CS 250: Software Security

Dynamic Taint Analysis

Dynamic Taint Analysis for Automatic Detection, Analysis and Signature Generation of Exploits on Commodity Software

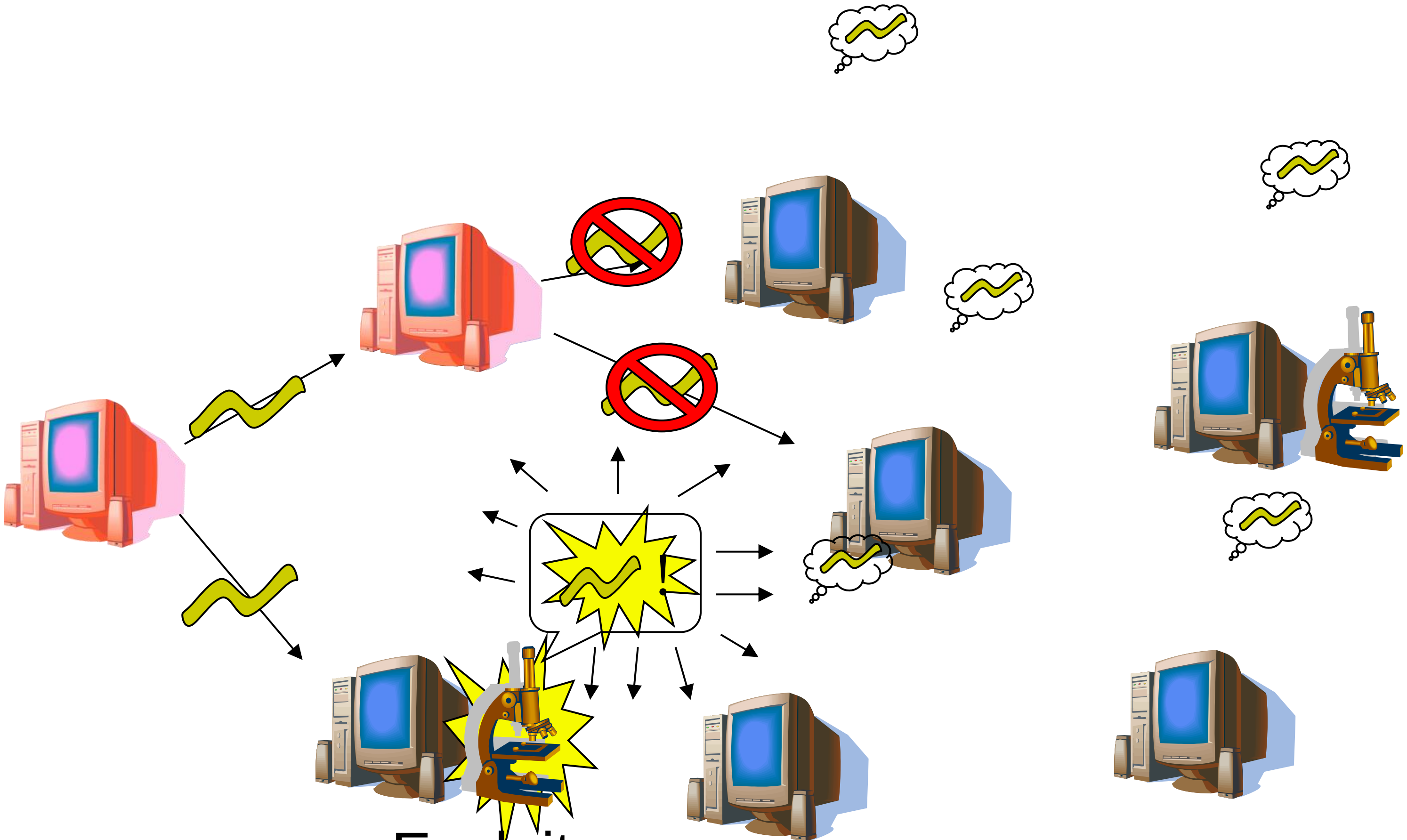
James Newsome and Dawn Song

Appeared in NDSS’06

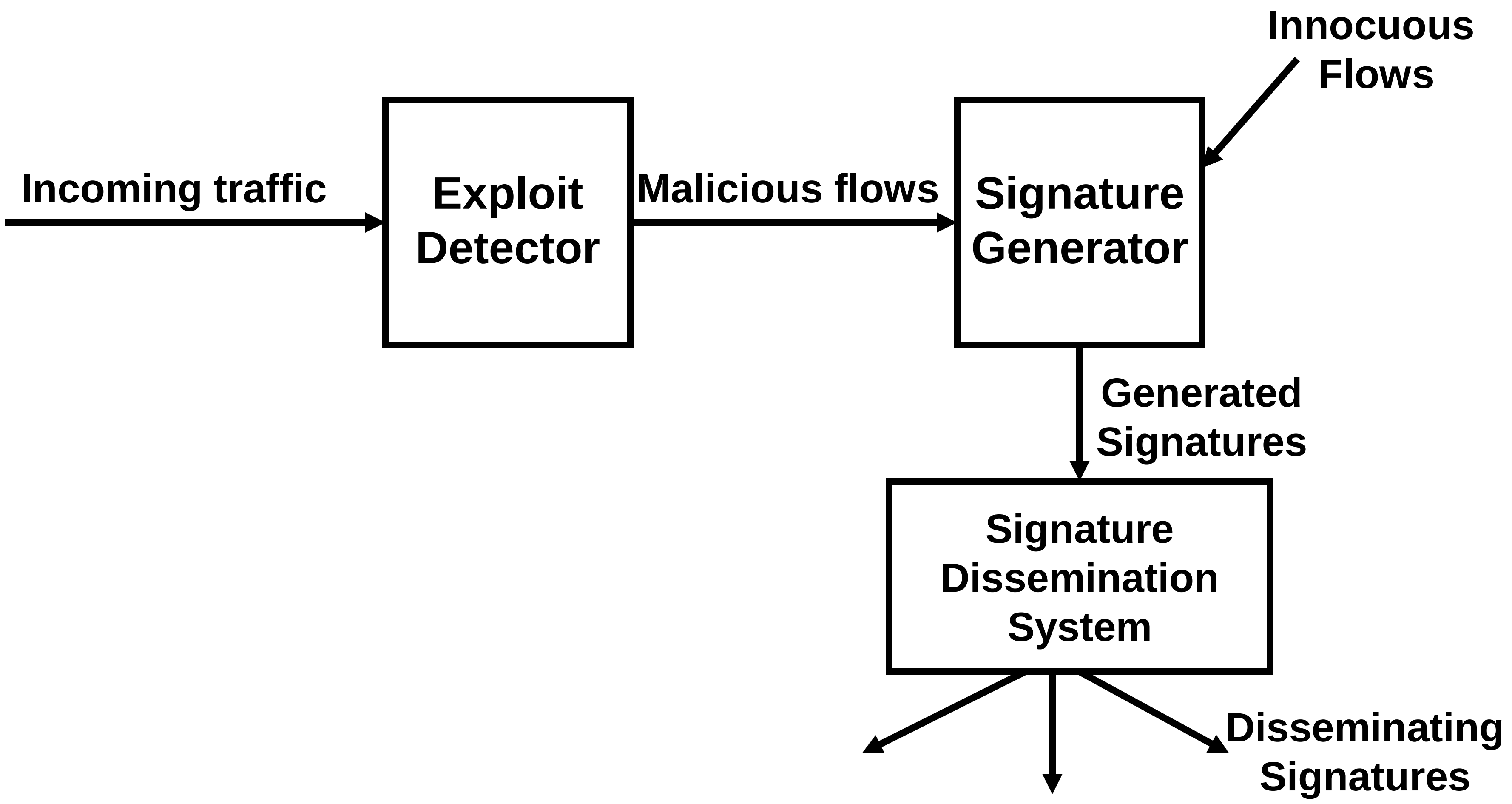
Problem: Internet Worms

- Propagate by exploiting vulnerable software
- No human interaction needed to spread
- Able to rapidly infect vulnerable hosts
  - Slammer scanned 90% of Internet in 10 minutes
- Need **automatic** defense against new worms

Automatic Worm Defense



Architecture



Common Traits of Software Exploits

- Most known exploits are *overwrite attacks*
- Attacker’s data overwrites sensitive data
- Common overwrite vulnerabilities:
  - Buffer overflows
  - Format string
  - Double-free
- Common overwrite targets:
  - Return address
  - Function pointer

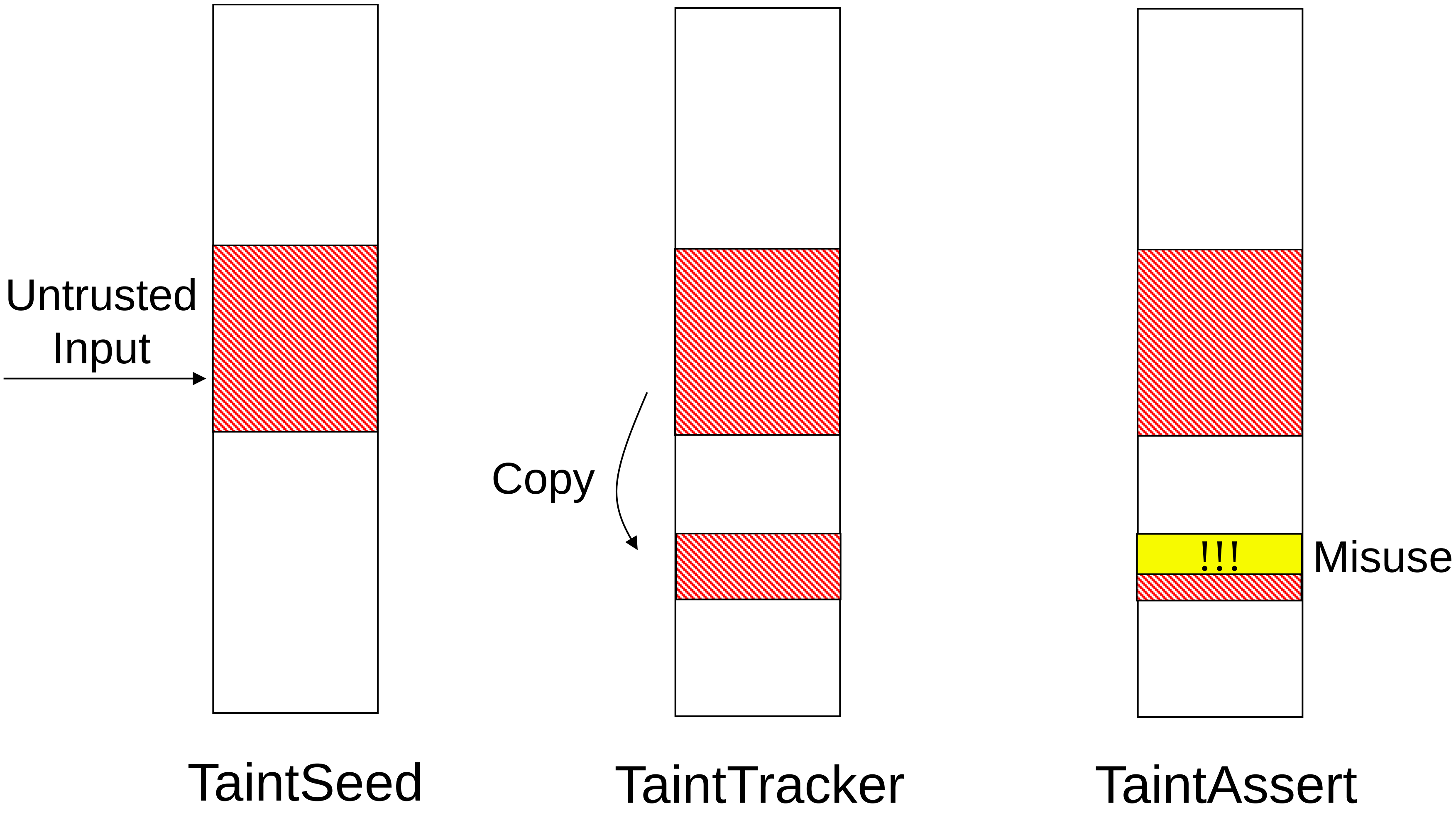
Approach: Dynamic Taint Analysis

- Hard to tell if data is sensitive when it is *written*
  - Binary has no type information
- Easy to tell it is sensitive when it is *used*
- Approach: *Dynamic Taint Analysis*:
  - Keep track of *tainted* data from untrusted sources
  - Detect when tainted data is used in a sensitive way
    - *e.g.*, as return address or function pointer

Design & Implementation: TaintCheck

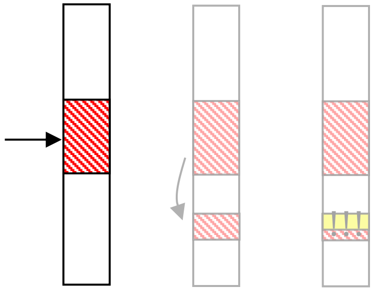
- Use Valgrind to monitor execution
  - Instrument program binary at run-time
  - No source code required
- Track a taint value for each location:
  - Each byte of tainted memory
  - Each register

TaintCheck Components



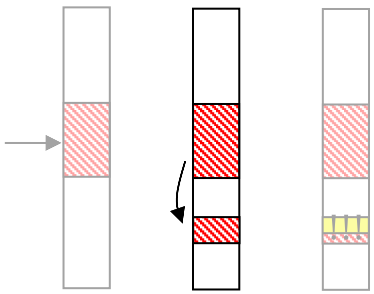
TaintSeed

- Monitors input via system calls
- Marks data from untrusted inputs as tainted
  - Network sockets (default)
  - Standard input
  - File input
    - (except files owned by root, such as system libraries)



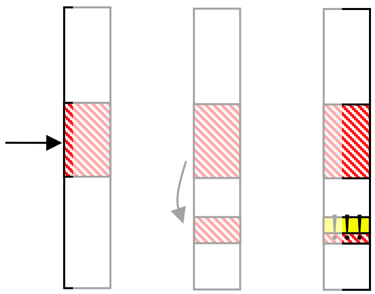
TaintTracker

- Propagates taint
- Data movement instructions:
  - *e.g.*, move, load, store, etc.
  - Destination tainted iff source is tainted
  - Taint data loaded via tainted index
    - *e.g.*, unicode = translation_table[tainted_ascii]
- Arithmetic instructions:
  - *e.g.*, add, xor, mult, etc.
  - Destination tainted iff *any* operand is tainted
- Untaint result of constant functions
    - xor eax, eax



TaintAssert

- Detects when tainted data is misused
  - Destination address for control flow (default)
  - Format string (default)
  - Argument to particular system calls (e.g., `execve`)
- Invoke Exploit Analyzer when exploit detected



Coverage: Attack Classes Detected

	Format String	Stack Overflow	Heap Overflow	Heap Corruption (Double Free)
Return Address	✓	✓	N/A	✓
Function Pointer	✓	✓	✓	✓
Fn Ptr Offset (GOT)	✓	✓	✓	✓
Jump Address	✓	✓	✓	✓

Other Applications

- Information leakage detection/analysis
- Malware analysis
- Fuzzing
- A base for symbolic execution/concolic testing
- …

Pointer Tainting

- mov eax, [ebx + 4]

  When ebx is tainted, shall eax be tainted?

- Often used for table lookup, e.g.,
  - Convert from ascii to Unicode
  - Convert a date from one format to another

- It may cause taint explosion

Over tainting & Under tainting

- xor eax, eax
- sub eax, eax

- Taint granularity is important (bit, byte, word, etc.)
  - Coarser granularity may cause over tainting

Examples of bit-level tainting rules

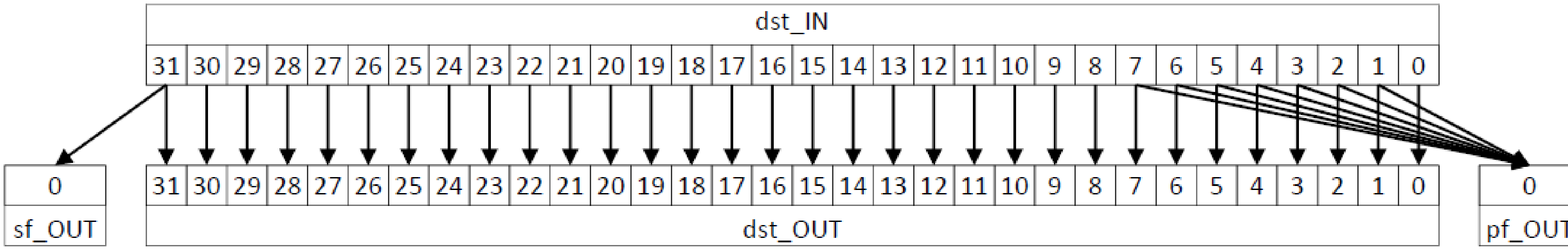


Figure 2: Information flows of *dst* in the `or` instruction

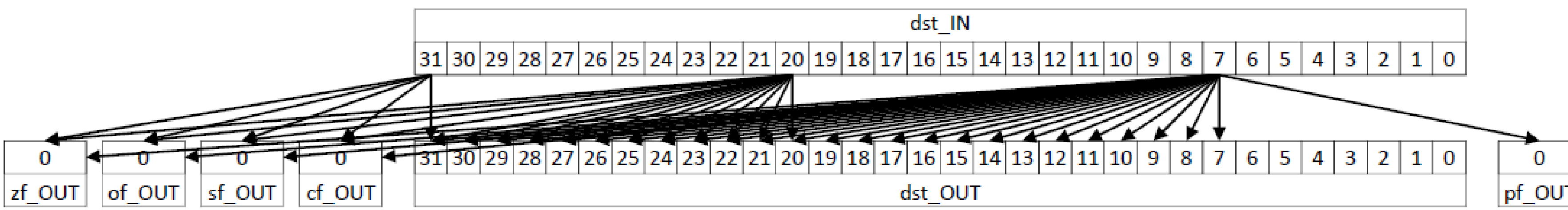


Figure 3: Information flow of bits 7, 20 and 31 of `dst` in `sbb`

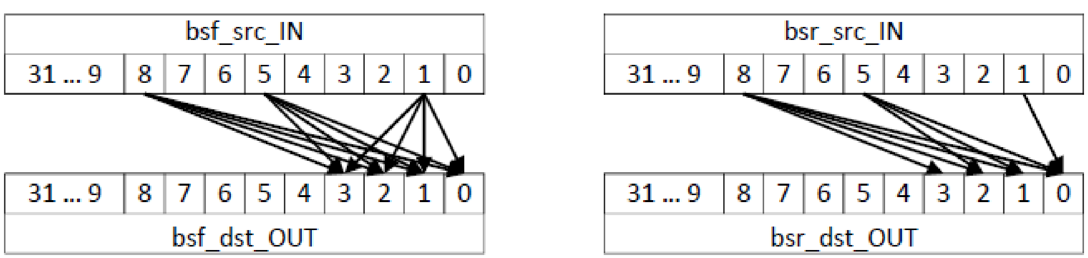


Figure 4: Comparison between *bsf* and *bsr*

Rules for x86 Instructions

Instruction	Inputs	Outputs	# Cases	Runtime	Flow Type	Droid Scope [24]	libdft [32]	Minemu [17]	TEMU [33]	Memcheck [26]	DECAF
adc *dst*, *src*	dst,src,cf	dsr,src,zf,of,sf,af,cf,pf	4,550	1m19s	U	**A**	I	**A**	S	U	S
add *dst*, *src*	dst,src	dst,src,zf,of,sf,af,cf,pf	4,480	1m13s	U	**A**	I	**A**	**A**	S	S
and *dst*, *src*	dst,src	dst,src,zf,sf,pf	4,288	1m05s	I	**A**	I	**A**	I	S	S
dec *dst*	dst	dst,zf,of,sf,af,pf	1,184	20s	U	**A**	I	**A**	**A**	U	S
div *rm*	edx,eax,rm	edx,eax,rm	9,216	95m48s	D	**A**	I	N	**A**	**A**	D
idiv *rm*	edx,eax,rm	edx,eax,rm	9,216	307m04	A	A	I	N	A	A	A
imul1 *rm*	eax,rm	edx,eax,rm,of,cf	6,272	289m51s	U	**A**	I	N	**A**	U	U
imul2 *dst*, *rm*	dst,rm	dst,rm,of,cf	4,224	52m37s	U	**A**	I	N	**A**	U	U
imul3 *dst*, *rm*, *imm*	rm,imm	dst,rm,imm,of,cf	6,272	53m56s	U	**A**	I	N	**A**	U	U
inc *dst*	dst	dst,zf,of,sf,af,pf	1,184	19s	U	**A**	I	**A**	**A**	U	S
mul *rm*	eax,rm	edx,eax,rm,of,cf	6,272	16m02s	U	**A**	I	N	**A**	U	U
not *dst*	dst	dst	1,024	15s	I	**A**	I	**A**	I	I	I
or *dst*, *src*	dst,src	dst,src,zf,sf,pf	4,288	1m05s	I	**A**	I	**A**	I	S	S
rcl *dst*, *imm8*	dst,imm8,cf	dst,imm8,of,cf	1,722	42s	A	A	N	A	A	A	S
rcr *dst*, *imm8*	dst,imm8,cf	dst,imm8,of,cf	1,722	42s	A	A	N	A	A	A	S
rol *dst*, *imm8*	dst,imm8	dst,imm8,of,cf	1,680	41s	A	A	N	A	A	S	S
ror *dst*, *imm8*	dst,imm8	dst,imm8,of,cf	1,680	41s	A	A	N	A	A	S	S
sal *dst*, *imm8*	dst,imm8	dst,imm8,zf,of,sf,af,cf,pf	1,840	35s	U	**A**	N	**A**	S	S	S
sar *dst*, *imm8*	dst,imm8	dst,imm8,zf,of,sf,af,cf,pf	1,840	34s	D	**A**	N	**A**	S	S	S
sbb *dst*, *src*	dst,src,cf	dst,src,zf,of,sf,af,cf,pf	4,550	1m21s	U	**A**	I*	**A***	**A**	**A**	S
shr *dst*, *imm8*	dst,imm8	dst,imm8,zf,of,sf,af,cf,pf	1,840	35s	D	**A**	N	**A**	S	S	S
sub *dst*, *src*	dst,src	dst,src,zf,of,sf,af,cf,pf	4,480	1m17s	U	**A**	I*	**A***	**A***	S	S
xor *dst*, *src*	dst,src	dsr,src,zf,sf,pf	4,288	1m05s	I	**A**	I*	**A***	**A***	I	I
bsf *dst*, *src*	src	dst,src,zf	2,080	31s	A	N	I	N	A	A	S
bsr *dst*, *src*	src	dst,src,zf	2,080	31s	S	N	I	N	**A**	**A**	S
cmpxchg *rm*, *r*	eax,rm,r	eax,rm,r,zf,of,sf,af,cf,pf	9,792	2m39s	S	N	**E**	N	**E**	**E**	S
TOTAL			102,064	13h52m48s							

Flow Types: (U)p, (D)own, (I)n-place, (A)ll-around, (S)pecial, (N)ot-Supported, (S)pecial, (E)ax is tainted in `cmpxchg`*, *—Zeroing Idiom,* ***Boldface****—Generated Policy is more precise.*

New Formally-verified Precise Rules

TABLE 4
New Precise Bit-Level Taint Rules: `rcr` and `bsr` Are Similar to `rcl` and `bsf` Respectively, and So Omitted

Operation	Rule (C-like pseudocode)

adc	`x1_min = x1 & ~t1; x2_min = x2 & ~t2; cf_min = cf & ~tcf;`
	`x1_max = x1 | t1; x2_max = x2 | t2; cf_max = cf | tcf;`
	`t1 | t2 | ((x1_min + x2_min + cf_min) ^ (x1_max + x2_max + cf_max))`

sbb	`t1 | t2 | ((x1_min - (x2_min + cf_min)) ^ (x1_max - (x2_max + cf_max)))`
rcl	`pcast(v) { v == 0 ? 0 : -1 /* all ones */ }` `pcast(t2) | rcl(t1, x2, tcf)`

bsf	`xc = x1_max & ~((x1_min << 1) | -(x1_min << 1));`
	`((xc & 0x5555) && (xc & 0xaaaa) ? 1 : 0) |`
	`((xc & 0x3333) && (xc & 0xcccc) ? 2 : 0) |`
	`((xc & 0x0f0f) && (xc & 0xf0f0) ? 4 : 0) |`
	`((xc & 0x00ff) && (xc & 0xff00) ? 8 : 0);`

The `bsf` rule is shown for a 16-bit value which must be non-zero, and the rule for `rcl` is precise only when the rotate amount is untainted. `x1`, `x2`, and `cf` (carry flag) are the operands while `t1`, `t2`, and `tcf` are the respective shadow taints.

Bit-Precision Tainting in DECAF

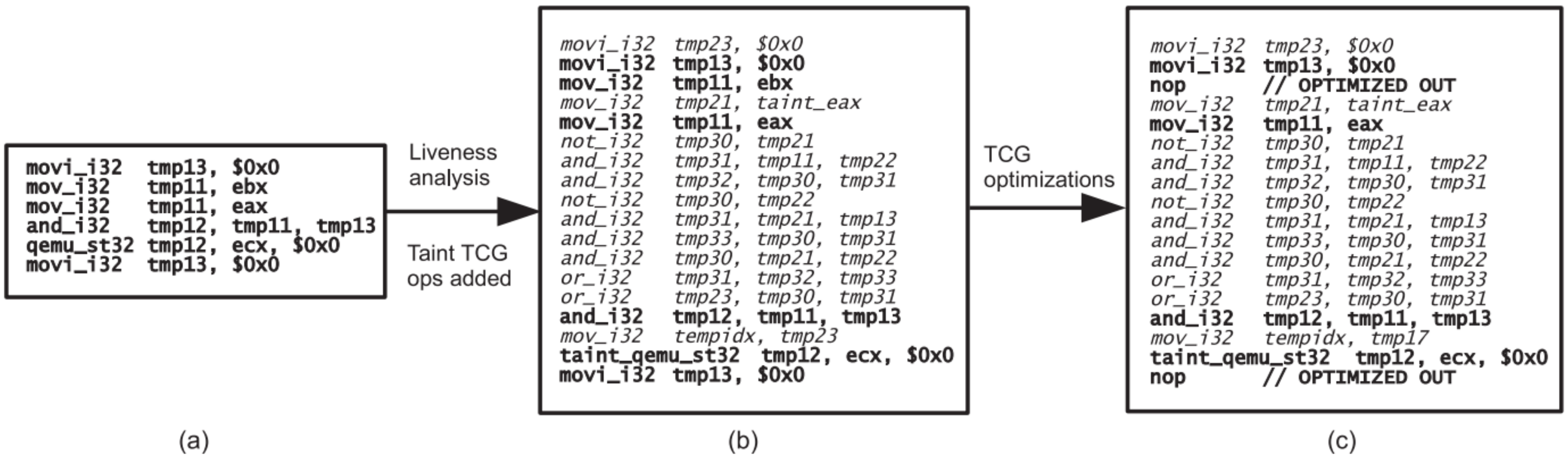


Fig. 5. Register liveness tests determine which TCG instructions in the TB (a) should be instrumented for taint propagation, and instrumentation is inserted as needed (b). TCG's optimization logic eliminates unnecessary opcodes, resulting in an optimized, instrumented TB (c).

Comparing DECAF with TEMU on Tainted Shell Commands

TABLE 8
Comparing DECAF with TEMU on Tainted Shell Commands

Windows		
Command	**DECAF**	**TEMU**
`dir`	207 / 0	639 / 0
`cd`	146 / 0	616 / 0
`cipher c:`	929 / 0	3,617 / 0
`echo hello`	660 / 0	3,808 / 0
`find "jone" a.txt`	967 / 0	5,684 / 0
`findstr /s /i jone ./*`	945 / 0	1,333 / 0
Linux		
Command	**DECAF**	**TEMU**
`ls`	350 / 3	34,923 / 0
`cd`	306 / 3	301 / 0
`cat ./readme`	545 / 31	26,619 / 0
`echo hello`	744 / 9	704 / 0
`ln -s a.txt nbench`	1,122 / 35	24,707 / 0
`mkdir test`	551 / 9	23,766 / 0

"n / m" indicates that "n" bytes are tainted, and "m" tainted EIPs are observed.