Enforcing C-level security policies using machine-level tags

Andrew Tolmach
Sean Anderson
Chris Chak
Portland State University

Cătălin Hriţcu, INRIA Paris
Benjamin Pierce, Univ. of Pennsylvania

DARPA/Draper Labs SSITH/Hope Project

INRIA Seminar 12 April 2019

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-18-C-0011. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

HW for better SW security

- Problem space: security for legacy software...
  - Especially in unsafe languages like C/C++
- ...using hardware support for reference monitors...
  - Processor extensions to handle metadata
- ...with a new focus on higher-level properties
  - Expressed in source-language terms
  - Extending beyond simple safety
  - Trusting or verifying the compiler

Outline

- Reference monitors and the PIPE
- ISA-level Tag-based policies
- C-level policies
- Semantics and implementation
- Conclusions

Reference Monitors
and the PIPE

Reference Monitors

- Explicitly check every potentially dangerous operation
  - e.g. "is this memory reference in bounds?" "is this top-secret value being written to an insecure channel?" "is this a valid address to jump to?"
- Useful when we don't trust program and/or programming language
- Works for any safety property ("this bad thing does not happen")
  - Doesn't work for liveness properties ("this good thing does happen")
- Gives "fail-stop" behavior
  - No direct support for recovering and continuing

Examples of safety policies

- "No secret data leak to a public output channel."
- "Every store operation occurs to a valid memory location."
- "The processor only jumps to places the programmer intended."
- "Code modules communicate only via their specified interfaces."

Example: confidentiality

- Data comes from secret or public sources
  - `x = read_secret(); y = read_public();`
- Any result of operating on secret data should also be treated as secret
  - `z = x + y; // z is secret ⇔ x or y is secret`
- Attempting to output secret data to a public channel should halt the program
  - `write_public(z);  // halt if z is secret`

Example: compartments

- Code is divided into compartments with explicit interfaces
- Inter-compartment access is mediated according to access control matrix
- Attempting an invalid access should halt the program

```
int foo@A[5]; // compartment A
int bar@A(int i) { // compartment A
    return foo[i];
}
int main@B() { //compartment B
    return bar(3);
}
```

	Main	Foo	Bar
A	—	Read, Write	Call
B	Call	—	Call

Example: memory safety

- Each pointer is associated with a precise region of memory
- Attempting to use a pointer to access outside its region or to access a deallocated region should halt the program
- In C/C++ this protects against one class of "Undefined Behaviors" (UB)
- Safe languages normally enforce this in software

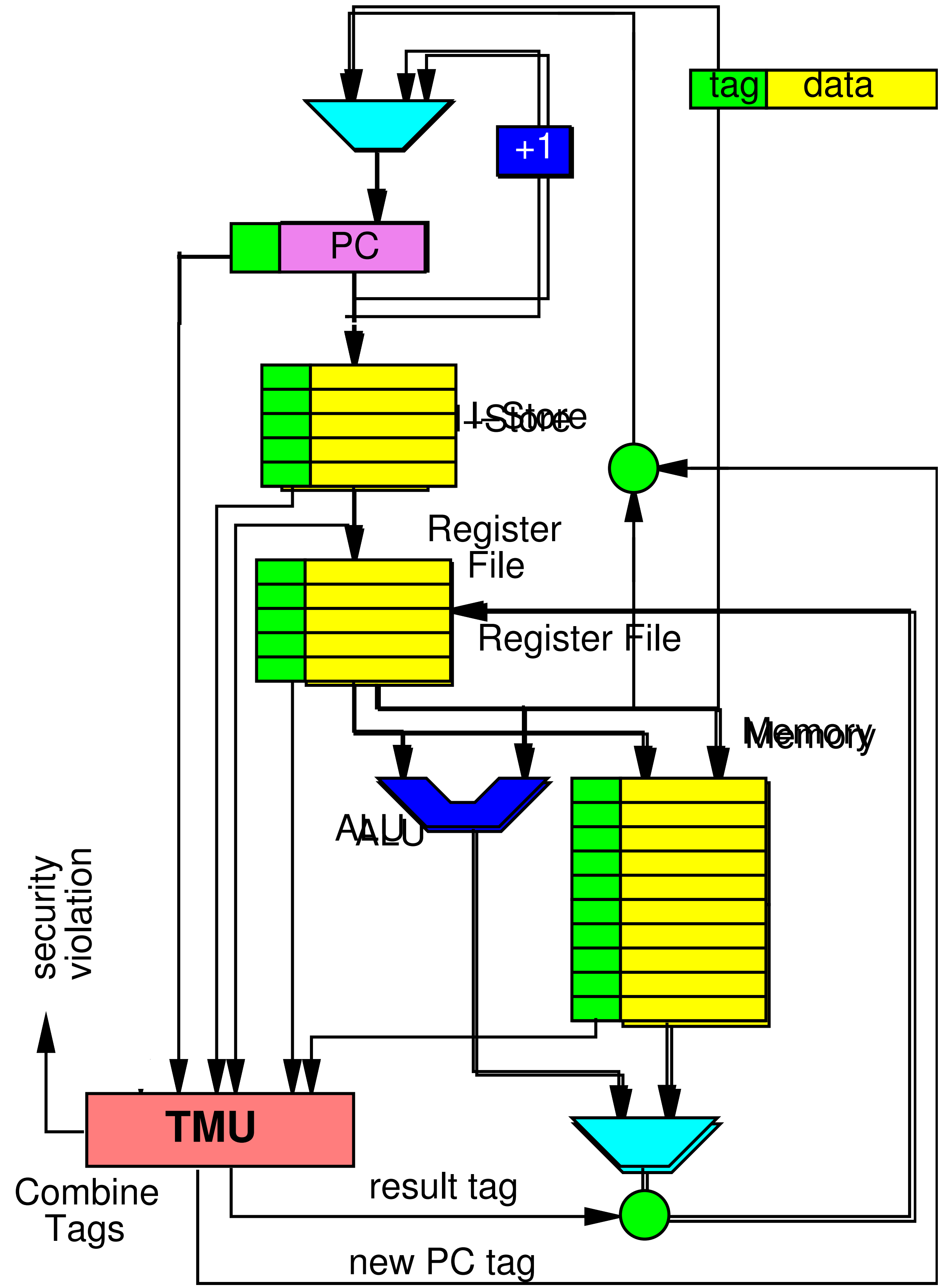
Monitoring in Hardware

- Implementing monitors in software can be very slow
- So, let's use silicon to improve security, not just performance
- Perform monitoring in parallel with normal execution, to avoid adding delay
- Checks occur at machine level, so cannot be evaded by buggy or malicious software
- Challenge: how to make hardware flexible enough to handle changing threats

Simplified PIPE Architecture

Typical RISC CPU

- \+ large tag on every word
- \+ tag management unit (rule cache)



(costs extra ~100% in area and ~50% in power)

Tag Management Unit

acts like a cache

key

(opcode,
pc-tag,
instruction-tag,
register-operand1-tag,
register-operand2-tag,
memory-operand-tag)

→

result

(new-pc-tag,
result-tag)

if key is not present, control traps to tag miss handler

Tag Miss Handler

- Ordinary machine code that lives at special location in OS (or runs on a special co-processor)
- Takes missing key as input
- Executes tagging decision algorithm
  - Hardware is completely independent of this algorithm
- EITHER generates result tags & stores in TMU cache
  - Instruction that faulted is then restarted
- OR discovers security violation and fail-stops the process (or whole processor)

PIPE performance

- Runtime cost depends on cache hit rate
- Varies widely for different choices of policies and program patterns
- Simulations using SPEC2006 benchmarks enforcing a fairly rich composite policy show <10% added runtime for most programs
- Keeping number of "live" tags low is essential
  - Also important for fault handler to run fast

Tag-based Policies

Anatomy of a policy

- Set of tags for labeling registers, memory, PC
  - Can be discrete symbols, numbers, or addresses pointing to arbitrarily complicated data structures
- Rules for checking and propagating tags as the machine executes each instruction
  - Rules can be arbitrarily complex and may maintain a persistent internal database
  - But they must be monotonic
- Initial configuration
  - Tags on memory contents; state of tag rule database

Policies are written in a domain-specific language

IFC Policy

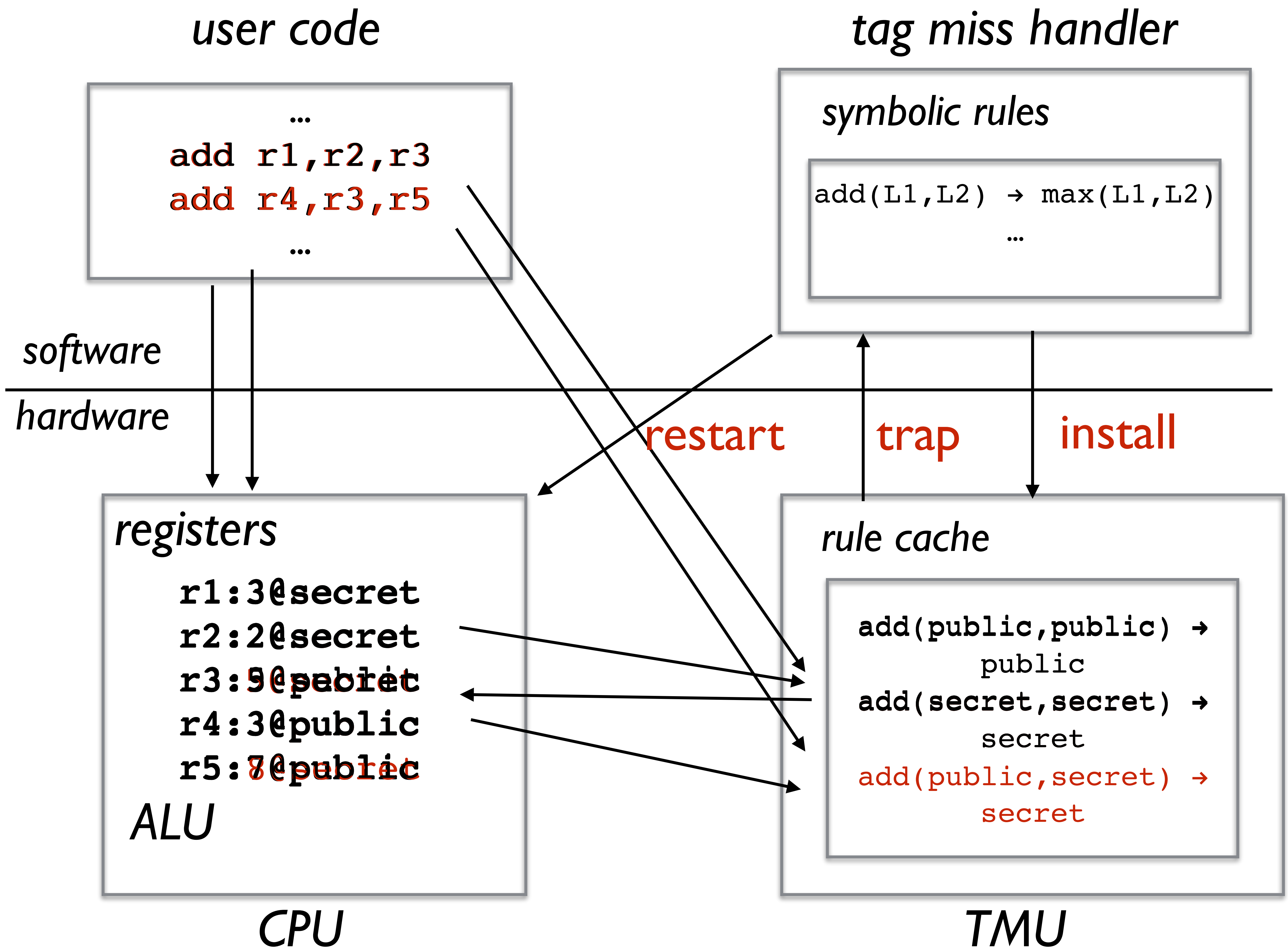
- Goal: ensure that secret data does not flow to public output
  - or, dually, ensure that public input data does not **taint** private data
- Tag = Value from security lattice, e.g.

  Secret
  |
  Public

  - attached to each data value
  - also attached to PC to record **implicit** flows
- Rules:
  - Computations produce result value tagged with join of argument value tags and PC tag
  - Conditional tests raise PC to join of argument values
  - Public values cannot be generated when PC is tagged secret

IFC example

```
secret > public
```



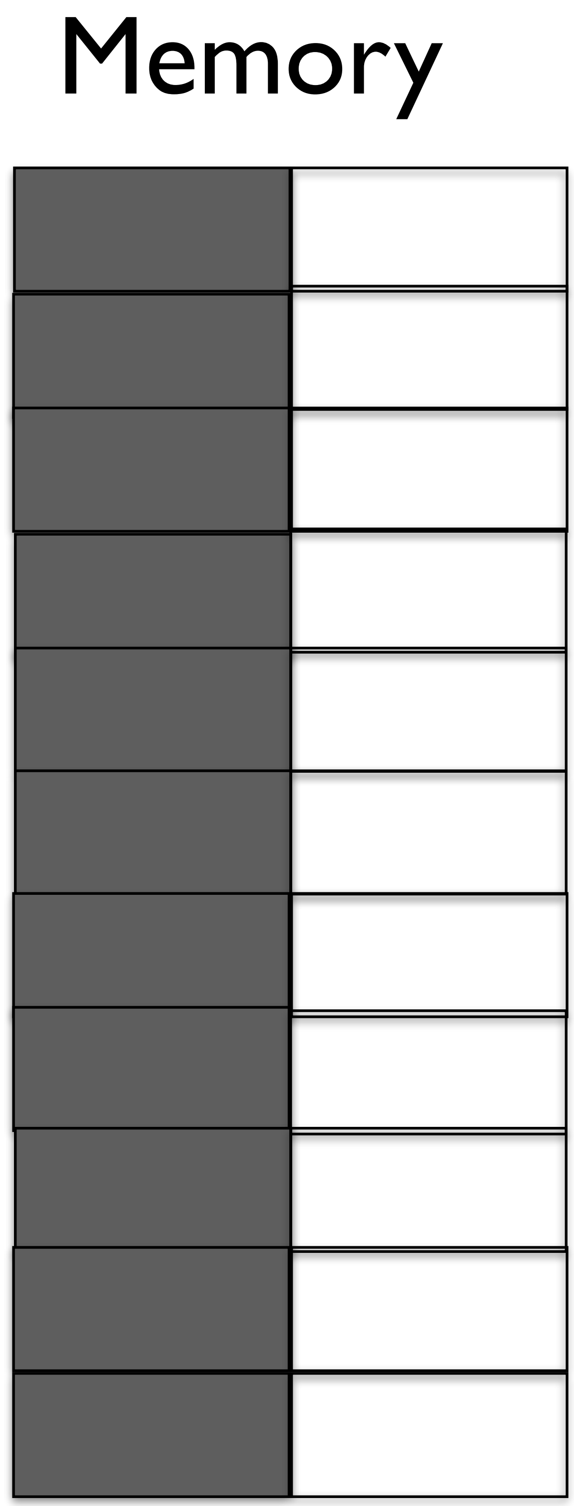
Example: Static Compartments

- Goal: Divide process memory into set of disjoint compartments which are protected from each other
  - Code in one compartment can jump or write to other compartments only at a pre-defined set of addresses (an interface)
- Tags = compartment ID or set of compartment IDs
  - PC is tagged with current compartment
  - Each memory location is tagged with set of compartments that can validly access it
- Rules:
  - On each write and after each branch, compare PC tag with tag of memory location being written or executed

Example: Heap Memory Safety

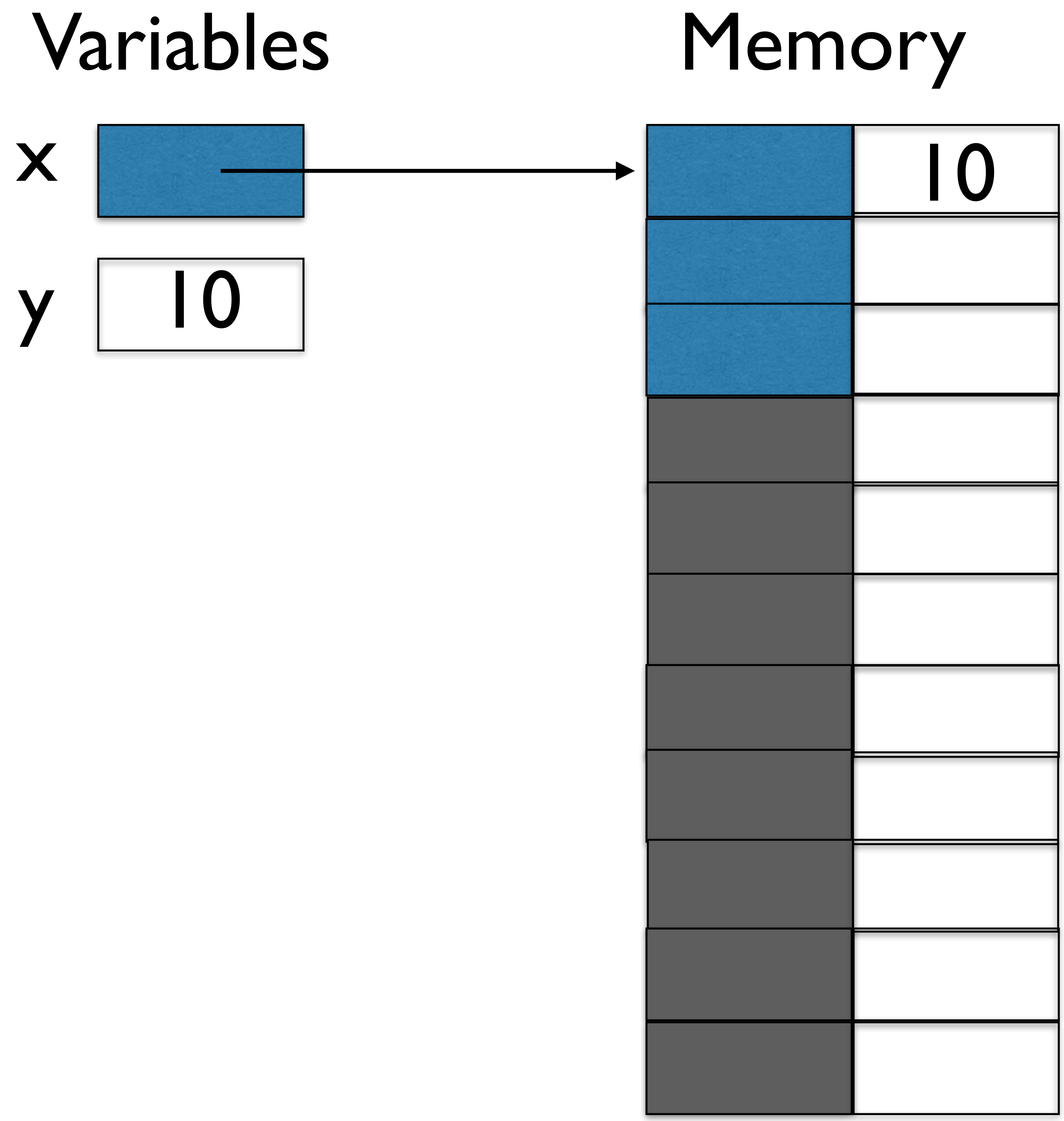
- Goal: prevent heap buffer overflows
- Tags = ValueTag | Cell(region#, ValueTag)
  where ValueTag = NonPtr | Ptr(region#)
  - Each call to malloc generates a fresh integer region# tag c -- a "color"
  - Pointer to new region is tagged Ptr(c)
  - All other values are tagged NonPtr
  - Values in newly-allocated memory cells are tagged with Cell(c,v) where v is tag of value in cell
- Rules:
  - Load and store instructions check that address pointer is tagged Ptr(c) and the referenced memory cell is tagged Cell(c,_)
  - Pointer arithmetic instructions preserve Ptr(c) tags

Heap Memory Safety Example



Heap Memory Safety Example

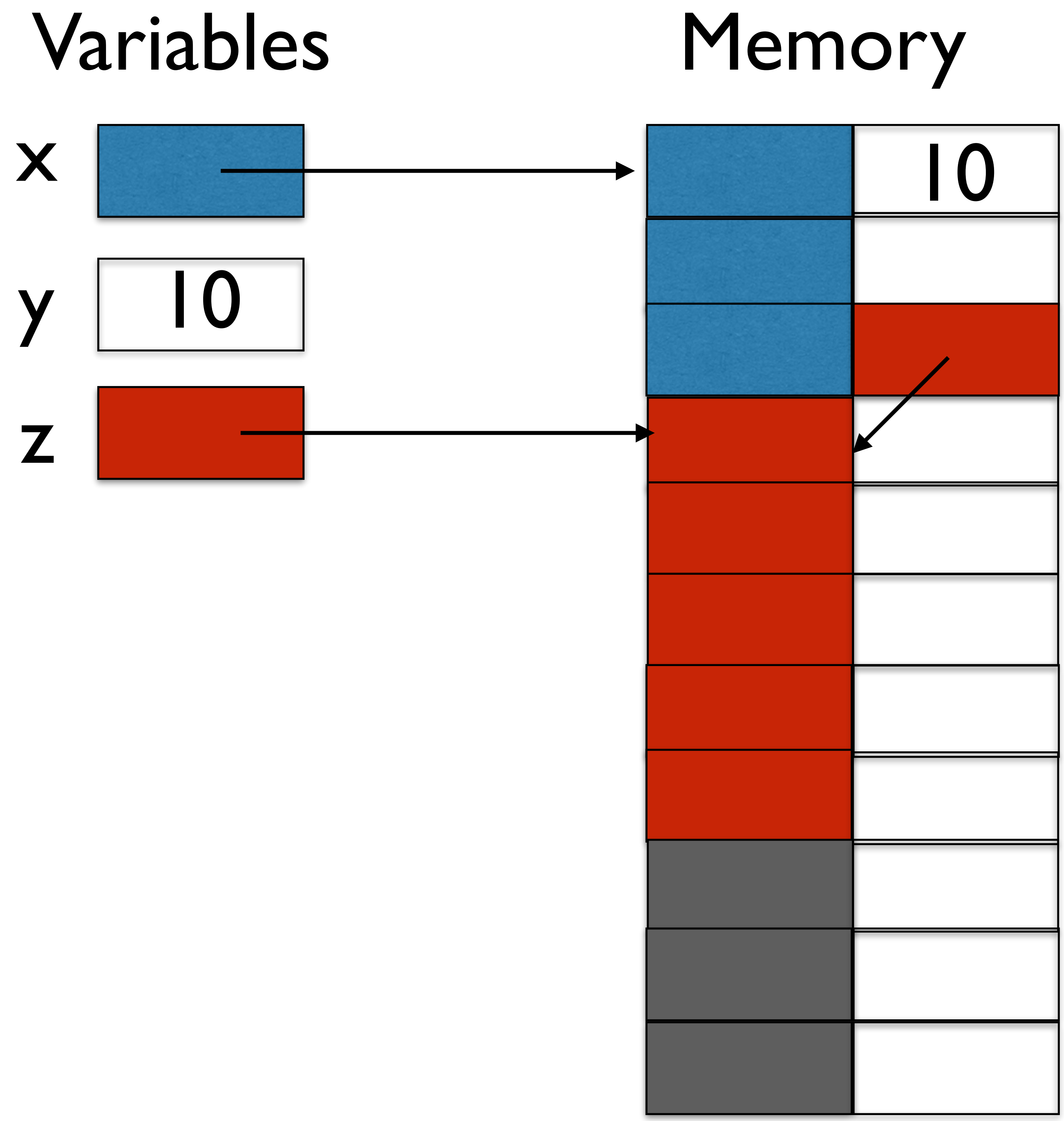
```
int *x = malloc(3)
y = 10
x[0] = y
```



Heap Memory Safety Example

```
int *x = malloc(3)
y = 10
x[0] = y
```

```
int *z = malloc(5)
x[2] = (int) z
```

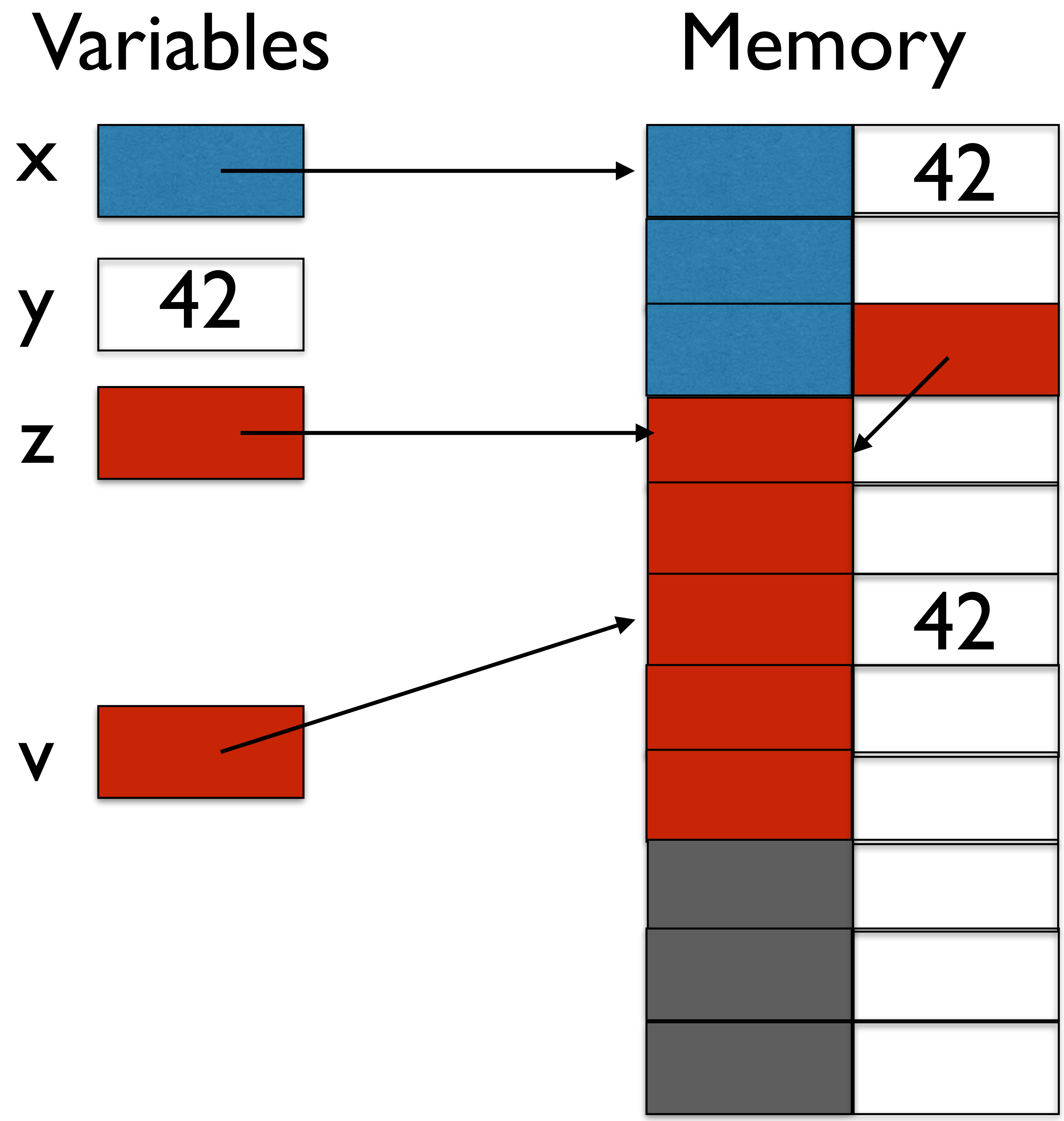


Heap Memory Safety Example

```
int *x = malloc(3)
y = 42
x[0] = y
```

```
int *z = malloc(5)
x[2] = (int) z
```

```
int *v = x[2] + 2
v[0] = y
```



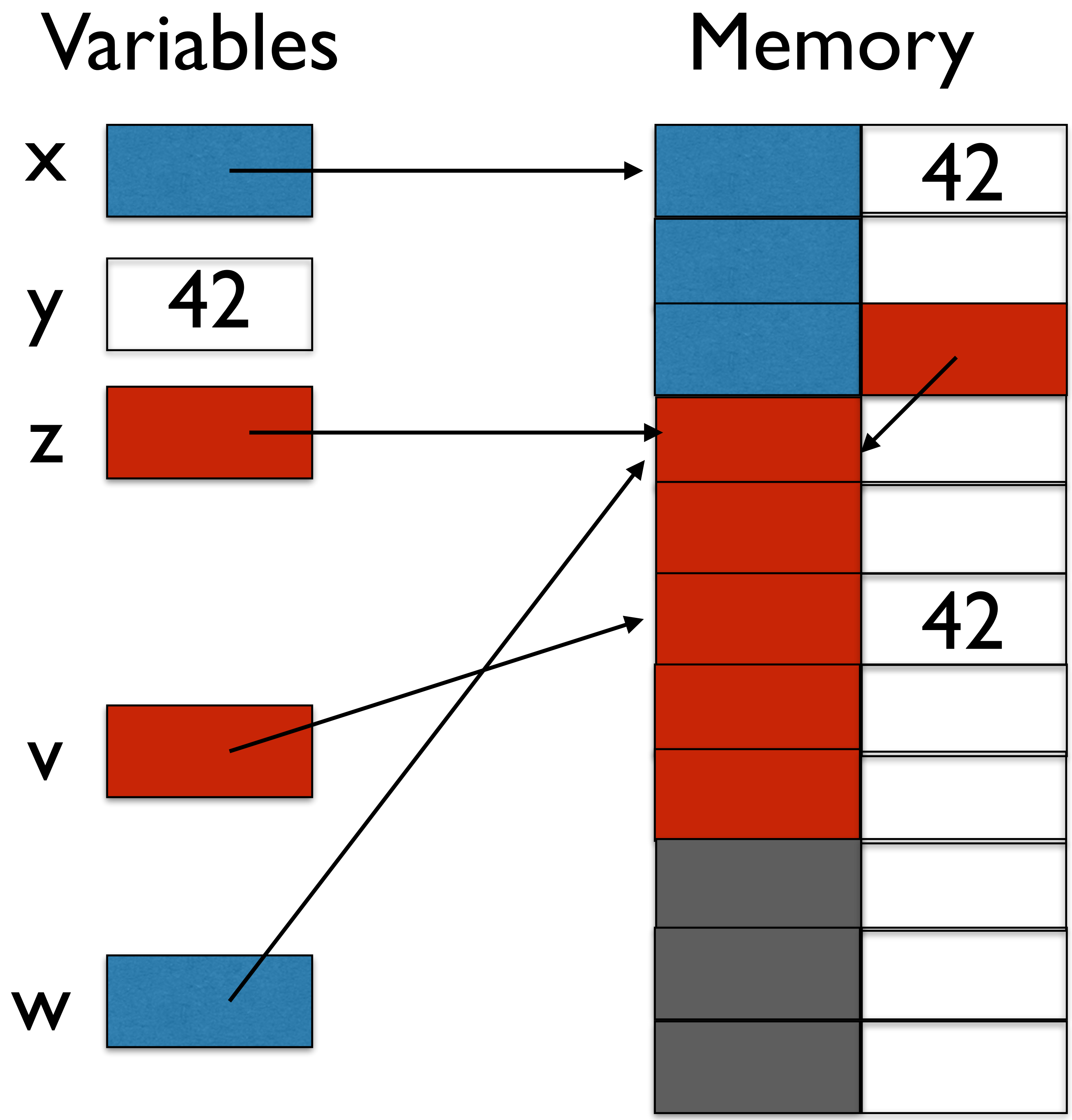
Spatial Safety

```
int *x = malloc(3)
y = 42
x[0] = y
```

```
int *z = malloc(5)
x[2] = (int) z
```

```
int *v = (int *) x[2] + 2
v[0] = y
```

```
int *w = x + 3
w[0] = y
```



Temporal Safety

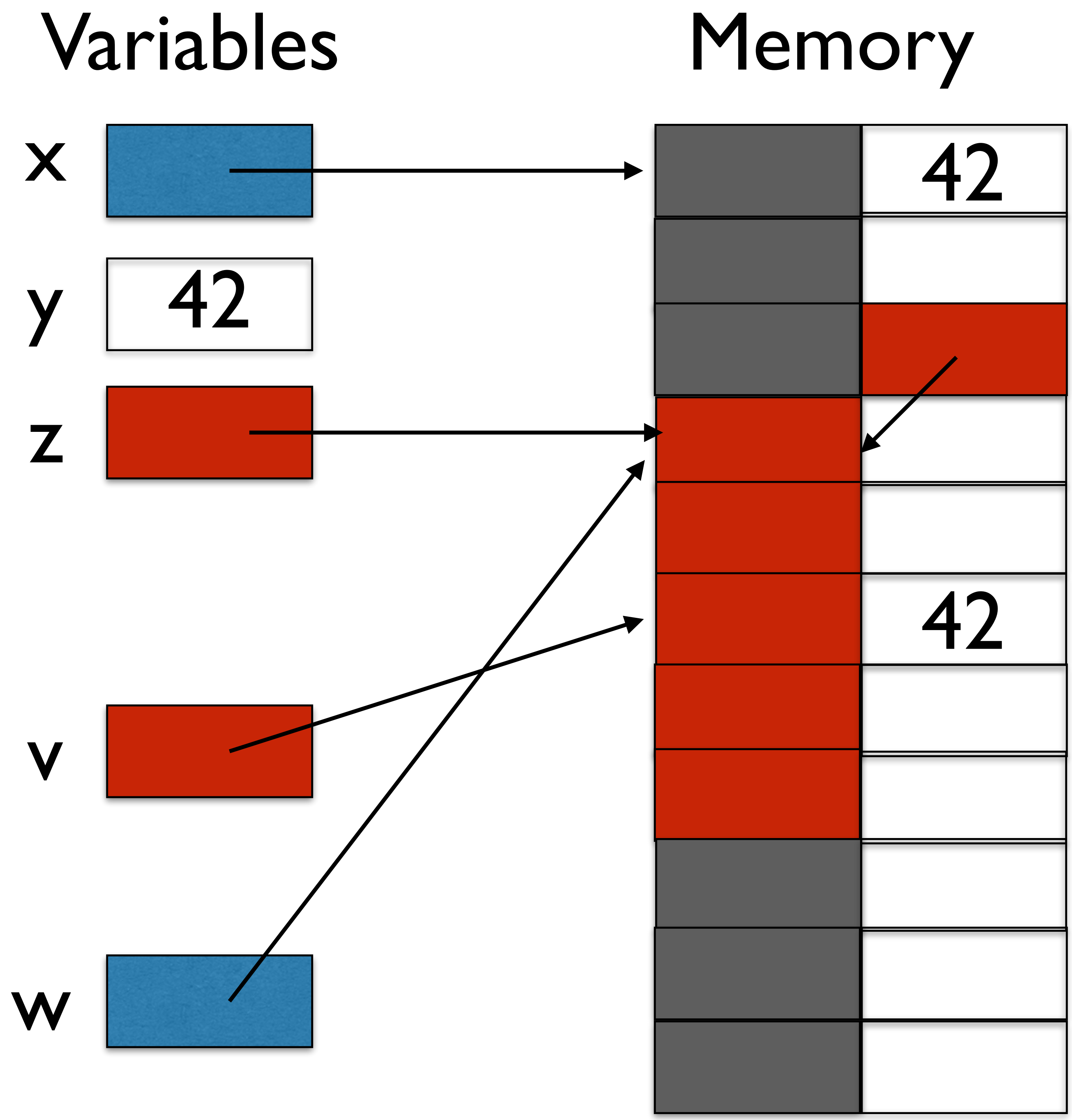
```
int *x = malloc(3)
y = 42
x[0] = y
```

```
int *z = malloc(5)
x[2] = (int) z
```

```
int *v = (int *) x[2] + 2
v[0] = y
```

```
int *w = x + 3
w[0] = y
```
✗

```
free(x)
x[0] = 100
```
✗



Composing policies

- In practice we want to compose policies
- Many policies are essentially orthogonal
  - e.g. A = Memory safety and B = IFC
  - Make tags be pointers to pairs (Atag,Btag)
  - Operations are allowed only if both policies say OK
- But others are not…
  - e.g. A = Memory safety and B = Compartments
  - because memory tags for these must be coherent
- General theory is a subject of ongoing research

C-Level Policies

Problem and Opportunity

- It is hard to specify and enforce some safety policies at hardware ISA level
  - No typing information
  - No structured control flow
  - Function boundaries and calling conventions may be obscured
- How about working directly at the level of C code instead?
  - Tie tag-based monitoring to C code execution points
  - Avoid reverse engineering of compiled code
  - Support specifying policies at higher level of abstraction

Approach

- Use tagging rules during C execution
  - Add monitoring/control points to C semantics
  - Customize by per-system or per-program rules
  - Compile to ISA-level tags for runtime enforcement
- Express high-level policies with rules
  - Fine-grained information flow control
  - Compartment enforcement and access control
- Combine with fixed base policy
  - Trap C undefined behaviors on pointers

Example: IFC in more detail

- Goal: prevent leakage of high-security information
- Tags = Security labels from a lattice
  - Initial memory values and pointers are labelled
  - PC carries "current" label

e.g.

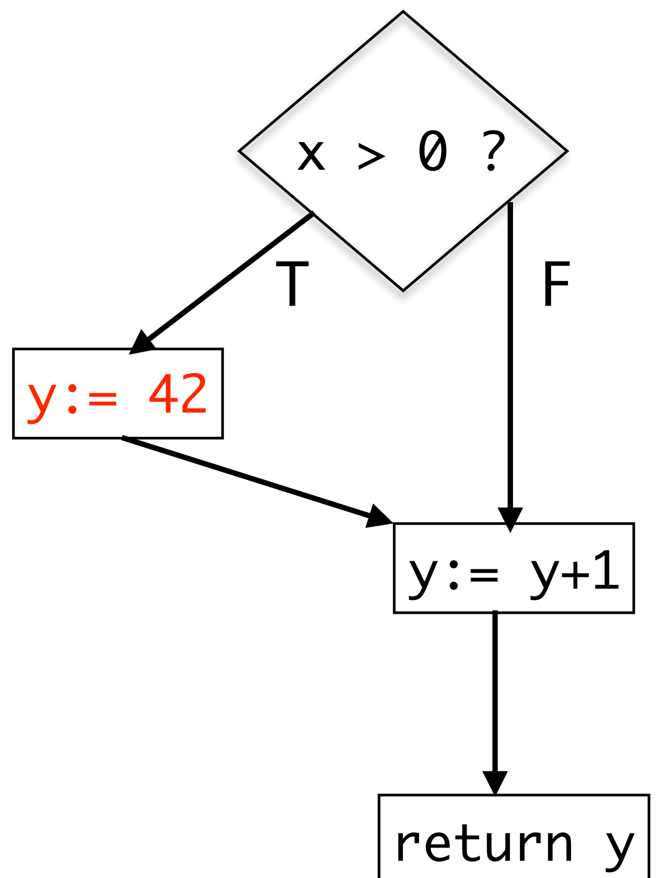
Secret
|
Public

- Rules:
  - Instructions that move values propagate labels
  - Binary operations compute lattice join of labels
  - Conditional jumps raise PC label level based on inspected val
  - "No sensitive upgrade" — stores are prevented if PC is higher than old value, thus avoiding "implicit flows"

IFC: example program

```
int f (int x, int y) {   // assume x value secret, y value public
  if (x > 0)
     y = 42;             // bad: sensitive upgrade
  y = y + 1;             // ok: not under control of secret
  return y;
}
```

Implicit flow if public return in y depends on secret x



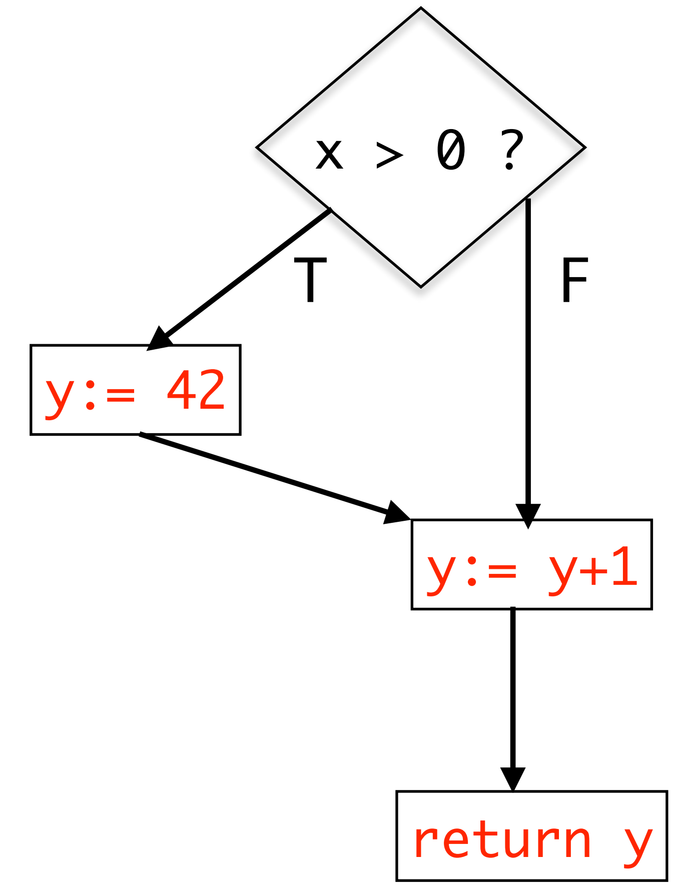
Calling `f (1,100)` should trigger tag violation

Calling `f (0,100)` should **not** trigger tag violation

Assumes that attacker cannot (efficiently) observe when tag violations occur (gives "error-insensitive non-interference")

ISA-level tagging: "label creep"

```
function f:          // args in r0, r1
0  : Mov    r0 r2   // fetch arg0
1  : Brnz   r2 3    // if arg0 > 0 goto 4
2  : Const 1  r6    // goto 6
3  : Brnz   r6 3    //
4  : Const 42 r5    // get 42
5  : Mov    r5 r1   // store into arg1
6  : Mov    r1 r7   // fetch arg1
7  : Const 1  r8    // add 1
8  : Add    r8 r7   //  to arg1 value
9  : Mov    r7 r1   // store back in arg1
10 : Mov    r1 r9   // fetch arg1
11 : Ret    r9      // and return it
12 : Halt
```



brnz pc_tag arg_tag := OK (pc_tag ∨ arg_tag)

mov pc_tag src_tag tgt_tag :=
  if pc_tag ≤ tgt_tag then OK (pc_tag ∨ src_tag) else FAIL

Calling `f (1,100)` triggers tag violation ✔

Calling `f (0,100)` also triggers tag violation ✘

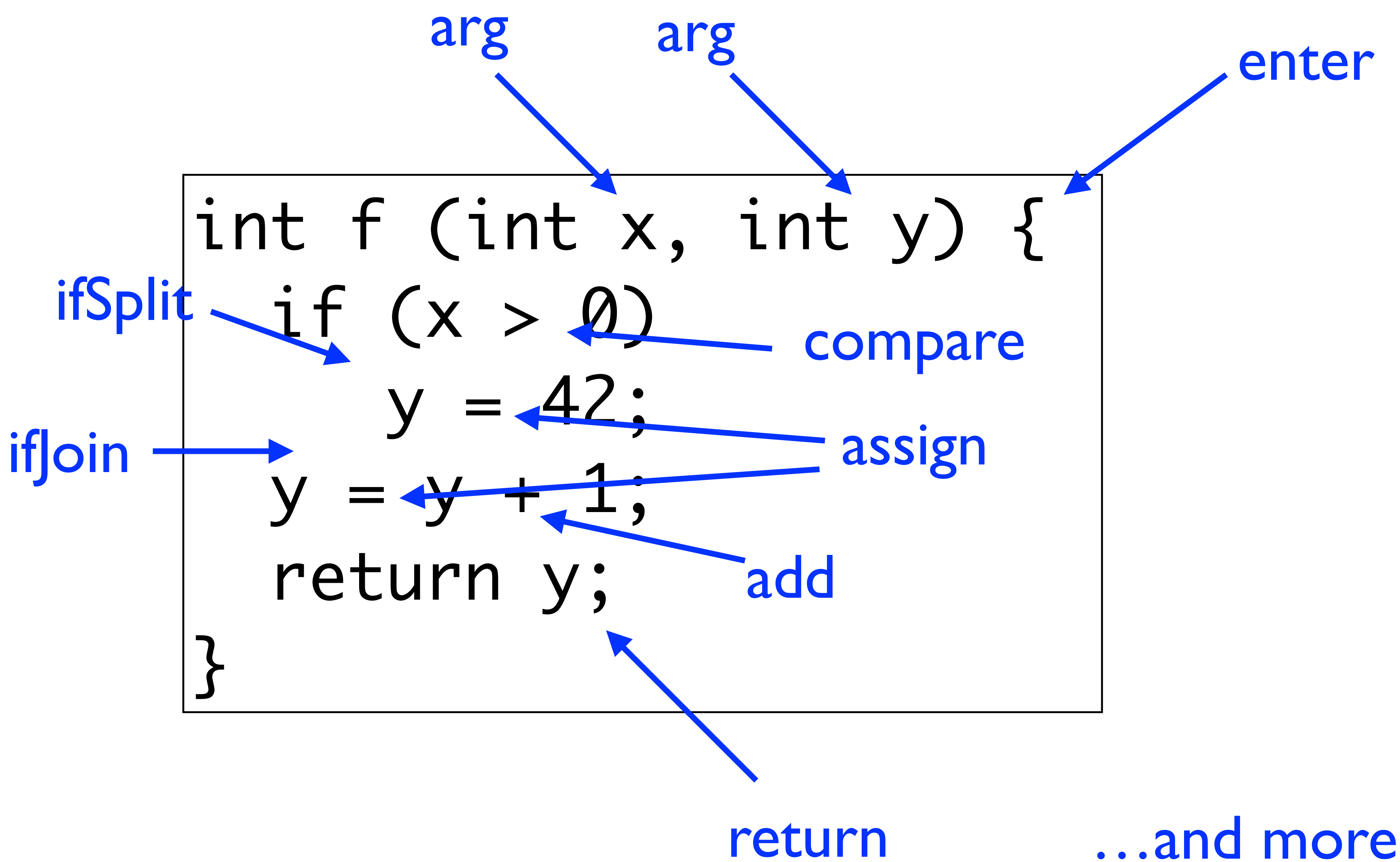
C-level tagging

- Express tagging policy at level of C expression operators and control structures, rather than of machine instructions
- Attach tags to C "program counter," values, memory locations (globals, malloc'ed heap records, …), functions, …
- Tag rules are invoked at fixed set of control points in C execution semantics
  - instead of at each instruction
  - similar to aspect-oriented programming "advice" points

Assumptions

- Access to C source code
  - But little or no ability to edit it
- All object code is produced by our custom compiler
  - Compiler is in TCB; we must trust or (better) verify it
- May want to "bake in" some policies
  - To guard against some C undefined behaviors

C-level control points




```
int f (int x, int y) {
   if (x > 0)
      y = 42;
   y = y + 1;
   return y;
}
```

Set of control points is defined once and for all for the C language.

C-level IFC

```
int f (int x, int y) {
  if (x > 0)
    y = 42;
  y = y + 1;
  return y;
}
```

ifSplit ifJoin assign

```
ifSplitT v_tag pc_tag := OK (v_tag v pc_tag)
ifJoinT v_tag old_pc_tag pc_tag := OK old_pc_tag
assignT pc_tag v_tag old_v_tag :=
   if pc_tag ≤ old_v_tag then
     OK (pc_tag v v_tag)
   else FAIL
```

rules are parameterized on tags of all relevant values and PC

PC tag is reset at join point, so second assignment is OK

Example: Compartments sharing data

- Goal: use interfaces to control how one compartment shares data with another
- Tags as in Memory safety policy, but with
  Value Tags = NonPtr | Pointer(a,c)
  where a is an access level = ReadOnly | ReadWrite
  and PC Tag = Function ID
- Auxiliary configuration data maps defines
  Compartments = sets of Function ID's
  and says which compartments grant which other compartments write access

Example program

```
void f@A (int *p) { *p = 1; }
void g@A ()       { int *q = malloc(1); f(q); }
void main@B ()    { int *r = malloc(1); g(); f(r); }
```

Assume access control configuration defines
`A = {f,g}, B ={h,main}`
and `A` has only read access to data allocated by `B`.

Store in first call to `f` (from g) should succeed;
store in second call (from main) should fail.

Rules:

- Each array malloc generates a fresh region "color" c and resulting pointer is tagged Pointer(ReadWrite,c)
- Memory writes require Pointer(ReadWrite,_); reads require only Pointer(_,_)
- On function calls, each pointer argument is potentially "downgraded" from ReadWrite to ReadOnly based on PC tags of caller, callee

Using fewer colors

- Should we really allocate a separate color for each malloc'ed buffer?
  - + Gives fine-grained control over sharing
  - + Prevents one class of C undefined behaviors
  - - Puts lots of pressure on the tag cache
- Idea: to control sharing, we only need to distinguish regions that we actually might share
  - according to programmer or (since we trust the compiler) an escape analysis

Using fewer colors (2)

- Distinguish definitely "private" buffers from potentially "sharable" buffers
- Each "Sharable" buffer gets a fresh color
- "Private" buffers are all given a single per-compartment color
- Trade-off: we no longer get as much intra-component protection against memory UBs

Semantics and Implementation

Architecture

- To use tagged C for a specific policy:
  - define vocabulary of tags and tag operators (just as for machine-level tagging)
  - instantiate rules for each control point
- To use it for a specific C program:
  - specify tag information for (at least) link-level C entities including functions and globals
  - ideally we will not need to change the C code
- Policies can be specified per system, per module or even per function

More detailed example: statements

One clause in C statement semantics

```
| IfS e s1 s2 =>
    v@v_tag <- eval e;
    pc_tag_0 <- get_pc_tag;
    pc_tag_1 <- ifSplitT v_tag pc_tag_0;
    set_pc_tag pc_tag_1;
    if v then exec s1 else exec s2;
    pc_tag_2 <- ifJoinT v_tag pc_tag_0 pc_tag_1;
    set_pc_tag pc_tag_2
```

this is designed once and for all

An instantiation for IFC tags:

```
Tag = PUBLIC | SECRET

ifSplitT v_tag pc_tag := OK (v_tag \/ pc_tag)
ifJoinT v_tag old_pc_tag pc_tag := OK old_pc_tag
```

this is written once for each policy (in policy DSL)

More detailed example: expressions

One clause in C expression semantics

```
| PlusE e1 e2 =>
  v1@t1 <- eval e1;
  v2@t2 <- eval e2;
  t <- plusT t1 t2;
  ret (v1+v2)@t
```

this is designed once and for all

An instantiation for IFC tags:

```
  Tag = PUBLIC | SECRET

  plusT v1_tag v2_tag := OK(v1_tag \/ v2_tag)
```

An instantiation for memory safety tags:

```
Tag = NotPtr | Ptr region

plusT v1_tag v2_tag :=
   match v1_tag,v2_tag with
   | NotPtr, NotPtr => OK NotPtr
   | Ptr a,  NotPtr => OK (Ptr a)
   | NotPtr, Ptr a  => OK (Ptr a)
   | _,      _      => FAIL
   end.
```

these are written once for each policy

Compilation

- Go from tagged C to tagged machine code
- Basic idea: specially tag the instructions in the generated code to indicate their C-level role
  - Machine-level rules for these special tags are built directly from the C-level rules
  - Must modify compiler (to generate appropriately tagged instructions)
  - Compilation scheme is independent of policy (although policy-specific schemes might give better code)

Compilation Example

One clause in C expression semantics

```
| PlusE e1 e2 =>
  v1@t1 <- eval e1;
  v2@t2 <- eval e2;
  t <- plusT t1 t2;
  ret (v1+v2)@t
```

Corresponding clause in C expression compiler

```
compileExp (e: Exp) : (reg * list Inst) =
…
| PlusE e1 e2 =>
  let (r1,code1) := compileExp e1 in
  let (r2,code2) := compileExp e2 in
  let r := fresh_reg() in
  (r, code1 ++
      code2 ++
      [MovI @ Icopy r1 r] ++
      [AddI @ Iplus r2 r])
```

Machine-level rules (defined once and for all)

```
MovI,tpc,Icopy,t1,_,_,_ -> tpc,t1,_
AddI,tpc,Iplus,t1,t2,_,_-> tpc,plusT t1 t2,_
```

```
| IfS e s1 s2 =>
   let (r,is) := compileExp e in
   let rt := fresh_reg() in
   let rt' := fresh_reg() in
   let is1 := compileStm s1 in
   let is2 := compileStm s2 in
   is ++
   getpctag rt ++
   combine r rt IifSplitT rt' ++
   setpctag rt' ++
   [BrnzI r (length is2+1) @ X] ++
   is2 ++
   [BrI (length is1) @ X] ++
   is1 ++
   combine rt rt' IifJoinT rt ++
   setpctag rt
```

where

```
getpctag r := [ConstI @ Igetpctag 0 r]
setpctag r := [ConstI @ Isetpctag 0 r]
combine r1 r2 I r3 := [MovI @ Icopy r1 r3] ++
                      [MovI @ I r2 r3]
```

Machine-level rules (defined once and for all)

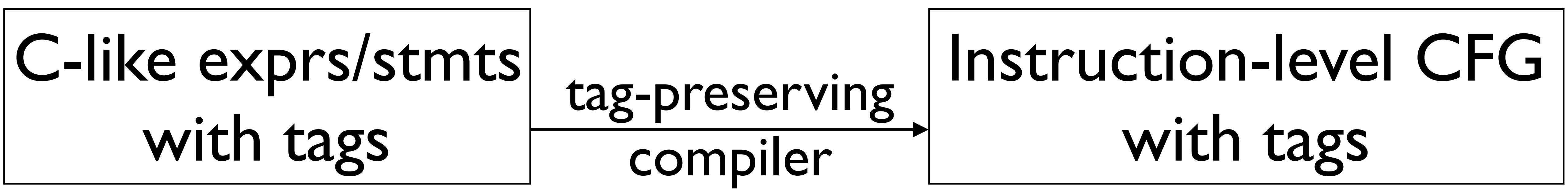
```
ConstI,tpc,Igetpctag,_,_,_,_      -> tpc,tpc,_
ConstI,_,Isetpctag,new_tpc,_,_,_ -> new_tpc,new_tpc,_
MovI,tpc,Icopy,t1,_,_,_           -> tpc,t1,_
MovI,tpc_,IifsplitT,t1,t2,_       -> tpc,ifSplitT tpc t1 t2
MovI,tpc,IifjointT,t1,t2,_        -> tpc,ifJoinT tpc t1 t2
```

saved tags are attached to dummy values
(spilling if necessary just as for real values)

Compiler Verification

- Compiler is now part of TCB, so ideally it should be verified

- First experiment (work in progress): verify semantic preservation for a tagged analog to the RTLGen phase of CompCert



- Longer-term goal: integrate with rest of CompCert pipeline, targeting RISC-V

Conclusions

Summary

- Expressing tag policies at C level extends the range of properties that we can enforce using the PIPE
- These extensions rely heavily on having higher-level semantic "hooks"
- We have a plausible compilation scheme
- But, the compiler must be trusted or verified

Status

- Have proof-of-concept compiler for toy versions of source language, machine, and policies
  - Source: while, if, functions, global arrays, local variables, malloc
  - Target: infinite register machine with argument marshaling primitives
  - Policies: IFC, compartments, memory safety
  - Verification of semantic preservation is in progress

Ongoing work

- Continuing to extend toy source language with more features of full C
  - types and casts, addressable locals, function ptrs, …
  - non-structured control flow?
- Designing, implementing, and validating compartmentalization policies
- Verification experiments with toy compiler

Some Open Questions

- How much can/should we modify C code?
  - e.g. tags on parameters, local variables?
  - who will be the "security engineer" applying policies?
- How should we handle C undefined behaviors?
  - Any properties enforced by C-level policies hold only if the C code does not trigger undefined behavior (UB)
  - We can detect some memory-based UBs using tag policies
  - Should we "bake in" these policies?
- What about alternatives to C?
  - lower-level, e.g. LLVM-based
  - higher-level, e.g. safe structured languages