KQguard: Binary-Centric Defense against Kernel Queue Injection Attacks

Jinpeng Wei, Feng Zhu
Florida International University
Miami, Florida, USA

Calton Pu
Georgia Institute of Technology
Atlanta, Georgia, USA

18th European Symposium on Research in Computer Security (ESORICS)
Egham, United Kingdom, September 2013

Motivation

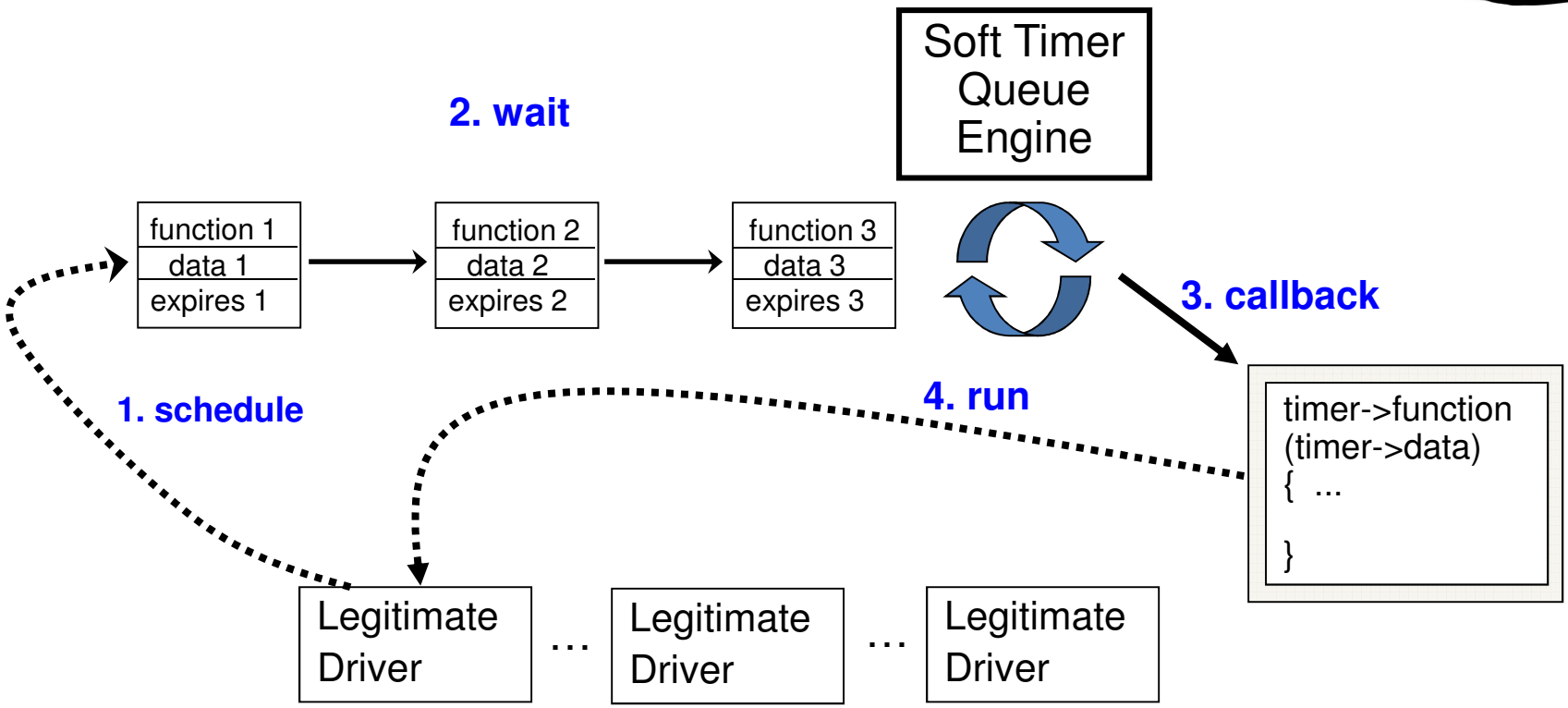
- Kernel level malware (e.g., rootkits) is among the most dangerous threats to systems security
  - e.g., hiding malicious processes and files, key logging, attacking security products, etc
- Existing defenses are effective at detecting malware that *tampers with* legitimate kernel code or data (e.g., function pointers)
- But they fall short of malware that *creates* malicious data (e.g., function pointers) in dynamic kernel data structures
  - This paper presents a case study of such malware: Kernel Queue Injection (KQI) attacks and defense

Kernel Queues (KQ)

- A mechanism of choice for handling events in modern kernels

- A kind of data structure that supports the callback of programmer-defined *event handlers* by the core kernel when the event of interest happens

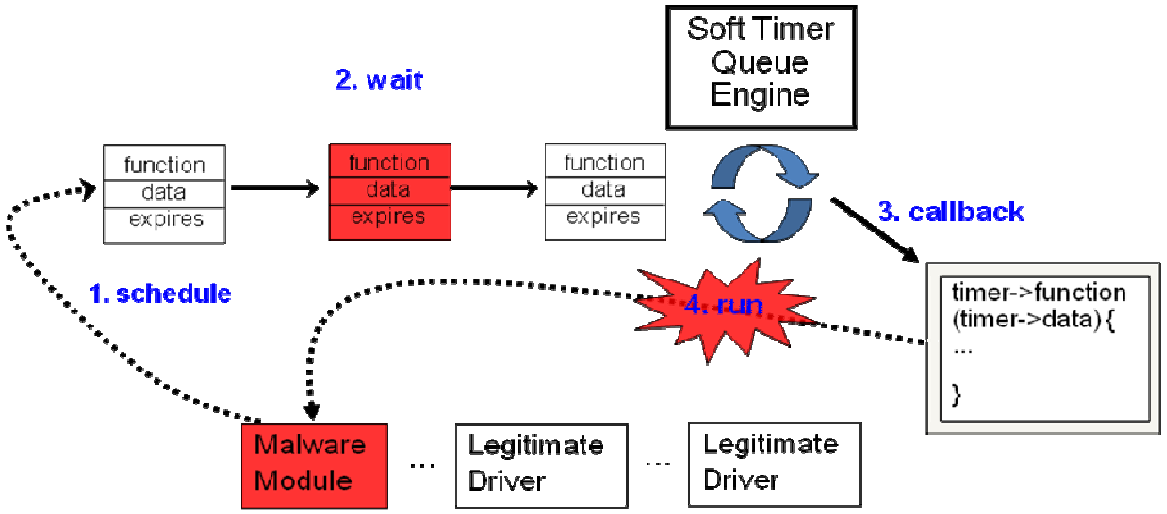
Example KQ: the Soft Timer Queue in Linux



- Common properties of KQs
  - Polymorphic: multiple handlers can exist for the same event type (in the same KQ)
  - Dynamic: event handlers can be registered or deregistered at runtime

KQ Injection Malware

- Kernel-level malware can abuse KQs to achieve malicious goals
  - by inserting malicious event handlers in an KQ
  - without modifying kernel code or static data structures (non-invasive)
  - without interfering with other installed kernel modules



Abuses of KQs by Real-wo[illegible]ware

Allows malware to track process creation or deletion events

K-Queue / Malware	Timer/DPC	Worker Thread	Load Image Notify	Create Process Notify			[illegible]yOp allback
Rustock.J			√	√	√		
Pushdo / Cutwail	√			√	√	√	√
Storm / Peacomm	√		√				
Srizbi	√				√		
TDSS			√		√	√	
Duqu	√				√		
ZeroAccess	√	√			√		√
Koutodoor	√			√			
Pandex					√		
Mebroot	√						

- Hide better against discovery
- Carry out covert operations
- Attack security products

Need for a New Defense

- Unique and more stealthy than existing kernel level attacks
- Therefore, it can evade detection of state-of-the-art anti-malware tools

Attacks	Action	Target	Stealth	Defense
Code modification	Inject	Code	Invasive	SecVisor, NICKLE
Kernel Object Hooking	Modify	Legitimate control data	Invasive	CFI, SBCFI, HookSafe
Direct Kernel Object Manipulation	Modify	Legitimate non-control data	Invasive	Gibraltar, Semantic Integrity Checker
KQ Injection	Insert	New control or non-control data	Non-invasive	KQguard

Defense Idea

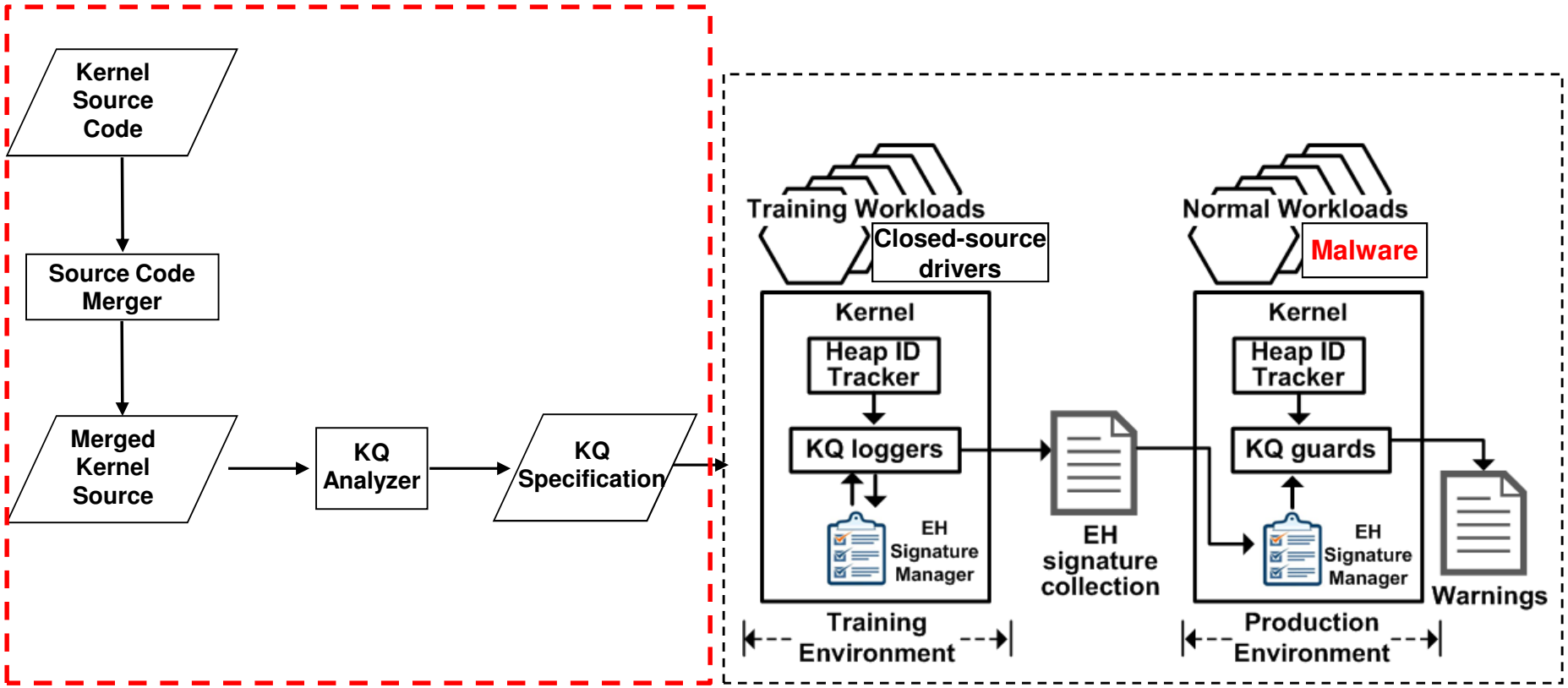
- Insert a guard into each KQ, which checks whether a KQ request is a legitimate event handler or a malicious KQ injection attack

- Legitimacy is defined by a policy specification called *EH-Signature Collection*

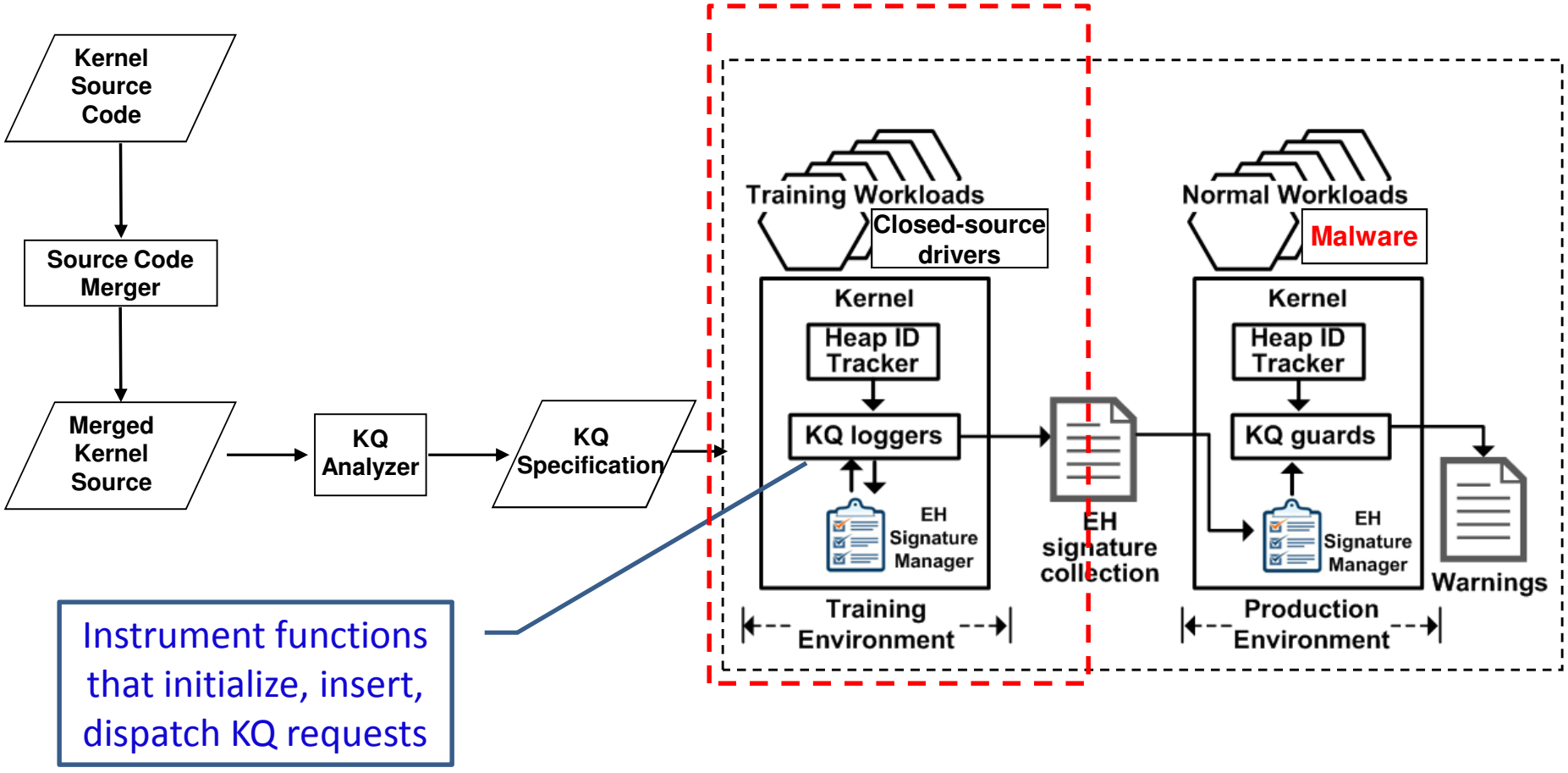
Design Goals of the Defense

Goal	Design Decision
Allow future legitimate device drivers to work properly	Isolate the knowledge of legitimate event handlers in a table (EH-Signature Collection) that is extensible
Support closed source device drivers	Employ dynamic analysis to gather EH-Signatures for closed source legitimate drivers
Guard all KQs against abuse	Automatic KQ detection tool based on source code analysis (when source code is available)

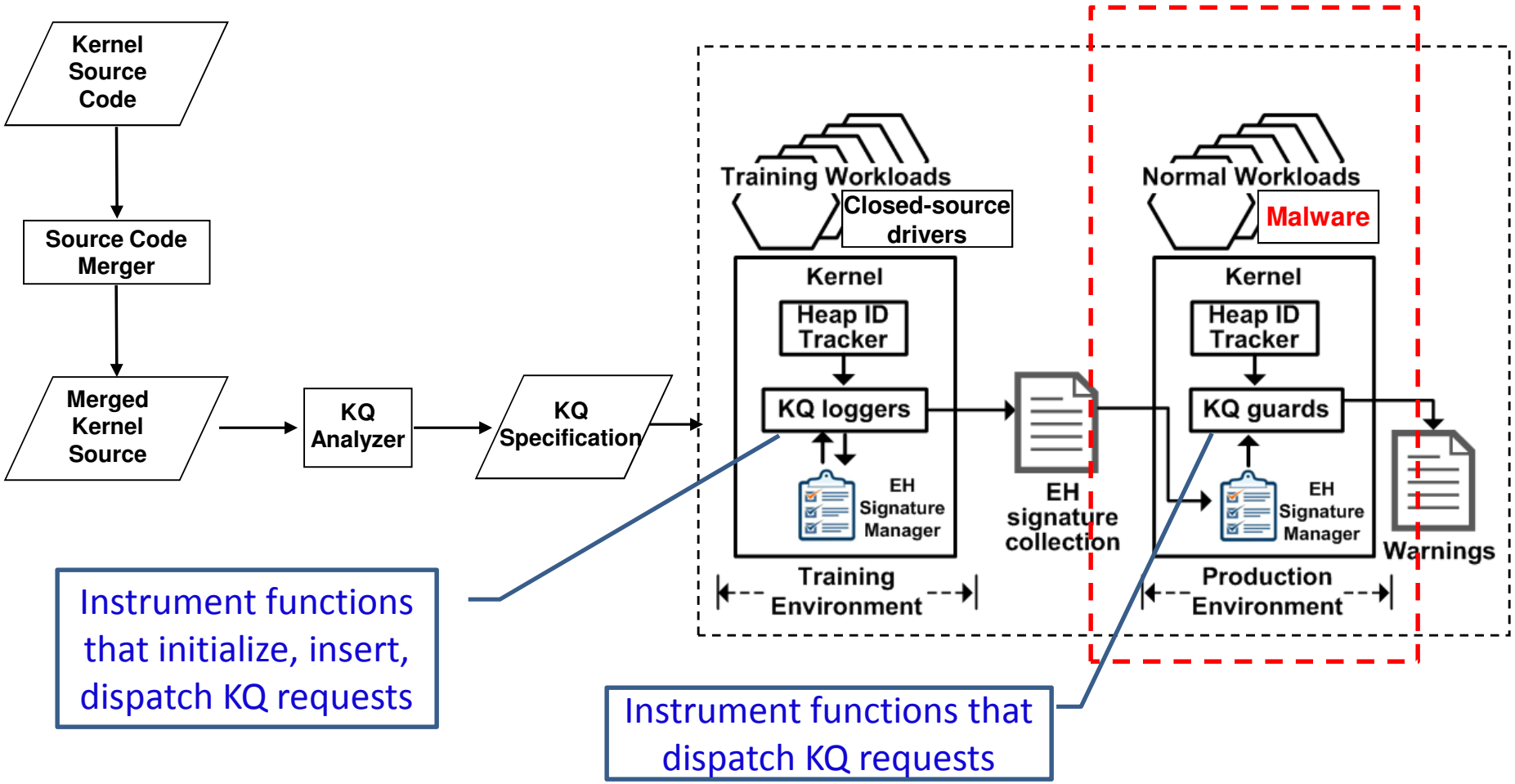
KQguard Architecture



KQguard Architecture



KQguard Architecture



EH-Signatures

- A specification that contains the right amount of information to identify a legitimate event handler
  - Our chosen specification: (callback function, relevant parameters, insertion path, allocation)

- Therefore, an EH-Signature specifies *rules* in terms of the KQ request data structure
  - Example rule: if callback function equals **`nt!VdmpQueueIntApcRoutine`**, param_1 equals **`nt!VdmpApc`**, request is inserted by **`acpi.sys+0x2c0`**, and the request data is a global variable at **`acpi.sys+0x4a00`**, the request is legitimate.

Practical Challenges of Robust EH-Signatures

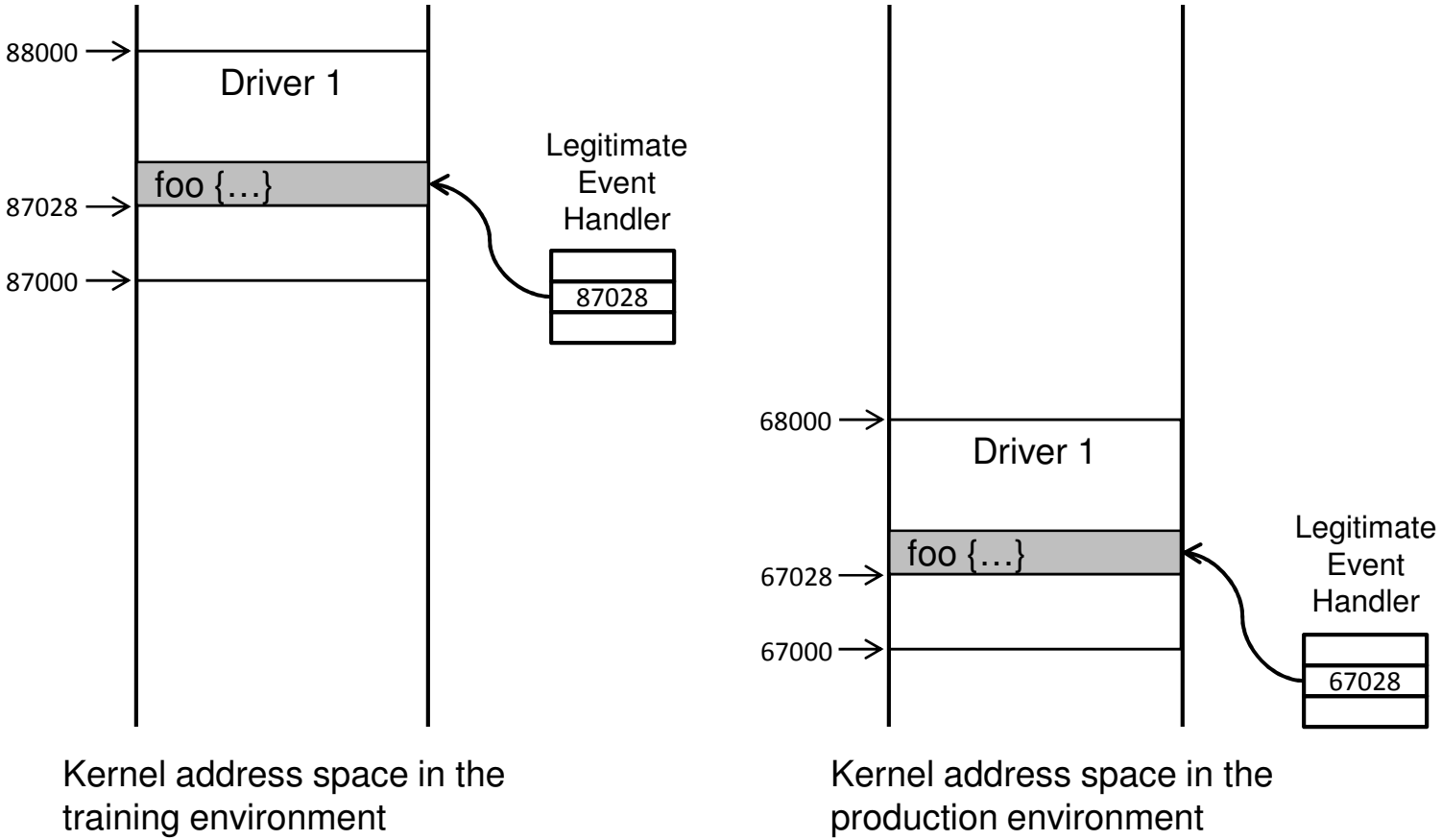
- Symbol information (e.g., `nt!VdmpQueueIntApcRoutine`) is not available for closed source device drivers. Instead, only low-level information (e.g., `0xbe07d0ac`) can be observed by the KQ guards

- The training environment is different from the production environment at the low level

- Dynamically allocated memory objects (on the heap or stack) have unpredictable low level addresses

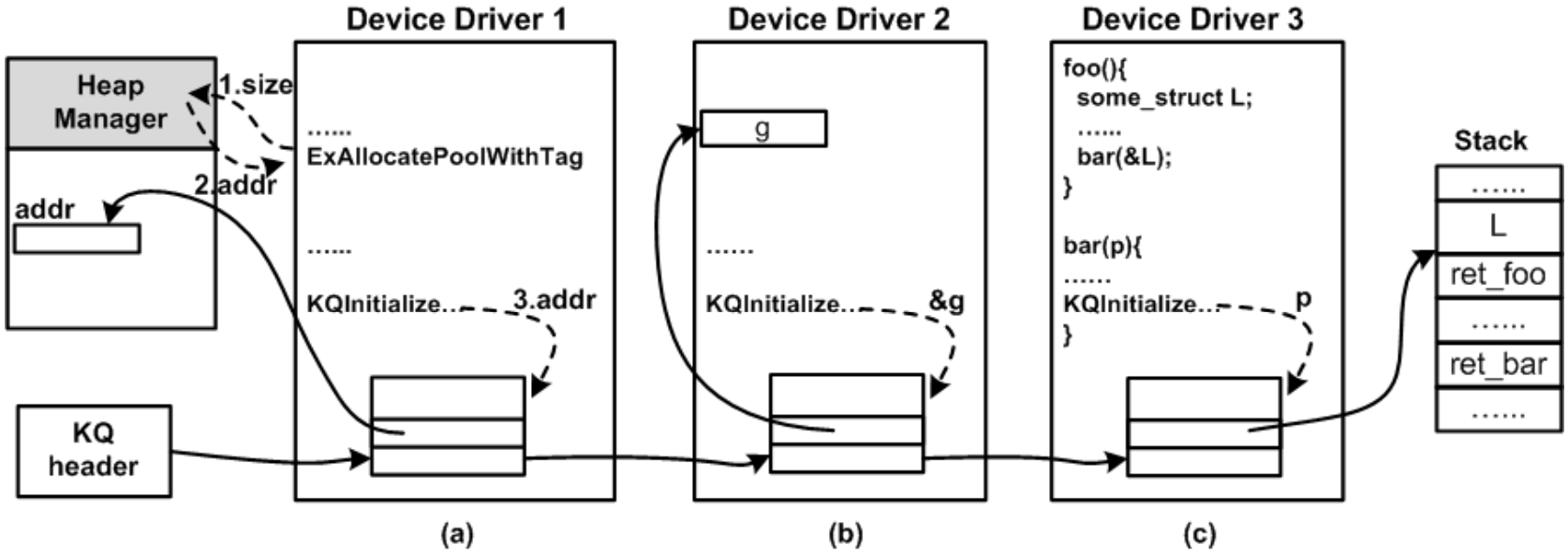
- Solution: the EH-Signatures must be specified at a higher level that can tolerate variations at the low level -> delinking

Example: Delinking the Pointer to a Global Variable



Absolute value (e.g., 87028) is not a robust representation of a pointer to **foo** that can carry over from training to production, while Driver 1_start + 28 is.

Types of KQ Request Data Fields that Need Delinking



(a) Pointer to a heap variable	(b) Pointer to a global variable	(c) Pointer to a local variable

Invariant Representation of KQ Request Data Fields

Type	Representation after delinking
Pointer to a global variable	(Driver ID, offset), e.g., (Driver 1, 28)
Pointer to a heap variable	Allocation call stack: (Driver ID_1, $offset_1$) … (Driver ID_{n-1}, $offset_{n-1}$) (Driver ID_n, $offset_n$)
Pointer to a local variable	((Driver ID_1, $offset_1$) … (Driver ID_{n-1}, $offset_{n-1}$) (Driver ID_n, $offset_n$) , local_variable_offset)
Not a pointer	Actual value

Automated Detection of KQs by Analyzing Source Code

```
/* linux-2.4.32/kernel/pm.c */
int pm_send_all (pm_request_t rqst, void *data)
{     ……
 entry = pm_devs.next;
 while (entry != &pm_devs) {
   struct pm_dev *dev=list_entry(entry, struct pm_dev, entry);
   if (dev->callback) {
     int status = pm_send(dev, rqst, data);
     ……}
   entry = entry->next;   }
 ……}
int pm_send(struct pm_dev *dev, pm_request_t rqst, void *data)
{……
   status = (*dev->callback)(dev, rqst, data);
……}
```

Detect a loop that iterates through a candidate data structure

Check whether a queue element is derived and acted upon inside the loop

Automated Detection of KQs by Analyzing Source Code

```
/* linux-2.4.32/kernel/pm.c */
int pm_send_all (pm_request_t rqst, void *data)
{     ……
 entry = pm_devs.next;
 while (entry != &pm_devs) {
   struct pm_dev *dev=list_entry(entry, struct pm_dev, entry);
   if (dev->callback) {
     int status = pm_send(dev, rqst, data);
     ……}
   entry = entry->next;   }
 ……}
int pm_send(struct pm_dev *dev, pm_request_t rqst, void *data)
{……
   status = (*dev->callback)(dev, rqst, data);
……}
```

Detect a loop that iterates through a candidate data structure

Check whether a queue element is derived and acted upon inside the loop

Performs a flow-sensitive taint propagation through the rest of the loop body

Automated Detection of KQs by Analyzing Source Code

```
/* linux-2.4.32/kernel/pm.c */
int pm_send_all (pm_request_t rqst, void *data)
{     ……
 entry = pm_devs.next;
 while (entry != &pm_devs) {
   struct pm_dev *dev=list_entry(entry, struct pm_dev, entry);
   if (dev->callback) {
     int status = pm_send(dev, rqst, data);
     ……}
   entry = entry->next;   }
 ……}
int pm_send(struct pm_dev *dev, pm_request_t rqst, void *data)
{……
   status = (*dev->callback)(dev, rqst, data);
……}
```

Detect a loop that iterates through a candidate data structure

Check whether a queue element is derived and acted upon inside the loop

Performs a flow-sensitive taint propagation through the rest of the loop body

If any tainted function pointer is invoked during the propagation, report a candidate KQ

Automated Detection of KQs by Analyzing Source Code

```
/* linux-2.4.32/kernel/pm.c */
int pm_send_all (pm_request_t rqst, void *data)
{     ……
 entry = pm_devs.next;
 while (entry != &pm_devs) {
   struct pm_dev *dev=list_entry(entry, struct pm_dev, entry);
   if (dev->callback) {
     int status = pm_send(dev, rqst, data);
     ……}
   entry = entry->next;   }
 ……}
int pm_send(struct pm_dev *dev, pm_request_t rqst, void *data)
{……
   status = (*dev->callback)(dev, rqst, data);
……}
```

Detect a loop that iterates through a candidate data structure

Check whether a queue element is derived and acted upon inside the loop

Performs a flow-sensitive taint propagation through the rest of the loop body

If any tainted function pointer is invoked during the propagation, report a candidate KQ

Implementation

- KQ Analyzer: ~2,000 lines of Objective Caml code, based on C Intermediate Language (CIL)

- Windows Research Kernel instrumentation
  - KQ Logger: ~600 lines of C code
  - Callback Signature collection: ~2,200 lines of C code
  - Heap Object Tracker: ~800 lines of C code
  - KQguards: ~300 lines of C code

- Linux kernel implementation (similar to Windows)

Experimental Evaluation of KQguard on Windows

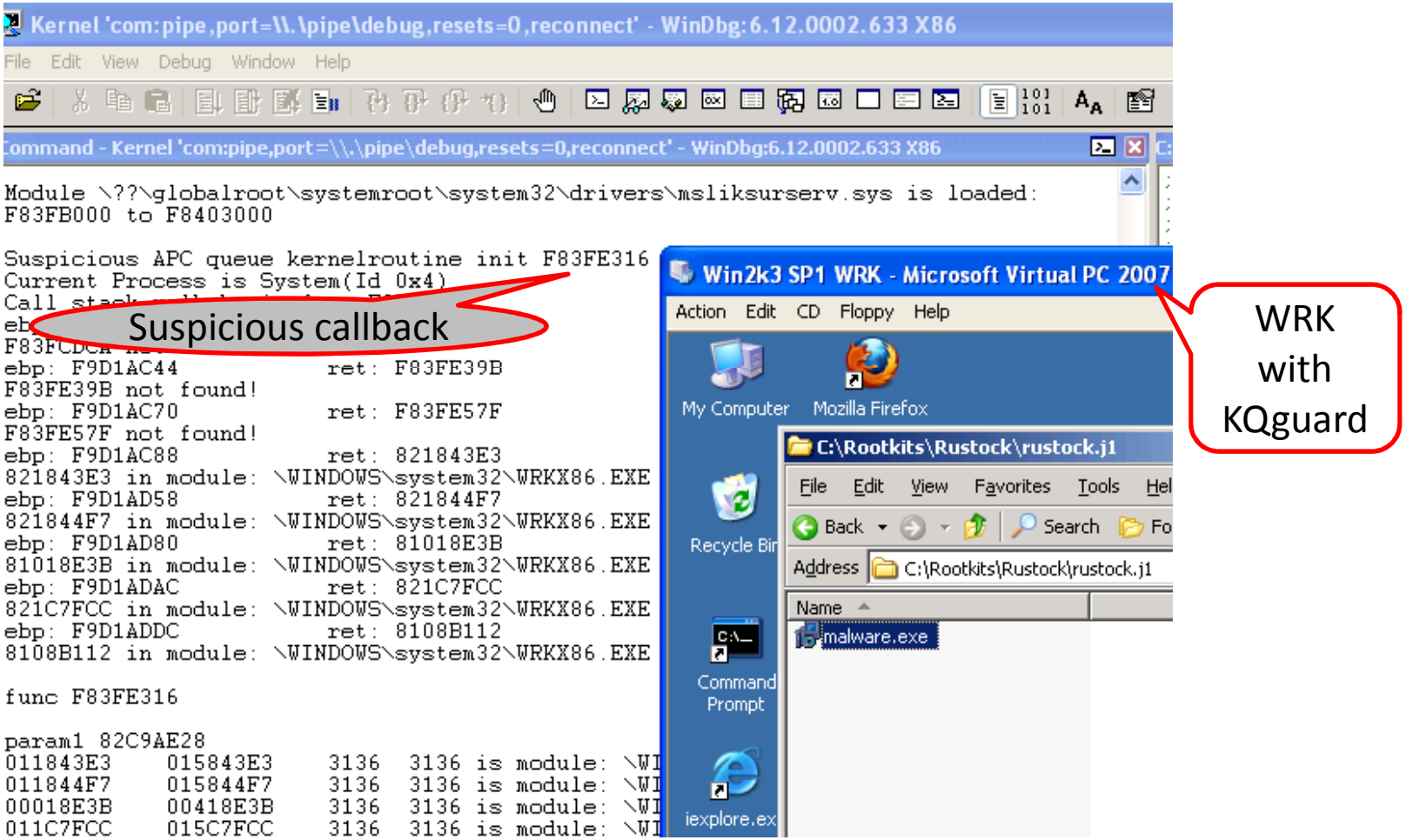
- False negatives
- False positives
- Overhead

False Negatives of KQguard on Windows

- Test cases: 125 KQ injection malware samples from the top 20 malware families and the top 10 botnet families, plus 9 synthetic malware
- Result: detected known KQ injection in 123 malware samples, and all synthetic malware

KQ name	Asynchronous Procedure Call (APC)	Timer/DPC	Load Image Notify	Create Process Notify	FsRegistration Change	RegistryOp Callback	System Worker Thread
# of malware samples	98	34	32	20	4	4	2

Detection of KQ Injection Attacks by Rustock.J on Windows Research Kernel



False Negatives of KQguard on Windows

- Test cases: 125 malware samples from the top 20 malware families and the top 10 botnet families, plus 9 synthetic malware
- Result: detected known KQ injection in 123 malware samples, and all synthetic malware
- Undetected ones: Duqu on load image notification queue, Storm on the APC queue

KQ name	Asynchronous Procedure Call (APC)	Timer/DPC	Load Image Notify	Create Process Notify	FsRegistration Change	RegistryOp Callback	System Worker Thread
# of malware samples	98	34	32	20	4	4	2

Experimental Evaluation of KQguard on Windows

- False negatives: able to detect known KQ abuses in 123 out of 125 real world malware, plus unreported ones

- False positives: zero after proper training
  - Tested with Acrobat Reader, Windows Driver Kit, Firefox, Windows Media Player, Easy Media Player, and several games.

- Overhead
  - Micro benchmarks: ~3.4%
    - Fraction of time spent in KQ validation
  - Macro benchmarks: 2.8% - 5.6% slowdown

Overhead of KQguard on WRK (Macro benchmarks)

Workload	Original (sec)	KQ Guarding (sec)	Slowdown
Super PI	2,108±41	2,213±37	5.0%
Copy directory (1.5 GB)	231±9.0	244±15.9	5.6%
Compress directory (1.5 GB)	1,113±24	1,145±16	2.9%
Decompress directory (1.5 GB)	181±4.1	186±5.1	2.8%
Download file (160 MB)	145±11	151±11	4.1%

Conclusion

- KQ Injection is a significant attack
- KQguard uses static analysis of kernel source code to detect KQ instances
- KQguard uses dynamic analysis of kernel and device drivers to learn the legitimate KQ event handlers without source code
- Evaluation on the WRK shows that KQ guarding is effective (very low false negative rate and false positive rate) and efficient (up to ~5% overhead)

Thank you!

Questions?

Jinpeng Wei
Assistant Professor
Florida International University
Miami, Florida, USA
Email: weijp@cs.fiu.edu