Overview

Slogan: “Safety through types”

- An architecture for safe mobile code
  - Download annotated binaries from an untrusted code producer
  - Verify code using a trusted typechecker
  - Link and execute without errors

- Security properties hinge on understanding behavior
  - Must reason precisely about programs
  - Define “good” and “bad” behaviors
  - Identify and rule out “bad programs”

- Typed Assembly Language (TAL) is a framework that accomplishes these goals in a setting where the programs in question are x86 executables
Schedule

Today
- Typed Assembly Language
- Prelim #2 hand back

Wednesday
- Polymorphism
- Stack Types

Friday
- Compilation
- Course Review
Acknowledgments

- These lectures developed by David Walker (Princeton)
- They describe Typed Assembly Language, a project at Cornell led by Greg Morrisett about 15 years ago
What is TAL?

In Theory

- A RISC-like assembly language
- A formal operational semantics
- A family of type systems that capture key safety properties of registers, stack, and the heap
- Rigorous proofs of soundness which demonstrate that TAL enforces security guarantees
What is TAL?

In Theory

- A RISC-like assembly language
- A formal operational semantics
- A family of type systems that capture key safety properties of registers, stack, and the heap
- Rigorous proofs of soundness which demonstrate that TAL enforces security guarantees

In Practice

- A typechecker for almost all of the Intel IA32 architecture
- A collection of tools for assembling linking, etc. TAL binaries
- A compiler for a safe C-like language called Popcorn
Example

High-level code:

```
fact (n,a) =
  if (n ≤ 0) then a
  else fact(n-1,a × n)
```

Assembly code:

```
% r₁ holds n, r₂ holds a, r₃₁ holds return address
fact:   ble r₁,L2  % if n ≤ 0 goto L2
       mul r₂,r₂,r₁ % a := a × n
       sub r₁,r₁,1  % n := n - 1
       jmp fact     % goto fact
L2 :    mov r₁,r₂ % result := a
       jmp r₃₁     % return
```
TAL Syntax

Models a simple RISC-like assembly language.

- Registers: \( r \in \{ r_1, r_2, r_3, \ldots \} \)
TAL Syntax

Models a simple RISC-like assembly language.

- Registers: \( r \in \{r_1, r_2, r_3, \ldots \} \)
- Labels: \( L \in \text{Identifier} \)
TAL Syntax

Models a simple RISC-like assembly language.

- Registers: $r \in \{r_1, r_2, r_3, \ldots\}$
- Labels: $L \in \text{Identifier}$
- Integers: $n \in [-2^{k-1} \ldots 2^{k-1})$
TAL Syntax

Models a simple RISC-like assembly language.

- Registers: \( r \in \{ r_1, r_2, r_3, \ldots \} \)
- Labels: \( L \in \text{Identifier} \)
- Integers: \( n \in [-2^{k-1} \ldots 2^{k-1}) \)
- Blocks: \( B ::= i; B \mid \text{jmp } v \)
Models a simple RISC-like assembly language.

- Registers: \( r \in \{r_1, r_2, r_3, \ldots \} \)
- Labels: \( L \in \text{Identifier} \)
- Integers: \( n \in [-2^{k-1} \ldots 2^{k-1}) \)
- Blocks: \( B ::= i; B \mid \text{jmp } v \)
- Instructions: \( i ::= aop \ r_d, r_s, v \mid bop \ r, v \mid \text{mov } r, v \)
Models a simple RISC-like assembly language.

- Registers: \( r \in \{r_1, r_2, r_3, \ldots \} \)
- Labels: \( L \in \text{Identifier} \)
- Integers: \( n \in \left[-2^{k-1} \ldots 2^{k-1}\right] \)
- Blocks: \( B ::= i; B \mid \text{jmp } v \)
- Instructions: \( i ::= \text{aop } r_d, r_s, v \mid \text{bop } r, v \mid \text{mov } r, v \)
- Operands: \( v ::= r \mid L \mid v \)
TAL Syntax

Models a simple RISC-like assembly language.

- Registers: \( r \in \{r_1, r_2, r_3, \ldots \} \)
- Labels: \( L \in \text{Identifier} \)
- Integers: \( n \in \left[ -2^{k-1} \ldots 2^{k-1} \right] \)
- Blocks: \( B ::= i; \; B \mid \text{jmp } v \)
- Instructions: \( i ::= \text{aop } r_d, r_s, v \mid \text{bop } r, v \mid \text{mov } r, v \)
- Operands: \( v ::= r \mid L \mid v \)
- Arithmetic Operations: \( \text{aop ::= add | sub | mul | \ldots} \)
Models a simple RISC-like assembly language.

- Registers: $r \in \{r_1, r_2, r_3, \ldots \}$
- Labels: $L \in \text{Identifier}$
- Integers: $n \in [-2^{k-1}, \ldots, 2^{k-1}]$
- Blocks: $B ::= i; B \mid \text{jmp } v$
- Instructions: $i ::= aop r_d, r_s, v \mid bop r, v \mid \text{mov } r, v$
- Operands: $v ::= r \mid L \mid v$
- Arithmetic Operations: $aop ::= \text{add} \mid \text{sub} \mid \text{mul} \mid \ldots$
- Branch Operations: $bop ::= \text{beq} \mid \text{bgt} \mid \ldots$
TAL Abstract Machine

Model evaluation using a transition function $\Sigma \mapsto \Sigma'$ from machine states to machine states
TAL Abstract Machine

Model evaluation using a transition function $\Sigma \mapsto \Sigma'$ from machine states to machine states

- Machine states: $\Sigma = (H, R, B)$
TAL Abstract Machine

Model evaluation using a transition function $\Sigma \mapsto \Sigma'$ from machine states to machine states

- Machine states: $\Sigma = (H, R, B)$
- The heap $H$ is a partial map from labels $L$ to blocks $B$
TAL Abstract Machine

Model evaluation using a transition function $\Sigma \mapsto \Sigma'$ from machine states to machine states

- Machine states: $\Sigma = (H, R, B)$
- The heap $H$ is a partial map from labels $L$ to blocks $B$
- The register file $R$ maps registers to values. Abusing notation slightly, we extend $R$ to a map on values as follows:

\[
R(n) = n \\
R(L) = L \\
R(r) = v \quad \text{if } R = \{\ldots, r \mapsto v, \ldots\}
\]
TAL Abstract Machine

Model evaluation using a transition function $\Sigma \mapsto \Sigma'$ from machine states to machine states

- Machine states: $\Sigma = (H, R, B)$
- The heap $H$ is a partial map from labels $L$ to blocks $B$
- The register file $R$ maps registers to values. Abusing notation slightly, we extend $R$ to a map on values as follows:

  $$
  \begin{align*}
  R(n) & = n \\
  R(L) & = L \\
  R(r) & = v \quad \text{if} \quad R = \{\ldots, r \mapsto v, \ldots\}
  \end{align*}
  $$

- The current block $B$ is the block associated to the (implicit) program counter
TAL Operational Semantics (Selected Rules)

\[(H, R, \text{mov } r_d, v; B) \mapsto (H, R[r_d := R(v)], B)\]
TAL Operational Semantics (Selected Rules)

\[(H, R, \text{mov } r_d, v; B) \mapsto (H, R[r_d := R(v)], B)\]

\[n = R(v) + R(r_s)\]

\[(H, R, \text{add } r_d, r_s, v; B) \mapsto (H, R[r_d := n], B)\]
TAL Operational Semantics (Selected Rules)

\[(H, R, \text{mov } r_d, v; B) \mapsto (H, R[r_d := R(v)], B)\]

\[n = R(v) + R(r_s)\]

\[(H, R, \text{add } r_d, r_s, v; B) \mapsto (H, R[r_d := n], B)\]

\[R(v) = L \quad H(L) = B\]

\[(H, R, \text{jmp } v) \mapsto (H, R, B)\]
TAL Operational Semantics (Selected Rules)

\[
(H, R, \text{mov } r_d, v; \; B) \mapsto (H, R[r_d := R(v)], B)
\]

\[
n = R(v) + R(r_s)
\]

\[
(H, R, \text{add } r_d, r_s, v; \; B) \mapsto (H, R[r_d := n], B)
\]

\[
R(v) = L \quad H(L) = B
\]

\[
(H, R, \text{jmp } v) \mapsto (H, R, B)
\]

\[
R(r) \neq 0
\]

\[
(H, R, \text{beq } r, v; \; B) \mapsto (H, R, B)
\]
TAL Operational Semantics (Selected Rules)

\[(H, R, \text{mov } r_d, v; B) \mapsto (H, R[r_d := R(v)], B)\]

\[n = R(v) + R(r_s)\]

\[(H, R, \text{add } r_d, r_s, v; B) \mapsto (H, R[r_d := n], B)\]

\[R(v) = L \quad H(L) = B\]

\[(H, R, \text{jmp } v) \mapsto (H, R, B)\]

\[R(r) \neq 0\]

\[(H, R, \text{beq } r, v; B) \mapsto (H, R, B)\]

\[R(r) = 0 \quad R(v) = L \quad H(L) = B'\]

\[(H, R, \text{beq } r, v; B) \mapsto (H, R, B')\]
Errors

- The machine is **stuck** if there does not exist a transition from the current state to some following state.

- We will use stuck states to define the “bad” behaviors that may occur at run-time.

- The type system will guarantee that well-typed machines never get stuck.

- Example stuck states:
  - \((H, R, \text{add } r_d, r_s, v; B)\) where \(r_s\) and \(v\) aren’t integers
  - \((H, R, \text{jmp } v)\) where \(v\) isn’t a label
  - \((H, R, \text{beq } r, v)\) where \(r\) isn’t an integer or \(v\) isn’t a label

- To distinguish integers and labels we need a type system!
Types

Syntax

- $\tau ::= \text{int} \mid \Gamma \rightarrow \{\}
- $\Gamma ::= \{r_1 : \tau_1, r_2 : \tau_2, \ldots\}$

Labels are like functions that take a record of arguments. Labels have types of the form $f_{r_1:1}; r_2:2; \ldots!fg$. To jump to code with this type, register $r_1$ must contain a value of type $1$, register $r_2$ must contain a value of type $2$, and so on. The order that register names appear is irrelevant. Note that functions never return—every block ends with a jmp.
Types

Syntax

- $\tau ::= \text{int} \mid \Gamma \rightarrow \{\}
- $\Gamma ::= \{r_1 : \tau_1, r_2 : \tau_2, \ldots\}$

Code Types

- Labels are like functions that take a record of arguments
- Labels have types of the form $\{r_1 : \tau_1, r_2 : \tau_2, \ldots\} \rightarrow \{\}$
- To jump to code with this type, register $r_1$ must contain a value of type $\tau_1$, register $r_2$ must contain a value of type $\tau_2$, and so on
- The order that register names appear is irrelevant
- Note that functions never return—every block ends with a jmp
Well-Typed Example

% $r_1$ holds n, $r_2$ holds a, $r_{31}$ holds return address

**fact:** \{ $r_1 : \text{int}$, $r_2 : \text{int}$, $r_{31} : \{ r_1 : \text{int} \} \rightarrow \{ \} \} \rightarrow \{ \}

ble $r_1$, L2  % if $n \leq 0$ goto L2
mul $r_2$, $r_2$, $r_1$  % $a := a \times n$
sub $r_1$, $r_1$, 1  % $n := n - 1$
jmp **fact**  % goto fact

L2: \{ $r_1 : \text{int}$, $r_2 : \text{int}$, $r_{31} : \{ r_1 : \text{int} \} \rightarrow \{ \} \} \rightarrow \{ \}

mov $r_1$, $r_2$  % result := a
jmp $r_{31}$  % return
Ill-Typed Example

% $r_1$ holds n, $r_2$ holds a, $r_{31}$ holds return address

**fact:** \( \{ r_1 : \text{int}, r_{31} : \{ r_1 : \text{int} \} \rightarrow \{ \} \} \rightarrow \{ \} \)

ble $r_1$, $L2$

mul $r_2$, $r_2$, $r_1$ \% Error! $r_2$ doesn’t have a type

sub $r_1$, $r_1$, 1

jmp $L1$ \% Error! No such label

$L2$ : \( \{ r_2 : \text{int}, r_{31} : \{ r_1 : \text{int} \} \rightarrow \{ \} \} \rightarrow \{ \} \)

mov $r_{31}$, $r_2$

jmp $r_{31}$ \% Error! $r_{31}$ not a label
Typechecking Overview

- Intuitively, the type system needs to keep track of:
  - The types of the registers at each point in the code
  - The types of the labels on the code

- Heap types: $\Psi$ maps labels to code types

- Register types: $\Gamma$ maps registers to types

- A family of typing (and subtyping) relations:
  - $\Psi; \Gamma \vdash v : \tau$
  - $\Psi \vdash i : \Gamma \rightarrow \Gamma'$
  - $\tau \leq \tau'$
  - $\vdash H : \Psi$
  - $\vdash R : \Gamma$
  - $\vdash (H, R, B)$
Typechecking Values

\[ \psi; \Gamma \vdash v : \tau \]
Typechecking Values

\[ \psi; \Gamma \vdash v : \tau \]

\[ \psi; \Gamma \vdash n : \text{int} \]
Typechecking Values

\[
\psi; \Gamma \vdash v : \tau
\]

\[
\psi; \Gamma \vdash n : \text{int}
\]

\[
\Gamma(r) = \tau \\
\psi; \Gamma \vdash r : \tau
\]
Typechecking Values

\[ \psi; \Gamma \vdash v : \tau \]

\[ \psi; \Gamma \vdash n : \text{int} \]

\[ \Gamma(r) = \tau \]

\[ \psi; \Gamma \vdash r : \tau \]

\[ \psi(L) = \tau \]

\[ \psi; \Gamma \vdash L : \tau \]
Subtyping

• A program won’t crash if the register file has more values that are needed to satisfy the typing conditions

• Formally, a register file with more components is a subtype of a register file with fewer components:

\[
\{ r_1 : \tau_1 , \ldots , r_i : \tau_i ; r_{i+1} : \tau_i + 1 \} \leq \{ r_1 : \tau_1 , \ldots , r_i : \tau_i \}
\]

Note that this is the ordinary rule for records!

• Code subtyping goes in the opposite direction: a label requiring \( r_1 \) and \( r_2 \) may be used as a label requiring \( r_1 , r_2 , \) and \( r_3 \).

\[
\Gamma' \leq \Gamma \\
\Gamma \to \{ \} \leq \Gamma' \to \{ \}
\]

Note that this is the ordinary contravariant rule for functions!
Subtyping

- Subtyping is also reflexive and transitive.

\[
\tau \leq \tau
\]

\[
\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3 \\
\hline
\tau_1 \leq \tau_3
\]

- A subsumption rule allows values to be used at supertypes:

\[
\Psi; \Gamma \vdash v : \tau_1 \\
\tau_1 \leq \tau_2 \\
\hline
\Psi; \Gamma \vdash v : \tau_2
\]
Typing Instructions

\[ \psi \vdash i : \Gamma_1 \rightarrow \Gamma_2 \]

- $\Gamma_1$ describes the registers before the execution of the instruction—a *precondition*

- $\Gamma_2$ describes the registers after the execution of the instruction—a *postcondition*

- $\psi$ is invariant. That is, the types of objects on the heap will not change (at least for now...)

Typing Instructions

\[ \psi \vdash i : \Gamma_1 \to \Gamma_2 \]

**Arithmetic operations**

\[
\frac{\psi ; \Gamma \vdash r_s : \text{int} \quad \psi ; \Gamma \vdash v : \text{int}}{\psi \vdash \text{aop } r_d, r_s, v : \Gamma \to \Gamma[r_d := \text{int}]} 
\]

**Conditional branches**

\[
\frac{\psi ; \Gamma \vdash r : \text{int} \quad \psi ; \Gamma \vdash v : \Gamma \rightarrow \{\}}{\psi \vdash \text{bop } r, v : \Gamma \rightarrow \Gamma} 
\]

**Data movement**

\[
\frac{\psi ; \Gamma \vdash v : \tau}{\psi \vdash \text{mov } r_d, v : \Gamma \rightarrow \Gamma[r_d := \tau]} 
\]
Typing Instructions

\[ \psi \vdash i : \Gamma_1 \rightarrow \Gamma_2 \]

Jumps

\[
\begin{align*}
\psi ; \Gamma & \vdash v : \Gamma \rightarrow \{\} \\
\hline
\psi & \vdash \text{jmp } v : \Gamma \rightarrow \{} 
\end{align*}
\]

Basic blocks

\[
\begin{align*}
\psi ; \Gamma & \vdash i : \Gamma_1 \rightarrow \Gamma_2 \\
& \psi ; \Gamma \vdash B : \Gamma_2 \rightarrow \{} \\
\hline
\psi & \vdash i; B : \Gamma_1 \rightarrow \{} 
\end{align*}
\]
Heap, Register File, and Machine Typing

Heaps

\[ \text{dom}(H) = \text{dom}(\psi) \quad \forall L \in \text{dom}(H). \, \psi \vdash H(L) : \psi(L) \]

\[ \vdash H : \psi \]

Register Files

\[ \forall r \in \text{dom}(\Gamma). \, \psi ; \{\} \vdash R(r) : \Gamma(r) \]

\[ \psi \vdash R : \Gamma \]

Machines

\[ \vdash H : \psi \quad \psi \vdash R : \Gamma \quad \psi \vdash B : \Gamma \to \{\} \]

\[ \vdash (H, R, B) \]
The type system satisfies the following theorem:

**Theorem (Type Safety)**

If $\vdash \Sigma$ and $\Sigma \xrightarrow{*} \Sigma'$, then $\Sigma'$ is not stuck.
The type system satisfies the following theorem:

**Theorem (Type Safety)**

\[ \Sigma \vdash \text{and } \Sigma \xrightarrow{\ast} \Sigma', \text{ then } \Sigma' \text{ is not stuck.} \]

**Proof:**
- **Progress:** if a state is well-typed, then it is not stuck
- **Preservation:** evaluation preserves types
The type system satisfies the following theorem:

**Theorem (Type Safety)**

If \( \vdash \Sigma \) and \( \Sigma \xrightarrow{\ast} \Sigma' \), then \( \Sigma' \) is not stuck.

**Proof:**

- Progress: if a state is well-typed, then it is not stuck
- Preservation: evaluation preserves types

**Corollary**

- Every jump in a well-typed program is to a valid label
- Every arithmetic operation in a well-typed program is done with integers—not labels!
Lemma

If \( \vdash H : \Psi \) and \( \Psi \vdash R : \Gamma \) and \( \Psi ; \Gamma \vdash v : \tau \) then

- \( \tau = \text{int} \) implies \( R(v) = n \)
- \( \tau = \{ r_1 : \tau_1, \ldots, r_k : \tau_k \} \rightarrow \{ \} \) implies \( R(v) = L \).

Moreover \( H(L) = B \) and \( \Psi \vdash B : \{ r_1 : \tau_1, \ldots, r_k : \tau_k \} \rightarrow \{ \} \)

Proof: by induction on typing derivations...
Progress (jmp Case)

Lemma

If $\vdash \Sigma_1$ then there exists a $\Sigma_2$ such that $\Sigma_1 \mapsto \Sigma_2$

$$
\vdash H : \psi \quad \psi \vdash R : \Gamma \quad \psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}
$$

$$
\vdash (H, R, \text{jmp } v)
$$
Lemma

If $\vdash \Sigma_1$ then there exists a $\Sigma_2$ such that $\Sigma_1 \rightarrow \Sigma_2$

$$
\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}
$$

$$
\vdash (H, R, \text{jmp } v)
$$

The third premise must be a derivation that ends in the rule:

$$
\Psi ; \Gamma \vdash v : \Gamma
$$

$$
\Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}
$$
Lemma

If $\vdash \Sigma_1$ then there exists a $\Sigma_2$ such that $\Sigma_1 \rightarrow \Sigma_2$

\[
\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash \text{jmp} \; \nu : \Gamma \rightarrow \{\}
\]

\[
\vdash (H, R, \text{jmp} \; \nu)
\]

The third premise must be a derivation that ends in the rule:

\[
\Psi; \Gamma \vdash \nu : \Gamma
\]

\[
\Psi \vdash \text{jmp} \; \nu : \Gamma \rightarrow \{\}
\]

By Canonical Forms, we have $R(\nu) = L$ and $H(L) = B'$. 
Lemma

If $\vdash \Sigma_1$ then there exists a $\Sigma_2$ such that $\Sigma_1 \hookrightarrow \Sigma_2$

$$
\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}
$$

$$
\vdash (H, R, \text{jmp } v)
$$

The third premise must be a derivation that ends in the rule:

$$
\Psi; \Gamma \vdash v : \Gamma \\
\Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}
$$

By Canonical Forms, we have $R(v) = L$ and $H(L) = B'$. Therefore:

$$
\begin{align*}
R(v) &= L \\
H(L) &= B' \\
(H, R, \text{jmp } v) &\hookrightarrow (H, R, B')
\end{align*}
$$
Preservation (jmp Case)

Lemma

If $\vdash \Sigma_1$ and $\Sigma_1 \leftrightarrow \Sigma_2$ then $\vdash \Sigma_2$

\[
\begin{array}{c}
\vdash H : \Psi \\
\Psi \vdash R : \Gamma \\
\Psi \vdash \text{jmp } \nu : \Gamma \rightarrow \{\}
\end{array}
\]

$\vdash (H, R, \text{jmp } \nu)$
Preservation (jmp Case)

Lemma

If $\vdash \Sigma_1$ and $\Sigma_1 \leftrightarrow \Sigma_2$ then $\vdash \Sigma_2$

\[
\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}
\]

\[
\vdash (H, R, \text{jmp } v)
\]

The third premise must be a derivation that ends in the rule:

\[
\Psi ; \Gamma \vdash v : \Gamma
\]

\[
\Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}
\]
Lemma

If $\vdash \Sigma_1$ and $\Sigma_1 \leftrightarrow \Sigma_2$ then $\vdash \Sigma_2$

$$
\begin{array}{c}
\vdash H : \Psi \\
\vdash R : \Gamma \\
\vdash \text{jmp } \nu : \Gamma \rightarrow \{\}
\end{array}
$$

$$
\vdash (H, R, \text{jmp } \nu)
$$

The third premise must be a derivation that ends in the rule:

$$
\begin{array}{c}
\Psi; \Gamma \vdash \nu : \Gamma \\
\end{array}
$$

$$
\vdash \text{jmp } \nu : \Gamma \rightarrow \{\}
$$

Moreover, the operational rule must be

$$
R(\nu) = L \quad H(L) = B'
$$

$$
(H, R, \text{jmp } \nu) \leftrightarrow (H, R, B')
$$
Preservation (jmp Case)

Lemma

If $\vdash \Sigma_1$ and $\Sigma_1 \leftrightarrow \Sigma_2$ then $\vdash \Sigma_2$

By Canonical Forms, we have $\Psi \vdash B : \Gamma \rightarrow \{\}$
Preservation (jmp Case)

Lemma

If \( \vdash \Sigma_1 \) and \( \Sigma_1 \mapsto \Sigma_2 \) then \( \vdash \Sigma_2 \)

By Canonical Forms, we have \( \Psi \vdash B : \Gamma \rightarrow \{\} \)

Therefore:

\[
\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash B : \Gamma \rightarrow \{\}
\]

\[
\vdash (H, R, B)
\]