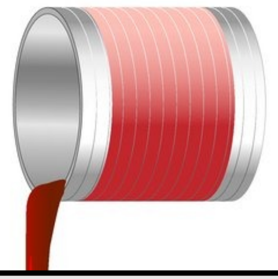
One Engine to Serve'em All: Inferring Taint Rules Without Architectural Semantics

Zheng Leong Chua, Yanhao Wang, Teodora Băluță,
Prateek Saxena, Zhenkai Liang, Purui Su

National University of Singapore

Chinese Academy of Sciences

Importance of Taint Analysis



- Taint analysis tracks the information flow within a program
- Taint analysis is the basis for many security applications
  - Information leakage detection
  - Enforcing CFI
  - Vulnerability detection
  - …

```
int parse_buffer(char buffer[100], struct
pkt_info *info) {
    char check_flag;

     check_flag = buffer[5] & 0x16;

     err = init_pkt_info(info);
     if (!err)
         return err;
     info->flag = check_flag;
    /* … */
    strncpy(info->data, buffer + 6, 50);
    info->seq = get_current_seq();
    return OK;
}
```

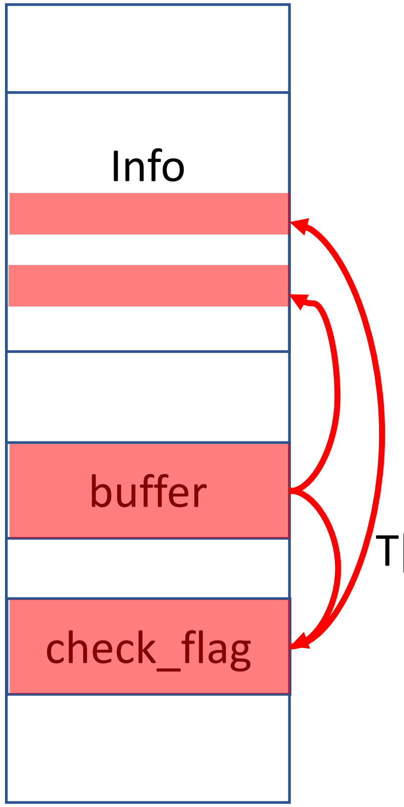
Taint Analysis on Binaries

```c
/* tainted input from network socket */
int parse_buffer(char buffer[100], struct
pkt_info *info) {
    char check_flag;

    check_flag = buffer[5] & 0x16;

    err = init_pkt_info(info);
    if (!err)
        return err;
    info->flag = check_flag;
    /* … */
    strncpy(info->data, buffer + 6, 50);
    info->seq = get_current_seq();
    return OK;
}
```

Taint Map
T[ ]



T[check_flag] = T[buffer+5]

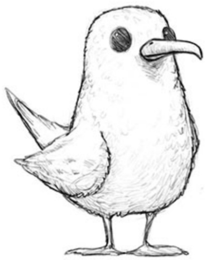
Write binary taint rules based on instruction operational semantics

```asm
movsx   eax, byte ptr [rsi + 5]
and     eax, 16
mov     cl, al
mov     byte ptr [rbp - 25], cl
```

Many Faces of Taint Rules

- What is the taint rule for `and eax, 16`?
  - Main instruction semantics: eax = eax & 16

Taint Engine 1	Taint Engine 2	Taint Engine 3
T[eax] = T[eax]	T[eax] = T[eax] T[pf] = T[sf] = T[zf] = T[eax] T[of] = T[cf] = 0	T[eax] = T[eax] T[pf] = T[sf] = T[zf] = T[eax] T[of] = T[cf] = T[eax] if imm == 0 { T[eax] = 0 }



Complexity of Taint Rules

- Input dependent propagation
- Size dependent propagation
- Architectural quirks for backwards compatibility

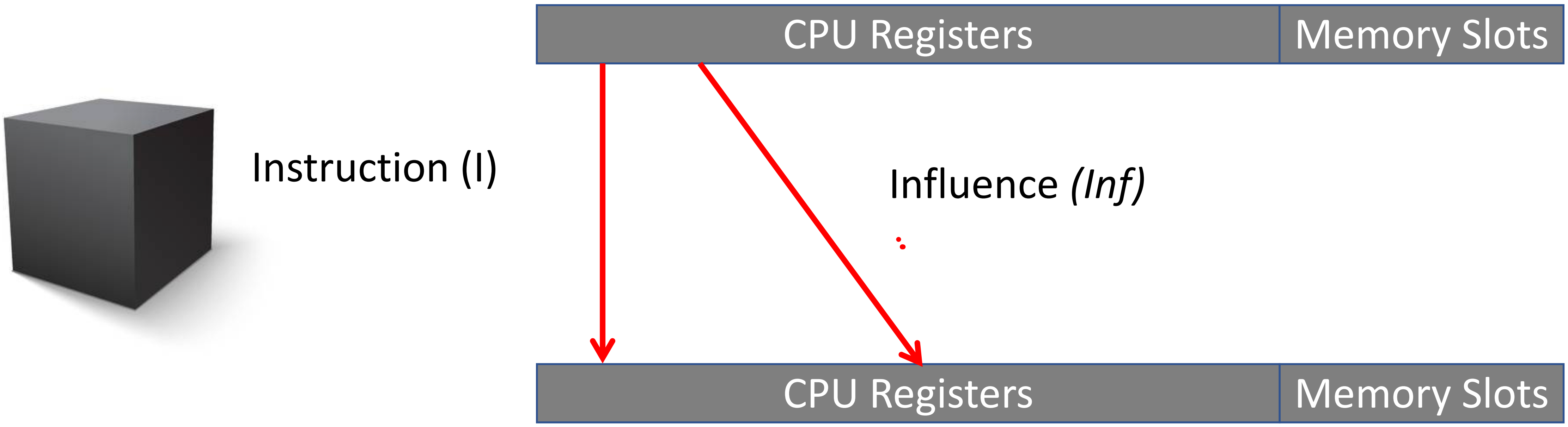
```
if (size == 64 || size == 32 || size == 16) {
  for (x = 0; x < size / 8; x++) {
    if (t1[x] & t2[x]) t1[x] = 1;
    else if (t1[x] and !t2[x])
      t1[x] = t1[x] & op2[x];
    else if (!t1[x] & t2[x])
      t1[x] = t2[x] & op1[x];
    else t1[x] = 0;
  } else if (size == 8) {
// 0 if it's lower 8 bits, 1 if it's upper 8 bits
    pos1 = isUpper(op1); pos2 = isUpper(op2);
    if (t1[pos1] & t2[pos2]) t1[pos1] = 1;
    else if (t1[pos1] & !t2[pos2])
      t1[pos1] = t1[pos1] & op2[pos2];
    else if (!t1[pos1] & t2[pos2])
      t1[pos1] = t2[pos2] & op1[pos1];
    else t1[pos1] = 0;}}
if (mode64bit == 1 and size == 64)
  for (x = 32; x < size; x++) t1[x] = 0;
```

Contributions

- A new way for representing **taint using influence**
  - Rather than instruction semantics
- An **inductive taint analysis** approach using *probe-and-observe*
  - With minimal architectural knowledge
- Our **tool**, TaintInduce, generates accurate taint rules for four architectures (x86, x64, AArch64, MIPS)

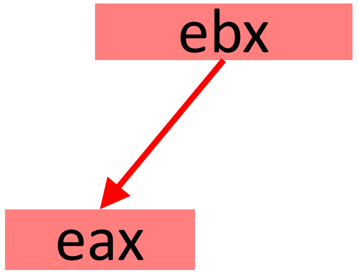
Problem (re-)definition

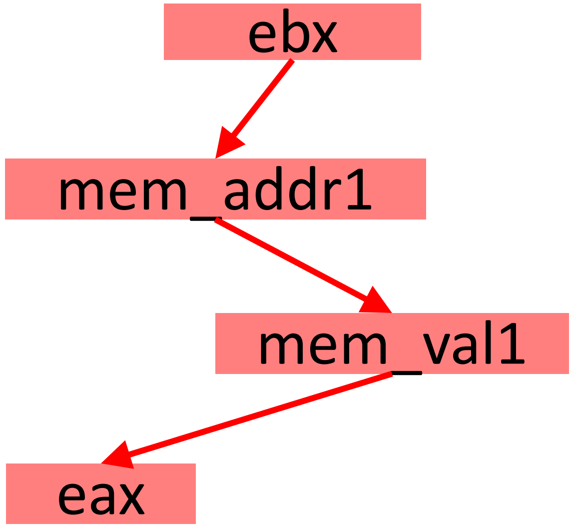
- Taint is defined as a collection of *influence* relations which are observed when executing the instruction as a black box

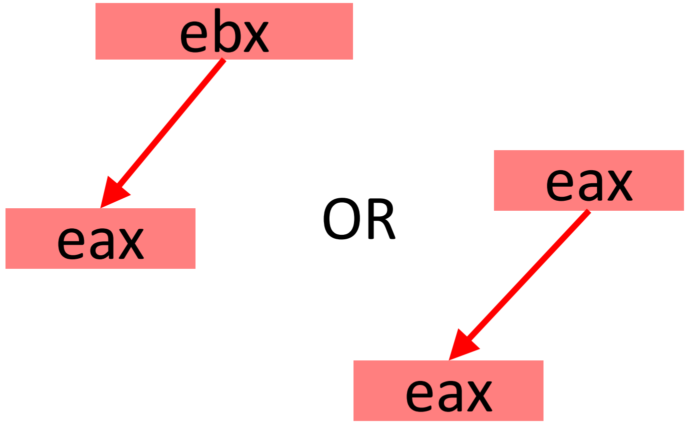


Direct-Indirect Dependencies Using Influence

Direct dependency	Indirect dependency	Implicit dependency
• Same influence relation across all executions	• Multiple direct dependencies	• Influence relation changes across executions
Example: `mov eax, ebx`	Example: `mov eax, [ebx]`	Example: `cmovb eax, ebx`



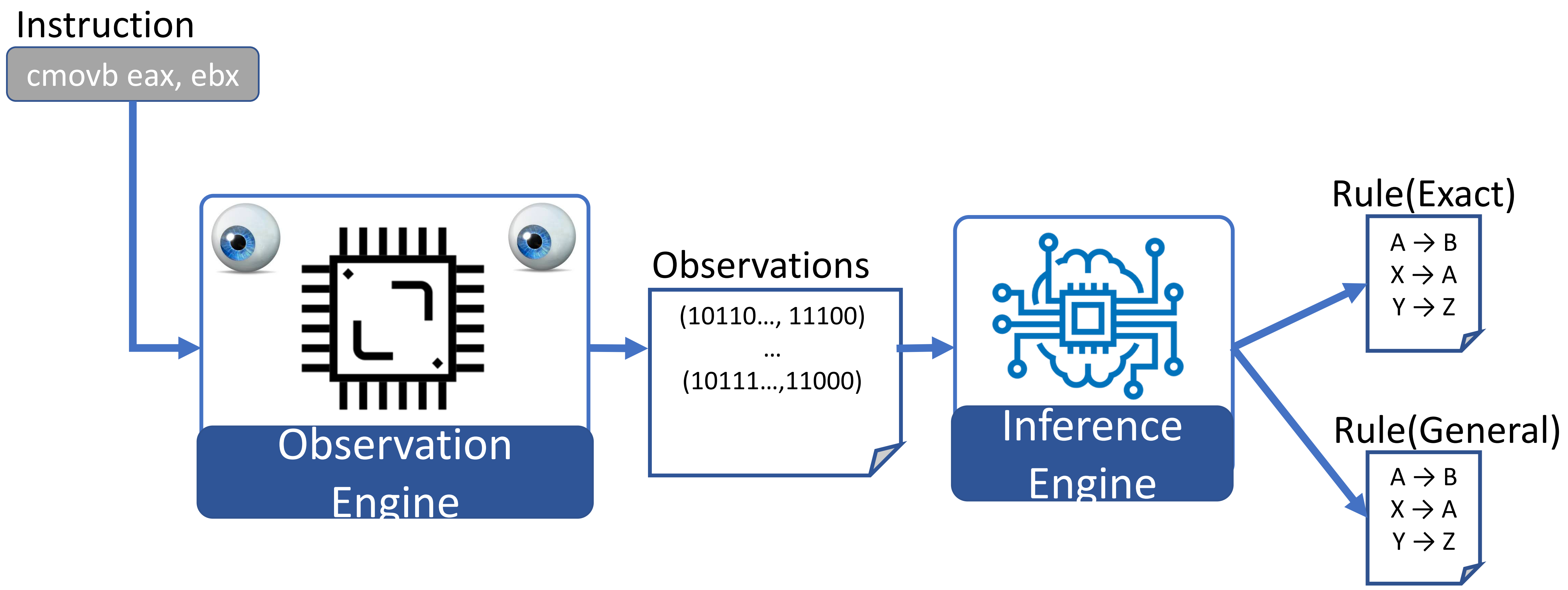




Soundness & Completeness

- No over-tainting: soundness
- No under-tainting: completeness
- Very hard to ensure sound and complete
  - Relax the requirements, aim to be useful in practice ☺

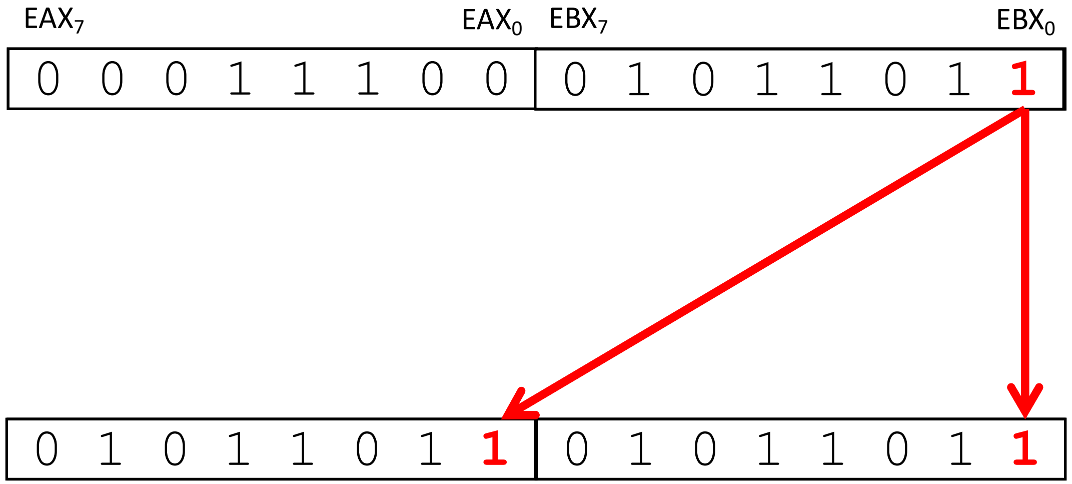
Approach


Instruction
cmovb eax, ebx
Observation Engine
Observations
(10110..., 11100)
...
(10111...,11000)
Inference Engine
Rule(Exact)
A → B
X → A
Y → Z
Rule(General)
A → B
X → A
Y → Z

TaintInduce – Exact Mode

- Flip a bit and observe the output for changes.
  - $\Delta EBX_0 \rightarrow \Delta\ EAX_0$
  - $\Delta EBX_0 \rightarrow \Delta\ EBX_0$
- Influence (Inf) only valid if :
  - EAX = 11100011, EBX = 00101000
- Form a truth table with all of the collected observations.
  - True if there is a change, False otherwise
- Unseen values are conservatively set to False

mov eax, ebx



EAX_0	EAX_1	…	EBX_0	EBX_1	…	Inf
1	1	…	0	0	…	1
1	1	…	1	0	…	1
0	0	…	1	1	…	1
0	0	…	0	0	…	1
…	…	…	…	…	…	0

TaintInduce – Boolean Minimization

- Boolean minimization using ESPRESSO algorithm
- More succinct representation
  - Not a conjunction of all the observed states

$EAX_0 \wedge EAX_1 \wedge \ldots$	True
$EAX_0 \wedge EAX_1 \wedge \ldots$	True
$!EAX_0 \wedge !EAX_1 \wedge \ldots$	True
$!EAX_0 \wedge !EAX_1 \wedge \ldots$	True
<other observations>	True
<unobserved values>	False

IF

$EAX_0 \wedge EAX_1 \wedge \ldots$	True
$!EAX_0 \wedge !EAX_1$	True
…	True

THEN $(EBX_0 \rightarrow EAX_0)$

TaintInduce – Generalization Mode

- We carefully trade-off soundness for generalization
  - We allow the Boolean minimization algorithm to pick values for the unseen inputs by setting them to don't care

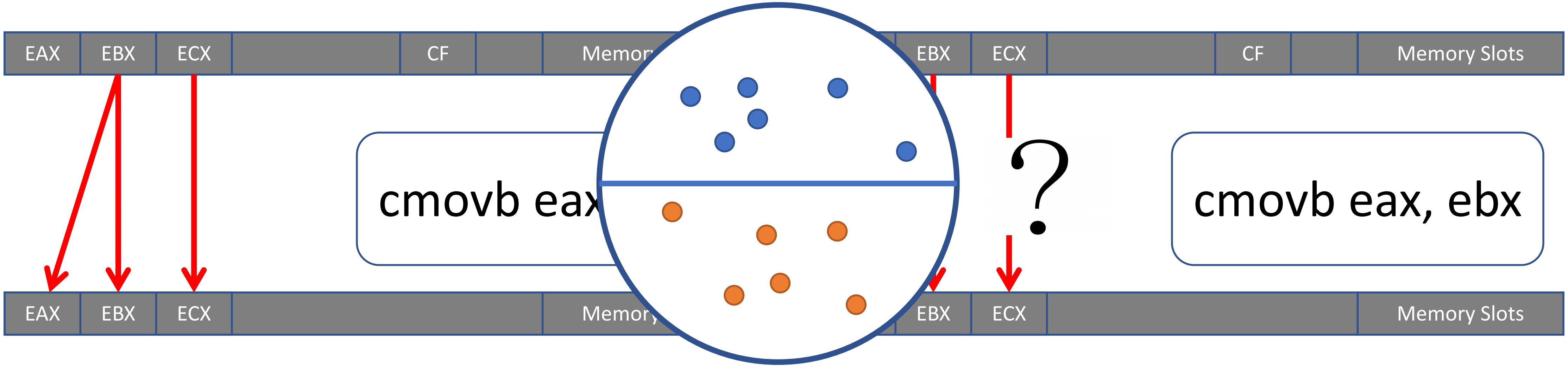
EAX_0 ^ EAX_1 ^ …	True
EAX_0 ^ EAX_1 ^ …	True
$!EAX_0$ ^ $!EAX_1$ ^ …	True
$!EAX_0$ ^ $!EAX_1$ ^ …	True
…	Don't Care

IF

Don't Care	True

THEN ($EBX_0 \rightarrow EAX_0$)

Condition Identification – Behavior Grouping



ebx → eax	eax → eax
CF=1, EAX=542, EBX=19, ECX=7, …	CF=0, EAX=12, EBX=4, ECX=1023…
CF=1, EAX=32, EBX=3, ECX=0, …	CF=0, EAX=42, EBX=11, ECX=13, …
CF=1, EAX=873, EBX=32, ECX=1, …	CF=0, EAX=2, EBX=3, ECX=33, …
…	…

Condition Inference – Generalized

CF=0, EAX=12, …Z	False
CF=1, EAX=333, …	True
CF=0, EAX=42, …	False
CF=0, EAX=44, …	False
CF=1, EAX=873, …	True
CF=0, EAX=1023, …	False
CF=0, EAX=33, …	False
CF=1, EAX=32, …	True
CF=0, EAX=2, …	False
…	DC

Boolean Minimization

IF

CF=1	True

THEN ($EBX_0 \rightarrow EAX_0$)

ELSE ($EAX_0 \rightarrow EAX_0$)

Evaluation

- Coverage and Correctness
  - How many instructions across multiple architectures can TaintInduce learn?
- Exploit Detection for real-world CVEs
  - Is the approach feasible in practice?
- Comparison with other tools
  - Is TaintInduce comparable to existing taint engines?

Coverage and Correctness

TaintInduce never over-taints for 71.51% of the instructions tested across 4 architectures: x86, x64, AArch 64, MIPS-I

Methodology: train for 100 seeds, test on 1000 random inputs for each instruction

	Arith	Comp	Jump	Move	Cond	FPU	SIMD	Misc
x86	√	√	√	√	√	√	√	√
x64	√	√	√	√	√	√	√	√
AArch64	√	√	√	√	√	√	√	√
MIPS-I	√	√	√	√	-	-	-	-

Exploit Detection for real-world CVEs

Detected taint at the sink in 24 / 26 of the exploit trace. Of the remaining 2, sink value is derived indirectly from the source.

- 26 CVEs from real-world programs
  - bind, sendmail, wu-ftpd, rpcss, mssql, atphttpd, ntpd, smbd, ghttpd, miniupnp, openjpeg, glibc, libsndfile, gnulib
  - Stack buffer overflows, heap corruption, floating-point division errors, integer divide-by-zero

- Track direct dependencies only similar to other approaches

Comparison with other Tools

Learns rules that propagate identically to existing tools between 93.27% and 99.5%.

X86 Instructionsxw	Arith	Comp	Jump	Move	Cond	FPU	SIMD	Misc	Total
TaintInduce	43	9	33	33	60	85	259	28	550
libdft	15	5	1	30	32	X	X	8	91
Triton	38	9	19	33	32	X	144	13	288
TEMU	7	1	2	3	X	X	X	X	13

Take Aways

- Re-define taint based on observations – propose an inductive approach with minimal architectural knowledge

- Reduces engineering effort and improves usability of taint

- TaintInduce works well in practice, comparable to existing manual tools

Backup Slides

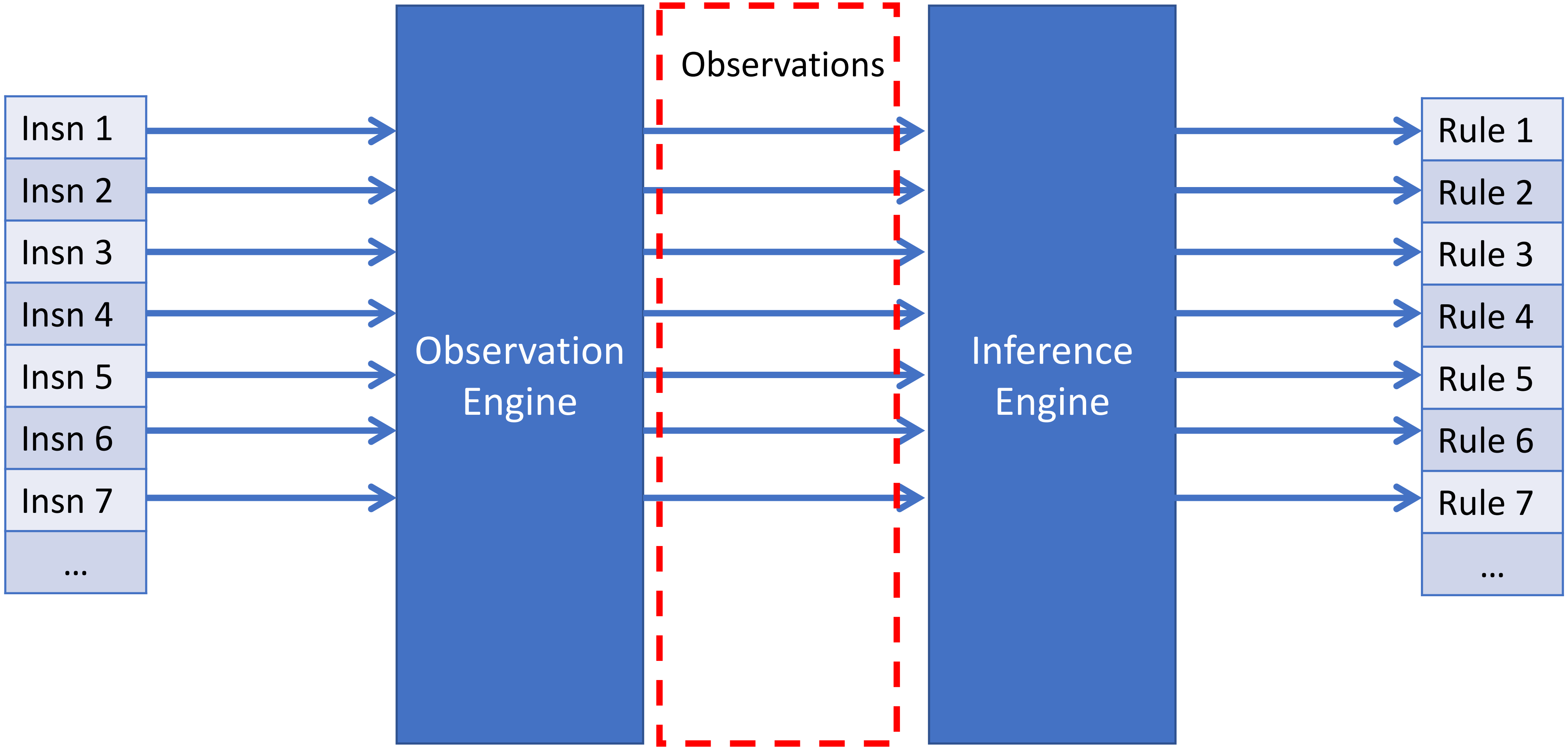
Performance

- 24 hrs for 27 traces using 20 servers.
  - 23 hours for rule inference, 30 mins for taint propagation
- Rule inference time scales linearly with the amount of compute power.

Utility as a cross-referencing tool

- Found 20 bugs in existing taint tools, 17 errors in unicorn, 3 description errors in ISA instruction manuals
- Intel Software Developer's Manual (bt r16/32, r16/32)
  - Manual states 3 or 5 bits, should be 4 or 5.
- Ambiguous behavior for tzcnt
  - If not support, silently fallback to bsf

Tool Implementation



Soundness & Completeness

- No over-tainting: $R_I(S,T)[j] \Longrightarrow \exists i, S \mid T[i] \wedge (< I, S, i, j > \in Inf)$
- No under-tainting: $\exists i, S \mid T[i] \wedge (< I, S, i, j > \in Inf) \Longrightarrow R_I(S,T)[j]$
- Very hard to ensure sound and complete
  - Relax the requirements, aim to be useful in practice ☺

Inference Engine

- Exact mode – Sound & Complete w.r.t to seen states

Complexity of Creating Taint Rules

Input dependent propagation

Size dependent propagation

Architectural quirks for backwards compatibility

What if we don’t have instruction manuals at all?

Taint rule for `and eax, 16`?

```
if (size == 64 || size == 32 || size == 16) {
  for (x = 0; x < size / 8; x++) {
    if (t1[x] & t2[x]) t1[x] = 1;
    else if (t1[x] and !t2[x])
      t1[x] = t1[x] & op2[x];
    else if (!t1[x] & t2[x])
      t1[x] = t2[x] & op1[x];
    else t1[x] = 0;
  } else if (size == 8) {
// 0 if it’s lower 8 bits, 1 if it’s upper 8 bits
    pos1 = isUpper(op1); pos2 = isUpper(op2);
    if (t1[pos1] & t2[pos2]) t1[pos1] = 1;
    else if (t1[pos1] & !t2[pos2])
      t1[pos1] = t1[pos1] & op2[pos2];
    else if (!t1[pos1] & t2[pos2])
      t1[pos1] = t2[pos2] & op1[pos1];
    else t1[pos1] = 0;}}
if (mode64bit == 1 and size == 64)
  for (x = 32; x < size; x++) t1[x] = 0;
```