Safety Checking of Machine Code

Zhichen Xu, Barton Miller and Thomas Reps
zhichen@cs.wisc.edu

Computer Science Department
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706-1685

Motivation

- Two prevailing trends:
  - Dynamic extensibility
    - Operating systems: custom policies (VINO, SPIN)
    - Performance Tools: measurement code (kernInst, Paradyn)
    - Databases: datablades
    - Web browsers: plug-ins
  - Component-based software (Java, COM)
    - Code from several sources, which could distrust each other

Safety of extensions and components is crucial

Related Work

Safety checking: nothing "bad" will happen

- Dynamic Techniques:
  - Hardware enforced address spaces, SFI, interpretation, etc.
- Hybrid Techniques
  - Safe languages: Java, ML, Modula 3, etc.

Recovery, runtime cost

- Static Techniques
  - Proof-Carry Code
  - Certifying Compiler, Typed-Assembly Language

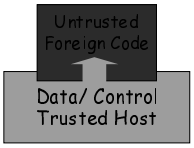
Restricts the things that can be expressed in the source language

Restricts the choices of language such as C, assembly

Building a certifying compiler is a complex task, avoid if possible

Our Approach

Safe code can be written in any language as long as nothing "bad" is expressed.



- Operate directly on binary code
  - Based on annotations on initial inputs
- Extend the host at a fine-grained level
  - Allow foreign code to manipulate the internal data structures of the host
- Extensibility:
  - Default collection of safety conditions, plus precise and flexible access policy
  - Naturally extends to the checking of security properties

Outline

- Motivation
- Related work
- Our approach
- Safety properties and safety policies
- Safety checking: basis
- Safety checking: analysis
- Initial experience
- Conclusion

Safety Properties

- Fine-grained memory protection
  - Array bounds, address alignment, initialized variables, valid pointer dereference, and stack manipulation.
- Safe interaction with the host
  - Call host functions with parameters of proper types, and with proper initializations
- Precise and flexible host access policy

We use: <type, state, access>, linear constraints

© 2000 zhichen@cs.wisc.edu

Type + State

- Typestate [Strom and Yemini' 1986]

  Use state to catch an uninitialized pointer

```
int *p, x=1;    //p is in an "uninitialized" state
*p=x ;          //dereference an uninitialized pointer
```

  Untrusted code releases lock before termination

```
mutex_lock(&lock)
...
mutex_unlock(&lock)        //lock will be "unlocked"
```


Type+State+Linear Constraints

- Null-pointer dereference

```
int *p=NULL, x=1;   //p is in an "initialized" state
*p = x ;            //Should also check p is valid pointer
```

- Array access bounds checking

```
int sum (int a[10]) {
        int s = 0;
        for (i=0; i<10; i++)
                s += a[i]; //Check that "i >= 0 && i < = 9"
}
```


Type+State+L. Constraints + Access

- Access permissions (least privilege)
  - Read(r), write(w), follow(f), execute(x)
- Example: kernel page-replacement extension: [Small and Seltzer, 1996]
  - Pick a non-hot page from global LRU list.

```
typedef struct _page_list {
  int page;                 //read-only access
  ...
  struct _page_list * next; //de-reference (follow access)
} page_list;
page_list* do_graft(page_list *candidate, page_list *hot)
```


Interaction with Host

- Access permission execute(x) specifies host functions (methods) that can be called
- Safety pre- and post- conditions specifies what is a safe call
  - Parameters are of the proper types and states

Example: JNI array accesses

```
jint Java_IntArray_sumArray(JNIEnv *env, jobject, jintArray arr) {
    jint *body =
    int i, sum = 0 ;
    body = (jint *)env->GetIntArrayElements(env, arr, 0);
    for (i=0; i<len; i++) {
      sum += body[i]; // do array bounds checking
    }
    return sum;
}
```

```
JNIEnv::GetIntArrayElements(env, arr)
    precondition: env: <JNIEnv*, initialized, r>
                  arr: <jintArray, initialized, r>
    postcondition: {retVal = arr.Elements}
```


Inputs to the Safety Checker

- A host-specified access policy

  <Region, Category, Access Permitted>

```
[Host : page_list.page : r]
[Host : page_list ptr, page_list.next : rf]

[Host : JNIEnv::GetIntArrayElements : x]
```

- Information about the initial inputs
  - A host typestate specification
    - Type and state of host data
    - Pre- and post conditions for host functions
  - A invocation specification

Safety Checking: Basis

- Intuition:
  - Figure out what each instruction in untrusted code does:
    - Based on initial inputs to untrusted code
  - Does it violate any safety properties?
    - Attach a safety precondition to each instruction
    - Check that each instruction obeys the precondition
- Formalizations
  - Abstract locations: registers, stack or heap allocated objects
  - Typestate : <type, state, access>
  - Operational semantics

Safety Checking: Analyses

1: Preparation — Produce initial annotations: information of initial inputs

2: Typestate propagation — Figure out typestate of each absLoc at each program point

3: Annotation — Facts, and safety preconditions (local and global)

4: Verifying Local Safety Preconditions

5: Verifying Global Safety Preconditions
Theorem Prover | Induction Iteration

A Running Example

Sum of array elements
[V : int[n] : rf] [V: int : r]

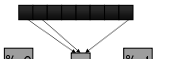
- initial annotation:

a: <int, i, r>,
%o0: <int[n], {a}, rwf>,
%o1: <int, i, rwf>

{%o1=n ∧ n≥0}

"a" is an abstract location that models the entire array

- Verify the safety of instruction at program point 6




```
1:  MOV 0,%o2
2:  CMP %o2, %o1
3:  BGE 11
4:  NOP
5:  SLL %o2,2,%g2
6:  LD [%o0+%g2],%g3
7:  ADD %o2,1,%o2
8:  CMP %o2, %o1
9:  BL 5
10: ADD %o3, %g3, %o3
11: RET
12: MOV %o3, %o0
```


Phase 2: Typestate Propagation

Finds out typestate of each absLoc at each program point

- a: <int, i, r>, %o0: <int [n], {a}, rwf>, %o1:<int, i, rwf>
- {%o1 = n ∧ n ≥ 0}

```
1:  MOV 0,%o2
2:  CMP %o2, %o1
3:  BGE 11
4:  NOP
5:  SLL %o2,2,%g2
6:  LD [%o0+%g2],%g3
7:  ADD %o2,1,%o2
8:  CMP %o2, %o1
9:  BL 5
10: ADD %o3, %g3, %o3
11: RET
12: MOV %o3, %o0
```


Phase 3: Annotation

- Find out safety requirements and facts

6: %o0: <int[n], {a}, rwf>; %g2 : <int, i, rwf>
%g3 = %o0[%g2/4] // %g3 = a

Facts: a mod 4=0

Local Safety Precondition:
assignable(a, %g3)

Global Safety Preconditions:
align(a+%g2,4)
inbounds(int[n],0,n,%g2)

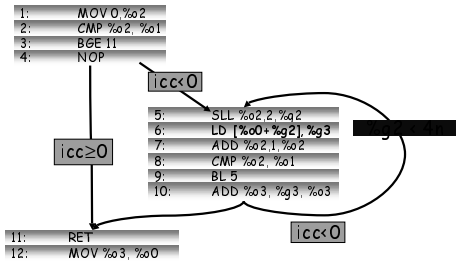
```
1:  MOV 0,%o2
2:  CMP %o2, %o1
3:  BGE 11
4:  NOP
5:  SLL %o2,2,%g2
6:  LD [%o0+%g2],%g3
7:  ADD %o2,1,%o2
8:  CMP %o2, %o1
9:  BL 5
10: ADD %o3, %g3, %o3
11: RET
12: MOV %o3, %o0
```


Phase 5: Verifying Global Safety Preconditions

- Floyd-style verification conditions
- Induction iteration method to synthesize loop invariants [Susuki and Ishihata, 1977]
  - Uses the weakest liberal precondition (wlp) of while statement to synthesize loop invariant inductively
  - Totally mechanical
  - Suitable to verify linear constraints
  - We have extended it to synthesize invariants for natural loops

Verifying Global Preconditions

Example: Proves that %g2 is less than array upper bound n at program point 6 in two iterations



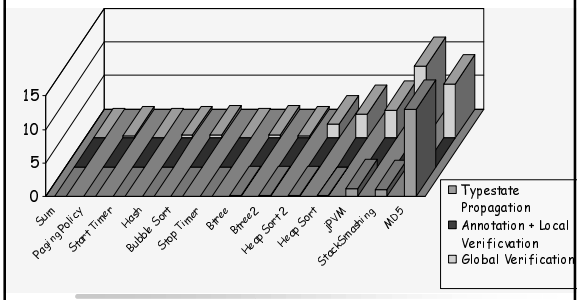
Initial Experience

- Test cases
  - Array sum, start/stop timer, b-tree,
  - kernel paging policy, hash, bubble sort, heap sort,
  - stack-smashing, MD5, jPVM
- Summary of Results
  - Finds safety violations in kernel policy, stack-smashing
  - Verifies all conditions, except for some calls in MD5, jPVM (precision lost due to array reference)
  - Checking time vary from 0.1 to 20 seconds

Characteristics of Test Cases

Number of Each Feature	Sum	Paging Policy	Start Timer	Hash	Bubble Sort	Stop Timer	Btree	Btree 2	Heap Sort 2	Heap Sort	jPVM	Stack Smashing	MD5
Instructions	13	20	22	25	25	36	41	51	71	95	157	309	883
Branches	2	5	1	4	5	3	11	11	9	16	12	89	11
Loops (Inner)	1	2(1)	0	1	2(1)	0	2(1)	2(1)	4(2)	4(2)	3	7(1)	5(2)
Procedure Calls (Trusted)	0	0	1(1)	1	0	2(2)	0	4	3	0	21 (21)	2	6
Global Safety Conditions	4	9	13	14	19	17	41	42	56	84	57	162	135

Timing



Conclusion

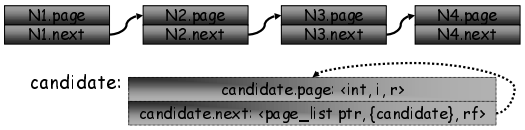
- Can certify object code produced by commodity compiler
  - Only requires annotations to the inputs of untrusted code
- Extensible:
  - host-specified access policy,
  - naturally extends to the checking of security properties
- Initial experience promising

Limitations

- Can only ensure safety properties that can be expressed using typestate + linear constraints
  - e.g., cannot prove termination
- Limitations in handling of array
  - Detecting bounds of local array and array in structure
  - Lost precision
- Inherited limitations of static techniques
  - Must reject code that can not be checked statically
  - Otherwise, there is the recovery problem

Abstract Location + Typestate

- Abstract Location: name, size, alignment, writable
- Typestate: <type, state, access>
  - Example:



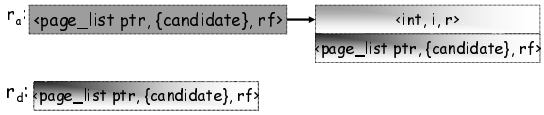
Abstract Operational Semantics

Example: Figure out the typestate of r_d of "LD [r_a+ n], r_d"

LD [r_a+ 4], r_d

```
typedef struct {
  int page;
  page_list * next;
} page_list;

page_list candidate;
```



r_d: <page_list ptr, {candidate}, rf>

Attachment of Safety Predicates

- Annotate the instruction with safety predicates
  - Local predicates and Global predicates+facts

  Example LD [r_a+n], r_d;
  - Local Predicates
    - There exists a field at offset n and of size 4
    - r_a is followable
    - It is legal to assign the fields pointed to by 'ra+n' to r_d
  - Global predicates:
    - The addresses stored in r_a is properly aligned,
    - and non-null