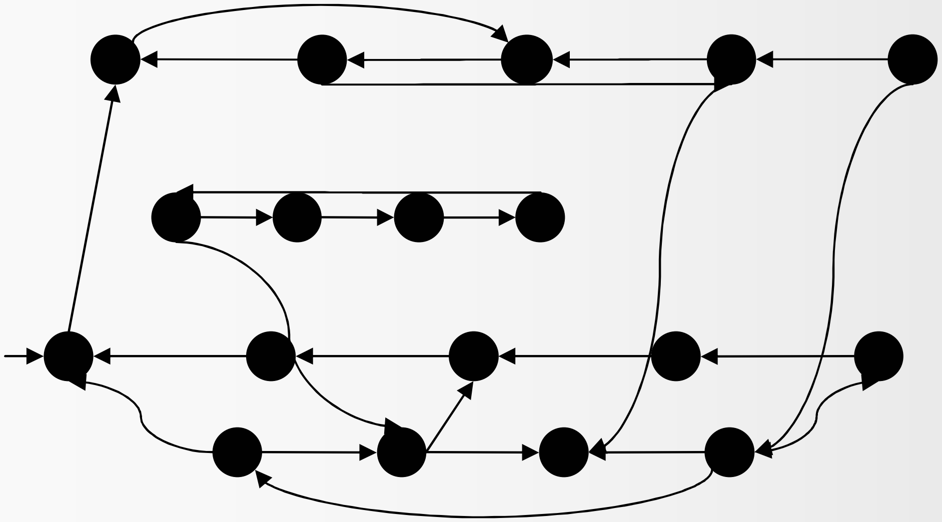
- Model:



Malicious Code Detection

- Specification of malicious behavior
- Model extraction from executable machine code
- Verification by Model Checking

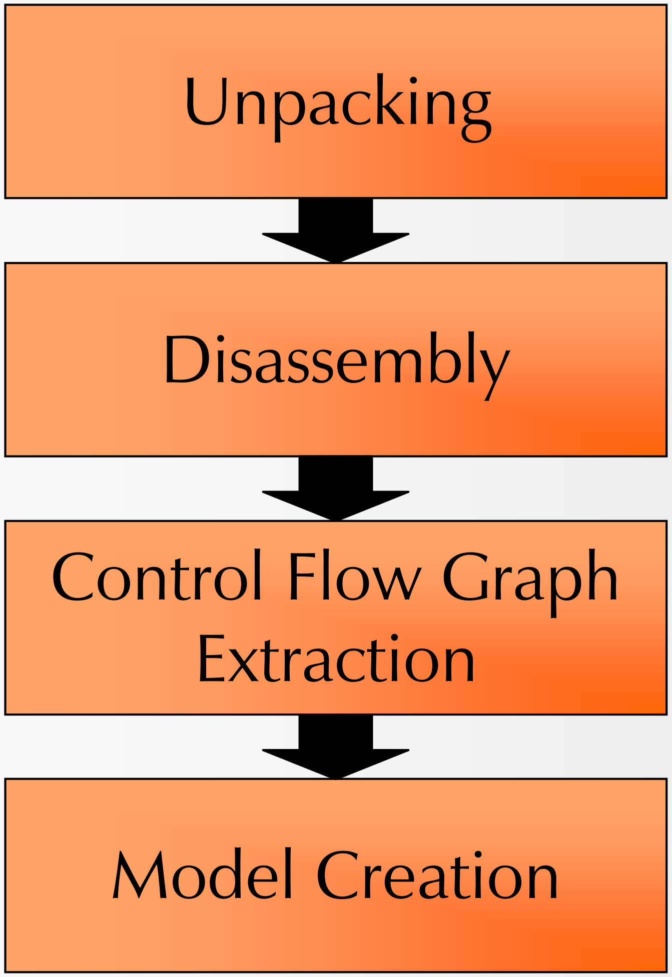
Model



Specification

$$
\begin{array}{l}
\exists L_m \exists L_c \exists v_{File}(\\
\quad \exists r_0 \exists r_1 \exists L_0 \exists L_1 \exists c_0(\\
\quad\quad \mathbf{EF}(\mathtt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{mov}(r_0,t) \vee \mathtt{lea}(r_0,t)))\mathbf{U}\,\#loc(L_0))\wedge \\
\quad\quad \mathbf{EF}(\mathtt{mov}(r_1, 0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{mov}(r_1,t) \vee \mathtt{lea}(r_1,t)))\mathbf{U}\,\#loc(L_1))\wedge \\
\quad\quad \mathbf{EF}(\mathtt{push}(c_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
\quad\quad\quad \mathbf{U}(\mathtt{push}(r_0) \wedge \#loc(L_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
\quad\quad\quad\quad \mathbf{U}(\mathtt{push}(r_1) \wedge \#loc(L_1) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
\quad\quad\quad\quad\quad \mathbf{U}(\mathtt{call}(\mathtt{GetModuleFileNameA}) \wedge \#loc(L_m))))) \\
\quad\quad) \\
\quad \wedge(\exists r_0 \exists L_0(\\
\quad\quad \mathbf{EF}(\mathtt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{mov}(r_0,t) \vee \mathtt{lea}(r_0,t)))\mathbf{U}\,\#loc(L_0))\wedge \\
\quad\quad \mathbf{EF}(\mathtt{push}(r_0) \wedge \#loc(L_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
\quad\quad\quad \mathbf{U}(\mathtt{call}(\mathtt{CopyFileA}) \wedge \#loc(L_c))) \\
\quad\quad)) \\
\quad \wedge\mathbf{EF}(\#loc(L_m) \wedge \mathbf{EF}\,\#loc(L_c)) \\
)
\end{array}
$$

Model Extraction



Unpacking

Disassembly

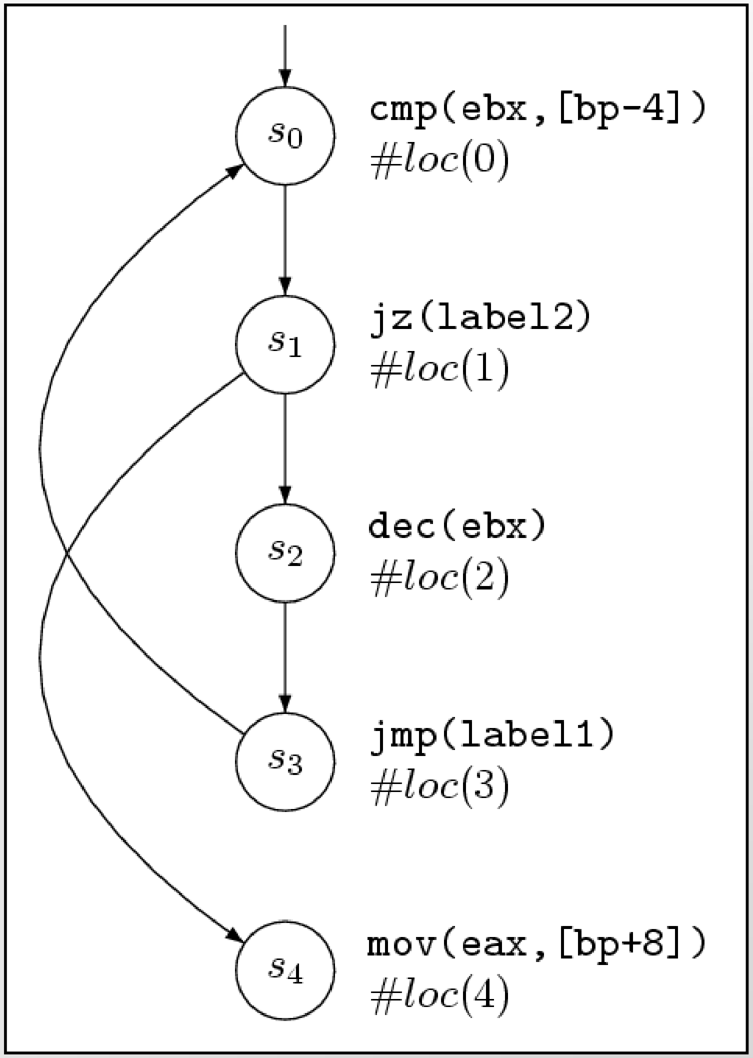
Control Flow Graph Extraction

Model Creation

- Worms are commonly packed by executable packers (e.g. UPX) and need to be unpacked
- Disassembly transforms an executable byte sequence into a sequence of instructions
- Control flow graphs display conditional branches and loops in the executable
- The graph is annotated with assembler instructions and locations (offsets)

Model Extraction – Example

```
label1:   cmp ebx, [bp-4]
          jz label2
          dec ebx
          jmp label1
label2:   mov eax, [bp+8]
          ...
```



Model Extraction – Problems

- Indirect jumps (jump targets calculated at runtime) cannot be resolved statically in general
- Thorough code obfuscation may thwart disassembly
- Self modifying code
- x86 allows unaligned jumps ‘into’ an instruction

State-of-the-art disassemblers are able to successfully process compiler generated code. This includes most of the prevalent E-mail worms.

Malicious Behavior – Example

```
...
xor     ebx,ebx                        # clear register
lea     eax, [ebp+ExFileName]          # store address of buffer
push    0x0104                         # size of string buffer
push    eax                            # push address
push    ebx                            # push a zero
call    ds:GetModuleFileNameA          # system call
lea     eax, [ebp+NewFileName]         # store destination address
push    ebx                            # push a zero
push    eax                            # push destination
lea     eax, [ebp+ExFileName]          # store source address
push    eax                            # push source address
call    ds:CopyFileA                   # system call
...
```

Code fragment of the Klez.h worm

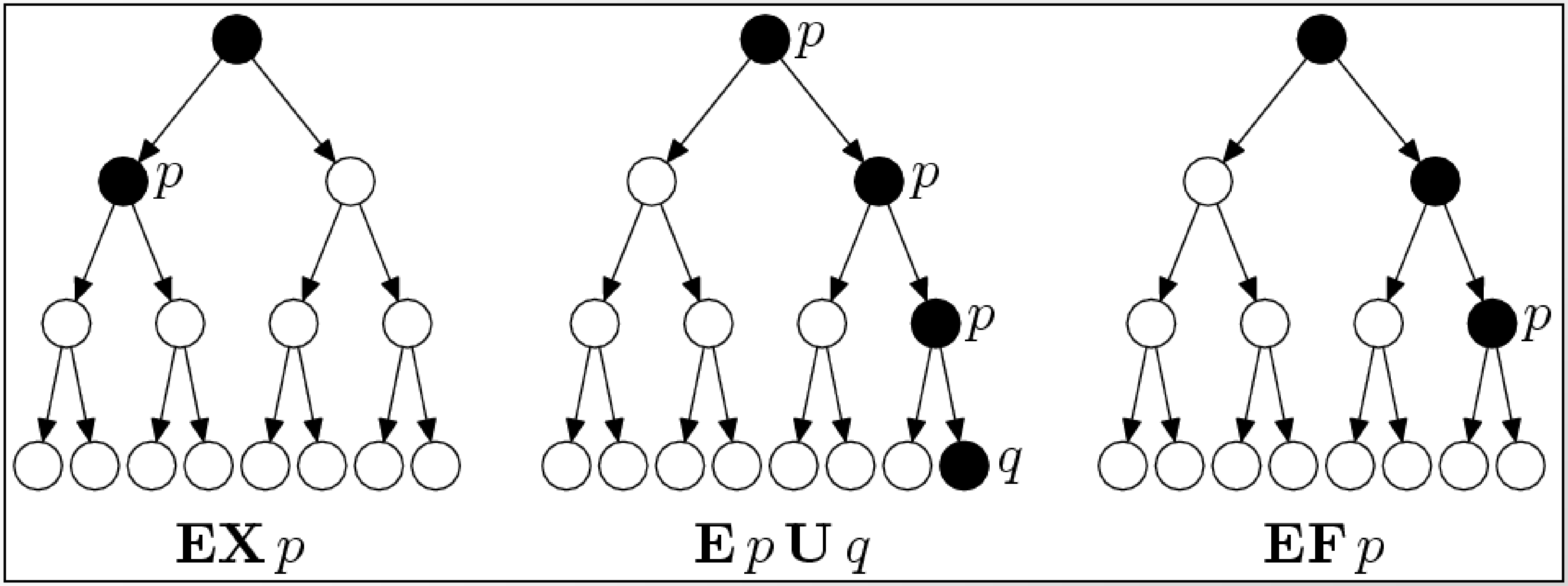
Malicious Behavior – Characteristics

```
...
xor     ebx,ebx
lea     eax, [ebp+ExFileName]
push    0x0104
push    eax
push    ebx
call    ds:GetModuleFileNameA
lea     eax, [ebp+NewFileName]
push    ebx
push    eax
lea     eax, [ebp+ExFileName]
push    eax
call    ds:CopyFileA
...
```

- Temporal and functional dependencies of system calls characterize behavior
- Arbitrary order of independent instructions
- Register and variable substitution
- Flexibility and and readability of specifications

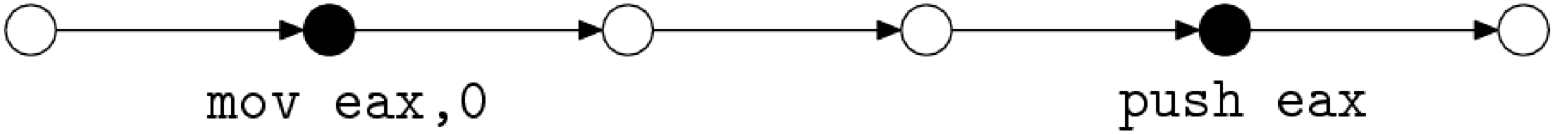
Specifying Behavior – CTL

- The logic CTL allows the specification of temporal properties of systems
- Examples:



Specifying Behavior – CTPL

- The new logic CTPL is based on CTL but allows free variables in propositions and quantifiers in formulas



$$\exists r\,\mathbf{EF}(\mathtt{mov}(r,0) \wedge \mathbf{EF}(\mathtt{push}(r)))$$

Through this extension, CTPL becomes particularly useful for specifying behavior of assembler code

CTPL Specifications

- Example 1: Initialize register with zero; later this register is pushed onto the stack

$$\boxed{\exists r\,\mathbf{EF}(\mathtt{mov}(r,\mathtt{0}) \wedge \mathbf{EF}(\mathtt{push}(r)))}$$

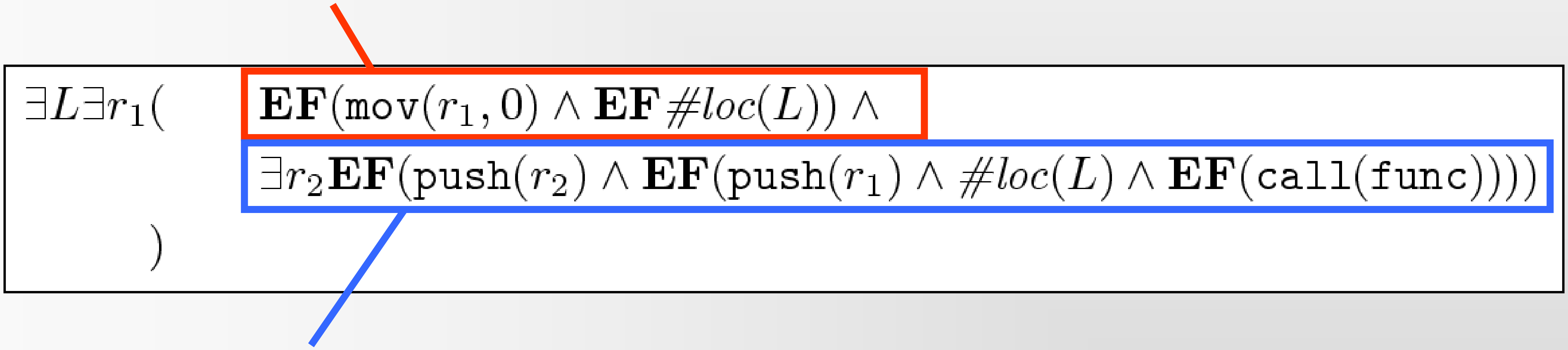
- Example 2: Same as 1, but ensure integrity of the register

$$\boxed{\exists r\,\mathbf{EF}(\mathtt{mov}(r,\mathtt{0}) \wedge \mathbf{E}(\neg\exists t\,\mathtt{mov}(r,t)\,\mathbf{U}\,\mathtt{push}(r)))}$$

CTPL Specifications – System Calls

- System call with parameter initialization:

Parameter Initialization



$$\begin{aligned} \exists L \exists r_1 (\quad & \boxed{\mathbf{EF}(\texttt{mov}(r_1, 0) \wedge \mathbf{EF}\,\#loc(L)) \wedge} \\ & \boxed{\exists r_2 \mathbf{EF}(\texttt{push}(r_2) \wedge \mathbf{EF}(\texttt{push}(r_1) \wedge \#loc(L) \wedge \mathbf{EF}(\texttt{call}(\texttt{func}))))} \\ ) \quad & \end{aligned}$$

Stack layout, invoke system call

CTPL Specifications – System Calls

- System call with parameter initialization:

$$\begin{aligned}
\exists L \exists r_1 (\quad & \mathbf{EF}(\mathtt{mov}(r_1, 0) \wedge \mathbf{EF}\boxed{\#loc(L))} \wedge \\
& \exists r_2 \mathbf{EF}(\mathtt{push}(r_2) \wedge \mathbf{EF}(\mathtt{push}(r_1) \wedge \boxed{\#loc(L)} \wedge \mathbf{EF}(\mathtt{call}(\mathtt{func})))) \\
)
\end{aligned}$$

Formulas are linked by the location predicate `#loc`

CTPL Specification Based on Klez

```
1.  $\exists L_m \exists L_c \exists v_{File}($
2.      $\exists r_0 \exists r_1 \exists L_0 \exists L_1 \exists c_0($
3.          $\mathbf{EF}(\texttt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{mov}(r_0, t) \vee \texttt{lea}(r_0, t)))\mathbf{U}\,\#loc(L_0))\wedge$
4.          $\mathbf{EF}(\texttt{mov}(r_1, \texttt{0}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{mov}(r_1, t) \vee \texttt{lea}(r_1, t)))\mathbf{U}\,\#loc(L_1))\wedge$
5.          $\mathbf{EF}(\texttt{push}(c_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t)))$
6.              $\mathbf{U}(\texttt{push}(r_0) \wedge \#loc(L_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t)))$
7.                  $\mathbf{U}(\texttt{push}(r_1) \wedge \#loc(L_1) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t)))$
8.                      $\mathbf{U}(\texttt{call}(\texttt{GetModuleFileNameA}) \wedge \#loc(L_m)))))$
9.          $)$
10.     $\wedge(\exists r_0 \exists L_0($
11.         $\mathbf{EF}(\texttt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{mov}(r_0, t) \vee \texttt{lea}(r_0, t)))\mathbf{U}\,\#loc(L_0))\wedge$
12.         $\mathbf{EF}(\texttt{push}(r_0) \wedge \#loc(L_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t)))$
13.             $\mathbf{U}(\texttt{call}(\texttt{CopyFileA}) \wedge \#loc(L_c)))$
14.         $))$
15.     $\wedge\mathbf{EF}(\#loc(L_m) \wedge \mathbf{EF}\,\#loc(L_c))$
16. $)$
```

CTPL Specification Based on Klez

$$\begin{array}{ll}
1. & \exists L_m \exists L_c \exists v_{File}(\\
2. & \quad \exists r_0 \exists r_1 \exists L_0 \exists L_1 \exists c_0(\\
3. & \qquad \mathbf{EF}(\texttt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{mov}(r_0,t) \vee \texttt{lea}(r_0,t)))\mathbf{U}\,\boxed{\#loc(L_0)}) \wedge \\
4. & \qquad \mathbf{EF}(\texttt{mov}(r_1,0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{mov}(r_1,t) \vee \texttt{lea}(r_1,t)))\mathbf{U}\,\#loc(L_1)) \wedge \\
5. & \qquad \mathbf{EF}(\texttt{push}(c_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t))) \\
6. & \qquad\quad \mathbf{U}(\texttt{push}(r_0) \wedge \boxed{\#loc(L_0)} \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t))) \\
7. & \qquad\qquad \mathbf{U}(\texttt{push}(r_1) \wedge \#loc(L_1) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t))) \\
8. & \qquad\qquad\quad \mathbf{U}(\texttt{call}(\texttt{GetModuleFileNameA}) \wedge \#loc(L_m))))) \\
9. & \qquad) \\
10. & \quad \wedge(\exists r_0 \exists L_0(\\
11. & \qquad \mathbf{EF}(\texttt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{mov}(r_0,t) \vee \texttt{lea}(r_0,t)))\mathbf{U}\,\#loc(L_0)) \wedge \\
12. & \qquad \mathbf{EF}(\texttt{push}(r_0) \wedge \#loc(L_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\texttt{push}(t) \vee \texttt{pop}(t))) \\
13. & \qquad\quad \mathbf{U}(\texttt{call}(\texttt{CopyFileA}) \wedge \#loc(L_c))) \\
14. & \qquad)) \\
15. & \quad \wedge \mathbf{EF}(\#loc(L_m) \wedge \mathbf{EF}\,\#loc(L_c)) \\
16. &)
\end{array}$$

CTPL Specification Based on Klez

1. $\exists L_m \exists L_c \exists v_{File}($

2. $\quad \exists r_0 \exists r_1 \exists L_0 \exists L_1 \exists c_0($

3. $\quad\quad \mathbf{EF}(\mathtt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{mov}(r_0,t) \vee \mathtt{lea}(r_0,t)))\mathbf{U}\,\#loc(L_0))\wedge$

4. $\quad\quad \mathbf{EF}(\mathtt{mov}(r_1,\mathtt{0}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{mov}(r_1,t) \vee \mathtt{lea}(r_1,t)))\mathbf{U}\,\#loc(L_1))\wedge$

5. $\quad\quad \mathbf{EF}(\mathtt{push}(c_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t)))$

6. $\quad\quad\quad \mathbf{U}(\mathtt{push}(r_0) \wedge \#loc(L_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t)))$

7. $\quad\quad\quad\quad \mathbf{U}(\mathtt{push}(r_1) \wedge \#loc(L_1) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t)))$

8. $\quad\quad\quad\quad\quad \mathbf{U}(\mathtt{call}(\mathtt{GetModuleFileNameA}) \wedge \boxed{\#loc(L_m))})))$

9. $\quad\quad)$

10. $\quad \wedge(\exists r_0 \exists L_0($

11. $\quad\quad \mathbf{EF}(\mathtt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{mov}(r_0,t) \vee \mathtt{lea}(r_0,t)))\mathbf{U}\,\#loc(L_0))\wedge$

12. $\quad\quad \mathbf{EF}(\mathtt{push}(r_0) \wedge \#loc(L_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t)))$

13. $\quad\quad\quad \mathbf{U}(\mathtt{call}(\mathtt{CopyFileA}) \wedge \boxed{\#loc(L_c))})$

14. $\quad\quad))$

15. $\quad \wedge\mathbf{EF}(\boxed{\#loc(L_m)}) \wedge \mathbf{EF}(\boxed{\#loc(L_c))}$

16. $)$

Macro-Supported CTPL

- Recurring patterns in specifications can be encapsulated by a set of macros

%nostack	%noassign	%syscall	%sysfunc
stack integrity	variable integrity	system call	system call with return value

- Unneeded variables are replaced by wildcards
- Allows succinct and natural specifications

```
EF(
    %syscall(GetModuleFileNameA, $*, $pFile, 0) &
      E %noassign($pFile) U %syscall(CopyFileA, $pFile)
)
```

CTPL specification based on Klez in prototype syntax

CTPL Model Checking Algorithm

- Based on classic explicit CTL Model Checking
  - Linear time algorithm by Clarke and Emerson
  - Bottom-up evaluation of the formula
  - Dynamic programming
- The CTPL algorithm has to collect variable bindings
- CTPL Model Checking is PSPACE-complete
- Efficient in real world settings:
  - Algorithm is exponential in size of the specification,
  - But linear in size of the model

Experimental Results

Badtrans.a	—	√	102.0
Bugbear.a	√	√	5.0
Bugbear.e	—	—	1.6
Dumaru.a	√	—	3.7
Dumaru.b	√	—	3.6
Klez.a	√	—	2.2
Klez.e	√	—	5.9
Klez.h	√	—	6.0
MyDoom.a	√	—	2.7
MyDoom.i	√	—	2.2
MyDoom.m	√	—	2.2
NetSky.b	√	—	5.6
NetSky.d	√	—	1.9
NetSky.p	√	—	0.6
Nimda.a	—	√	3.4
Nimda.e	—	√	4.9

🟥 `CopySelf`

🟪 `ExecOpened`

🟨 Time (s)

Summary

- Model Checking is suited for mutation tolerant detection of malware
- One specification fits a large class of worms
- Proactive detection raises skill threshold for malware writers
- Future directions:
  - Abstraction of assembler code
  - Extensible macro language
  - Efficient implementation (e.g. with OBDDs)
  - Make use of program analysis techniques (data flow, slicing, interval analysis)

Thank you

Thank you for your attention.

Questions?