Ada Taint Checker

Peter C. Chapin
Vermont Technical College

January 20, 2009

Outline

- Background
- Theory of Operation
- Implementation
- Limitations of Current Implementation
- Future Work
- Demonstration

Secure Information Flow

Confidentiality. *If my program reads private inputs, can a malicious user deduce some of that private information by observing the program's public outputs?*

Data Integrity. *If my program reads public inputs, can a malicious user modify, in a controlled way, the private information that the program writes by manipulating those inputs?*

For many programs both of these questions are issues.

Ada Taint Checker (ATC) focuses on the Data Integrity problem by tracking "tainted" input data to see if it is ever used in the program's outputs.

Noninterference

Noninterference means

- The public outputs are not affected by changes to the private inputs.

- The private outputs are not affected by changes to the public inputs.

Real programs can't usually follow this principle (why read an input if it doesn't affect the output in some way?).

Instead confidentiality is maintained by passing private inputs through "obfuscation" functions. Data integrity is maintained by passing public inputs through "sanitizing" functions.

General Formulation

- Let $\mathcal{C} = \{S_1, S_2, \ldots, S_n\}$ be a set of security classes.

- Let $\leq$ be a partial order on $\mathcal{C}$ such that $S_i \leq S_j$ means that the data in security class S_i is more sensitive than the data in security class S_j.

- Let $(\mathcal{C}, \leq)$ be a lattice where $\bot$ is the most sensitive security class and $\top$ is the most public class.

When two security classes combine the resulting class should be $S_i \wedge S_j$ if confidentiality is a concern or $S_i \vee S_j$ if data integrity is a concern.

ATC uses only two security classes: tainted and untainted with the obvious meaning.

Theory of Operation

- ATC uses a static analysis.
  - No run-time overhead.
  - Imprecise. ATC is conservative: intended to flag any program with a security error, but will also flag some correct programs.
- ATC uses dataflow analysis to analyze individual subprograms.
- ATC uses a simplistic type system to handle interprocedural issues.

Subprogram Classifications

ATC assigns a classification (type) to each subprogram.

- *Input*. An input subprogram is one which returns tainted data regardless of the status of its arguments.

- *Sanitizing*. A sanitizing subprogram is one which returns untainted data regardless of the status of its arguments.

- *Passive*. A passive subprogram is one which returns information with the same tainting status as its arguments.

Subprogram Output Flag

In addition, ATC marks certain subprograms as "Output" subprograms.

- *Output*. An output subprogram is one for which it is an error to send tainted data.

This flag is independent of the subprogram's classification. It is used after analysis is complete to generate warnings, but otherwise plays no role in the analysis.

Subprogram Information Database

ATC maintains an internal database of subprogram information.

User can initialize this database using a configuration file.

```
Ada.Integer_Text_IO.Get:  I # Input
Ada.Integer_Text_IO.Put: O  # Default with Output flag
Numerics.Sin:             P # Passive
Validate_File_Name:       S # Sanitizing
```

ATC should allow wildcards here, but currently does not.

Default Classification

What classification should ATC use when none is provided? One might think *Input* since it is the most conservative. However, for many functions, this is clearly the wrong choice.

```
package Standard is
  function "+"(L : Integer; R : Integer) return Integer;
end Standard;
...
A := B + C;  -- Invokes Standard."+"
```

Side effect free, or "pure," functions are clearly *Passive*.

Currently ATC uses *Passive* by default.

Dataflow Analysis

ATC builds a control flow graph (CFG) for the subprogram being analyzed.

- $I[n]$. Tainted variables coming into node n.
- $O[n]$. Tainted variables coming out of node n.
- $D[n]$. Variables that are made tainted inside node n. These are the variables that are written by *Input* subprograms or by *Passive* subprograms with tainted inputs.
- $C[n]$. Variables that are sanitized inside node n. These are the variables that are written by *Sanitizing* subprograms or by *Passive* subprograms with untainted inputs.

Dataflow Equations

The dataflow equations are

$$\begin{aligned} I[n] &= \bigcup_{p \in \mathit{pred}[n]} O[p] \\ O[n] &= D[n] \cup (I[n] - C[n]) \end{aligned}$$

The value of I for the initial node is given by the subprogram's input arguments (assumed tainted).

The subprogram is considered secure if the following are both true

1. None of the subprogram's outputs are in O for the final CFG node (sanitizing).

2. For each node in the CFG, no input to a *Output* subprogram is in the I set for that node.

Implementation

- ATC is itself written in Ada.

- ATC uses ASIS: the Ada Semantic Information Standard. ASIS is a library of packages that give an analysis program "easy" access to the syntactic and semantic information computed by the compiler.

ASIS presents the abstract syntax tree of all compilation units in the program.

ASIS allows corresponding declarations of entities used in the program to be looked up. For example, given a procedure call statement one can get the declaration of the procedure and examine the modes (IN, OUT, INOUT) on the parameters.

Computing CFG

ATC's CFG generator walks the syntax tree of the (one) sub-program being analyzed. Constructs the CFG as it goes.

```
procedure Process_While_Paths(E : Element) is
  My_StatementList : Statement_List := Loop_Statements(E);
  Predicate    : Element := While_Condition(E);
  Old_Current : Vertex_Index_Type;
begin
  Add_Simple_Vertex(Predicate);
  Old_Current := Current_Vertex;
  for I in My_StatementList'Range loop
    Process_Construct(My_StatementList(I));
  end loop;
  Create_Edge(CFG, Current_Vertex, Old_Current);
  Current_Vertex := Old_Current;
end Process_While_Paths;
```

Computing Dataflow

Multiple passes are made of the two dataflow equations until `Changed` remains false.

```
-- forall i . (Out[i] = D[i] U (In[i] - C[i]))
for I in Out_Vars'Range loop
  Empty(Temp);
  Empty(Dirty_Set);
  Empty(Clean_Set);
  DC(Get_Vertex(CFG, I), Dirty_Set, Clean_Set, In_Vars(I));
  Temp := In_Vars(I) - Clean_Set;
  Temp := Temp + Dirty_Set;
  if Temp /= Out_Vars(I) then
    Out_Vars(I) := Temp;
    Changed := True;
  end if;
end loop;
```

Computing Statement Effects

The dirty and clean sets are computed for each statement kind.

```
procedure Handle_Assignment is
  Expr   : Expression := Assignment_Expression(Vertex_Data);
  Target : Expression := Assignment_Variable_Name(Vertex_Data);
  Target_Name : Program_Text := Name_Image(Target);
begin
  if Is_Tainted(Expr, Current_Tainted) then
    Add_Element(D_Result, Target_Name);
  else
    Add_Element(C_Result, Target_Name);
  end if;
end Handle_Assignment;
```

Computing Expression Effects

Each expression kind is explored as appropriate. Note the recursive processing of subexpressions.

```
case Expression_Kind(E) is
  -- Literals are not tainted.
  when An_Integer_Literal  |
       A_String_Literal    |
       An_Enumeration_Literal =>
    return False;

  -- Unwind one layer of parentheses using recursion.
  when A_Parenthesized_Expression =>
    return Is_Tainted(Expression_Parenthesized(E), Tainted);

  -- Most expressions are function calls (eg built-in operators).
  when A_Function_Call =>
    return Handle_Function_Call;
end case;
```

Computing Function Call Effects

```
case Get_Subprogram_Class(Function_Name_Image) is
  when Input => return True;
  when Sanitizing => return False;
  when Passive =>
    declare
      Function_Arguments : Association_List :=
        Function_Call_Parameters(E);
    begin
      for I in Function_Arguments'Range loop
        if Is_Tainted(Function_Arguments(I), Tainted) then
          return True;
        end if;
      end loop;
      return False;
    end;
end case;
```

Example

This example shows the sort of code ATC can currently handle.

```
while X < Y loop
  Get(A);  -- A is tainted.
  A := F(A) + (2 * G(A));
  Put(A);  -- Possible error.
  if A = 0 then
    Get(A);
  end if;
  Put(A);  -- Error: Might output a tainted value.
end loop;
```

If `F` and `G` are both sanitizing, ATC finds `A` untainted after the assignment.

If `G` is only passive, then ATC (correctly) finds `A` tainted after the assignment.

Limitations of Current Implementation

The current version of ATC suffers from a number of undesirable limitations.

- Skips Declarative Region
- Analyzes Just One Subprogram
- Incomplete Support for Statements and Expressions
- Mishandles Named Parameter Associations and Default Arguments
- No Support for Control Dependencies

Skips Declarative Region

Currently ATC ignores executable expressions in a subprogram's declarative region.

```
procedure Do_Something is
  A : Integer := Read_A;
  B : Integer := A + F(A);
begin
  ...
end Do_Something;
```

ATC assumes all local variables are initially untainted. Obviously this is a significant problem with the current implementation.

Analyzes Just One Subprogram

Currently ATC only analyzes one subprogram. ATC should deduce the taint classification of each subprogram encountered and add that information to its database for use in future analysis.

```
procedure Outer is
  function Inner(A : Integer) return Integer is ...
    -- Deduce the classification for Inner.
begin
  X := Inner(Y); -- Use deduced classification here.
end Outer;
```

Analyzing entire packages could be done similarly.

No Whole Program Analysis

ATC could first explore all packages in the main procedure's context clause (recursively), using the approach mentioned previously, and then analyze the main procedure.

```
with Ada.Text_IO;
with Numerics.Extended_Integer;
with Crypto.Hashes;
  -- Analyze all subprograms in these packages.

procedure Main is
begin
  -- Analyze Main using information collected previously.
end Main;
```

Incomplete Support for Statements

Currently ATC only understands a subset of Ada control flow constructs. Here is an example of an unsupported construct

```
loop
   ...
   exit when X < Y;
   ...
end loop;
```

The CFG generator doesn't know how to handle the `exit` statement and will raise an exception if it sees one.

Incomplete Support for Expressions

Currently ATC only understands a subset of Ada expression constructs. Examples of expression kinds not supported include

- Record field selection
- Array and record aggregates
- Attribute expressions
- Short circuit expressions
- Type conversions

This list is not exhaustive. Function `Is_Tainted` will raise an exception if it encounters a form it doesn't recognize.

Mishandles Named Parameter Associations

Currently ATC gets confused when seeing named parameter associations in an "unnatural" order. It also ignores default arguments.

```
procedure P(A : Integer; B : Integer; C : Integer := 0);
...
P(B => 2, A => 1);
```

ASIS-for-GNAT does not implement "normalized" parameter association lists, which would make dealing with this trivial.

ATC could untangle the associations manually, but currently does not do so.

No Support for Control Dependencies

Currently ATC ignores taintedness added due to control dependencies.

```
if X < Y then
  A := 1;
else
  A := 2;
end if;
```

If the expression `X < Y` has a tainted value, the user can control which assignment takes place. Thus `A` becomes tainted.

All variables written under the control of a tainted predicate should be marked tainted regardless of the expressions that are used to write those variables.

Future Work

Ada is a rich language. Full taint checking would be involved.

- Dispatching Calls, Subprogram Access
- Exceptions
- Tainted Types
- Access Types
- Generic Units
- Tasking

Dispatching Calls

```
procedure Do_Something_With(Object : in T'Class) is
begin
  Operate_On(Object);
end Do_Something_With;
```

If `Operate_On` is dispatching, the precise procedure `Operate_On` calls depends on `Object`'s runtime type and is not known to the compiler (or ASIS).

ATC could require all corresponding dispatching subprograms have the same taint classification.

Subprogram Access

```
type Error_Handler_Type is
   access procedure(Message : String);
Current_Handler : Error_Handler_Type := Logger'Access;
...
Current_Handler("Bad thing happened");
```

As with dispatching calls, the precise subprogram invoked can only be known at runtime.

ATC could require the programmer to annotate type declarations with taint classification information and then enforce consistency of usage (in effect, extend the Ada type system).

Exceptions

```
begin
  Get(A);           -- A is tainted.
  Do_Something;
  A := Clean(A);    -- A is not tainted.
  Do_Something_Else;
exception
  when Constraint_Error =>
    Put(A);         -- Is A tainted?
end;
```

Exceptions complicate control flow.

ATC could assume that if `A` is tainted anywhere in the block, it is tainted in the exception handlers.

Tainted Types

```
A : Integer := Read_A;
subtype Restricted_Type is Integer range 0..A;
X : Restricted_Type := 10;
```

Subtypes can have dynamic ranges and thus can use potentially tainted values in their definitions.

Ada inserts run-time checks to ensure that values are in bounds. However, a malicious user could force an inappropriate range and thus force (or prevent) exceptions under certain cases.

ATC could define a notion of a "tainted type" and then take any values of that type as unconditionally tainted.

Access Types

```
type Integer_Ptr is access Integer;
P1 : Integer_Ptr := new Integer'(1);
P2 : Integer_Ptr := P1;
...
P1.all := Read_Value;  -- Now P2.all is tainted.
```

Access types are Ada's pointers. They bring issues of aliasing and corresponding alias analysis.

Ada 95 allows pointers to point at explicitly declared variables. However such usage requires special syntax (both on the declaration of the pointer and of the target object).

Generic Units

```
generic
  with procedure Error_Handler(Message : String);
  type Restricted_Type is range <>;
package Whatever is
  ...
end Whatever;
```

Unlike dispatching calls, generics are handled at compile time.

ATC could attempt to analyze each instantiation separately. Alternatively ATC could require the programmer to annotate the generic formal parameters with taint classification information and then enforce that usage when the generic unit is instantiated.

Tasking

```
entry Meeting(Data : in Integer; Result : out Integer);
...
accept Meeting(Data : in Integer; Result : out Integer) do
  Local_Storage := Data;
  Result := Temporary_Result;
end Meeting;
```

Concurrency always adds complications.

It may be possible to assign taint classifications to `entry`/`accept` pairs.

However, Ada 95's tasking model is reasonably rich. Many issues to consider.

Demonstration!

Questions

?