Static API Usage Verifier

Motivating Example: Device Drivers

- Defacto Mechanism for adding functionality into the OS
  - Virus protection software
  - Virtual machine emulation
  - Performance monitoring
- Extremely complicated API
  - Plug and play
  - Power management
  - Asynchronous I/O

Challenge

- The system is left in a “consistent” state
- for all possible execution paths

Approach

- Obeys API usage rules
- Environment model
- Analysis Engine

Outline

- API Usage Rules
- Environment Model
- Analysis Engine
- Automatic Abstraction

Outline

- **API Usage Rules**
- Environment Model
- Analysis Engine
- Automatic Abstraction

API Usage Rules

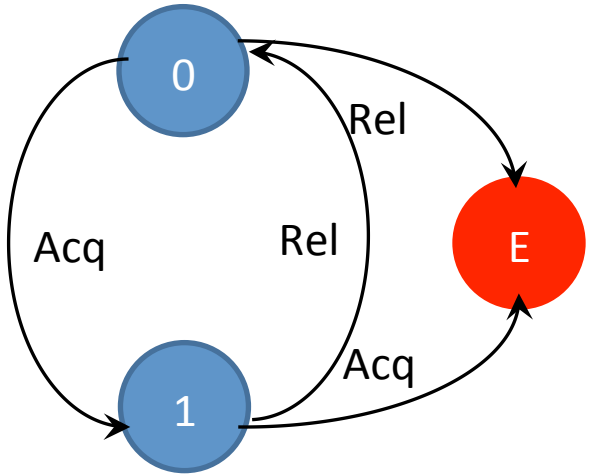
- Safety automaton
  - State variables
  - State transitions
    - Before / after API calls
    - Check pre / post conditions

API Usage Rules: Example

```
state {
        enum { Unlocked=0, Locked=1 }
        state = Unlocked;
}

KeAcquireSpinLock.return {
        if (state == Locked)        error();
        else                        state = Locked;
}

KeReleaseSpinLock.return {
        if (!(state == Locked))     error();
        else                        state = Unlocked;
}
```



Outline

- API Usage Rules
- **Environment Model**
- Analysis Engine
- Automatic Abstraction

Environment Model

- Harness Code
  - Initialization of the module
  - Invocation of all exported functions:
    - symbolic input + arbitrary initial states
    - In isolation or in a sequence of calls as dictated by the exported API
      - » Fewer false positive vs. Increased analysis time
- Stub Code
  - Semantics of the API
  - Both successful and failing behavior

Outline

- API Usage Rules
- Environment Model
- **Analysis Engine**
- Automatic Abstraction

Analysis Engine: Key Idea

- For each usage rule R:
  - Find S, minimum state relevant to R
  - Build P’ = an abstraction of P w.r.t S
  - Model check P’ against R

Analysis Engine: Instrument

```
void example() {
 do {
  KeAcquireSpinLock();

  nPacketsOld = nPackets;
  req = devExt->WLHV;
  if(req && req->status){
    devExt->WLHV = req->Next;
    KeReleaseSpinLock();

    irp = req->irp;
    if(req->status > 0){
      irp->IoS.Status = SUCCESS;
      irp->IoS.Info = req->Status;
    } else {
      irp->IoS.Status = FAIL;
      irp->IoS.Info = req->Status;
    }
    SmartDevFreeBlock(req);
    IoCompleteRequest(irp);
    nPackets++;
  }
 } while(nPackets!=nPacketsOld);
 KeReleaseSpinLock();

}
```

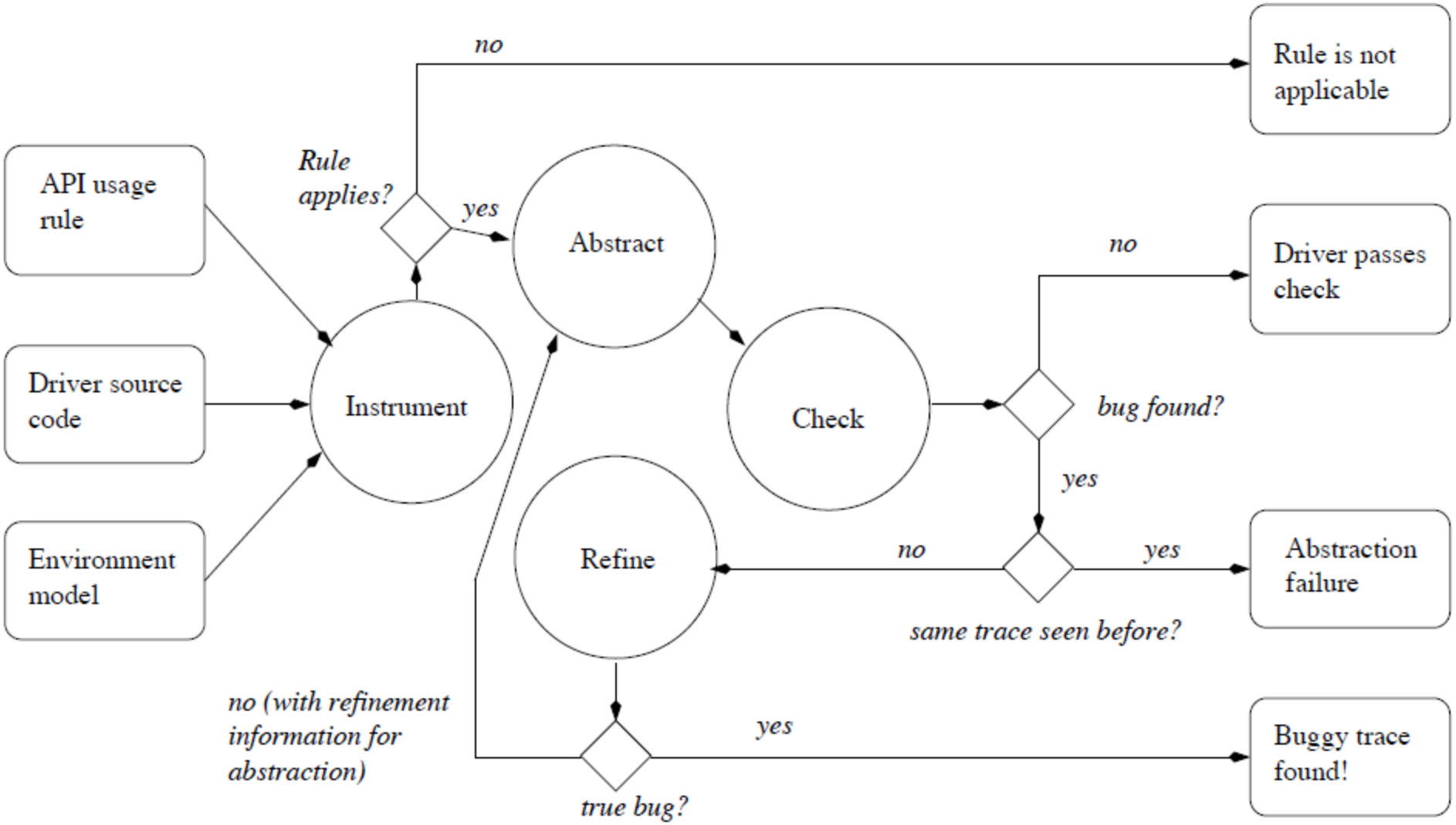
```
void example() {
   do {
    KeAcquireSpinLock();
A:  KeAcquireSpinLock_return();
    nPacketsOld = nPackets;
    req = devExt->WLHV;
    if(req && req->status){
      devExt->WLHV = req->Next;
      KeReleaseSpinLock();
B:    KeReleaseSpinLock_return();
      irp = req->irp;
      if(req->status > 0){
        irp->IoS.Status = SUCCESS;
        irp->IoS.Info = req->Status;
      } else {
        irp->IoS.Status = FAIL;
        irp->IoS.Info = req->Status;
      }
      SmartDevFreeBlock(req);
      IoCompleteRequest(irp);
      nPackets++;
    }
   } while(nPackets!=nPacketsOld);
   KeReleaseSpinLock();
C: KeReleaseSpinLock_return();
 }
```

Analysis Engine: Abstract, Check & Refine

```
void example() {
   do {
    KeAcquireSpinLock();
A:  KeAcquireSpinLock_return();
    nPacketsOld = nPackets;
    req = devExt->WLHV;
    if(req && req->status){
      devExt->WLHV = req->Next;
      KeReleaseSpinLock();
B:    KeReleaseSpinLock_return();
      irp = req->irp;
      if(req->status > 0){
        irp->IoS.Status = SUCCESS;
        irp->IoS.Info = req->Status;
      } else {
        irp->IoS.Status = FAIL;
        irp->IoS.Info = req->Status;
      }
      SmartDevFreeBlock(req);
      IoCompleteRequest(irp);
      nPackets++;
    }
   } while(nPackets!=nPacketsOld);
   KeReleaseSpinLock();
C: KeReleaseSpinLock_return();
 }
```

```
    void example() {
     do {
      ;
A:    KeAcquireSpinLock_return();
      b2 = false;
      ;
      if (SdvMakeChoice()) then
        ;
        ;
B:      KeReleaseSpinLock_return();
        ;
        if (SdvMakeChoice()) {
          ;
          ;
        } else {
          ;
          ;
        }
        ;
        ;
        b2 = !b2 ? true : SdvMakeChoice();
      }
     } while (b2);
     ;
C:   KeReleaseSpinLock_return();
    }
```

Analysis Engine: Architecture



False Positives

- Inaccurate stub code
- Inaccurate harness code
  - Limited set of execution sequences of exported functions
- Inaccurate API rules

Outline

- API Usage Rules
- Environment Model
- Analysis Engine
- **Automatic Abstraction**

Automatic Abstraction

- Given
  - a C program: P
  - a set of predicates: E
- Produce a *boolean program* BP(P,E)
  - Same control flow
  - Only boolean type
  - Only |E| boolean variables = state of BP(P,E)
  - Model how every statement in P modifies the value of the predicates in E

Automatic Abstraction

- For every $s \in P$ establish the value of every $\phi_i \in E$ after the execution of s

P	BP(P,E)
…	…
s: x = e	$\phi_{i\_aft\_s} = \phi_{i\_bef\_s}$ [e/x]
…	…

- How to establish the value of $\phi_{i\_aft\_s}$, since we only know $\phi_{j\_bef\_s}$, j=1..n ?

Predicate Strengthening of $\phi_{i\_aft\_s}$

- Consider all possible conjunctions C_k of $\phi_{j\_bef\_s}$ or $!\phi_{j\_bef\_s}$, j=1..n
- Select those that imply $\phi_{i\_aft\_s}$=true: $F(\phi_{i\_aft\_s}, \phi_j)$
  - Exponential no. of calls to a theorem prover

- We obtain an approximation of $\phi_{i\_aft\_s}$=true only in terms of $\phi_{j\_bef\_s}$, j=1..n
- Similarly, we compute $F(!\phi_{i\_aft\_s}, \phi_j)$ as an approximation for $\phi_{i\_aft\_s}$=false

Abstraction of *s*

- Update every ϕ_i:

  $\phi_{i\_aft\_s}$ = true, if $F(\phi_{i\_aft\_s}, \phi_j)$ holds

  = false, if $F(!\phi_{i\_aft\_s}, \phi_j)$ holds

  = unknown, otherwise

Assignments and Pointer Aliasing

- For s: x=e, $\phi_{i\_aft\_s} = \phi_{i\_bef\_s}$ [e/x]

- If y is a location mentioned by $\phi_{i\_bef\_s}$
  - $\phi_{i\_aft\_s}$ = (&x = &y ^ $\phi_{i\_bef\_s}$ [e/x,e/y]) V
    (&x != &y ^ $\phi_{i\_bef\_s}$ [e/x])

Conditionals

```
if( e ){
        …
} else {
        …
}
```

```
if( * ){ assume( G(e, ϕj))
        …
} else { assume( G(!e, ϕj))
        …
}
```

- $G(e, \phi_j)$ is the set of conjunctions C_k over ϕ_j, j=1..n such that e implies C_k
- Taking one of the branches effectively updates the state of *BP(P,E)*

Procedure Calls

- 1st pass - determine the signature of each procedure R
  - E_f Input predicates – dependent only on parameters
  - E_r Output predicates – dependent on
    - » Return value
    - » Global variables
    - » Dereferencing of parameters

Procedure Calls

- 2nd pass - Abstract procedure call to R from S
  - Parameter passing = assignments
  - On return to S
    - update state with all the output predicates
    - update predicates dependent on
      - » Return value
      - » Global variable
      - » Argument dereferenced by R

Optimizations

- Limit the number of calls to the theorem prover
  - Consider conjunctions C_k of increasing length
  - Consider conjunctions C_k of a transitive closure of predicates ϕ_j dependent on variables in $\phi_{i\_aft\_s}$
  - For s: x = e, update only $\phi_{i\_aft\_s}$ dependent on x

Conclusion

- Proposed framework model checks C programs based on boolean abstraction
- Results dependent on the power of the:
    - Theorem prover
    - Pointer alias analysis

QA

- Abstracting nested calls to API? 4
- Rewrite APIs/API usage rules as to avoid all bugs? 10
- Develop self describing code s.t. we have API rule extraction? 9
- What guarantees are implied by the API usage rules? 3
- Abstract the most state from P, what do you get? 4