Implement A Dynamic Information Flow System in Valgrind

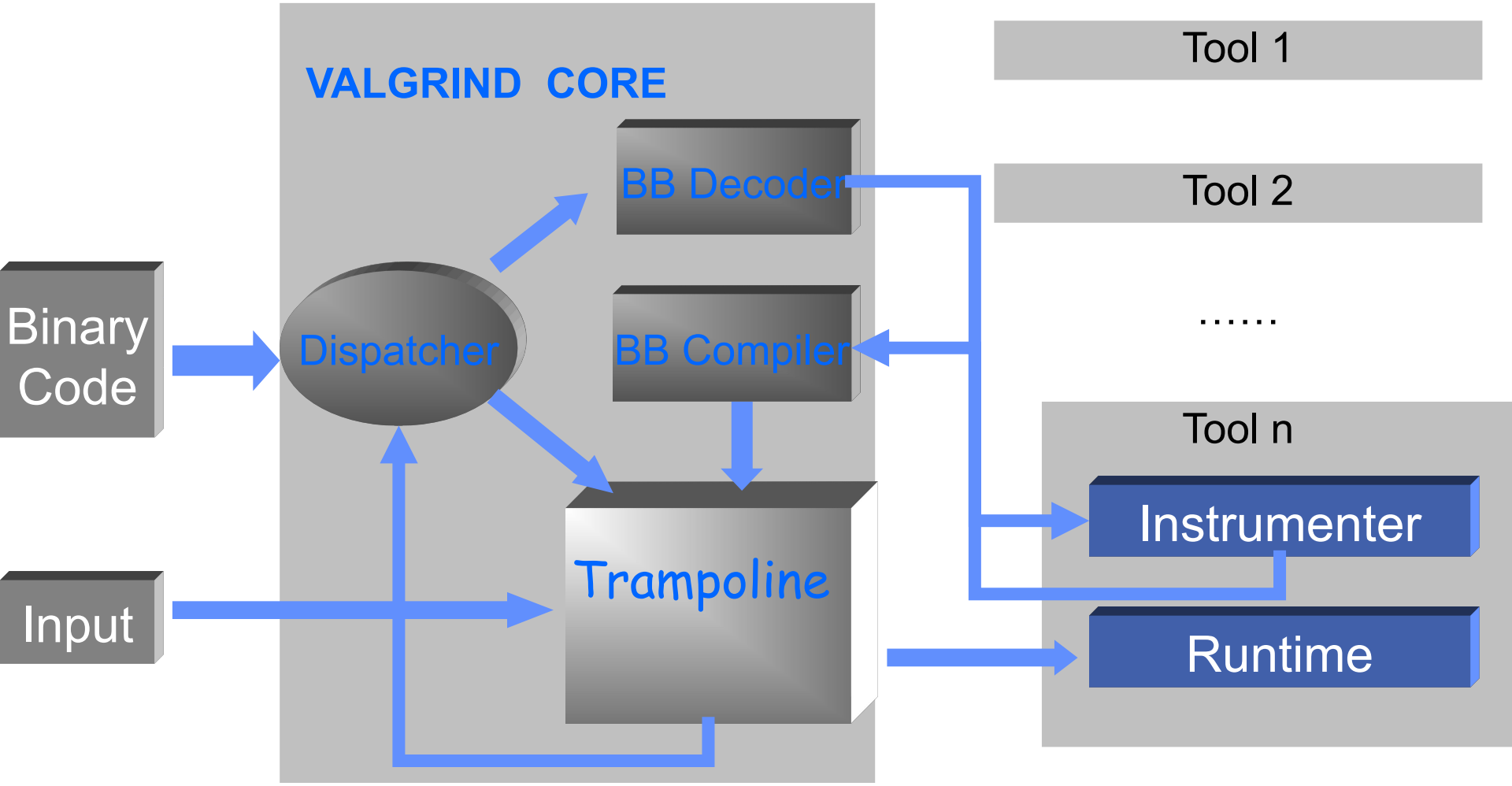
Information Flow System

- Dynamic Instrumentation
  - Difference from hooking
- What is IFS
  - A dynamic system that tracks the propagation of information.
    - E.g. a value is secret, what are the set of values in other places that are also secret.
- IFS is important
  - Confidentiality at runtime = IFS
  - Taint analysis = IFS
  - Memory reference errors detection = IFS
  - Data lineage system = IFS
  - Data dependence detection in dynamic slicing = IFS
- An IFS implemented on Valgrind.

Language and Abstract Model

- Valgrind is in essence a virtual machine that uses:
  - just-in-time (JIT) compilation
  - dynamic recompilation
- Intermediate Representation (VEX IR)
  - RISC Based
- Abstract state
  - One bit/byte, the security bit (taint bit).
    - For each variable (each memory location), we maintain one bit information.
  - Prevent call at tainted value.

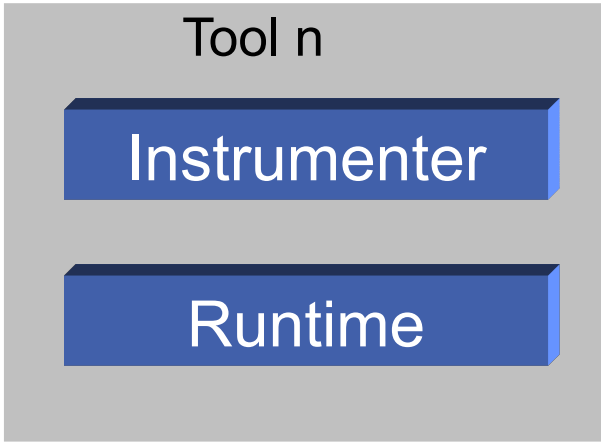
Valgrind Infrastructure



OUTPUT:

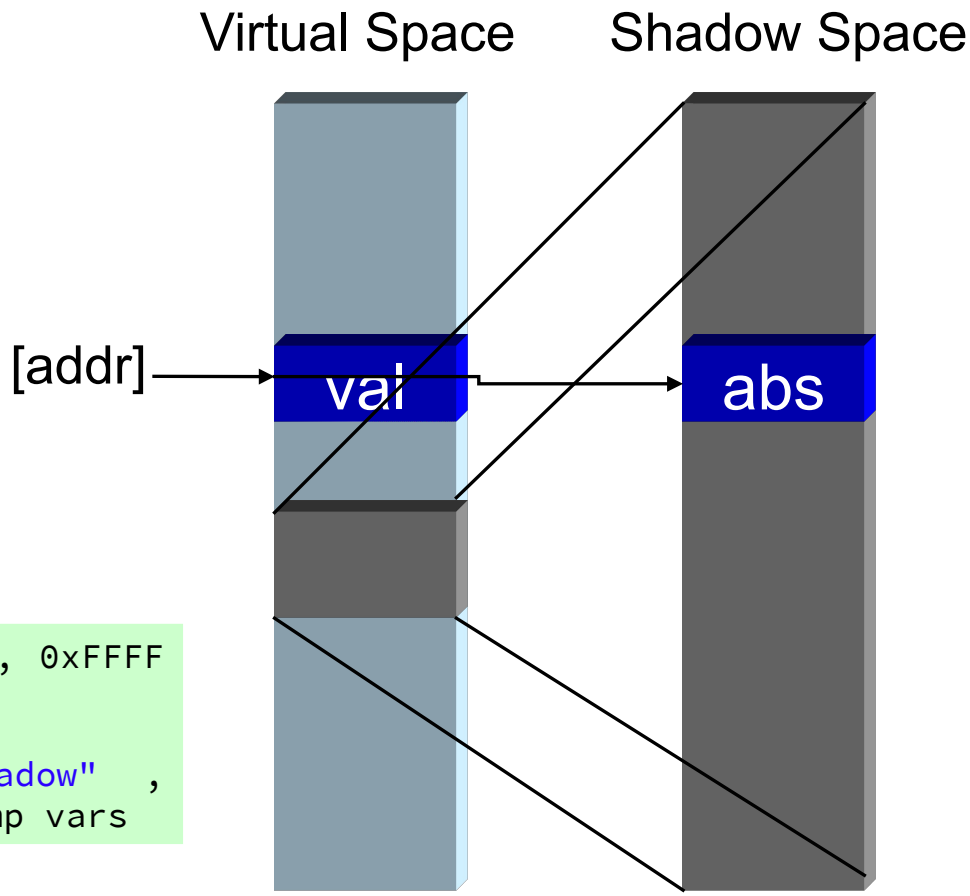
Implement A New Tool In Valgrind

- Use a template
  - The tool lackey is a 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
  - Three types of shadow memory (for mapping)
    - Mem. Addr → Abstract State
    - Registers → Abstract State
    - Temp → Abstract State
  - Registers can be handled by Valgrind utilities

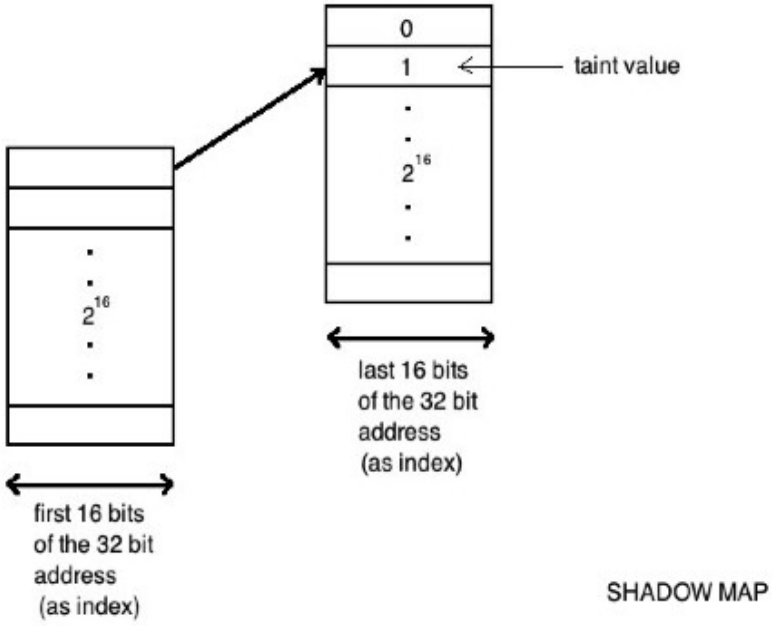



```
table = (Bool**)VG_(malloc)("Memory shadow", 0xFFFF
* sizeof(Bool*));

tempshadow  =  (Bool*)VG_(malloc)("Temp  shadow"  ,
0xFFFF * sizeof(Bool)); //64K limits for temp vars
```

How to Store Abstract State

Valgrind provides API for *shadow registers*.

- VG (get shadow regs area)
  - VG (get shadow regs area)




```
table = (Bool**)VG_(malloc)("Memory shadow", 0xFFFF * sizeof(Bool*));

tempshadow = (Bool*)VG_(malloc)("Temp shadow" , 0xFFFF * sizeof(Bool));


        for(i=0; i<0xFFFF; i++){ //initialization
                table[i] = NULL;
                tempshadow[i] = False;
        }
```

Acccessing Shadow Memory for Memory

```
static Bool get_shadow_mem(Addr addr){
        Int up = (((addr)&(0xFFFF0000)) >> 16);
        Bool* lookup = table[up];
        if(lookup == NULL)
                return False;
        else{
                Int low = (addr)&(0x0000FFFF);
                return lookup[low];
        }
}

static void set_shadow_mem(Addr addr, Bool value){
        //VG_(printf)("set_shadow_mem %x %d \n" , addr, value);
        Int up = (((addr)&(0xFFFF0000)) >> 16); //first level of indirection
        Bool* lookup = table[up];
        if(lookup == NULL){ //on-demand allocation!
          table[up] = (Bool*)VG_(malloc)("Memory shadow", 0xFFFF * sizeof(Bool));
        }
        else{
                Int low = (addr)&(0x0000FFFF);
                lookup[low] = value;
        }
}
```

Accessing Shadow Memory for Temp

```
static Bool get_shadow_temp(IRTemp temp){
         return (temp==-1)?False:tempshadow[temp];
}

static void set_shadow_temp(IRTemp temp, Bool value){
         tempshadow[temp] = value;
}
```

Initialization

```
VG_DETERMINE_INTERFACE_VERSION(ta_pre_clo_init)

static void ta_pre_clo_init(void)
{
         VG_(details_name)             ("Taint analysis");
         VG_(details_version)          (NULL);
         VG_(details_description)      ("the minimal Valgrind tool");
         ...
         VG_(details_bug_reports_to)   (VG_BUGS_TO);

         VG_(basic_tool_funcs)         (ta_post_clo_init,
                                        ta_instrument,
                                        ta_fini);

         VG_(needs_syscall_wrapper)(ta_pre_call, ta_post_call);

         //Abstract memory definition and initialization
}
```

Finalization

```
static void ta_fini(Int exitcode)
{
        //Reporting Statistics/collected information if needed
        //Clean up allocated memory

}
```

Instrumentation & Runtime

```
static IRSB* ta_instrument ( VgCallbackClosure*
                             closure,
                             IRSB* sbIn,
                              …)
{
        …
        IRSB* sbOut  = deepCopyIRSBExceptStmts(sbIn);
        …
        for (i = 0; i < sbIn->stmts_used; i++) {
         IRStmt* st = sbIn->stmts[i];
         if(!st) continue;
         switch (st->tag) {
               case Ist_Put:
                                        …
               case Ist_Store:
                  …
               case Ist_PutI:
                  …
               case Ist_WrTmp:
                  …
              default: addStmtToIRSB(sbOut, st);
       return sbOut;
}
```

* Please consult the following header file for details of the valgrind VEX instructions:

<Valgrind>/VEX/pub/libvex_ir.h

Ist_IMark

```
case Ist_IMark: //Specifies the beginning of a machine instruction
// Start tracing in main
//VG_(printf)("Ist_IMark\n");
if (VG_(get_fnname_if_entry)(st->Ist.IMark.addr, fnname, sizeof(fnname))) {
        if(VG_(strcmp)(fnname, "main") == 0) {
                trace = True;
        }
        else if(VG_(strcmp)(fnname, "exit") == 0) {
                trace = False;
        }
}
```

Instrumentation & Runtime - Ist_Put

```
case Ist_Put: //puts value of temporary variable or constant into the register.
//Write into a guest register, at a fixed offset in the guest state.
if(trace) { //to start tainting from main()
        offset = st->Ist.Put.offset; // Look at: <Valgrind>/VEX/pub/libvex_ir.h
        data = st->Ist.Put.data;
        //VG_(printf)("Ist_Put offset %d data %d\n", offset, data);

        argv = mkIRExprVec_2( mkIRExpr_HWord((HWord)offset),
mkIRExpr_HWord( (HWord) (data->tag == Iex_RdTmp)?(data->Iex.RdTmp.tmp):-1));

        dirty = unsafeIRDirty_0_N( 2, "ta_put", VG_(fnptr_to_fnentry)
(ta_put),argv ); //constructing dirty helper calls
        addStmtToIRSB( sbOut, IRStmt_Dirty(dirty) );
}
addStmtToIRSB( sbOut, st );
break;
```

```
static VG_REGPARM(2) void ta_put(Int offset, Int tmp) {
//VG_(printf)("ta_put %d %d\n", offset, tmp);
ThreadId tid = VG_(get_running_tid)();
Bool shadow_reg = get_shadow_temp(tmp); //If tmp is tainted then propagate
VG_(set_shadow_regs_area) ( tid, 1, offset, 1, &shadow_reg);
}
```

```
case Ist_WrTmp: //Assign a value to a temporary.
tmp = st->Ist.WrTmp.tmp;
data = st->Ist.WrTmp.data;
//VG_(printf)("Ist_WrTmp tmp %d datw %x\n", tmp, data);
switch(data->tag) {
          switch(data->tag) {
                    case Iex_RdTmp:
                    argv = mkIRExprVec_2( mkIRExpr_HWord((HWord)tmp),
mkIRExpr_HWord( (HWord)data->Iex.RdTmp.tmp));
                    dirty = unsafeIRDirty_0_N( 2, "ta_wrtmp_const", VG_(fnptr_to_fnentry)
(ta_wrtmp_rdtmp), argv );
                    addStmtToIRSB( sbOut, IRStmt_Dirty(dirty) );
                    break;
                    case Iex_Const:
                    ...
                    case Iex_Load:
                    ...
                    case Iex_Get:
                    ...
                    case Iex_Binop:
                    arg1 = data->Iex.Binop.arg1;
                    arg2 = data->Iex.Binop.arg2;
                    argv = mkIRExprVec_3( ...);
                    dirty = unsafeIRDirty_0_N( ..., "ta_wrtmp_binop", ... );
                    addStmtToIRSB( sbOut, IRStmt_Dirty(dirty) );
                    break;
                    case Iex_Triop:
                    ...
          }
}
```

Instrumentation & Runtime – SYS CALLs

```
VG_(needs_syscall_wrapper)(ta_pre_call, ta_post_call); //to be notified
```

```
static void ta_post_call(ThreadId id, UInt syscallno, UWord* args, UInt nargs)
{
        if(trace){
                if(syscallno == __NR_read){
                        // Read system call
                        VG_(printf)("read %x %d\r\n", args[1], args[2]);
                        Int i;
                        for(i=0; i< args[2]; i++){
                                set_shadow_mem(args[1]+i, True);
                        }
                }
        }
}
```

Instrumentation & Runtime - CALL

```
static VG_REGPARM(3)
void RT_Exit(OpAddr* guard1, HWord guard2, Addr dst) {
  // do not need to check taint types since taint bits
  // subset taint type
  if (get_shadow_mem(dst)) {
    VG_(printf)("Call address tainted.\n");
    VG_(exit)(1);
  }
```

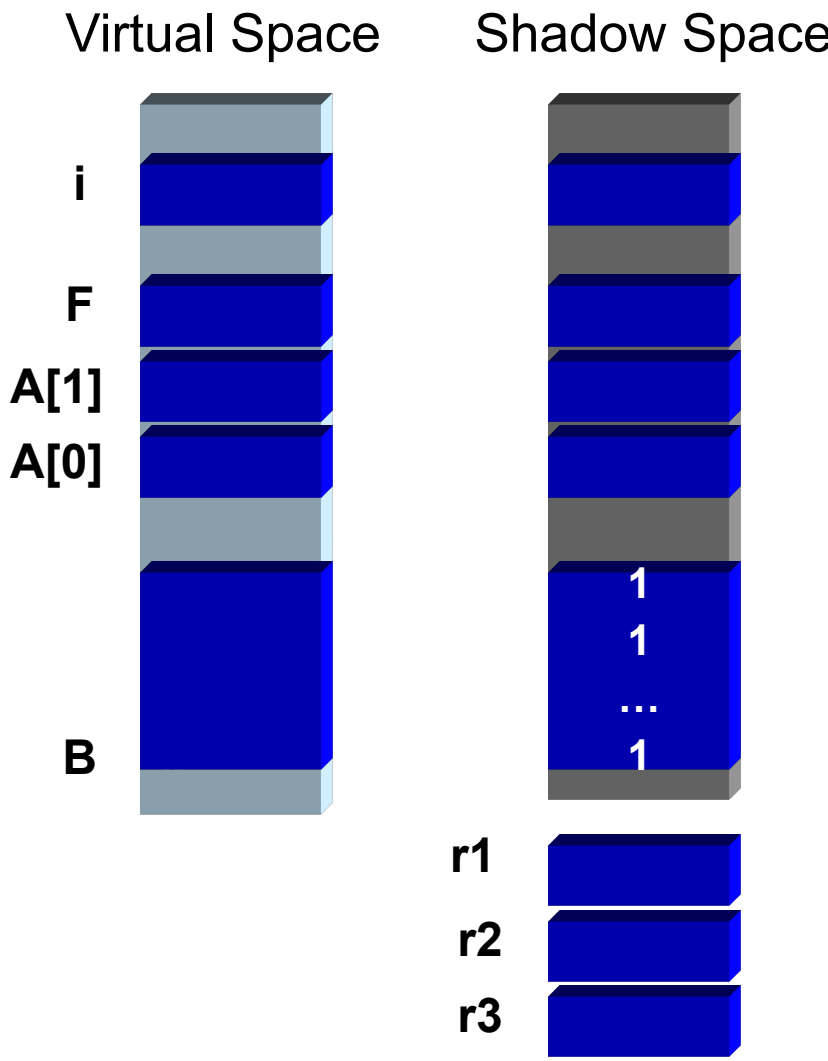
Done!

- Let us run it through a buffer overflow exploit

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

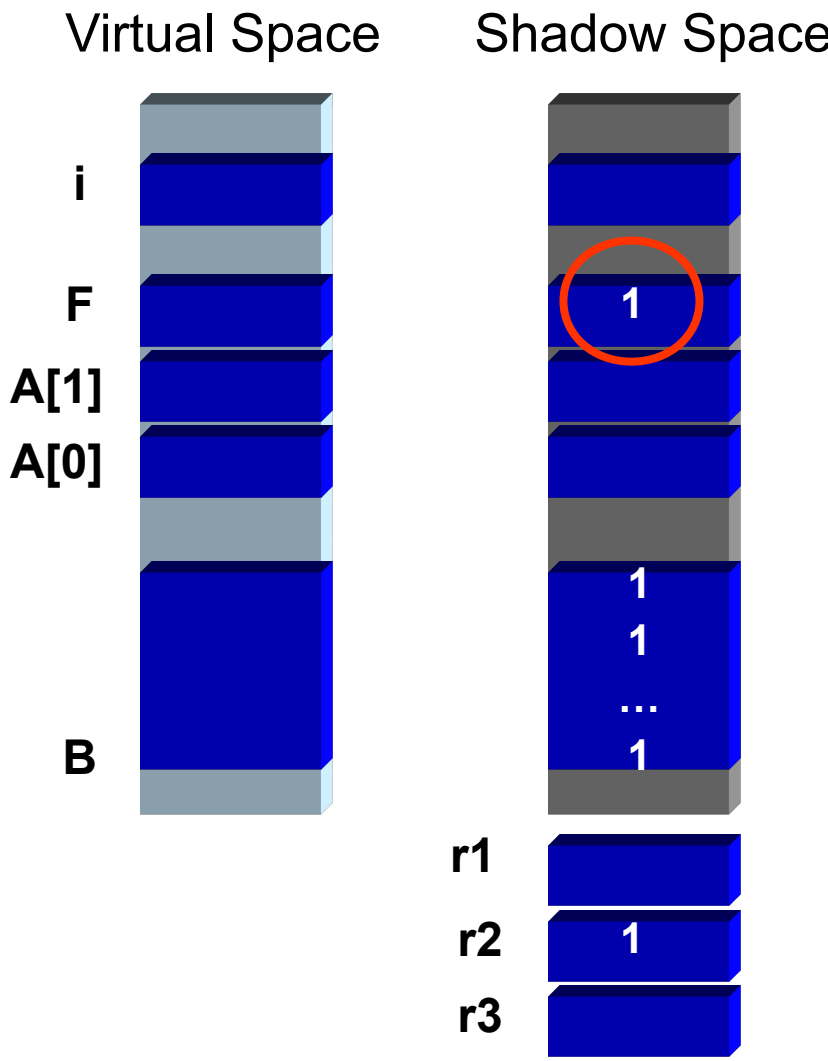
```
void (* F) ();
char A[2];
...
read(B, 256);

...

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
```



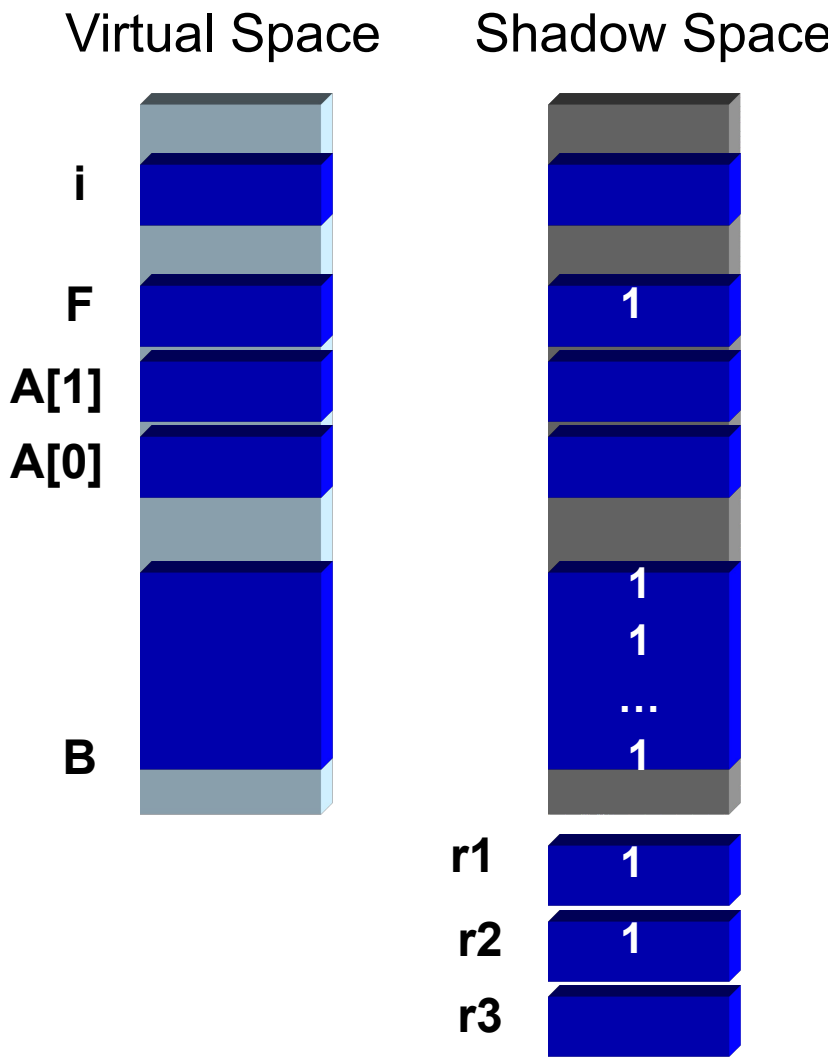
```
void (* F) ();
char A[2];
...
read(B, 256);

...

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
```



```
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
```



Final Notes

- Completeness of tainting?
- Information flow through control dependence
- Untainting