Tools and Implementation

Xiangyu Zhang

Outline

- Dynamic analysis tools
- Binary Decision Diagram
- Tools for undeterministic executions
- Static analysis tools

Dynamic Analysis Tools

- Introduction
  - Static instrumentation vs. dynamic instrumentation
- How to implement a dynamic information flow

What Is Instrumentation

```
Max = 0;
for (p = head; p; p = p->next)
{
    printf("In loop\n");
    count[0]++;
    if (p->value > max)
    {
        count[1]++;
        printf("True branch\n");
        max = p->value;
    }
}
```

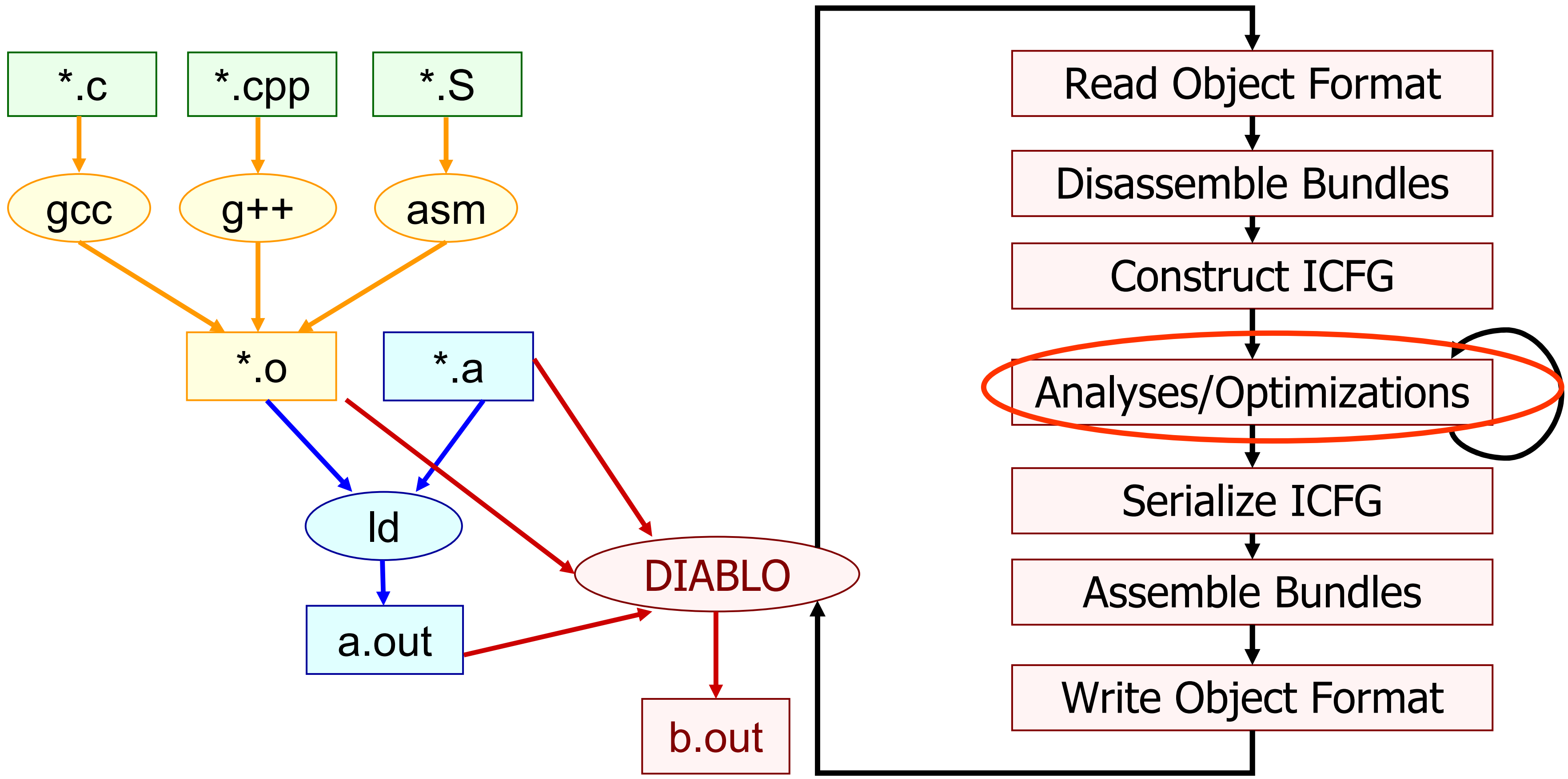
What Can Instrumentation Do?

- Profiler for compiler optimization:
  - Basic-block count
  - Value profile
- Micro architectural study:
  - Instrument branches to simulate branch predictors
  - Generate traces
- Bug checking:
  - Find references to uninitialized, unallocated address
- Software tools that use instrumentation:
  - Valgrind, Pin, Purify, ATOM, EEL, Diablo, …

Binary Instrumentation Is Dominant

- Libraries are a big pain for source code level instrumentation
  - Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries).
- Easily handle multi-lingual programs
  - Source code level instrumentation is heavily language dependent.
    - More complicated semantics
- Turning off compiler optimizations can maintain a almost perfect mapping from instructions to source code lines
- Worms and viruses are rarely provided with source code
- We will be talking about binary instrumentation only
  - Static
  - Dynamic

Static Instrumentation (Diablo)



Diablo Is A Better Link-time Optimizer

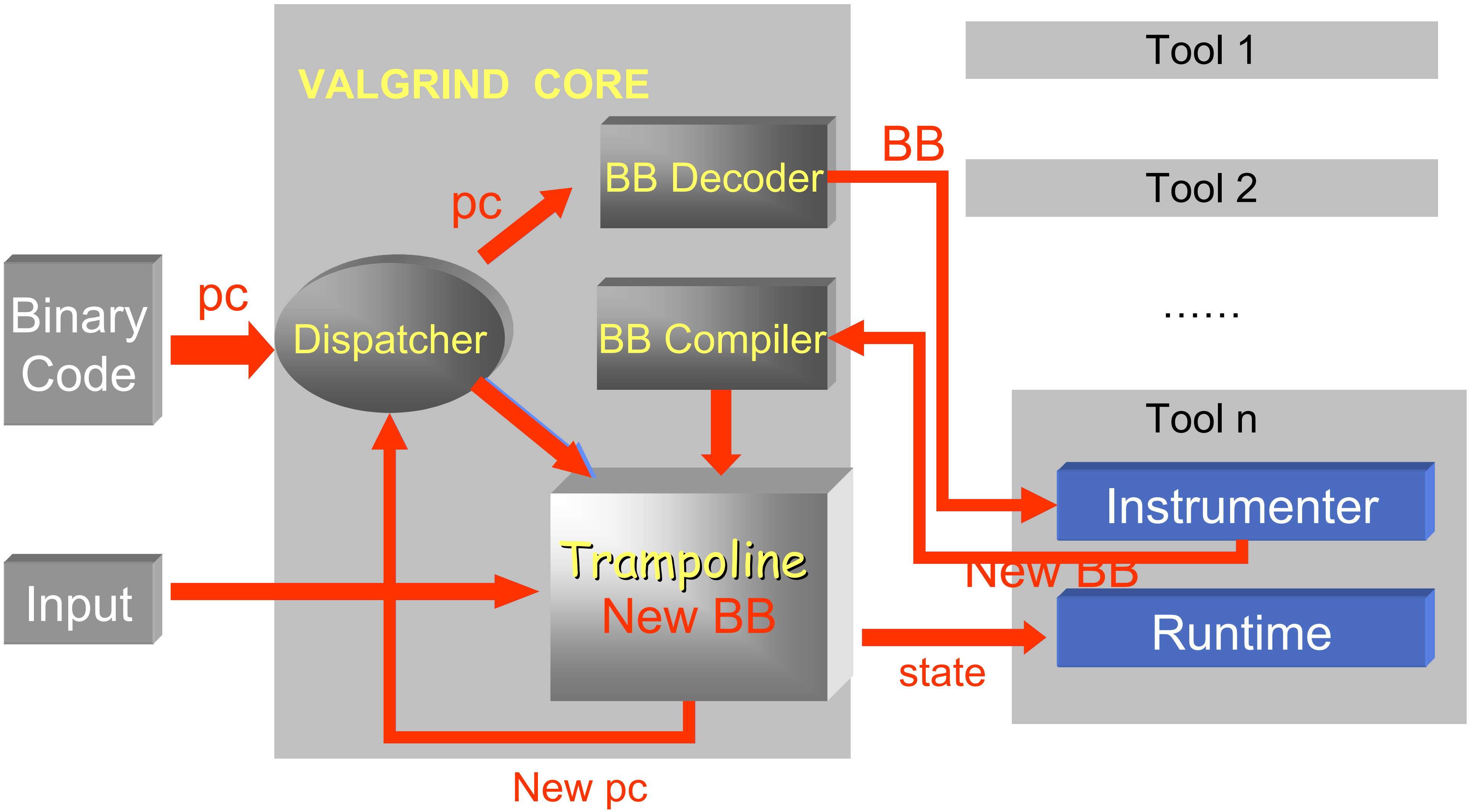
Static Instrumentation Characteristics

- Perform the instrumentation before the code is run
  - New binary = original binary + instrumentation
  - Raise binary to IR, transform IR, transfer back to binary
- All libraries are usually statically linked
  - The size of the binary is big
- Program representations are usually built from the binary
  - CFG
  - Call graph
  - PDG is hard to build from binary
    - Points-to analysis on binary is almost impossible
    - Simple DFA is possible
    - CDG is almost precise (PDG=CDG+DDG)

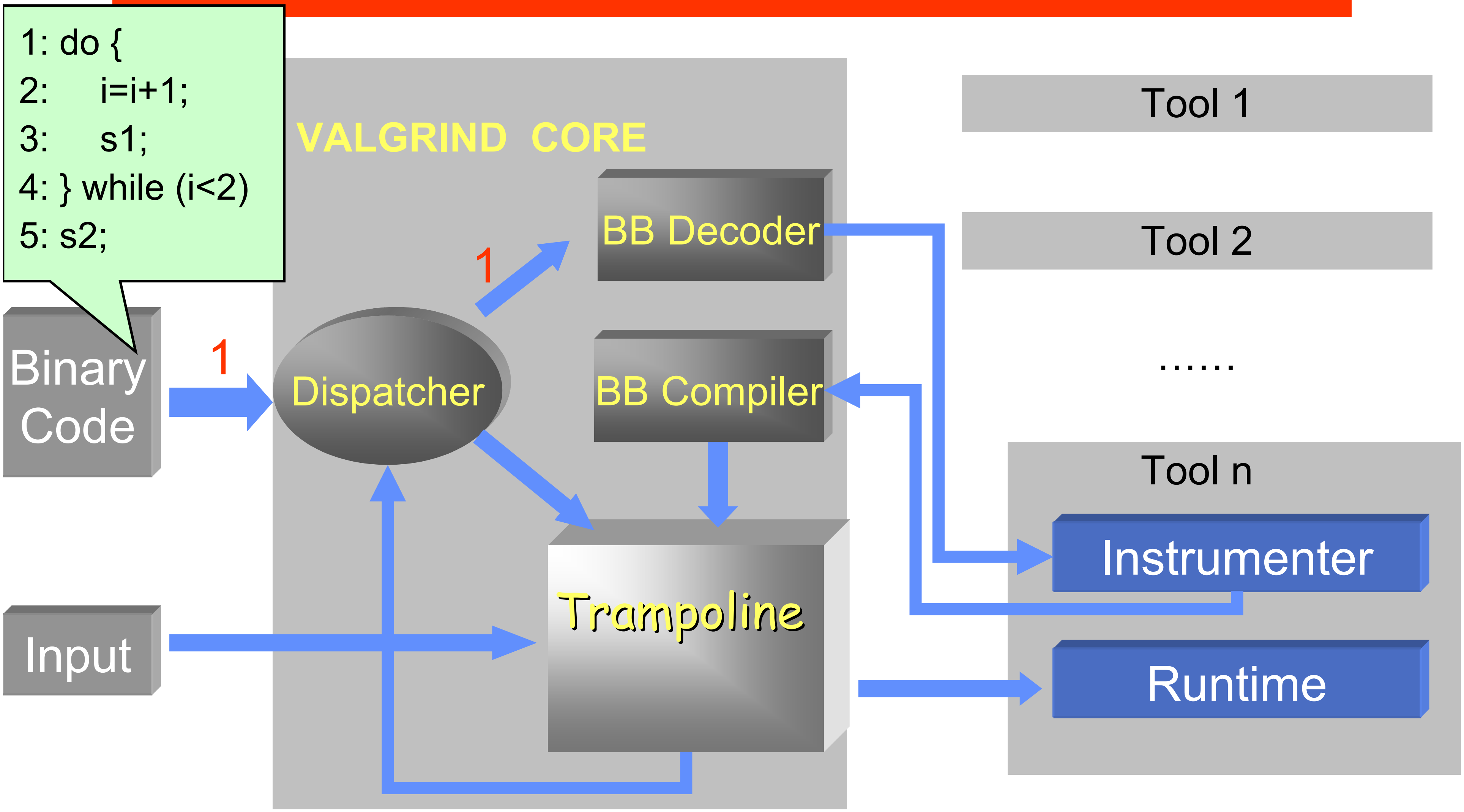
Dynamic Instrumentation - Valgrind

- Developed by Julian Seward at/around Cambridge University,UK
  - Google-O'Reilly Open Source Award for "Best Toolmaker" 2006
  - A merit (bronze) Open Source Award 2004
- Open source
  - works on x86, AMD64, PPC code
- Easy to execute, e.g.:
  - valgrind --tool=memcheck ls
- It becomes very popular
  - One of the two most popular dynamic instrumentation tools
    - Pin and Valgrind
  - Very good usability, extendibility, robust
    - 25MLOC
  - Mozilla, MIT, CMU-security, Me, and many other places
- Overhead is the problem
  - 5-10X slowdown without any instrumentation

Valgrind Infrastructure

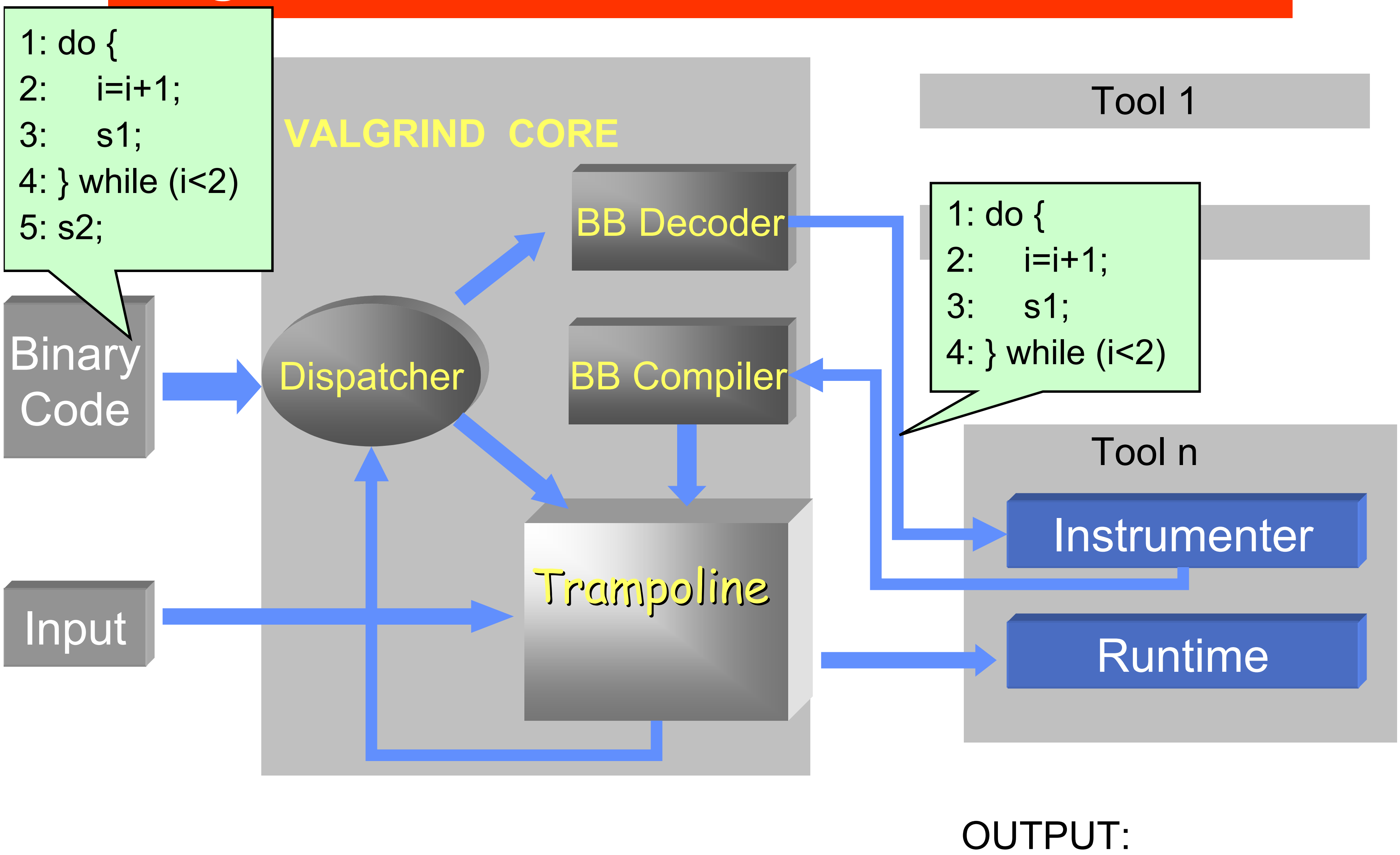


Valgrind Infrastructure

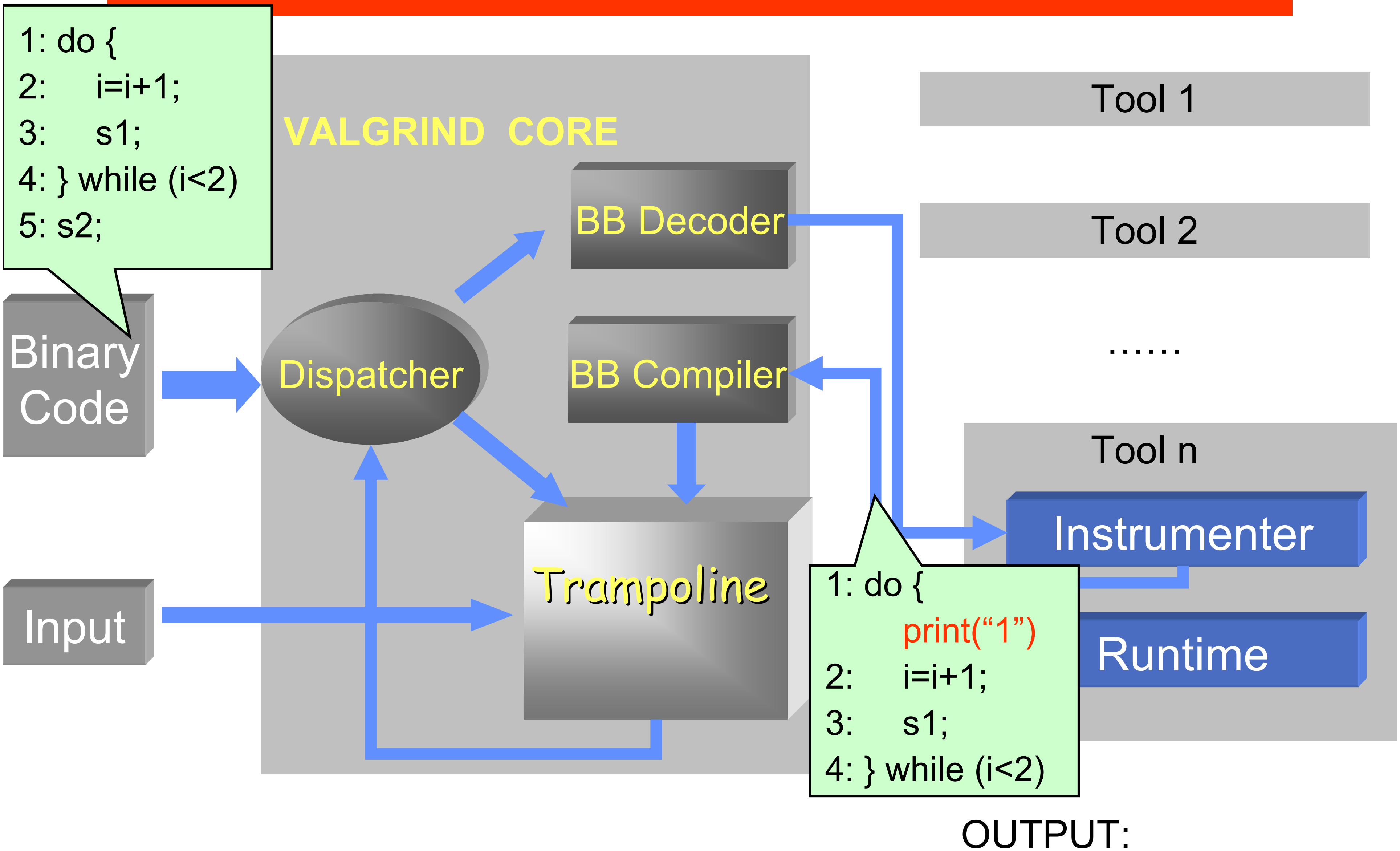


OUTPUT:

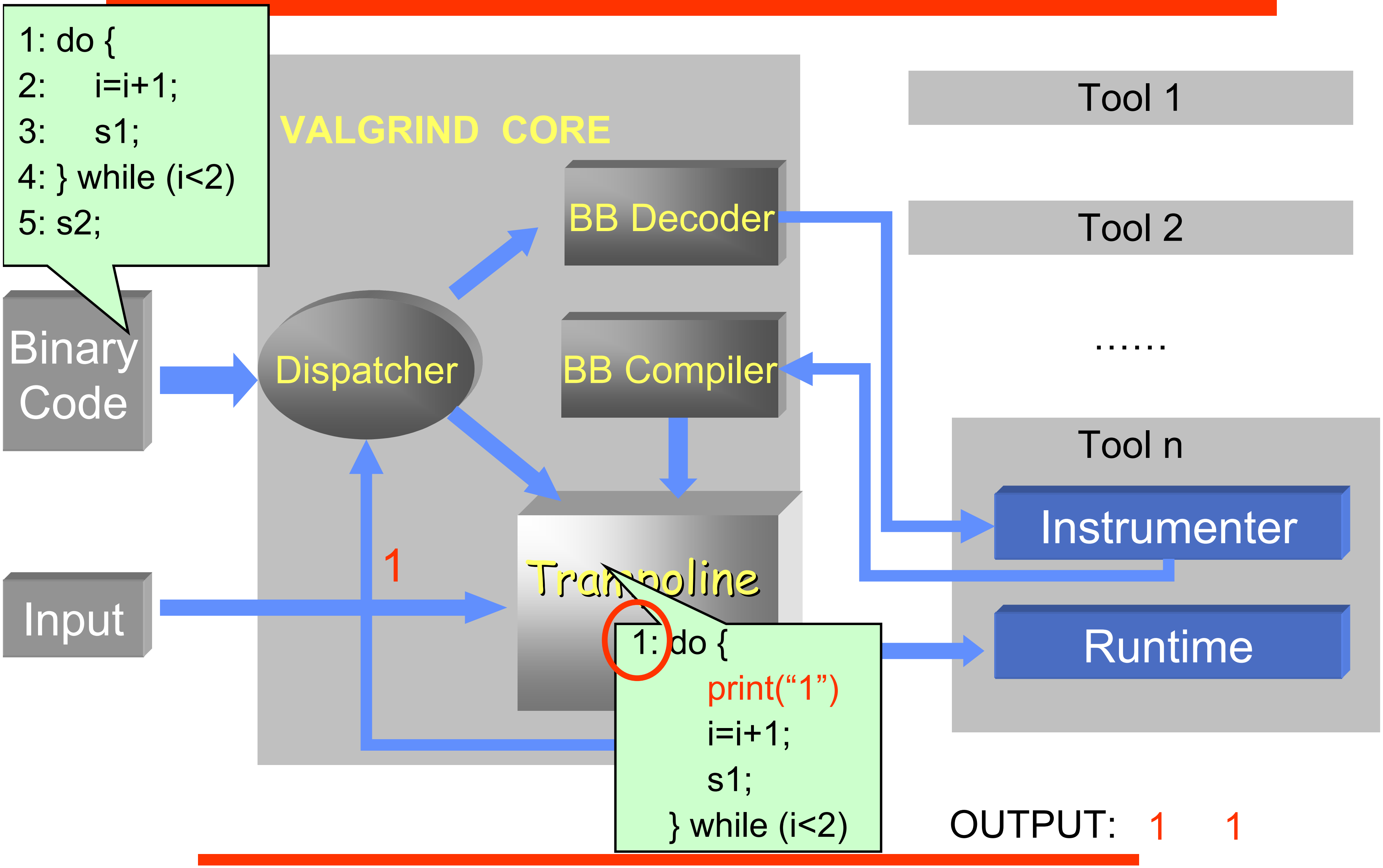
Valgrind Infrastructure



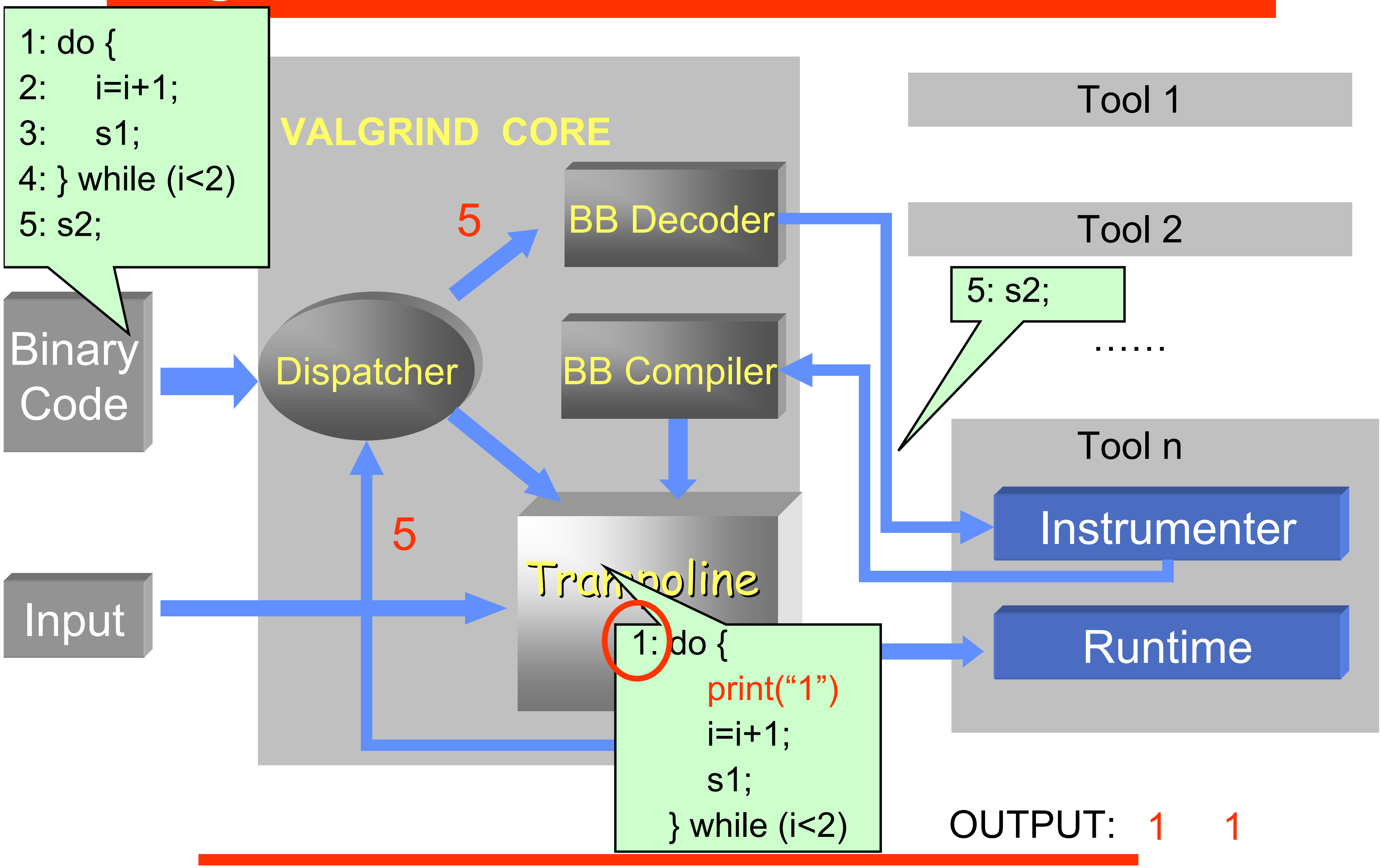
Valgrind Infrastructure



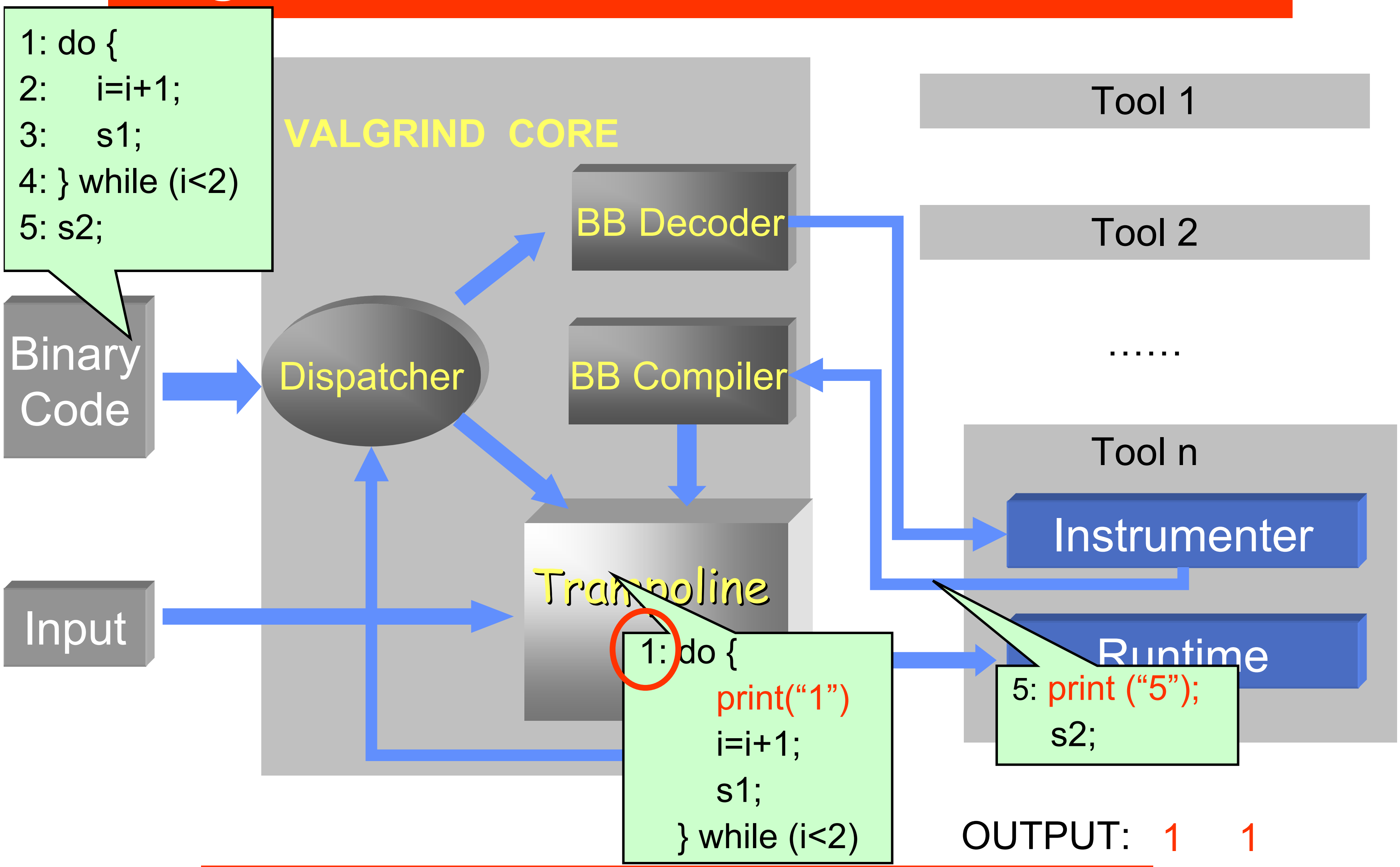
Valgrind Infrastructure



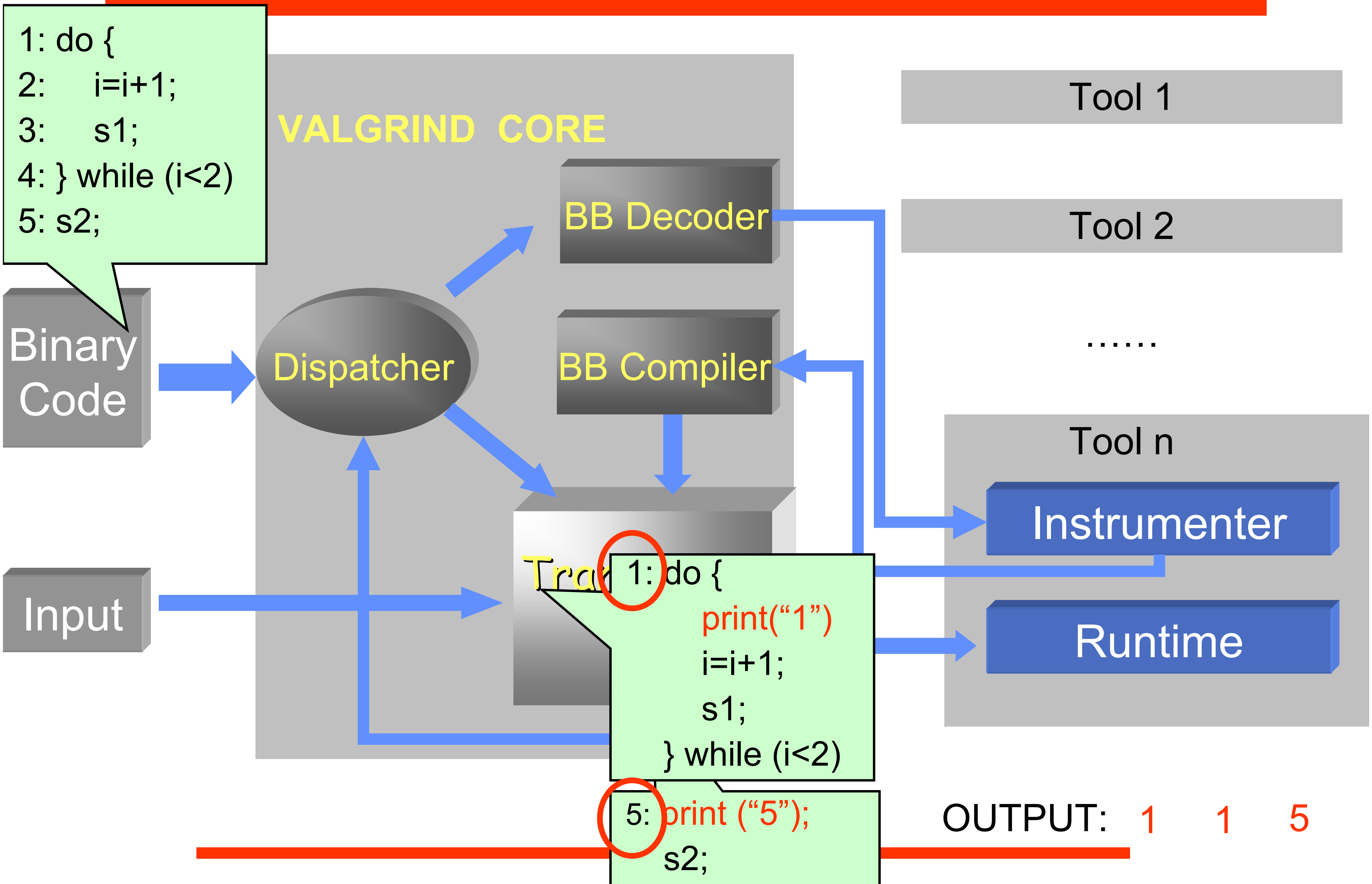
Valgrind Infrastructure



Valgrind Infrastructure



Valgrind Infrastructure



Dynamic Instrumentation Characteristics

- A trampoline is required.
- Does not require recompiling or relinking
  - Saves time: compile and link times are significant in real systems.
  - Can instrument without linking (relinking is not always possible).
- Dynamically turn on/off, change instrumentation
  - From t1-t2, I want to execute F', t3-t4, I want F''
    - Can be done by invalidating the mapping in the dispatcher.
- Can instrument running programs (such as Web or database servers)
  - Production systems.
- Can instrument self-mutating code.
  - Obfuscation can be easily get around.

Dynamic Instrumentation Characteristics

- Overhead is high
  - Dispatching, indexing;
  - Dynamic instrumentation
- Usually does not provide program representations at run time
  - Hard to acquire
  - Unacceptable runtime overhead
  - Simple representations such as BB are provided
  - GET AROUND: combine with static tools
    - Diablo + valgrind

Case Study: Implement A Dynamic Information Flow System in Valgrind

Information Flow System

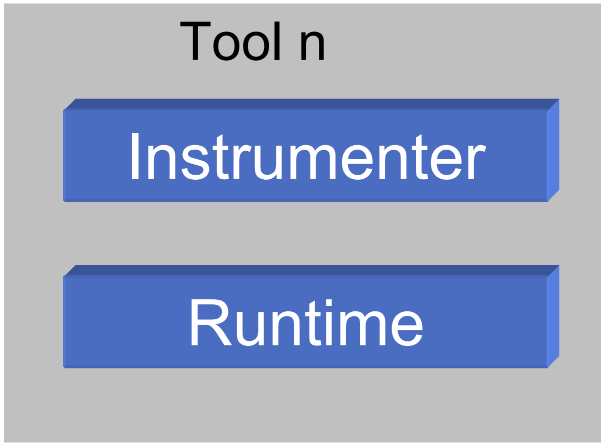
- IFS is important
  - Confidentiality at runtime = IFS
  - Tainted analysis = IFS
  - Memory reference errors detection = IFS
  - Data lineage system = IFS
  - Dynamic slicing is partly an IFS
- Essence of an IFS
  - A runtime abstract interpretation engine
    - Driven by the executed program path
- Implementation on Valgrind is surprisingly easy
  - Will see

Language and Abstract Model

- Our binary (RISC)
  - ADD r1 / #Imm, r2
  - LOAD [r1 / #Imm], r2
  - STORE r1, [r2 / #Imm]
  - MOV r1 / #Imm, r2
  - CALL r1
  - SYS_READ r1, r2
    - r1 is the starting address of the buffer, r2 is the size
- Abstract state
  - One bit, the security bit (tainted bit)
  - Prevent call at tainted value.

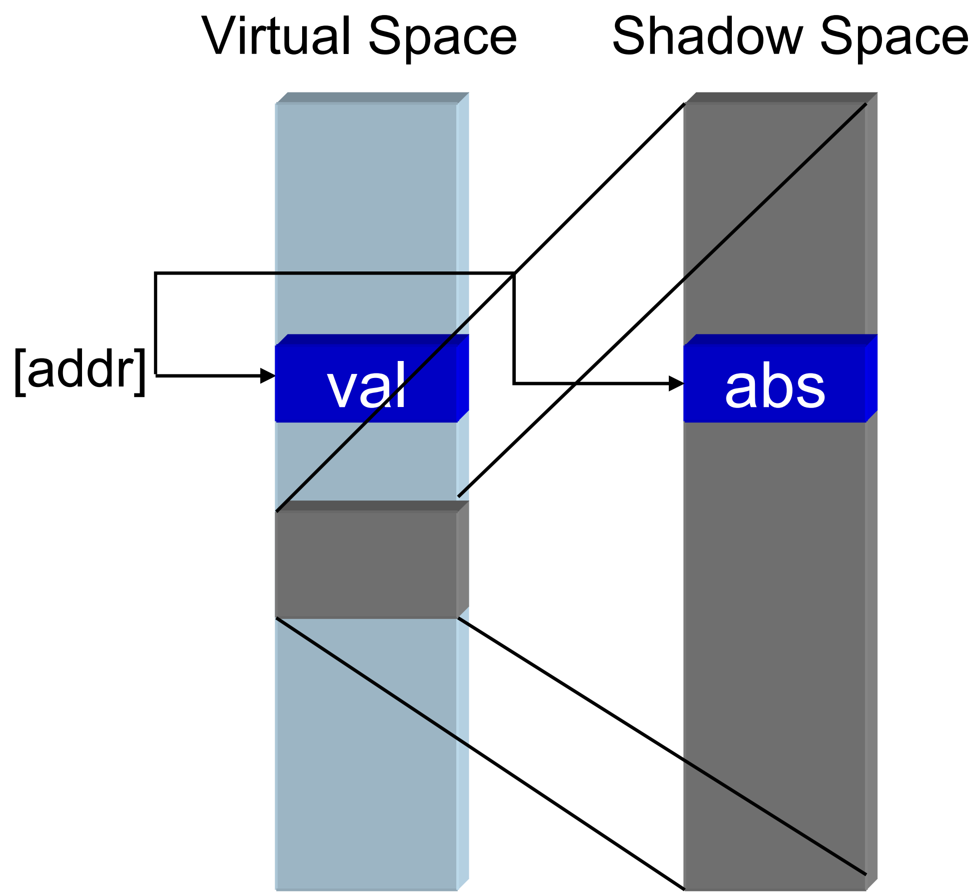
Implement A New Tool In Valgrind

- Use a template
  - The tool lackey is good candidate
  - Two parts to fill in
    - Instrumenter
    - Runtime
- Instrumenter
  - Initialization
  - Instrumentation
  - Finalization
  - System calls interception
- Runtime
  - Transfer functions
  - Memory management for abstract state



How to Store Abstract State

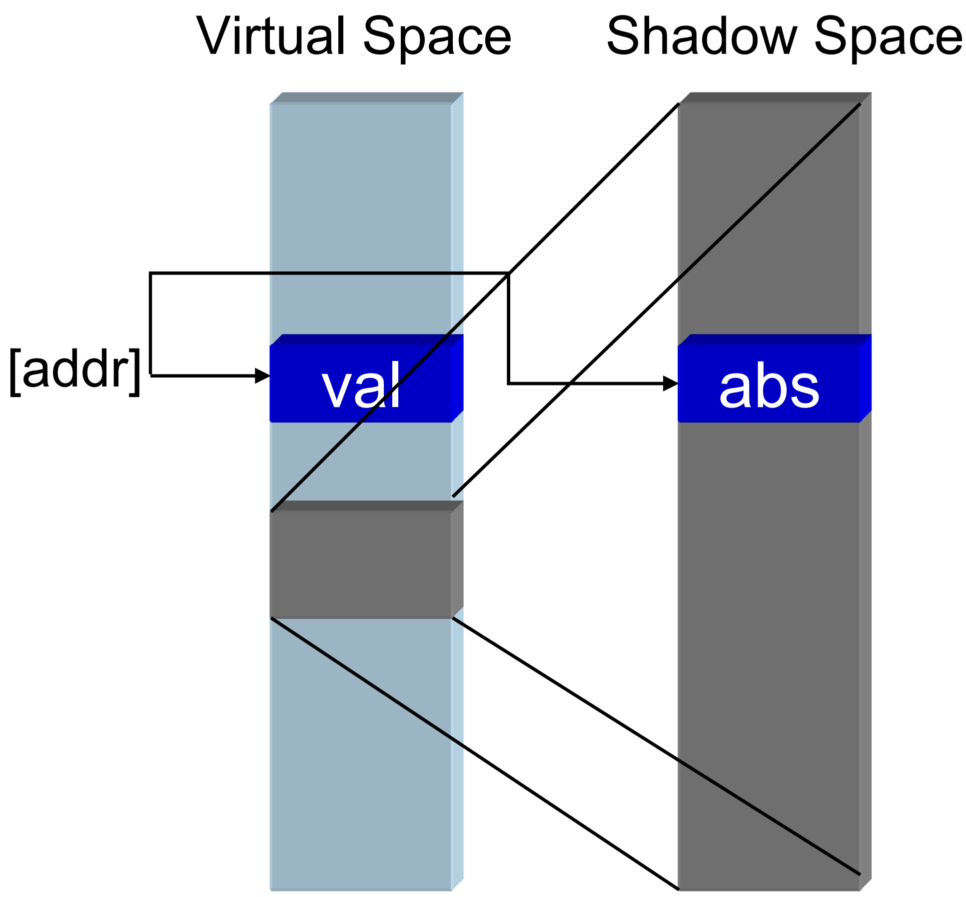
- Shadow memory
  - We need a mapping
    - Addr → Abstract State
    - Register → Abstract State




```
typedef
   struct {
      UChar abits[65536];
}   SecMap;

static SecMap* primary_map[65536];
static SecMap  default_map;
```

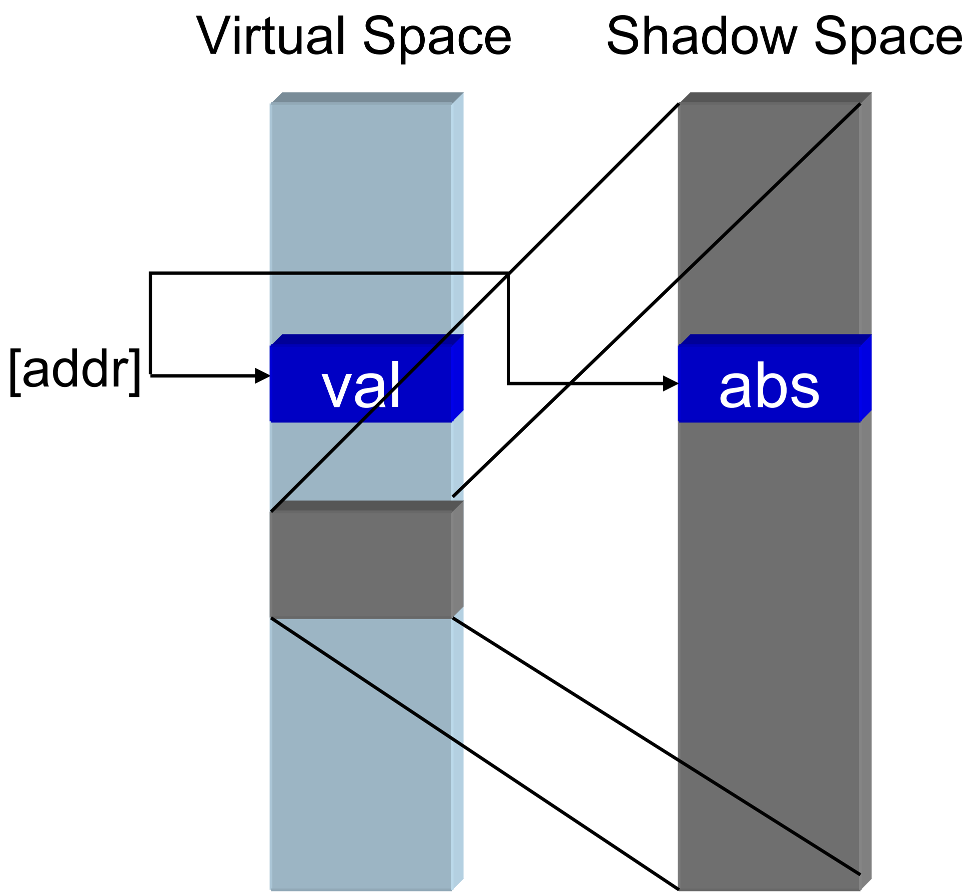
How to Store Abstract State

```
typedef
   struct {
      UChar abits[65536];
}   SecMap;

static SecMap* primary_map[65536];
static SecMap  default_map;

static void init_shadow_memory ( void )
{
  for (i = 0; i < 65536; i++)
      default_map.abits[i] = 0;
  for (i = 0; i < 65536; i++)
      primary_map[i] = &default_map;
}
```



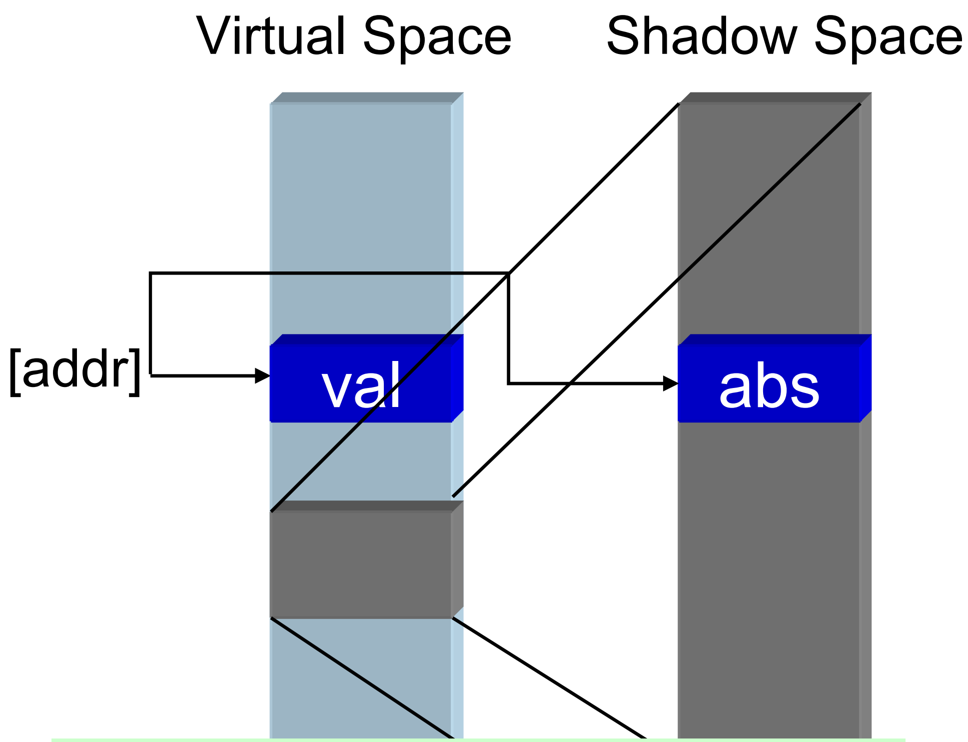
How to Store Abstract State

```
typedef
   struct {
      UChar abits[65536];
}   SecMap;

static SecMap* primary_map[65536];
static SecMap  default_map;

static void init_shadow_memory ( void )
{
  for (i = 0; i < 65536; i++)
      default_map.abits[i] = 0;
  for (i = 0; i < 65536; i++)
      primary_map[i] = &default_map;
}
static SecMap* alloc_secondary_map ()
{
   map =VG_(shadow_alloc)(sizeof(SecMap));
   for (i = 0; i < 65536; i++)
          map->abits[i] = 0;
   return map;
}
```



How to Store Abstract State

```
typedef
   struct {
      UChar abits[65536];
}   SecMap;

static SecMap* primary_map[65536];
static SecMap  default_map;

static void init_shadow_memory ( void )
{
  for (i = 0; i < 65536; i++)
      default_map.abits[i] = 0;
  for (i = 0; i < 65536; i++)
      primary_map[i] = &default_map;
}
static SecMap* alloc_secondary_map ()
{
   map =VG_(shadow_alloc)(sizeof(SecMap));
   for (i = 0; i < 65536; i++)
          map->abits[i] = 0;
   return map;
}
```




```
void Accessible (addr)
{
   if (primary_map[(addr) >> 16]
       == default_map)
          primary_map[(addr) >> 16] =
                 alloc_secondary_map(caller);
}
```

Initialization

```
void SK_(pre_clo_init)(void)
{
     VG_(details_name)           ("CS590F IFS");
     …
      init_shadow_memory();
      …
       VG_(needs_shadow_memory)     ();
       VG_(needs_shadow_regs)     ();
       …
       VG_(register_noncompact_helper)((Addr) & RT_load);
       VG_(register_noncompact_helper)((Addr) & …);
       …
}
```

Finalization

- EMPTY

```
void SK_(fini)(Int exitcode)
{
}
```

Instrumentation & Runtime

```
UCodeBlock* SK_(instrument)(UCodeBlock* cb_in, …)
{
        …
        UCodeBlock cb = VG_(setup_UCodeBlock)(…);
        …
        for (i = 0; i < VG_(get_num_instrs)(cb_in); i++) {
                u = VG_(get_instr)(cb_in, i);
                 switch (u->opcode) {
                      case LD:
                                …
                      case ST:
                                …
                      case MOV:
                                …
                      case ADD:
                                …
                      case CALL:
                                …
       return cb;
}
```

Instrumentation & Runtime - LOAD

LD [r1], r2

```
switch (u->opcode) {
    case LD:
      VG_(ccall_RR_R) (cb, (Addr) RT_load, u->
          r1, SHADOW (u->r1), SHADOW(U->r2)
}
```

SHADOW(r2)=SM(r1) | SHADOW (r1)

```
UChar RT_load (Addr r1, UChar sr1)
{
    UChar s_bit=primary_map[a >> 16][a && 0xffff];
    return (s_bit  | sr1);

}
```

Instrumentation & Runtime - STORE

ST r1, [r2]

```
switch (u->opcode) {
    case ST:
     VG_(ccall_RRR_0) (cb, (Addr) RT_store,
            u->r2, SHADOW (u->r1), SHADOW(u->r2);
}
```

SM(r2)=SHADOW(r1) | SHADOW (r2)

```
void RT_store (Addr a, UChar sr1, UChar sr2)
{
    UChar s_bit= sr1 | sr2;
    Accessible(a);
    primary_map[a >> 16][a && 0xffff]=s_bit;
}
```

Instrumentation & Runtime - MOV

MOV r1, r2

```
switch (u->opcode) {
    case MOV:
      uInstr2(cb, MOV,…, SHADOW(u->r1), …
                 SHADOW(u->r2)
}
```

SHADOW(r2) = SHADOW (r1)

Instrumentation & Runtime - ADD

ADD r1, r2

```
switch (u->opcode) {
    case ST:
      VG_(ccall_RR_R) (cb, (Addr) RT_add, SHADOW(u->r1),
                       SHADOW (u->r2), SHADOW(u->r2);
}
```

SHADOW(r2) = SHADOW (r1) | SHADOW (r2)

```
UChar RT_add (UChar sr1, UChar sr2)
{
    return sr1 | sr2;
}
```

Instrumentation & Runtime - CALL

CALL r1

```
switch (u->opcode) {
    case ST:
      VG_(ccall_R_0) (cb, (Addr) RT_call, SHADOW(u->r1));

}
```

if (SHADOW(r1)) printf ("Pleae call CS590F")

```
UChar RT_call (UChar sr1)
{
    if (sr1) VG_(printf) ("Please call CS590F\n");
}
```

Instrumentation & Runtime – SYS_READ

SYS_READ r1, r2

SM (r1[0-r2])=1

```
void * SK_(pre_syscall) (… UInt syscallno…)
{
    …
    if (syscallno==SYSCALL_READ) {
        get_syscall_params (…, &r1, &r2,…);
        for (i=0;i<r2;i++) {
            a= &r1[i];
            Accessible(a);
            primary_map[a >> 16][a && 0xffff]=1;
        }
    }
    …
}
```

Done!

- Let us run it through a buffer overflow exploit

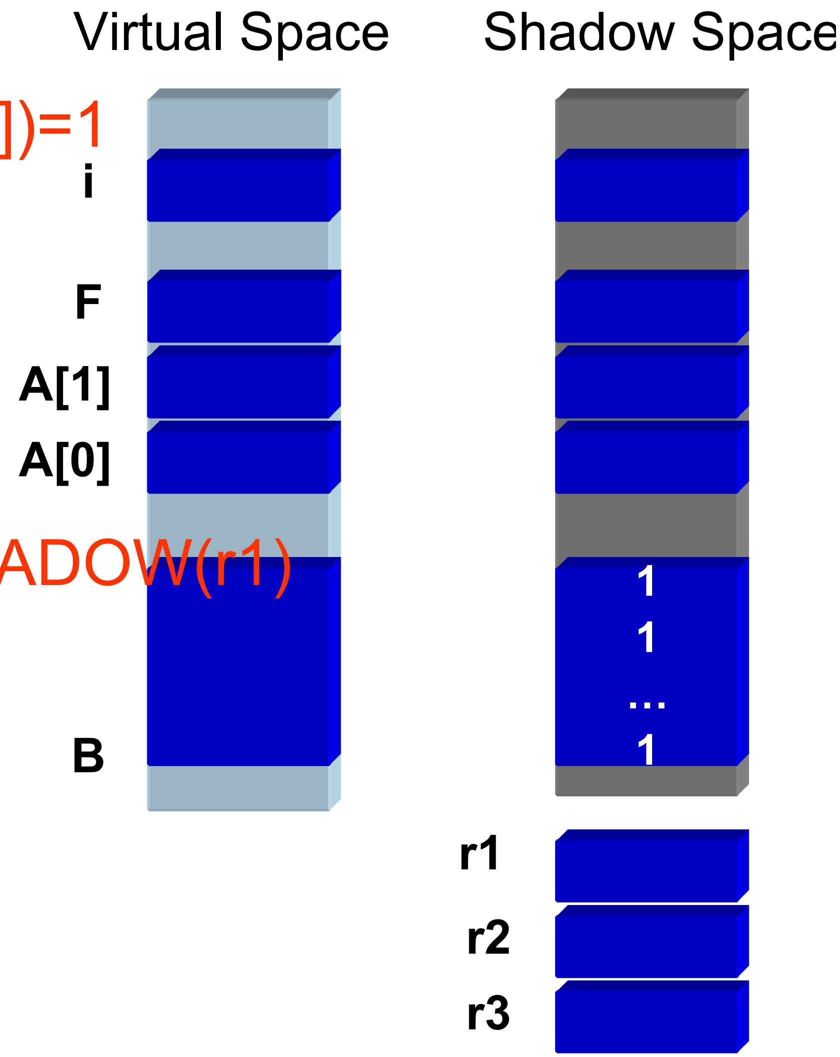
```
void (* F) ();
char A[2];
...
read(B, 256);
i=2;
A[i]=B[i];
...
(*F) ();
```

```
void (* F) ();
char A[2];
...
read(B, 256);

...

i=2;
...

A[i]=B[i];
...




(*F) ();
```

```
...
MOV &B, r1
MOV 256, r2
SYS_Read r1, r2
...
MOV 2, r1
ST r1, [&i]
...
LD [&i], r1
MOV &B, r2
ADD r1, r2
LD [r2], r2
MOV &A, r3
ADD r1, r3
ST r2, [r3]
...
MOV F, r1
CALL r1
```

SM (r1[0-r2])=1

SM(&i)=SHADOW(r1)

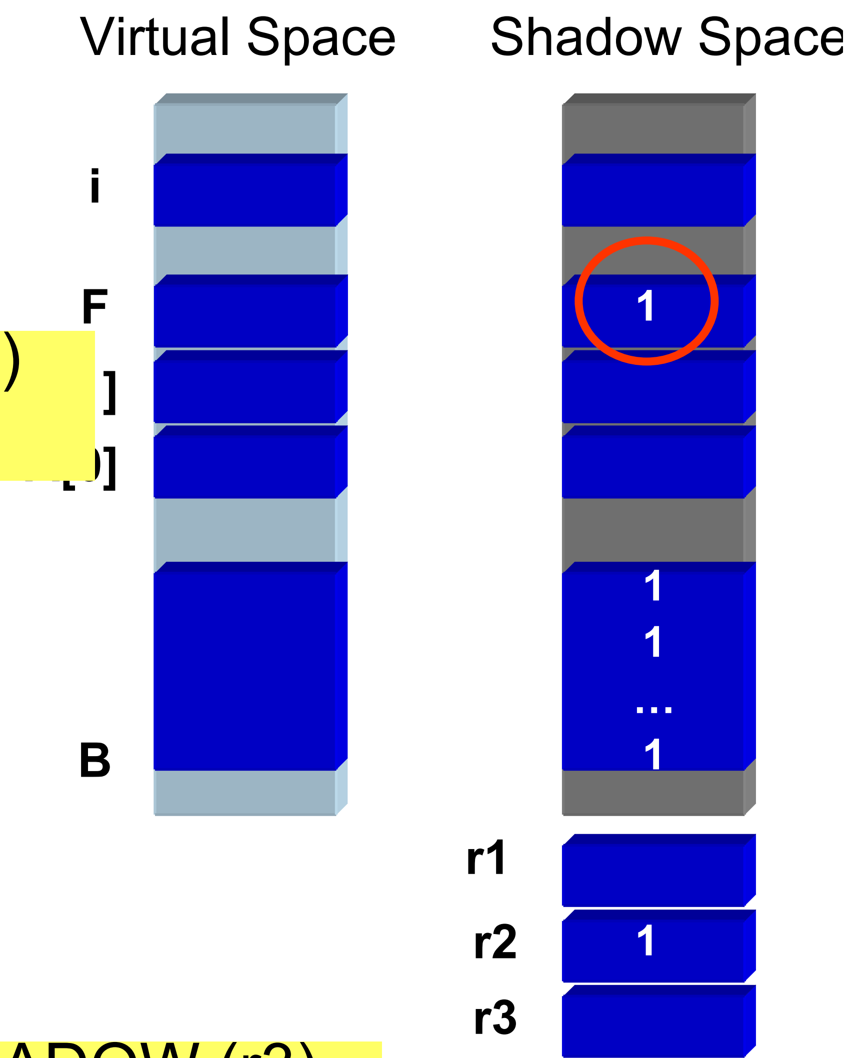


```
void (* F) ();
char A[2];
...
read(B, 256);
...
...
A[i]=B[i];
...
(*F) ();
```

```
...
MOV &B, r1
MOV 256, r2
SYS  Read r1, r2
ST r1, [&i]
...
LD [&i], r1
MOV &B, r2
ADD r1, r2
LD [r2], r2
MOV &A, r3
ADD r1, r3
ST r2, [r3]
...
MOV F, r1
```

SHADOW(r2)=SM(r2) | SHADOW (r2)
r2=&B[2];

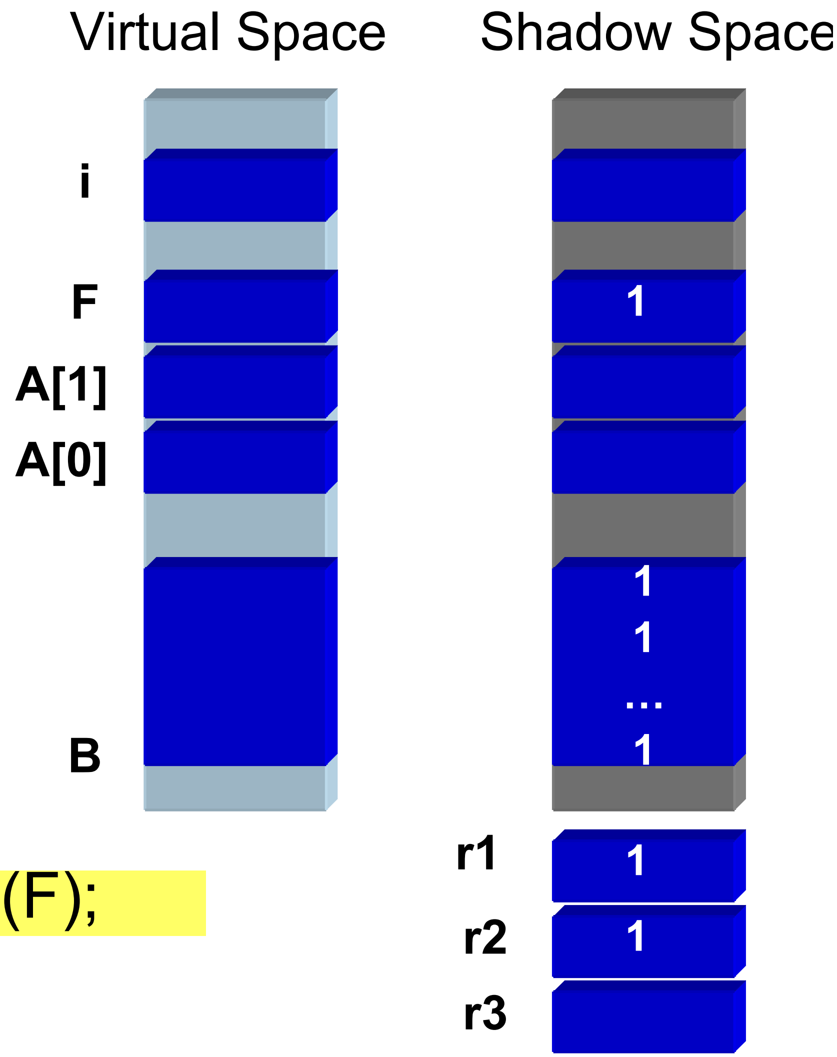
SM (r3)=SHADOW(r2) | SHADOW (r3)
r3=&A[2]



```
void (* F) ();
char A[2];
...
read(B, 256);

...

i=2;
...

A[i]=B[i];
...






(*F) ();
```

```
...
MOV &B, r1
MOV 256, r2
SYS_Read r1, r2
...
MOV 2, r1
ST r1, [&i]
...
LD [&i], r1
MOV &B, r2
ADD r1, r2
LD [r2], r2
MOV &A, r3
ADD r1, r3
ST r2, [r3]
...
MOV F, r1
CALL r1
```

SHADOW(r1)=SM(F);



if (SHADOW(r1)) printf (“Call …”);

What Is Not Covered

- Information flow through control dependence
  - Valgrind is not able to handle
  - Valgrind + diablo

```
p=getpassword( );
…
if (p==“zhang”) {
    send (m);
}
```

Outline

- Dynamic analysis tools
- Binary Decision Diagram
- Tools for undeterministic executions
- Static analysis tools

Why BDD?

- It is an efficient representation for boolean functions
  - What can be represented by boolean functions?
    - Sets, relations, …
  - What is program analysis about (both static and dynamic)
    - Manipulating sets
- Existing applications
  - In PA
    - Points-to analysis
    - Dynamic slicing
    - Data lineage
    - Test prioritization (??)
  - Others
    - Circuit optimization

Points-to Analysis Using BDD

```
X: a = new O();
Y: b = new O();
Z: c = new O();
a = b;
b = a;
c = b;
```

Points-to set:
{ (a,X) (b,Y) (c,Z) (a,Y) (b,X) (c,X) (c,Y) }

Unification based flow-insensitive analysis

BDD representation

- A BDD is a compact representation of a boolean function
- The points-to relations can be encoded into a boolean function

$a \rightarrow 00 \quad X \rightarrow 00$

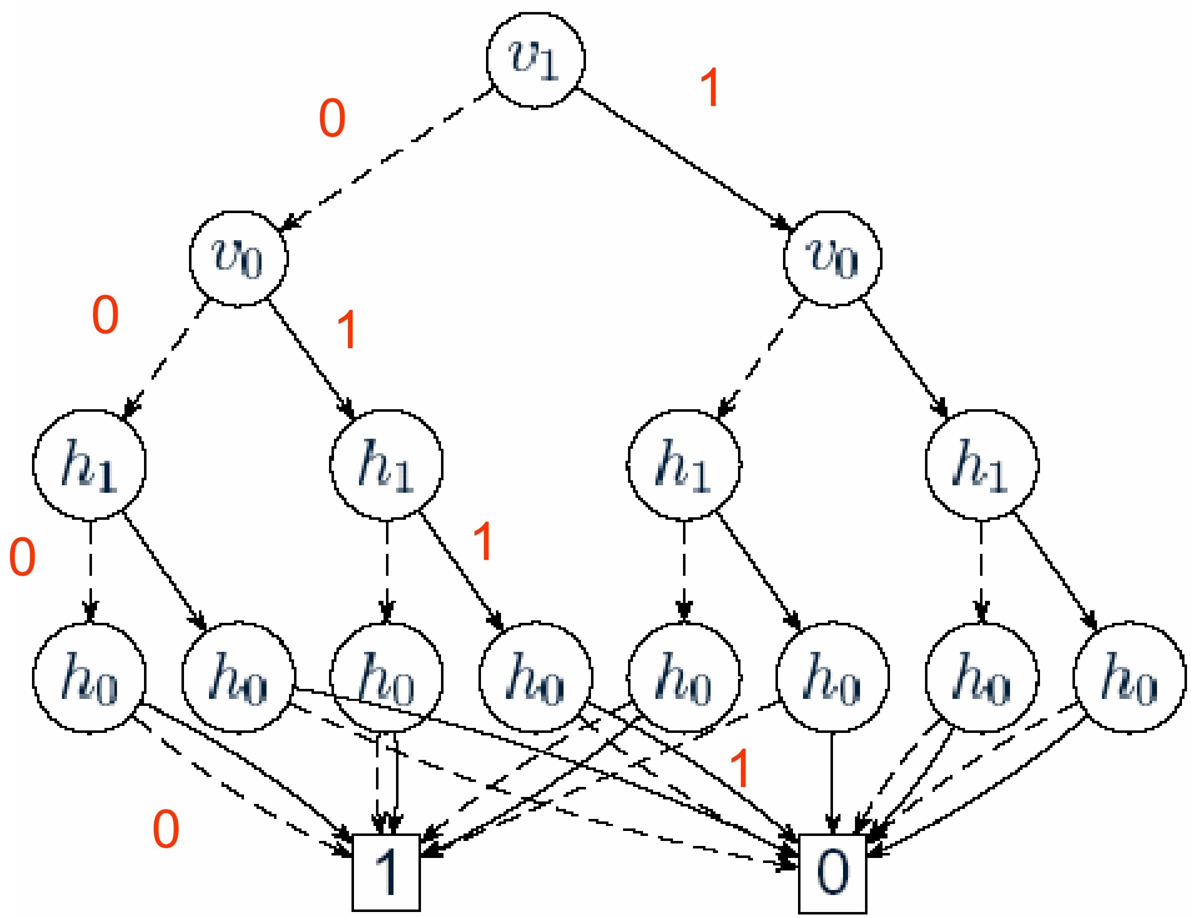
$b \rightarrow 01 \quad Y \rightarrow 01$

$c \rightarrow 10 \quad Z \rightarrow 10$

Domains: V H

$$\begin{array}{c} \phantom{(a,Y) \rightarrow{}} v_1 v_0 h_1 h_0 \\ (a,Y) \rightarrow 00\ 01 \end{array}$$

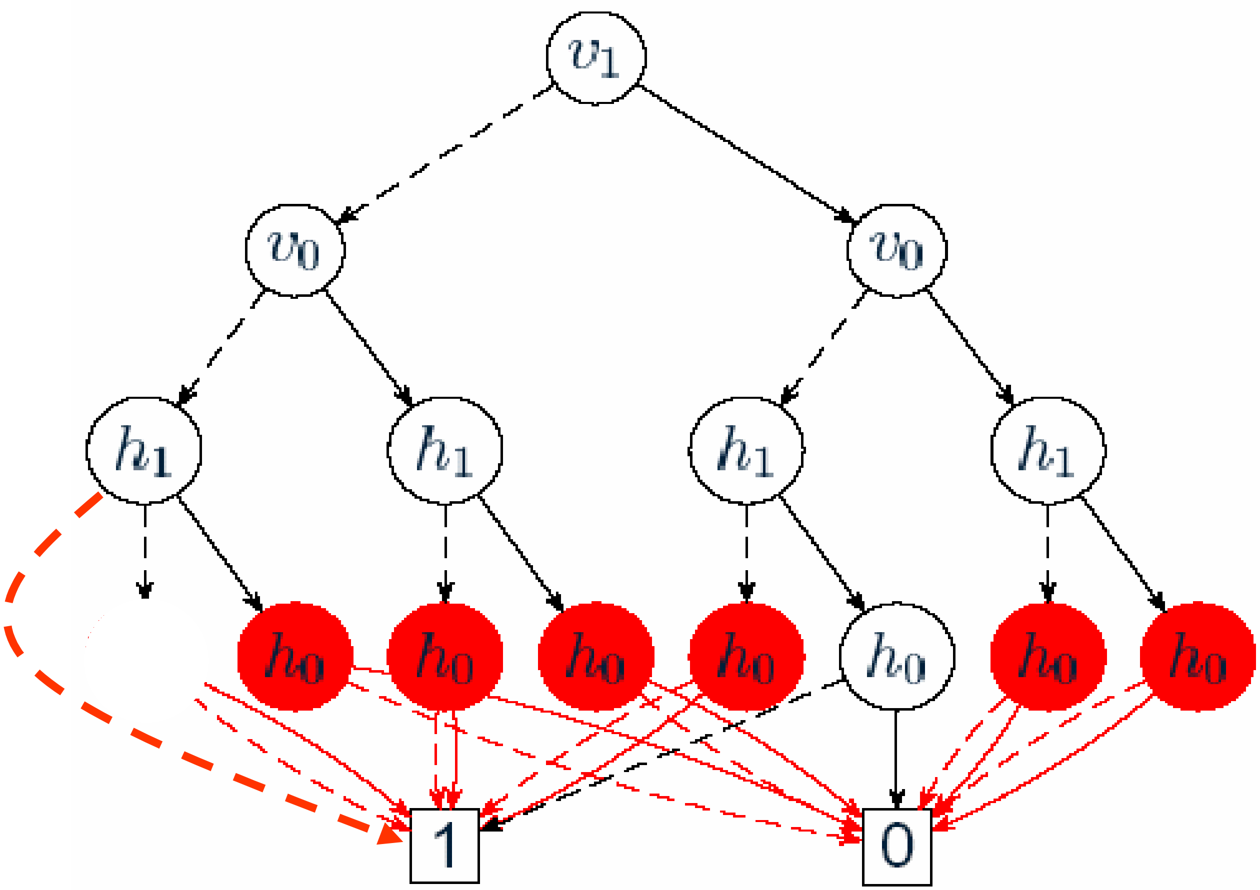
BDD representation



a/X $\rightarrow$ 00
b/Y $\rightarrow$ 01
c/Z $\rightarrow$ 10

	V H
	$v_1 v_0 h_1 h_0$
(a,X)	00 00
(a,Y)	00 01
(b,X)	01 00
(b,Y)	01 01
(c,X)	10 00
(c,Y)	10 01
(c,Z)	10 10

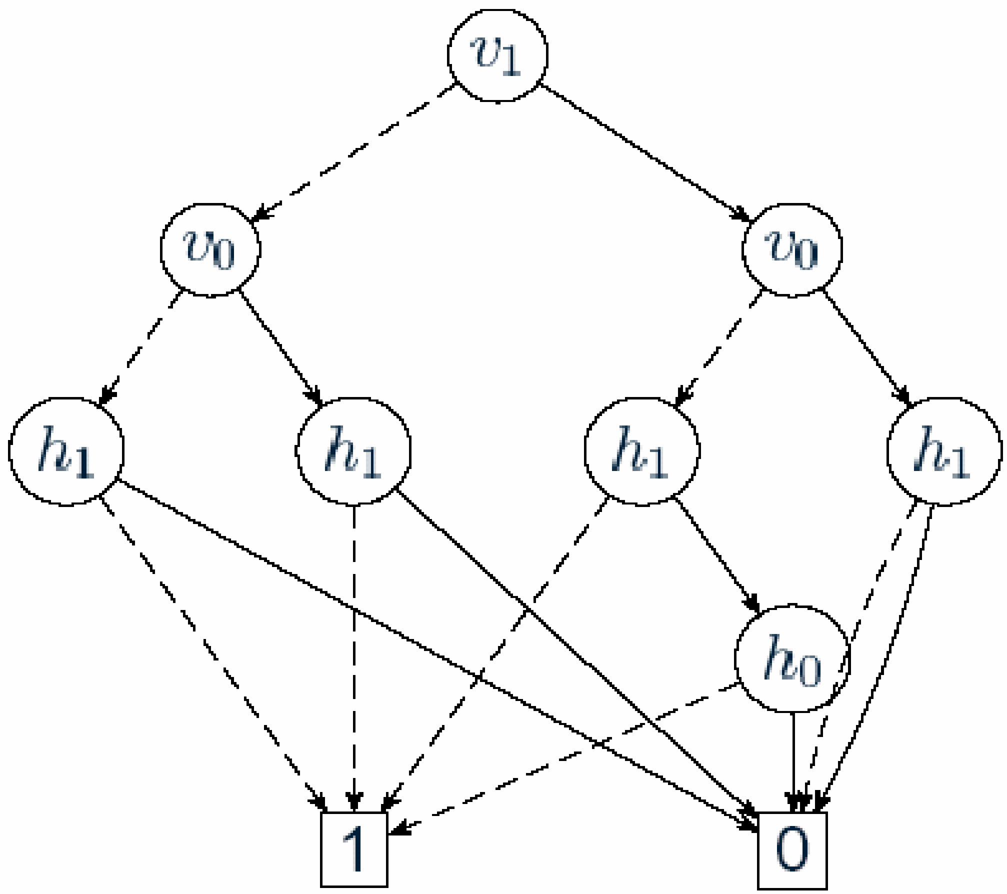
BDD Representation



a/X $\rightarrow$ 00
b/Y $\rightarrow$ 01
c/Z $\rightarrow$ 10

	V	H
	$v_1 v_0$	$h_1 h_0$
(a,X)	00	00
(a,Y)	00	01
(b,X)	01	00
(b,Y)	01	01
(c,X)	10	00
(c,Y)	10	01
(c,Z)	10	10

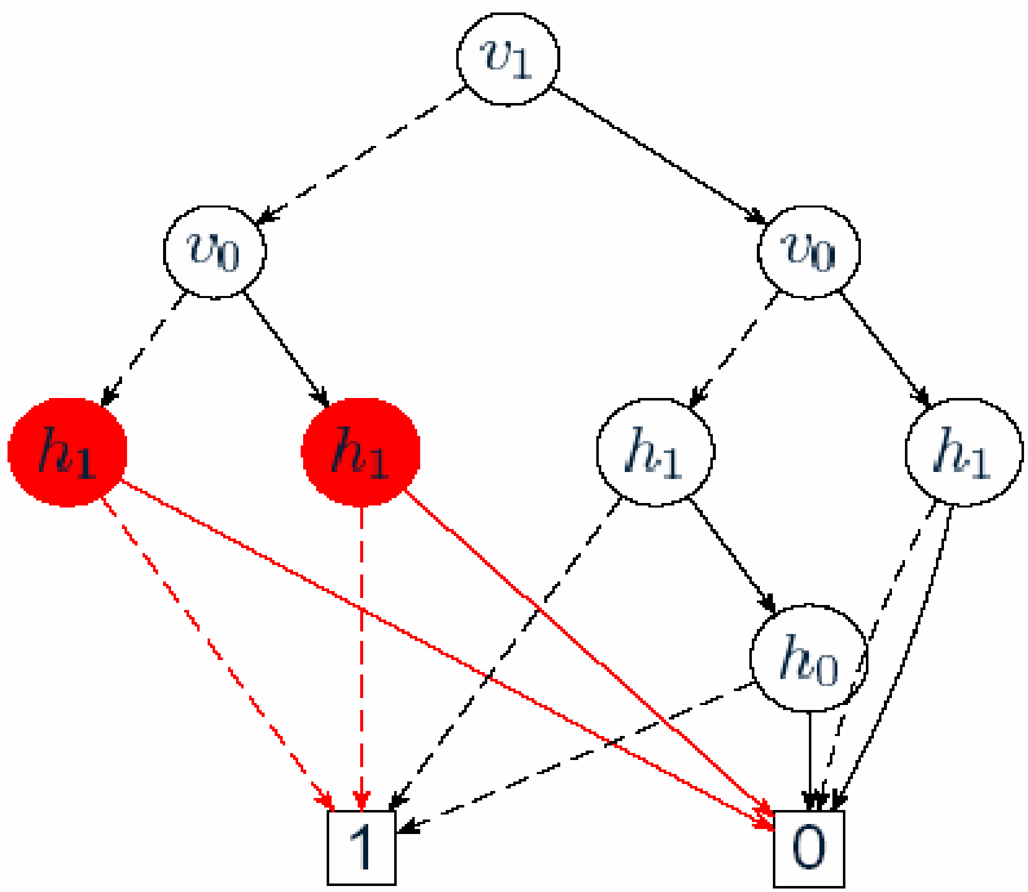
BDD Representation



a/X → 00
b/Y → 01
c/Z → 10

	V	H
	$v_1 v_0$	$h_1 h_0$
(a,X)	00	00
(a,Y)	00	01
(b,X)	01	00
(b,Y)	01	01
(c,X)	10	00
(c,Y)	10	01
(c,Z)	10	10

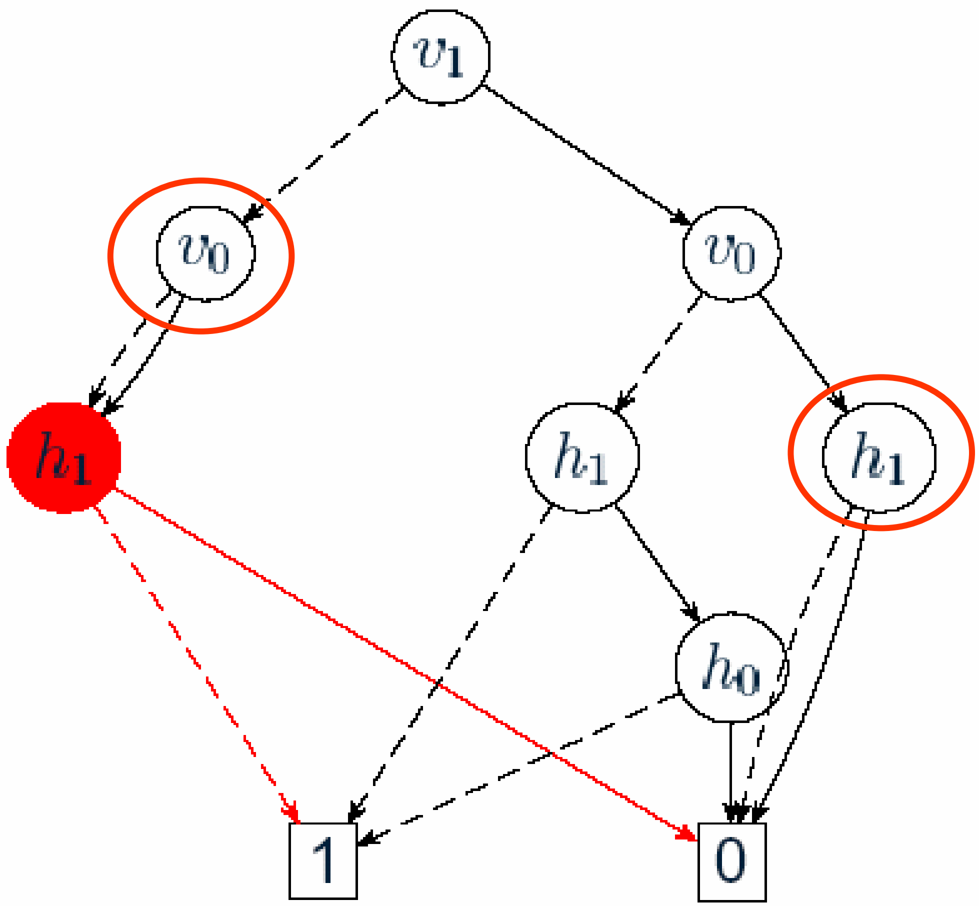
BDD Representation



a/X $\rightarrow$ 00
b/Y $\rightarrow$ 01
c/Z $\rightarrow$ 10

	V	H
	$v_1 v_0$	$h_1 h_0$
(a,X)	00	00
(a,Y)	00	01
(b,X)	01	00
(b,Y)	01	01
(c,X)	10	00
(c,Y)	10	01
(c,Z)	10	10

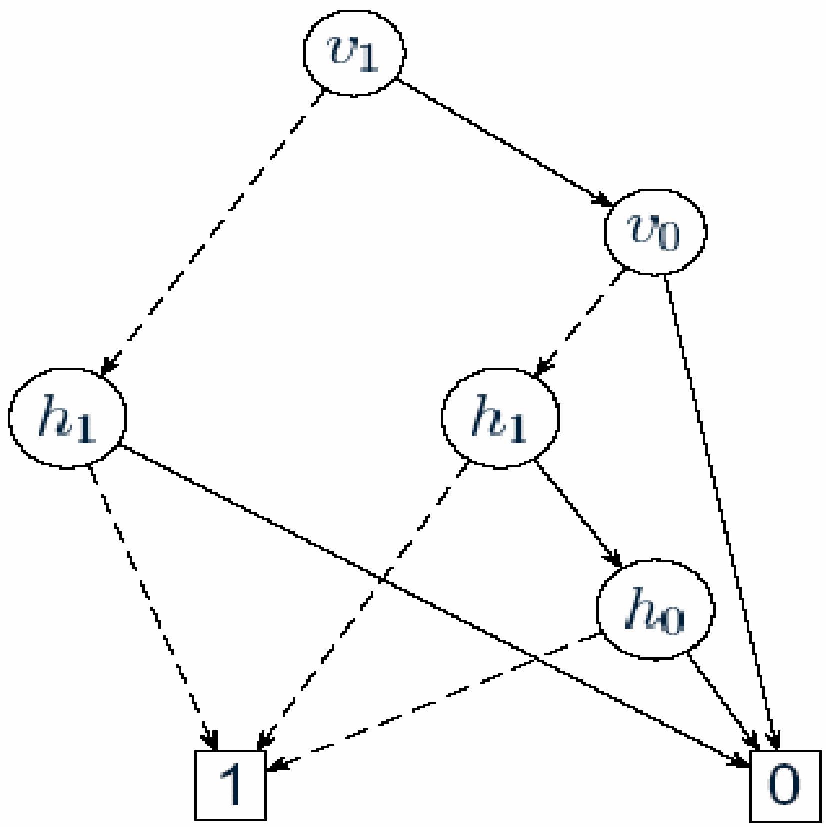
BDD Representation



$a/X \rightarrow 00$
$b/Y \rightarrow 01$
$c/Z \rightarrow 10$

	V H
	$v_1 v_0 h_1 h_0$
(a,X)	00 00
(a,Y)	00 01
(b,X)	01 00
(b,Y)	01 01
(c,X)	10 00
(c,Y)	10 01
(c,Z)	10 10

Final Reduced BDD



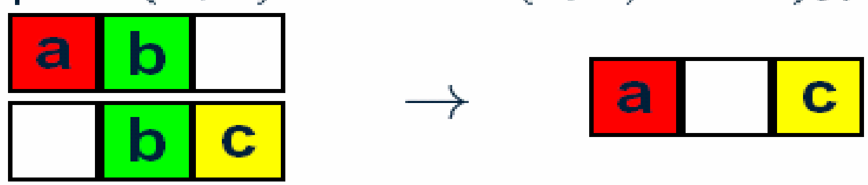
a/X → 00
b/Y → 01
c/Z → 10

	V	H
	$v_1 v_0$	$h_1 h_0$
(a,X)	00	00
(a,Y)	00	01
(b,X)	01	00
(b,Y)	01	01
(c,X)	10	00
(c,Y)	10	01
(c,Z)	10	10

BDD Operations

- Set operations
  - Union, intersection,…
- Relational product

$$(\{(a,c) \mid \exists b.(a,b) \in X \wedge (b,c) \in Y\})$$



- Cost of the operations is proportional to the number of nodes, not the elements in the set (relation)

Mapping Points-to Transfer Functions to BDD Operations

X=(V,V) Y=(V,H)

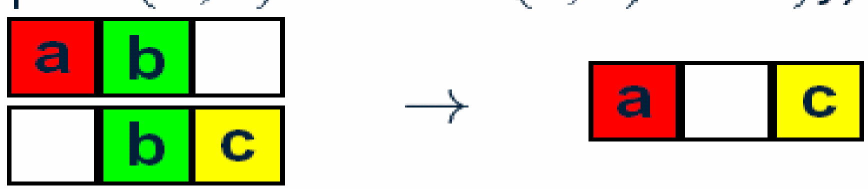
```
X: a = new O();
Y: b = new O();
Z: c = new O();
```

(a,X)
(b,Y)
(c,Z)

b=a; (b,a) (b,X)

Relational product rule

$$(\{(a, c) \mid \exists b.(a, b) \in X \wedge (b, c) \in Y\})$$



BDD in Dynamic Analysis (Data Lineage)

- What is Data Lineage
  - Given a value during the execution, the lineage of the value is the set of input that contributes to computation of the value.
- BDD is the perfect choice for lineage sets
  - Z=X+Y ➔ L(Z) = L(X) U L(Y)
- BDD in dynamic slicing
- BDD in …
- Tool
  - BuDDy

Outline

- Dynamic analysis tools
- Binary Decision Diagram
- Tools for undeterministic executions
- Static analysis tools

Jockey

- Execution record/replay tool (OPEN SOURCE)
  - X86 binaries
  - Used as a user-space library
  - Handle multi-threading programs
  - Checkpointing
- How it works
  - Use code pattern matching to identify all the system calls and replace them
  - Record phase
  - Replay phase

Simics-A Simulator

- full system simulation technology (NOT FULLY OPEN SOURCE)
  - the software cannot detect the difference between real production hardware and Simics' virtual environment.
  - Have the full control over the entire execution context
    - Application code
    - OS code
    - Driver code
  - Fast
- Widely used in multi-core related research

Outline

- Dynamic analysis tools
- Binary Decision Diagram
- Tools for undeterministic executions
- Static analysis tools

Static Analysis Tool

- Previously
  - SUIF
  - TRIMARAN
- Currently
  - CodeSurfer
  - CIL