A Portable Virtual Machine Target for Proof-Carrying Code

Michael Franz,
Deepak Chandra, Andreas Gal,
Vivek Haldar, Ning Wang
University of California, Irvine

Fermin Reig
University of Nottingham

Christian W. Probst
Technical University of Denmark
Overview of the talk

• Overview of current safe code systems
  – Virtual Machines
  – Proof Carrying Code

• Overview of our architecture
  – Scalars
  – Objects
  – Typemaps

• Conclusions & Outlook
Overview of the talk

• Overview of current safe code systems
  – Virtual Machines
  – Proof Carrying Code

• Overview of our architecture
  – Scalars
  – Objects
  – Typemaps

• Conclusions & Outlook
Virtual Machines

• considerable verification effort
• verification inhibits optimizations at code producer
  – redundancy of type checks
  – array bounds
  – common sub-expression elimination
Overview of the talk

• Overview of current safe code systems
  – Virtual Machines
  – Proof Carrying Code

• Overview of our architecture
  – Scalars
  – Objects
  – Typemaps

• Related work
Proof Carrying Code

• code producer
  – computes verification condition VC
  – generates proof for transmitted code that discharges VC
• code consumer re-computes verification condition
  – VC must be discharged by proof
• safety policies are fixed
  – must be known at proof-generation time
• very low level of reasoning
  – typically, large portions of a proof re-establish typing of data (integers are integers are not booleans and are not pointers)
Proof Carrying Code

• proof shortcuts verification
  – stack maps in the Java KVM

• verification at the code producer
  – reduces TCB and verification effort at the code consumer

• Certifying Compilation
  – SafeC to Alpha assembly code
  – Java bytecode to x86

• not portable: platform-specific binaries are certified
Overview of the talk

• Overview of current safe code systems
  – Virtual Machines
  – Proof Carrying Code

• Overview of our architecture
  – Scalars
  – Objects
  – Typemaps

• Conclusion & Outlook
Challenge

Is there a middle ground such that

– The architecture level is low enough so that most optimizations are possible

– At the same time has high enough semantic level so that the complexity of proofs is reasonable
PCC

HLL

static compilation proof generation

machine code
static compilation proof generation

HLL

machine code

PCC

HLL

JVM

dynamic compilation

HLL

JVM

machine code

static compilation proof generation

HLL

machine code
This work introduces a method for generating proofs of correctness for high-level languages (HLL) using a virtual machine (VM) and dynamic compilation. The process involves:

1. **PCC**:
   - HLL (High-Level Language)
   - Static compilation proof generation
   - Machine code

2. **This work**:
   - HLL (High-Level Language)
   - Proof generation
   - VM (Virtual Machine)
   - Dynamic compilation
   - Machine code

3. **JVM**:
   - HLL (High-Level Language)
   - Dynamic compilation
   - Machine code

The diagram illustrates the flow from specification to machine code, highlighting the role of dynamic compilation and proof generation in ensuring the correctness of the compiled code.
Overview: This Architecture

- Typed registers
- Separation of pointers and data registers
- Balance between
  - safe by construction vs unsafe operations
- Safety by proofs for unsafe operations
  - type maps
Splitting the burden of proof

Safety = type-safety + memory safety

- Scalar types: By construction of the VM
- Reference types: Using easily verifiable typemaps
- By construction of the VM
Overview of the talk

• Overview of current safe code systems
  – Virtual Machines
  – Proof Carrying Code

• Overview of our architecture
  – Scalars
  – Objects
  – Typemaps

• Conclusion & Outlook
Typed registers

- Dedicated register planes for each primitive type: integer, boolean, float, ...
- Operations on scalar registers are inherently safe and require no further proof
- Pointers are modeled as separate type
- A type map is used to prove safety of memory access operation
Inherently Safe Instructions

• Designed so that scalar operations cannot be unsafe
  − e.g.
    • $add \, R_i \, R_j \, R_k$
    • $R_i \, R_j \, R_k$ have to be in the same register plane
    • No need for proofs

• We measured that the fraction of Java bytecode that operated on primitive type and thereby require no proofs
  − Java Grande: average: 24% min: 5% max: 56%
  − specJVM98: average: 29% min: 22% max: 43%
Type-safety for Scalars

```plaintext
procedure fact (n : int) : int % n in i1
begin
    f : int;
    f := 1;
    while (n > 0) do
        f := f * n;
        n := n - 1;
    end;
    fact := f;
end

% n in i1

icnst 1, i0
loophhead:
icnst 0, i3
bls i3, i1, b0
brfalse b0, loophpnd
imul i0, i1, i0
icnst 1, i3
isub i1, i3, i1
goto loophpnd
loophpnd:

% return value in i0
```
Overview of the talk

- Overview of current safe code systems
  - Virtual Machines
  - Proof Carrying Code

- Overview of our architecture
  - Scalars
  - Objects
  - Typemaps

- Conclusion & Outlook
Object Layout

- instruction set guarantees memory safety
- single object creation statement
  \[ p = \text{new} \ (t, \ i) \]
  - allocates \( i \) elements of object with tag \( t \)
- allocated objects are tagged with \( t \)
- VM ensures that tag can not be overwritten
- tag can be read and checked
Object Layout

• each type has
  – a type tag
  – a characteristic tuple (scalar, pointer)
    • consists of the size of the scalar section and the pointer section of the object
  – a structure for the pointer section
    • states possible runtime types for entries in the pointer section
Necula’s Example Program [POPL 1997]

datatype T = Int of int | Pair of int * int
fun sum (l : T list) =
    let
        fun foldr f nil a = a
            | foldr f (h::t) a = foldr f t (f(a, h))
    in
        foldr (fn (acc, Int i) => acc + I
            | (acc, Pair (i, j)) => acc + i + j)
            l 0
Necula’s Example Program [POPL 1997]

datatype T = Int of int | Pair of int * int

fun sum (l : T list) =
  let
    fun foldr f nil a = a
    | foldr f (h::t) a = foldr f t (f(a, h))
  in
    foldr (fn (acc, Int i) => acc + I
    | (acc, Pair (i, j)) => acc + i + j)
      l 0
Example Heap

![Example Heap Diagram]

1 1 1

2 17 (4,0)

3 (0,4)

2 42 (4,0)

4 4 21 (8,0)

1 (0,8)

1 (0,8)

1 (0,8)
Guard Statements

- **checklen(p,i)**
  - guarantee that array accesses are within bounds

- **checknonnull(p)**
  - check that p is not null

- **checktag(p,t)**
  - type check
  - check that the object pointed to by p has tag t
Overview of the talk

- Overview of current safe code systems
  - Virtual Machines
  - Proof Carrying Code

- Overview of our architecture
  - Scalars
  - Objects
    - Typemaps

- Conclusion & Outlook
Typemaps

- indicate information about values at basic block entries
  - type
  - not null-ness
  - ...

- Verification - one pass
  - Derive typemap at end of block
  - Across basic block boundaries, check for type-compatibility of typemaps
i1 = iconst(0)
bls(i0, i1, b0)
brfalse(b0, Lless)

p0 = new(2, 1)
goto(Lrest)

Lless:
p0 = new(2, 1)

i0 = iload(2, p0, 0)
Necula’s Example Program [POPL 1997]

datatype T = Int of int | Pair of int * int

fun sum (l : T list) = 
  let 
    fun foldr f nil a = a 
      | foldr f (h::t) a = foldr f t (f(a, h))
  in 
    foldr (fn (acc, Int i) => acc + I 
      | (acc, Pair (i, j)) => acc + i + j)  
    l 0 

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>tag</td>
<td>layout</td>
<td>structure</td>
</tr>
<tr>
<td>---------</td>
<td>-----</td>
<td>-------</td>
<td>----------------</td>
</tr>
<tr>
<td>T list</td>
<td>1</td>
<td>[0,8]</td>
<td>⟨{2,3}, {1}⟩</td>
</tr>
<tr>
<td>Int of int</td>
<td>2</td>
<td>[4,0]</td>
<td>⟨⟩</td>
</tr>
<tr>
<td>Pair of int*int</td>
<td>3</td>
<td>[0,4]</td>
<td>⟨{4}⟩</td>
</tr>
<tr>
<td>int*int</td>
<td>4</td>
<td>[8,0]</td>
<td>⟨⟩</td>
</tr>
</tbody>
</table>
Overview of the talk

• Overview of current safe code systems
  – Virtual Machines
  – Proof Carrying Code
• Overview of our architecture
  – Scalars
  – Objects
  – Typemaps
• Conclusion & Outlook
Conclusion

- PPC and VMs are two extremes with respect to independency from the actual hardware targeted
- our approach tries to unify the best of the two worlds
  - proofs to ease the burden of verification
  - VM support for guaranteeing memory and type safety
Outlook

• extend safety supported by the VM to ease proofs
• extend annotations to
  – cover more detailed type information
  – cover other analysis information
    • must be easily checkable at the code consumer side