CE 874 - Secure Software Systems

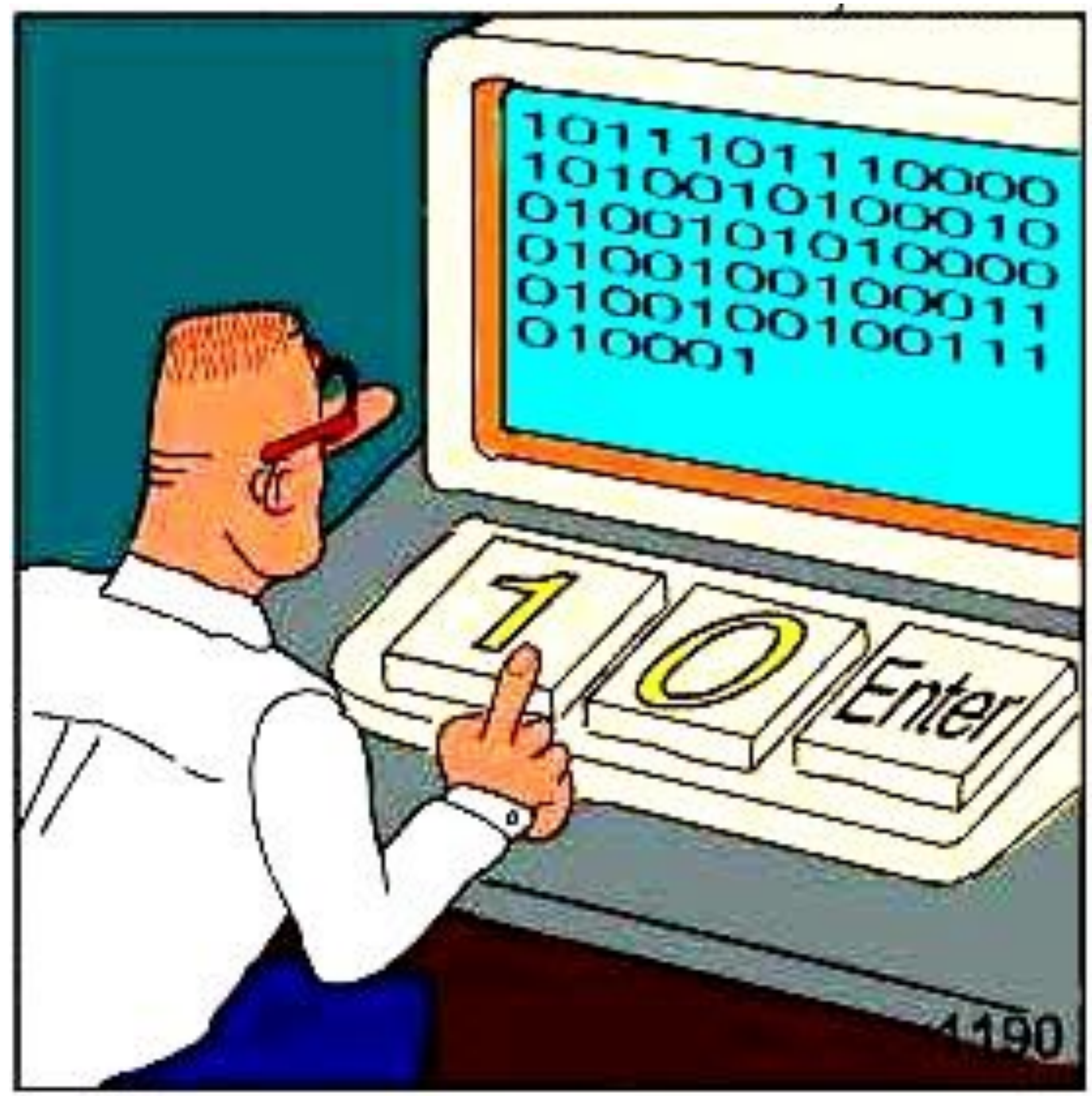
Run-Time protection/enforcement

Mehdi Kharrazi
Department of Computer Engineering
Sharif University of Technology

Acknowledgments: Some of the slides are fully or partially obtained from other sources. Reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.

Run-Time protection/enforcement

- In many instances we only have access to the binary
- How do we analyze the binary for vulnerabilities?
- How do we protect the binary from exploitation?
- This would be our topic for this lectures



REAL Programmers code in BINARY.

Why Binary Code?

- Access to the source code often is not possible:
  - Proprietary software packages
  - Stripped executables
  - Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries)
- Binary code is the only authoritative version of the program
  - Changes occurring in the compile, optimize and link steps can create non-trivial semantic differences from the source and binary
- Worms and viruses are rarely provided with source code

Binary Analysis and Editing

- **Analysis**: processing of the binary code to extract syntactic and symbolic information
  - Symbol tables (if present)
  - Decode (disassemble) instructions
  - Control-flow information: basic blocks, loops, functions
  - Data-flow information: from basic register information to highly sophisticated (and expensive) analysis

Binary Analysis and Editing

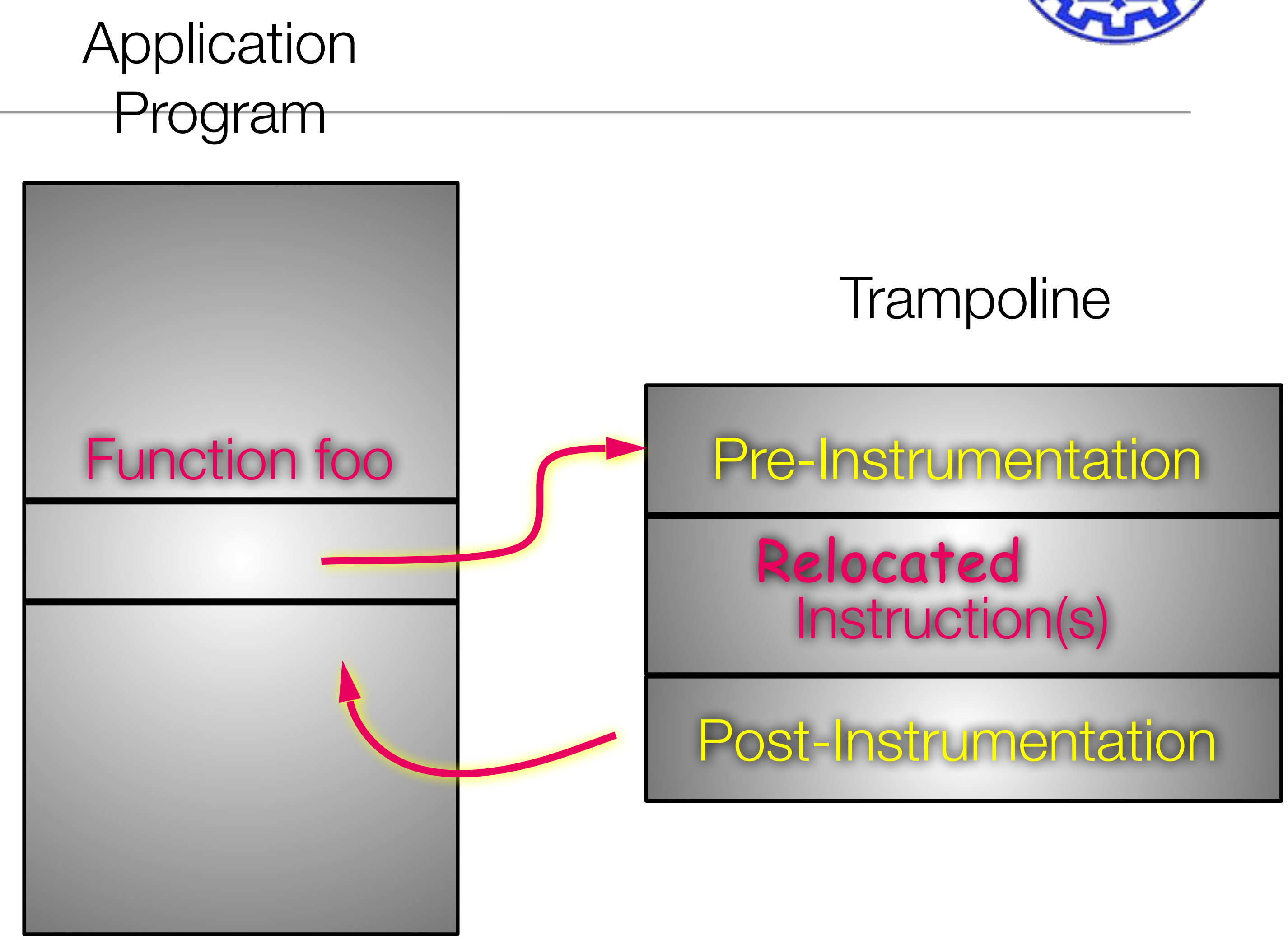
- **Binary rewriting**: static (before execution) modification of a binary program
  - Analyze the program and then insert, remove, or change the binary code, producing a new binary

- **Dynamic instrumentation**: dynamic (during execution) modification of a binary program
  - Analyze the code of the running program and then insert, remove, or change the binary code, changing the execution of the program
  - Can operate on running programs and servers

Uses of Binary Analysis and Editing

- Cyber-forensics
  - Analysis: understand the nature of malicious code
  - Binary-rewriting: produce a new version of the code that might be instrumented, sandboxed, or modified for study
  - Dynamic instrumentation: same features, but can do it interactively on an executing program
  - Hybrid static/dynamic: control execution and produce intermediate versions of the binary that can be re-executed (and further instrumented)
- Program tracing: instructions, memory accesses, function calls, system calls, . . .
- Debugging
- Testing, Performance profiling Performance modeling
- Reverse engineering

Binary patch



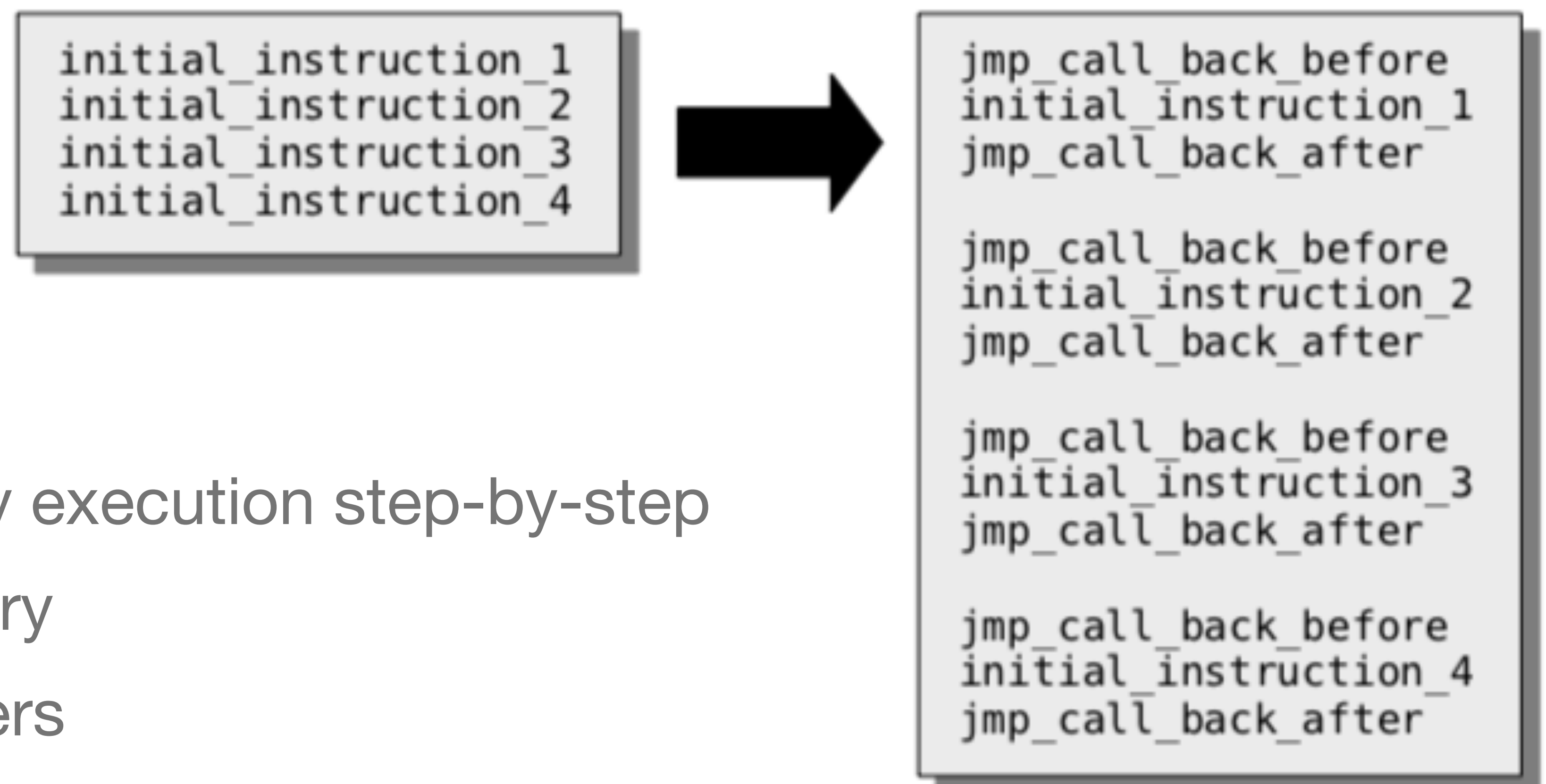
pop ecx; puts the return address to ecx
jmp ecx; jumps to the return address

After Patch:

pop ecx; puts the return address to ecx
cmp ecx , 0x08048456 ; check that we return to the right place
jne 0x41414141 ; crash
jmp ecx; effectively return

Dynamic Binary Instrumentation

- A DBI is a way to execute an external code before or/and after each instruction/routine



- With a DBI you can:
  - Analyze the binary execution step-by-step
    - Context memory
    - Context registers
  - Only analyze the executed code

Available Tools

- Binary re-writing:
  - e.g.: Alto, Vulcan, Diablo, etc.
- Binary Instrumnetation:
  - e.g. PIN, Valgrind, DynInst, etc

Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software, J. Newsome and D. Song, NDSS 2005.

Motivation

- Worms exploit several software vulnerabilities
  - buffer overflow
  - “format string” vulnerability
- Attack detectors ideally should:
  - Detect new attacks and detect them early
  - Be easy to deploy
  - Few false positives and false negatives
  - Be able to automatically generate filters and sharable fingerprints

Motivation (contd.)

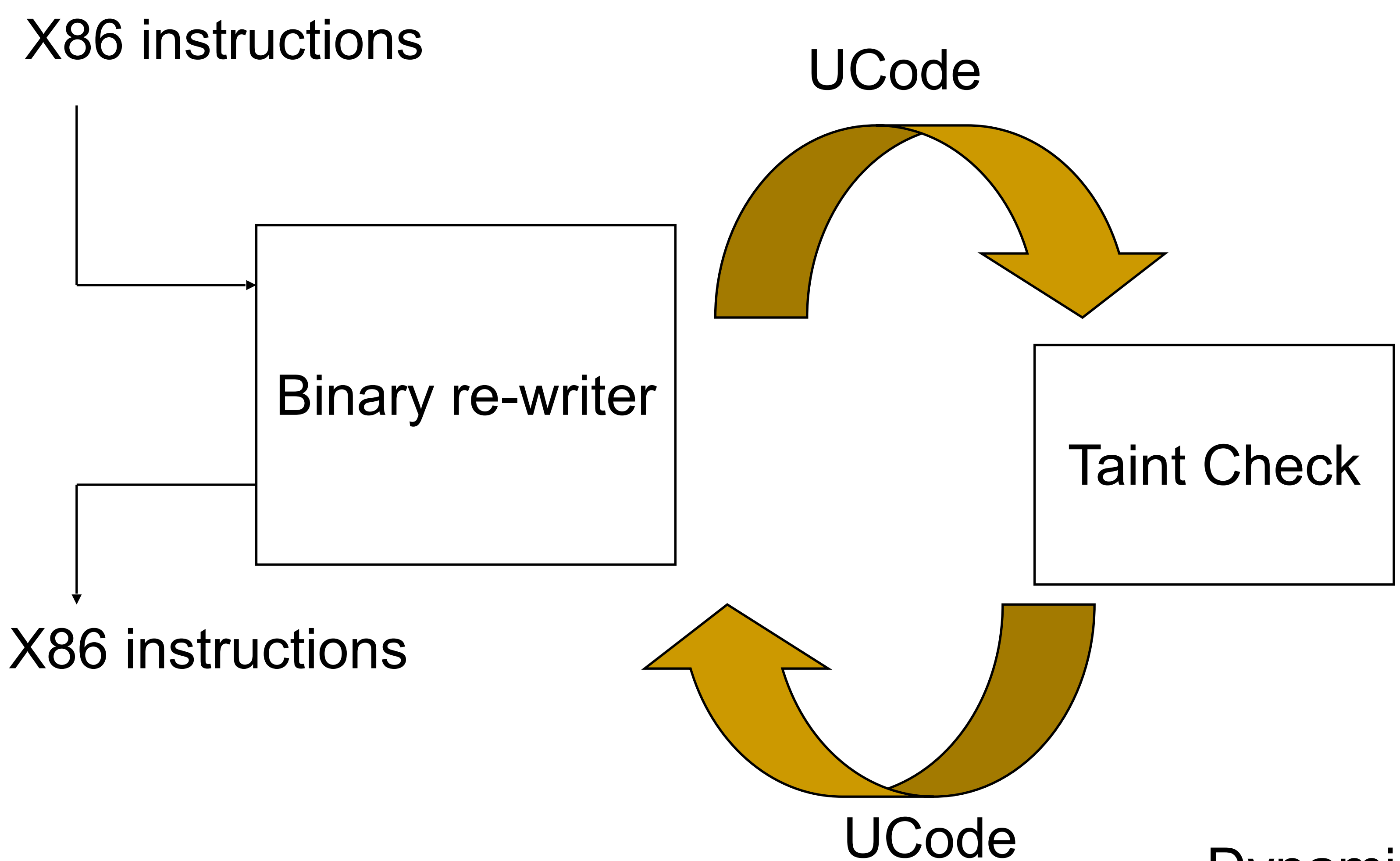
- Attack detectors are:
  - Coarse grained detectors
    - Detect anomalous behavior but do not provide detailed information about the vulnerability
    - Scan detectors, anomaly detectors
  - Fine grained detectors are highly desirable
    - Detect attacks on programs vulnerabilities and hence provide detailed information about the attack
    - But some require source code (typically not available for commercial software), recompilation, bounds checking, library recompilation, source code modification, etc.
  - Other options: content-based filtering (e.g., IDS' such as snort and Bro), but automatic signature generation is hard

TaintCheck: Basic Ideas

- Program execution normally derived from trusted sources, not attacker input
- Mark all input data to the computer as “tainted” (e.g., network, stdin, etc.)
- Monitor program execution and track how tainted data propagates (follow bytes, arithmetic operations, jump addresses, etc.)
- Detect when tainted data is used in dangerous ways

Step 1: Add Taint Checking code

- TaintCheck first runs the code through an emulation environment (Valgrind) and adds instructions to monitor tainted memory.



Dynamic taint analysis

TaintCheck Detection Modules

Figure 1. TaintCheck detection of an attack. (Exploit Analyzer not shown).

- TaintSeed: Mark untrusted data as tainted
- TaintTracker: Track each instruction, determine if result is tainted
- TaintAssert: Check is tainted data is used dangerously
  - Jump addresses: function pointers or offsets
  - Format strings: is tainted data used as a format string arg?
  - System call arguments
  - Application or library customized checks

TaintSeed

- Marks any data from untrusted sources as "tainted"
  - Each byte of memory has a four-byte shadow memory that stores a pointer to a Taint data structure if that location is tainted
    - records the system call number, a snapshot of the current stack and a copy of the data that was written.
  - Else store a NULL pointer

Memory is mapped to TDS

TaintTracker

- Tracks each instruction that manipulates data in order to determine whether the result is tainted.
  - When the result of an instruction is tainted by one of the operands, TaintTracker sets the shadow memory of the result to point to the same Taint data structure as the tainted operand.

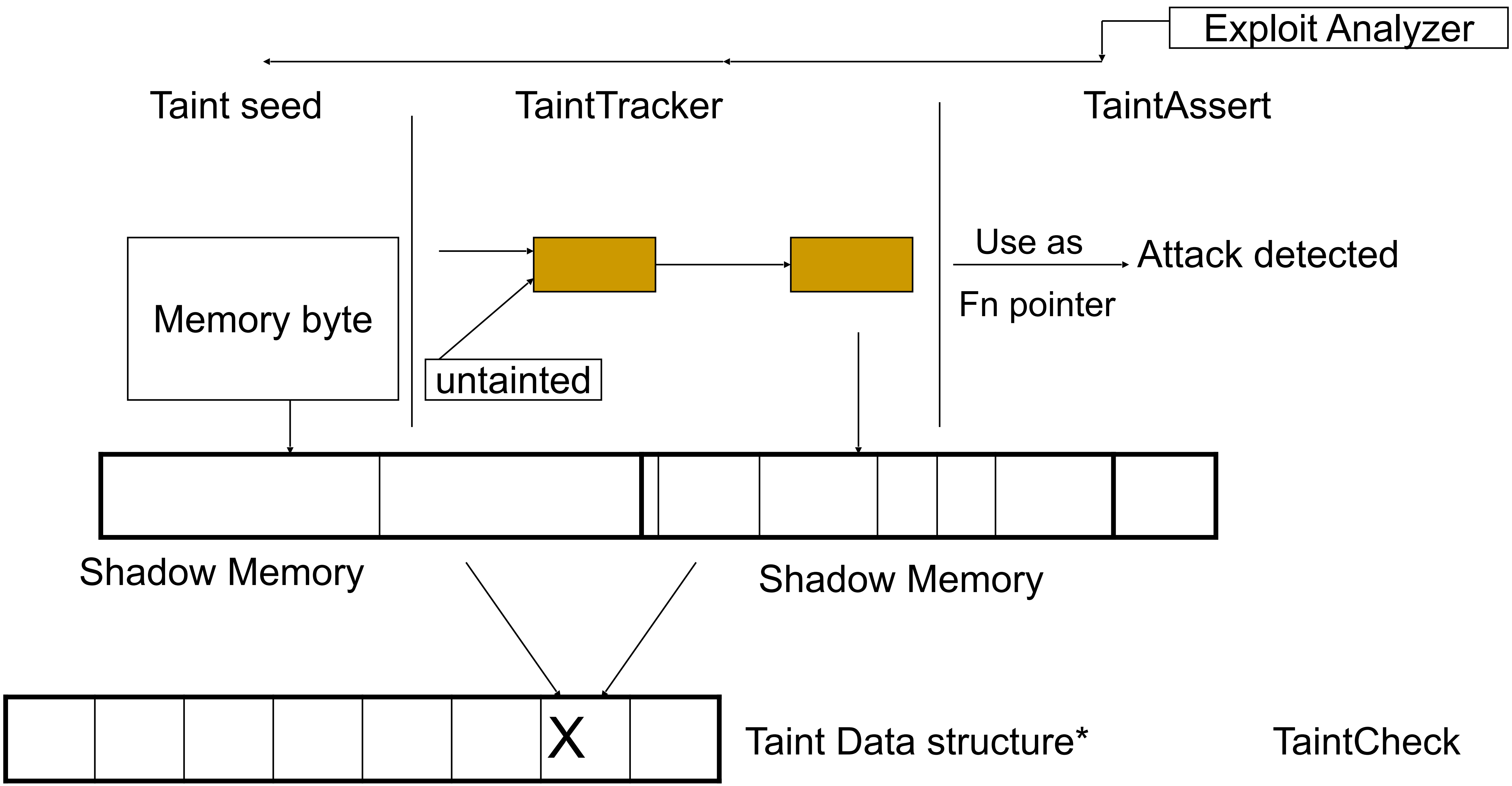
Memory is mapped to TDS

Result is mapped to TDS

TaintAssert

- Checks whether tainted data is used in ways that its policy defines as illegitimate

TaintCheck Operation



*TDS holds the system call number, a snapshot of the current stack, and a copy of the data that was written

Exploit Analyzer

- Provides useful information about how the exploit happened, and what the exploit attempts to do
- Useful to generate exploit fingerprints

- Usage:
  - Identifying vulnerabilities.
  - Generating exploit signature.

Dynamic Taint Analysis

- Jump addresses:
  - Checks whether tainted data is used as a jump target
  - Instrument before each Ucode jump instruction
- Format strings:
  - Checks whether tainted data is used as format string argument
  - Intercept calls to the printf family of functions
- System call arguments:
  - Checks whether the arguments specified in system calls are tainted
  - Optional policy for execv system call
- Application or library-specific checks:
  - To detect application or library specific attacks

When does TaintCheck Fail?

- A false negative occurs if an attacker can cause sensitive data to take on a value without that data becoming tainted
    - E.g. if (x == 0)y = 0; else if (x == 1) y = 1; ...
- If values are copied from hard-coded literals, rather than arithmetically derived from the input
    - IIS translates ASCII input into Unicode via a table
- If TaintCheck is configured to trust inputs that should not be trusted
    - data from the network could be first written to a file on disk, and then read back into memory

When does TaintCheck give a False Positive?

- TaintCheck detects that tainted data is being used in an illegitimate way even when there is no attack taking place. Possibilities:
  - There are vulnerabilities in the program and need to be fixed, or
  - The program performs sanity checks before using the data

tainted untainted

```
● = get_input( 
```


```
)
y = x + 42
...
goto y
```

Input is tainted

TaintSeed

Δ

Var	Val

τ

Var	Tainted?

tainted untainted

= get_input()

= +

…

goto y

Data derived from user input is tainted

TaintTracker

Δ

Var	Val
x	7
y	49

τ

Var	Tainted?
x	T
y	T

tainted untainted

= get_input()

= +

...

goto

Policy Violation Detected

TaintAssert

Δ

Var	Val
x	7
y	49

τ

Var	Tainted?
x	T
y	T

x = get_input()

y = ...

...

goto y

Jumping to overwritten return address

```
...
strcpy(buffer,argv[1])
;
...
return ;
```

Memory Load

Variables

Δ

Var	Val
x	7

τ

Var	Tainted?
x	T

Memory

μ

Addr	Val
7	42

τ_{μ}

Addr	Tainted?
7	F

Problem: Memory Addresses

```
  = get_input(        )
y = load(  )
...
goto  y
```

All values derived from user input are tainted??

Δ

Var	Val
x	7

μ

Addr	Val
7	42

τ_μ

Addr	Tainted?
7	F

Policy 1: Taint depends only on the memory cell

● = get_i
● = load(
…
goto ●

Taint Propagation

Undertainting

Failing to identify tainted values
- e.g., missing exploits

Δ

Var	Val
	7

ldr	Val
7	42

τ_μ

Addr	Tainted?
7	F

Policy 2: If either the address or the memory cell is tainted, then the value is tainted

= get_i
= load

...

goto

Overtainting

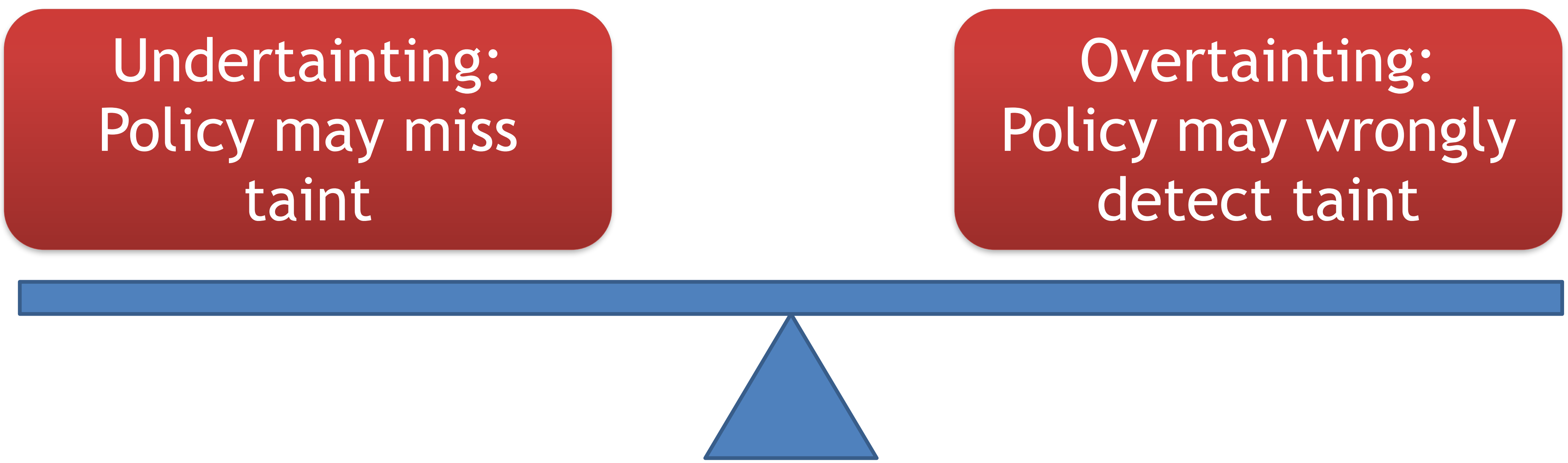
Unaffected values are tainted
- e.g., exploits on safe inputs

Memory
Address expression is tainted
printa
printb

Taint Propagation

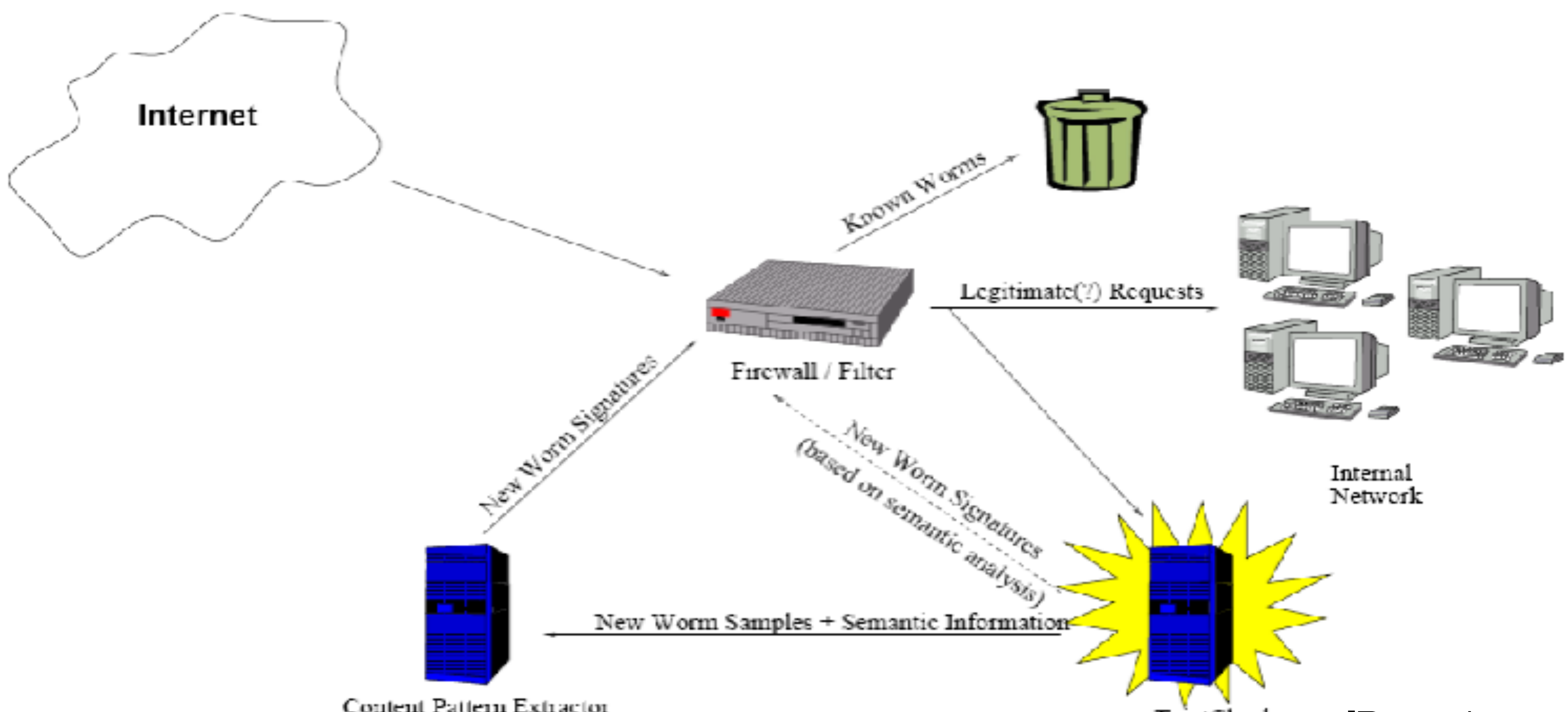
General Challenge

- State-of-the-Art is not perfect for all programs



Automatic Signature Generation

- Automatic semantic analysis based signature generation
  - Find value used to override return address – typically fixed value in the exploit code
  - Sometimes as little as 3 bytes! See paper for details



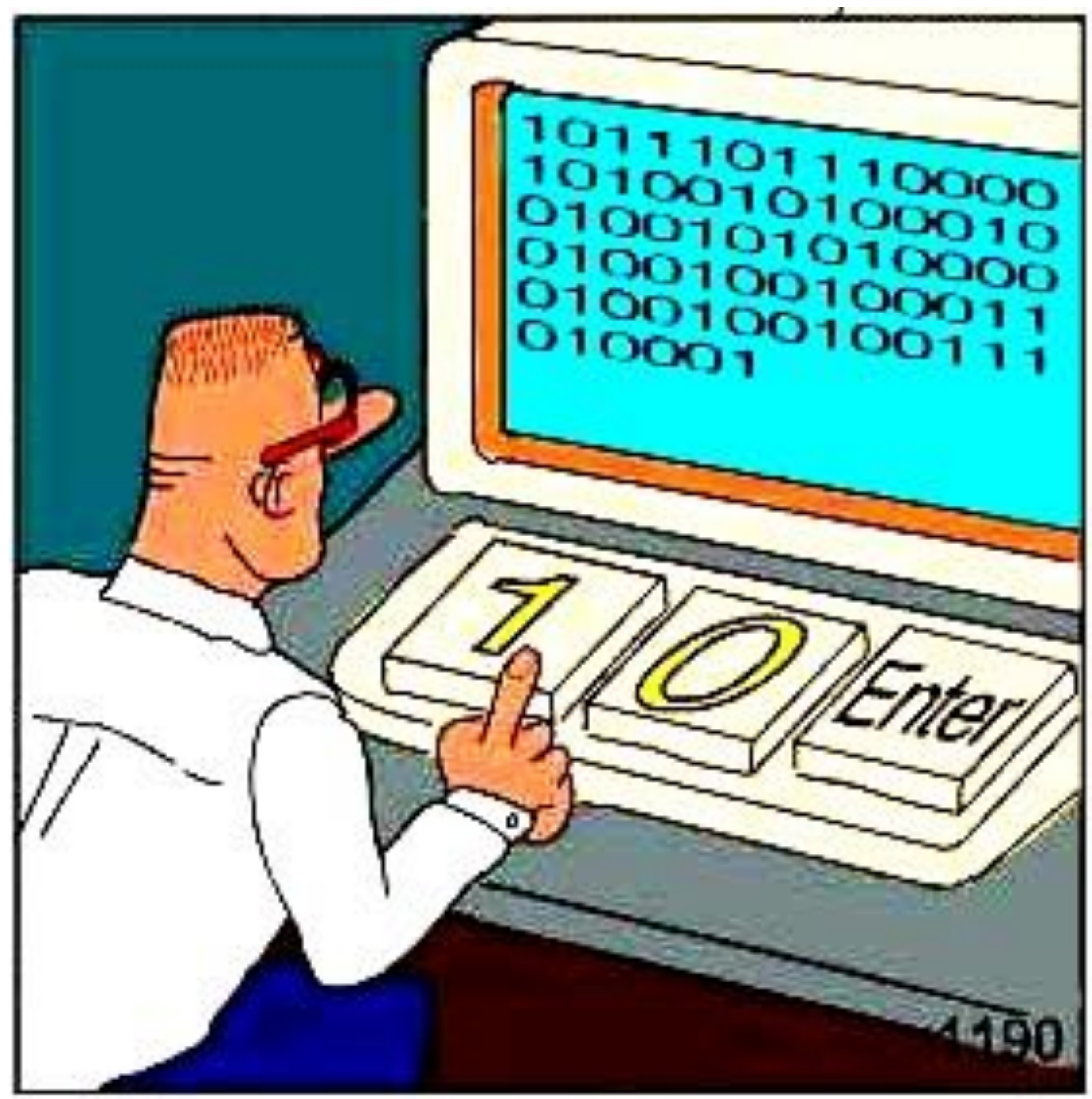
[Papadopoulos'11]

More recent work

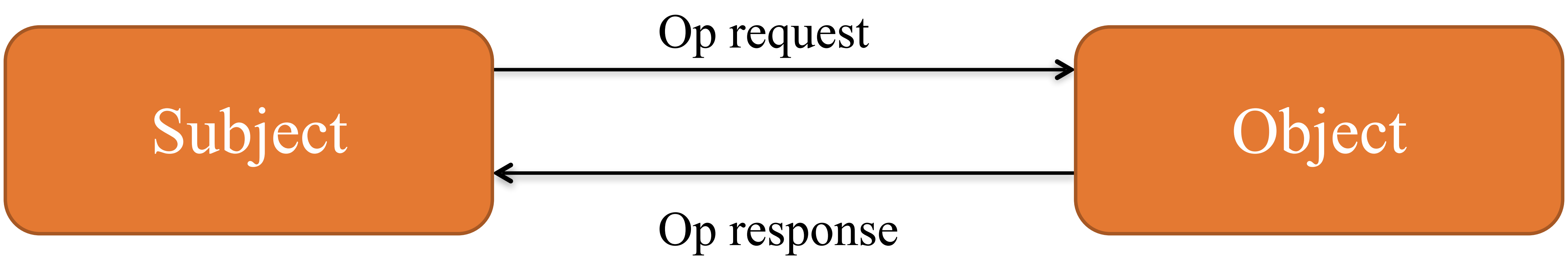
- Improving performance:
  - TaintPipe: Pipelined Symbolic Taint Analysis, Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu, Usenix Security 2015.
  - DECAF++: Elastic Whole-System Dynamic Taint Analysis, Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin, Raid 2019.
  - SelectiveTaint: Efficient Data Flow Tracking With Static Binary Rewriting, Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang, Usenix Security, 2021
- Extending to GPU
  - GPU Taint Tracking, Ari B. Hayes, Lingda Li, Mohammad Hedayati, Jiahuan He, Eddy Z. Zhang, Kai Shen, Usenix ATC, 2017.

Run-Time protection/enforcement

- In many instances we only have access to the binary
- How do we analyze the binary for vulnerabilities?
- How do we protect the binary from exploitation?
- This would be our topic for the next few lectures



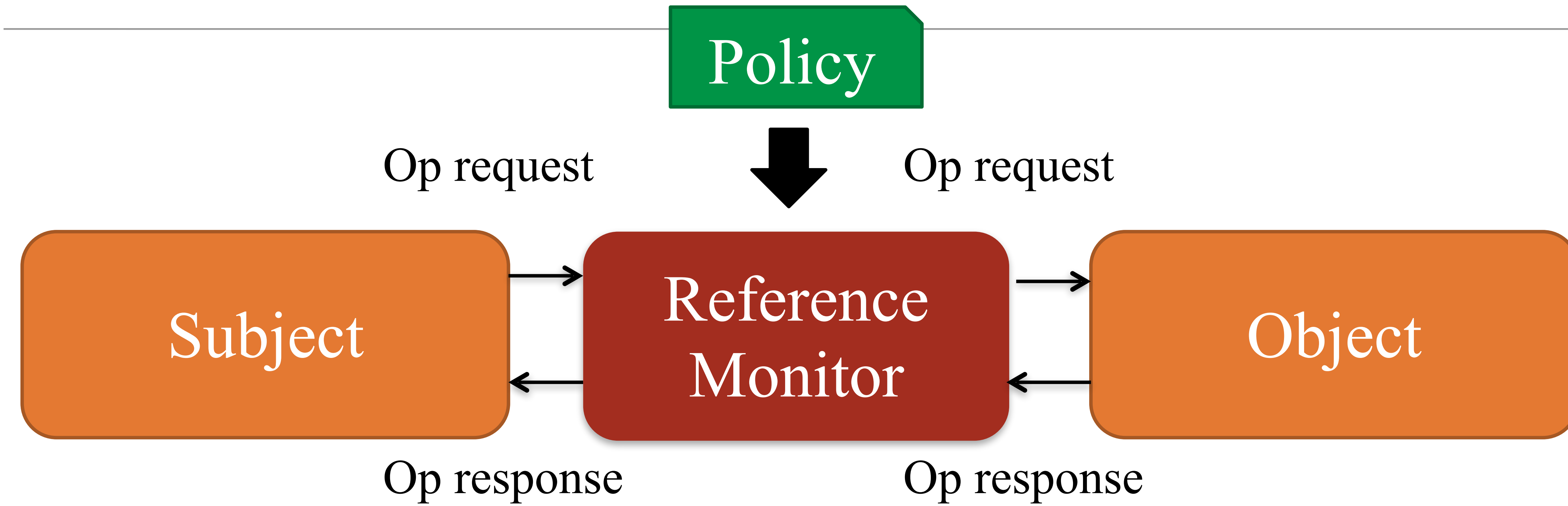
REAL Programmers code in BINARY.



People
Processes
Computer Operations

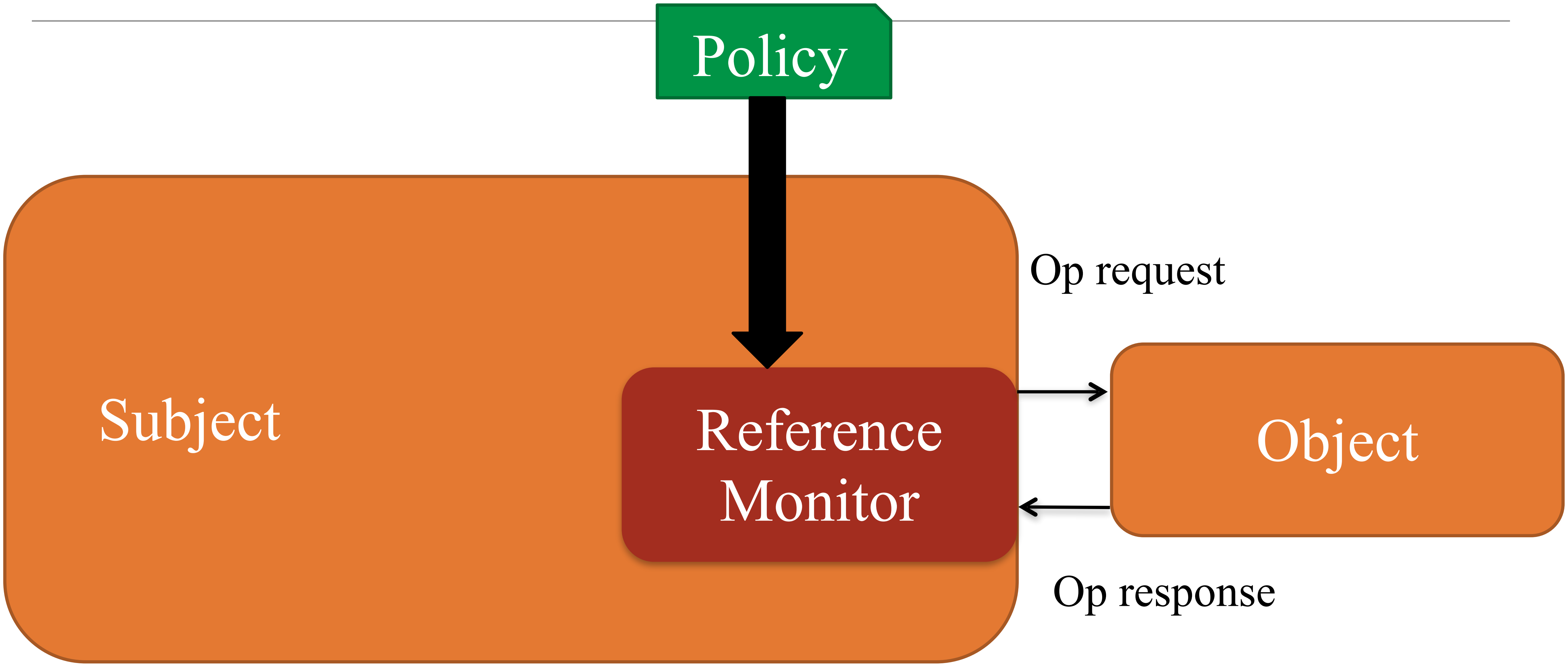
Files
Sockets
Computer Operations

Reference Monitor: Principles



- **Complete Mediation:** The reference monitor must always be invoked
- **Tamper-proof:** The reference monitor cannot be changed by unauthorized subjects or objects
- **Verifiable:** The reference monitor is small enough to thoroughly understand, test, and ultimately, verify.

Inlined Referenced Monitor



Today’s Example:
Inlining a control flow policy into a program

Control-Flow Integrity: Principles, Implementations, and Applications

Martin Abadi, Mihai Budiu, U´lfar Erlingsson, Jay Ligatti, CCS 2005

Control Flow Integrity

- protects against powerful adversary
  - with full control over entire $\boxed{\text{data memory}}$
- widely-applicable
  - language-neutral; requires binary only
- provably-correct & trustworthy
  - formal semantics; small verifier
- efficient
  - hmm… 0-45% in experiments; average 16%

CFI Adversary Model

Can

- Overwrite any data memory at any time
  - stack, heap, data segs
- Overwrite registers in current context

Can Not

- Execute Data
  - NX takes care of that
- Modify Code
  - text seg usually read-only
- Write to %ip
  - true in x86
- Overwrite registers in other contexts
  - kernel will restore regs

CFI Overview

- Invariant: Execution must follow a path in a control flow graph (CFG) created ahead of run time.

“static”

- Method:
  - build CFG statically, e.g., at compile time
  - instrument (rewrite) binary, e.g., at install time
    - add IDs and ID checks; maintain ID uniqueness
  - verify CFI instrumentation at load time
    - direct jump targets, presence of IDs and ID checks, ID uniqueness
  - perform ID checks at run time
    - indirect jumps have matching IDs

Control Flow Graphs

Basic Block

• D

control is "straight"
(no jump targets except at the beginning,
no jumps except at the end)

sequence

1. x = y + z
2. z = t + i

3. x = y + z
4. z = t + i
5. jmp 1

6. jmp 3

3 static basic blocks

1. x = y + z
2. z = t + i
3. x = y + z
4. z = t + i
5. jmp 1

1 dynamic basic block

CFG Definition

- A static Control Flow Graph is a graph where
  - each vertex v_i is a basic block, and
  - there is an edge (v_i, v_j) if there may be a transfer of control from block v_i to block v_j.

- Historically, the scope of a "CFG" is limited to a function or procedure, i.e., intra-procedural.

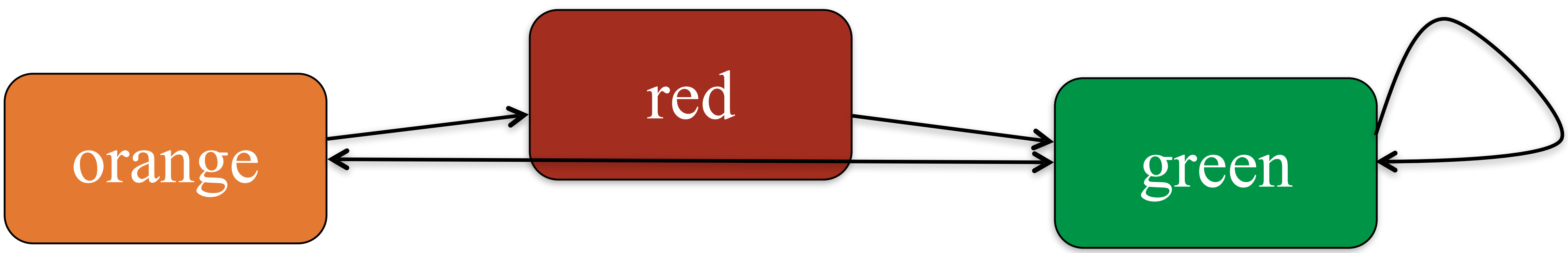
Call Graph

- Nodes are functions. There is an edge (v_i, v_j) if function v_i calls function v_j.

```
void orange()
{
1. red(1);
2. red(2);
3. green();
}
```

```
void red(int x)
{
green();
...
}
```

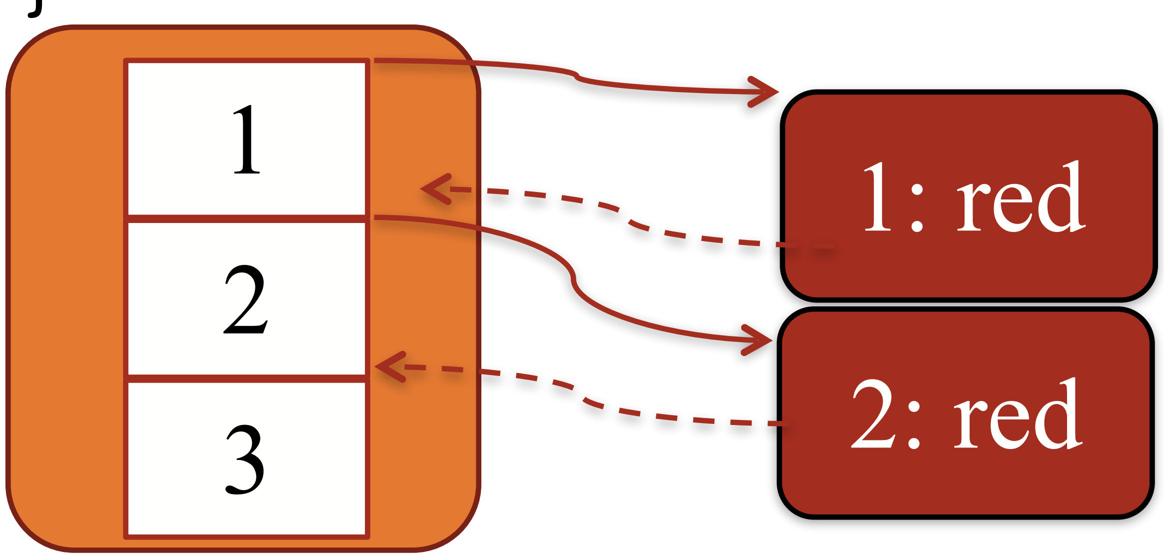
```
void green()
{
  green();
  orange();
}
```



Super Graph

- Superimpose CFGs of all procedures over the call graph

```
void orange()      void red(int x)      void green()
{                  {                    {
1. red(1);         ..                     green();
2. red(2);         }                      orange();
3. green();                             }
}
```



A *context sensitive* super-graph for orange lines 1 and 2.

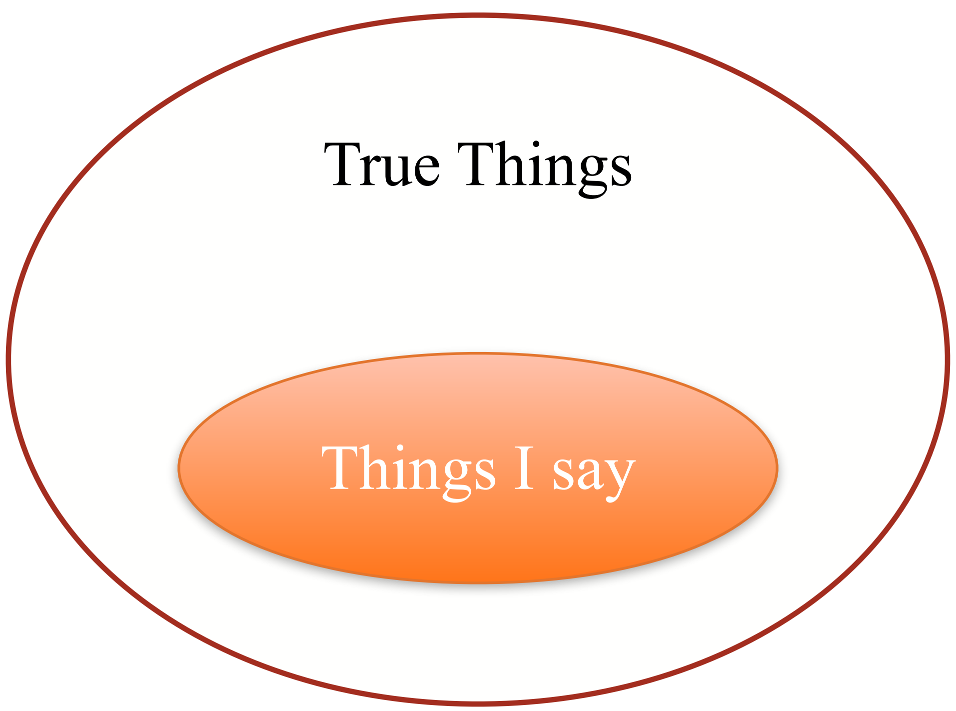
Precision: Sensitive or Insensitive

- The more precise the analysis, the more accurate it reflects the “real” program behavior.
  - More precise = more time to compute
  - More precise = more space
  - Limited by soundness/completeness tradeoff

- Common Terminology in any Static Analysis:
  - Context sensitive vs. context insensitive
  - Flow sensitive vs. flow insensitive
  - Path sensitive vs. path insensitive

Soundness

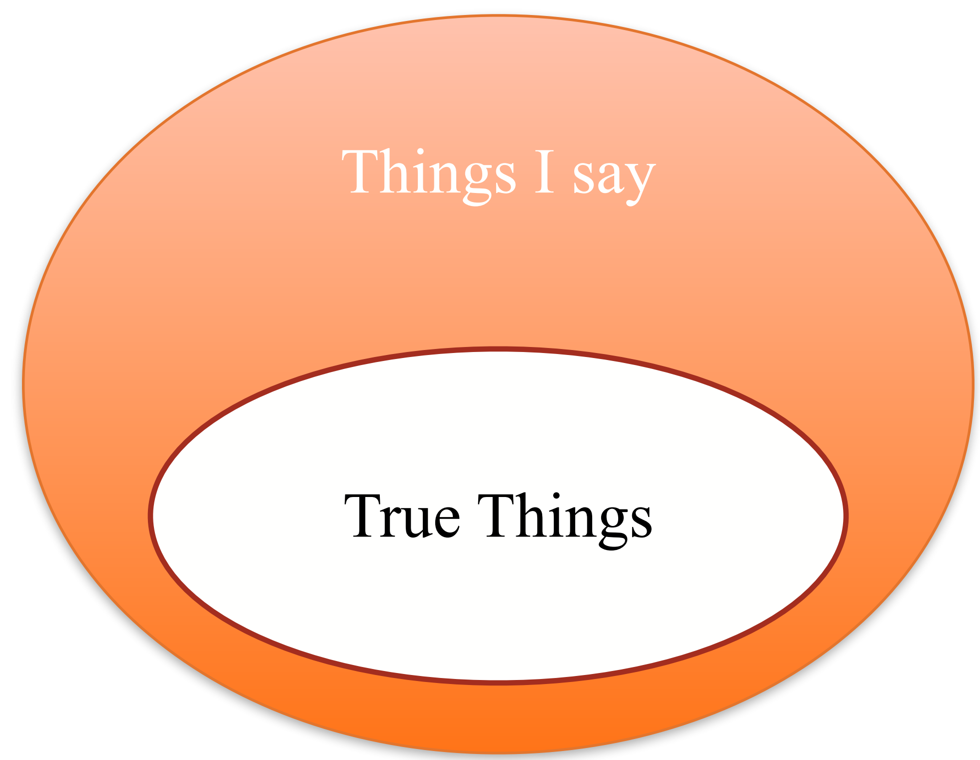
If analysis says X is true, then X is true.



Trivially Sound: Say nothing

Completeness

If X is true, then analysis says X is true.

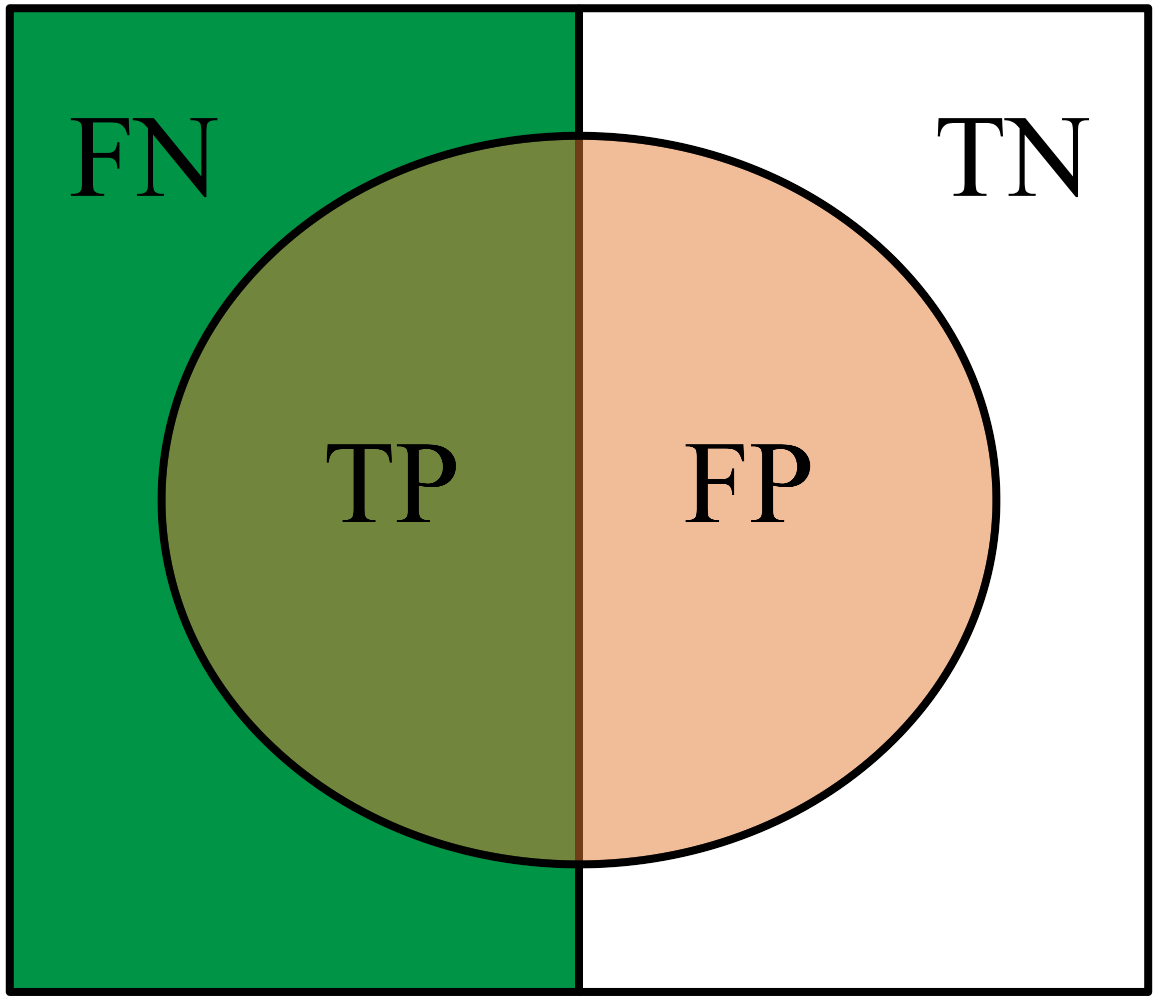


Trivially complete: Say everything

Sound and Complete: Say exactly the set of true things!

Soundness, Completeness, Precision, Recall, False Negative, False Positive, All that Jazz…

Imagine we are building a *classifier*.
Ground truth: things on the left is “in”.
Our classifier: things inside circle is “in”.



Sound means FP is empty
Complete means FN is empty

Precision = TP/(TP+FP)
Recall = TP/(FN+TP)
False Positive Rate = FP/(TP+FP)
False Negative Rate = FN/(FN+TN)
Accuracy = (TP+TN)/(Σ everything)

Context Sensitive

Whether different calling contexts are distinguished

```
void yellow()
{
1. red(1);
2. red(2);
3. green();
}
```

```
void red(int x)
{
..
}
```

```
void green()
{
  green();
  yellow();
}
```

Context sensitive distinguishes 2 different calls to red(-)

Context Sensitive Example

```
a = id(4);

b = id(5);
```

```
void id(int z)
{ return z; }
```

Context-Sensitive
(color denotes matching call/ret)

Context sensitive can tell one call returns 4, the other 5

```
a = id(4);

b = id(5);
```

```
void id(int z)
{ return z; }
```

Context-Insensitive
(note merging)

Context insensitive will say both calls return {4,5}

Flow Sensitive

- A flow sensitive analysis considers the order (flow) of statements
- Examples:
  - Type checking is flow insensitive since a variable has a single type regardless of the order of statements
  - Detecting uninitialized variables requires flow sensitivity

```
x = 4;
....
x = 5;
```

Flow sensitive can distinguish values of x, flow insensitive cannot

Flow Sensitive Example

```
1. x = 4;
....
n. x = 5;
```

Flow sensitive:
x is the constant 4 at line 1, x is the constant 5 at line n

Flow insensitive:
x is not a constant

Path Sensitive

- A path sensitive analysis maintains branch conditions along each execution path
  - Requires extreme care to make scalable
  - Subsumes flow sensitivity

Path Sensitive Example

```
1. if(x >= 0)
2.    y = x;
3. else
4.    y = -x;
```

path sensitive:
y >= 0 at line 2,
y > 0 at line 4

path insensitive:
y is not a constant

Precision

Even path sensitive analysis approximates behavior due to:

- loops/recursion
- unrealizable paths

```
1. if(aⁿ + bⁿ = cⁿ && n>2 && a>0 && b>0 && c>0)
2.     x = 7;
3. else
4.     x = 8;
```

Unrealizable path.
x will always be 8

Control Flow Integrity (Analysis)

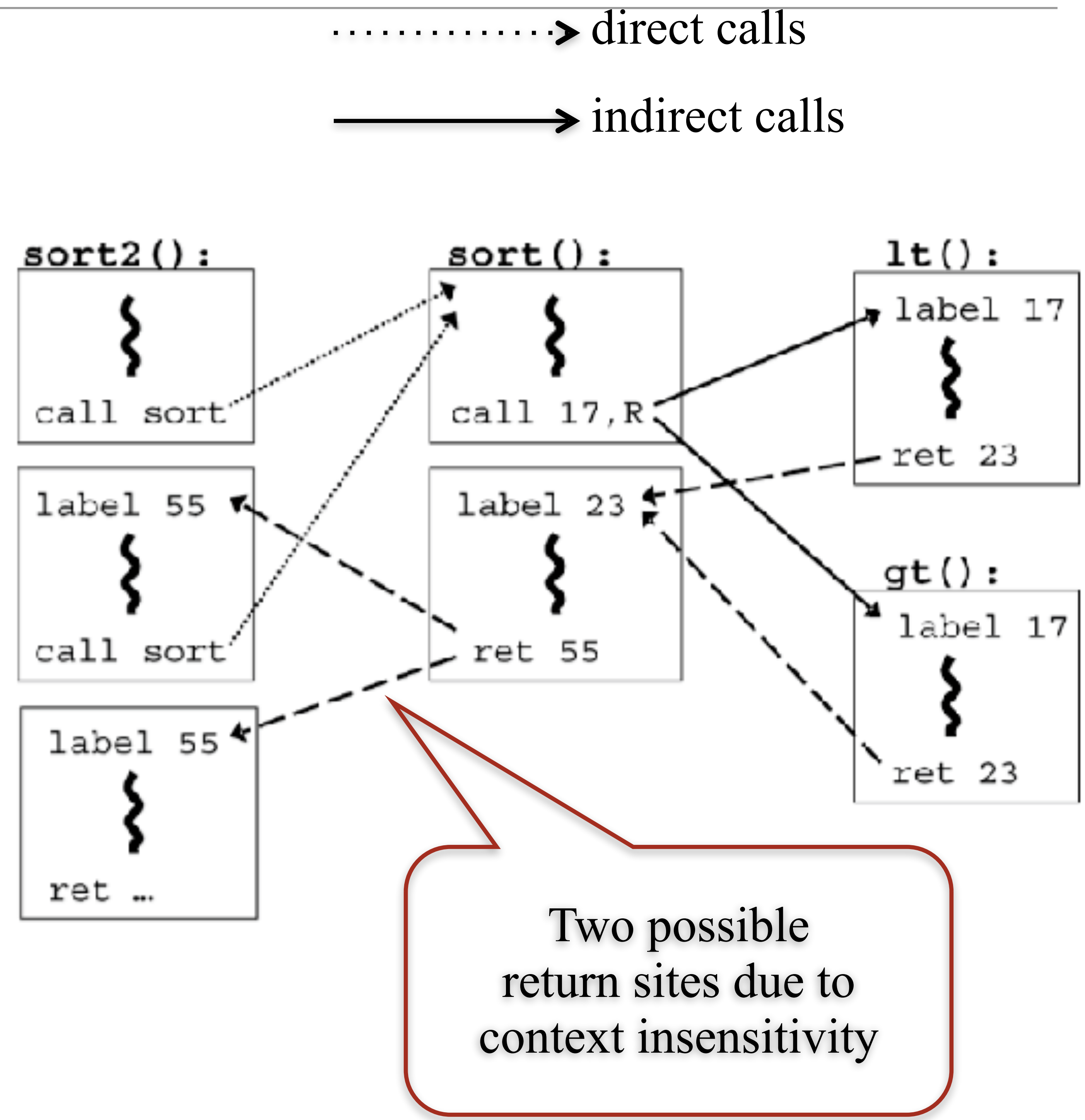
CFI Overview

- Invariant: Execution must follow a path in a control flow graph (CFG) created ahead of run time.

- Method:
  - build CFG statically, e.g., at compile time
  - instrument (rewrite) binary, e.g., at install time
    - add IDs and ID checks; maintain ID uniqueness
  - verify CFI instrumentation at load time
    - direct jump targets, presence of IDs and ID checks, ID uniqueness
  - perform ID checks at run time
    - indirect jumps have matching IDs

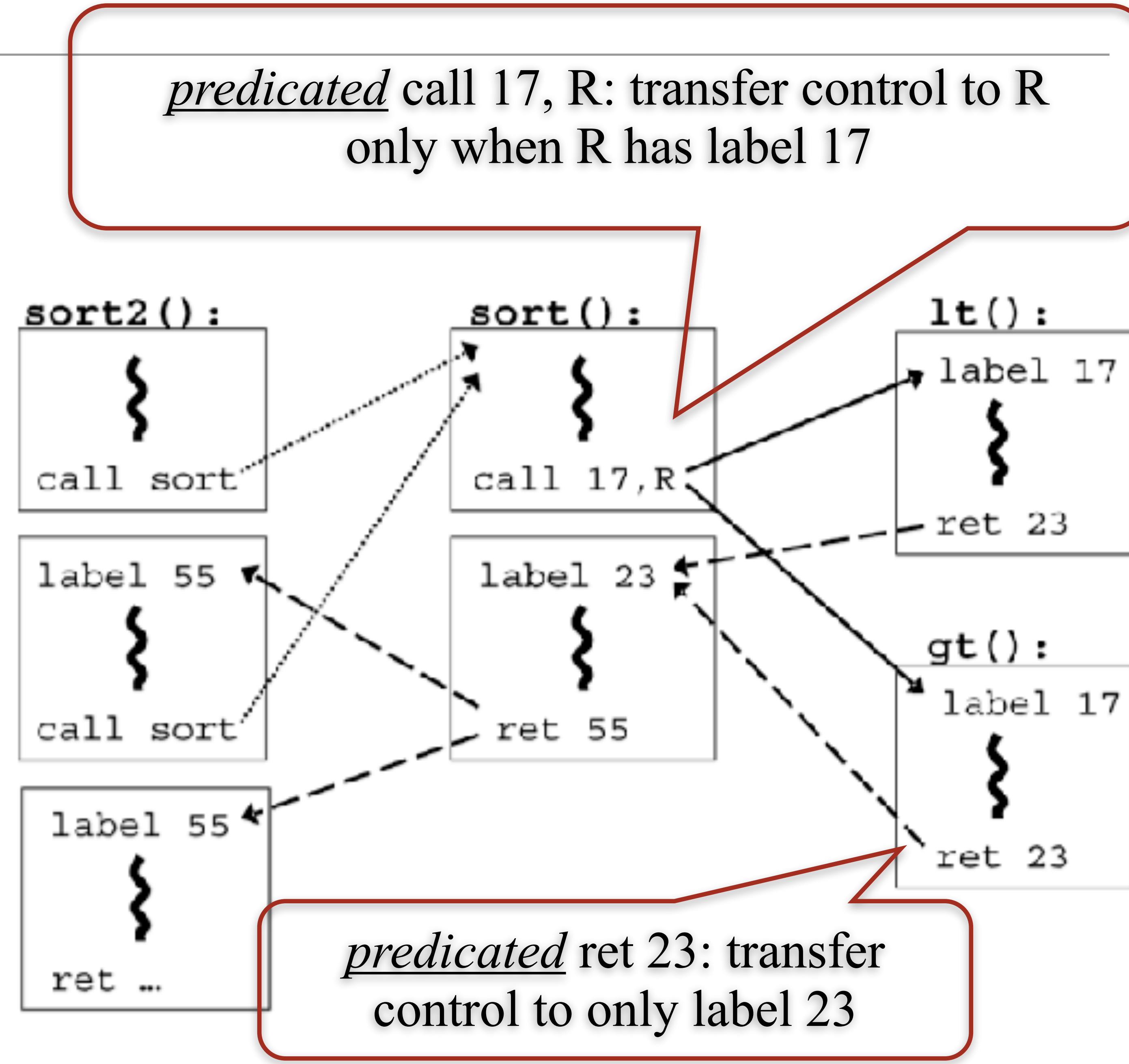
Build CFG

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



Instrument Binary

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



- Insert a unique number at each destination
- Two destinations are equivalent if CFG contains edges to each from the same source

Verify CFI Instrumentation

- Direct jump targets (e.g. call 0x12345678)
  - are all targets valid according to CFG?
- IDs
  - is there an ID right after every entry point?
  - does any ID appear in the binary by accident?
- ID Checks
  - is there a check before every control transfer?
  - does each check respect the CFG?

easy to implement correctly => trustworthy

ID Checks

Check dest label

```
FF 53 08                 call   [ebx+8]          ; call a        ion pointer
```

is instrumented using prefetchnta destination   , to become:

```
8B 43 08                 mov   eax, [ebx+8]       ; load pointer into register
3E 81 78 04 78 56 34 12  cmp   [eax+4], 12345678h ; compare opcodes at destination
75 13                    jne   error_label        ; if not ID value, then fail
FF D0                    call eax                 ; call function pointer
3E 0F 18 05 DD CC BB AA  prefetchnta [AABBCCDDh]  ; label ID, used upon the return
```

Fig. 4. Our CFI implementation of a call through a function pointer.

Check dest label

Bytes (opcodes)	x86 assembly code	Comment
C2 10 00	ret 10h	; return

is instrumented using prefetchnta destination IDs, to   e:

```
8B 0C 24                 mov   ecx, [esp]         ; load    address into register
83 C4 14                 add   esp, 14h           ; p  p 20 bytes off the stack
3E 81 79 04 DD CC BB AA  cmp   [ecx+4], AABBCCDDh ; compare opcodes at destination
75 13                    jne   error_label        ; if not ID value, then fail
FF E1                    jmp   ecx                ; jump to return address
```

Performance

- Size: increase 8% avg
- Time: increase 0-45%; 16% avg

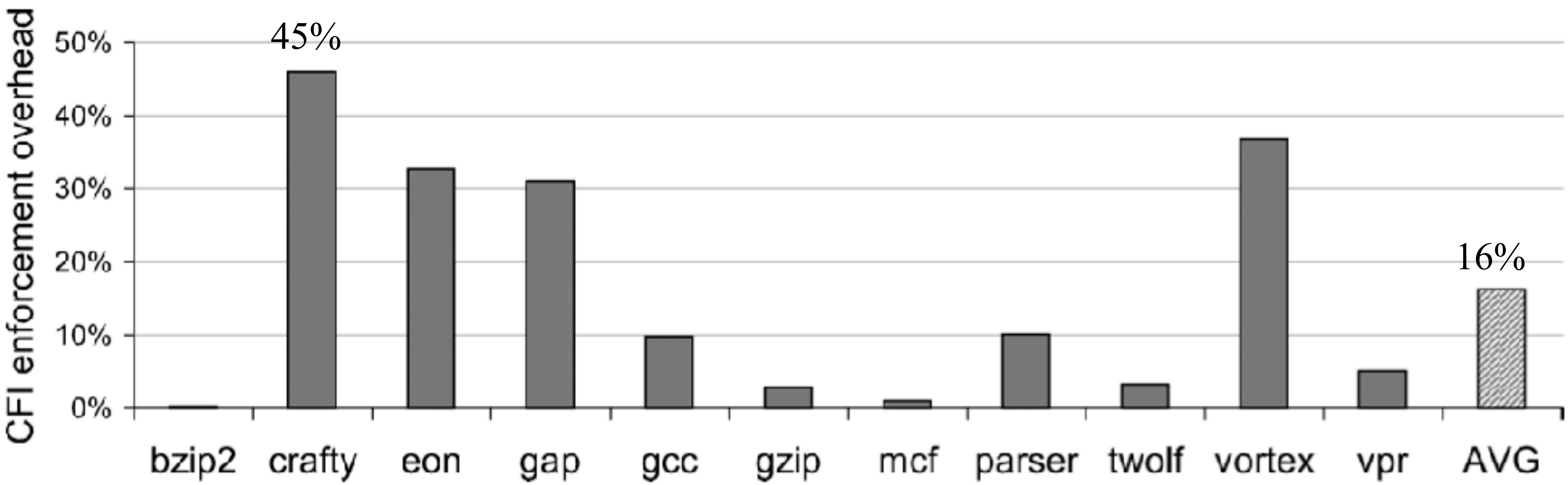


Fig. 6. Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

Security Guarantees

- Effective against attacks based on illegitimate control-flow transfer
  - buffer overflow, ret2libc, pointer subterfuge, etc.

Any check becomes non-circumventable.

- Allow data-only attacks since they respect CFG!
  - incorrect usage (e.g. printf can still dump mem)
  - substitution of data (e.g. replace file names)

Software Fault Isolation

- SFI ensures that a module only accesses memory within its region by adding checks
  - e.g., a plugin can accesses only its own memory

```
if(module_lower < x < module_upper)
    z = load[x];
```

SFI Check

- CFI ensures inserted memory checks are executed

Inline Reference Monitors

- IRMs inline a security policy into binary to ensure security enforcement

- Any IRM can be supported by CFI + Software Memory Access Control
    - CFI: IRM code cannot be circumvented

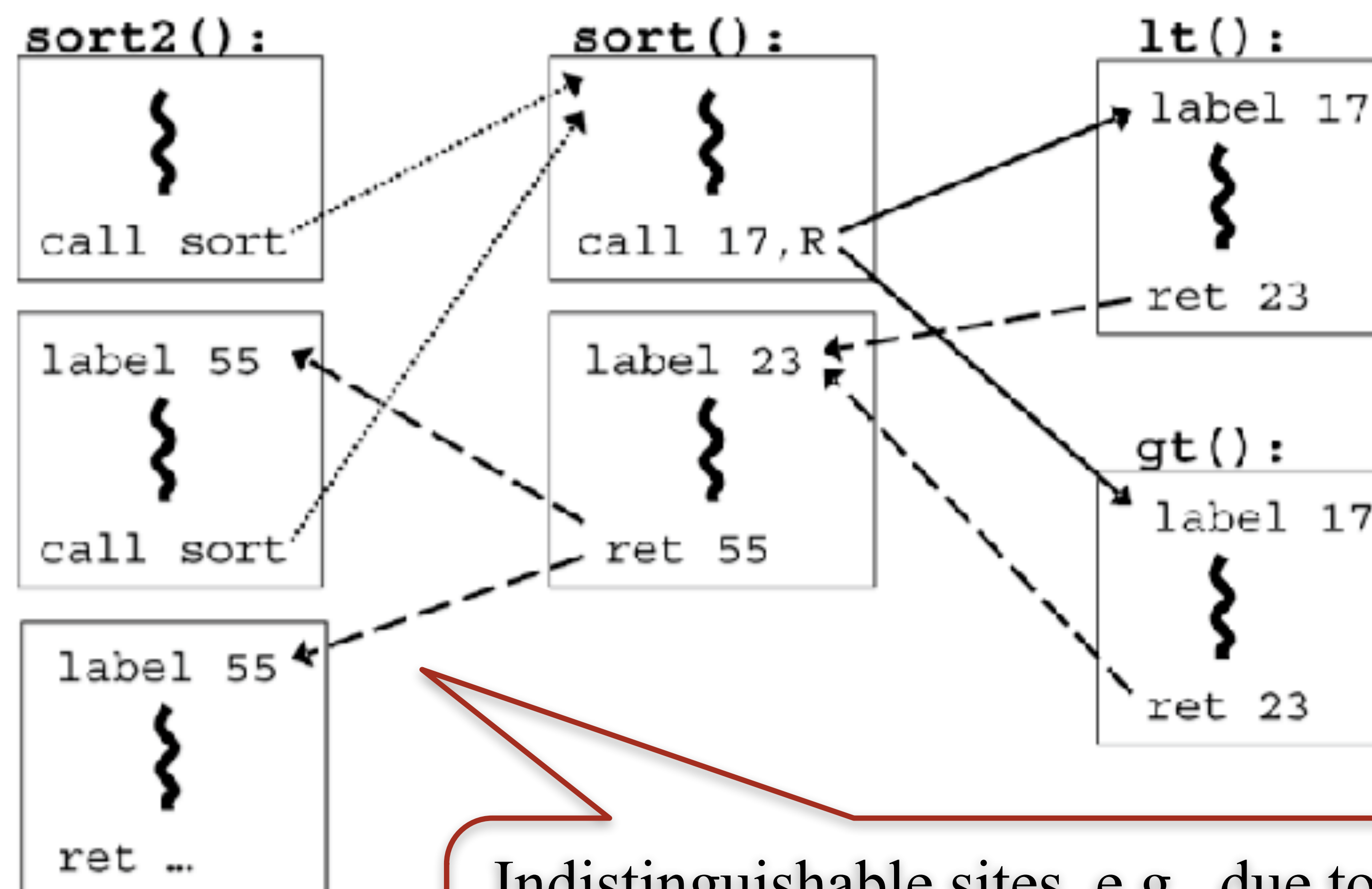
      +

    - SMAC: IRM state cannot be tampered

Accuracy vs. Security

- The accuracy of the CFG will reflect the level of enforcement of the security mechanism.

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



Indistinguishable sites, e.g., due to lack of context sensitivity will be merged

Context Sensitivity Problems

- Suppose A and B both call C.
- CFI uses same return label in A and B.

- How to prevent C from returning to B when it was called from A?
- Shadow Call Stack
  - a protected memory region for call stack
  - each call/ret instrumented to update shadow
  - CFI ensures instrumented checks will be run

CFI Summary

- Control Flow Integrity ensures that control flow follows a path in CFG
  - Accuracy of CFG determines level of enforcement
  - Can build other security policies on top of CFI

Acknowledgments/References (1/2)

- [B. P. Miller'06] A Framework for Binary Code Analysis, and Static and Dynamic Patching, Barton P. Miller, Jeffrey Hollingsworth, February 2006.
- [Papadopoulos’11] CS451, Christos Papadopoulos, CSU, Spring 2011. Original slides by Devendra Salvi (2007). Based on “Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software”, J. Newsome and D. Song, NDSS 2005.
- [EECS 583’12] – Class 21 Research Topic 3: Dynamic Taint Analysis, University of Michigan December 5, 2012. Based on “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”, E. J. Schwartz, T. Avgerinos, D. Brumley, IEEE S&P, 2010.
- [Brumley’10] All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask), E. J. Schwartz, T. Avgerinos, D. Brumley, IEEE S&P, 2010.

Acknowledgments/References (2/2)

- [CS-6V81] System Security and Malicious Code Analysis, S. Qumruzzaman, K. Al-Naami, Spring 2012. Based on "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software", J. Newsome and D. Song, NDSS 2005.
- [Salwan'15] Dynamic Binary Analysis and Instrumentation Covering a function using a DSE approach, J. Salwan, Security Day, January 2015.
- [TaintPipe'15] TaintPipe: Pipelined Symbolic Taint Analysis, Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu, Usenix Security 2015.
- [Brumley'15] Introduction to Computer Security (18487/15487), David Brumley and Vyas Sekar, CMU, Fall 2015.
- [Kuznetsov'14] Code-Pointer Integrity, Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, Dawn Song, Slides from OSDI 2014.
- [Payer'14] Code-Pointer Integrity, Mathias Payer, Slides in (Chaos Communication Congress) CCC 2014.