

Program semantics-Aware Intrusion Detection

Prof. Tzi-cker Chiueh
Computer Science Department
Stony Brook University
chiueh@cs.sunysb.edu

Introduction

- Computer attacks that exploit software flaws
 - ◆ Buffer overflow: heap/stack/format string
 - Most common; building blocks for worm attacks
 - ◆ Syntax loopholes: SQL injection, Directory traversal
 - ◆ Race conditions: mostly local attacks
- Other attacks
 - ◆ Social engineering
 - ◆ Password cracking
 - ◆ Denial of service

Control- Hijacking Attacks

- Network applications whose control gets hijacked because of software bugs: Most worms, including MSBlast, exploit such vulnerabilities
- Three-step recipe:
 - ◆ Insert malicious code into the attacked application
Sneaking weapons into a plane
 - ◆ Trick the attacked application to transfer control to the inserted code
Taking over the victim plane
 - ◆ Execute damaging system calls as the owner of the attacked application process
Hit a target with the plane

Stack Overflow Attack

```
main() {  
    input();  
}  
  
input() {  
    int i = 0;;  
    int userID[5];  
  
    while ((scanf("%d", &(userID[i]))) != EOF)  
        i ++;  
}
```

STACK LAYOUT

128 Return address of input() **100**

FP → 124 Previous FP

120 Local variable i

116 userID[4]

112 userID[3]

108 userID[2]

INT 80

104 userID[1]

SP → 100 userID[0]

Palladium (since 1999...)

- **Array bound checking:** Preventing code insertion through buffer overflow
- **Integrity check for control-sensitive data structure:** Preventing unauthorized control transfer through over-writing return address, function pointer, and GOT
- **System call policy check:** Preventing attackers from issuing damaging system calls
- **Repairable file service:** Quickly putting a compromised system back to normal order after detecting an intrusion

Array Bound Checking

- Prevent unauthorized modification of sensitive data structures (e.g., return address or bank account) through buffer overflowing → The **cleanest** solution
- Check each pointer reference with respect to the limit of its associated object
 - ◆ Figure out which is the associated object (**shadow variable** approach)
 - ◆ Perform the limit check (major overhead)
- Current software-based array bound checking methods: **3-30** times slowdown

Segmentation Hardware

X86 architecture's virtual memory hardware supports both segmentation and paging

Virtual Address = Segment Selector + Offset

segmentation ↓ **base + offset ≤ limit**

Linear Address

paging ↓

Physical Address

Checking Array bound using Segmentation Hardware (CASH)

- Exploiting segment limit check hardware to perform array bound checking for free
- Each array or buffer is treated as a separate segment and referenced accordingly

```
for (i = M; i < N; I++) {  
    B[i] = 5;  
}
```

```
offset = &(B[M]) - B_Segment_Base;  
GS = B_Segment_Selector;  
for (i = M; i < N; i++) {  
    GS:offset = 5;  
    offset += 4;
```

Performance Overhead

CASH

BCC

SVDPACK	1.82%	120.00%
Volume Rendering	3.26%	126.38%
2D FFT	3.95%	72.19%
Gaussian Elimination	1.61%	92.40%
Matrix Multiply	1.47%	143.77%
Edge Detection	2.23%	83.77%

Return Address Defense (RAD)

- To prevent the return address from being modified, keep a **redundant** copy of the return address when calling a procedure, and make sure that it has not been modified at procedure return
- Include the bookkeeping and checking code in the function prologue and epilogue, respectively

Binary RAD Prototype

- Aims to protect Windows Portable Executable (PE) binaries
- Implementing a fully operational disassembler for X86 architecture
- Inserting RAD code at function prolog and epilog without disturbing existing code
- Transparent initialization of RAR

Performance Overhead

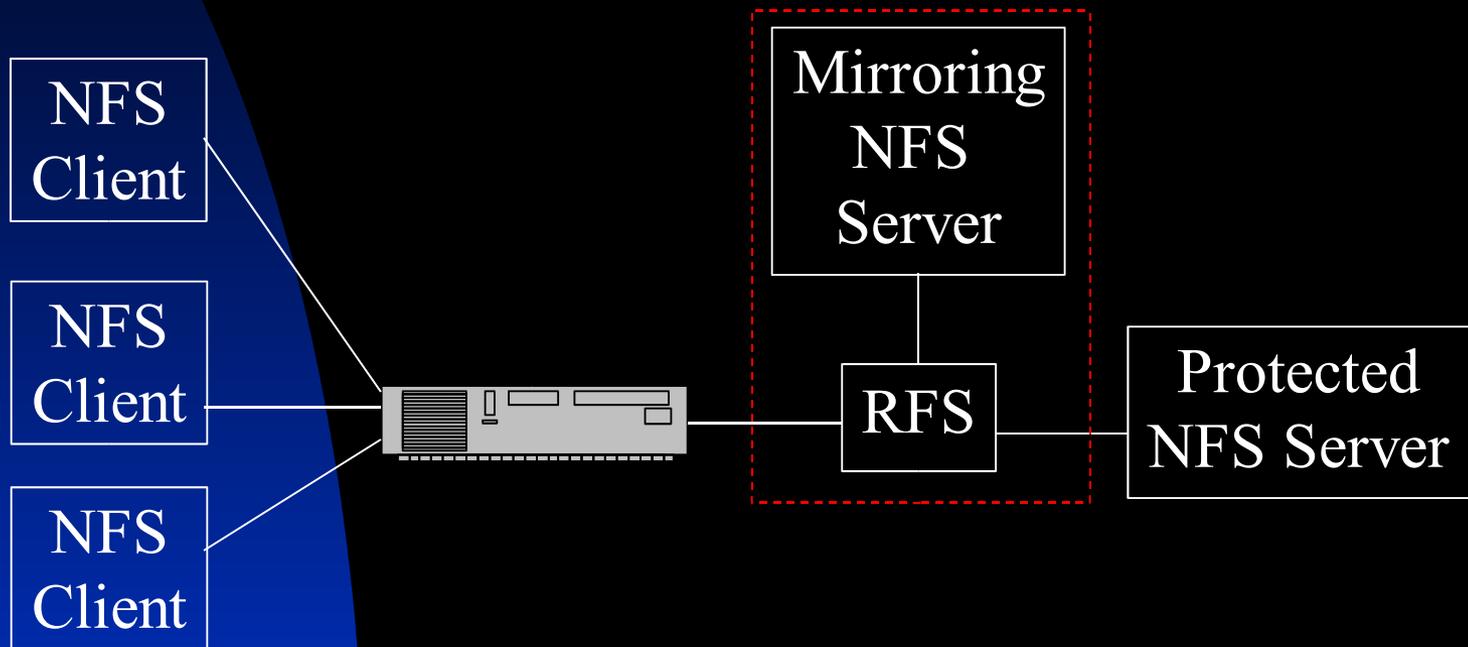
Program	Overhead
BIND	1.05%
DHCP Server	1.23%
PowerPoint	3.44%
Outlook Express	1.29%

Repairable File Service (RFS)

- There is no such thing as unbreakable computer systems, e.g., insider job and social engineering
- A significant percentage of financial loss of computer security breaches is productivity loss due to unavailability of information and personnel
- Instead of aiming at 100% penetration proof, shift the battleground to fast recovery from intrusion: reliability vs. availability → $MTTF/(MTTF+MTTR)$
- Key problem: Accurately identify the damaged file blocks and restore them quickly

RFS Architecture

Transparent to protected network file server



Fundamental Issues

- Keeping the before image of all updates so that every update is **undoable**: transparent file server update logging
- Tracking inter-process dependencies for **selective** undo
- Contamination analysis based on inter-process dependencies and ID of the first detected intruder process, P
 - ◆ All updates made by P and its children
 - ◆ All updates by processes that read in contaminated blocks after P's birth time

RFS Prototype

- Implemented on Red Hat 7.1
- Works for both NFSv2 and NFSv3
- A client-side system call logger whose resulting log is tamper proof
- A wire-speed NFS request/response interceptor that deals with network/protocol errors
- A repair engine that performs contamination analysis and selective undo
- **Undo operations are themselves undoable**

Performance Results

- Client-side logging overhead is **5.4%**
- Additional latency introduced by interceptor is between 0.2 to 1.5 msec
- When the write ratio is below **30%**, there is no throughput difference between NFS and NFS/RFS
- Logging storage requirement: 709MBytes/day for a 250-user NFS server in a CS department → a **100-Gbyte** disk can support a detection window of **8** weeks

Program semantics-Aware Intrusion Detection (**PAID**)

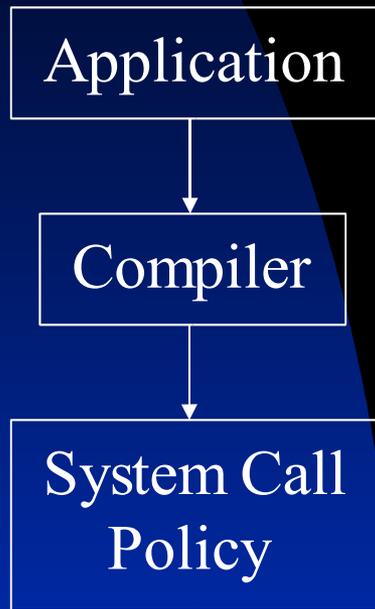
- As a last line of defense, prevent intruders from causing damages even when they successfully take control of a target victim application
- Key observation: Most damages can only be done through system calls, including denial of service attacks
- Idea: prohibit hijacked applications from making arbitrary system calls

System Call Policy/Model

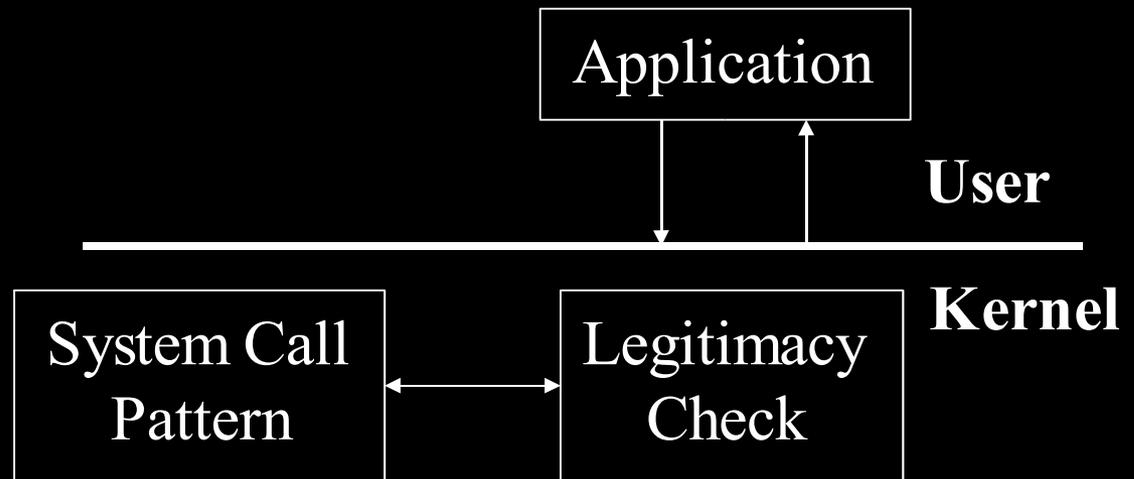
- Manual specification: error-prone, labor intensive, non-scalable
- Machine learning: error-prone, training efforts required
- Our approach: Use compiler to extract the *sites* and *ordering* of system calls from the source code of any given application automatically
- Only host-based intrusion detection systems that guarantees **zero** false positives and **very-close-to-zero** false negatives
- System call policy is extracted **automatically** and **accurately**

PAID Architecture

Compile Time Extraction



Run Time Checking



The Mimicry Attack

- Hijack the control of a victim application by overwriting some control-sensitive data structure, such as return address
- Issue a legitimate sequence of system calls after the hijack point to fool the IDS until reaching a desired system call, e.g., `exec()`
- None of existing commercial or research host-based IDS can handle mimicry attacks

Mimicry Attack Details

- To mount a mimicry attack, attacker needs to
 - ◆ Issue each intermediate system call without being detected
 - Nearly all syscalls can be turned into no-ops
 - For example `(void) getpid()` or `open(NULL, 0)`
 - ◆ Grab the control back during the emulation process
 - Set up the stack so that the injected code can take control after each system call invocation

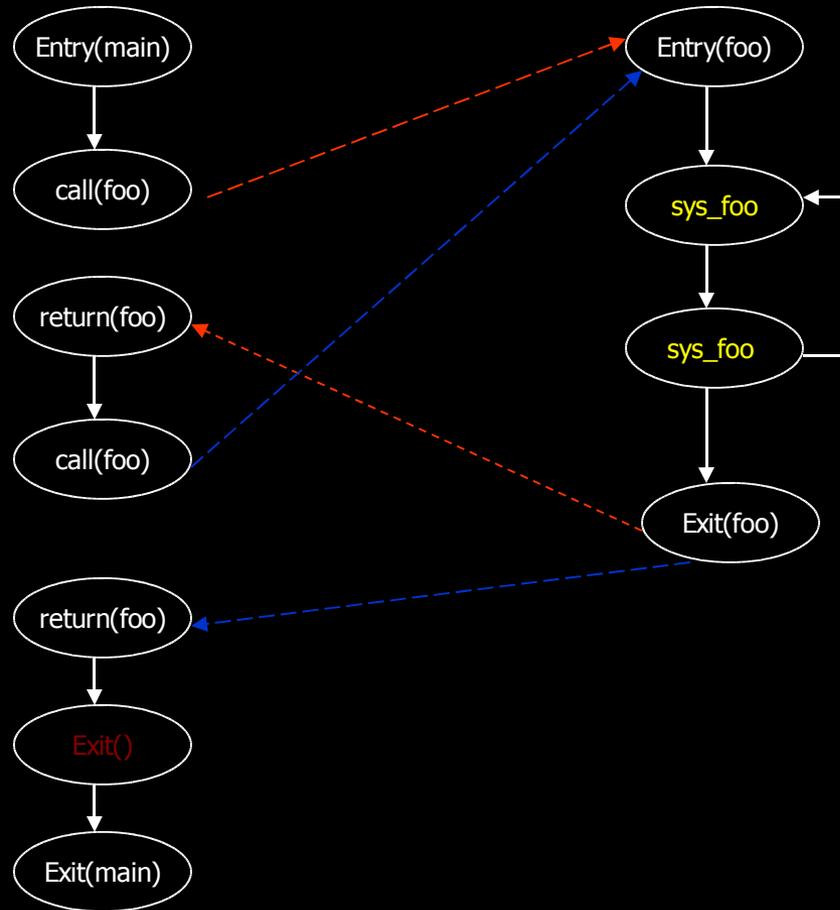
Countermeasures

- Checking system call argument values whenever possible
- Checking the return address chain on the stack to verify the call chain
- Minimize ambiguities in the system call model
 - ◆ If ($a > 1$) { open(..) } else { open(..); write(..) }
 - ◆ Multiple calls to a function that contains a system call

Example

```
main()
{
  foo();
  foo();
  exit();
}

foo()
{
  for(...){
    sys_foo();
    sys_foo();
  }
}
```



System Call Policy Extraction

- From a given program, build a system call graph from its function call graph (FCG) and per-function reduced control flow graph (RCFG)
- For each system call, extract its **memory location**, and derive the **following system call set**
- Each system call site is in-lined with the actual code sequence of entering the kernel (e.g., INT 80), and thus can be uniquely identified

Dynamic Branch Targets

- Not all branch targets are known at compile time: function pointers and indirect jumps
- Insert a **notify** system call to tell the kernel the target address of these indirect branch instructions
- The kernel moves the current cursor of the system call graph to the designated target accordingly
- Notification system call is itself protected

Asynchronous Control Transfer

- Setjmp/Longjmp
 - ◆ At the time of setjmp(), store the current cursor
 - ◆ At the time of longjmp(), restore the current cursor
- Signal handler
 - ◆ When signal is delivered, store the current cursor
 - ◆ After signal handler is done, restore the current cursor
- Dynamically linked library
 - ◆ Load the library's system call graph at run time

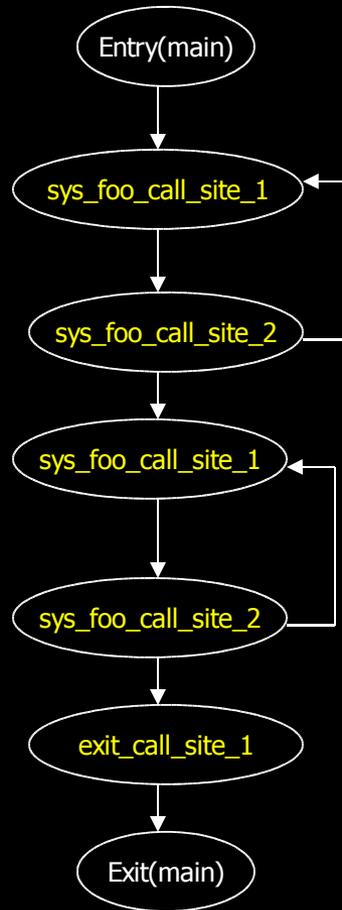
From NFA to DFA

- Use **graph in-lining** to disambiguate the return address for a function with multiple call sites
 - ◆ Every recursive call chain is in-lined and turned into self-recursive call
- Use **system call stub in-lining** to disambiguate two system calls that are identical and that are at two arms of a conditional branch
 - ◆ Does not completely solve the problem: `F1 → system_call()`
 - ◆ Difficult to implement because some glibc functions are written in assembly
- Adding extra `notify()` for further disambiguation

PAID Example

```
main()
{
  foo();
  foo();
  exit();
}
```

```
foo()
{
  for(...){
    sys_foo();
    sys_foo();
  }
}
```

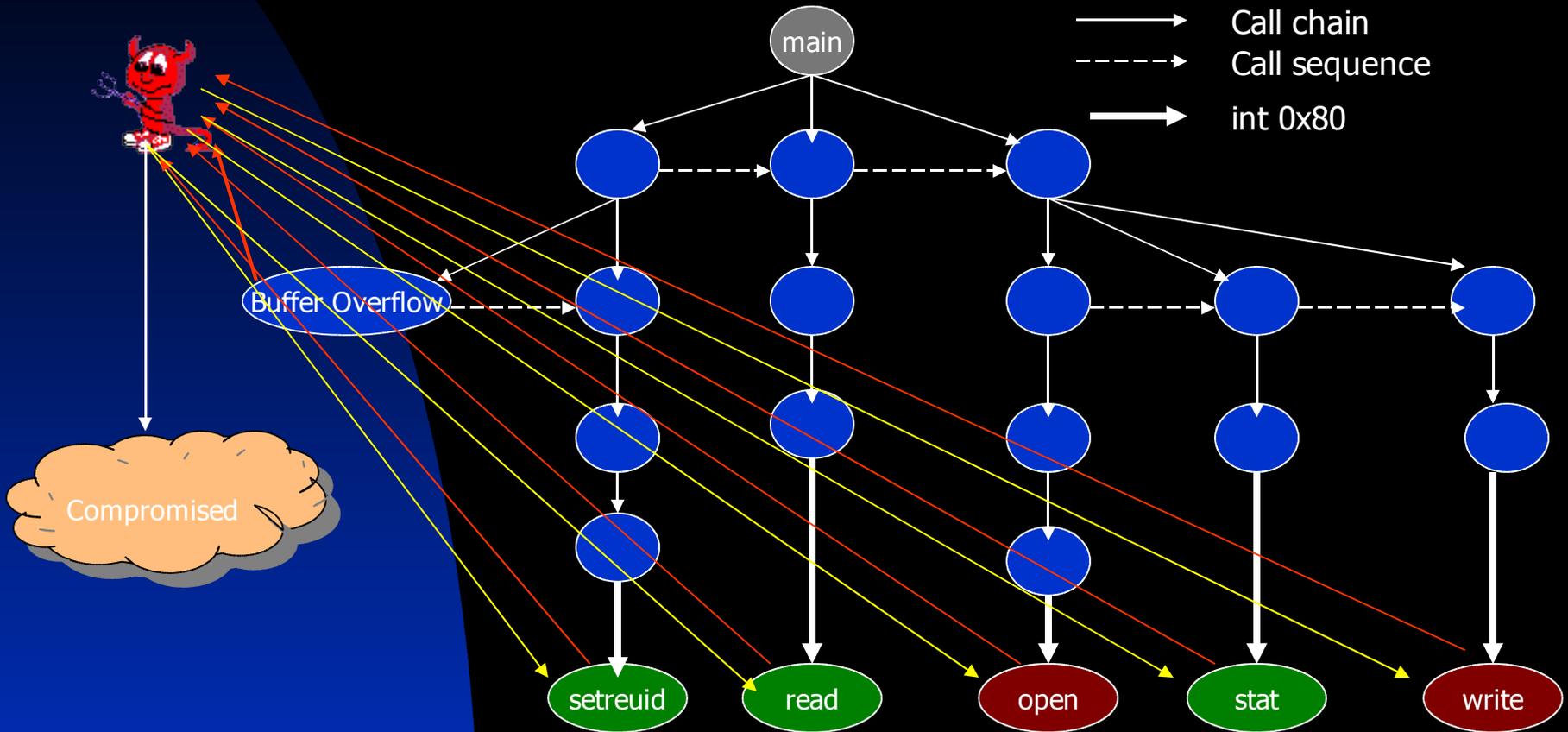


```
foo()
{
  for(...){
    int ret;
    __asm__ ("movl sys_foo_n, %eax\n"
            "int $0x80\n"
            "sys_foo_call_site_1:\n"
            "movl %eax, ret\n"
            "...");
  }
  int ret;
  __asm__ ("movl sys_foo_n, %eax\n"
          "int $0x80\n"
          "sys_foo_call_site_2:\n"
          "movl %eax, ret\n"
          "...");
}
}
```

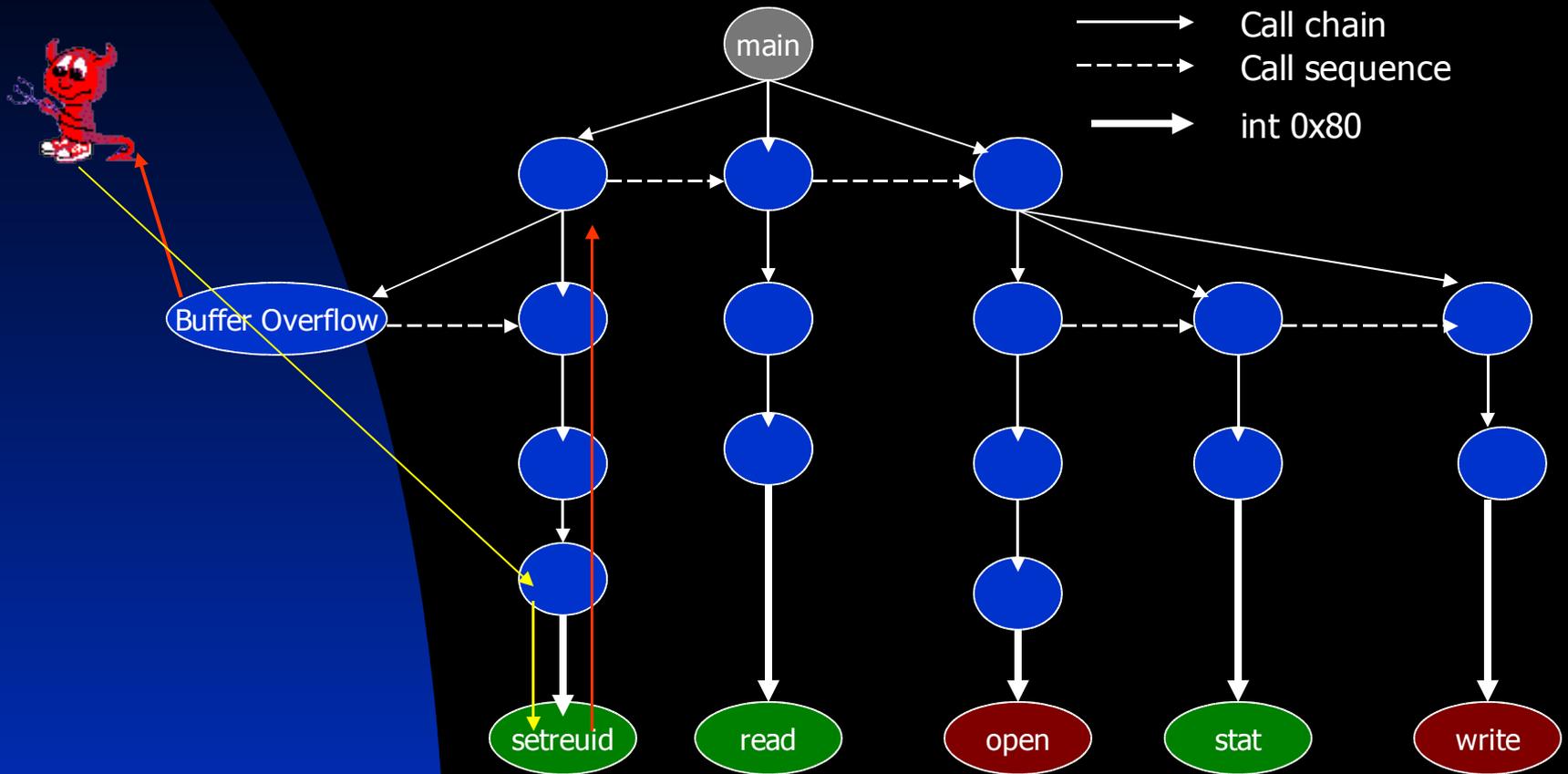
PAID Checks

- Ordering
- Site
- Insertion of random notify() at load time
 - ◆ Different for different instance
- Stack return address check
 - ◆ Ensure they are in the text area
- Checking performed in the kernel
 - In most cases, only two comparisons are needed

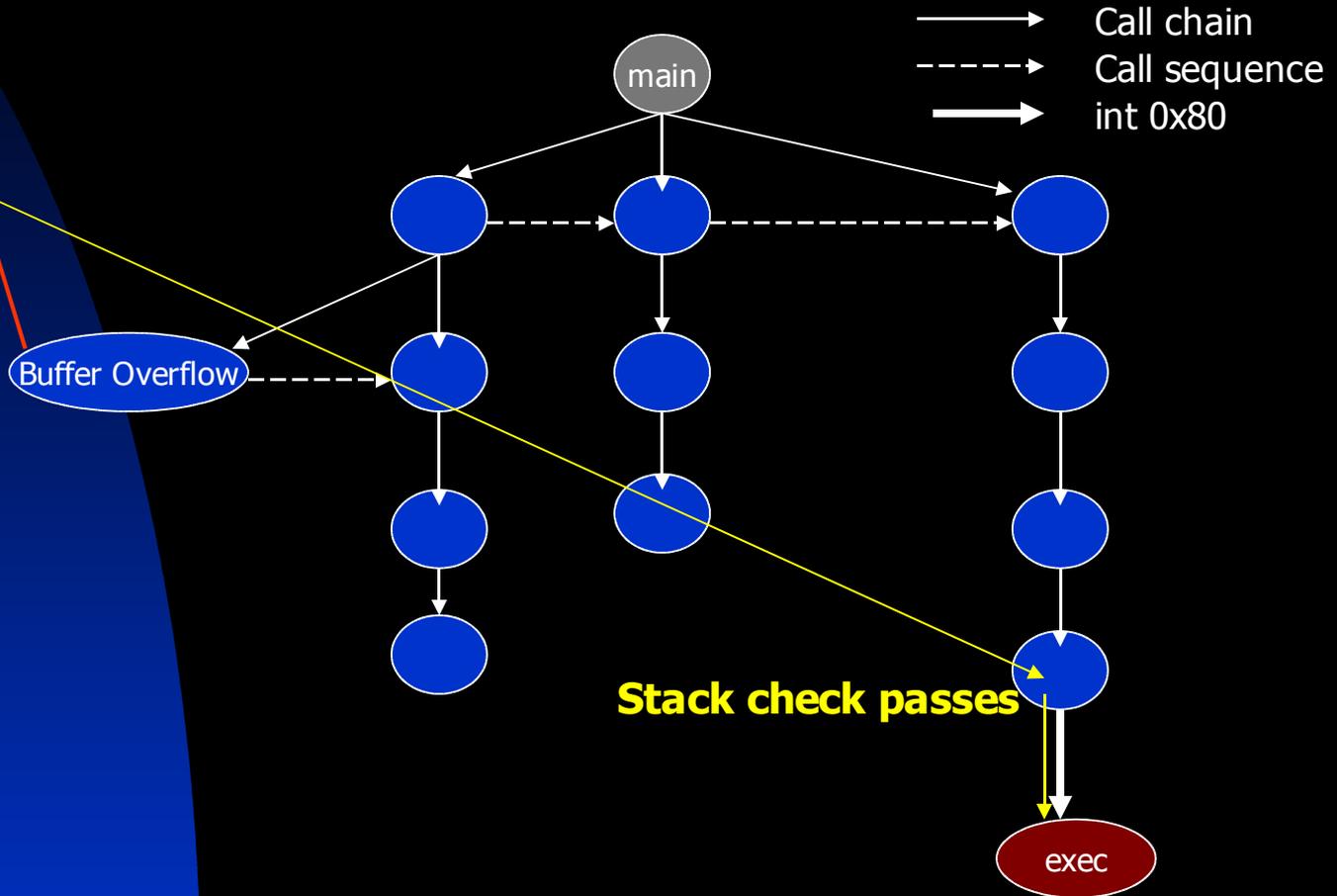
Ordering and Site Check



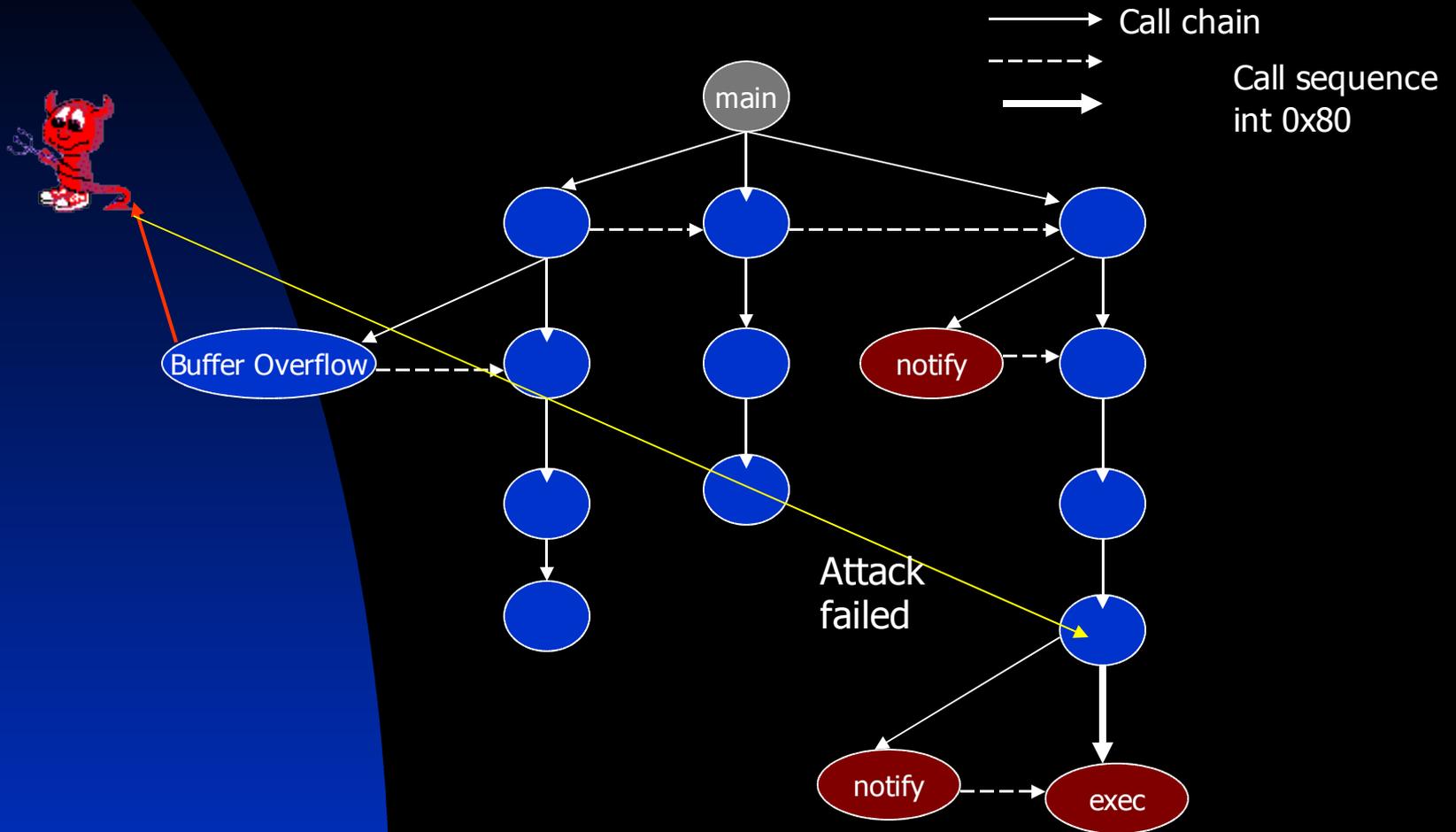
Ordering, Site and Stack Check (1)



Ordering, Site and Stack Check (2)



Random Insertion of Notify Calls



Alternative Approach

- Check the return address chain on the stack every time a system call is made
 - ◆ Every system call instance can be uniquely identified by a function call chain **and** the return address for the INT 80 instruction
 - ◆ Main → F1 → F2 → F4 → system_call_1 vs.
Main → F3 → F5 → F4 → system_call_1
- Need to check the legitimacy of transitioning from one system call to another
- No graph or function in-lining is necessary

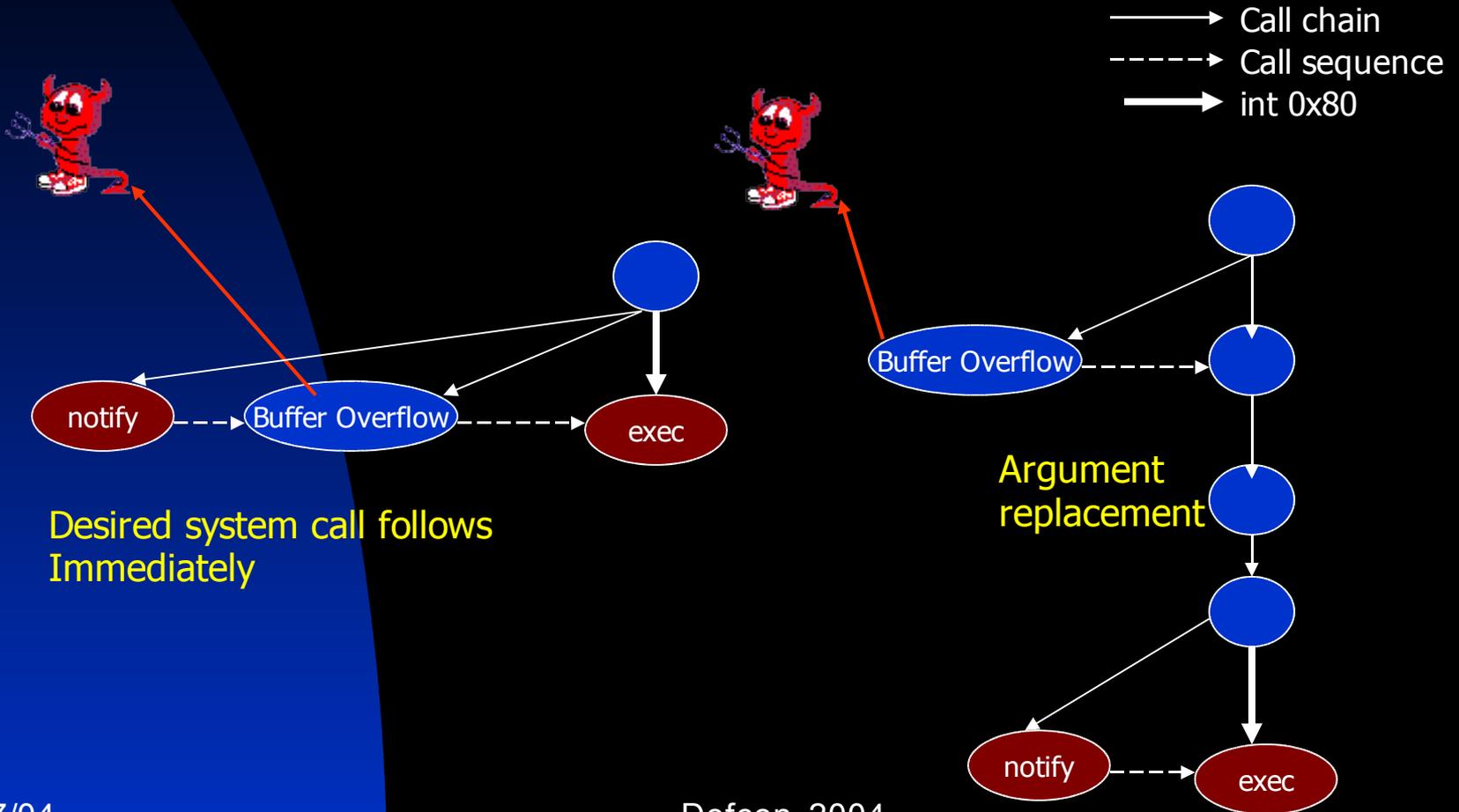
System Call Argument Check

- Start from each “file name” system call argument, e.g., `open()` and `exec()`, and compute a backward slice,
- Perform symbolic constant propagation through the slice, and the result could be
 - ◆ A constant: **static constant**
 - ◆ A program segment that depends on initialization-time inputs only: **dynamic constant**
 - ◆ A program segment that depends on run-time inputs: **dynamic variables**

Dynamic Variables

- Derive partial constraints, e.g., prefix or suffix, “/home/httpd/html”
- Enforce the system call argument computation path by inserting null system calls between where dynamic inputs are entered and where the corresponding system call arguments are used

Vulnerabilities



Prototype Implementation

- GCC 3.1 and Gnu ld 2.11.94, Red Hat Linux 7.2
- Compiles GLIBC successfully
- Compiles several production-mode network server applications successfully, including Apache-1.3.20, Qpopper-4.0, Sendmail-8.11.3, Wuftpd-2.6.0, etc.

Throughput Overhead

	PAID	PAID/stack	PAID/random	PAID/stack random
Apache	4.89%	5.39%	6.48%	7.09%
Qpopper	5.38%	5.52%	6.03%	6.22%
Sendmail	6.81%	7.73%	9.36%	10.44%
Wuftp	2.23%	2.69%	3.60%	4.38%

Conclusion

- Paid is the most efficient, comprehensive and accurate host-based intrusion prevention (HIPS) system on Linux
 - ◆ Automatically generates per-application system call policy
 - ◆ System call policy is in the form of deterministic finite automata to eliminate ambiguities
 - ◆ Extensive system call argument checks
 - ◆ Can handle function pointers and asynchronous control transfers
 - ◆ Guarantee no false positives
 - ◆ Very small false negatives
 - ◆ Can block most mimicry attacks

Future Work

- Support for threads
- Integrate it with SELinux
- Derive a binary PAID version for Windows platform
- Further reduce the latency/throughput overhead
- Reduce the percentage of “dynamic variable” category of system call arguments

For more information

Project Page: <http://www.ecsl.cs.sunysb.edu/PAID>

Thank You!