Effective *and* Efficient Malware Detection at the End Host

Clemens KOLBITSCH, Paolo MILANI COMPARETTI, Engin KIRDA, Christopher KRUEGEL, Xiaoyong ZHOU, XiaoFeng WANG

ck@iseclab.org

Secure Systems Lab [TU Vienna, Institute Eurecom Sophia Antipolis, UC Santa Barbara]
Indiana University at Bloomington

Motivation

Why do we propose yet another malware detection scheme (yamds)?

- Binary signature based detection inherently *ineffective*
  - We all know the problems...
  - Arms-race, pretty much a lost battle

- Network based approaches *evadable*
  - Systems scan for communication artifacts
  - Encryption / blending thwart detection

Motivation

Why do we propose yet another malware detection scheme (yamds)?

- Don't rely on artifacts of malware *instances*
  - Instead focus on generic patterns

- Proposed solution:
  - Detection based on malware's behavior
  - Behavior is hard to obfuscate
  - Behavior is hard to randomize
  - Behavior is often stable across various malware version

Motivation

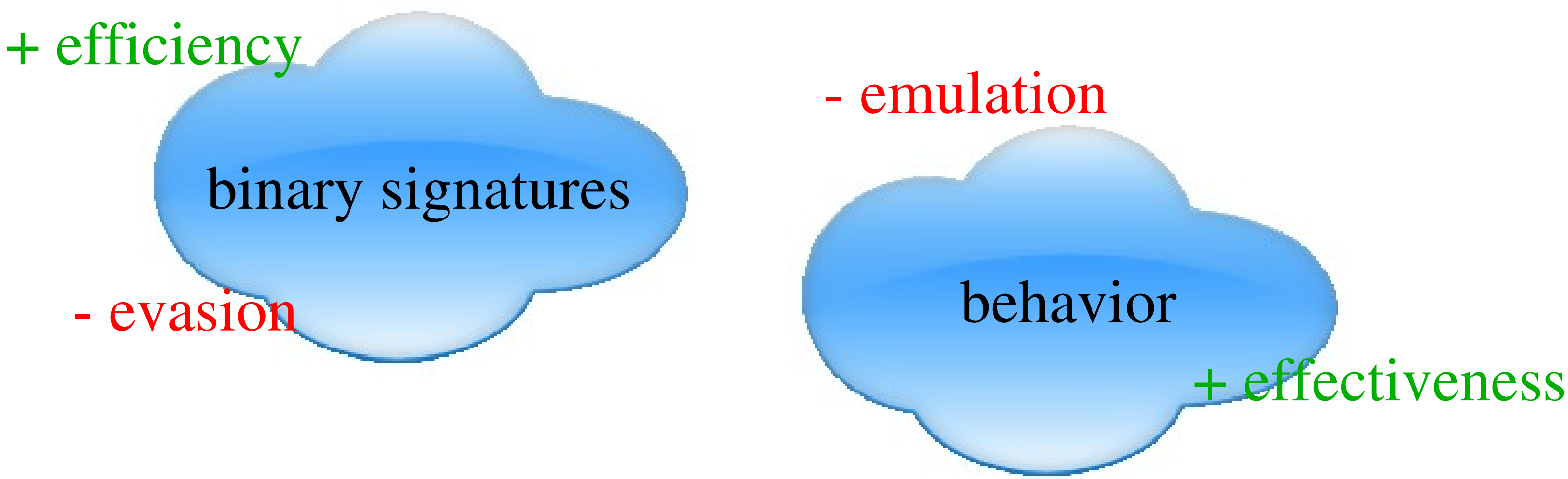
- Behavior-based detection received some attention over last couple of years

- Despite promising detection results, binary signatures remain the method of choice

Motivation

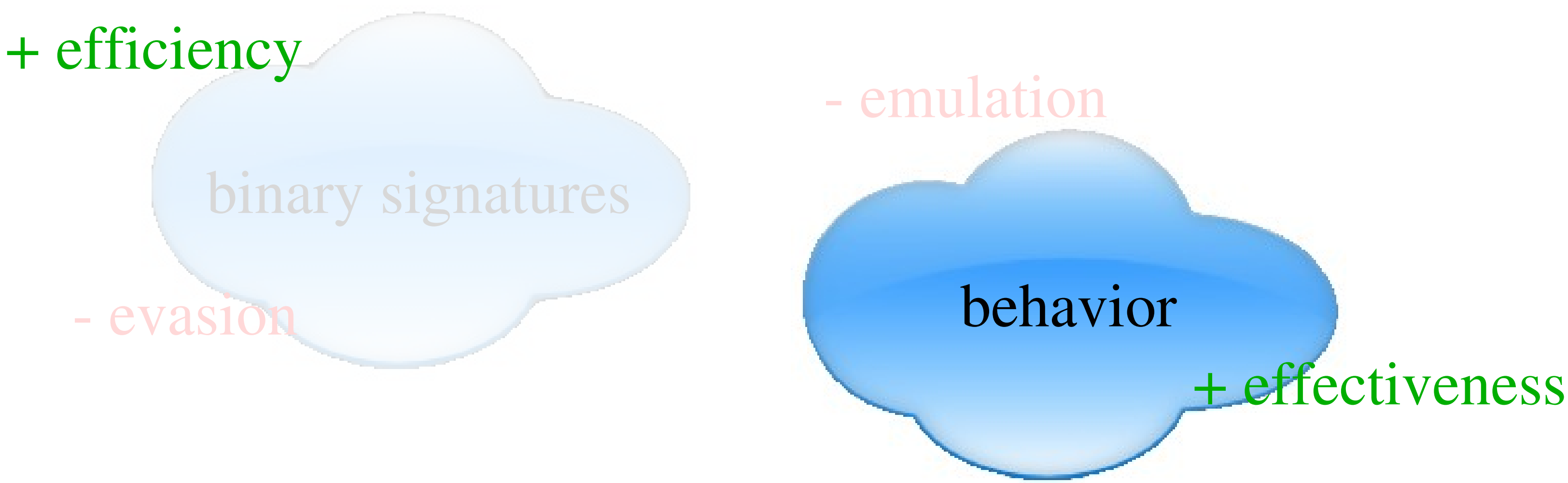
- Behavior-based detection received some attention over last couple of years

- Despite promising detection results, binary signatures remain the method of choice



Motivation

- Behavior-based detection received some attention over last couple of years
- Despite promising detection results, binary signatures remain the method of choice



Outline

- Motivation
- Detecting Behavior
  - Motivating example (Agent)
- Matching Behavior Graphs
- Extracting Behavior Graphs
- Evaluation

Detecting Behavior

Detecting Behavior

- Characteristic malware behavior
  - Manifest on system (i.e., survive reboot)
    - (Over-) write system executables, dlls, files
    - Create registry entries (autorun)
    - Register as Windows (startup) service
  - Conceal from being detected
    - Restart under some *stealthy* name (e.g., svchost.exe)
    - Inject into legitimate processes
  - Replicate
    - Send eMails ('*check out this picture I found: pic.jpg.exe*')
    - Copy to Samba shares, USB drives, etc.
    - Scan and exploit services on LAN or WAN

Detecting Behavior
System Overview

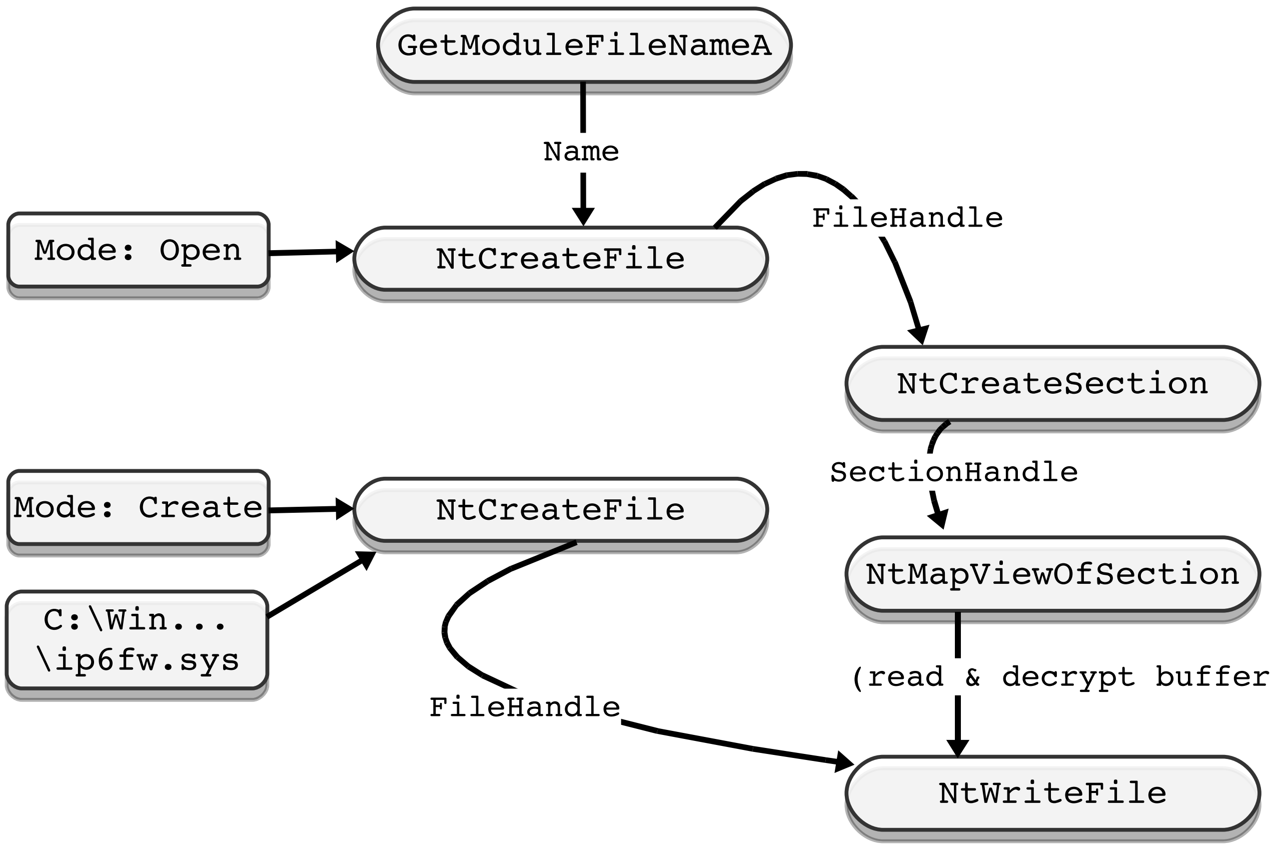
- Detection based on *execution characteristics*
  - Execute malware in full system emulator (Anubis)
  - Monitor interaction with the operating system
  - Perform detailed (taint-) analysis
  - Generate *detection graphs*
    - Describe sequence of *required* system calls leading to *security relevant* system activity
    - Include dependencies to related, previous calls (using taint dependencies)
- Detect described behavior on end host
  - Log system call activity of unknown executable
  - Match against behavior graph

Detecting Behavior
Developer Perspective

- Example: Agent (trojan horse)

- As part of its system manifestation, it
  - Reads content from binary image
  - Decrypts binary content
    - Proprietary decryption routine
    - Simple, XOR based algorithm
  - Stores binary in system file (`C:\Wind...\drivers\ip6fw.sys`)
  - Later, restarts IPv6 firewall
    - Turns itself into a system service

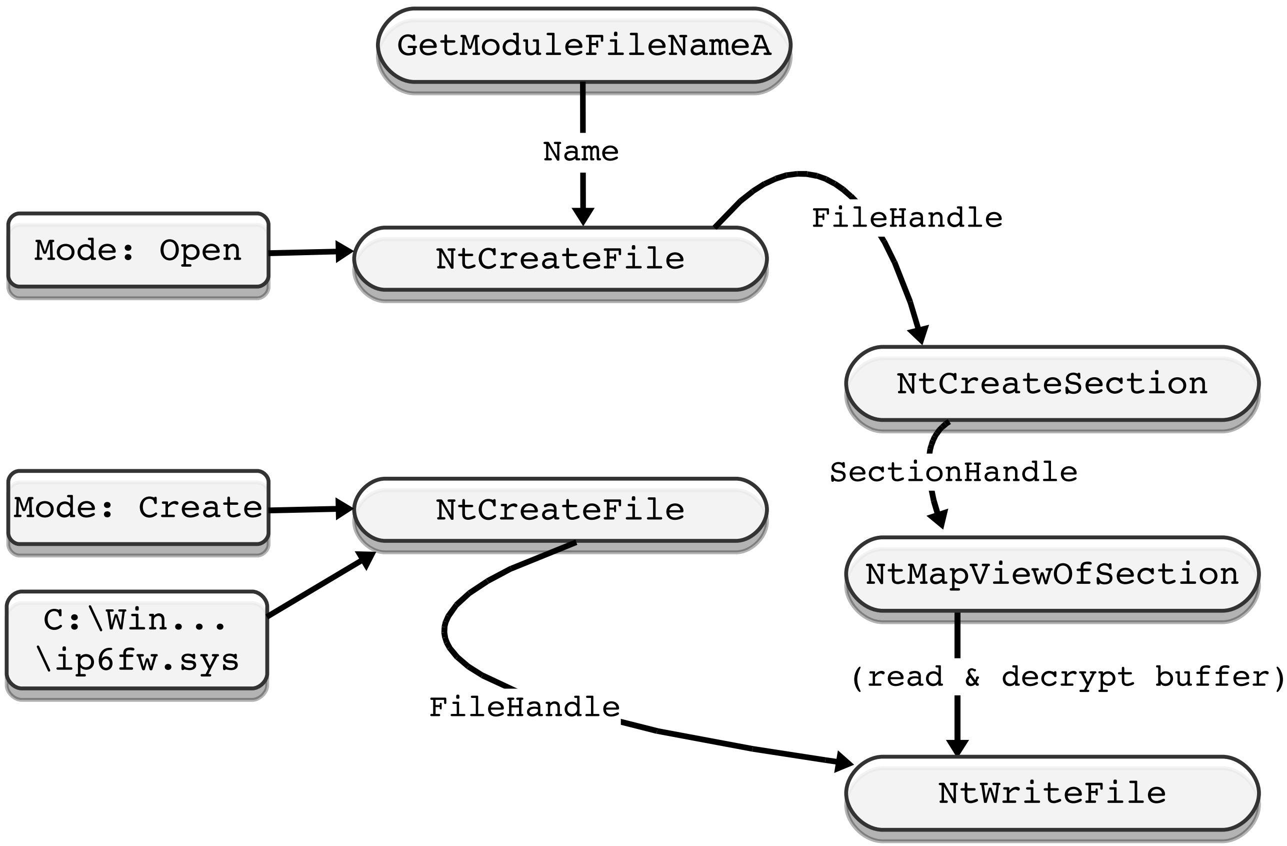
Detecting Behavior
Taint-Trace Perspective



Detecting Behavior

System Perspective

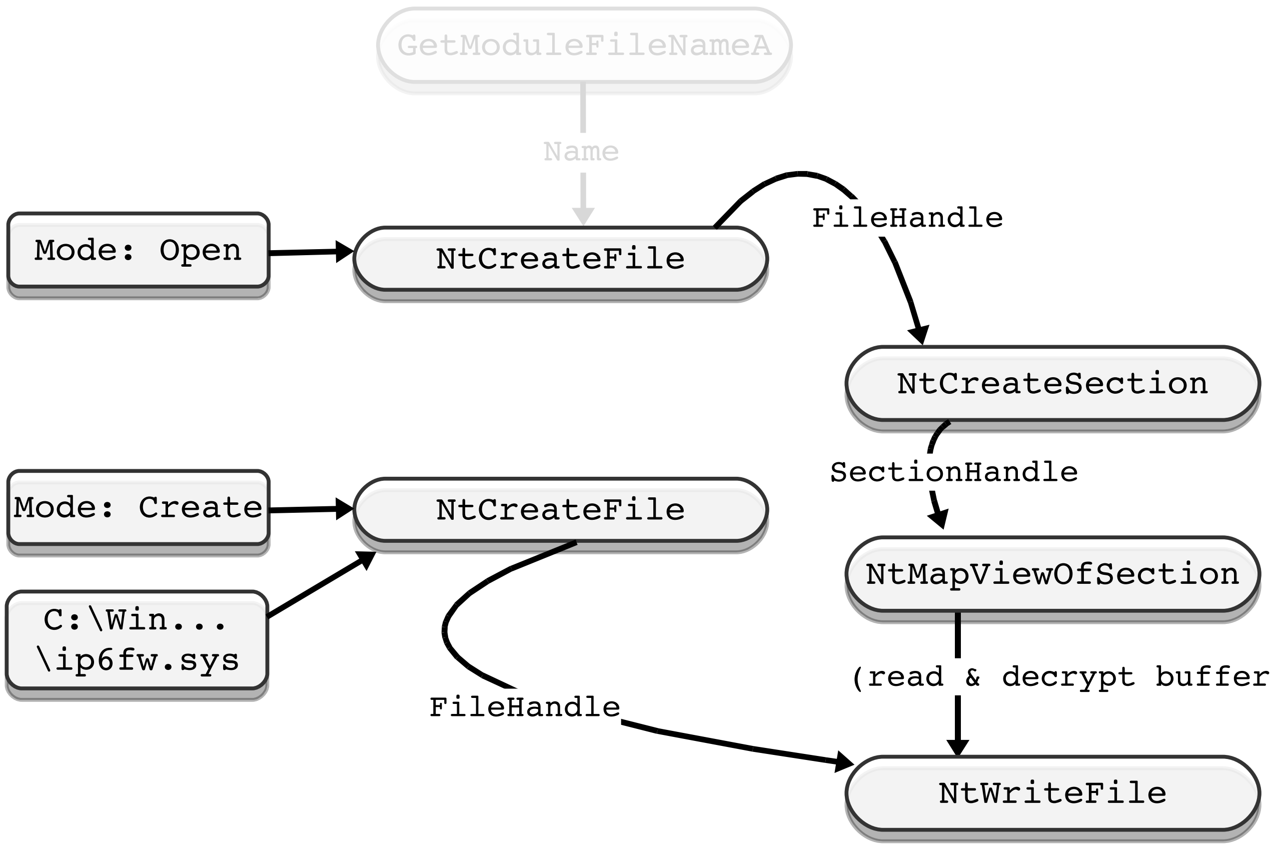
Secure Systems Lab
Technical University Vienna



Detecting Behavior

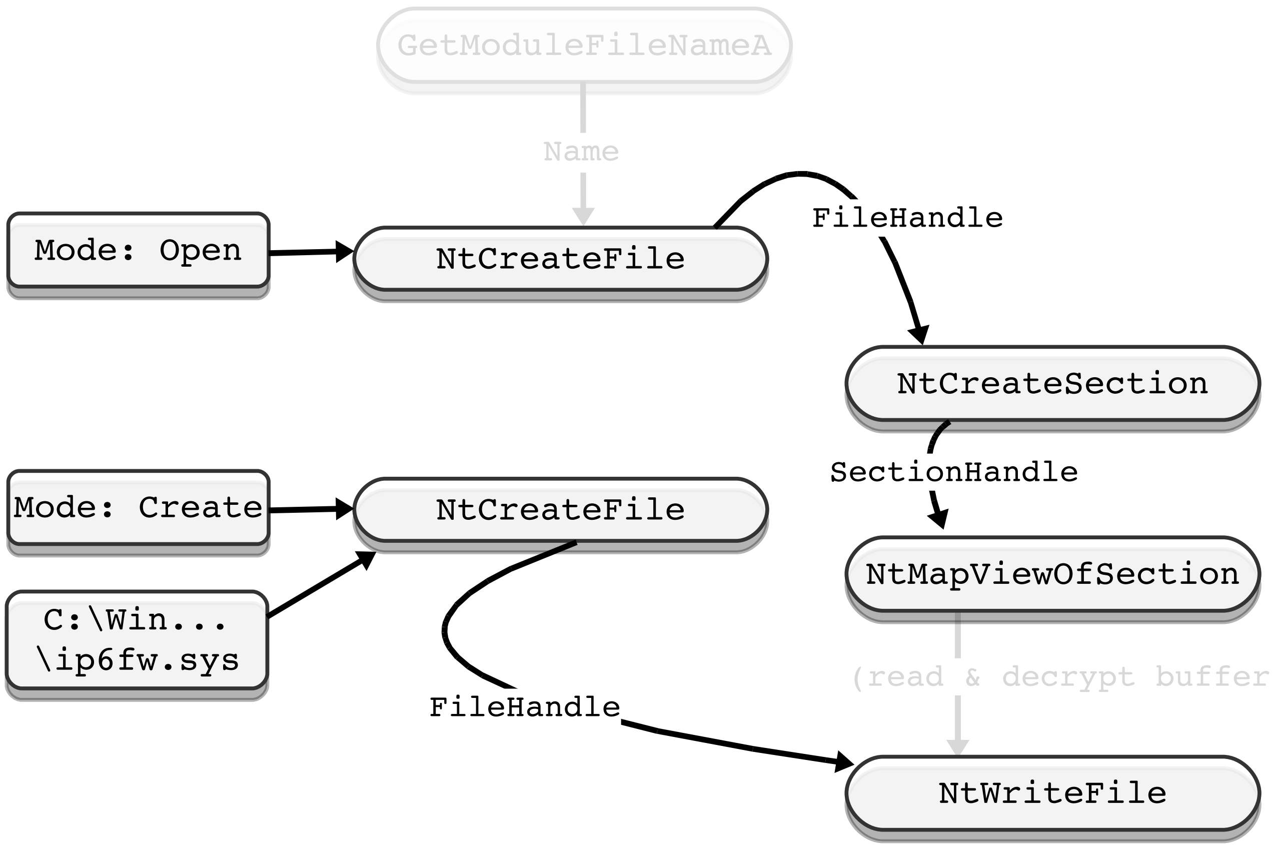
System Perspective

Secure Systems Lab
Technical University Vienna



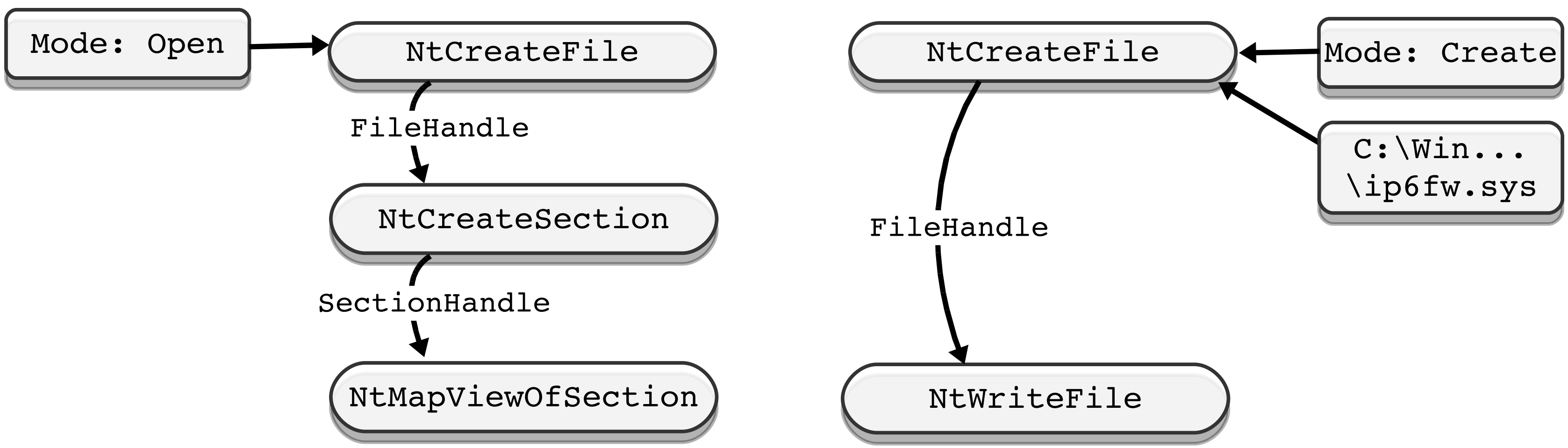
Detecting Behavior
System Perspective

Secure Systems Lab
Technical University Vienna

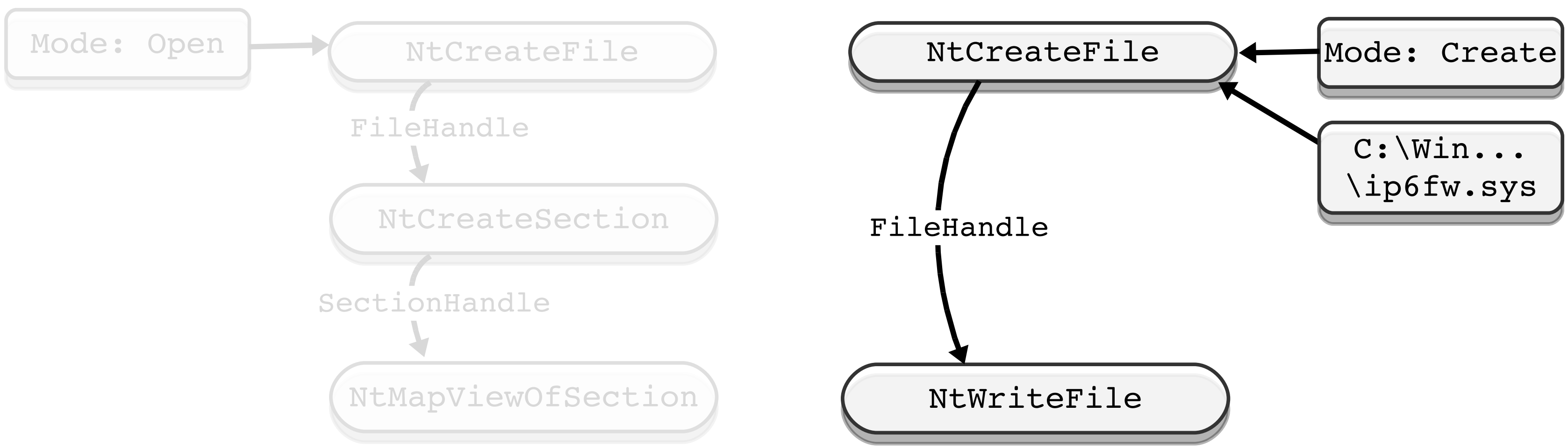


Detecting Behavior
System Perspective

Secure Systems Lab
Technical University Vienna



Detecting Behavior
System Perspective



Detecting Behavior

- Detection based on *execution characteristics*
  - Works well as long as we can see *all types of* dependencies between system calls
  - *Handle* dependencies
    - Insufficient for detection
    - Behavior graphs break into trivial subgraphs
  - *Data* dependencies
    - *Convenient* for behavior graph generation
    - *Necessary* for behavior detection

Matching Behavior Graphs

Matching Behavior Graphs

- Maintaining dependencies using taint propagation
  - Performance overhead: Extended emulation engine
  - Memory overhead: Shadow memory
  - Not applicable to production systems / end hosts

- Maintaining dependencies *without* taint propagation
  - *Handle* dependencies
    - Direct value propagation
    - System provided identifiers
      - File, section, process, thread handles
      - Registry keys
      - Socket identifiers
  - *Must* be constant between call invocations

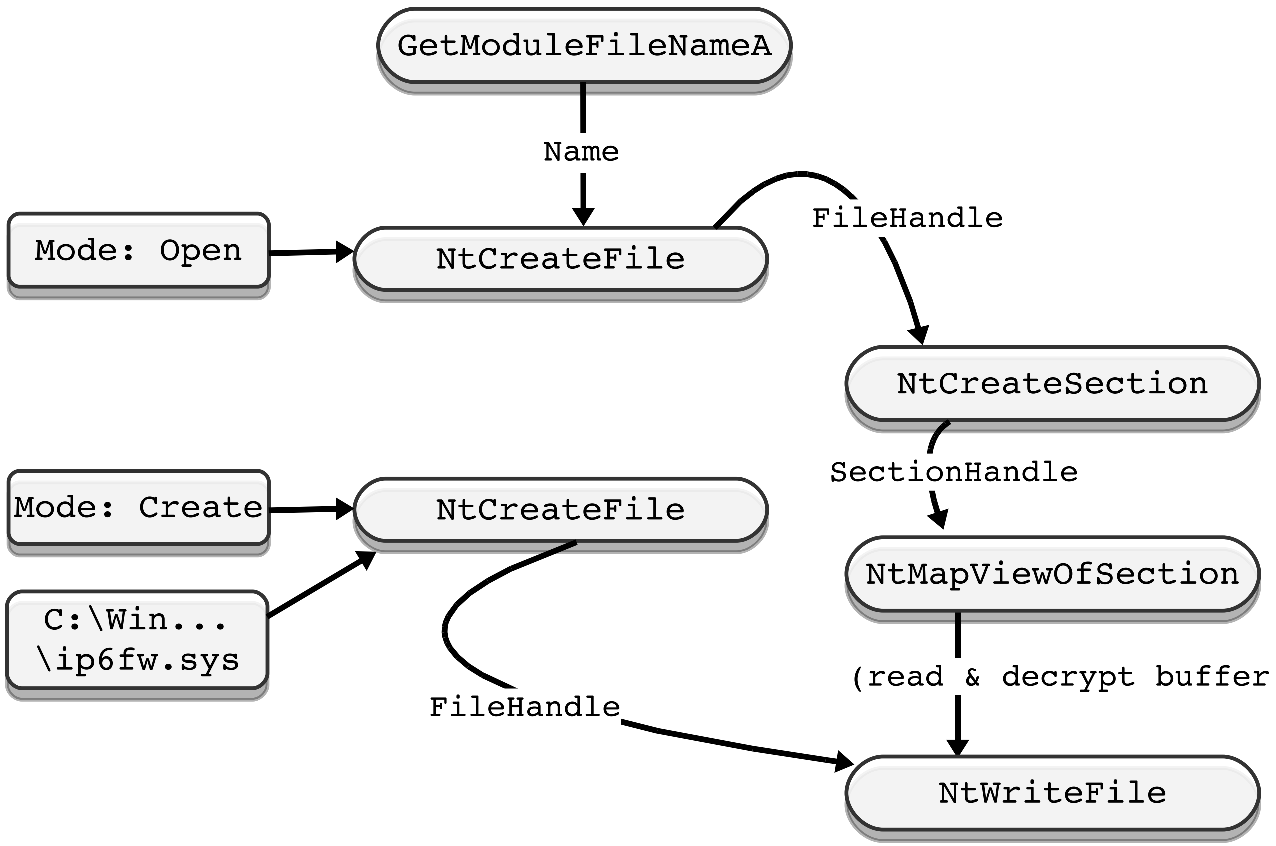
Matching Behavior Graphs

- Maintaining dependencies *without* taint propagation
  - *Data* dependencies
    - Arbitrary data (& control) dependency between system calls
    - Might modify values between system calls
  - **Our proposal: *Anticipate* precise call arguments**
    - Use recorded execution semantics
    - Extract data *propagation/manipulation formulas*
    - *Emulate* taint dependency between system call *A* and *B*
      - Log *outgoing* parameters of call *A*
      - Use as input to propagation formula
      - Predicted *incoming* parameters for system call *B*
      - Compare predicted and monitored input parameters
      - Assume dependency between *A* and *B* if prediction holds

Matching Behavior Graphs

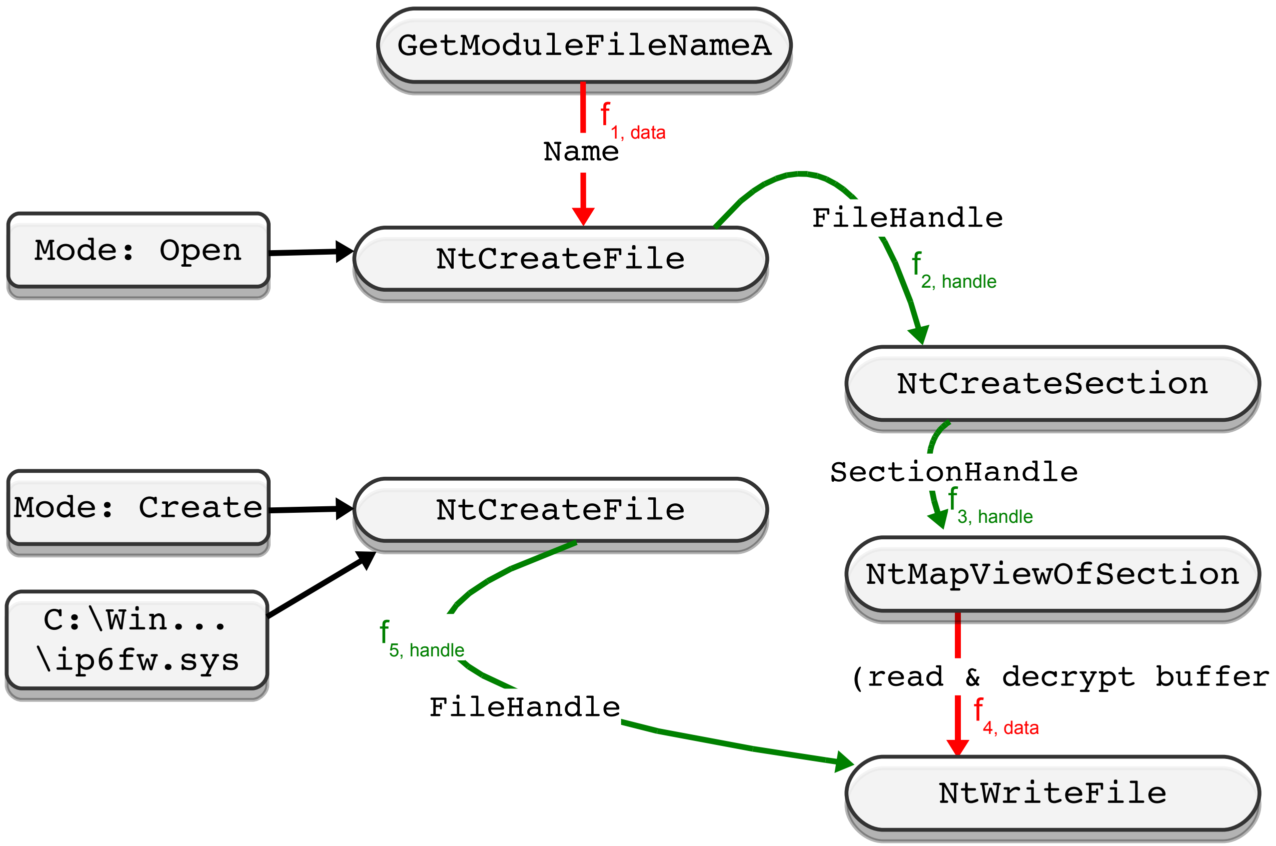
System Perspective

Secure Systems Lab
Technical University Vienna



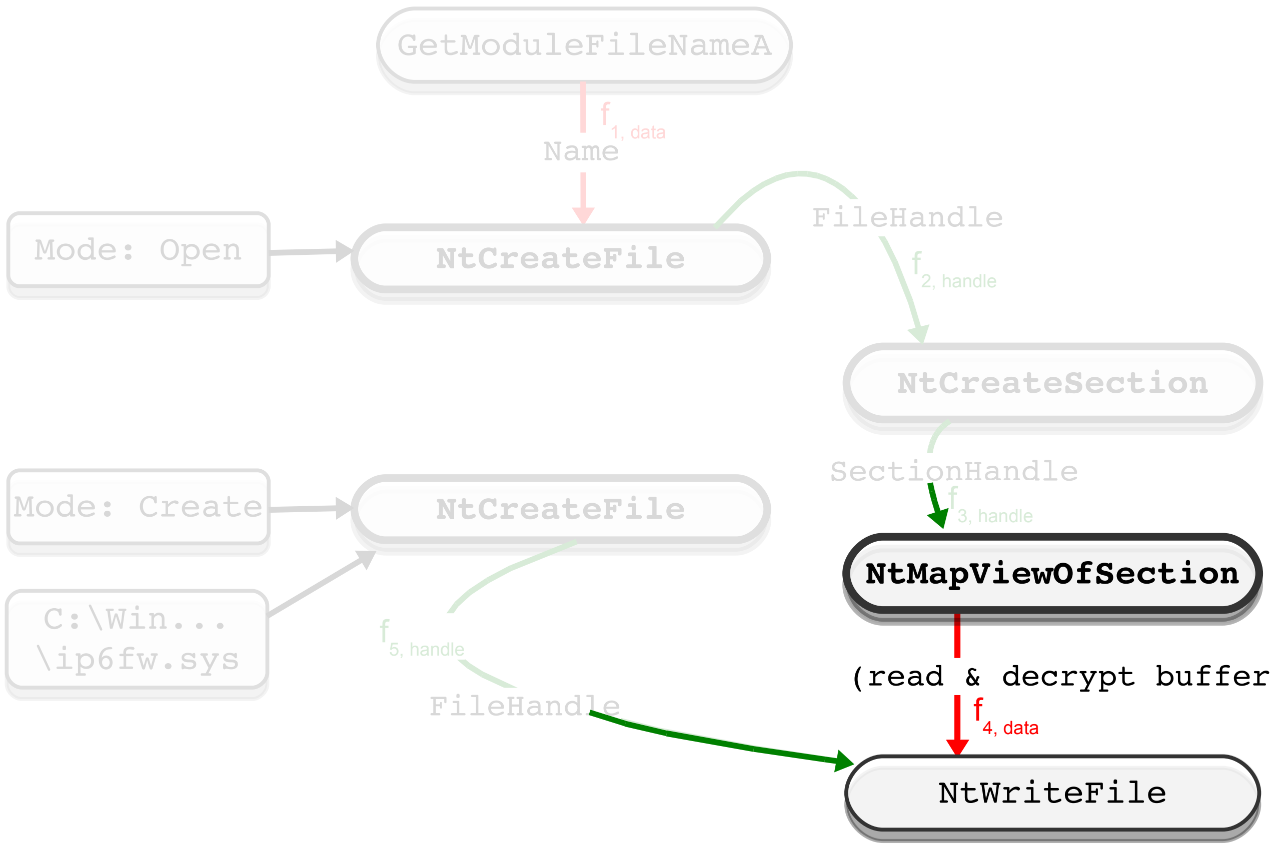
Matching Behavior Graphs
System Perspective

Secure Systems Lab
Technical University Vienna



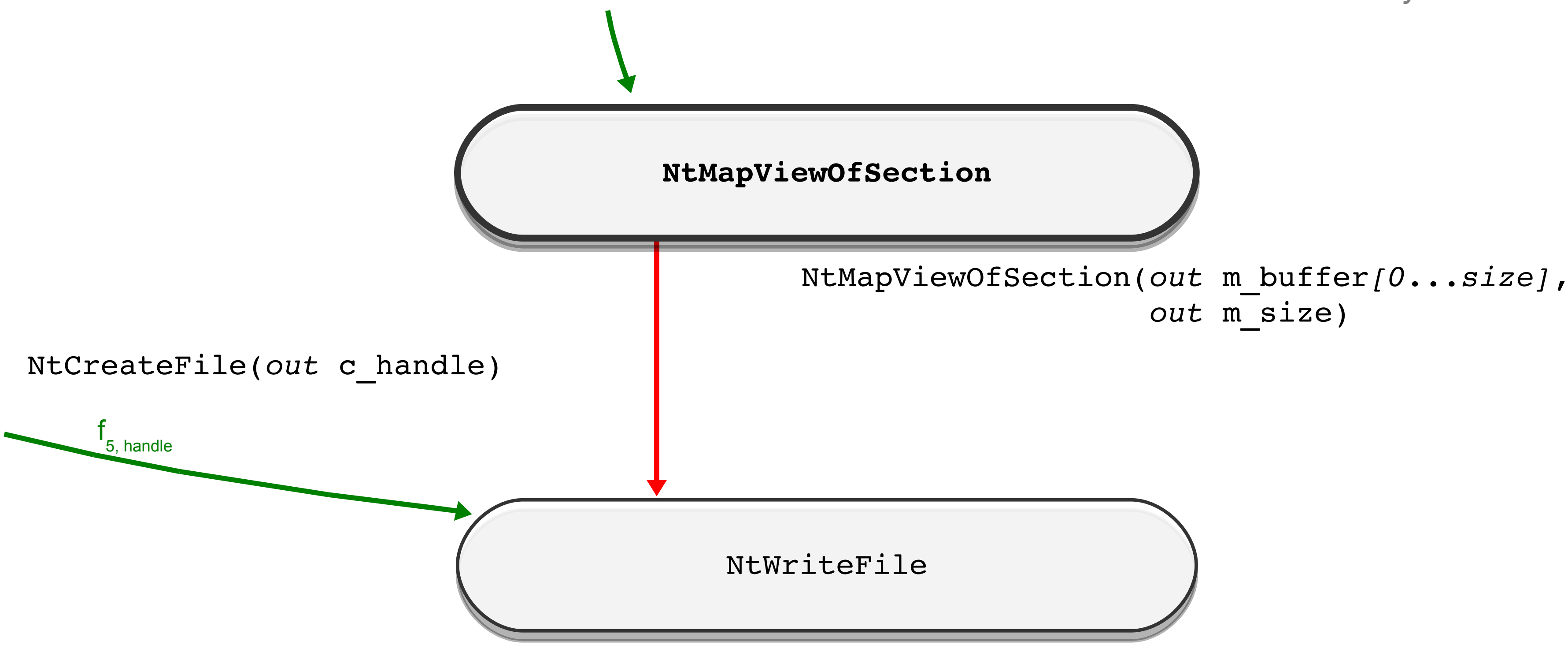
Matching Behavior Graphs
System Perspective

Secure Systems Lab
Technical University Vienna



Matching Behavior Graphs
System Perspective

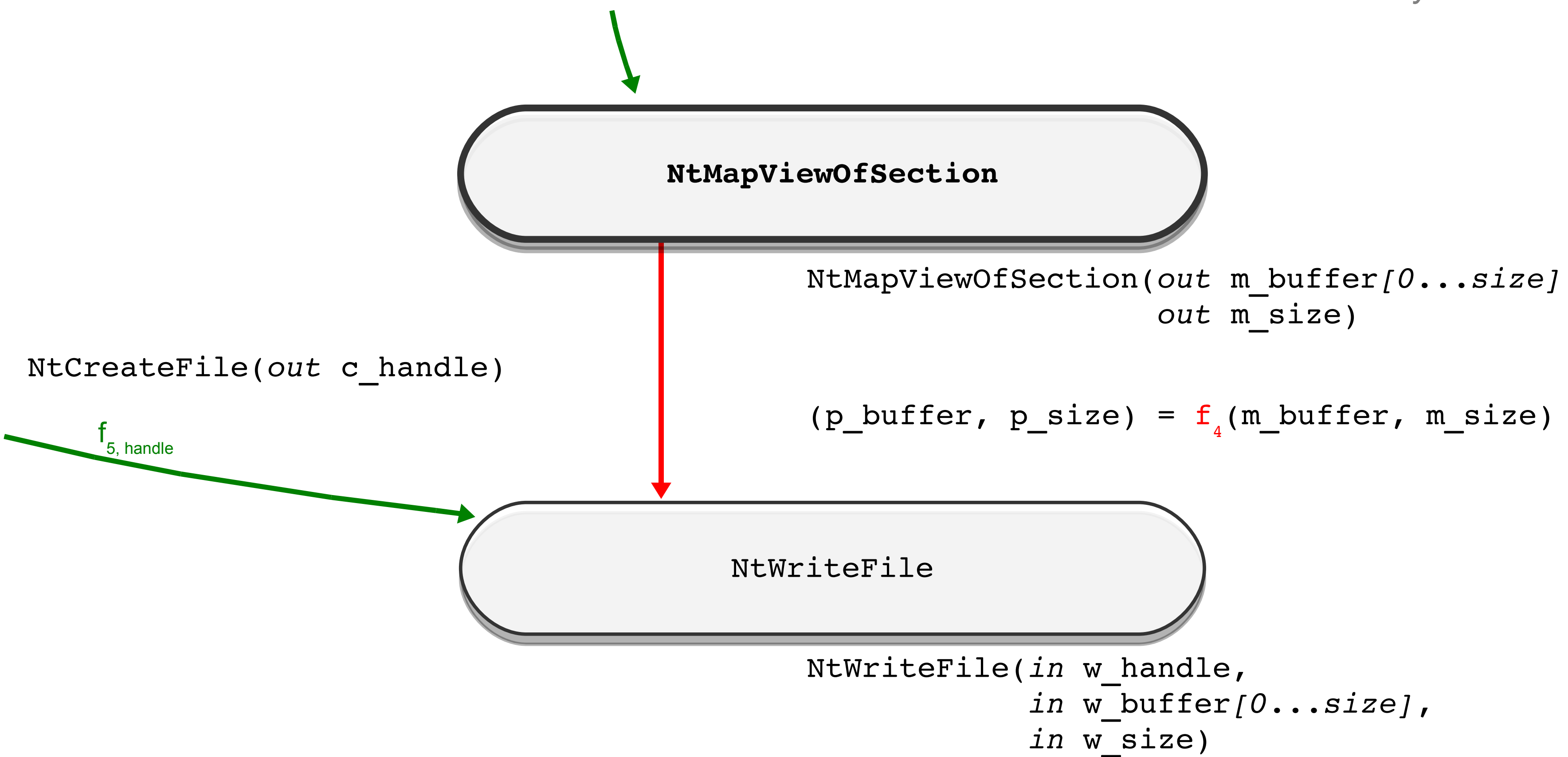
Secure Systems Lab
Technical University Vienna



Matching Behavior Graphs

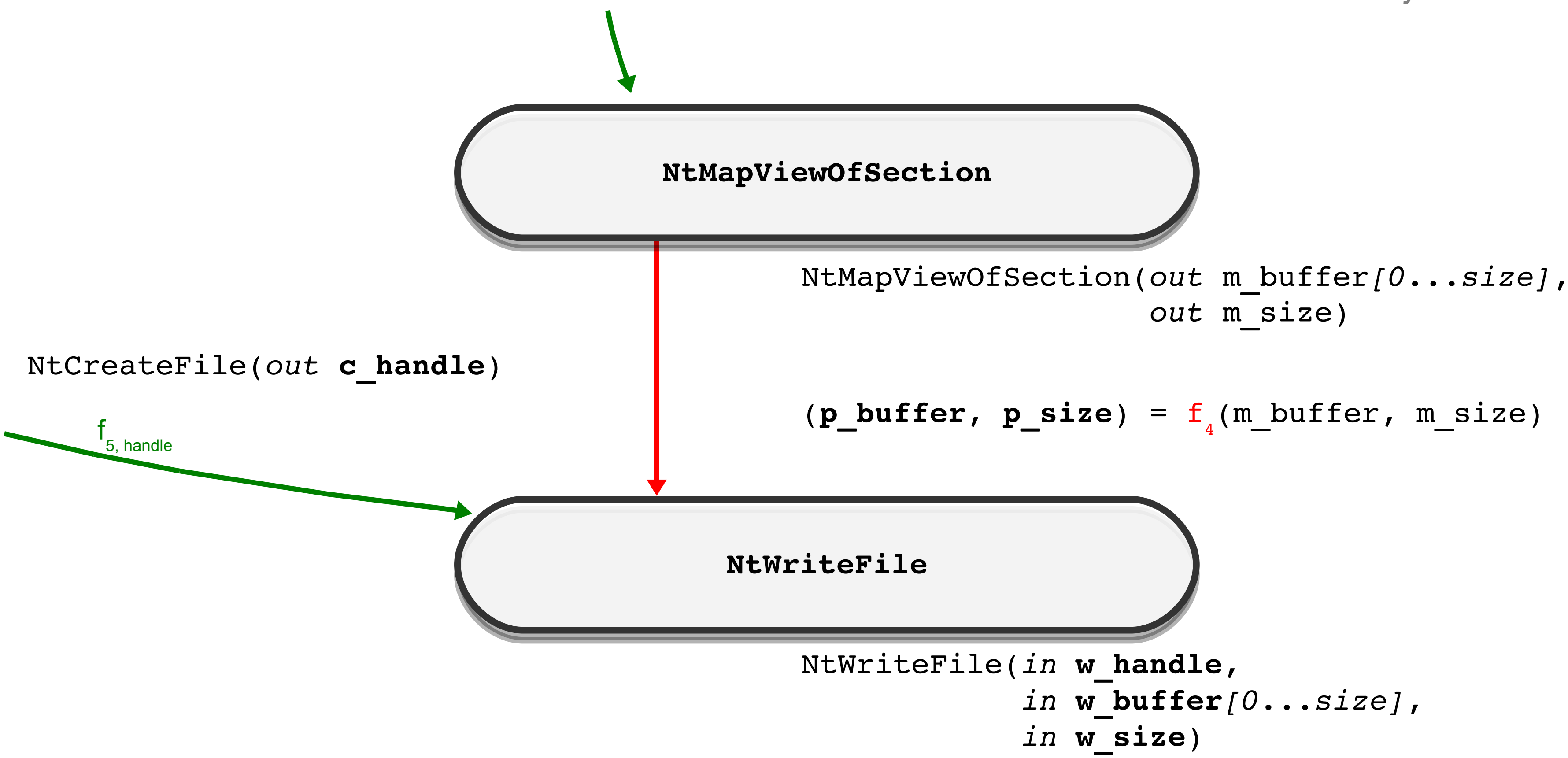
System Perspective

Secure Systems Lab
Technical University Vienna



Matching Behavior Graphs

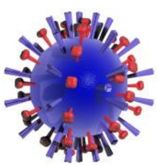
System Perspective



Extracting Behavior Graphs

Extracting Behavior Graphs

- Analyze executable in *Anubis* sandbox
  - Obtain instruction level log
    - Defeats packers
  - Obtain program flow log
  - Obtain memory access log
  - Generate *precise* taint propagation trees
    - Data/control dependencies
    - Instructions that access/generate tainted data
    - Link system calls consuming data (sinks) with all taint generating calls (sources)



Extracting Behavior Graphs

- Scan logs for *security relevant* behavior
  - Provided with a list of *interesting* system calls

- Extract graphs matching behavior
  - Include triggering system call *X*
  - Link in system calls providing tainted data to *X*
  - Analyze dependencies:
    - Label edges with *handle* dependencies
    - Call *slicer* for all *data* dependencies

Extracting Behavior Graphs

- Find encoding formula for each data dependency
- Binary *program slicing*
  - Resolve def-use chains
    - Starting at selected call invocation
    - Iterate backwards (using program flow logs)
    - Aided by *taint information* and *memory access logs*
  - Optional:
    - Symbolic execution to simplify encoding function
  - Embed into dynamically loadable library (dll)
  - Label graph edges with appropriate function (dll)

Extracting Behavior Graphs

- Resolving def-use chains
  - Three possible sources

1) Previous system call invocation
   - Replaced with stub
   - Provides input values to slice (i.e., recorded, outgoing system call parameters)

2) Immediate values
   - Implicitly encoded in binary slice (e.g., `push $0x3`)

Extracting Behavior Graphs

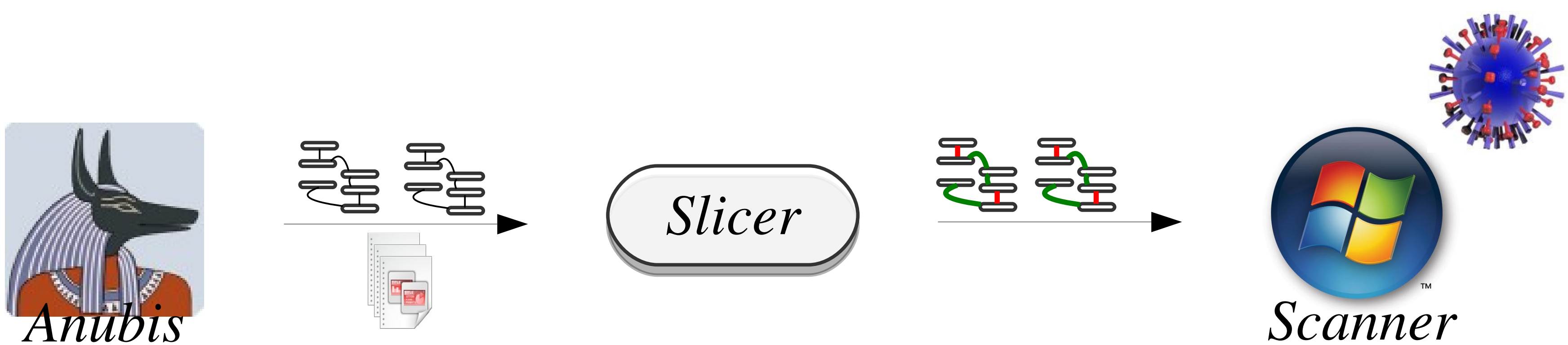
- Resolving def-use chains
  - Three possible sources

3) Preinitialized data segments

- BSS section
  - Constants
  - Static strings
- Two-sided approach:
  - Use static values from Anubis analysis
  - Dynamically inspect running process

Extracting Behavior Graphs

- Fully automated process
  - Analyze binary
  - Generate behavior graph(s)
  - Extract propagation formulas
  - Verify graph on binary
    - Run binary & scanner on real host
    - Verify behavior graph matches (only) on intended executable



Evaluation

Evaluation

- *Effectiveness* of behavior graphs
  - Applicable to polymorphic variants of a malware *sample*?
  - General enough for whole malware *families*?

- *Efficiency of behavior graph matching*
  - Overhead through system call logging
  - Additional system load through dependency verification

Effectiveness

- Six current threats / threat families
- Identified using AV (binary) signature
- Encountered *0 false positives*

Name	*Type*	*Samples*	*Variants AV*	*Variants Our*	*Samples detected*	*Eff.*
Allaple	Exploit-based worm	50	2	1	50	1.00
Beagle	Mass-mailing worm	50	20	14	46	0.92
Mydoom	Mass-mailing worm	50	32	12	47	0.94
Mytob	Mass-mailing worm	50	20	2	41	0.82
Netsky	Mass-mailing worm	50	22	12	46	0.92
Agent	Trojan horse	50	6	3	49	0.98
Total		300	102	44	279	0.93

Effectiveness

Secure Systems Lab
Technical University Vienna

- Experiment:

  Can the system detect malware instances never seen by the graph generator?

Name	*Samples*	*AV variants New*	*AV variants Known*	*Samples detected*	*Eff.*
Allaple	50	0	50	45	0.90
Beagle	50	24	26	30	0.60
Mydoom	50	24	26	36	0.72
Mytob	50	46	4	5	0.10
Netsky	13	8	5	7	0.54
Agent	50	6	44	45	0.90
Total	263	108	155	168	0.63

Effectiveness

- Experiment:

  Can the system detect malware instances never seen by the graph generator?

Name	*Samples*	*AV variants New*	*AV variants Known*	*Samples detected*	*Eff. New*	*Eff. Known*
Allaple	50	0	50	45	0.90	
Beagle	50	24	26	30	0.60	
Mydoom	50	24	26	36	0.72	
Mytob	50	46	4	5	0.10	
Netsky	13	8	5	7	0.54	
Agent	50	6	44	45	0.90	
Total	263	108	155	168	0.23	0.92

Efficiency

- I-O bound activity
  - Compressing, archiving
- CPU bound computation
  - Compilation, rendering

Test	*Baseline*	*Log*		*Full scanner*	
7-zip (benchmark)	114 sec	117 sec	2.3 %	118 sec	2.4 %
7-zip (compress)	318 sec	328 sec	3.1 %	333 sec	4.7 %
7-zip (archive)	213 sec	225 sec	6.2 %	231 sec	8.4 %
IE (rendering)	0.41 pages/s	0.39 pages/s	4.4 %	0.39 pages/s	4.4 %
VC++ (compile)	104 sec	117 sec	12.2 %	146 sec	39.8 %

Summary

- Behavior can be detected
  - Monitor from system perspective
  - Match against behavior graphs
  - Link graph nodes through argument dependencies
- *Handle* dependencies
  - Vital for checking
  - BUT not specific enough for doing detection
- *Data* dependencies
  - Anticipate future call arguments
  - Efficient replacement for taint dependencies
  - Provided through slicing malware semantics

Summary

- Evaluation
  - Behavior detection is fast enough for end hosts
  - Approach intrinsically robust against polymorphism and metamorphism
  - To some extent, behavior graphs are usable across malware variants

Thanks for your attention!