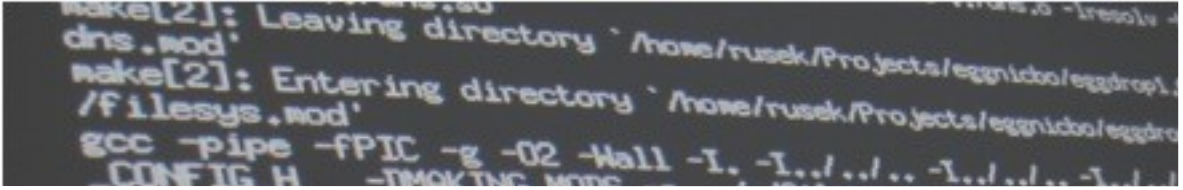


Traps, Events, Emulation and Enforcement: the Yin and Yang of virtualization-based security

Sergey Bratus, Michael E. Locasto,
Ashwin Ramaswamy, Sean W. Smith

Dartmouth College
INSTITUTE FOR SECURITY TECHNOLOGY STUDIES
Cyber Security and Trust Research & Development
http://www.ISTS.dartmouth.edu

Motivation

- *Security through virtualization* is "hot":
  - Xen & its modifications
  - Linux Vservers, Solaris Zones
  - Several VMware products

- PROs:
  - Simpler policies => better usability
  - Isolation is easy and natural to express
    - Compare, e.g., with SELinux types

But, are we going in the right direction?

What's the catch?

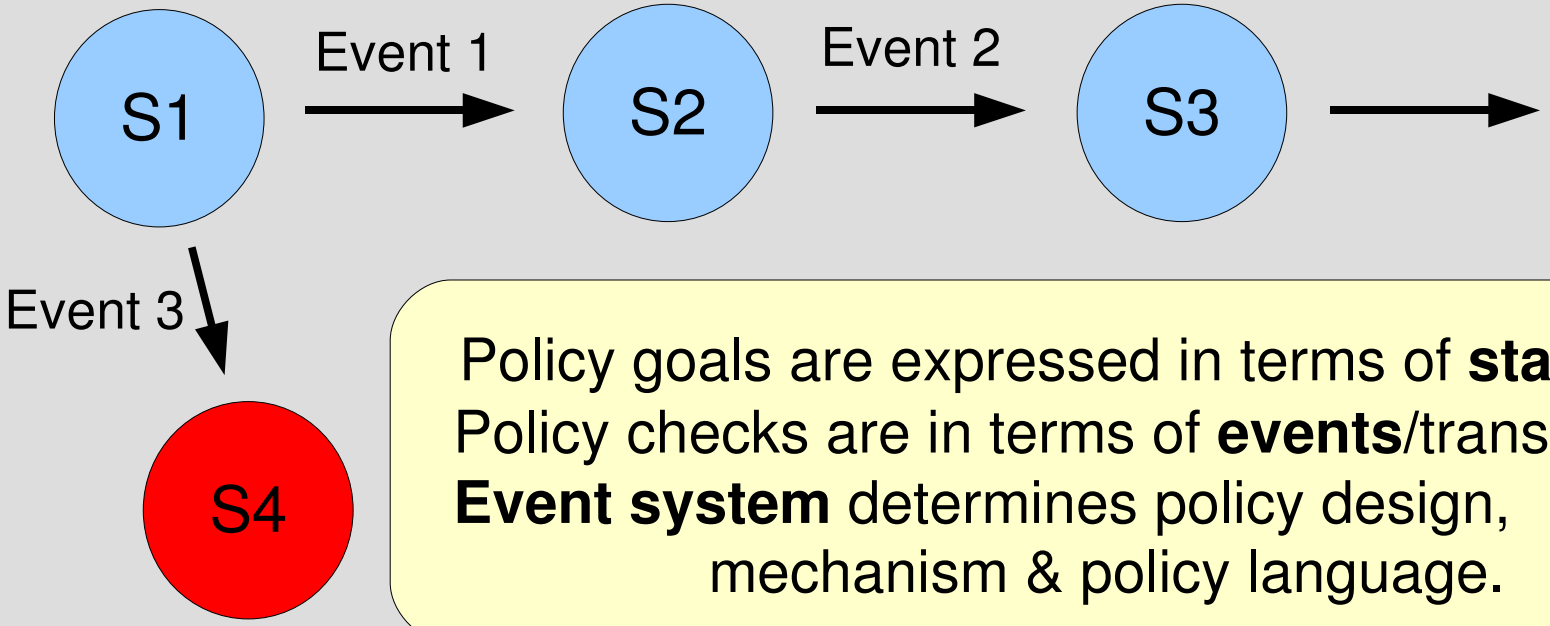
Emulating entire machines comes at a high security price:

- "Virtual devices" drivers bloat in the TCB
  - *Most are irrelevant to security goals*
- Privileged admin & management interfaces
  - *Make VMs easier to manage,* ***but***
  - *Create a new attack surface*
    - *e.g., "VM escape" attacks*

Increased complexity => less trustworthiness

Policy's “trust events”

Policy mechanisms must watch for “trust events”
- such process state transitions that can affect the program's *trustworthiness*

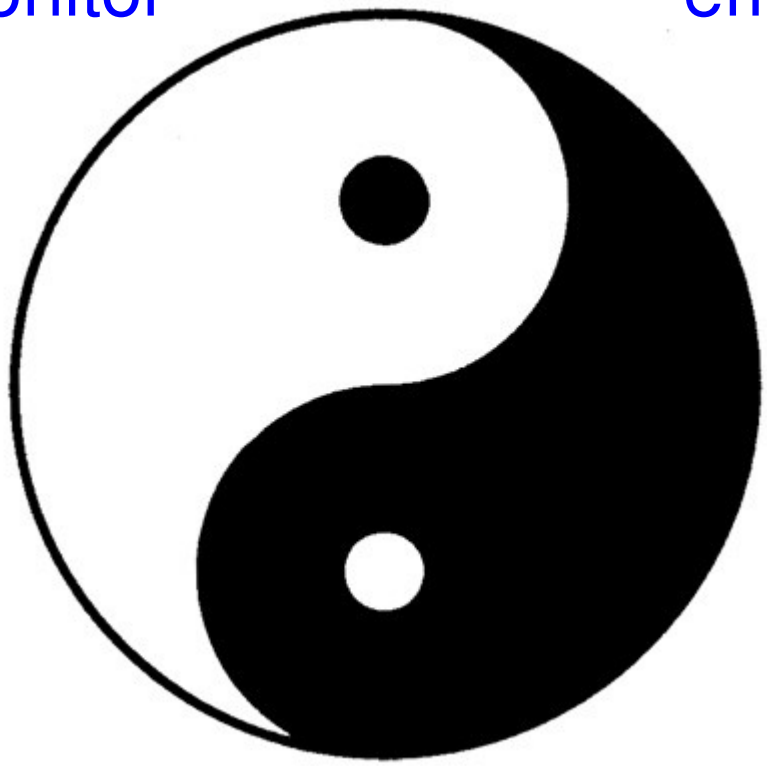


Policy goals are expressed in terms of **states.**
Policy checks are in terms of **events**/transitions.
Event system determines policy design, mechanism & policy language.

Observations

Virtualization's power to isolate & monitor execution comes from trapping HW and OS events

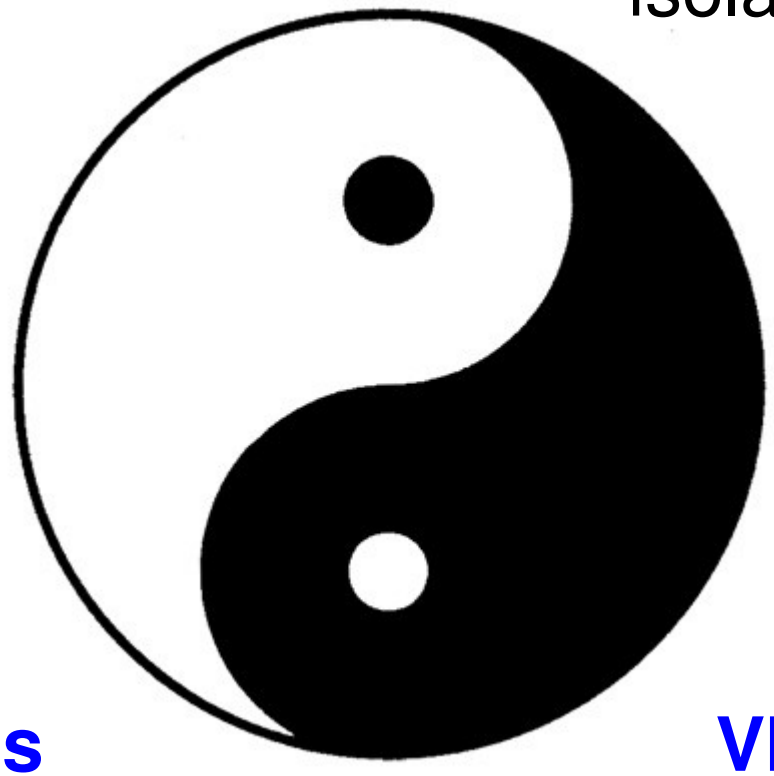
Ability to multiplex & emulate devices comes from trapping HW and OS events



Observations

Isolation & monitoring of processes' execution

Emulation of multiple isolated machines



Security goals

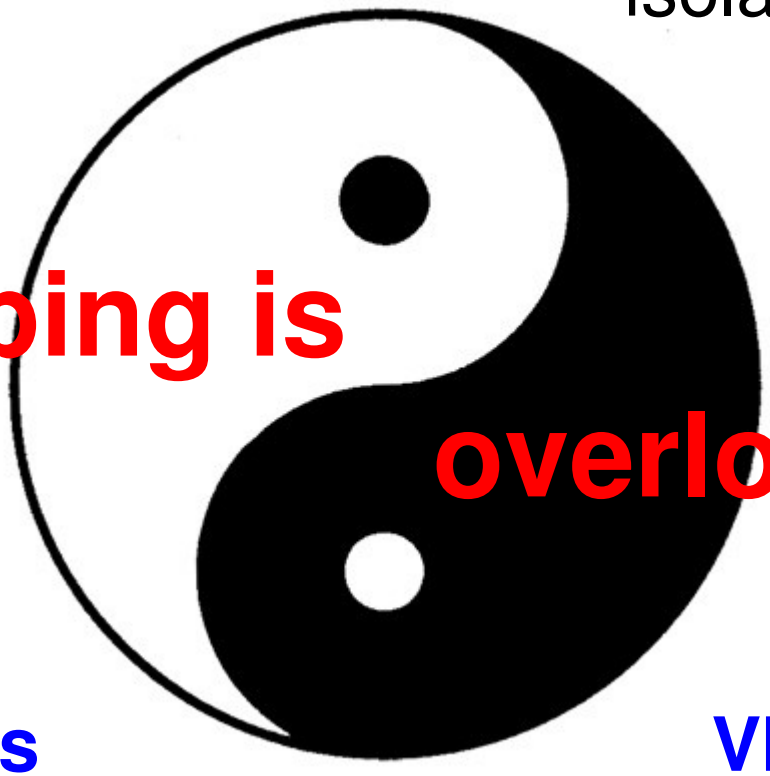
Policy enforcement

VMM acts as a resource provider

Key observation

Isolation & monitoring of processes' execution

Emulation of multiple isolated machines



Trapping is overloaded

Security goals
Policy enforcement

VMM acts as a resource provider

Trapping is overloaded?

"So what?" -- Well...

VMs rely on trapping certain classes of events in order to multiplex physical devices.

This is a major design constraint on the platform's conceptual "event system" and the actual hardware trap mechanisms.

Are all these trapped events important from security perspective?

Are they sufficient to implement flexible policies?

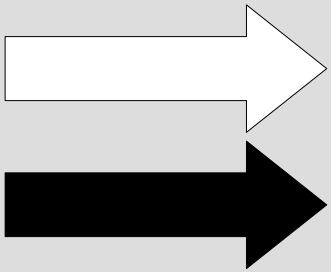
How it happens now

VMs provide isolation and inspection
to achieve security goals

Device multiplexing

Emulation

Isolated virtual machines



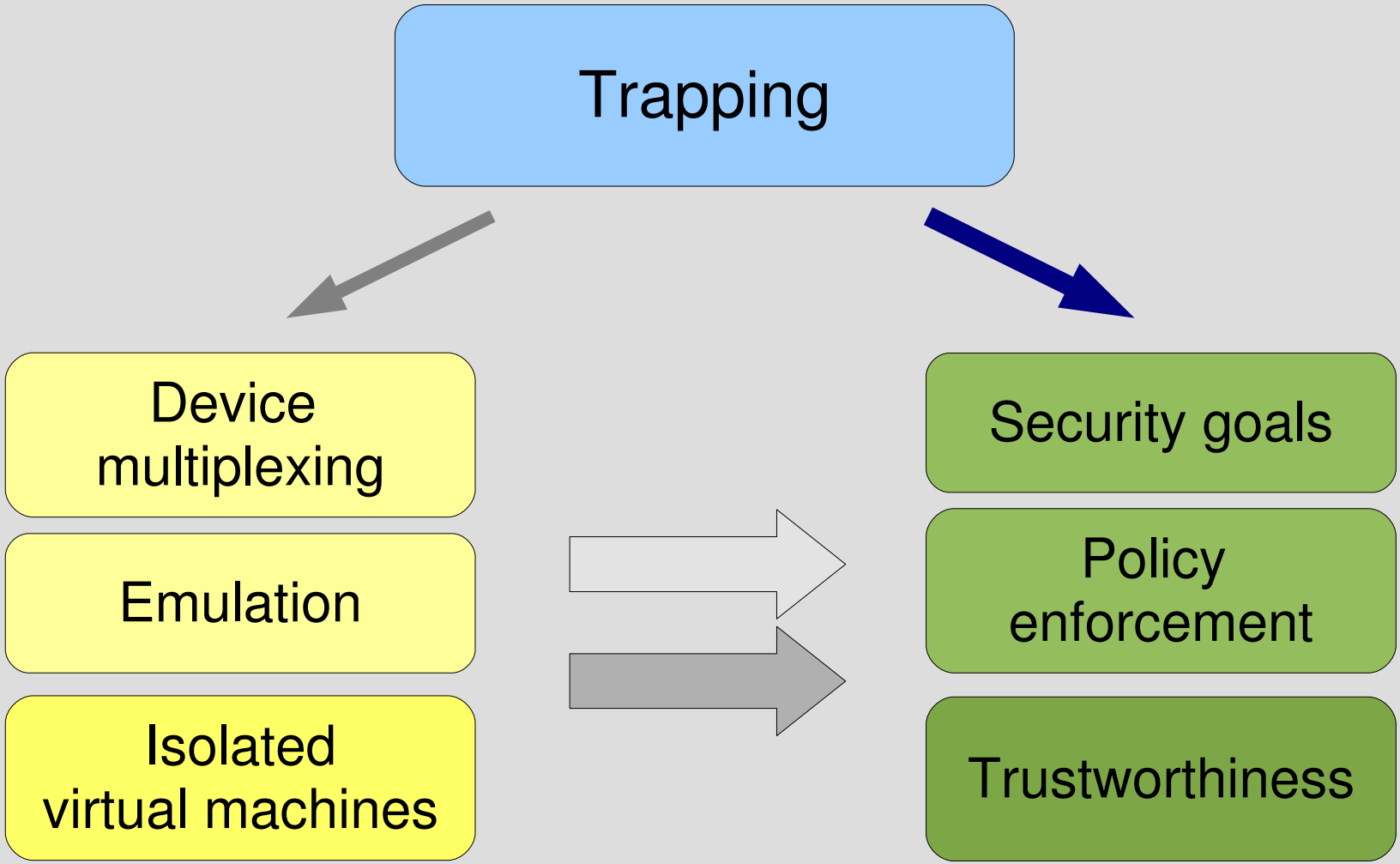
Security goals

Policy enforcement

Trustworthiness

What we want

Dartmouth College
INSTITUTE FOR SECURITY
TECHNOLOGY STUDIES



Our propositions

Dartmouth College
INSTITUTE FOR SECURITY TECHNOLOGY STUDIES

1. Trapping is the foundation of virtualization's power as a security primitive.

2. The need to emulate devices (i.e., entire machines) dilutes this power.

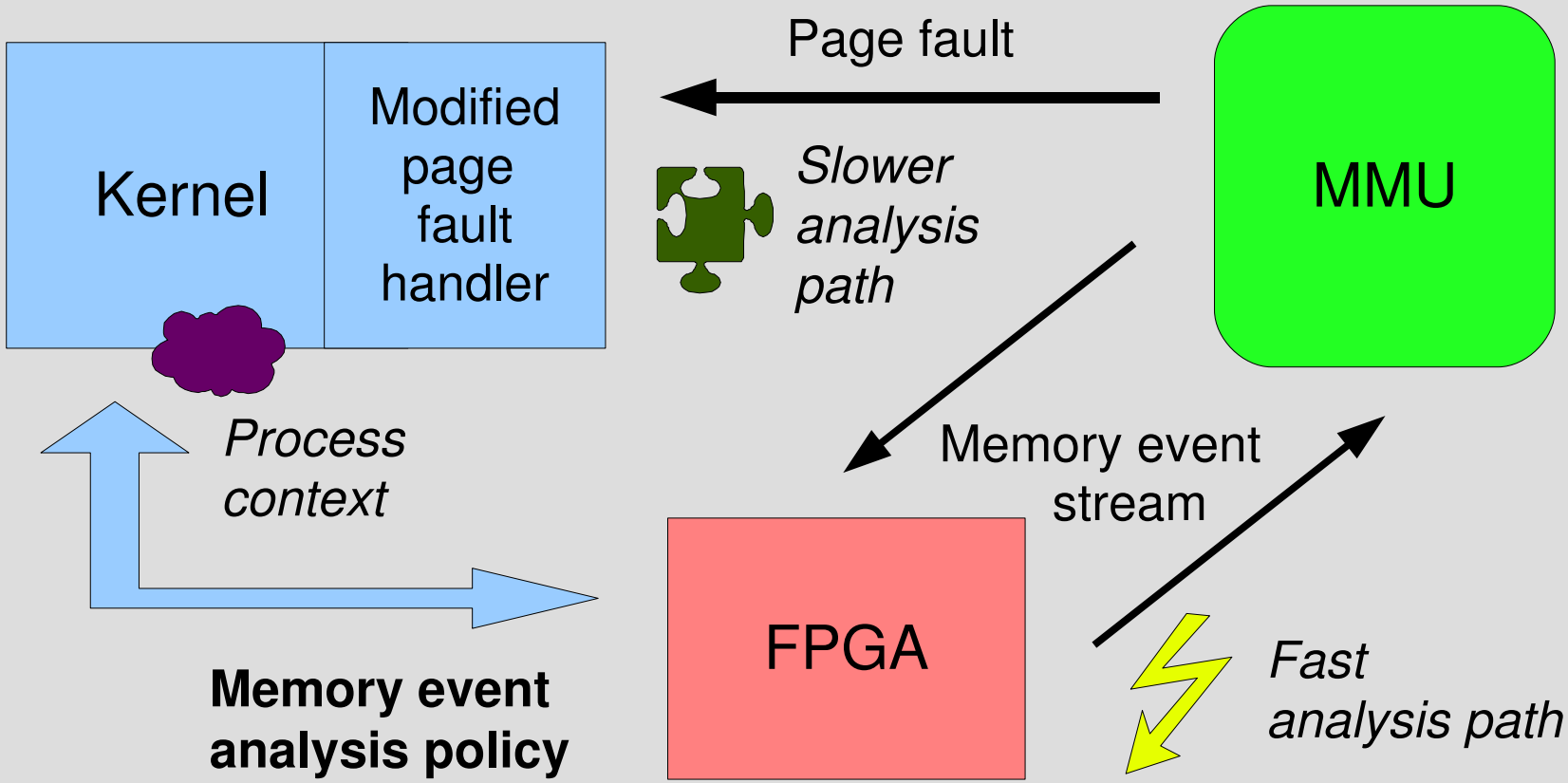
3. Trapping for security policy enforcement should be untangled & separated from emulation proper.

But how? And what kinds of events to trap?

How: the architecture

Dartmouth College
INSTITUTE FOR SECURITY
TECHNOLOGY STUDIES

Trap and dispatch a richer set of events at HW speed – with a memory fault-handling FPGA



“A Better Mousetrap”

- ***“Build it, and the policies will come”***
- FPGA provides non-invasive, low-burden event analysis unit
  - Page-granular “false alarms” get dispatched at faster-than-kernel speed
  - Richer sets of events and contexts (only in debuggers before, slow) – now feasible
  - E.g.: watchpoints that “fire” only under particular conditions & process context

What: a new angle?

The right trap/event system for improving trustworthiness is one that decreases the cost of **debugging** and **runtime program analysis**.

"*What's good for debugging can be useful for policy enforcement, too*"

Why debugging?

Debugging ⟺ Policy?

Dartmouth College
INSTITUTE FOR SECURITY
TECHNOLOGY STUDIES

- Debugging is an activity that establishes the link between expected behavior and actual behavior
- **So does policy!**
- One definition of "Trust" is *relying on the trusted entity to behave in expected ways*
- A "bug" is what breaks programmer's trust!

Developer knowledge?

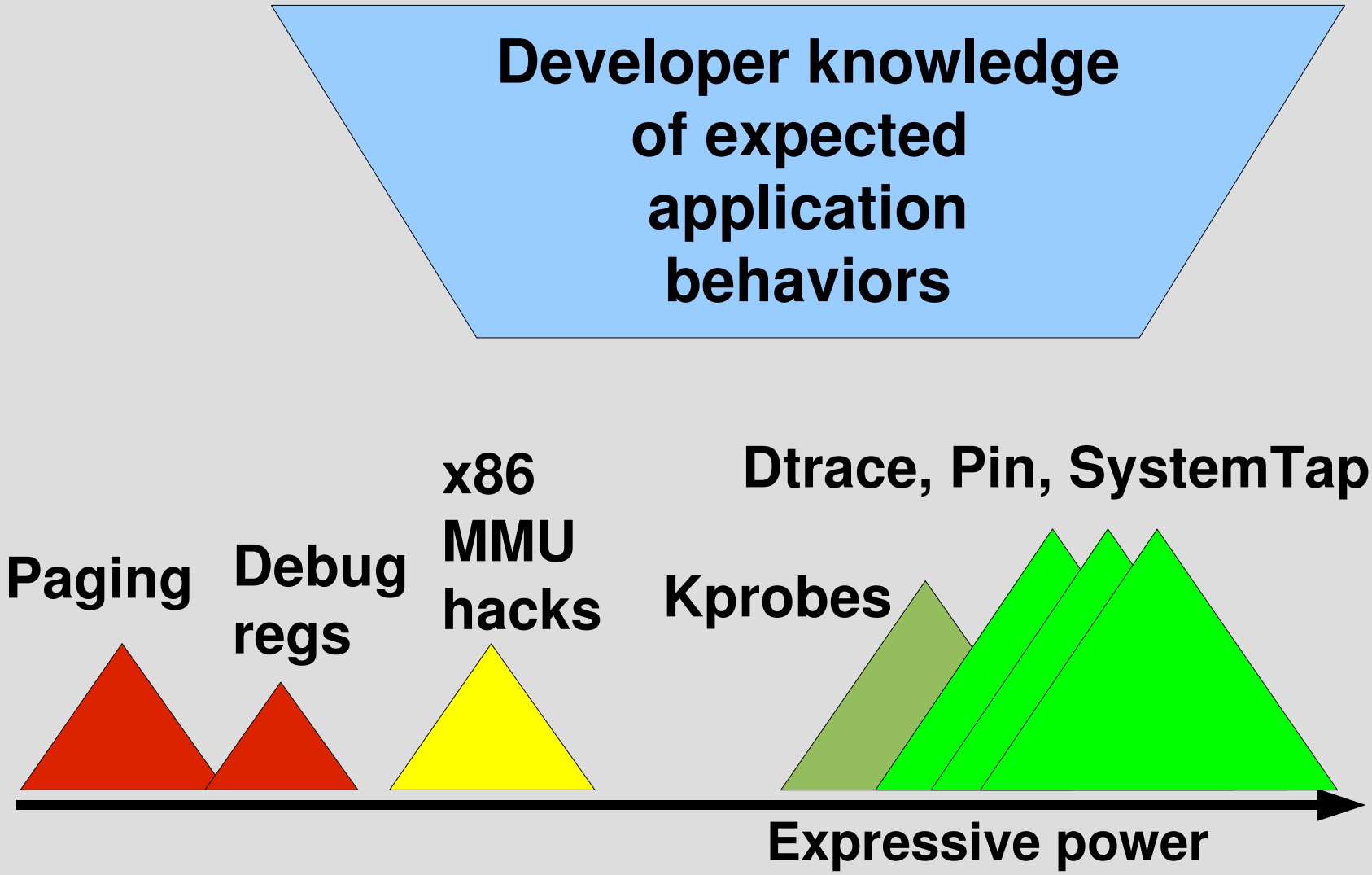
Consider *developer's "worst nightmare" approach* to crafting policy :

- Developer knows his "crown jewels"
- Developer knows his "worst nightmare"
- Often, developer cannot easily impart such data protection priorities/relative importance to runtime environments

Policy that describes only "worst nightmares" for trustworthiness could still improve it a lot.

Expressing developer knowledge

Dartmouth College
INSTITUTE FOR SECURITY TECHNOLOGY STUDIES



A strange disparity

- "*Show me your flow charts and conceal your tables and I shall continue to be mystified, show me your tables and I won't usually need your flow charts; they'll be obvious.*"
  -- Brooks, The Mythical Man-Month
- Yet traditional debugging support is almost entirely control-flow centric, not data-centric
- Watchpoints with predicates are arguably the biggest disappointment of a novice debugger user - too slow in practice
  (*"May still be worth it"* -- GDB manual)

x86 hacks: examples

- Page-granular read/write/exec permission bits
- x86 segmentation (PaX, OpenWall)
- "Invalid" bit in PTEs, PDEs (UML, Xen)
  - used with overloaded PageFault handler
- Split TLB: separate Icache, Dcache (OllyBone)
  - "***execute from a location after a write***"
  - also used by ShadowWalker rootkit

All of these and more are used to express
combinations of elementary trap conditions

Lessons from prominent designs

The choice of the underlying event / trap system dictates a lot about the policies.

- SELinux type enforcement
- UNIX daemon privilege drop support

SELinux: an example

- Mediates system calls as “trust events”
  - Syscalls are privileged ops -- can indeed change system's trustworthiness
  - But: *not all trust events are syscalls:* e.g., memory object read/writes aren't
  - **`Sensitive, trusted' ≠ `held by the kernel'**
- The only state info kept about a process is in its type (label)
  - When process enters a new phase, it must execve(2) or setcon(3) to a new type

Privilege drop: an example

- Expected UNIX daemon behavior is to not make privileged syscalls after a certain phase
- Privilege drop ensures that if it does, this event gets caught
- Interpretation: *the daemon process is then no longer trustworthy*

  – This case of "least privilege" is *special*:
  It recognizes that *not all* privileges are *equally important* (compare to SELinux)

"Policy/trust events"

Some events are critical for trustworthiness,
but <u>really expensive</u> to **trap** & **mediate**:

- writes to crucial data objects in RAM
- counts or order of operations on objects
  - ***"100th write to variable X"***
  - ***"write to variable X after event Y occurred"***

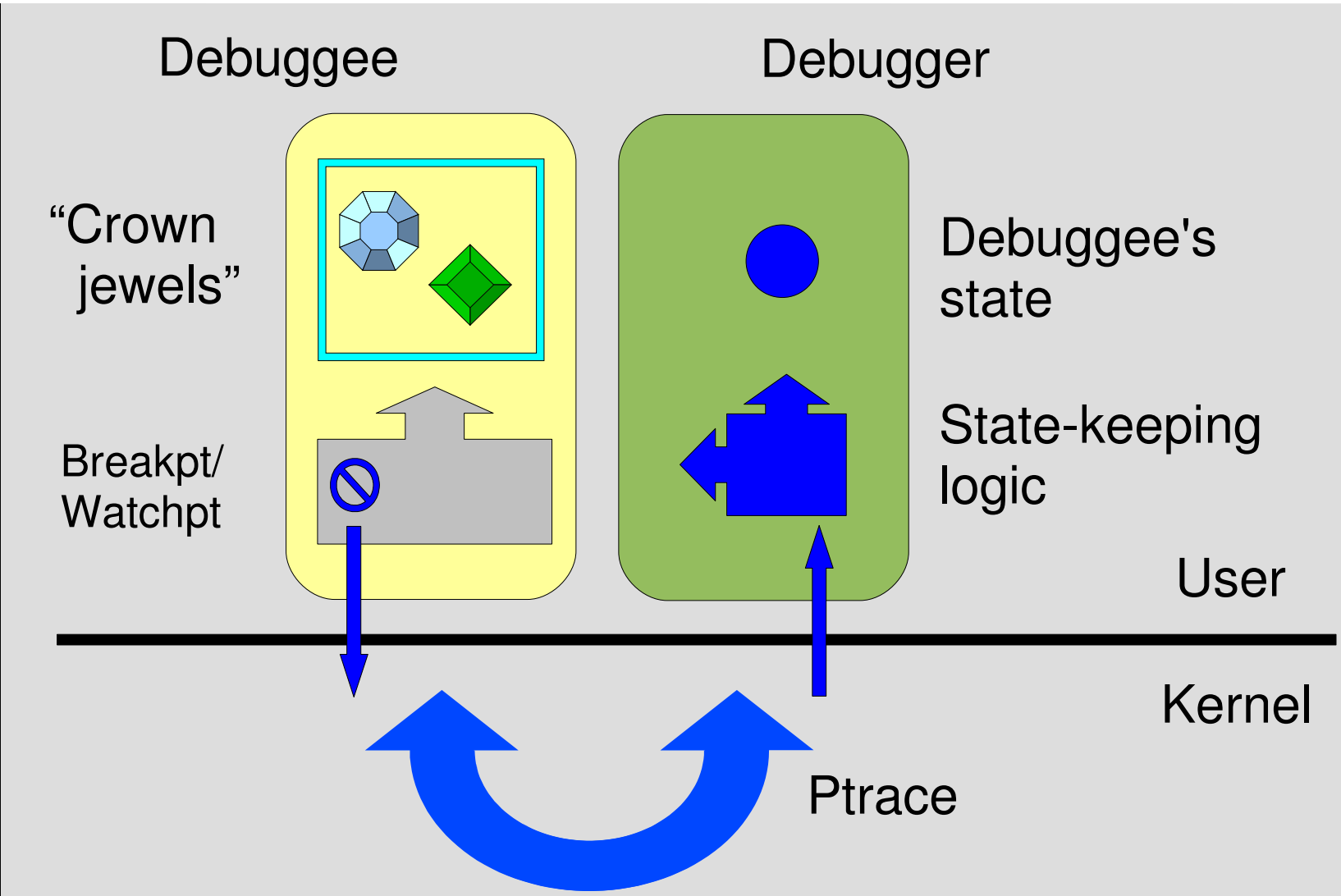
That is, not arbitrary asynchronous OS-level events or all system calls, but rather:

"What the developer trusts to **not happen**"

"What the developer trusts to **happen**"

State in a debugger

Dartmouth College
INSTITUTE FOR SECURITY
TECHNOLOGY STUDIES



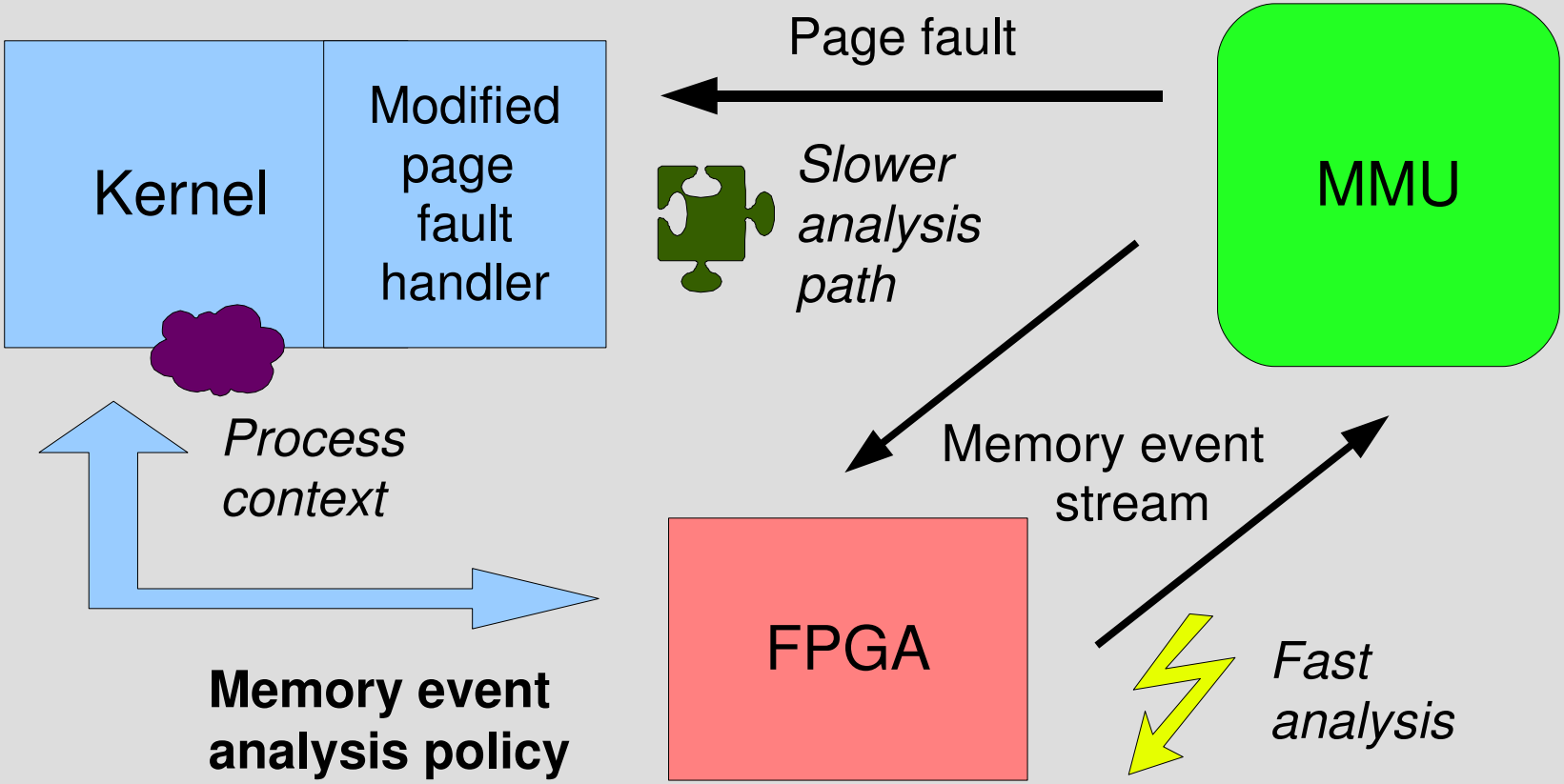
State-keeping quandary

- State is very useful
  - Allows expression of fine properties, much closer to app logic/policy concepts than any existing trap semantics (e.g., write/read/execute at addr)
- DTrace keeps probe state and logic (byte-compiled) in the OS kernel
  - -- complex predicates, can reference user and OS level objects
  - -- still too slow (probes can be skipped)

Efficient stateful trapping

Dartmouth College
INSTITUTE FOR SECURITY TECHNOLOGY STUDIES

"**A Better Mousetrap**": let the FPGA hold state data and handle state-related policy logic.



The Bright Future

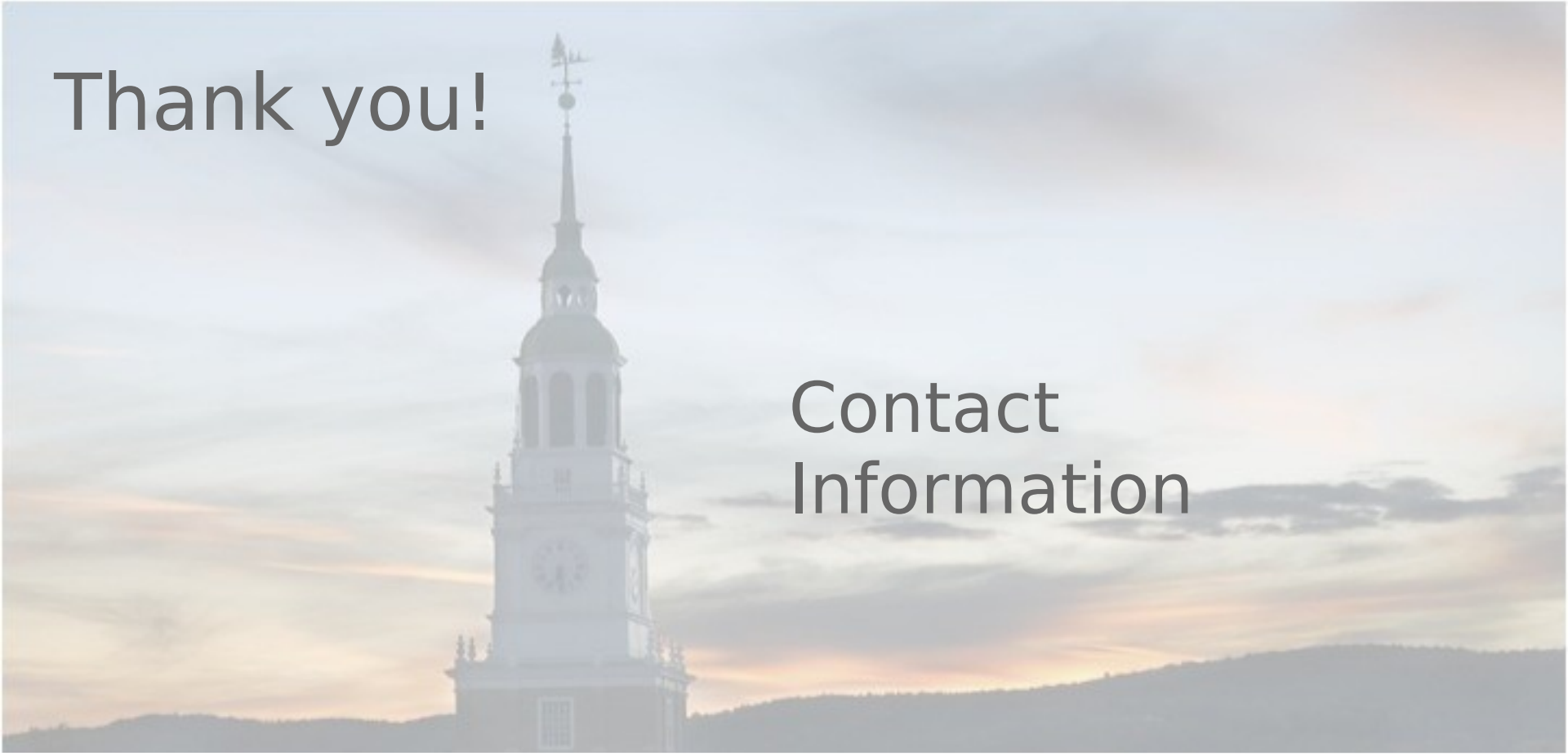
- Policy designers can naturally express many security goals as conditions for a tracing debugger to check
- Recall: trust ~ expected behaviors
  - *Thinking in terms of expected behaviors is natural for developers*
- The FPGA makes it possible to trace a set of such conditions efficiently
  - place to store necessary state info
  - fast logic to update and check it

Summary (1)

- Virtualization is not the real answer for better security policies - it's something that underlies virtualization
- VMM traps and security-related traps (including those that support debugging) must be conceptually separated
- Hardware needs to support more flexible, debugger script-like memory traps

Summary (2)

- There is a considerable policy engineering gap: developers cannot easily express their application knowledge with current trap semantics
- Give them tools to express richer event & context conditions, debugger-style
  - must find place to keep state
  - must process it fast (context-switching ptrace is a non-starter, dtrace is still too slow)
- Richer trap semantics => better trustworthiness!



Thank you!

Contact Information

Institute for Security Technology Studies
&
PKI/Trust Lab
Dartmouth College
6211 Sudikoff Laboratory
Hanover, NH 03755

info@ists.dartmouth.edu

Monitoring events

- Once developer decides which trust events are key, they must be monitored **cheaply** and **inline** (i.e. mediated)

- If audit-style policy is acceptable, then **a DTrace-like tracer can do policy:**
  - RE::Trace, RE::Dbg (DTrace extensions)
  - monitor data structures for changes
  - halt process (asynchronously) if basic trust assumptions are violated (overflow, improper memory use, ...)