Laint: Static Taint Tracking using Linear Types

Machiry Aravind Kumar

UCSB

1 Taint Tracking

Any variable that can be modified/provided by an outside user poses a potential security risk, if used unverified in an program. Example: strcpy of an user provided data without checking for length. Taint Tracking tracks these variables inside a program and warns if used in places where untainted or safe data is expected.
This can be done statically or dynamically (by instrumenting the program).
Any taint tracking system needs following information:

- Taint Introduction.

  How taint is introduced into the program. Ex: function that reads input from user.

- Taint Policy

  Functions or Points in program where corresponding variables should be checked for taint and appropriate action must be taken.

- Taint Propagation

  Rules on how taint propagates in the program.

More details about static taint tracking can be found at [1], [2]

2 Goal

Viewing static taint tracking as a type checker for linear type system and implementing the same.

3 Types

3.1 Linear Types or (Semi-Linear Types)

- Taint (or taint)

  This type signifies something which is unverified and could be potentially malicious.

3.2 Non Linear Types

- Un Taint (or untaint)

  This type signifies something which is verified and safe.

3.3 Typing Rules

Information required for static taint tracking are represented as typing rules.

Informally following are the Taint tracking rules.

- Taint Introduction: Explicit declaration or assignment can introduce or change the variable type to taint.
- Taint Policy: An expression used in if must be of untainted type
  This is to avoid any side channel attacks.
- Taint Policy: untainted variables can become tainted but not the other way round (i.e Non Linear types can be converted into linear types but not the other way round).
- Taint Propagation: Any expression involving taint type will be of taint type, expect for Sanitizers. **Sanitizers** are functions, which have atleast one parameter of type taint and return untaint value.
- Taint Policy : Sanitizers consume taint arguments, once used as an argument to sanitizer a tainted variable cannot be further used in the program.

These can be represented formally as typing rules as shown below. Note that there are syntactic rules of the language that are omitted i.e a function cannot be defined multiple times, function signature has to match declaration and definition, variable need to be declared before used.

3.3.1 Default Types

These are the default types for variables and constants. Note that unknown is not actually a type, this is to indicate that corresponding variable is not declared or not given a specific type.

$$\frac{}{\Gamma \vdash v : unknown}\ \text{UntypedVariable}$$

$$\frac{}{\Gamma \vdash int : untaint}\ \text{IntegerConstant}$$

$$\frac{}{\Gamma \vdash string : untaint}\ \text{StringConstant}$$

3.3.2 Variable Declaration (Taint Introduction)

These rules specify how variables can be declared. There is a partial order for the available types as shown below:

$$unknown \sqsubseteq untaint \quad unknown \sqsubseteq taint \quad untaint \sqsubseteq taint$$

Variable can be declared (if initial type is unknown) or redeclared to promote to a higher type in the partial order. This specifies one of the important property of linear types, non-linear types can be converted into linear but not vice versa.

$$\frac{\Gamma, v : t \vdash eval(t1\ v; stmts) \quad t \sqsubseteq t1}{\Gamma, v : t1 \vdash eval(stmts)}\ \text{VariableDeclaration}$$

3.3.3 Binary Expression (Taint Propagation)

Binary expression follow below rules to get their types, as you can see expression is untainted only if all values used are untainted.

$$\frac{\Gamma \vdash x : taint}{\Gamma \vdash x \bigotimes y : taint} \text{ TaintedBinaryExpression}$$

$$\frac{\Gamma \vdash x : untaint, y : untaint}{\Gamma \vdash x \bigotimes y : untaint} \text{ UnTaintedBinaryExpression}$$

3.3.4 Assignment Expression (Taint Propagation)

As specified from rules below, assignment can be done according to the partial order and all variables involved are declared. Note that this allows assigning tainted values to untainted variable, resulting in tainted the variable.

$$\frac{\Gamma, x : t1, y : t2 \vdash eval(y = x; stmts) \quad t1 \neq unknown \quad t2 \neq unknown \quad t2 \sqsubseteq t1}{\Gamma, x : t1, y : t1 \vdash eval(stmts)} \text{ Assignment}$$

3.3.5 Untainted Control (Taint Policy)

This is one of our taint policy i.e not to allow any tainted value to control program flow. This is captured by below rule which doesn't allow tainted conditional expression in if.

$$\frac{\Gamma \vdash c : untaint \quad \Gamma \vdash eval(if(c)\{condstmts;\}stmts)}{\Gamma \vdash eval(condstmts; stmts)} \text{ UntaintedCondition}$$

3.3.6 Function Definition evaluation (Taint Propagation)

To evaluative function definition, we follow the below rule, as you can see function body is evaluated by adding types of the arguments and function type itself. whereas rest of the statements are evaluated by adding function type.

$$\frac{\Gamma \vdash eval(tret\ func(t\ arg1, t2\ arg2, ...)\{funcstmts\}\ stmts)}{\Gamma, arg1 : t, arg2 : t2, func : (t, t2, ...) \rightarrow tret \vdash eval(funcstmts) \quad \Gamma, func : (t, t2, ...) \rightarrow tret \vdash eval(stmts)} \text{ FunctionEvaluvation}$$

3.3.7 Function Declaration evaluation (Taint Propagation)

Similar to function definition, declaration is evaluated by just adding function type to the set of assumptions.

$$\frac{\Gamma \vdash eval(rtype\ foo(t1\ v1, t2\ v2, ...); stmts)}{\Gamma, foo : (t1, t2, ...) \rightarrow rtype \vdash eval(stmts)} \text{ FunctionTypeIntroduction}$$

3.3.8 Evaluating Sanitizer (Taint Policy)

This is an important rule and captures the linearity. Here, a taint variable used as an argument to a sanitizer function will be consumed by it and is no longer available for evaluation.

$$\frac{\Gamma, v : taint, sanitizer : (..., taint, ...) \rightarrow untaint \vdash eval(sanitizer(..., v, ...); stmts)}{\Gamma, sanitizer : (..., taint, ...) \rightarrow untaint \vdash eval(stmts)} \text{ TaintConsumption}$$

3.3.9 Function call evaluation with untainted types (Taint Propagation)

As untainted type is non linear type it can be used unlimitedly. The below rules captures this:

$$\frac{\Gamma, v : untaint, foo : (..., t, ...) \rightarrow * \vdash eval(foo(..., v, ...); stmts)}{\Gamma, v : untaint, foo : (..., t, ...) \rightarrow * \vdash eval(stmts)} \text{ UnTaintPropogation}$$

3.3.10 Non Sanitizer evaluation with tainted types (Taint Propagation)

A non-sanitizer function cannot consume a taint variable as shown in below rule:

$$\frac{\Gamma, v : taint, foo : (..., t, ...) \rightarrow taint \vdash eval(foo(..., v, ...); stmts)}{\Gamma, v : taint, foo : (..., t, ...) \rightarrow taint \vdash eval(stmts)} \text{ NonSanitizerFunction}$$

References

[1] *Wikipedia.* http://en.wikipedia.org/wiki/Taint_checking

[2] *Static Taint Analysis for C.* https://code.google.com/p/tanalysis/