Assignment 4: Flows

Due: 10/30/25 at 11:59pm

In this assignment, you will solve a few problems about the theoretical properties of dataflow analysis. You will also implement a taint analysis based on the information flow techniques discussed in class.

4a: Dataflow Theory (25 points)

Submission link: https://www.gradescope.com/courses/1107485/assignments/6981233

Problem 1: Lattices (10 points)

A partial order $\sqsubseteq$ on a set $\mathcal{L}$ is a binary relation that is reflexive, transitive and anti-symmetric. For some $x, y \in \mathcal{L}$, $x \sqcup y$ is the "join" or least-upper-bound of x and y wrt the partial order. More precisely, if $x \sqcup y = z$ then:

1. x and y are less than z, i.e., $x \sqsubseteq z \wedge y \sqsubseteq z$
2. z is smaller than everything larger than both x and y, i.e., $x \sqsubseteq w \wedge y \sqsubseteq w \implies z \sqsubseteq w$.

$\mathcal{L}$ is a join-semilattice if $x \sqcup y$ is defined for all $x, y \in \mathcal{L}$.

In our analyses, we have often taken lattices like the zero/nonzero lattice and put them into a dictionary mapping variables to lattice elements. This dictionary is, itself, a lattice. More formally, the *map lattice* $\mathcal{M}$ is defined as a function $K \to \mathcal{L}$ over a set of keys K and a value lattice $\mathcal{L}$.

Define a partial order on $\mathcal{M}$ and a join function that models how we have implemented the map lattice for dataflow analysis. Prove that if $\mathcal{L}$ is a join-semilattice, then $\mathcal{M}$ is a join-semilattice.

Problem 2: Termination (5 points)

Recall our constant analysis where $\sigma \in \mathcal{M}$ with $K = \mathsf{Var}$ and $\mathcal{L} = \{\bot, \top\} \cup \mathsf{Const}$. Consider the transfer function for constant analysis over binary operators:

$$f(x = y \oplus z, \sigma) = \sigma \left[x \mapsto \left\{ \begin{array}{ll} \bot & \text{if } \sigma[y] = \bot \vee \sigma[z] = \bot \\ \top & \text{if } \sigma[y] = \top \vee \sigma[z] = \top \\ \sigma[y] \oplus \sigma[z] & \text{otherwise} \end{array} \right\} \right]$$

Prove that this case of the transfer function is monotonic, i.e., $\sigma_1^{\mathsf{in}} \sqsubseteq \sigma_2^{\mathsf{in}} \implies \sigma_1^{\mathsf{out}} \sqsubseteq \sigma_2^{\mathsf{out}}$.

Problem 3: Soundness (10 points)

A dataflow analysis computes an abstract state $\sigma \in \mathsf{AbsState}$ for each instruction in a bytecode program, where $\mathsf{AbsState}$ is a lattice. An analysis is *sound* with respect to an abstraction function $\alpha : \mathsf{Configuration} \to \mathsf{AbsState}$ if for all programs P, for all traces T arising from P, and for all configurations $C_i \in T$, then $\alpha(C_i) \sqsubseteq \sigma^{\mathsf{in}}_{C_i.\mathsf{pc}}$. In other words, the abstract state for the current instruction is a sound approximation of the abstraction of the current state. The syntax and semantics for the bytecode and configurations are given in Figure 1 and Figure 2.

A dataflow analysis is implemented with a transfer function $f : \mathsf{Instr} \times \mathsf{AbsState} \to \mathsf{AbsState}$. To prove that an analysis is sound wrt α, we will need to prove that the transfer function correctly updates the abstract state. More formally, an analysis is *locally sound* wrt α under the following conditions. Given a configuration C and an abstract state σ^{in}, assume σ^{in} approximates C, i.e., $\alpha(C) \sqsubseteq \sigma^{\mathsf{in}}$. Next, assume $C \hookrightarrow C'$. Finally, we must show that σ^{out} approximates C', i.e., $\alpha(C') \sqsubseteq \sigma^{\mathsf{out}}$ where $\sigma^{\mathsf{out}} = f(C.\mathsf{prog}[C.\mathsf{pc}], \sigma^{\mathsf{in}})$.

Constant analysis should be sound wrt the abstraction function:

$$\alpha(C) = \left\{ \left\{ \begin{array}{ll} C.\mathsf{locals}[x] & \text{if } x \in \mathsf{dom}(C.\mathsf{locals}) \\ \bot & \text{otherwise} \end{array} \right\} \mid x \in \mathsf{Var} \right\}$$

Prove that the binary operator case of the transfer function is locally sound wrt α.

4b: Taint Analysis (75 points)

Submission link: https://www.gradescope.com/courses/1107485/assignments/6981243

Motivation

The goal of taint analysis is to identify information flows from secure sources to insecure sinks. For this assignment, a secure source is data returned from a function marked `#[secure]` and an insecure sink is the function `println`. For example, here is a safe use of secure information:

```
#[secure] fn secure() -> int { 0 }

fn main() {
  let p = secure() in
  if p == 0 {
    // do secure things...
  };
  println!("Done!")
}
```

The compiler should allow the program above to execute. However, the compiler should reject programs with explicit flows, like this:

```
fn main() {
  let p = secure() in
  println(int_to_string(p))
}
```

And it should reject programs with implicit flows, like this:

```
fn main() {
  let p = secure() in
  if p == 0 {
    println("Done!")
  }
}
```

Specification

To start, you should call into your taint analysis from the function `bc::analyze`. Your taint analysis should be an interprocedural dataflow analysis which is flow-sensitive, field-sensitive, and context-sensitive. Flow-sensitivity means this program should pass:

```
fn main() {
  let p = secure() in
  p := 0;
  println(int_to_string(p))
}
```

Field-sensitivity means this program should pass:

```
fn main() {
  let p = (0, secure()) in
  println(int_to_string(p.0))
}
```

Context-sensitivity means this program should pass:

```
fn id(x: int) -> int { x }

fn main() {
  let p1 = id(secure()) in
  let p2 = id(0) in
  println(int_to_string(p2))
}
```

Handling pointers

Your analysis should reuse and build upon the pointer analysis you implemented for Assignment 3. You do **not** need to make your pointer analysis interprocedural[1]. When encountering a function call, your pointer analysis should assume the worst: all reachable allocations from all arguments of compatible types must be assumed to alias. For example:

```
fn main() {
  let a = (0,) in
  let b = (1,) in
  let c = ("",) in
  let d = (a,) in
  let e = (c,) in
  mystery(a, b, c, d, e)
}
```

After calling `mystery`, your pointer analysis should assume that `d.0` aliases the allocations for `a` or `b`. Your pointer analysis should **not** assume that `a` aliases `b`, because `a` is not assignable to `b` in `mystery`. Your pointer analysis should **not** assume that `e.0` aliases `a` or `b`, because those allocations have a different type.

Your pointer analysis should distinguish between *concrete* allocations (those occurring in the body of a function) and *abstract* allocations (those provided as input to a function). For example, consider this program:

```
fn set_val(x: (int,), y: int) {
  x.0 := y
}

fn main() {
  let t = (0,) in
  set_val(t, secure());
  println(int_to_string(t.0))
}
```

Your analysis should analyze `set_val` in the context that `y` is tainted. The transfer function for `x.0 := y` should therefore add `x.0` to the tainted set. However, the model of memory locations (the output of the `aliases` function) used in Assignment 3 did not let us describe the location `x.0` because `x` points to an allocation coming from the caller. You should use the technique discussed in the second information flow lecture to extend your model of allocations to include abstract allocations.

[1] Interprocedural pointer analysis has been the subject of much research. See Hind et al. "Interprocedural Pointer Alias Analysis" for an early approach to this problem.

Handling function calls

Your analysis should be interprocedural. To handle function calls, you first need to determine the function being called. For this, you should reuse your constant analysis from Assignment 3[2]. If a function cannot be resolved, then your analysis should assume the worst: all reachable places from the fuction's arguments (and its return) are tainted. If a function is resolved to a standard library function, then your analysis should just assume that the output is tainted if any reachable input is tainted.

Your analysis should be context-sensitive, where a context is the set of tainted parameters. You should use the algorithm described in the first information flow analysis lecture to recursively invoke the intraprocedural algorithm on a called function starting with a given context. For simplicity, you can treat recursive function calls as a call to an unknown function.

Handling indirect flows

To handle indirect flows, you will need to first implement the algorithm for computing control dependences described in the second information flow analysis lecture[3]. You should use the dominator functions provided by `petgraph` to compute the reverse CFG and post-dominator tree. See: `dominators::simple_fast` and `Dominators`.

Implementation Tips

To help you design your analysis, I'll give you a high-level picture of how the reference solution is implemented. The entry point to the analysis has three stages: (1) compute intraprocedural facts, (2) compute interprocedural taint, (3) check for erroneous use of tainted values.

1. **Compute intraprocedural facts:** Intraprocedural facts include pointer analysis, constant analysis, and control dependencies. For each function, I store all these data structures in a single `Facts` struct.
2. **Compute interprocedural taint:** I have two structures, `GlobalAnalysis` and `LocalAnalysis`. The global structure holds the taint analysis results for all functions analyzed thus far, along with all the intraprocedural facts computed in step 1. When asked to analyze a single function, `GlobalAnalysis` build a `LocalAnalysis` struct which implements the `Analysis` trait and holds a pointer to the `GlobalAnalysis` struct.
   - The `LocalAnalysis` transfer function handles three main cases: function calls, allocations, and a generic implementation for all remaining instructions.
   - The `GlobalAnalysis` struct holds a field `results` of type:

```
RefCell<
  HashMap<Symbol,
          HashMap<Vec<Place>,
                  IndexSet<MemLoc>>>>
```

[2] A more sophisticated analysis could do an interprocedural constant analysis to analyze calls to higher-order functions. This approach is called *control-flow analysis*, since it functional languages it's necessary to recover the global control flow structure of a program. See Olin Shriver's 1991 dissertation "Control-Flow Analysis of Higher-Order Languages."

[3] See Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph" (Section 4) for a full description of the algorithm.

The `RefCell` allows this structure to be mutated even behind an immutable reference. The `HashMap<Symbol, ...>` defines a map from a function's name to its data. The `HashMap<Vec<Place>, IndexSet<MemLoc>>` maps from function contexts to tainted inputs/outputs.

3. **Check for erroneous use of tainted values:** This step should walk the program and look for calls to `println`. If a function has multiple calling contexts, you should consider whether the call to `println` is tainted in any possible context.

The reference implementation is about 500 SLOC.

Grading

Your implementation will be tested against a suite of programs which are expected to pass or fail, and you get points for each successful test. All tests have been provided to you as a ZIP file on the course website.

[3] We use `Vec<Place>` instead of `HashSet<Place>` because in Rust, hash sets are not themselves hashable. Note that it's therefore important that your contexts are sorted, so `[x, y]` is not treated differently than `[y, x]`.

$$
\begin{aligned}
\text{Program } P &::= \mathcal{I}^* \\
\text{Instr } \mathcal{I} &::= x = rv \mid \text{goto } i \mid \text{if } x \text{ goto } i_1 \text{ else } i_2 \\
\text{Rvalue } rv &::= c \mid x \mid x_1 \oplus x_2 \\
\text{Binop } \oplus &::= + \mid / \mid =
\end{aligned}
$$

Figure 1: Bytecode syntax

$$\text{Configuration } C \in \{\mathsf{prog} : \text{Program}, \mathsf{pc} : \mathbb{N}, \mathsf{locals} : \text{Var} \rightharpoonup \text{Const}\}$$

$\boxed{C \vdash rv \Downarrow c}$

$$\frac{}{C \vdash c \Downarrow c}\ \textsc{Rv-Const} \qquad \frac{x \in \mathsf{dom}(C.\mathsf{locals})}{C \vdash x \Downarrow C.\mathsf{locals}(x)}\ \textsc{Rv-Var}$$

$$\frac{C \vdash x_1 \Downarrow c_1 \qquad C \vdash x_2 \Downarrow c_2 \qquad c_3 = c_1 \oplus c_2}{C \vdash x_1 \oplus x_2 \Downarrow c_3}\ \textsc{Rv-Binop}$$

$\boxed{C \hookrightarrow C'}$

$$\frac{C.\mathsf{prog}[C.\mathsf{pc}] = \text{“}x = rv\text{”} \qquad C \vdash rv \Downarrow c}{C \hookrightarrow \{C \ \underline{\text{with}}\ \mathsf{pc} = C.\mathsf{pc} + 1, \mathsf{locals} = C.\mathsf{locals}[x \mapsto c]\}}\ \text{E-}\textsc{Stmt}$$

$$\frac{C.\mathsf{prog}[C.\mathsf{pc}] = \text{“goto } i\text{”}}{C \hookrightarrow \{C \ \underline{\text{with}}\ \mathsf{pc} = i\}}\ \text{E-}\textsc{Goto}$$

$$\frac{C.\mathsf{prog}[C.\mathsf{pc}] = \text{“if } x \text{ goto } i_1 \text{ else } i_2\text{”} \qquad C \vdash x \Downarrow \mathsf{true}}{C \hookrightarrow \{C \ \underline{\text{with}}\ \mathsf{pc} = i_1\}}\ \text{E-}\textsc{If-True}$$

$$\frac{C.\mathsf{prog}[C.\mathsf{pc}] = \text{“if } x \text{ goto } i_1 \text{ else } i_2\text{”} \qquad C \vdash x \Downarrow \mathsf{false}}{C \hookrightarrow \{C \ \underline{\text{with}}\ \mathsf{pc} = i_2\}}\ \text{E-}\textsc{If-False}$$

Figure 2: Bytecode semantics