

An Implementation of Static Trust Flow Analysis for the Simply Typed Lambda Calculus CMPS 221: Advanced Computer Security

Damian Eads
1156 High Street
University of California, Santa Cruz
Santa Cruz, CA

June 6, 2006

Abstract

We describe a Scheme implementation of a variant of the trust system first formalized by Jens Palsberg and Peter Ørbæk. The language is a basic extension of System F_1 . The typing rules, constraint formation rules, and contraction rules are provided when they differ from the original paper.

1 Introduction

With the advent of broadband, most computers today are always connected to the web giving the attacker an unlimited window of time to mount an attack. Information flows into the computer from the internet and travels to many places. Some of this information comes from attackers wanting to do harm. If this information (e.g. “delete hard drive”) is displayed in a web browser window, the harm is minimal. If the data flows to some critical functions then the damage could be catastrophic. This paper explores mechanisms for preventing such exploits. We present a variant of a type system first proposed by Jens Palsberg to statically check whether a program will ever use untrustworthy data in critical places[3].

In a typical scenario, packets arrive from the internet and propagate through a complicated network of software components. These packets often undergo several transformations, and may be combined with other pieces of data, which can be trusted or untrusted. It is expensive to manually analyze the flow of all of this data. It is also impractical to expect a human to do this task, especially when it is common for all software on a computer to span millions of lines of code. Given a piece of data on a computer, where did it come from? Should it be involved in making critical system decisions? These questions fall in the domain of information trust analysis. A formal approach provides a foundation upon which theorems can be proven and guarantees can be made.

Examining and checking the flow of trusted and untrusted data is called trust flow analysis. A complete and thorough manual examination is often infeasible so automated approaches are very helpful. Performing checks on the trustworthiness of data elements at run-time is known as dynamic trust analysis. One major problem with dynamic approaches is that they incur considerable run-time overhead in terms of computation and memory. In contrast, static flow analysis incur the computational overhead beforehand, thereby eliminating run-time overhead. They also can prevent a program from compiling or running if any of the static checks fail.

Jens Palsberg and Peter Ørbæk provide a framework for performing flow analysis in the lambda calculus using static type inference[3, 4]. Trustworthiness is expressed as an annotation to a type. Programmers indicate “critical parts” of a program—parts that should only be influenced by trusted data—using check commands. A trusted routine can also untaint data (i.e. deem it to be trustworthy) by invoking a trust command. Similarly, distrust are used to tag data as untrustworthy to prevent their flow to critical functions.

$$\begin{array}{l} \mathbf{T}, \mathbf{A}, \mathbf{B} ::= \mathbf{K} \\ \quad \quad \quad | (\mathbf{A} \rightarrow \mathbf{B}) \end{array}$$

Figure 1: **Type grammar:** defines well-formed type expressions.

$$\begin{array}{l} E, F ::= \lambda x : \mathbf{T}. F \\ \quad \quad | \text{check } E \\ \quad \quad | \text{trust } E \\ \quad \quad | \text{distrust } E \\ \quad \quad | \text{unit} \\ \quad \quad | EF \end{array}$$

Figure 2: **Program grammar:** defines well-formed programs.

The formalisation in [4] supports subtyping, and it extends to arbitrary trust lattices. The system ensures proper type inference and trust flow behavior by imposing constraints. These constraints are formulated as a set of constraint expressions. This set is built inductively using the type judgement rules. An attempt is then made to solve this system of constraints to unify the types and trust annotations.

2 Project

This project demonstrates the use of static trust analysis. A static type checker and constraint solver were written using a scaled down version of the original T-system language described in [4]. We used a simple extension to System F_1 , also known as the simply typed lambda calculus used [1]. Types in the language are the simple base type \mathbf{K} or an arrow type; the type grammar is shown in Figure 1. There is no loss of generality here—other base types could be incorporated into this system.

The same constructs found in Palsberg’s paper were used with two exceptions. First, lambda expressions are expressed as Church encodings. Second, the `unit` construct creates a literal value of type \mathbf{K} . The grammar defining acceptable programs is shown in Figure 2.

The contraction rules are the same as the original paper except for λ and the `unit` construct,

$$\frac{E \longrightarrow \text{unit}}{\text{trust } E \longrightarrow \text{unit}} \quad \frac{E \longrightarrow \text{unit}}{\text{check } E \longrightarrow \text{unit}}. \quad (1)$$

Since subtyping is not permitted in this language, constraints governing the inference of types and subsumption are not considered. We adopt the use of $\llbracket E \rrbracket$ to denote the trust value of an expression. We employ several variants of this notation as described below.

x the trust value of a variable x

x_{in} the trust value of the domain of the procvl variable x .

x_{out} : the trust value of the range of the procvl variable x .

$\llbracket E \rrbracket$: the trust value of the expression E .

$\llbracket E \rrbracket_{\text{in}}, \llbracket E \rrbracket_{\text{out}}$: the trust value of the domain and range, respectively, of the procvl expression E . The value is meaningless if E does not evaluate to a procvl.

$\langle E \rangle, \langle E \rangle_{\text{in}}, \langle E \rangle_{\text{out}}$ denotes the trust value of the input x to a lambda expression $E = \lambda x : \mathbf{T}. G$ at the time of application EF . They are equated to $x, x_{\text{in}},$ and x_{out} when the constraints for E are formed.

These variables are simply used for implementation convenience, i.e. the constraint builder need not examine the contents of E to ascertain the appropriate input variable to use in the constraint expression.

3 Type Rules

Before a program is processed, we assume that it is alpha-renamed so that each bound variable has a unique name. We employ the constraint-typing judgement notation used in Pierce[5],

$$\frac{}{\Gamma \vdash t : \mathbf{T} \mid_{\mathcal{X}} C}. \quad (2)$$

The notation means that “a term t has type \mathbf{T} under the assumptions Γ whenever constraints C are satisfied” [5]. Note that \mathcal{X} denotes the set of all variables used in constraint expressions.

The type of a variable is its type in the current context,

$$\text{TVar} \frac{\Gamma \vdash x : \mathbf{T} \in \Gamma}{\Gamma \vdash x : \mathbf{T} \mid_{\{ \llbracket x \rrbracket, \llbracket x \rrbracket_{\text{in}}, \llbracket x \rrbracket_{\text{out}}, x_{\text{in}}, x_{\text{out}}, x \} } \{ x = \llbracket x \rrbracket, x_{\text{in}} = \llbracket x \rrbracket_{\text{in}}, x_{\text{out}} = \llbracket x \rrbracket_{\text{out}} \}}. \quad (3)$$

The x variable represents the trust value of the bound variable, and x_{in} and x_{out} are the trust annotation of the domain and range, respectively, of x . The $\llbracket x \rrbracket$ and $\llbracket x \rrbracket_{\text{in}, \text{out}}$ variables are used to denote the trust annotations for x in the context of x being an expression.

The `check` construct carries over the type of the inner expression E and equates its the trustworthiness to `tr`. It also carries over the $\llbracket E \rrbracket_{\text{in}, \text{out}}$, regardless if the values are meaningful,

$$\text{TCheck} \frac{\frac{}{\Gamma \vdash E : \mathbf{T} \mid_{\mathcal{X}} C} \quad C' = \{ \llbracket \text{check } E \rrbracket = \llbracket E \rrbracket, \llbracket E \rrbracket = \text{tr}, \llbracket \text{check } E \rrbracket_{\text{in}, \text{out}} = \llbracket E \rrbracket_{\text{in}, \text{out}} \} \quad \mathcal{X}' = \{ \llbracket \text{check } E \rrbracket, \llbracket \text{check } E \rrbracket_{\text{in}, \text{out}} \}}{\Gamma \vdash \text{check } E : \mathbf{T} \mid_{\mathcal{X} \cup \mathcal{X}'} C \cup C'}. \quad (4)$$

The `trust` construct also carries over the type judgement of the inner expression. However, its trustworthiness is ignored. The trustworthiness $\llbracket \text{trust } E \rrbracket$ is equated to `tr`,

$$\text{TTrust} \frac{\frac{}{\Gamma \vdash E : \mathbf{T} \mid_{\mathcal{X}} C} \quad C' = \{ \llbracket E \rrbracket = \text{tr}, \llbracket \text{trust } E \rrbracket_{\text{in}, \text{out}} = \llbracket E \rrbracket_{\text{in}, \text{out}} \} \quad \mathcal{X}' = \{ \llbracket \text{trust } E \rrbracket, \llbracket \text{trust } E \rrbracket_{\text{in}, \text{out}} \}}{\Gamma \vdash \text{trust } E : \mathbf{T} \mid_{\mathcal{X} \cup \mathcal{X}'} C \cup C'}. \quad (5)$$

The `distrust` construct is almost identical to `trust` except it generates a constraint to make $\llbracket \text{distrust } E \rrbracket = \text{dis}$,

$$\text{TDistrust} \frac{\frac{}{\Gamma \vdash E : \mathbf{T} \mid_{\mathcal{X}} C} \quad C' = \{ \llbracket E \rrbracket = \text{dis}, \llbracket \text{distrust } E \rrbracket_{\text{in}, \text{out}} = \llbracket E \rrbracket_{\text{in}, \text{out}} \} \quad \mathcal{X}' = \{ \llbracket \text{distrust } E \rrbracket, \llbracket \text{distrust } E \rrbracket_{\text{in}, \text{out}} \}}{\Gamma \vdash \text{distrust } E : \mathbf{T} \mid_{\mathcal{X} \cup \mathcal{X}'} C \cup C'}. \quad (6)$$

The `unit` construct represents a literal constant so we omit the trust value; this gives,

$$\text{TUnit} \frac{}{\Gamma \vdash \text{unit} : \mathbf{T} \mid_{\{ \llbracket \text{unit} \rrbracket \}} \emptyset}. \quad (7)$$

The x_{in} , x_{out} , and x trust variables are assigned as explained earlier. The trustworthiness of the procval is `tr`. The range of the procval $\llbracket \lambda x : \mathbf{T}. F \rrbracket_{\text{out}}$ is equal to the trust value $\llbracket F \rrbracket$ of the body F ,

$$\text{TLambda} \frac{\frac{}{x : \mathbf{A}, \Gamma \vdash F : \mathbf{B} \mid_{\mathcal{X}} C} \quad C' = C \cup \{ \llbracket \lambda x : \mathbf{T}. F \rrbracket = \text{tr}, \llbracket \lambda x : \mathbf{T}. F \rrbracket_{\text{in}} = x, \llbracket \lambda x : \mathbf{T}. F \rrbracket_{\text{out}} = \llbracket F \rrbracket, \langle F \rangle_{\text{in}, \text{out}} = x_{\text{in}, \text{out}}, \llbracket F \rrbracket = x \} \quad \mathcal{X}' = \{ \llbracket \lambda x : \mathbf{T}. F \rrbracket, \llbracket \lambda x : \mathbf{T}. F \rrbracket_{\text{in}, \text{out}}, \langle F \rangle_{\text{in}, \text{out}}, x_{\text{in}, \text{out}} \}}{\Gamma \vdash \lambda x : \mathbf{T}. F : \mathbf{A} \rightarrow \mathbf{B} \mid_{\mathcal{X} \cup \mathcal{X}'} C'}. \quad (8)$$

The application rule assigns the trust annotations of the result of F to the input to E . It also stipulates the trust of the EF expression is given by

$$C'' = \left\{ \llbracket E \rrbracket_{\text{in}} = \llbracket F \rrbracket, \llbracket EF \rrbracket = \llbracket E \rrbracket_{\text{out}}, \llbracket EF \rrbracket_{\text{in}, \text{out}} = \llbracket F \rrbracket_{\text{in}, \text{out}}, \langle E \rangle = \llbracket F \rrbracket, \langle E \rangle_{\text{in}, \text{out}} = \llbracket F \rrbracket_{\text{in}, \text{out}} \right\}$$

$$\text{TApp} \frac{\frac{E : \mathbf{A} \rightarrow \mathbf{B} \quad |_{\mathcal{X}} C}{E : \mathbf{A} \rightarrow \mathbf{B} \quad |_{\mathcal{X}} C} \quad \frac{F : \mathbf{B} \quad |_{\mathcal{X}'} C'}{F : \mathbf{B} \quad |_{\mathcal{X}'} C'} \quad \mathcal{X}'' = \left\{ \llbracket EF \rrbracket, \llbracket EF \rrbracket_{\text{in}, \text{out}}, \langle E \rangle, \langle E \rangle_{\text{in}, \text{out}}, \llbracket F \rrbracket_{\text{in}, \text{out}} \right\}}{EF : \mathbf{B} \quad |_{\mathcal{X} \cup \mathcal{X}' \cup \mathcal{X}''} C \cup C' \cup C''} \quad (9)$$

The algorithm proceeds as follows:

1. Accept a program E as input.
2. Perform alpha renaming on the program so that variable names are unique. This results in a new program E' . Alpha renaming permits variable names to be used as prefixes to trust variable names.
3. Tag each term in E with a unique number. This unique number is used as a prefix for trust variables.
4. Type-check E . If successful, continue with the trust checking.
5. Build a set of constraint equations.
6. Determine if the deductive closure of the system of constraint equations results in any contradictions (e.g. $\text{tr} = \text{dis}$ or $\text{dis} \leq \text{tr}$). If so, the program is untrustworthy.

4 Example

Consider the following well-typed, untrustworthy program,

$$\text{check distrust unit.} \quad (10)$$

We assume that it is trustworthy. We build a set of constraint equations by following the typing rules,

$$\llbracket \text{distrust unit} \rrbracket_{\text{in}} = \llbracket \text{unit} \rrbracket_{\text{in}}, \quad (11)$$

$$\llbracket \text{distrust unit} \rrbracket_{\text{out}} = \llbracket \text{unit} \rrbracket_{\text{out}}, \quad (12)$$

$$\llbracket \text{distrust unit} \rrbracket = \text{dis}, \quad (13)$$

$$\llbracket \text{check distrust unit} \rrbracket_{\text{in}} = \llbracket \text{distrust unit} \rrbracket_{\text{in}}, \quad (14)$$

$$\llbracket \text{check distrust unit} \rrbracket_{\text{out}} = \llbracket \text{distrust unit} \rrbracket_{\text{out}}, \quad (15)$$

$$\llbracket \text{check distrust unit} \rrbracket = \llbracket \text{distrust unit} \rrbracket, \text{ and} \quad (16)$$

$$\llbracket \text{check distrust unit} \rrbracket = \text{tr}. \quad (17)$$

The proof is simple: it follows that

$$\llbracket \text{check distrust unit} \rrbracket = \text{tr} = \llbracket \text{distrust unit} \rrbracket = \text{dis}, \quad (18)$$

which is a contradiction. Therefore the program must be untrustworthy.

5 Implementation

The static checker is written in Scheme and runs on the Petite Chez interpreter. The source code can be downloaded from <http://www.soe.ucsc.edu/~eads/tsys>. The code is made up of over 1000 lines and contains an unfinished work-in-progress version for solving the problem with subtyping. Note that the version without subtyping, System F_1 , is complete. Friedman's pattern matching library was exceptionally helpful[2].

To load the checker into Scheme, do:

```
(load "F1-System.ss")
```

Use the `report` function to perform type checking and constraint verification:

```

> (report '(check (app (lambda (x : (K -> K)) (distrust x)) (lambda (x : K) x))))
Original Expression : (check (app (lambda (x : (k -> k))
                                (distrust x)) (lambda (x : k) x)))
Alpha-renamed      : (check (app (lambda (x1 : (k -> k)) (distrust x1))
                                (lambda (x2 : k) x2)))
Tagged             : (1 check (2 app (3 lambda (x1 : t) (4 distrust (5 x1)))
                                (6 lambda (x2 : t) (7 x2))))
Type               : (k -> k)
Constraint set     : ((e-5 = var-x1) (in-5 = varin-x1) (out-5 = varout-x1)
                    (in-4 = in-5) (out-4 = out-5) (e-4 = dis) (varin-3 = varin-x1)
                    (varout-3 = varout-x1) (var-3 = var-x1) (in-3 = var-x1) (e-3 = tr)
                    (out-3 = e-4) (e-7 = var-x2) (in-7 = varin-x2) (out-7 = varout-x2)
                    (varin-6 = varin-x2) (varout-6 = varout-x2) (var-6 = var-x2)
                    (in-6 = var-x2) (e-6 = tr) (out-6 = e-7) (in-3 = e-6) (var-3 = e-6)
                    (varin-3 = in-6) (varout-3 = out-6) (e-2 = out-3) (in-2 = in-6)
                    (out-2 = out-6) (in-1 = in-2) (out-1 = out-2) (e-1 = e-2) (e-2 = tr))
Trustworthy       : No
> (report '(check (lambda (x : K) x)))
Original Expression : (check (lambda (x : k) x))
Alpha-renamed      : (check (lambda (x1 : k) x1))
Tagged             : (1 check (2 lambda (x1 : t) (3 x1)))
Type               : (k -> k)
Constraint set     : ((e-3 = var-x1) (in-3 = varin-x1) (out-3 = varout-x1)
                    (varin-2 = varin-x1) (varout-2 = varout-x1) (var-2 = var-x1)
                    (in-2 = var-x1) (e-2 = tr) (out-2 = e-3) (in-1 = in-2)
                    (out-1 = out-2) (e-1 = e-2) (e-2 = tr))
Trustworthy       : Yes
> (report '(lambda (x : K) x))
Original Expression : (lambda (x : k) x)
Alpha-renamed      : (lambda (x1 : k) x1)
Tagged             : (1 lambda (x1 : t) (2 x1))
Type               : (k -> k)
Constraint set     : ((e-2 = var-x1) (in-2 = varin-x1) (out-2 = varout-x1)
                    (varin-1 = varin-x1) (varout-1 = varout-x1) (var-1 = var-x1)
                    (in-1 = var-x1) (e-1 = tr) (out-1 = e-2))
Trustworthy       : Yes
> (report '(lambda (x : K) (check x)))
Original Expression : (lambda (x : k) (check x))
Alpha-renamed      : (lambda (x1 : k) (check x1))
Tagged             : (1 lambda (x1 : t) (2 check (3 x1)))
Type               : (k -> k)
Constraint set     : ((e-3 = var-x1) (in-3 = varin-x1) (out-3 = varout-x1) (in-2 = in-3)
                    (out-2 = out-3) (e-2 = e-3) (e-3 = tr) (varin-1 = varin-x1)
                    (varout-1 = varout-x1) (var-1 = var-x1) (in-1 = var-x1)
                    (e-1 = tr) (out-1 = e-2))
Trustworthy       : Yes
> (report '(check (lambda (x : K) (distrust (trust x)))))
Original Expression : (check (lambda (x : k) (distrust (trust x))))
Alpha-renamed      : (check (lambda (x1 : k) (distrust (trust x1))))
Tagged             : (1 check (2 lambda (x1 : t) (3 distrust (4 trust (5 x1)))))
Type               : (k -> k)
Constraint set     : ((e-5 = var-x1) (in-5 = varin-x1) (out-5 = varout-x1)
                    (in-4 = in-5) (out-4 = out-5) (e-4 = tr) (in-3 = in-4)
                    (out-3 = out-4) (e-3 = dis) (varin-2 = varin-x1)
                    (varout-2 = varout-x1) (var-2 = var-x1) (in-2 = var-x1)
                    (e-2 = tr) (out-2 = e-3) (in-1 = in-2) (out-1 = out-2))

```

```

      (e-1 = e-2) (e-2 = tr))
Trustworthy      : Yes
> (report '(check (distrust (lambda (x : K) x))))
Original Expression : (check (distrust (lambda (x : k) x)))
Alpha-renamed      : (check (distrust (lambda (x1 : k) x1)))
Tagged             : (1 check (2 distrust (3 lambda (x1 : t) (4 x1))))
Type               : (k -> k)
Constraint set     : ((e-4 = var-x1) (in-4 = varin-x1) (out-4 = varout-x1)
      (varin-3 = varin-x1) (varout-3 = varout-x1) (var-3 = var-x1)
      (in-3 = var-x1) (e-3 = tr) (out-3 = e-4) (in-2 = in-3)
      (out-2 = out-3) (e-2 = dis) (in-1 = in-2) (out-1 = out-2)
      (e-1 = e-2) (e-2 = tr))
Trustworthy      : No

```

6 Conclusion

We have implemented a simpler, scaled-down version of the trust system for the simply typed lambda calculus. The project was inspired by earlier work by Palsberg and Ørbæk, and many of their ideas were borrowed. The typing and constraint rules were formalized, and a simple example was given on how to conclude non-trustworthiness. The next step would be to attempt to prove a soundness theorem for the typing rules I outlined. An implementation of the system presented in this paper is complete. However, the work on an implementation with subtyping using exactly the same rules as presented in Palsberg and Ørbæk is underway.

References

- [1] Luca Cardelli. *Type Systems*, chapter 103. CRC Press, Boca Raton, FL, 1997.
- [2] Dan Friedman. `match.ss`: A pattern matcher for scheme. <http://www.cs.indiana.edu/classes/c311/match.ss>, 2006.
- [3] Jens Palsberg and Peter Ørbæk. Trust in the λ -calculus. In Alan Mycroft, editor, *SAS'95: Static Analysis*, volume 983 of *Lecture Notes in Computer Science*, pages 314–330, Glasgow, September 1995. Springer-Verlag. A full version with proofs appears in [4].
- [4] Jens Palsberg and Peter Ørbæk. Trust in the λ -calculus. Technical Report BRICS-RS-95-31, BRICS, University of Aarhus, June 1995.
- [5] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.