Whole-System Dynamic Binary Analysis

Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis

Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, Engin Kirda,

Appeared in CCS'07

Outline

- Motivation
- Overview
- Design & Implementation: Panorama
- Taint-Graph Based Detection and Analysis
- Evaluation
- Summary

Motivation I -- Problem

- Malicious code creeps into users' computers, performs malicious behaviors
  - spyware/adware
  - keyloggers
  - password thieves
  - network sniffers
  - backdoors
  - rootkits
- Even software from reputable vendors
  - Google Desktop
  - SONY Media Player

Motivation II – Previous Solutions

- Malware Detection
  - Signature based
    - Cannot detect new malware and variants
    - Semantic-aware signatures can detect some variants
  - Behavior based
    - Heuristics: high false positives and false negatives
    - Strider Gatekeeper checks auto-start extensibility points
    - VICE and System Virginity Verifier check various hooks
- Malware Analysis
  - Manual process mostly
  - Coarse-grained

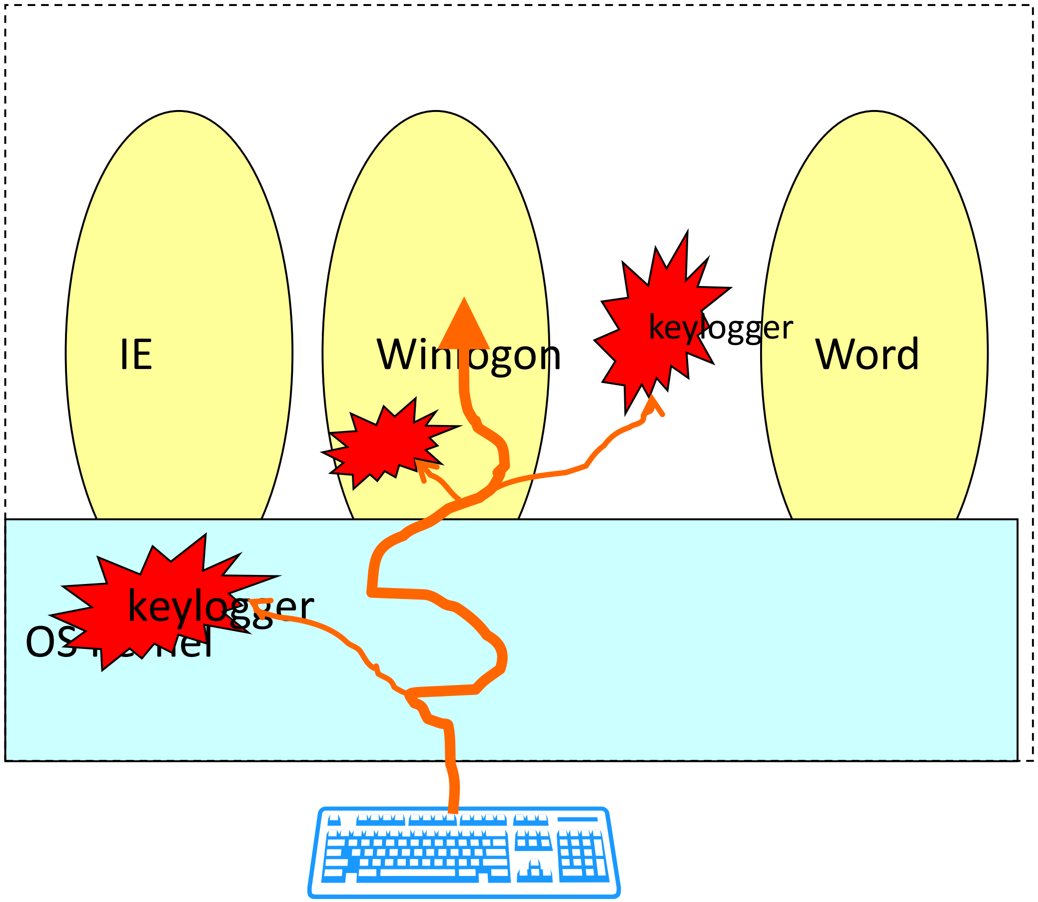
Outline

- Challenges & Motivation
- Overview
- Design & Implementation: Panorama
- Taint-Graph Based Detection and Analysis
- Evaluation
- Summary

Overview I – Our Observation

- Information access and processing (IAP) behavior
  - Many different kinds of malware present malicious/suspicious IAP behavior
  - Steal, tamper, or leak sensitive information
    - Spyware leaks URLs
    - Keyloggers steals keystroke information
    - Password thieves steals passwords
    - Rootkits tamper with directory information
    - Network sniffers eavesdrop the network traffic

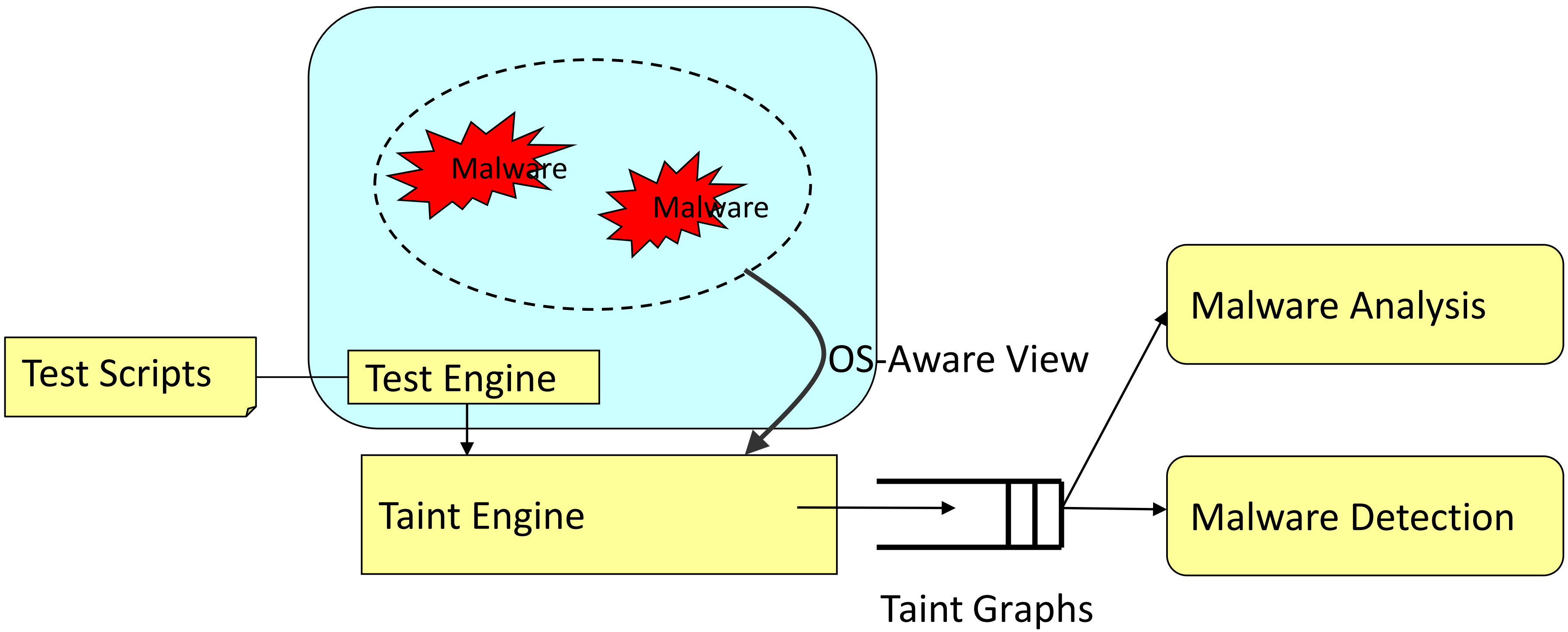
Overview II - A Example



Overview III – Our Approach

- Whole-system dynamic taint analysis with OS awareness
  - Run the system to be analyzed in an emulator
  - Selectively mark data as tainted
  - Monitor taint propagation
  - Extract OS-level knowledge
  - Generate taint graphs
  - Taint-graph based detection and analysis

Overview II – Big Picture



Outline

- Motivation
- Overview
- Design & Implementation: Panorama
    - Hardware-level Dynamic taint analysis
    - OS-aware Analysis
    - Automated testing
- Taint-Graph Based Detection and Analysis
- Evaluation
- Summary

Design & Implementation – Hardware Level Taint Analysis

- Build on QEMU
- Shadow Memory
  - RAM, registers, hard disk, and NIC buffer
  - Page-table-like structure
- Extend CPU
  - Propagate taint status for each instruction
- Extend Kbd, Disk and NIC
  - Taint inputs
  - For disk, propagate taint status

Design & Implementation – Hardware-Level Taint Analysis (2)

- Instrument CPU Instructions (at byte granularity)
  - Movement: MOV AL, BH
    -- AL is tainted iff BH is tainted
  - Arithmetic: ADD EAX, EBX
    -- EAX is tainted iff EAX or EBX is tainted
  - Table lookup: MOV EAX, [EBX]
    -- EAX is tainted if EBX or MEM[EBX] is tainted)
  - Constant function: XOR EAX, EAX
    -- EAX will be untainted

Design & Implementation – OS-Aware Analysis

- Resolving process and module information
  - Q: when an instruction accesses taint, which process and module is it from?
  - A: A kernel module is inserted into the guest system
- Resolving filesystem information
  - Q1: when tainting a file/directory, which disk blocks should be tainted?
  - Q2: when the tainted data propagate to a disk block, while file is tainted?
  - A: The Sleuth Kit (TSK), a disk forensic tool
- Resolving network information
  - Q1: When tainting an incoming packet, which connection is it from?
  - Q2: when a tainted byte is sent out, which connection is it from?
  - A: Simply check the packet header

Design & Implementation – OS-Aware Analysis (2)

- How to identify the actions performed by the code sample?
- Challenge 1: packed code and encrypted code
- A: taint the binary file with a special label
- Challenge 2: call a function in the system libraries
- A:
  - check stack pointers
  - Check asynchronous kernel functions

Design & Implementation – Automated Testing

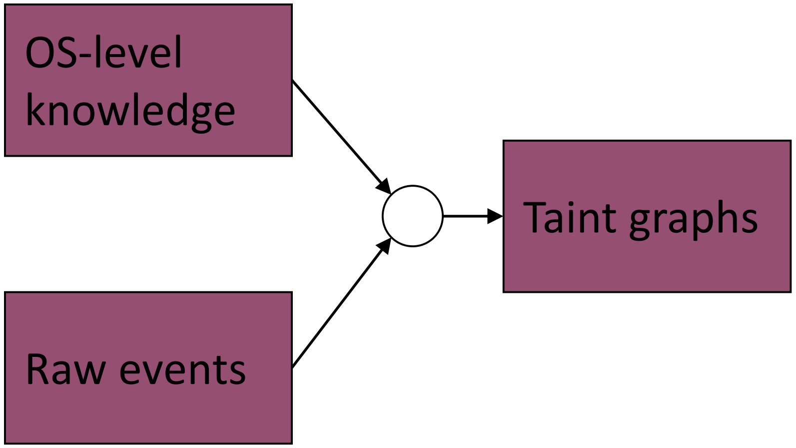
- Goal
  - Perform test cases without human intervention
  - Introduce tainted information sources
- We use "AutoHotkey"
  - Record the test cases into scripts
  - Replay the scripts in Panorama
  - Will describe the test cases later

Outline

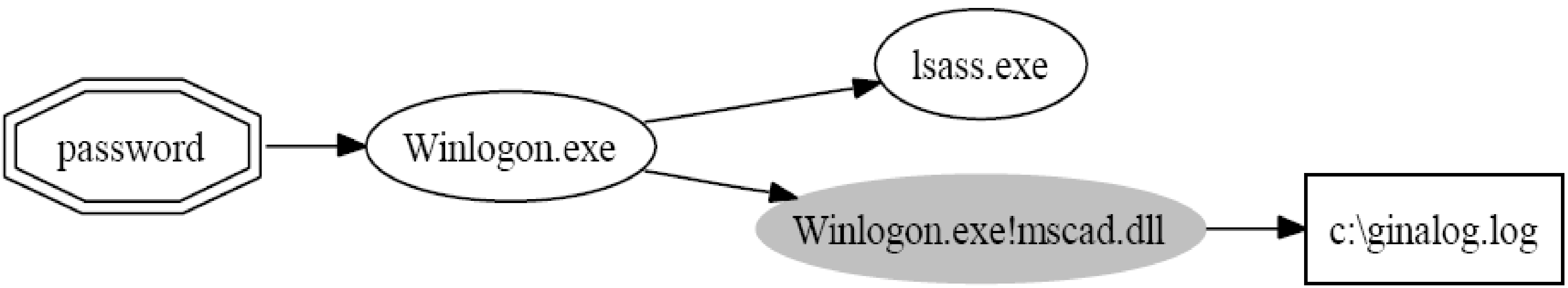
- Motivation
- Overview
- Design & Implementation: Panorama
- Taint-Graph Based Detection and Analysis
  - Taint Graph
  - Taint-Graph Based Policies
- Evaluation
- Summary

Detection & Analysis – Taint Graph

- Taint Graph
  - Input 1: Raw events present dependencies among instructions, hardware inputs and outputs
  - Input 2: OS-level Knowledge
  - Output: taint graph



Detection & Analysis – Taint Graph(2)



- An example of taint graph
  - This graph reflects the procedure for Windows user authentication.
  - A password thief catches the password and saves them into a log file

Detection & Analysis – Taint-Graph Based Detection

- Anomalous information access
  - *text*: when sending keystrokes to a text editor, a command console, keyloggers …
  - *password:* when sending passwords to a web form, a password field, password thieves and keyloggers…
  - *ICMP*: when pinging a remote host, packet sniffers and stealth backdoors …
  - *FTP*: when logging into an FTP server, packet sniffers and stealth backdoors …
  - UDP: when sending in a UDP packet, packet sniffers and stealth backdoors …
  - Others: …

Detection & Analysis – Taint-Graph Based Detection (2)

- Anomalous information leakage
  - *URL*: the keystrokes sent to the address bar,
  - *HTTP:* the incoming HTTP traffic,
  - HTTPS: the incoming HTTPS traffic,
  - *document:* .txt, .pdf, .ppt, .doc
  - Others: …

Detection & Analysis – Taint-Graph Based Detection (3)

- Excessive information Access
  - *directory*: when recursively listing several directories, the disk blocks belonging to the directories
  - Rootkits will access all of the disk blocks and tamper with some entries
  - Compared with Cross-view based techniques, such as Rootkit Revealer, Blacklight, and Strider Ghostbuster, …

Detection & Analysis – Taint-Graph Based Detection(4)

Test case description	Introduced inputs
1. Edit a text file and save it	text, document
2. Enter password in a GUI program	password
3. Log in a secure website	URL, password, HTTPS
4. Visit several websites	URL, HTTP
5. Log into an FTP server	text, password, FTP
6. Recursively list a directory	directory
7. Send UDP packets into the system	UDP
8. Ping a remote host	ICMP

Detection & Analysis -- Taint-Graph Based Detection

$$\forall g \in G, (\exists v \in g.V, v.type = \texttt{module}) \land$$
$$g.root.type \in \{\texttt{text}, \texttt{password}, \texttt{FTP}, \texttt{UDP}, \texttt{ICMP}\}$$
$$\rightarrow Violate(v, \text{``}\texttt{No Access}\text{''}) \quad (1)$$

$$\exists g \in G, (\exists v \in g.V, v.type = \texttt{module}) \land$$
$$(g.root.type \in \{\texttt{URL}, \texttt{HTTP}, \texttt{HTTPS}, \texttt{document}\}) \land$$
$$(\exists u \in descendants(v), u.type \in \{\texttt{file}, \texttt{network}\})$$
$$\rightarrow Violate(v, \text{``}\texttt{No Leakage!}\text{''}); \quad (2)$$

$$(\forall g \in G,\ g.root.type = \texttt{directory} \rightarrow \exists v \in g.V, v.type = \texttt{module})$$
$$\rightarrow Violate(v, \text{``}\texttt{No Excessive Access}\text{''}) \quad (3)$$

Outline

- Motivation
- Overview
- Design & Implementation: Panorama
- Taint-Graph Based Detection and Analysis
- Evaluation
  - Malware detection
  - Malware analysis
  - Performance
- Summary

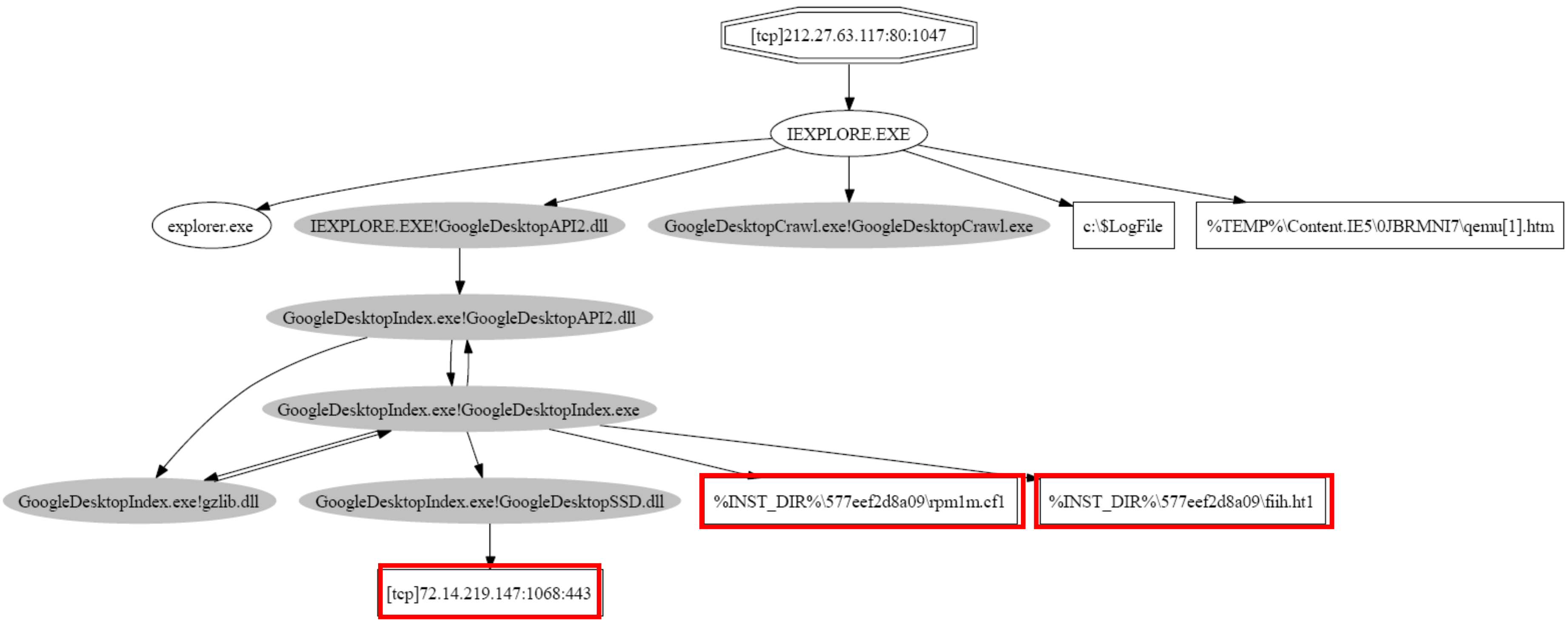
Evaluation – Malware Detection

Category	Total	FNs	FPs
Keyloggers	5	0	-
Password thieves	2	0	-
Network sniffers	2	0	-
Stealth backdoors	3	0	-
Spyware/adware	22	0	-
Rootkits	8	0	-
Browser plugins	16	-	1
Multi-media	9	-	0
Security	10	-	2
System utilities	9	-	0
Office productivity	4	-	0
Games	4	-	0
Others	4	-	0
Sum	98	0	3

Browser accelerator

Personal firewall

Evaluatoin -- Malware Analysis



Google Desktop obtains the incoming HTTP traffic, saves it into two index files, and then sends it out though an HTTPS connection, to a remote Google Server

Evaluation – Performance

- curl, scp, gzip, bzip2: 20 times slowdown on average
- Test cases: 10~15 mins
- Performance improvement:
    - On-demand emulation
    - Static analysis

Summary

- Propose to rely on IAP behavior to detect and analyze malware
  - No signature is required: can detect new malware
  - Stems from intent: difficult to evade
  - Fine grained analysis
  - Capture the behaviors of kernel-level attacks
- Propose to use the technique of whole-system dynamic taint analysis with OS-awareness to capture IAP behavior
- Design and develop a system Panorama
  - Yields no false negative and very few false positives
  - Correctly capture the behavior of Google Desktop

Make It Work, Make It Right, Make It Fast: Building a Platform-Neutral Whole-System Dynamic Binary Analysis Platform

Andrew Henderson*, Aravind Pravash*, Lok Kwong Yan†,

Xunchao Hu*, Xujiewen Wang*, Rundong Zhou*, Heng Yin*

* Department of EECS, Syracuse University

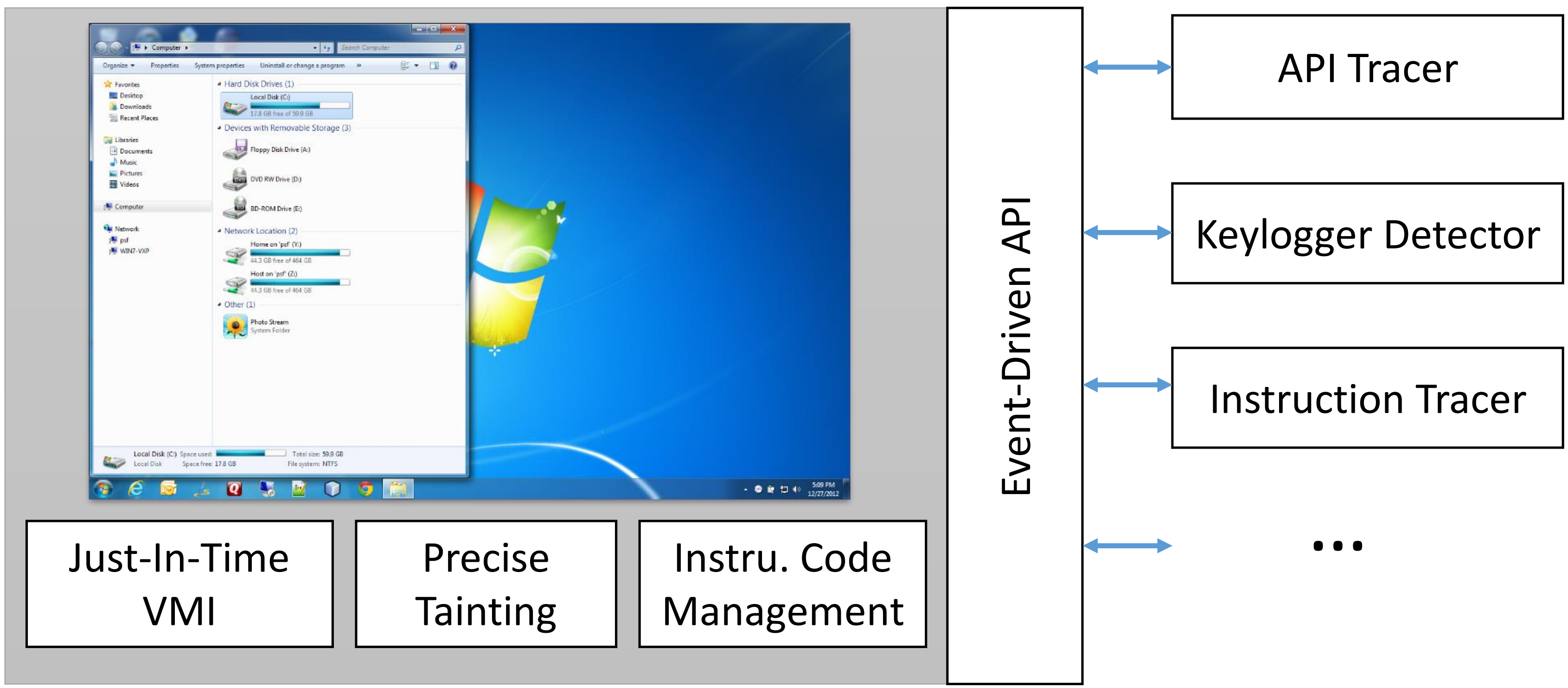
† Air Force Research Laboratory, Rome

Motivation: We need a practical solution for platform-neutral whole-system binary analysis

- **Binary analysis of malware**
  - No source code available to us
  - Need to analyze malicious binary activity
- **Whole system**
  - Multiple components in both userspace and kernel
- **Platform-neutral (as much as possible)**
  - Architecture neutral
  - Guest OS neutral

DECAF: System Architecture

DECAF and Guest Environment **Plugins**



Does DECAF work?

- Sycure Lab (Syracuse University) actively uses DECAF for our cybersecurity research efforts
- Sycure Lab team is using DECAF for the Cyber Grand Challenge competition
- McAfee currently uses DECAF to detect and analyze keylogger malware behaviors
- Numerous other academic labs are currently utilizing DECAF in their own research efforts

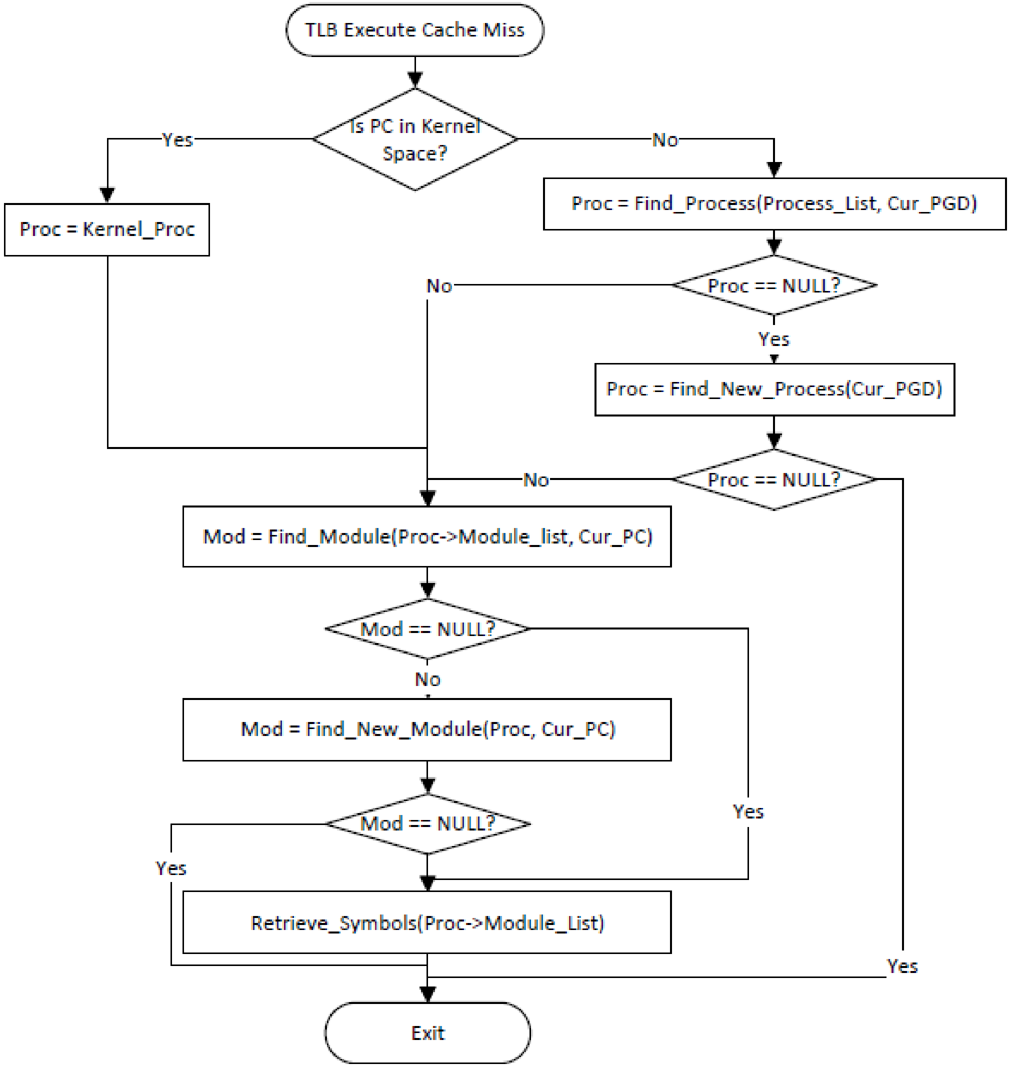
Just-In-Time VMI

- **Virtual machine introspection (VMI)**
  - Inspect the guest environment from the outside
  - Bridge the "semantic gap"
- **Other VMI implementations focus on *how*, not *when***
  - We must be aware of changes within the guest *when those changes occur*
- **VMI must be as platform-neutral as possible**
- **VMI must introduce *minimal overhead***

Just-In-Time VMI

- **Observation 1: A process must have its own memory space**
  - Each CPU architecture provides a register to store the “base” of these memory spaces (CR3 in x86, CP15 in ARM, etc.)
- **Observation 2: The translation look-aside buffer (TLB) reveals information about guest behavior**
  - An “execute” cache miss will occur when new code pages are loaded and executed (new process, loading shared libraries, context switch)
- **Observation 3: Location and structure of key kernel data structures are known**
  - Kernel contains linked lists of modules, processes, threads
- **Result: Rely on hardware events to discover “when” and “what”, rely on kernel data for “who”**

Just-In-Time VMI: Solution



- TLB Miss triggers VMI
- PC tells us where event occurred
- Guest kernel data structures give more detail
- Other systems perform VMI using guest software:
  - Hook system calls
  - Use kernel module
  - Use custom device driver
  - Increases dependence on guest platform

Tainting

- **Tainting must be *whole-system***
  - Tainted data should be trackable throughout the entire guest environment (kernel, processes, devices)
- **Tainting policy must be *sound and precise***
  - Minimize under- and over-tainting of data
  - We performed formal verification of our taint policy correctness at the instruction level [1]
- **Tainting must be *fast***

[1] L. K. Yan, A. Henderson, X. Hu, H. Yin, S. McCamant. On soundness and precision of dynamic taint analysis. Technical Report SYR-EECS-2014-04, Syracuse University, 2014.

Tainting: Using QEMU for propagation

- **QEMU's Tiny Code Generator (TCG) is a binary translator**
  - Guest CPU instructions are translated into *intermediary representation (IR) instructions*
  - TCG's IR instruction set implements standard CPU operations that all instruction sets have (MOV, ADD, XOR, etc.)
  - These IRs and then translated into host CPU instructions
- **Execution details of the IRs and their arguments are invisible to the guest**

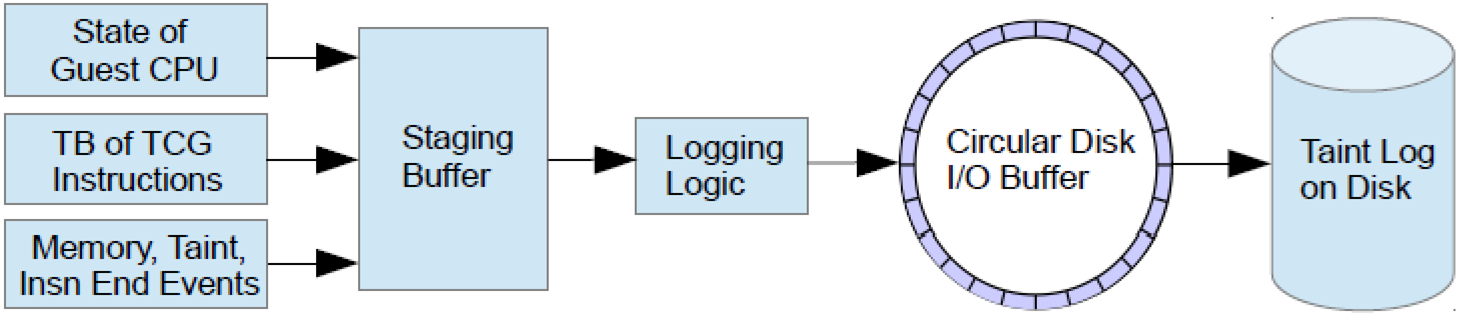
Tainting: Lightweight inline propagation

```
mov $0x8f, %eax
and  $0x01, %eax
```

- Begin with guest instructions
- Translate guest instructions into IR
- Analyze each IR to determine taint rule to apply
- Insert taint propagation IRs

```
movi_i32  taint_eax, $0x0
movi_i32  eax, $0x8f
movi_i32  tmp21, $0x0
movi_i32  tmp11, $0x01
mov_i32   tmp23, taint_eax
mov_i32   tmp13, eax
not_i32   tmp30, tmp21
and_i32   tmp31, tmp11, tmp21
and_i32   tmp32, tmp30, tmp31
not_i32   tmp30, tmp22
and_i32   tmp31, tmp21, tmp13
and_i32   tmp33, tmp30, tmp31
and_i32   tmp30, tmp21, tmp22
or_i32    tmp31, tmp32, tmp33
or_i32    tmp23, tmp30, tmp31
and_i32   tmp12, tmp11, tmp13
```

Tainting: Heavyweight plugin propagation



- Taint *state* is propagated inline via IRs
- When tainted data is present, the IRs can be logged to disk via a plugin
- Taint tags are written to this log when created
- The generated log is sliced backward to reconcile taint with its source tag

Event-Driven Instrumentation

- **Instrumentation occurs at two points:**
  - Translation-time
  - Runtime
- **At translation time, callbacks are embedded in the TCG IR stream**
- **At runtime, DECAF uses a dispatch mechanism to route these callbacks to plugins**
- **Example: Shared library**
  - Are we in the right process?
  - Should the plugin’s callback be triggered?

Event-Driven Instrumentation: Translation time

```
orl  %ebx, %eax
     ...
```

```
movi_i32    tmp21, $<CURRENT_ADDRESS>
movi_i32    tmp22, $DECAF_invoke_block_begin_callback
call        tmp22, $0x0, $0, env, tmp21
movi_i32    tmp23, $DECAF_invoke_insn_begin_callback
call        tmp23, $0x0 $0, env
mov_i32     tmp11, ebx
mov_i32     tmp12, eax
or_i32      tmp13, tmp12, tmp11
movi_i32    tmp26, $DECAF_invoke_insn_end_callback
call        tmp26, $0x0 $0, env
                         ...
movi_i32    tmp27, $DECAF_invoke_block_end_callback
call        tmp27, $0x0, $0, env
```

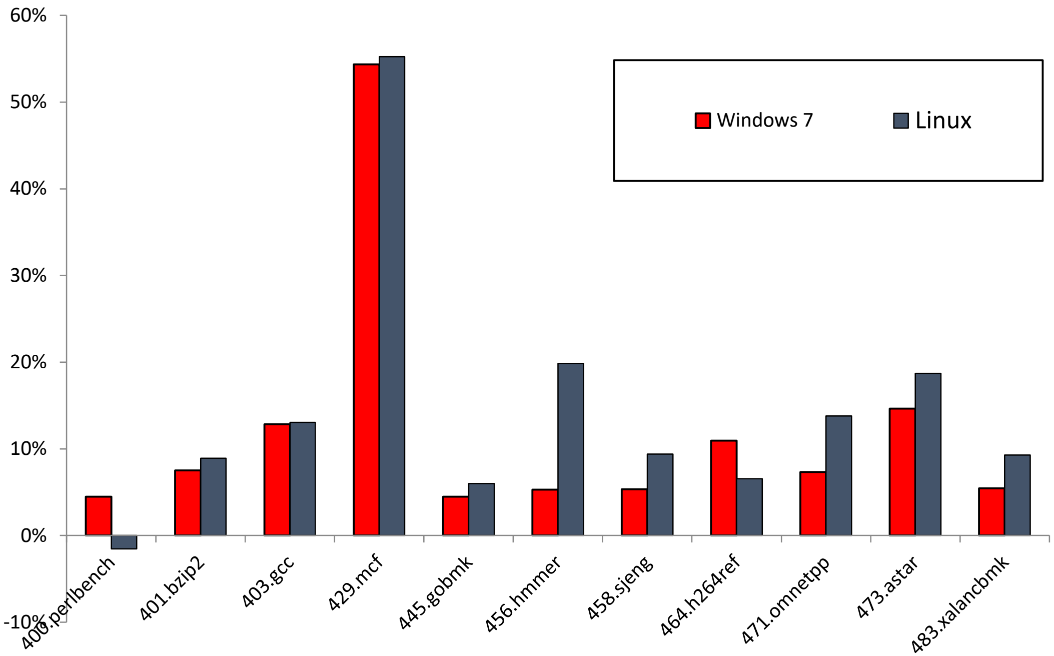
- Begin with guest ops
- Translate guest ops into IRs
- Insert helper functions to mark begin/end of block
- Insert helper functions to mark begin/end of guest op
- **Either the *whole-system* or just *modules of interest* can be instrumented**

Event-Driven Instrumentation:

A sample tainted keystroke plugin

```
1. plugin_interface_t my_interface;
2. DECAF_Handle keystroke_cb_handle = DECAF_NULL_HANDLE;
3. DECAF_Handle handle_read_taint_mem = DECAF_NULL_HANDLE;
4. int taint_key_enabled = 0;

5. void my_read_taint_mem(DECAF_Callback_Params *param) {
6.   char name[128];
7.   tmodinfo_t tm;
8.   if(VMI_locate_module_c(DECAF_getPC(cpu_single_env),
       DECAF_getPGD(cpu_single_env),name,&tm) == 0)
9.     DECAF_printf("INSN 0x%08x From Module %s Read Keystroke\n",
         DECAF_getPC(cpu_single_env),tm.name);
   }

10. void my_send_keystroke_cb(DECAF_Callback_Params *params) {
11.   *params->ks.taint_mark = taint_key_enabled;
12.   taint_key_enabled = 0;
13.   DECAF_printf("taint keystroke %d \n", params->ks.keycode);
    }
14. void do_taint_sendkey(Monitor *mon,const QDict *qdict) {
15.   if (qdict_haskey(qdict, "key")) {
16.     taint_key_enabled = 1; //enable keystroke taint
17.     do_send_key(qdict_get_str(qdict, "key")); //Send the key
      }
    }

18. mon_cmd_t my_term_cmds[] = {
      {
19.     .name = "taint_sendkey",
20.     .args_type = "key:s",
21.     .mhandler.cmd = do_taint_sendkey,
22.     .params = "taint_sendkey key",
23.     .help = "send a tainted key to system"
      },
      {NULL, NULL, },
    };
24. void my_cleanup(){......}

/* Register the plugin and the callbacks */
25. plugin_interface_t * init_plugin() {
26.   my_interface.mon_cmds = my_term_cmds;
27.   my_interface.plugin_cleanup = my_cleanup;
28.   handle_read_taint_mem = DECAF_register_callback(
        DECAF_READ_TAINTMEM_CB, my_read_taint_mem, NULL);
29.   keystroke_cb_handle = DECAF_register_callback(
        DECAF_KEYSTROKE_CB, my_send_keystroke, NULL);
30.   return &keystrokeInterface;
    }
```

Evaluation: VMI performance



SPEC CPU2006

Windows: 12%

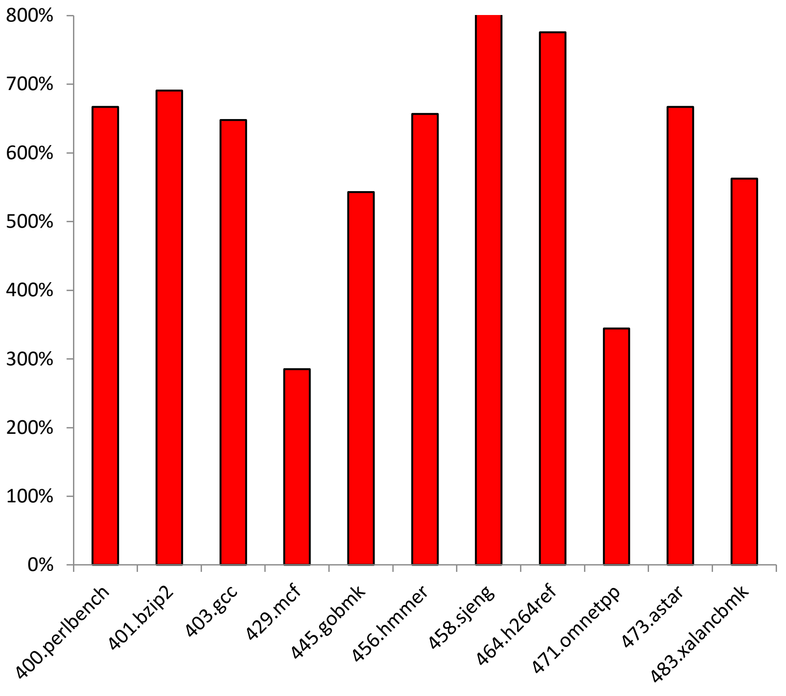
Linux: 14%

Configuration	Xubuntu	WinXP SP3	Debian Squeeze (ARM)
DECAF w/ VMI	3m 25.9s	1m 4.36s	2m 50.16s
QEMU 1.0.1	2m 45.85s	0m 52.79s	2m 36.52s
Overhead %	24.14	21.91	8.72

Common Case:

OS Boot Time

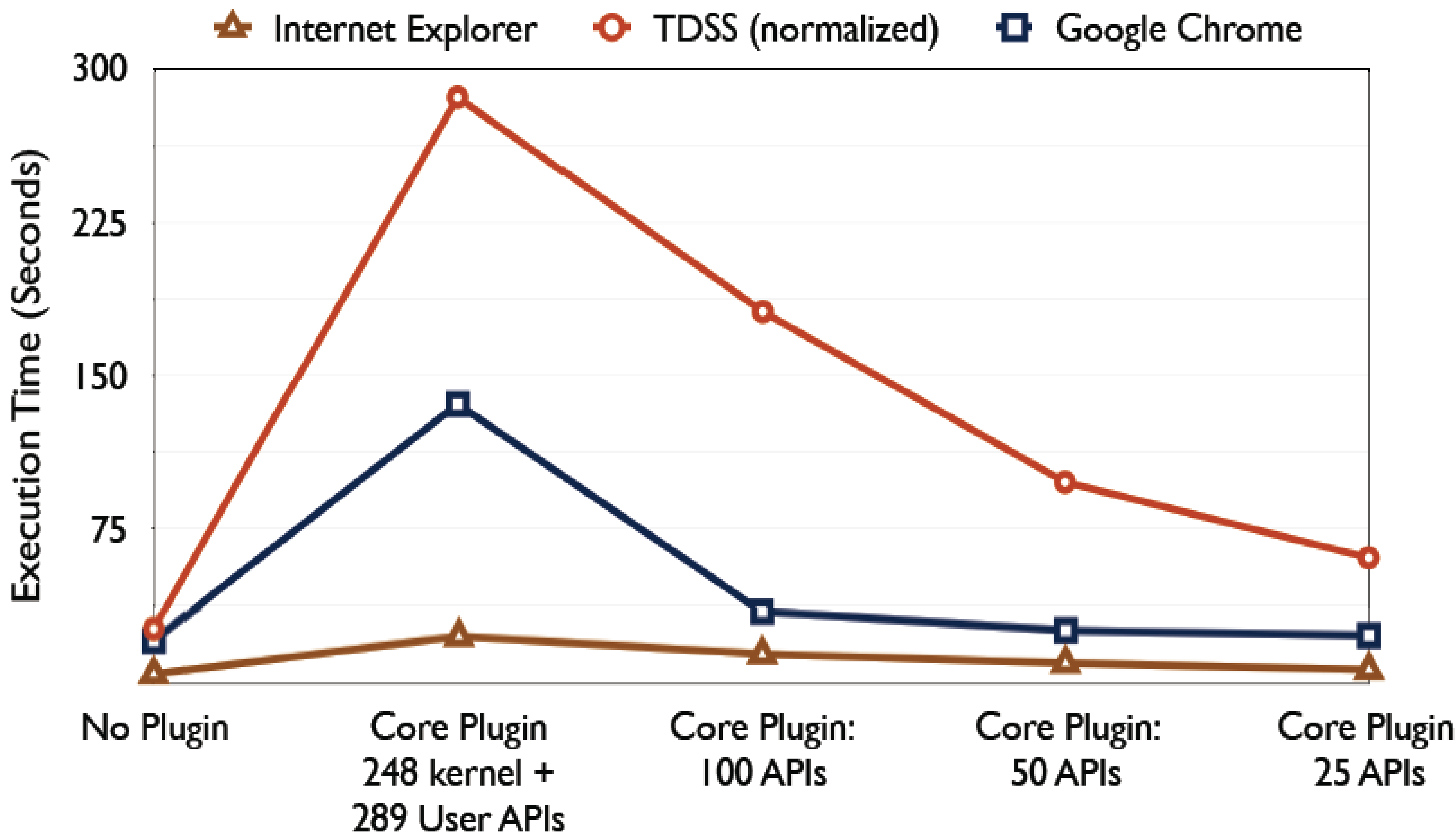
Evaluation: Tainting performance



Tainting Software	Whole System	Guest OS		Arch Support			Bitwise Granularity	Expected Overhead
		Win	Linux	X86	ARM	MIPS		
Dytan			X	X				30x
LIFT		X		X				3.6x
libdft			X	X				3.65x
Minemu		X		X				2.3x
Memcheck			X	X			X	26x
TaintBochs	X	X	X	X				10x
TEMU	X	X	X	X				20x
DECAF	X	X	X	X	X	X	X	6x

- Tainting experiences 605% overhead on SPEC CPU2006
- Heaviest performance impact on CPU-bound benchmarks

Evaluation: HookAPI plugin performance



Evaluation: Development effort

Software	OS/Arch-Independent (LOC)	OS/Arch-Specific (LOC)	Total (LOC)
DECAF	18470	1350	19820
Insn Tracer	3770	90	3860
API Tracer	840	880	1720
Key Logger	120	0	120

- Most architecture-specific code is related to accessing CPU registers
- Most OS-specific code is related to VMI

Conclusion

- DECAF provides whole-system emulation and instrumentation that ***works correctly*** and is ***fast***
- DECAF is open source and available for download:

https://github/sycurelab/decaf