Bug-finding Using Meta-level Compilation

Michelle Mazurek

18732: Secure Software Systems
Fall 2010

(some slides from Jon McCune, Dawson Engler)

Electrical & Computer ENGINEERING

Meta-outline

- Main lecture: Meta-level compilation
- Mini-lecture: Introducing HW2a

- Wednesday: Lecture by Lorrie Cranor
- No office hours today – Friday by appointment instead

Well-known system rules, not always obeyed correctly

- Don’t access variable A without lock B
- Don’t free the same buffer twice
- Asserts shouldn’t have side effects
- Don’t block while holding a spin lock
- Don’t use floating-point in kernel code
- Etc.

- For correctness, for performance

How can we find bugs?

- Model checking
- Testing
- Inspection
- **Static analysis**

Model checking

- ☺ Rigorous, effective

- ☹ Model is hard to construct
- ☹ Model may not accurately reflect implementation
- ☹ Model may be oversimplified
- ☹ State space explosion / will it terminate?

Testing

- ☺ Simpler
- ☺ Works with actual code, not a model

- ☹ Hard to exercise every code path
- ☹ # test cases to write scales with code size
- ☹ Identifying root causes can be hard (i.e., delayed system crash)
- ☹ Run every device driver?
- ☹ Redo when code is changed

Manual inspection

☺ Consider all semantic levels

☺ Developers are smart, adaptive

☹ Lots of code, deep and complex code paths

☹ People are not machines

☹ Redo when code is changed

Static analysis

- ☺ Examine all execution paths (sort of)
- ☺ Works with real code, not a model
- ☺ Scales well, sort of
  - ☺ Write rule once, test everywhere

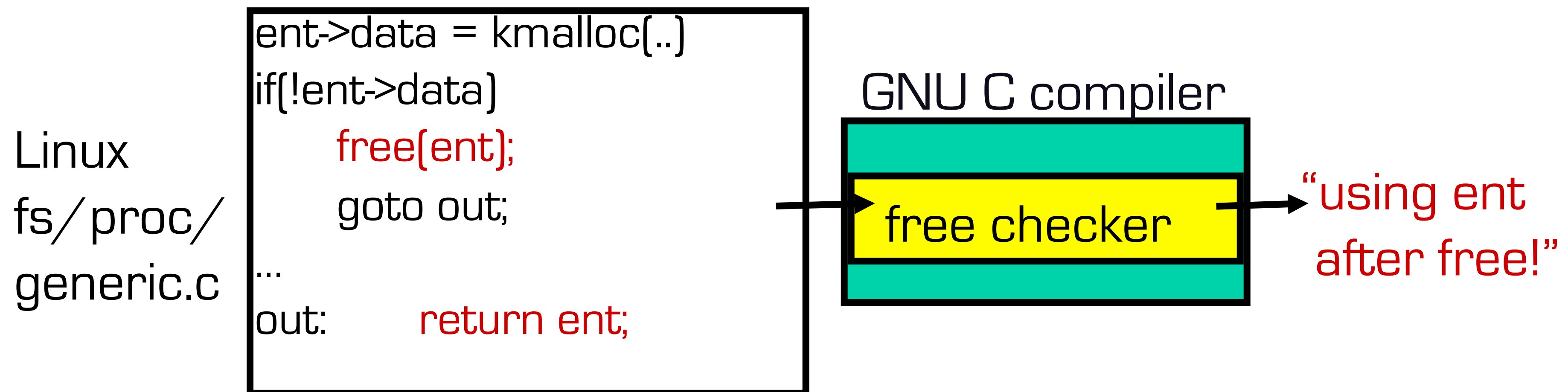
- ☹ What about non-determinism?
- ☹ How much state/context can you keep?
- ☹ Soundness, completeness?

Meta-level compilation

- Engler et al., OSDI 2000
- Compilers are good at verifying rules
  - But don't have system-specific domain knowledge
- Developers have domain knowledge
  - But manual inspection is painful, erratic
- Metacompilation (MC): developers tell the compiler what extra rules to check for

MC overview

- Rules written as state-machine
- Dynamically linked into xg++ compiler
- Applied down all paths in input source code
  - Transition states, detect and output errors



- Scales to millions of LOC
- Found 1500+ errors in Linux source

Outline

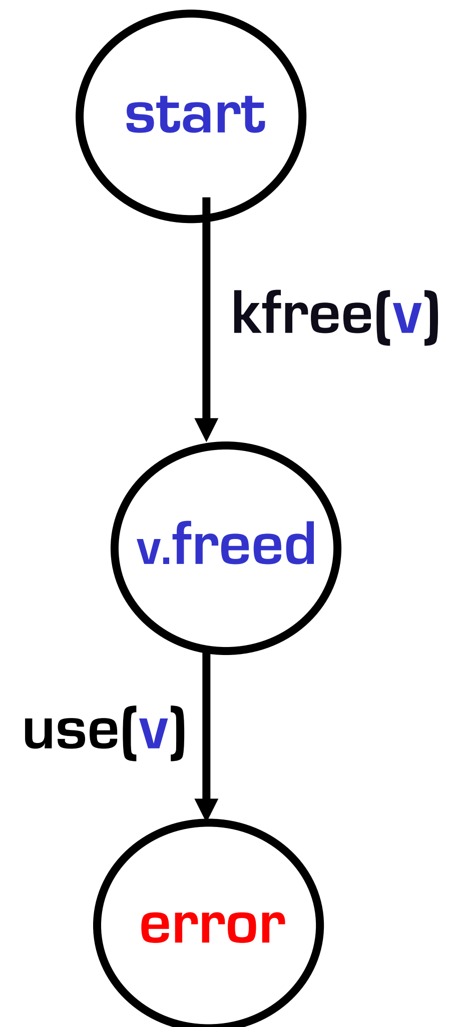
- Intro – bug finding
- Metacompilation overview
- Basic static analysis example
- Three metacompliation examples
- Inferring rules (SOSP ’01)

Example: Don’t use freed memory

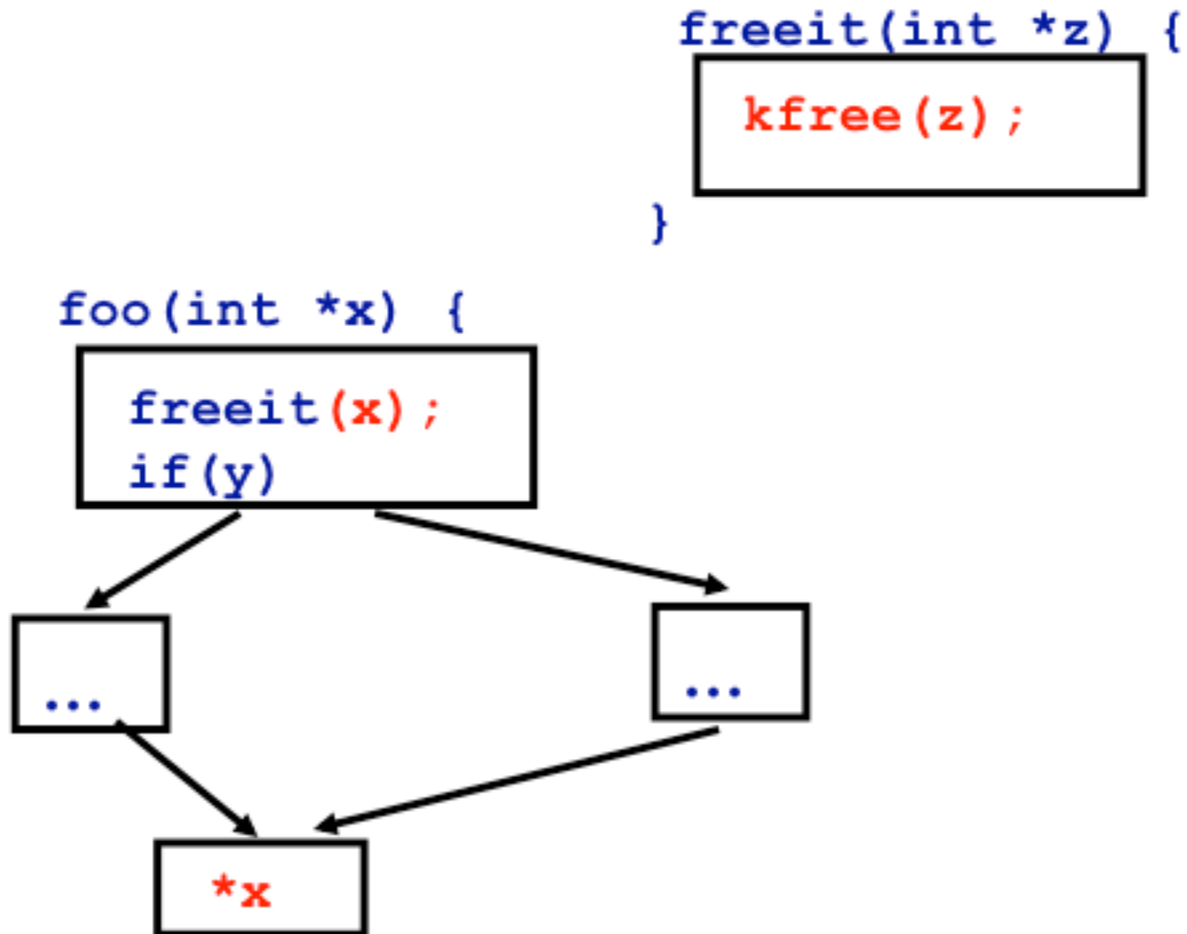
```
sm free_checker {
 state decl any_pointer v;
 decl any_pointer x;

 start: { kfree(v); } ==> v.freed
  ;
 v.freed:
     { v == x }
   | { v != x } ==> { /* suppress fp */ }
   | { v } ==> { err("Use after free!");
   ;
}
```

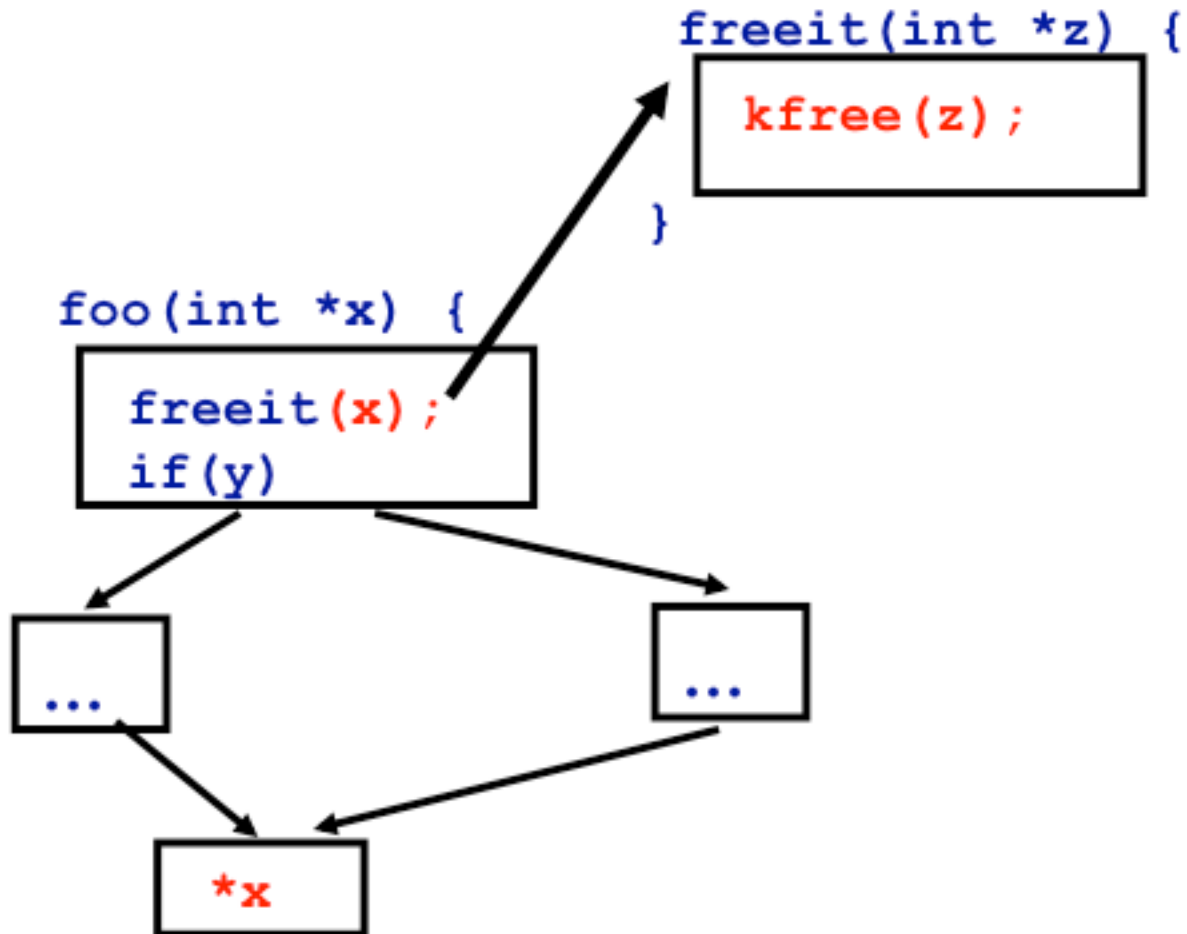
Define start state, transition rules, error conditions



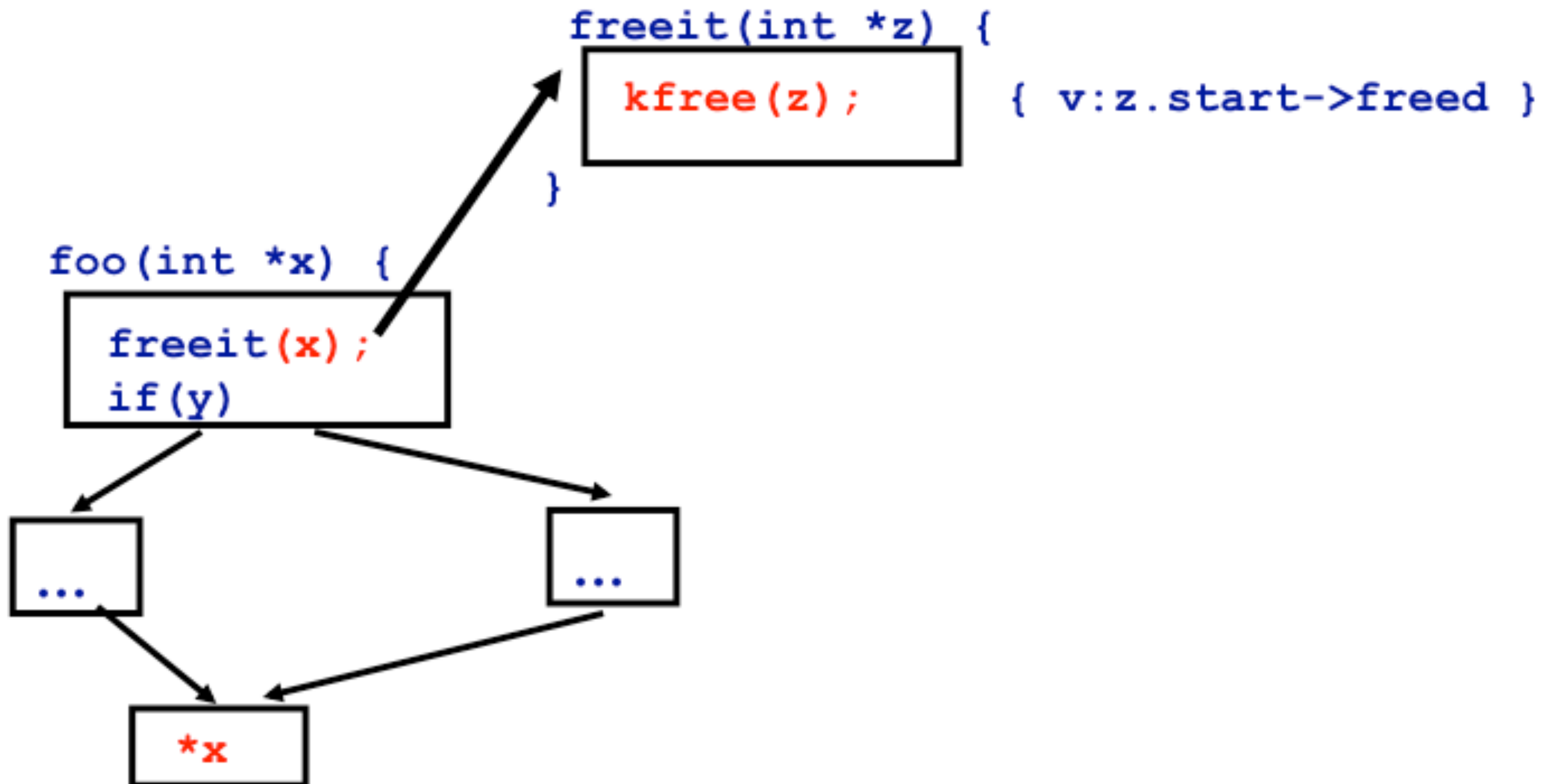
Example: Don't use freed memory



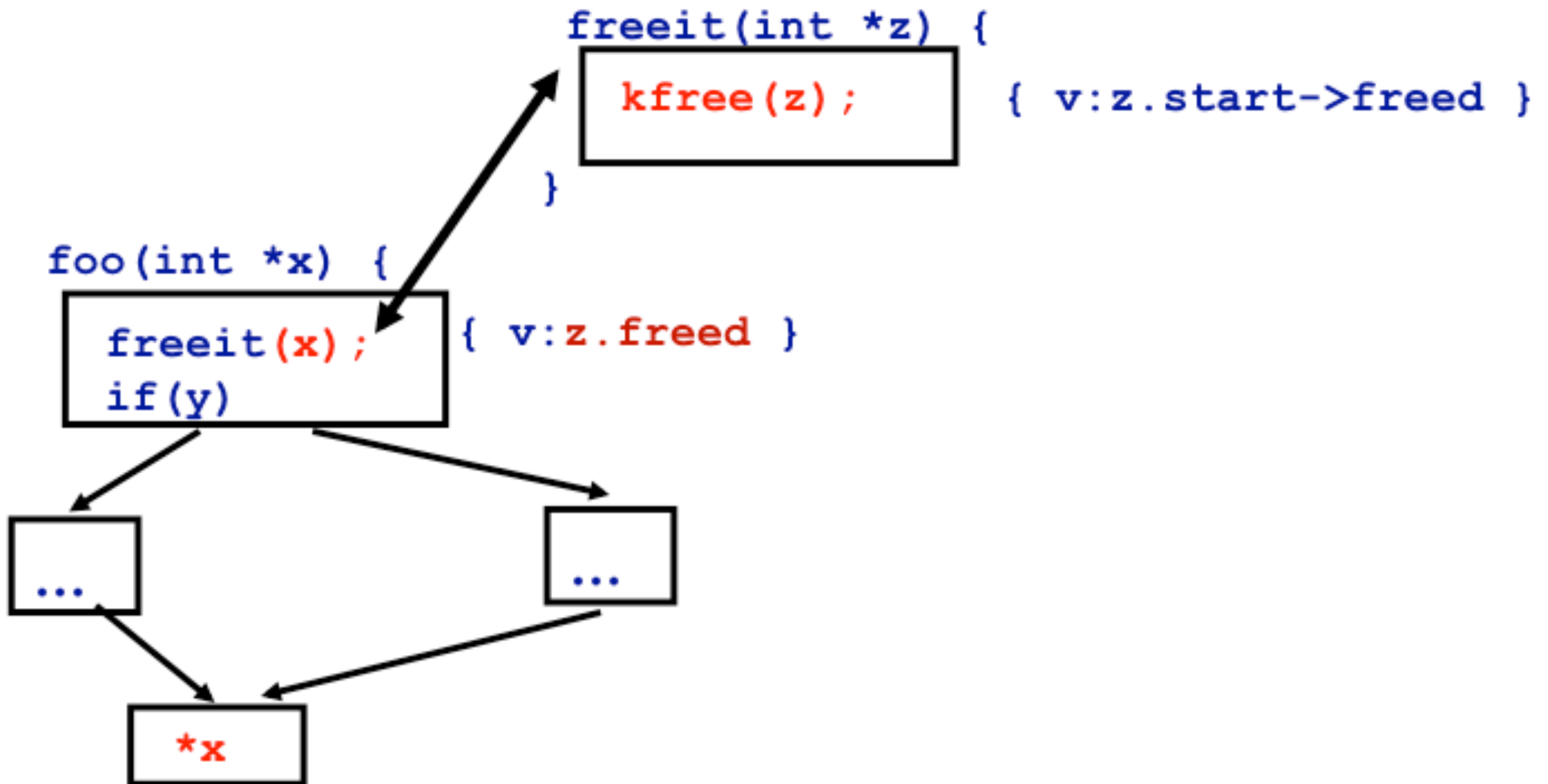
Example: Don’t use freed memory



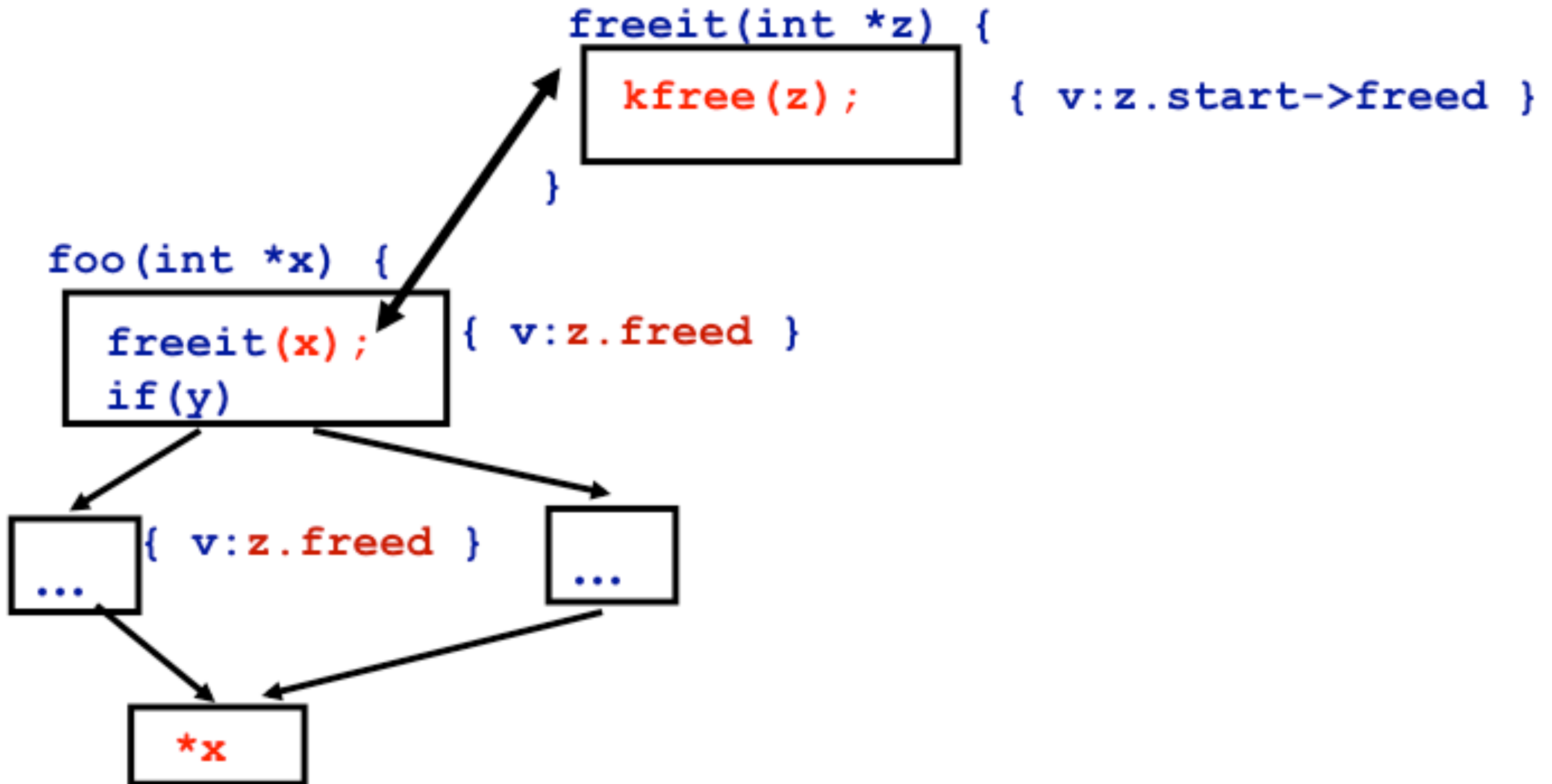
Example: Don't use freed memory



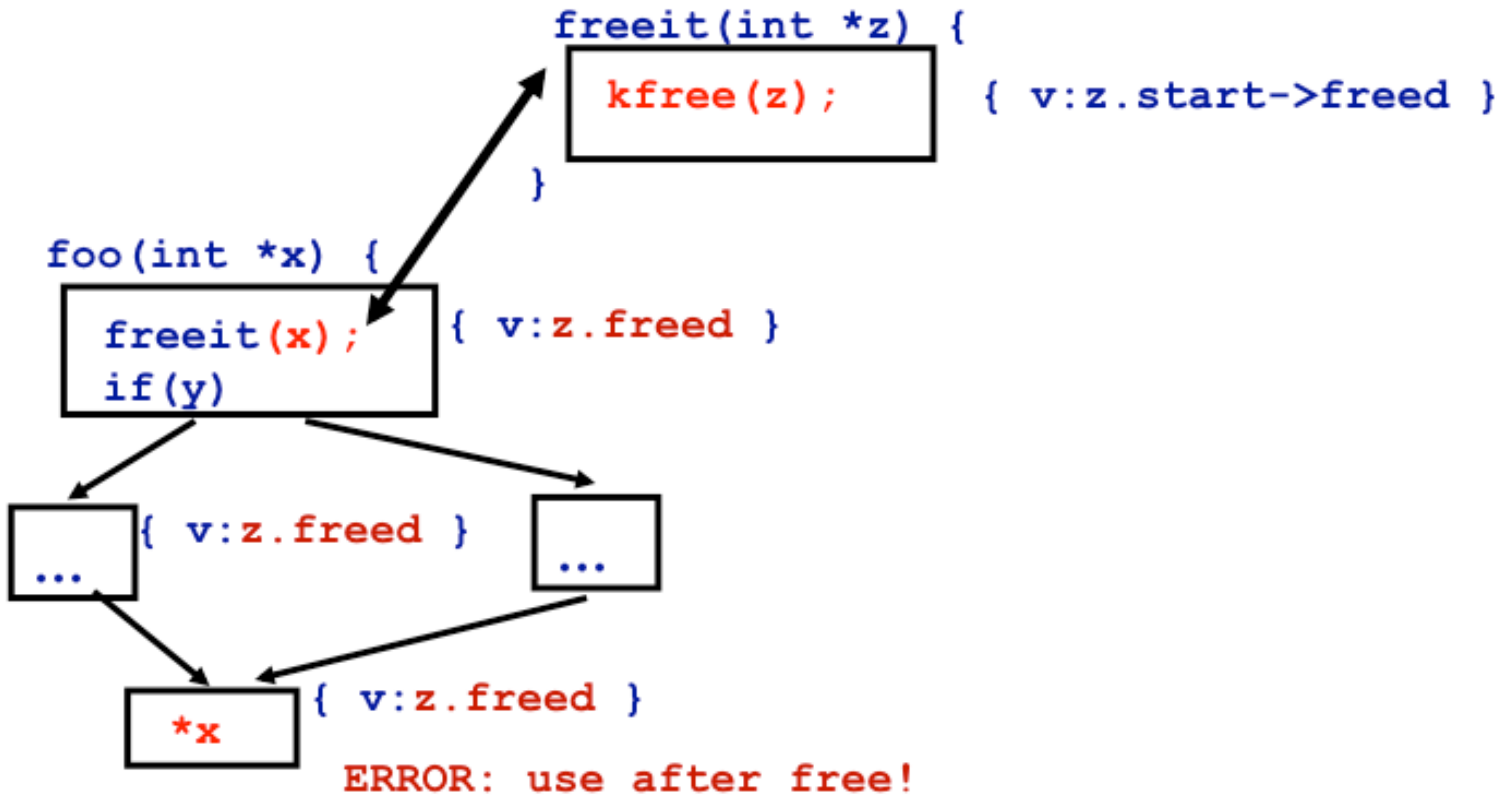
Example: Don't use freed memory



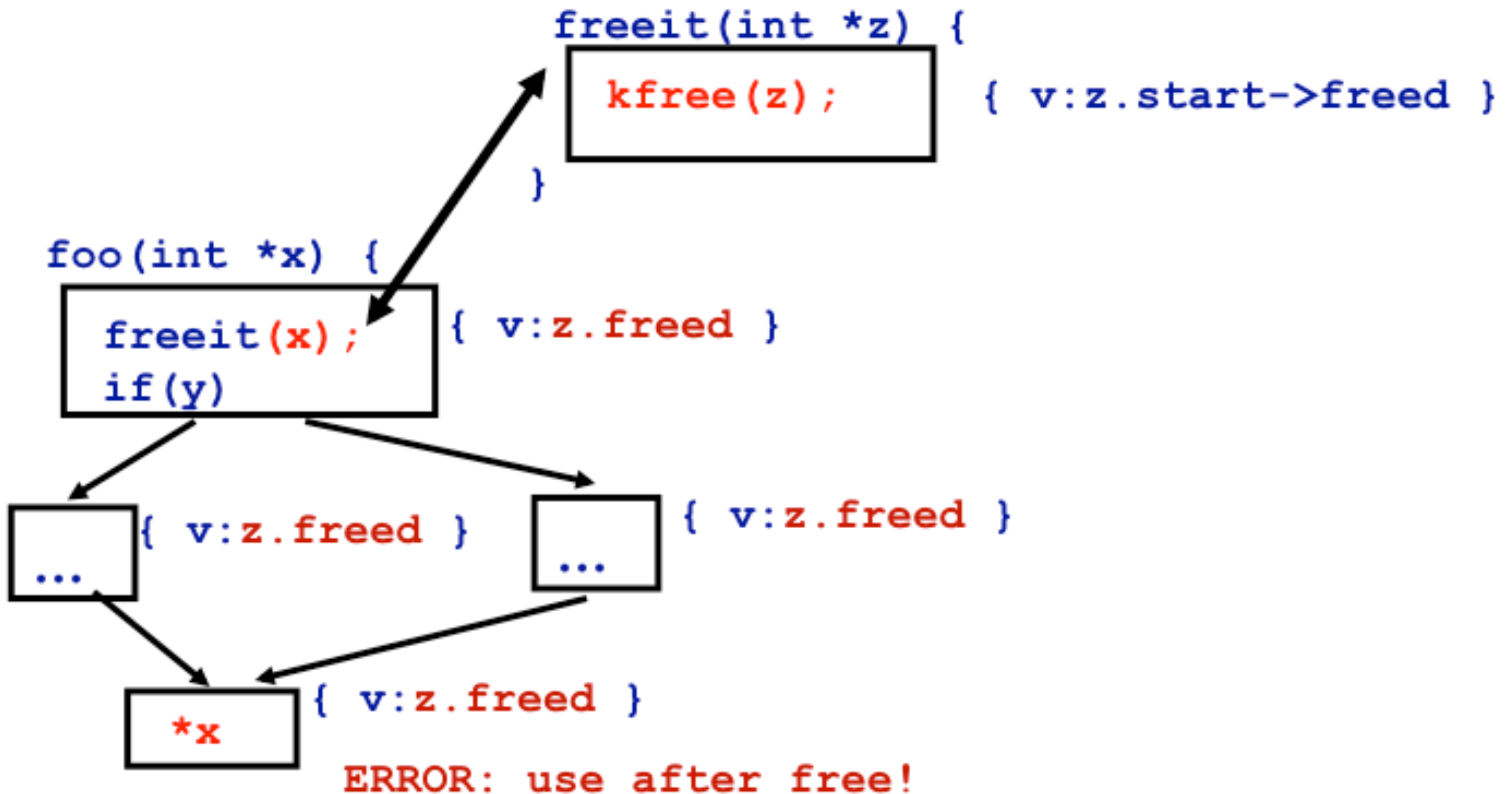
Example: Don't use freed memory



Example: Don’t use freed memory



Example: Don't use freed memory

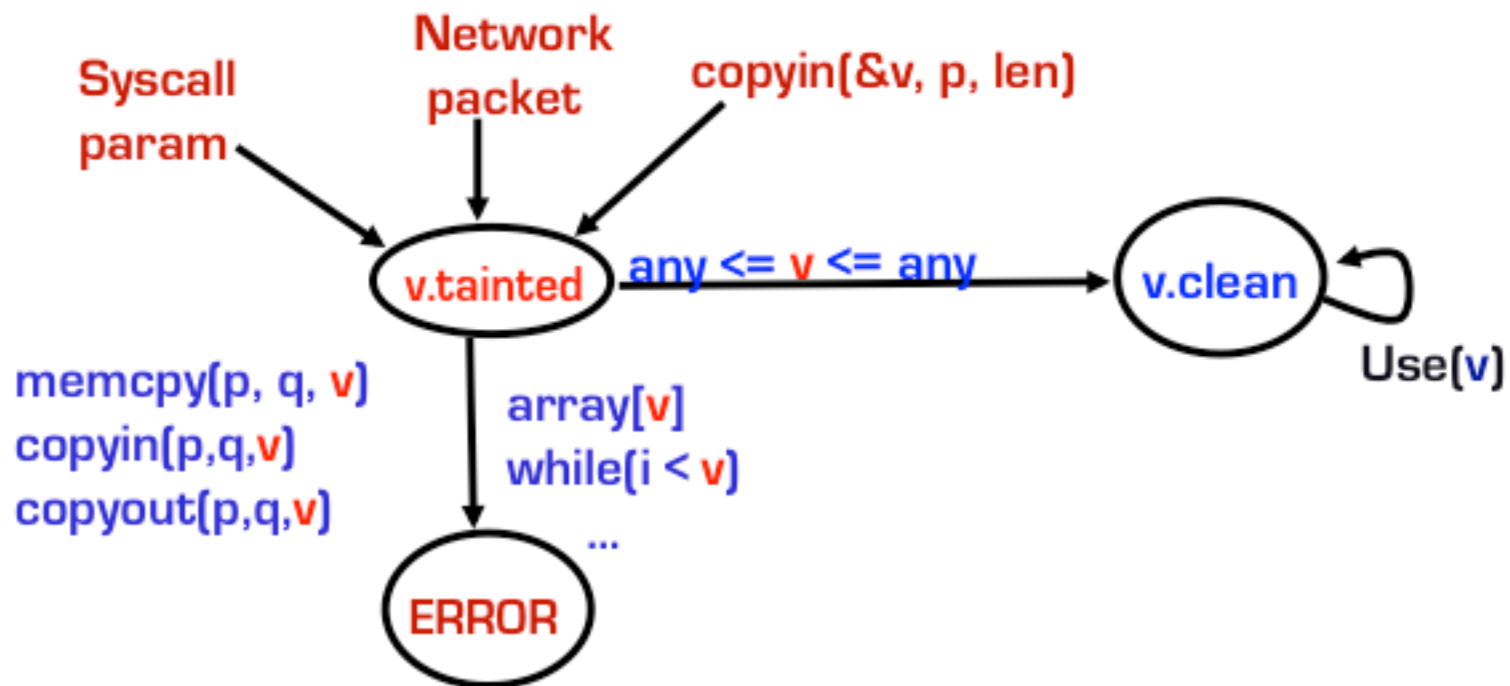


Outline

- Intro – bug finding
- Metacompilation overview
- Basic static analysis example
- Three metacompliation examples
    1. Temporal rule: sanitize user input before use
    2. Context rule: blocking
    3. Dynamic to static: assert
- Inferring rules (SOSP ’01)

Kernel: sanitize integers before use

- MC checker: Warn when unchecked integers from untrusted sources reach trusting sinks
  - Written by undergrad; found 101 real Linux errors



Integer sanitization: Real bugs

No check:

```
2.4.5-ac8/drivers/usb/se401.c:
copy_from_user(&frame, arg, sizeof(int));
wframe(se401, frame);
[frame].grabstate = FRAME_UNUSED;
```

Unexpected overflow:

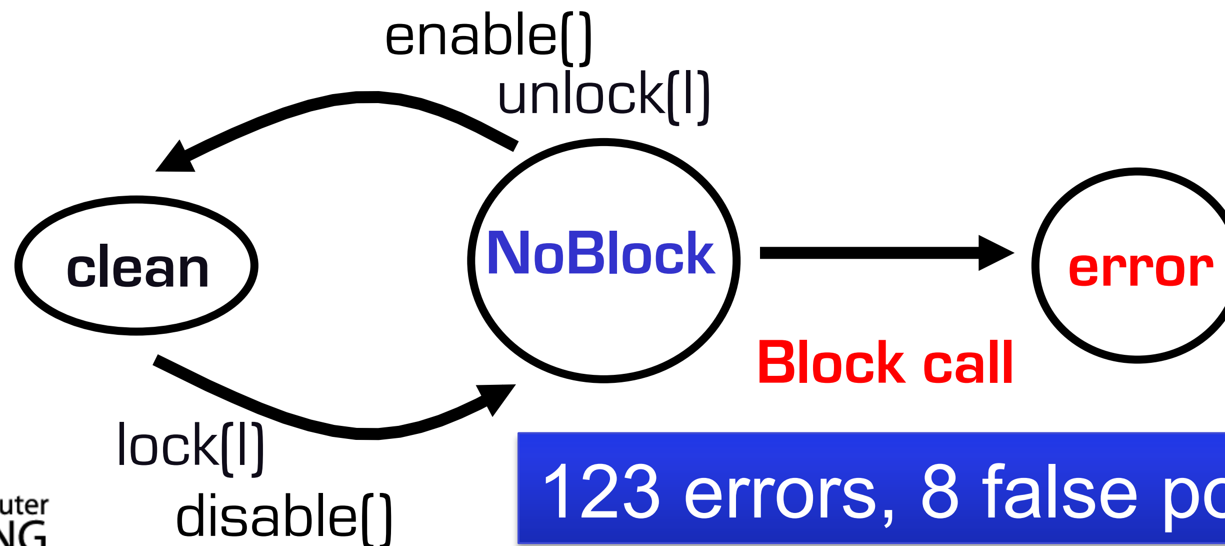
```
/* 2.4.9: drivers/net/wan/farsync.c */
copy_from_user(&wrthdr, addr, sizeof wrthdr);
if ( wrthdr.size + wrthdr.offset > FST_MEMSIZE )
dr.offset,data,wrthdr.size)
```

Remote exploit (no check):

```
/* 2.4.9/drivers/isdn/act2000/capi.c:actcapi_dispatch */
isdn_ctrl cmd;
...
while ((skb = skb_dequeue(&card->rcvq))) {
   msg = skb->data;
   ...
   memcpy(cmd.parm.setup.phone,msg->msg.connect_ind.addr.num,
                               msg->msg.connect_ind.addr.len - 1);
```

Blocking and deadlock prevention

- Don't block while interrupts are disabled, or while holding a spin-lock
- Requires knowledge of call-chain ***context***
  1. List all possible blocking functions (flow graph)
  2. Check disabled/locked state when calling



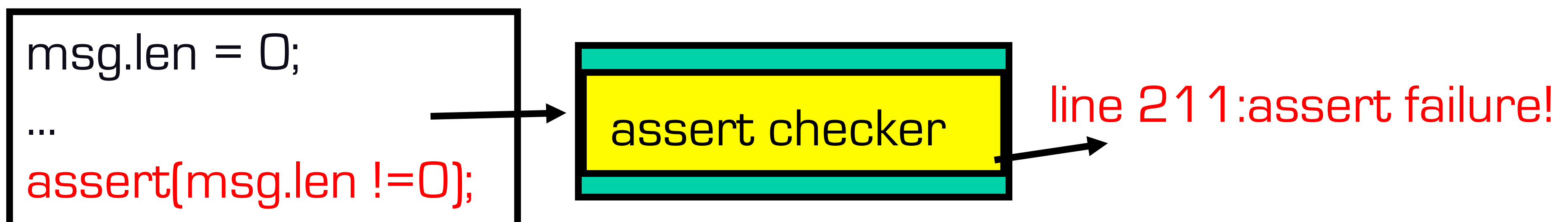
Blocking: Real bug

- Copy to/from user space blocks on page fault

```
/* drivers/net/pcmcia/wavelan_cs.c */
spin_lock_irqsave (&lp->lock, flags);/* 1889 */
switch(cmd)
...
  case SIOCGIWPRIV:                    /* 2304 */
   if(copy_to_user(wrq->u.data.pointer, …))
```

Statically checking assert

- Track scalar values through compiler dataflow
  - Constant / deterministic assignments only
  - Indeterminate values reach “unknown” state
- Test assert against known values
- Found 5 errors in FLASH
  - Assignment, assert up to 300 lines, many ifs apart



Metacompilation summary

- System correctness rules can be expressed as concrete source patterns
- Check by making compilers system-specific
- Write state-machine checker per rule
- Result: precise, immediate error diagnosis

Limitations

- Heuristic, not formal (neither sound nor complete)
- False positives:
  - Rules abused strategically
  - Incomplete global state knowledge
  - Doesn’t prune impossible paths
- False negatives: shallow, tightly bound to syntax
- Control flow limitations (loops, branches)
- Still, better than not finding bugs!

Outline

- Intro – bug finding
- Metacompilation overview
- Basic static analysis example
- Three metacompliation examples
- Inferring rules (SOSP '01)
  - Very briefly

Bugs as deviant behavior

- Engler et al., SOSP 2001 (http://doi.acm.org/10.1145/502034.502041)
- Writing rules is nice, but can we do better?
  - What if the rules wrote themselves?
  - Do we need to know what is correct a priori?
- Goal: Use statistics to find out what programmers commonly do, call it a rule
  - Flag anything that doesn't behave typically

Statistical consistency

- Rule templates: <a> before <b>, lock <x> protects variable <y>, function <foo> frees pointer <p>
- Assume all possible instances are rules
- Count how often the rule is / is not followed
- Rank using statistical likelihood of deviancy
  - Leverage "latent specifications": naming conventions, return negative error code, etc.
  - Noisy but useful

HW2A: Model Checking

(some slides borrowed from Anupam)

Logistics

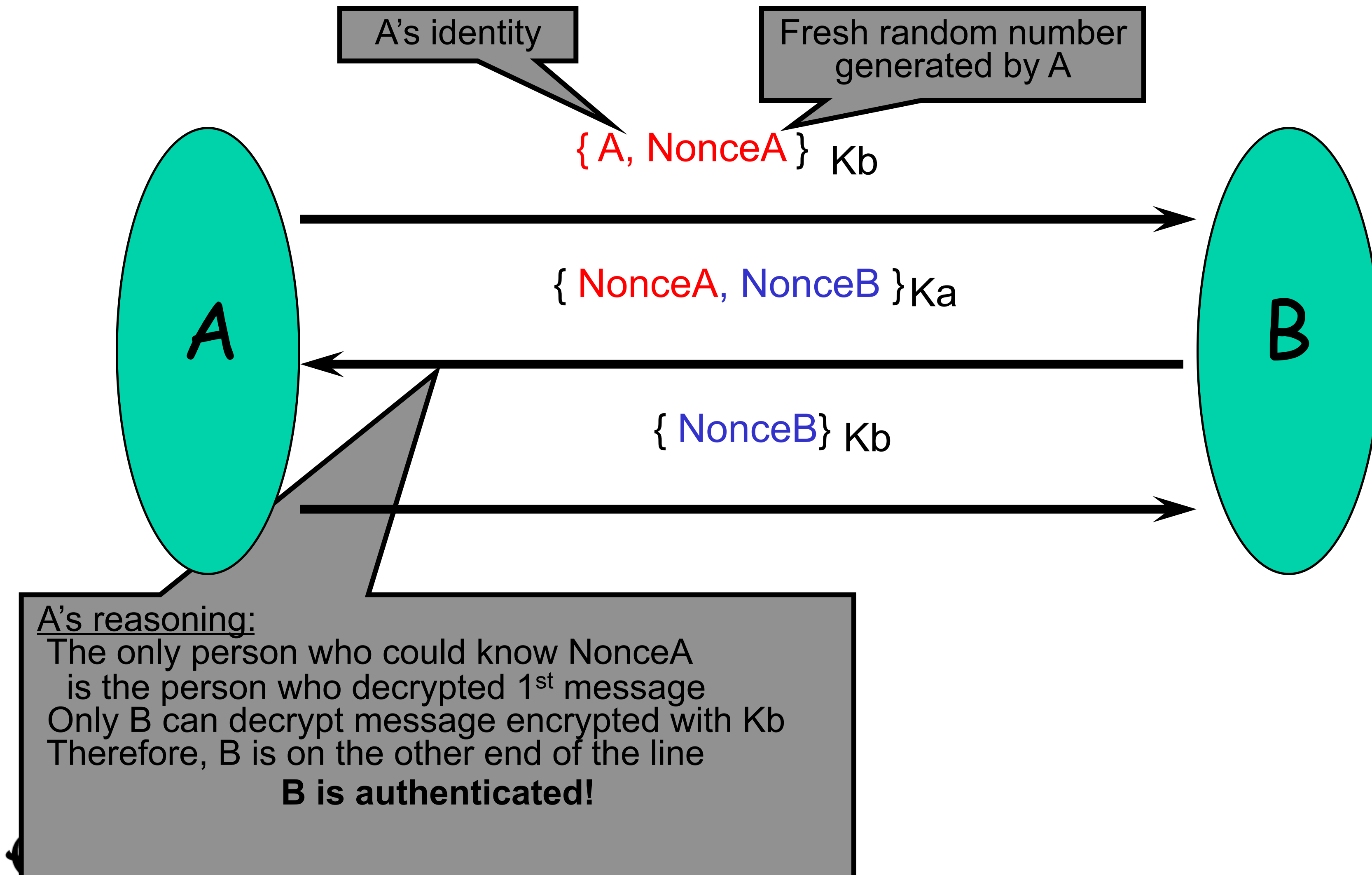
- Part A, 50% of the total, posted on blackboard and the course website after class
- Due: **Monday, October 18**
- No office hours today – Friday by appointment instead

Model-checking with Murphi (Dill et al.)

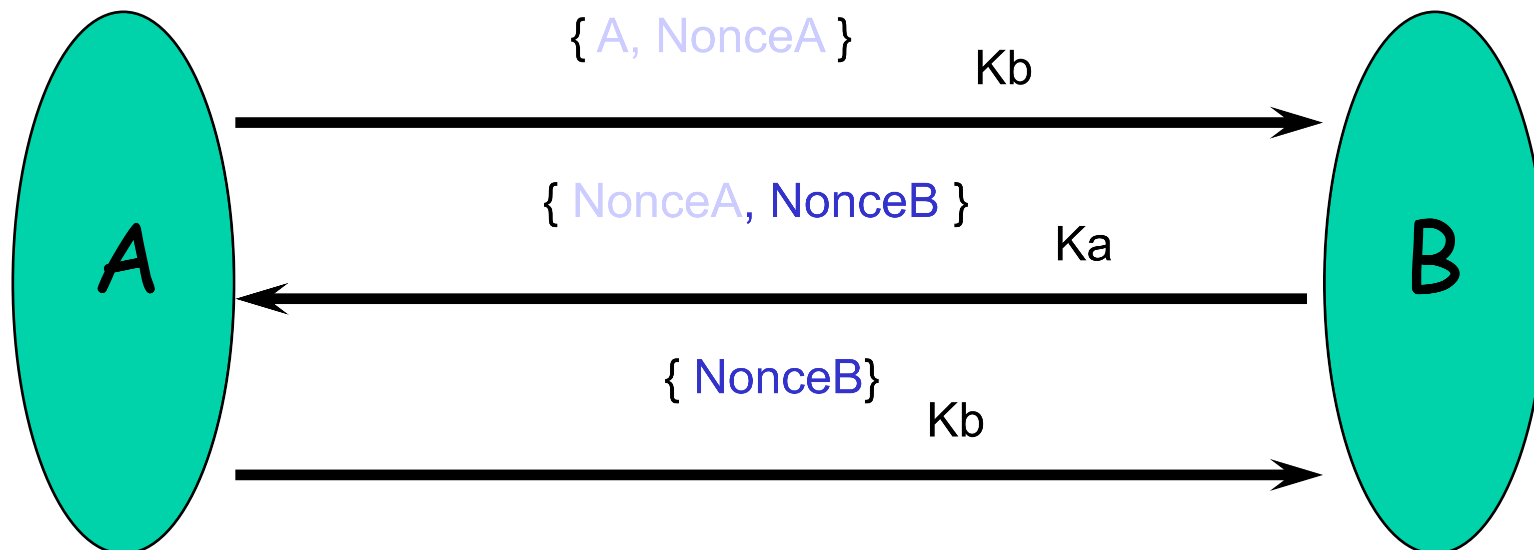
- Describe finite-state system
  - State variables with initial values
  - Transition rules for each participant & attacker
  - Communication by shared variables
- Specify security condition as a state invariant
  - Predicate over state variables that must be true in every state reachable by the protocol
- Automatic exhaustive state enumeration
  - Can use hash table to avoid repeating states
- Example: Needham-Schroeder bug

[Needham-Schroeder 1978]

Authentication by Encryption



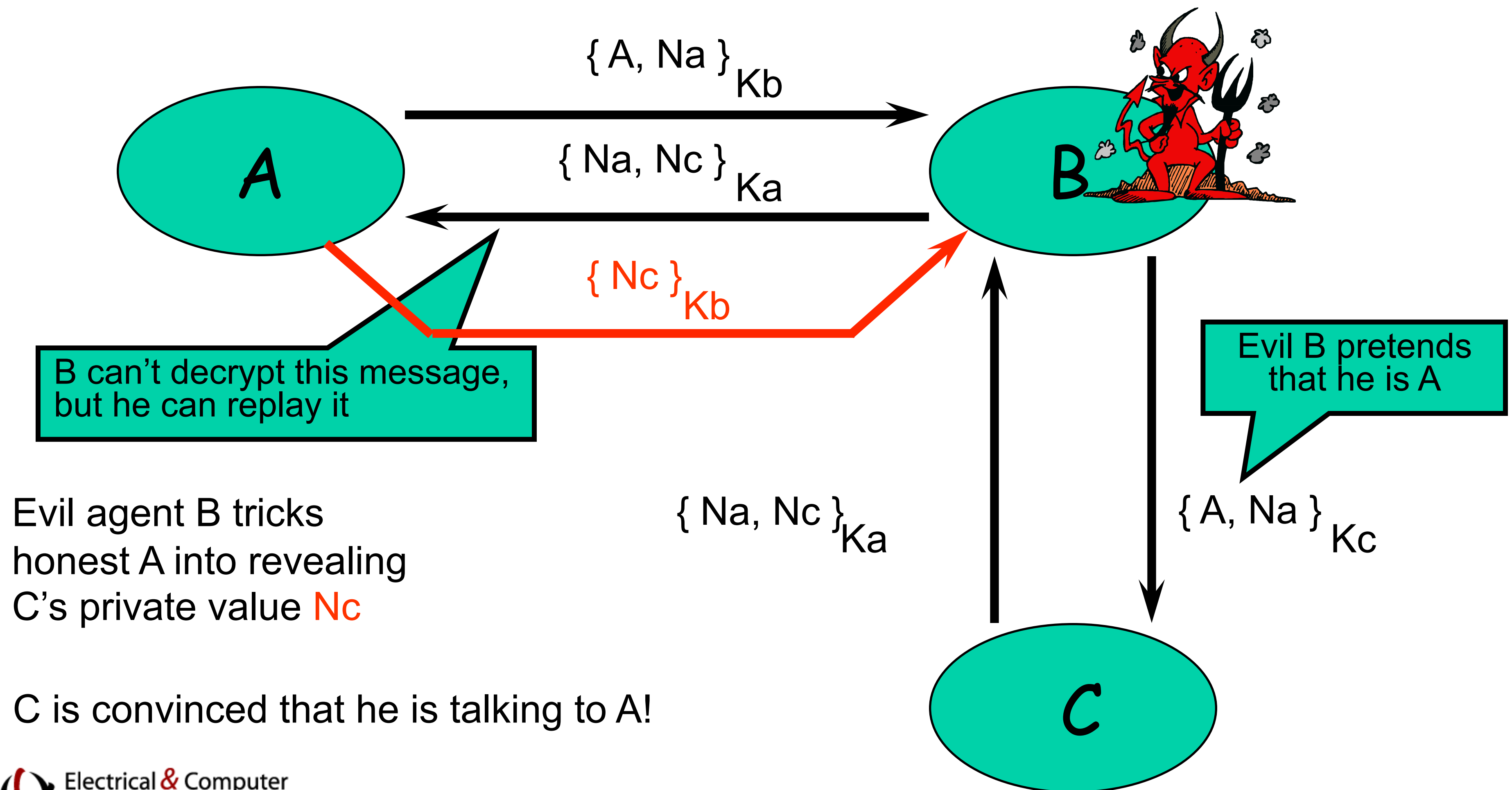
What Does This Protocol Achieve?



- Protocol aims to provide both authentication and secrecy
- After this exchange, only A and B know NonceA and NonceB

Anomaly in Needham-Schroeder

[published by Lowe 1995]



Evil agent B tricks honest A into revealing C's private value Nc

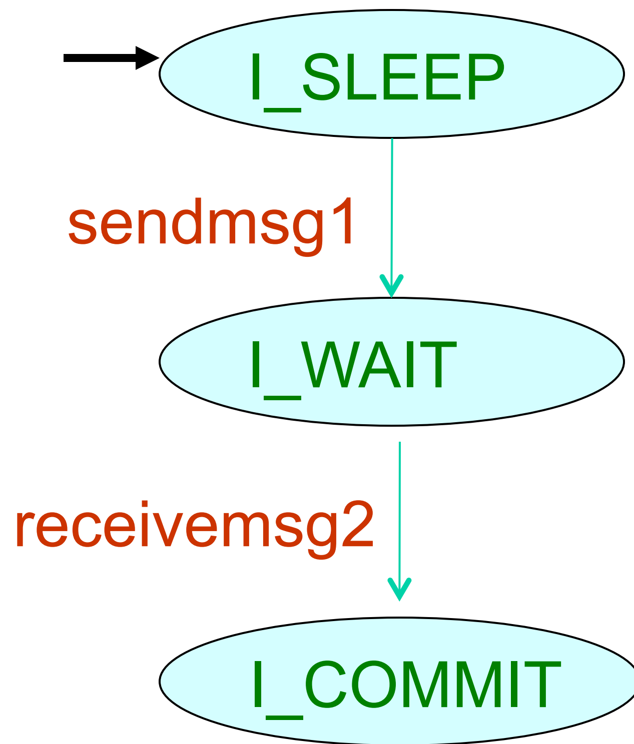
C is convinced that he is talking to A!

Applying Murphi to protocols

- Formulate the protocol
  - Define a datatype for each message format
  - Describe finite-state behavior of each participant
    - If received message M3, then create message M4, deposit it in the network buffer, and go to state WAIT
  - Describe security condition as state invariant
- Add adversary
  - Full control over the "network" (shared buffer)
  - Nondeterministic choice of actions
    - Intercept message; split it into parts; remember parts
    - Generate new messages from observed data and initial knowledge (e.g., public keys)

Murφ will try all possible combinations

NS Initiator as a State Machine



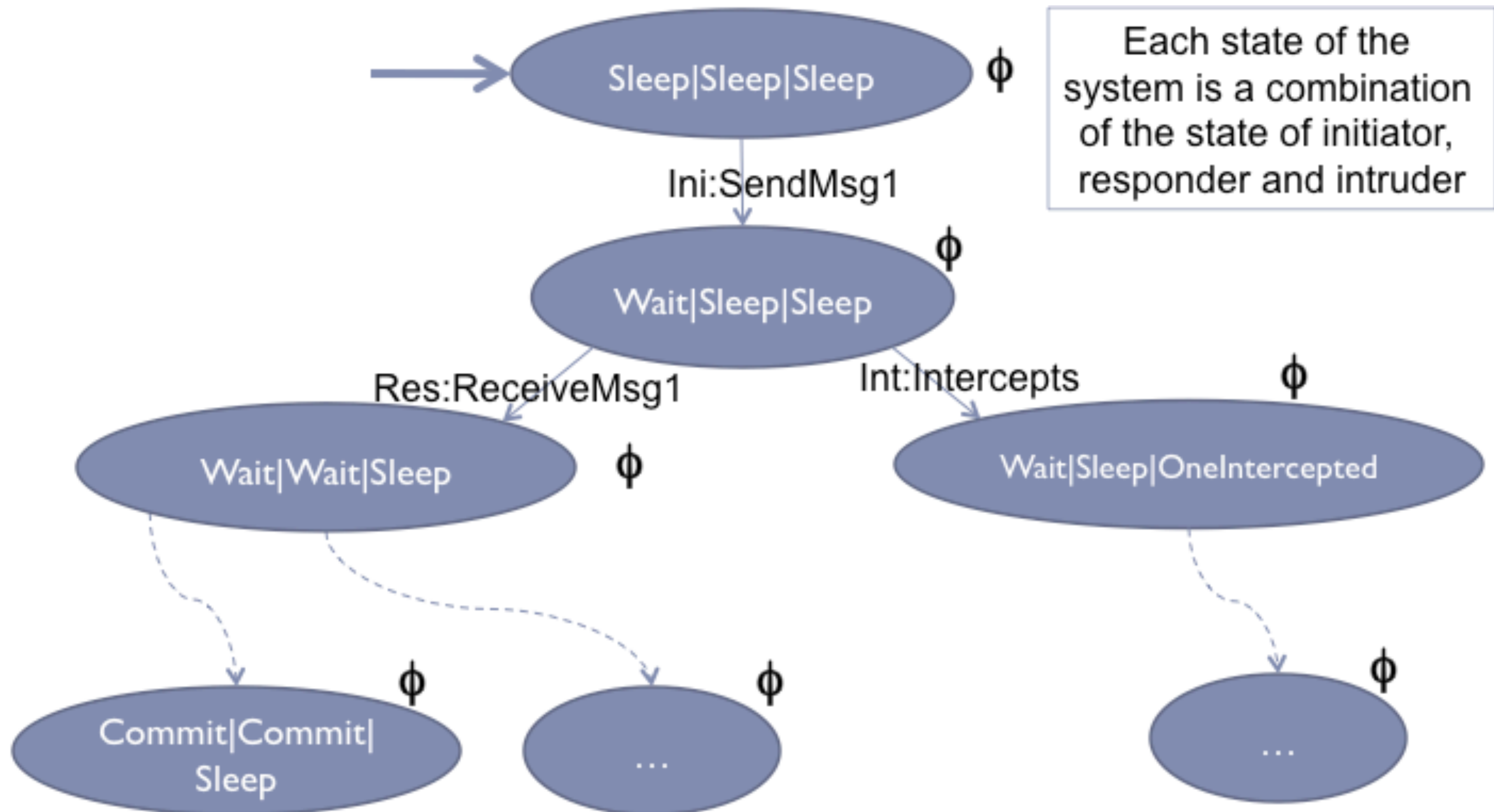
- State machine

(S, s_0, Σ, δ) where

- S: set of states
- s_0:initial state
- Σ: set of actions
- δ: transition relation

$$\delta: S \times \Sigma \rightarrow S$$

Needham-Schroeder continued



ϕ should hold in all states

Messages and network

```
MessageType : enum {        -- types of messages
  M_NonceAddress,           --  {Na, A}Kb  nonce and addr
  M_NonceNonce,             --  {Na,Nb}Ka  two nonces
  M_Nonce                   --  {Nb}Kb     one nonce
};

Message : record
    source:   AgentId;      -- source of message
    dest:     AgentId;      -- intended destination of msg
    key:      AgentId;      -- key used for encryption
    mType:    MessageType;  -- type of message
    nonce1:   AgentId;      -- nonce1
    nonce2:   AgentId;      -- nonce2 OR sender id OR empty
end;
var
  net: multiset[NetworkSize] of Message; -- state variable for n/w
```

States in NS (S, s_0, Σ, δ)

- `var                                            -- state variables for`
- `  net: multiset[NetworkSize] of Message;     --  network`
- `  ini: array[InitiatorId] of Initiator;      --  initiators`
- `  res: array[ResponderId] of Responder;      --  responders`
- `  int: array[IntruderId] of Intruder;        --  intruders`

Initial State in NS (S, s_0, Σ, δ)

```
• startstate
•   -- initialize initiators
•   undefine ini;
•   for i: InitiatorId do
•     ini[i].state     := I_SLEEP;
•     ini[i].responder := i;
•   end;
•   -- initialize responders
•     -- make all responder states R_SLEEP
•   -- initialize intruders
•     -- make all intruder stored nonces empty
•   -- initialize network
•   undefine net;
• end;
```

Actions in NS (S, s_0, Σ, δ)

- Actions in Murphi model of NS update the state variables

```
•       multisetadd (outM,net); -- add message to the network

•       res[i].state     := R_WAIT;  -- change responder state
```

Transition Relation in NS (S, s_0, Σ, δ)

- Transition relation for protocol steps (honest parties)
- Transition relation for adversary steps

Modeling Protocol Actions

```
ruleset i: InitiatorId do
  ruleset j: AgentId do
    rule 20 "initiator starts protocol"
      ini[i].state = I_SLEEP & !ismember(j,InitiatorId) &
multisetcount (l:net, true) < NetworkSize ==>
    var
      outM: Message;   -- outgoing message
    begin
      undefine outM;
      outM.source  := i; outM.dest    := j;
      outM.key     := j; outM.mType   := M_NonceAddress;
      outM.nonce1  := i; outM.nonce2  := i;
      multisetadd (outM,net); ini[i].state :=I_WAIT;
      ini[i].responder := j;
    end; end;end;
```

Adversary Model

- Formalize “knowledge”
  - initial data
  - observed message fields
  - results of simple computations

Modeling the attacker

```
-- intruder i sends recorded message
ruleset i: IntruderId do          -- arbitrary choice of
  choose j: int[i].messages do        --  recorded message
    ruleset k: AgentId do             --  destination
      rule "intruder sends recorded message"
        !ismember(k, IntruderId) &    -- not to intruders
        multisetcount (l:net, true) < NetworkSize
      ==>
       var  outM: Message;
       begin
          outM        := int[i].messages[j];
          outM.source := i;
          outM.dest   := k;
          multisetadd (outM,net);
end; end; end; end;
```

Modeling Properties of NS

```
invariant "responder correctly authenticated"
  forall i: InitiatorId do
    ini[i].state = I_COMMIT &
    ismember(ini[i].responder, ResponderId)
    ->
    res[ini[i].responder].initiator = i &
    ( res[ini[i].responder].state = R_WAIT |
      res[ini[i].responder].state = R_COMMIT )
  end;
```

Two papers you should read

- About SecVisor: http://www.sosp2007.org/papers/sosp079-seshadri.pdf
- Model-checking SecVisor: http://www.andrew.cmu.edu/user/danupam/fcds-oakland2010.pdf
- Both listed in instructions under resources