Dynamic Binary Instrumentation-based Framework for Malware Defense

Najwa Aaraj[†], Anand Raghunathan[‡], and Niraj K. Jha[†]

† Department of Electrical Engineering, Princeton University, Princeton, NJ 08544, USA

‡ NEC Labs America, Princeton, NJ 08540, USA

Outline

- **Motivation**
- Proposed framework
- Framework details
  - *Testing* environment
  - *Real* environment
- Experimental evaluation
- Related work

Motivation

- Malware defense is a primary concern in information security
  - Steady increase in the prevalence and diversity of malware
  - Escalating financial, time, and productivity losses

- Minor enhancements to current approaches are unlikely to succeed
  - Increasing sophistication in techniques used by virus writers
  - Emergence of zero-day and zero-hour attacks
- Recent advances in virtualization allows the implementation of isolated environments

Motivation *(Contd.)*

- Advances in analysis techniques such as dynamic binary instrumentation (DBI)
  - DBI injects instrumentation code that executes as part of a normal instruction stream
  - Instrumentation code allows the observation of an application's behavior
  - "Rather than considering what may occur, DBI has the benefit of operating on what actually does occur"

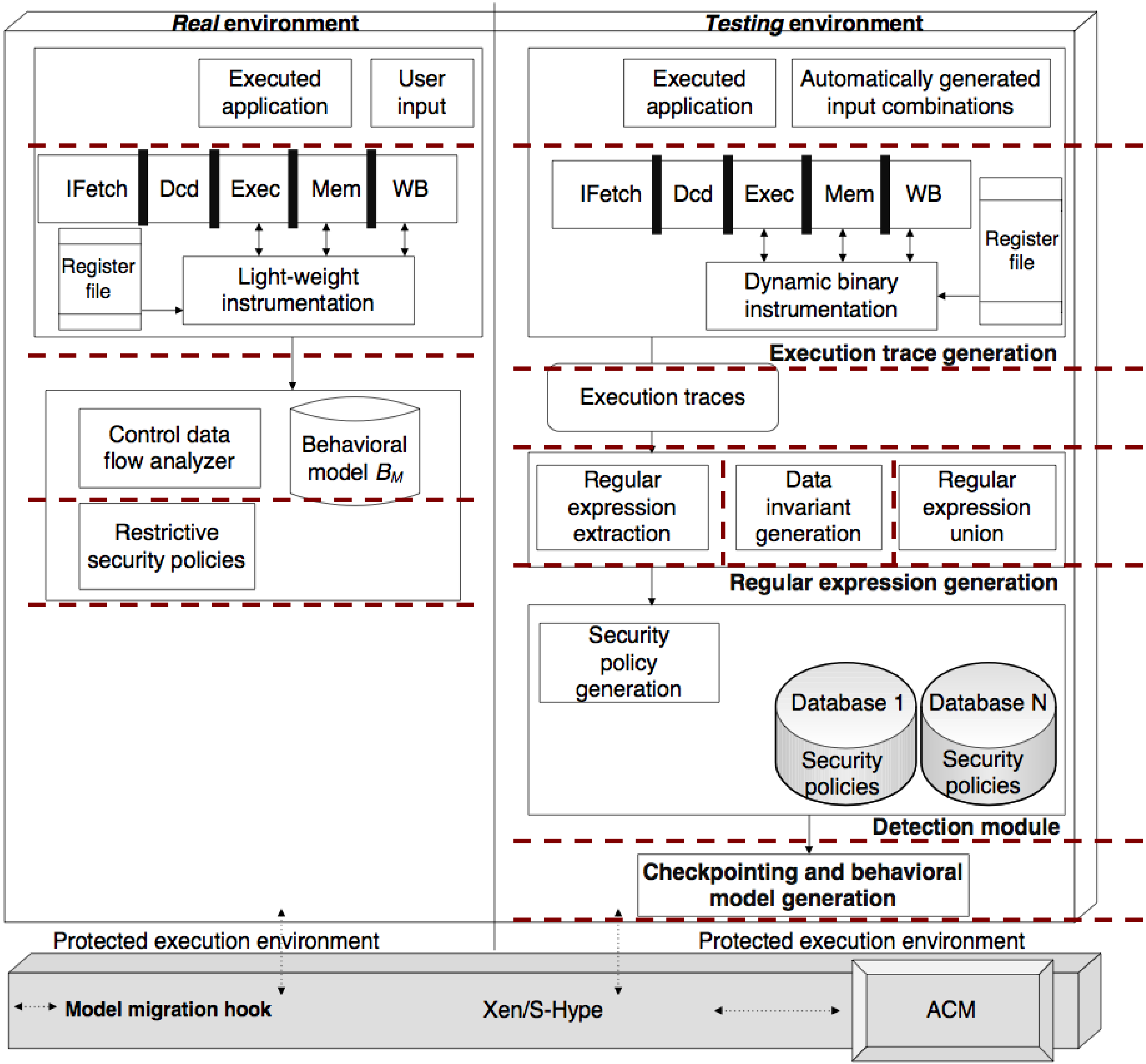
Ability to test untrusted code in an isolated environment without corrupting a "live" environment, under DBI

Outline

- Motivation
- **Proposed framework**
- Framework details
  - *Testing* environment
  - *Real* environment
- Experimental evaluation
- Related work

Proposed Framework

- Execute an untrusted program in a *Testing* environment
- Use DBI to collect specific information
- Build execution traces in the form of a hybrid model: dynamic control and data flow in terms of regular expressions, R_k's, and data invariants
- R_k's alphabet: $\sum = \{BB_1, \ldots, BB_n\}$, where BB_j captures data relevant to detecting malicious behavior
- Subject R_U, a recursive union of generated R_k's, to post-execution security policies
- Based on policy application results, data invariants, and program properties, derive monitoring model *M*
- Move *M* into a *Real* (real-user) environment, and use it as a monitoring model, along with a continuous learning process


Real environment
Testing environment
Executed application
User input
Executed application
Automatically generated input combinations
IFetch
Dcd
Exec
Mem
WB
Register file
Light-weight instrumentation
IFetch
Dcd
Exec
Mem
WB
Register file
Dynamic binary instrumentation
Execution trace generation
Execution traces
Control data flow analyzer
Behavioral model B_M
Restrictive security policies
Regular expression extraction
Data invariant generation
Regular expression union
Regular expression generation
Security policy generation
Database 1
Security policies
Database N
Security policies
Detection module
Checkpointing and behavioral model generation
Protected execution environment
Protected execution environment
Model migration hook
Xen/S-Hype
ACM

Outline

- Motivation
- Proposed framework
- Framework details
  - ***Testing* environment**
  - *Real* environment
- Experimental evaluation
- Related work

Execution Traces and Regular Expressions

- Execution trace generation
  - Step built on top of DBI tool Pin
  - Control and data information generated to check against security policies
- Regular expression generation
  - Each execution trace transformed into regular expression, R_k
  - R_k's alphabet: $\sum = \{BB_1, \ldots, BB_n\}$
  - BB_j is a one-to-one mapping to a basic block in the execution trace
  - BB_j contains data components, d_i's, if instruction I_i in basic block executes action A_i
  - d_i's can reveal malicious behavior when they assume specific values

Execution Trace Union

- Completeness of testing procedure depends on number of exposed paths
- Each application tested under multiple automatically- and manually-generated user inputs
- Recursive union of R_k's performed in order to generate R_U

Generation of Data Invariants

- Data invariants
  - Refer to properties assumed by the d_i's in each BB_j
  - Invariant categories:
    - Acceptable or unacceptable constant values
    - Acceptable or unacceptable range limits
    - Acceptable or unacceptable value sets
    - Acceptable or unacceptable functional invariants
  - Data fields, d_i's, over which invariants are defined:
    - Arguments of system calls that involve the modification of a system file or directory
    - Arguments of the "exec" function or any variant thereof
    - Arguments of symbolic and hard links
    - Size and address range of memory access

Generation of Data Invariants *(Contd.)*

- Updating data invariants:
  - Single or multiple invariant types for all d_i's in each BB_j
  - Observe value of all d_i's in each execution trace
  - Start with strictest invariant form (invariant of constant type)
  - Progressively relax stored invariants for each d_i

Security Policies and Malicious Behavior Detection

- Security policy, P_i:
  - P_i specifies fundamental traits of malicious behaviors
  - Each P_i is a translation of a high-level language specification of a series of events
    - If events are executed in a specific sequence, they outline a security violation
  - Malicious behaviors detected by performing $R_U \cap \Sigma(P_i)$
  - Example of P_i

    A malicious modification of an executable, detected post-execution, implies a security violation

Security Policies and Malicious Behavior Detection *(Contd.)*

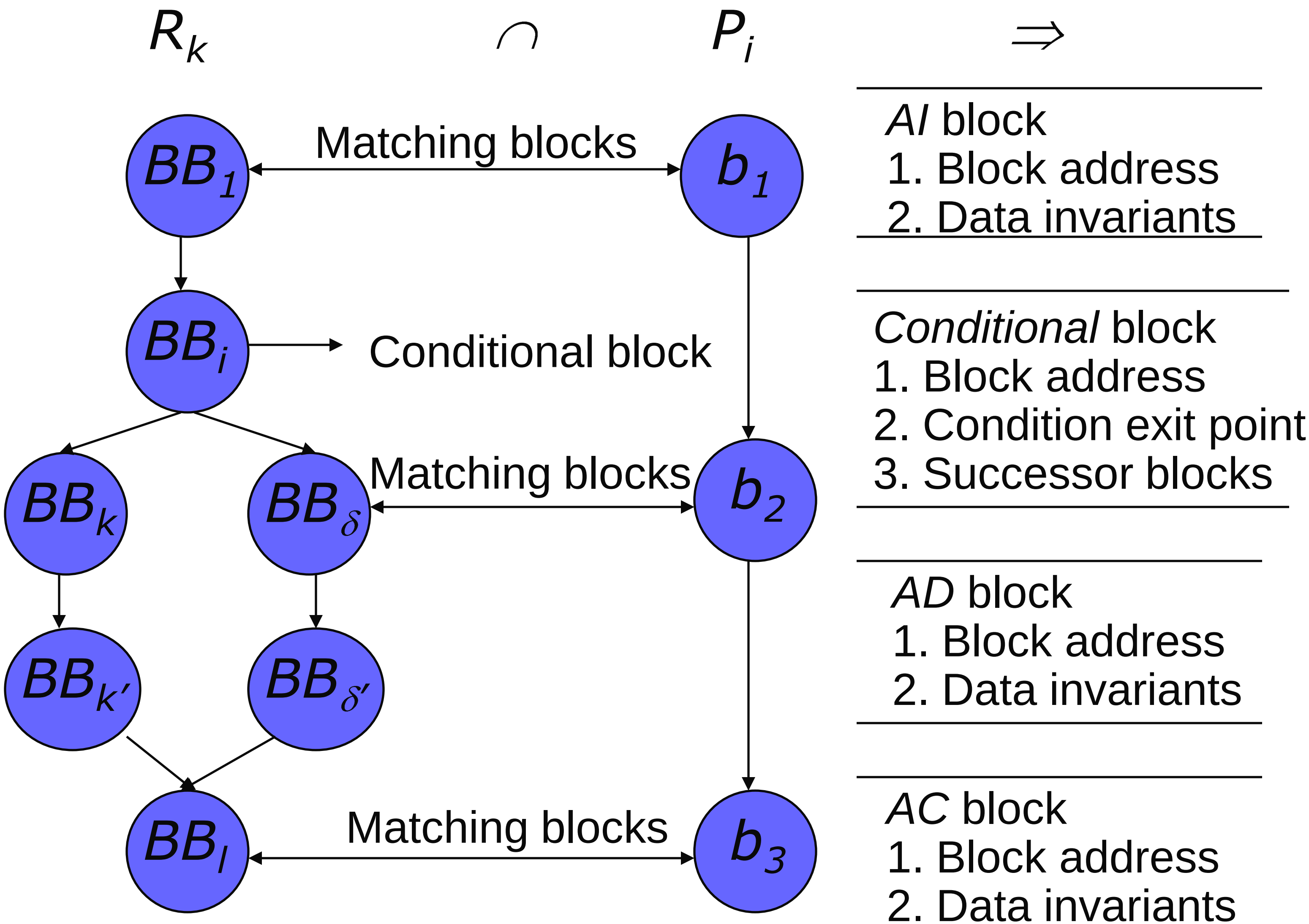
- Malicious modifications include:

  1. File appending, pre-pending, overwriting with virus content
  2. Overwriting executable cavity blocks (*e.g.,* CC-00-99 blocks)
  3. Code regeneration and integration of virus within executable
  4. Executable modifications to incorrect header sizes
  5. Executable modifications to multiple headers
  6. Executable modifications to headers incompatible with their respective sections
  7. Modifications of control transfer to point to malicious code
  8. Modifications of function entry points to point to malicious code (API hooking)
  9. Executable entry point obfuscations
  10. Modifications of Thread Local Storage (TLS) table
  11. Modifications to /proc/pid/exe

Behavioral Model Generation

- Generation of behavioral model, *M*
  - *M* is composed of a reduced set of BB_i blocks
  - *M* embeds permissible or non-permissible real-time behavior
  - Program execution run-time monitored against *M*
  - Blocks included in *M*
    - Anomaly-initiating (*AI*) blocks
    - Anomaly-dependent (*AD*) blocks
    - Anomaly-concluding (*AC*) blocks
    - Conditional blocks
  - Data invariants and flags are added to each block in *M* to instruct an inline monitor what to do at run-time

Example: Deriving M



Outline

- Motivation
- Proposed framework
- Framework details
  - *Testing* environment
  - ***Real* environment**
- Experimental evaluation
- Related work

Framework Details: *Real* Environment

- Run-time monitoring and on-line prevention of malicious code
- Composed of two parts:
  - Check instrumented basic blocks against blocks in behavioral model *M*
  - Check observed data flow against invariants and flags embedded in *M*'s blocks
  - Apply conservative security policies on executed paths not observed in the *Testing* environment

Outline

- Motivation
- Proposed framework
- Framework details
  - *Testing* environment
  - *Real* environment
- **Evaluation results**
- Conclusion

Evaluation Results

- Experimental set-up
  - Prototype on both Linux and Windows-XP operating systems
  - Linux operating system:
    - *Testing* and *Real* environments implemented as two Xen virtual domains
  - Windows-XP operating system:
    - *Testing* and *Real* environments implemented as a custom-installed VMWare virtual Windows-XP operating system image
  - Experiments with 72 real-world Linux viruses and 45 Windows viruses
    - Also obfuscated versions of available viruses

Evaluation Results *(Contd.)*

- Virus detection in the *Testing* environment:
  - Original and obfuscated virus detection rate = 98.59% (Linux), 95.56% (Windows XP)
  - Best commercial antivirus tool:
    - Detected original viruses = 97.22% (Linux), 95.23% (Windows-XP)
    - Detected obfuscated viruses = 50.00% (Linux), 57.14% (Windows-XP)
  - False negatives = 1.41% (Linux), 4.44% (Windows XP)
    - Malicious effects not specified in security policies
  - False positives = 0% (benign programs with behavior resembling that of computer viruses)

Evaluation Results *(Contd.)*

- Virus detection in the *Real* environment:
  - Monitoring against behavioral model halts malicious execution in the *Real* environment
  - Restrictive policies applied 6.8% of the time (i.e., new paths exercised 6.8% of the time)
- Execution time effects:
  - Execution time increases by 26.81X (Linux) and 30.35X (Windows-XP) in the *Testing* environment
    - Does not impose severe limitations on the approach
    - Offline malicious code detection, transparently to the user
  - Execution time increases by 1.20X (Linux) and 1.31X (Windows-XP) in the *Real* environment

Outline

- Motivation
- Proposed framework
- Framework details
  - *Testing* environment
  - *Real* environment
- Evaluation results
- **Conclusion**

Conclusion

- Current techniques fall short of meeting dramatically increasing challenges of malware threats
- New defense mechanism against malware introduced
- Described system successfully detected a high percentage of various malicious behaviors
- Acceptable penalty in the real user environment
- Approach depends on the accuracy of the security policies used

Thank you!