Exercises (TD)
Compilation and Program Analysis (CAP)

1 Scheduling and register allocation

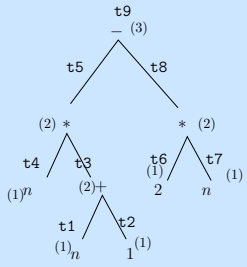
Consider the expression $E = ((a * (b + c)) - (2 * d))$. Assume:

- The RISC-V instruction set seen in course (with the multiplication `mul t1,t2,t3` that computes `t1 := t2*t3`, the addition `add t1,t2,t3` and the subtraction `sub t1,t2,t3`).
- a (resp. b, c, d) is stored in the stack slot referred as $[a]$ (resp. $[b]$, $[c]$, $[d]$) for the load instruction.

Question #1

Applying the Sethi-Ullman algorithm, give the minimum number of registers required to compute E.

Solution: Here is the tree of a similar expression, and the steps:



The minimum number of registers is therefore 3.

Question #2

Generate the corresponding 3-address code, using temporaries $t_1, t_2, \ldots$ (no register allocation).

Solution: Code in the following solution.

Question #3

Draw the liveness intervals, then check the register requirements.

Solution:

code	t1	t2	t3	t4	t5	t6	t7	t8	t9
`ld t1,[b]`									
`ld t2,[c]`	*								
`add t3,t1,t2`	*	*							
`ld t4,[a]`			*						
`mul t5,t4,t3`			*	*					
`li t6,2`					*				
`ld t7,[d]`					*	*			
`mul t8,t6,t7`					*	*	*		
`sub t9,t5,t8`					*		*		
`...`								*	

The red stars confirm that 3 registers will be ok.

Question #4

We now suppose we have an instruction `muladd r1, r2, r3, r4` that executes `r1 := r2 * r3 + r4`. Update your RISC-V code to take advantage of this new instruction.

Solution: We replace the `li t6, 2` with `li t6, -2` so that muladd can be used on a, $b + c$, and $(-2) * d$ (or on -2, d and $a * (b + c)$).

2 Program Slicing

Let us now consider the following scenario: through testing, we have identified a variable taking an incorrect value. We want to inspect only the part of the program that might influence this variable: such a part of the program is called a "slice". In this exercise, we will design an algorithm to statically compute the slice of an SSA program.

In order to compute the slice with respect to a given variable v, we need the dependencies of v. We consider both direct (i.e. non-transitive) dependencies, and transitive dependencies. We assume we can take the transitive closure of direct dependencies (by denoting it with a star *).

2.1 Data Dependencies

Let us consider a first notion of dependencies.

Definition 1 (Direct data dependencies) *A variable v depends directly on a variable u in a program P if P contains an instruction that reads u and defines v, e.g. $v := u + 1$.*

```
int x = 42;
int y = 3;
int z = 2;
while (z <= 100) {
  if (y > 10) {
    x = x + 1;
    y = y / 2;
  } {
    x = x - 1;
    z = z * y;
  };
}
return x
```

Figure 1: Program 1

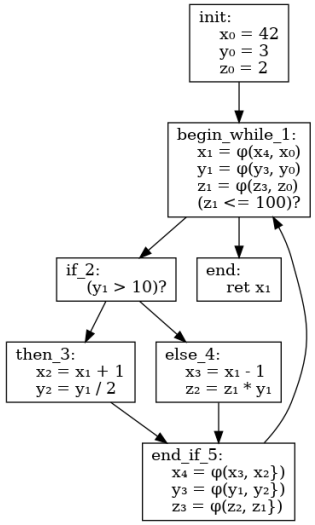


Figure 2: Program 1 in SSA form

Question #1

Give the direct data dependencies of all variables in the SSA representation of Program 1 (given on Figure 2).

Solution:

x_0	
x_1	x_0, x_4
x_2	x_1
x_3	x_1
x_4	x_2, x_3
y_0	
y_1	y_0, y_3
y_2	y_1
y_3	y_1, y_2
z_0	
z_1	z_0, z_3
z_2	z_1, y_1
z_3	z_1, z_2

Question #2

What are all the variables that influence the value of y_1 in Program 1, according to the direct data dependencies computed in the previous question?

Solution: We take the transitive closure, which yields for y_1: y_0, y_1, y_2, y_3.

The transitive closure is the following table (here we need only to compute the y).

x_0	
x_1, x_2, x_3, x_4	x_0, x_1, x_2, x_3, x_4
y_0	
y_1, y_2, y_3	y_0, y_1, y_2, y_3
z_0	
z_1, z_2, z_3	z_0, z_1, z_2, z_3, y_1

Question #3

Give an algorithm $DD(P)$ computing direct data dependencies in an SSA program P as a dictionary.

Solution: Walk through the CFG under SSA form, at each declaration $v := e$, add an entry $v \mid var(e)$ in the dictionary.

Question #4

Give an algorithm $DD^*(P, v)$ computing transitive data dependencies of a variable v in an SSA program P. You can use the transitive closure operation.

Solution: Take the transitive closure of $DD(P)$, and access on v.

Question #5

Remove all the instructions of Program 1 on which y_1 has no transitive data dependencies. Does this slice captures every instructions that might influence the value of y_1? What is missing?

Solution: We obtain a slice without any z nor x. The loop will spin: control is missing.

2.2 Control Dependencies

We now consider a new kind of dependencies, to take into account what was missing with the previous notion.

Definition 2 (Direct control dependencies) *A variable v depends directly on a variable u in a program P if u is used as the predicate of a branch of P that determines the value that v is assigned, e.g. if $u > 0$ then $v = 0$ else $u = 0$.*

Question #6

Give the direct control dependencies of all variables in the SSA representation of Program 1 (given on Figure 2).

Solution:

x_0	
x_1	
x_2	z_1, y_1
x_3	z_1, y_1
x_4	z_1
y_0	
y_1	
y_2	z_1, y_1
y_3	z_1
z_0	
z_1	
z_2	z_1, y_1
z_3	z_1

Question #7

Compute the slice of Program 1 with respect to y_1 using both (transitive) data and control dependencies.

Solution: All x_i are removed, the control structure remains.

Question #8

We now assume to have an algorithm $CD(P)$ computing the direct control dependencies in an SSA program P as a dictionary. Write an algorithm $slice(P, v)$ which slices the program P with respect to the variable v using both data and control dependencies.

Solution:

1. Compute the transitive closure of $DD(P) \cup CD(P)$ (possible by hypothesis) and evaluate it on v, obtaining this way the transitive dependencies of v: we call this $D(P, v)$.
2. For any assignment to some variable x (including ϕ nodes), keep the assignment if and only if $x \in D(P, v)$.
3. If a test has no variables in $D(P, v)$, remove this test and keep only the negative edge of the jump (this is to prevent having two edges to a same block without any branching, we choose arbitrarily).
4. Remove all empty blocks for cleaning; and if we obtain a test with both edges to the same block, remove this test and one of these edges.

NB: $CD(P)$ is computed using post-dominators.

3 Security levels

3.1 Introduction

Security conscientious developers must always take care to distinguish two types of data: public data, which can be printed and sent on the network, and private data, which should never be printed, stored, or revealed in any way. As the last few years of security vulnerability have shown, ensuring that no private data can be revealed is cumbersome and error prone to do by hand.

In this exercise, we will consider how to help developers identify if their program can leak information, through a static analysis. The general approach will be to "tag" values with either `HIGH` security or `LOW` security, and prevent mixing of security levels. The purpose of the analysis performed is to prevent `HIGH` level security information to leak toward `LOW` level context. In real life a typical leak would happen through printing but we only consider assignments to public variables here. Values with `HIGH` security must not be disclosed and thus any operation that uses a `HIGH` security value must produce a `HIGH` security value.

We consider the following syntax for a **WHILE** language with security (without functions). We denote $\mathcal{S} = \{HIGH, LOW\}$ and let s range in $\mathcal{S}$ ($s \in \{HIGH, LOW\}$).

We first define expressions e (only constants are changed wrt. **WHILE**):

$$\begin{array}{rcll} e & ::= & c^s & \textit{constant} \\ & | & x & \textit{variable} \\ & | & e + e & \textit{addition} \\ & | & e \times e & \textit{multiplication} \\ & | & e < e & \textit{boolean expression} \\ & | & \ldots & \end{array}$$

Then we redefine statements S (unchanged wrt. **WHILE**):

$$\begin{array}{rcll} S & ::= & x := e & \text{assign} \\ & | & \mathit{skip} & \text{do nothing} \\ & | & S_1; S_2 & \text{sequence} \\ & | & \texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2 & \text{test} \\ & | & \texttt{while } e \texttt{ do } S \texttt{ done} & \text{loop} \end{array}$$

And finally programs P with variable declarations are defined as follows (only types are changed wrt. **WHILE**):

$$\begin{array}{rcll} P & ::= & D; S & \text{program} \\ D & ::= & \mathit{var}\ x : \tau_s \mid D; D & \text{variable declaration} \end{array}$$

τ_s are types with security information, their syntax is as follows:

$$\tau_s ::= \mathit{int}^s \mid \mathit{bool}^s$$

3.2 Small-step semantics and static analysis for private and public data

We define the static semantic for a language enforcing the absence of information leak, based on the **WHILE** language. Each value is a couple of a normal value and a security tag. Variable declarations are also tagged.

Additionally to the store σ that maps variables to values and has no security information, we have a mapping from variable names to security levels: $\rho : Var \to \mathcal{S}$. The small-step semantics now has the form: $(S, \sigma, \rho) \Rightarrow (S', \sigma', \rho')$ or $(S, \sigma, \rho) \Rightarrow (\sigma', \rho')$ for the last stage. We insist that σ is unchanged and has no security information.

Expression evaluation should be a function of the form $Val(e, \sigma, \rho) = v^s$ where v is an integer or boolean value; note that Val now also returns a security level s.

Solution: Do not forget to print the companion sheet for the students!

Question #1

Inspired by the semantics for expression evaluation (in the companion sheet) write the semantics for expression evaluation with security. You can use a function max (resp. min) that takes the maximum (resp. minimum) of two security tags (we consider that `HIGH` > `LOW`). It should encode the fact that `HIGH` security values must not be disclosed and thus any operation that uses a `HIGH` security value must produce a `HIGH` security value.

Note: recall that $\sigma(x)$ is a value with no security information.

Solution: We define $Val(e, \sigma, \rho)$ by induction on the expression e.

$Val(c^s, \sigma, \rho) = c^s$

$Val(x, \sigma, \rho) = \sigma(x)^{\rho(x)}$

For what follows, assume $Val(e, \sigma, \rho) = v^s$ and $Val(e', \sigma, \rho) = v'^{s'}$.

$Val(e + e', \sigma, \rho) = (v + v')^{max(s,s')}$ (we use max so that if one of the value has `HIGH` security level, then the result too; this is provided v and v' are int)

$Val(e \times e', \sigma, \rho) = (v \times v')^{max(s,s')}$

$Val(e < e', \sigma, \rho) = (v < v')^{max(s,s')}$

The semantics of the **WHILE** language is unchanged except:

1. The security levels are remembered at runtime in the ρ mapping.
2. Assignment checks the compatibility between the assigned value and the variable storing the value.

Question #2

Write the inference rules that build ρ from the set of variable declarations D. It should build judgments of the form $\vdash D \mapsto \rho$.

Solution:

$$\vdash var\ x : \tau_s \mapsto [x \mapsto s] \qquad \frac{\vdash D \mapsto \rho \qquad \vdash D' \mapsto \rho' \qquad dom(\rho) \cap dom(\rho') = \emptyset}{\vdash D; D' \mapsto \rho \cup \rho'}$$

NB: We need $dom(\rho) \cap dom(\rho') = \emptyset$ do not declare several time the same variable with different security levels.

Question #3

Explain the following rule (small-step semantics for assignment) in 2-3 lines.

$$\frac{Val(e,\sigma,\rho) = v^s \qquad \rho(x) \geq s}{(x := e, \sigma, \rho) \Rightarrow (\sigma[x \mapsto v], \rho)}$$

Solution: Basic explanations of assign: the value of the variable x is updated in σ with the value v of e.

Check for security level: a `HIGH` security value v cannot be stored in a `LOW` security value variable x, because this would yield a wrong information flow from `HIGH` to `LOW`, thus we cannot have $\rho(x) < s$; but anything else can go, whence the requirement of $\rho(x) \geq s$.

Question #4

Write the rules for evaluating the sequence (small-step).

Solution:

$$\frac{(S,\sigma,\rho) \Rightarrow (S'',\sigma',\rho')}{(S;S',\sigma,\rho) \Rightarrow (S'';S',\sigma',\rho')} \qquad \frac{(S,\sigma,\rho) \Rightarrow (\sigma',\rho')}{(S;S',\sigma,\rho) \Rightarrow (S',\sigma',\rho')}$$

Question #5

Evaluate the following program according to your rules (we use `^H` and `^L` for the security tags in the code snippets). What is the configuration reached at the end? Explain.

```
var x:int^L;
x:= 5^L+3^L;
x:= 5^L−2^H
```

Solution: The final configuration is $(x := 5^L - 2^H, [x \mapsto 8], [x \mapsto L])$. We are stuck because of the non-applicable assign: $Val(5^L - 2^H, [x \mapsto 8], [x \mapsto L]) = 3^H$, which cannot be assigned to x of security level $L < H$.

Question #6

Modify the semantics so that any invalid assignment due to a security reason reaches a new configuration called `SecurityError`. What is changed in the evaluation of the program above with your new semantics?

Solution: You should add one more rule that states:

$$\frac{Val(e,\sigma,\rho) = v^s \qquad \rho(x) < s}{(x := e, \sigma, \rho) \Rightarrow \texttt{SecurityError}}$$

This rule is applicable on what was previously the last state of the evaluation: $(x := 5^L - 2^H, [x \mapsto 8], [x \mapsto L]) \Rightarrow$ `SecurityError`.

Question #7

Inspired by the type system of **WHILE**, write a "security" type system for our new language. You do not have to check standard typing. You should provide a security type judgment for expressions, statements and programs. Security judgments should use $\Vdash$ and have the form $\rho \Vdash e : s$ (e has security level s under ρ), $\rho \Vdash S$ (S is well-typed for security under ρ) and $\Vdash P$ (P is well-typed for security).

Solution: For expressions:

$$\rho \Vdash c^s : s \qquad \rho \Vdash x : \rho(x) \qquad \frac{\rho \Vdash e : s \qquad \Gamma, \rho \Vdash e : s'}{\rho \Vdash e + e' : max(s, s')} \qquad \frac{\rho \Vdash e : s \qquad \Gamma, \rho \Vdash e : s'}{\rho \Vdash e \times e' : max(s, s')}$$

$$\frac{\rho \Vdash e : s \qquad \Gamma, \rho \Vdash e : s'}{\rho \Vdash e < e' : max(s, s')}$$

For statements:

$$\frac{\rho \Vdash e : s \qquad \rho(x) \geq s}{\rho \Vdash x := e} \qquad \rho \Vdash skip \qquad \frac{\rho \Vdash S_1 \qquad \rho \Vdash S_2}{\rho \Vdash S_1; S_2} \qquad \frac{\rho \Vdash S_1 \qquad \rho \Vdash S_2}{\rho \Vdash \texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2}$$

$$\frac{\rho \Vdash S}{\rho \Vdash \texttt{while } e \texttt{ do } S \texttt{ done}}$$

For programs:

$$\frac{\vdash D \Mapsto \rho \qquad \rho \Vdash S}{\Vdash D; S}$$

NB: It is not necessary to add $\rho \Vdash e : s$ as hypothesis in `if` e `then` S_1 `else` S_2 and `while` e `do` S `done`: it is always possible to compute a security level for any expression (by taking the max at each step)!

Question #8

Show that the following property does not hold, you should explain why and provide a counter example:

$$\rho \Vdash e : s \implies \exists v,\ Val(e, \sigma, \rho) = v^s$$

Solution: The judgment $\rho \Vdash e : s$ does not check "standard" typing. Therefore, with $e = 3^H + True^H$, we get $\rho \Vdash e : H$, whereas $Val(e, \sigma, \rho) = (3 + True)^L$, and we refuse such nonsensical values.

Question #9

Using the type-system we have seen in the course, i.e. supposing $\Gamma \vdash e : \tau$ for the right Γ, state a property that ensures that $Val(e, \sigma, \rho)$ reduces to a value of the right security level.

Solution:

$$\rho \Vdash e : s \wedge \Gamma \vdash e : \tau \wedge (\forall x\ \tau,\ \emptyset \vdash \sigma(x) : \tau \iff \Gamma(x) = \tau) \implies Val(e, \sigma, \rho) = v^s$$

With words: we furthermore need e to be typable, of type τ, under a typing Γ such that Γ corresponds to the types of the value stored for all variables.

Question #10

Prove the following property above for a reduced syntax for expression, which contains only constants c^s, addition $e + e$, and comparison $e < e$:

$$\rho \Vdash e : s \wedge Val(e, \sigma, \rho) = v^{s'} \implies s = s'$$

Solution: By structural induction on e. No difficulty as the rules are similar with respect to security (and our rule system is deterministic).

For $e = c^t$. We have $\rho \Vdash c^t : t$ and $Val(c^t, \sigma, \rho) = c^t$, thence $s = t = s'$.

For $e = e_0 + e_1$, assuming the property holds for e_0 and e_1. As $\rho \Vdash e : s$, we concluded by $\dfrac{\rho \Vdash e_0 : s_0 \qquad \Gamma, \rho \Vdash e_1 : s_1}{\rho \Vdash e_0 + e_1 : max(s_0, s_1)}$, so $s = max(s_0, s_1)$. Similarly, for $Val(c^t, \sigma, \rho) = c^t$, we concluded through $Val(e_0 + e_1, \sigma, \rho) = (v_0 + v_1)^{max(t_0, t_1)}$ with $Val(e_0, \sigma, \rho) = v_0^{t_0}$ and $Val(e_1, \sigma, \rho) = {v_1}^{t_1}$, thence $s' = max(t_0, t_1)$. By induction hypothesis, $s_0 = t_0$ and $s_1 = t_1$. Whenceforth $s = max(s_0, s_1) = max(t_0, t_1) = s'$.

The case for comparison $<$ (and for multiplication $\times$ as well) is identical to the one for addition $+$.

Question #11

Express a simple preservation property for your "security" type system and informally explain why it is true (it is simpler to prove it here than for the classical type system).

Solution:

$$\rho \Vdash S \wedge (S, \sigma, \rho) \Rightarrow (S', \sigma', \rho') \implies \rho' \Vdash S'$$

In fact $\rho = \rho'$ (because no $\Rightarrow$ will change it, it depends only on the initial variable declarations) and the cases are resolved by a very simple induction (S' is inside S in general).

Question #12

Express formally the property of the form "well-typed programs do not go wrong" guaranteed by your "security" type system. Explain informally what this guarantees.

Solution:

$$\forall \sigma,\ \rho \Vdash S \implies \neg\left((S, \sigma, \rho) \Rightarrow \texttt{SecurityError}\right)$$

OR

$$\Vdash D; S \implies \neg\left((S, \emptyset, \rho_0) \Rightarrow^* \texttt{SecurityError}\right)$$

with $\vdash D \mapsto \rho_0$

Question #13

Prove the property you stated in the question above in the case of a single assignment statement.

Solution: Consider the following property:

$$\forall \sigma,\ \rho \Vdash S \implies \neg\left((S, \sigma, \rho) \Rightarrow \texttt{SecurityError}\right)$$

Assuming $\rho \Vdash x := e$, by security typing of assignment:

$$\frac{\rho \Vdash e : s \qquad \rho(x) \geq s}{\rho \Vdash x := e}$$

we get $\rho \Vdash e : s \wedge \rho(x) \geq s$.

Only one assignment reduction rule can lead to the error:

$$\frac{Val(e, \sigma, \rho) = v^{s'} \qquad \rho(x) < s'}{(x := e, \sigma, \rho) \Rightarrow \texttt{SecurityError}}$$

so assume $Val(e, \sigma, \rho) = v^{s'} \wedge \rho(x) < s'$.

By the property proven in question 10, $\rho \Vdash e : s \wedge Val(e, \sigma, \rho) = v^{s'} \implies s = s'$, we deduce $s = s'$. This is contradictory as $\rho(x) \geq s$ and $\rho(x) < s' = s$.

Question #14

A type system always has to be conservative and reject programs that would not lead to an error. Give an example of a program that does not reach `SecurityError` but is rejected by your "security" type system.

Solution: Just do an invalid assignment in a conditional branch that cannot be taken.

$$D; S := \left(var\ x : int^L; \texttt{if}\ true^L\ \texttt{then}\ skip\ \texttt{else}\ x := 0^H\right)$$

Then we have $\neg\left((S, \emptyset, \rho_0) \Rightarrow^* \texttt{SecurityError}\right)$ but not $\Vdash D; S$: the reciprocate for the property question 12 is false.

Information could leak due to a conditional test: consider the program in Figure 3.

Question #15

What is wrong with this program? Explain how confidential information can flow without raising an error.

Solution: The value of b, with `LOW` security level, and henceforth a public data, depends on the `HIGH` security value of $confidential$. The problem is that branching on a `High` security value can affect values of `LOW` security.

```
var confidential:int^H;
var b:bool^L;
confidential:=5^H;
if (confidential > 3^L)
  then b:=true^L
  else b:=false^L
```

Figure 3: A program branching on secrets

Question #16 (Difficult)

Propose an extension of the semantics for **WHILE** with security that tracks such information flow (based on the security level of the if condition) and raise an error in the case above. Explain your changes and only provide formal definitions for rules that are significantly changed.

You should make sure that you take into account nested ifs. You should make sure that the while condition is correctly tracked too.

Hint: one possible solution is to extend the syntax for statements and tag some of the statements with security information.

Solution: Principle: Syntax now has $:=^H$ statements in addition to the existing ones. S^H tags all assignments inside S with H, you obtain $x :=^H e$. S^L does nothing (i.e. $S^L = S$).

$$\frac{Val(e,\sigma,\rho) = tt^s}{(\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2, \sigma, \rho) \Rightarrow (S_1^s, \sigma, \rho)} \qquad \frac{Val(e,\sigma,\rho) = ff^s}{(\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2, \sigma, \rho) \Rightarrow (S_2^s, \sigma, \rho)}$$

$$\frac{Val(e,\sigma,\rho) = v^s \qquad \rho(x) = L}{(x :=^H e, \sigma, \rho) \Rightarrow \texttt{SecurityError}}$$

Other rules are unchanged or similar or trivially adapted. While is evaluated by translation to if, it thus requires no additional treatment.

Question #17 (Difficult)

Propose an extension of your "security" type system that allows you to guarantee that no `SecurityError` configuration is reachable.

Solution: Add a security status s to the judgement: $\rho, s \Vdash S$.

$$\frac{\rho \Vdash e : s \qquad \rho, max(s,s') \Vdash S_1 \qquad \rho, max(s,s') \Vdash S_2}{\rho, s' \Vdash \texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2} \qquad \frac{\rho \Vdash e : s \qquad \rho(x) \geq s \qquad \rho(x) \geq s'}{\rho, s' \Vdash x := e}$$

$$\frac{\vdash D \Mapsto \rho \qquad \rho, L \Vdash S}{\Vdash D; S}$$

While and other statements are trivially adapted from the previous cases.