Securing Operating Systems Against Advanced Malware

Ashvin Goel
University of Toronto

Advanced Host Level Surveillance (AHLS)
DRDC AHLS Workshop
Feb 06, 2013

Protecting Operating Systems

- Operating system kernel is fully privileged
- Kernel compromises are devastating
  - Remote attacker takes control of (i.e., owns) machine
  - Local user gets root privilege

Attacking the Kernel

- Gain limited access to the system
  - Exploit a known software vulnerability
  - Crack weak passwords
  - Steal passwords
  - Buffer overflow in user-level software
  - Using social engineering
  - E.g., deceive user into installing malicious program

- Escalate privileges to gain elevated access
  - Exploit vulnerability in privileged programs
  - E.g., get root shell by targeting vulnerable setuid program

Attacking the Kernel

- Gain limited access to the system
  - Exploit a known software vulnerability
  - Crack weak passwords
  - Steal passwords
  - Buffer overflow in user-level software
  - Using social engineering
  - E.g., deceive user into installing malicious program
- Escalate privileges to gain elevated access
  - Exploit vulnerability in privileged programs
  - E.g., get root shell by targeting vulnerable setuid program
  - Exploit kernel vulnerability

Linux Kernel Vulnerabilities

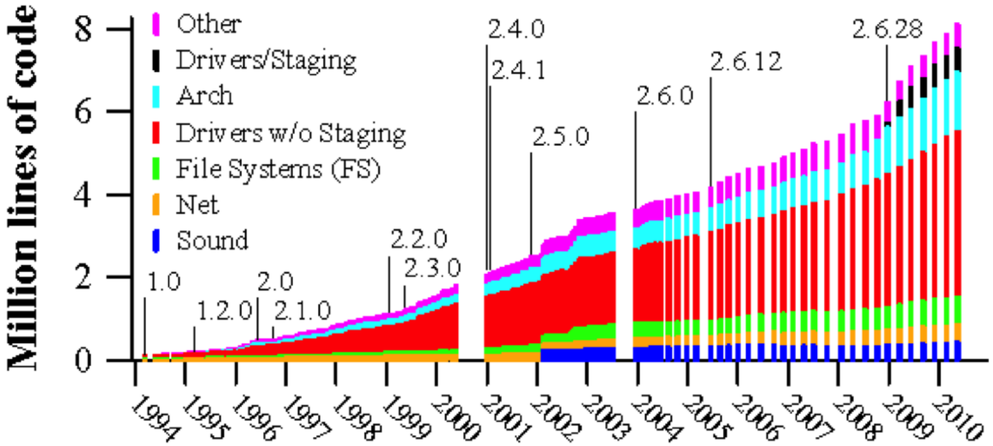
- Vulnerabilities are routinely discovered in Linux
- CVE security vulnerability database for last 3

Year	# of vulnerabilities	DoS	Code execution	Overflow	Memory corruption	Bypass checks	Gain info.	Gain privileges
2011	83	62	1	21	10	1	21	9
2012	115	83	4	24	10	6	19	11
2013	190	101	6	41	13	11	58	26

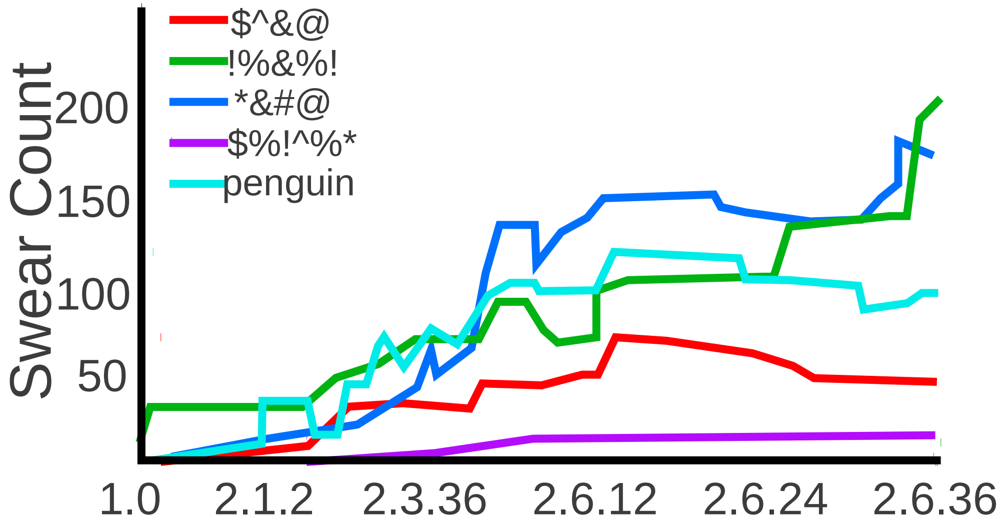
- Why are vulnerabilities increasing?

Linux Kernel Complexity

- Growth in code size
  - Palix, ASPLOS 2011
  - Many new drivers!



- More swearing
  - Vidar Holen, 2012



- Bugs and vulnerabilities are inevitable

Kernel Threat Landscape

- Fastest rising threat in last 2 years is mobile malware
  - Typical mobile malware uses fake programs, adware
    - Most common platform is Android (runs Linux variant)
      - Mcafee: 35000 collected in 2013, expected to double in 2013
      - Users deceived into installing these programs from third-party sites

- Social engineering + kernel vulnerability: deadly
  - Initially, programs would send premium SMS messages
    - Andr/KongFu-L is a fake Angry Birds program
      - Exploits kernel vulnerability in Gingerbread to gain root access, communicate with remote sites, install additional malware
    - Backdoor.AndroidOS.Obad is very sophisticated
      - Uses encryption, obfuscation, exploits multiple kernel vulnerabilities to obtain device administrator privileges, impossible to remove

Residing in the Kernel

- Gain limited access to the system
  - Exploit a known software vulnerability
  - Crack weak passwords
  - Steal passwords
  - Buffer overflow in user-level software
  - Using social engineering
  - E.g., deceive user into installing malicious program
- Escalate privileges to gain elevated access
  - Exploit vulnerability in privileged programs
  - E.g., get root shell by targeting vulnerable setuid program
  - Exploit kernel vulnerability
- Take steps to continue accessing the system
  - Install kernel rootkit

Kernel Rootkits

- A kernel exploit that is designed to hide its presence
  - May open backdoors, steal information or actively disable kernel-based defenses

- Often installed using social engineering
  - Example: Sony rootkit
  - In 2005, Sony provided a music player on Windows
  - Player installed a kernel rootkit that limited the user’s ability to access a CD
  - Unfortunately, other kernel malware then took advantage of a vulnerability in this rootkit
  - When Sony attempted to uninstall its rootkit, it exposed users to an even more serious vulnerability

How do Kernel Rootkits Work?

- Modern kernels allow installing third-party, untrusted **modules** to extend kernel functionality
  - Loaded on demand, e.g., when USB camera is plugged in
  - Executed with the **same** privileges as the core kernel

- A kernel rootkit can either be
  - A malicious module, or
  - A benign, vulnerable module that has been subverted

- After rootkit is installed, it can fully control the machine, because it runs with the highest privileges

Understanding Rootkits

- A “perfect rootkit” is similar to a “perfect crime”: one that nobody realizes has taken place

- Rootkits have complete access to kernel code & data
  - Install or modify other module or core kernel code
  - Replace system calls, disable page protection
  - Load code into user processes
  - Conceal running processes, installed modules, files
  - Tamper with event logging facility
  - Bypass tools that monitor system calls or file modifications because they can execute entirely in kernel context

Access to Code and Data

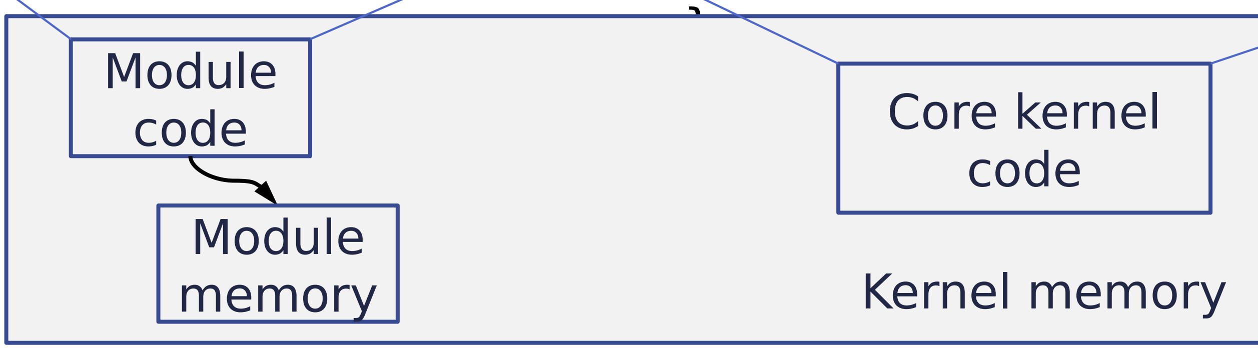
- Kernel modules call core kernel functions, core kernel calls module functions
- Kernel modules share data with kernel, e.g., stack

```
// Correct module code

spinlock_t mylock;
spin_lock_init(&mylock)
;
```

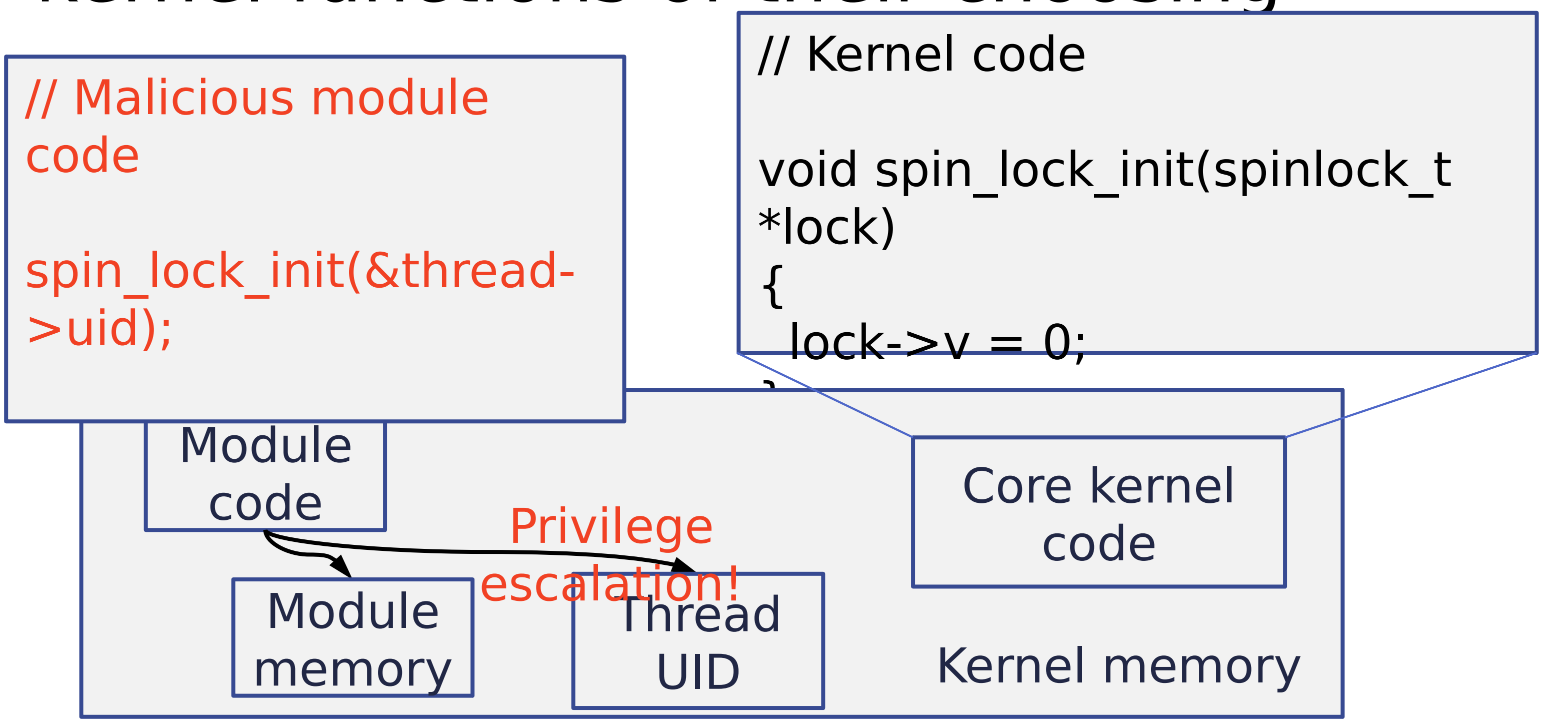
```
// Kernel code

void spin_lock_init(spinlock_t
*lock)
{
  lock->v = 0;
}
```



Attack

- Attacker tricks kernel to overwrite UID to root
- Similarly, attacker can trick kernel to call kernel functions of their choosing



Goals of Project

- Goal is to protect operating system kernels
  - Analyze and detect kernel bugs and vulnerabilities
  - Protect kernel against module code
  - Vulnerable modules
    - E.g., module calls unexported function, overwrites kernel stack
    - Need to detect disallowed behavior
  - Malicious modules (rootkits)
    - E.g., CD module calls exported network send function
    - Need to detect anomalous behavior
  - Requires understanding module behavior
  - What modules do, what they should be allowed to do

Challenges

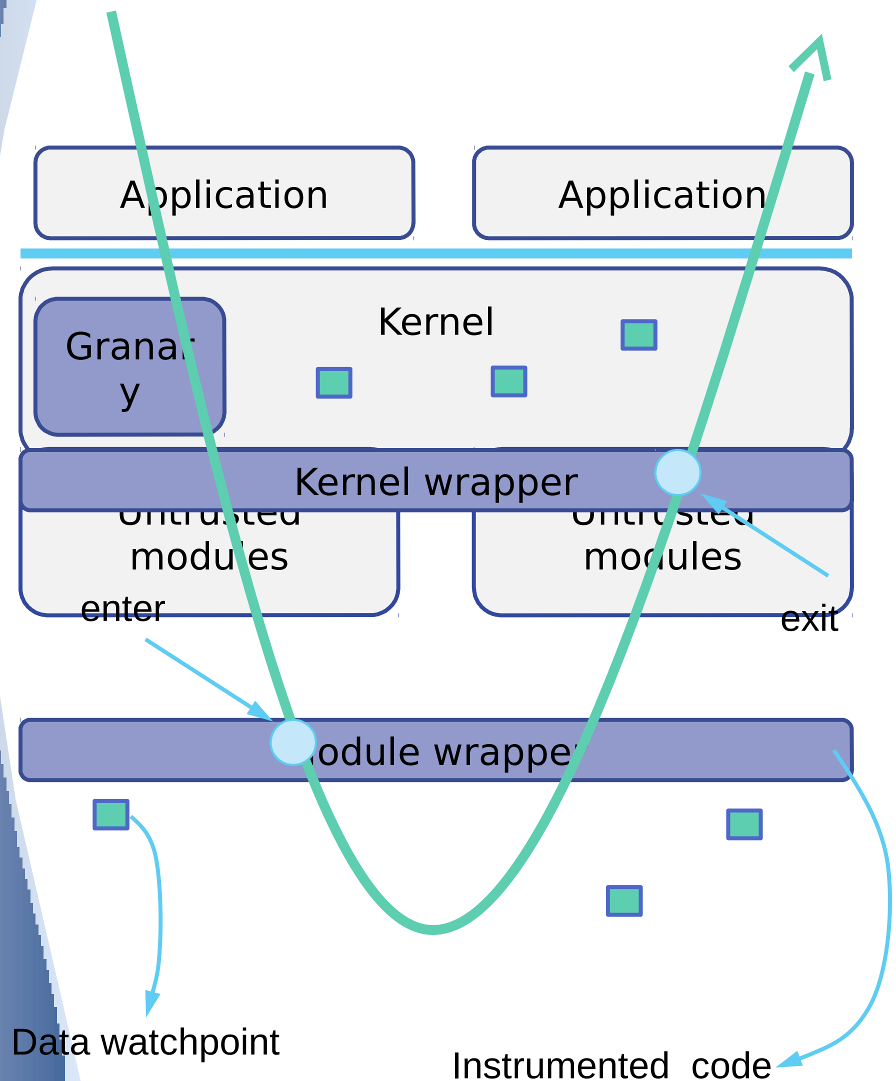
- Kernel APIs are not written defensively
  - Assume modules obey implicit rules
  - Do not check arguments, permissions, etc.
- Modules cannot be trusted to follow rules
  - Module can trick kernel into performing unexpected actions
- Existing solutions
  - Anti-virus software protects against user-level malware
  - Can be disabled by kernel malware

Approach

- Instrument all module related code at runtime using dynamic binary translation (DBT)
  - Rewrite binary module code on-the-fly during execution
    - Operates at instruction granularity
    - Provides complete control over program execution
  - Requires no module sources to be available
  - Building a system called **Granary**

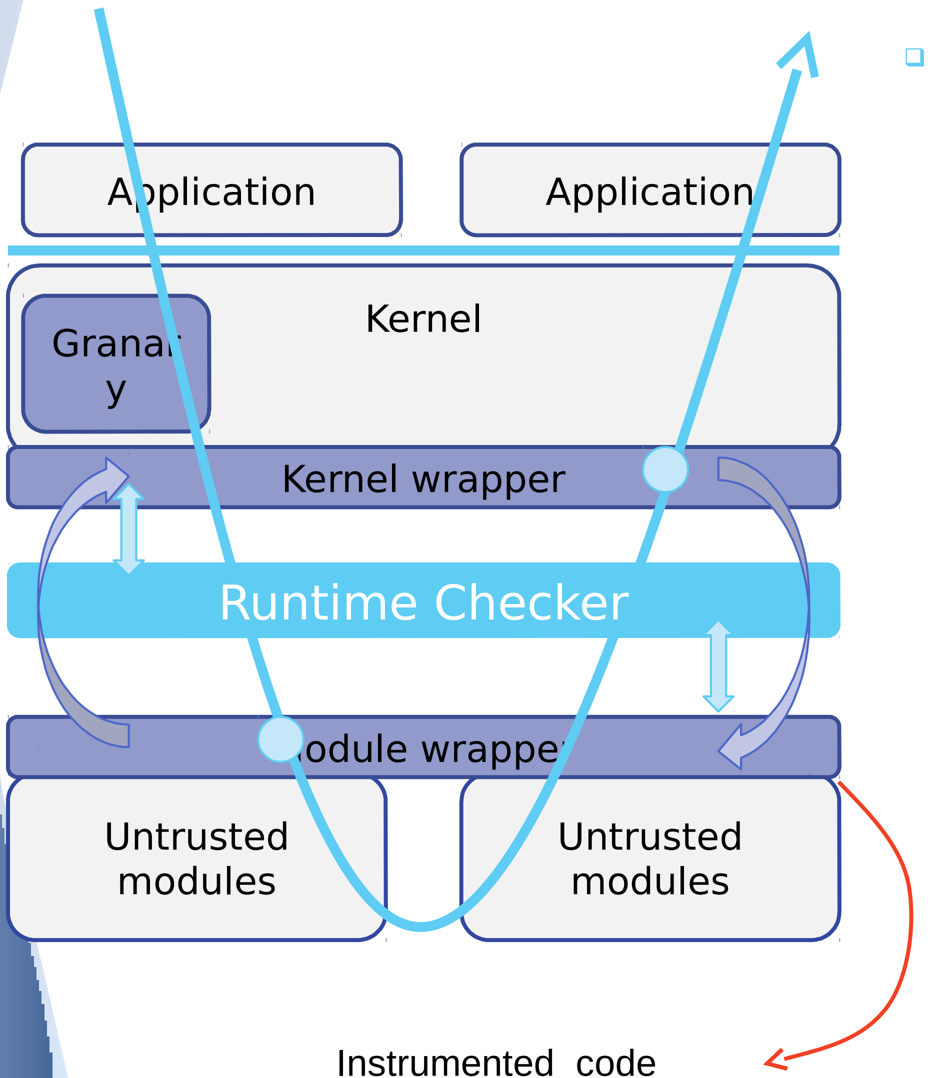
- Two key ideas
  - Add module and kernel interface **wrappers**
    - Allows mediating all control transfers between kernel and modules
  - Verify memory accesses by modules using **watchpoints**
    - Allows mediating all data accesses by modules

Overview of Granary



- Add kernel and module wrappers and watchpoints
  - Granary starts at module wrapper
  - Granary stops at kernel wrapper
  - Minimal overhead when kernel is running
- Wrappers allow adding arbitrary integrity checking instrumentation code
- Watchpoints allow instrumenting data accesses

Using Wrappers to Ensure Integrity



- Runtime checker enforces CFI
  - CFI: Execution only follows paths determined by the static control-flow graph (CFG)
  - Checker integrated in the kernel and module wrappers
  - Verifies the target address on any cross control transfer between kernel and the modules
  - Maintains call-return consistency to protect from the return-oriented attacks
  - Verifies function call arguments to maintain argument integrity

Using Watchpoints to Instrument Data Accesses

- Designing address watchpoints
  - Instrument data accesses by mangling memory addresses
  - Triggers the invocation of a type-specific function when watched memory address is dereferenced to access object
  - Support millions of object-granularity watchpoints
  - Addresses limitations of h/w watchpoints

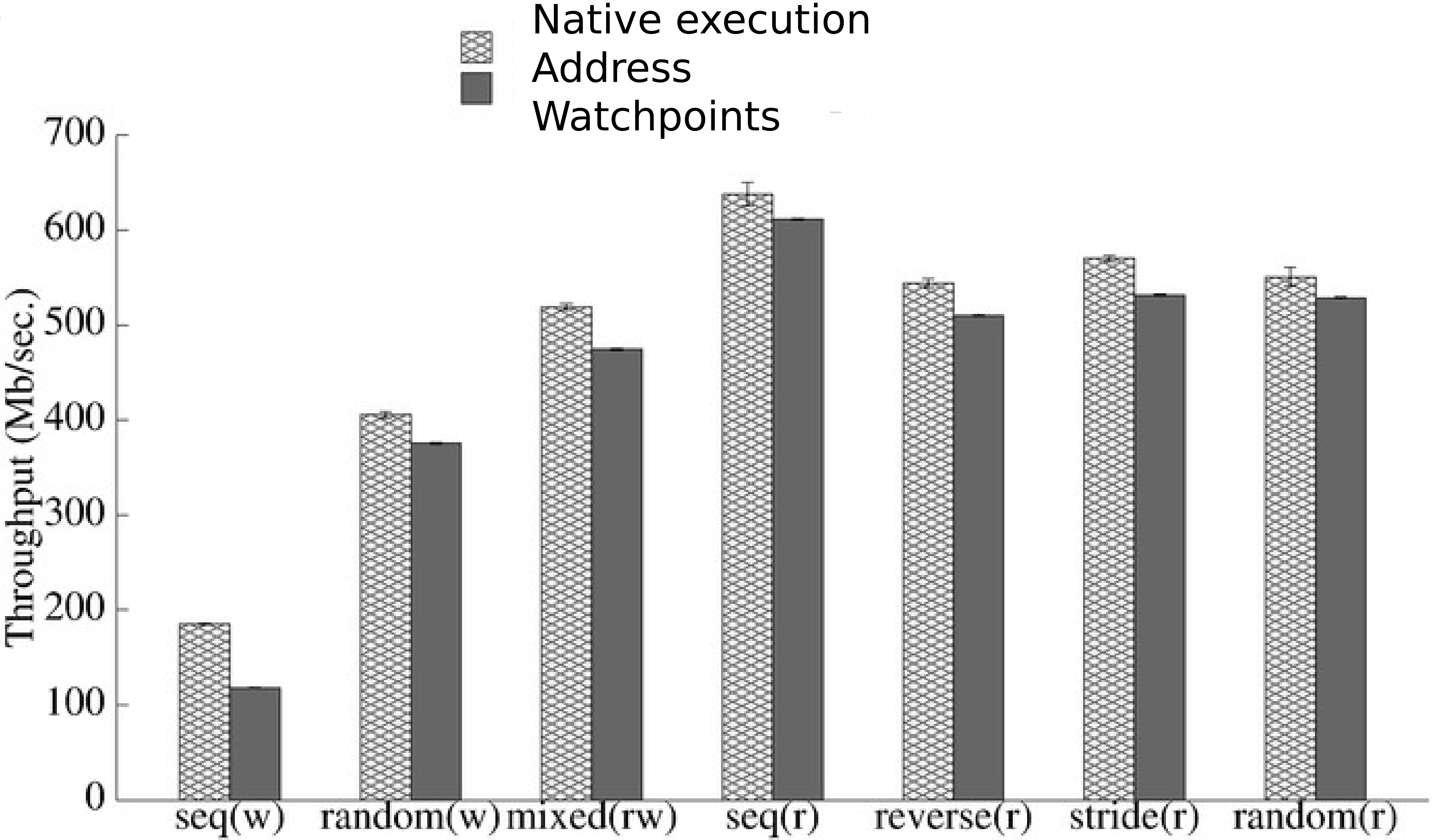
- Example
  - When a module (e.g., a file sytem module) accesses any inode, an inode-specific watchpoint function is invoked

Watchpoint Applications

- Detecting kernel buffer overflows
- Detecting read-before-write bugs, double free bugs
- Detecting memory leaks using garbage collector
- Debugging usage bugs, e.g., RCU bugs
- Enforcing fine-grained memory access policies
- Ensuring kernel data structure integrity

Evaluation

- Goal: Measure CPU overhead of selective instrumentation
- Preliminary evaluation with a microbenchmark
  - Data-centric instrumentation on objects primarily accessed by the Ext3 file system module
- Ran iozone file system benchmark
  - We mounted Ext3 file system on a 2 GB ramdisk
  - Buffer cache disabled



- Watched roughly 30% of all object accesses to Ext3 allocated objects
- 8% average overhead

Current Status

- Building a system called **Granary** that allows
  - Analyzing bugs/vulnerabilities in the Linux kernel
  - Enables securing kernel against module code

- Granary instruments binary Linux kernel modules
  - Uses wrappers for interposing on all code crossing the kernel/module boundary
  - Granary uses watchpoints for interposing on data accesses
  - Enables highly selective code, data instrumentation
  - Preliminary evaluation shows low overhead

Future Work

- Improvements in instrumentation performance
  - Improve watchpoint performance
  - Optimize instrumentation tools
- Build rich set of tools
  - Detect kernel buffer overflows, memory corruption, privilege escalation
  - Enforce fine-grained memory access policies to ensure kernel data structure integrity
- Perform experimentation
  - Whether it detects known rootkits
  - Whether it generates false alarms for benign modules

Deliverables

- We will make the following available:
  - All code for Granary
  - All code for analyzing and testing module behavior
  - All Granary tools
- Maturity level
  - All this code will run on standard x86 machines, running a standard Linux kernel, Granary requires installing a module
- Target deployment
  - System administrators deploy Granary tools
  - Developers create vulnerability analysis, detection tools

Thanks!

- Questions