TALx86: A Realistic Typed Assembly Language

Dan Grossman, Fred Smith
Cornell University

Joint work with: Greg Morrisett,
Karl Crary (CMU), Neal Glew, Richard Samuels,
Dave Walker, Stephanie Weirich, Steve Zdancewic
Everyone wants extensibility:

- Web browser
  - applets, plug-ins
- OS Kernel
  - packet filters, device drivers
- “Active” networks
  - service routines
- Databases
  - extensible ADTs
The Language Approach

Extension is written in a “safe” language:

• Java, Modula-3, ML, Scheme
• Key point: language provides abstractions
  • ADTs, closures, objects, modules, etc.
  • Can be used to build fine-grained capabilities

Host ensures code respects abstractions:

• Static checking (verification)
• Inserting dynamic checks (code-rewriting)
Example: Your Web Browser

Java Source

javac

JVM bytecode

JVM verifier

Low-Level IL

Optimizer

Binary

Browser

System Interface

System Binary
JVM Pros & Cons

Pros:
- Portability
- Hype: $, tools, libraries, books, training

Cons:
- Performance
  - unsatisfying, even with off-line compilation
- Only really suitable for Java (or slight variants):
  - relatively high-level instructions tailored to Java
  - type system is Java specific
- and...
No complete formal model

Must insert right checks

Good code $\Rightarrow$ big optimizer $\Rightarrow$ bugs in optimizer

Lots of “native” methods (i.e., not-safe code)
Ideally:

Your favorite language

Low-Level IL

optimizer

machine code

trusted computing base

verifier

System Interface

System Binary
The Types Way

Your safe language

Typed Low-Level IL

Typed optimizer

TAL

trusted computing base

Verifier

System Interface

System Binary

- Verifier is a type-checker
- Type system flexible and expressive
- A useful instantiation of the "proof carrying code" framework
TALx86 in a Nutshell

- Most of the IA32 80x86 flat model assembly language
- Memory management primitives
- Sound type system
- Types for code, stacks, structs
- Other advanced features
- Future work (what we can’t do yet)
TALx86 Basics:

Primitive types: (e.g., \texttt{int})

Code types: \{r_1: \tau_1, \ldots, r_n: \tau_n\}

• "I'm code that requires register $r_i$ to have type $\tau_i$ before you can jump to me."

• Code blocks are annotated with their types
  • Think pre-condition
  • Verify block assuming pre-condition
Sample Loop

C:

```c
int sum(int n){
    int s=0;
    while(!n) {
        s+=n;
        --n;
    }
    return n;
}
```

TAL sketch:

```talc
<n and retn addr as input>
sum: <type>
    <initialize s>
loop: <type>
    <add to s, decrement n>
test: <type>
    <return if n is 0>
```
Verification

sum: \( \{\text{ecx}: \text{int}, \text{ebx}: \{\text{edx}: \text{int}\}\} \)

\text{mov eax,0} \ \{\text{ecx}: \text{int}, \text{ebx}: \{\text{edx}: \text{int}\}, \text{eax}: \text{int}\}

\text{jmp test} \quad \text{OK: sub-type of type labeling test}

loop: \( \{\text{ecx}: \text{int}, \text{ebx}: \{\text{edx}: \text{int}\}, \text{eax}: \text{int}\} \)

\text{add eax,ecx} \ \{\text{ecx}: \text{int}, \text{ebx}: \{\text{edx}: \text{int}\}, \text{eax}: \text{int}\}

\text{dec ecx} \quad \{\text{ecx}: \text{int}, \text{ebx}: \{\text{edx}: \text{int}\}, \text{eax}: \text{int}\}

\text{OK: sub-type of type labeling next block}

test: \( \{\text{ecx}: \text{int}, \text{ebx}: \{\text{edx}: \text{int}\}, \text{eax}: \text{int}\} \)

\text{cmp ecx,0} \ \{\text{ecx}: \text{int}, \text{ebx}: \{\text{edx}: \text{int}\}, \text{eax}: \text{int}\}

\text{jne loop} \quad \text{OK: sub-type of type labeling loop}

\text{mov edx,eax} \quad \{\text{ecx}: \text{int}, \text{ebx}: \{\text{edx}: \text{int}\}, \text{eax}: \text{int}, \text{edx}: \text{int}\}

\text{jmp ebx} \quad \text{OK: sub-type of \{edx:int\} -- type of ebx}
Stack Types (lists):

$$\sigma ::= \text{nil} \mid \tau::\sigma \mid \rho$$

where $\rho$ is a stack type variable.

Examples using C calling convention:

```c
int square(int);  int mult(int,int);
```

$$\forall \rho_1 \{ \text{esp: } \tau_1::\text{int::}\rho_1 \}$$

where

$$\tau_1 = \{ \text{eax: int, esp: int::}\rho_1 \}$$

$$\forall \rho_2 \{ \text{esp: } \tau_2::\text{int::int::}\rho_2 \}$$

where

$$\tau_2 = \{ \text{eax: int, esp: int::int::}\rho_2 \}$$
Stacks & Verification

square: \( \forall \rho_1 \{ \text{esp: } \tau_1::\text{int::} \rho_1 \} \)

where \( \tau_1 = \{ \text{eax: int, esp: int::} \rho_1 \} \)

push \[ \text{esp+4} \]

push \[ \text{esp+8} \]

call mult[with \( \rho_2 = \tau_1::\text{int::} \rho_1 \)]

\( \tau_{\text{aft}} = \{ \text{eax:int, esp: int::int::} \tau_1::\text{int::} \rho_1 \} \)

add esp,8

retn

mult: \( \forall \rho_2 \{ \text{esp: } \tau_2::\text{int::} \rho_2 \} \)

where \( \tau_2 = \{ \text{eax: int, esp: int::int::} \rho_2 \} \)

\( \{ \text{esp: } \tau_2::\text{int::int::} \tau_1::\text{int::} \rho_1 \} \)

where \( \tau_2 = \{ \text{eax: int, esp: int::int::} \tau_1::\text{int::} \rho_1 \} \)

1 May 1999
Important Properties

• Abstraction

“Because the type of the rest of the stack is abstract the callee cannot read/write this portion of the stack”

• Flexibility

Can encode and enforce many calling conventions (stack shape on return, callee-save, tail calls, etc.)
Callee-Save Example

\texttt{mult:}

\[ \forall \alpha \ \forall \rho_2 \{ \text{ebp: } \alpha, \ \text{esp: } \tau_2::\text{int}::\text{int}::\rho_2 \} \]

\text{where } \tau_2=\{ \text{ebp: } \alpha, \ \text{eax: } \text{int}, \ \text{esp: } \text{int}::\text{int}::\rho_2 \} \]
**Structs**

- **Goals:**
  - Prevent reading uninitialized fields
  - Permit flexible scheduling of initialization
- **MALLOC “instruction”**
  - returns uninitialized record
- **Type of struct tracks initialization of fields**
- **Example:**
  
  ```
  {ecx: int}
  MALLOC eax,8 [int,int] ; eax : ^*[int^u, int^u]
  mov [eax+0], ecx ; eax : ^*[int^rw,int^u]
  mov ecx, [eax+4] ; type error!
  ```
Much, much more

• Arrays (see next slide)
• Tagged Unions
• Displays, Exceptions [TIC'98]
• Static Data
• Modules and Interfaces [POPL'99]

• Run-time code generation
  [PEPM'99 Jim, Hornof]
Mis-features

• MALLOC and garbage collection in trusted computing base [POPL’99]

• No way to express aliasing

• No array bounds check elimination [Walker]

• Object/class support too primitive [Glew]
Summary and Conclusions

• We can type real machine code
  Potential for
  performance + flexibility + safety

• Challenge:
  Finding generally useful abstractions

• Lots of work remains

http://www.cs.cornell.edu/talc