BRICS RS-95-24 P. Ørbæk: Can you Trust your Data?

BRICS

Basic Research in Computer Science

Can you Trust your Data?

Peter Ørbæk

BRICS Report Series **RS-95-24**

ISSN 0909-0878 **April 1995**

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@daimi.aau.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.aau.dk/BRICS/`
`ftp ftp.brics.aau.dk (cd pub/BRICS)`

Can you Trust your Data?

Peter Ørbæk
BRICS, University of Aarhus*
poe@daimi.aau.dk

March 27, 1995

Abstract

A new program analysis is presented, and two compile time methods for this analysis are given. The analysis attempts to answer the question: "Given some trustworthy and some untrustworthy input, can we trust the value of a given variable after execution of some code". The analyses are based on an abstract interpretation framework and a constraint generation framework respectively. The analyses are proved safe with respect to an instrumented semantics. We explicitly deal with a language with pointers and possible aliasing problems. The constraint based analysis is related *directly* to the abstract interpretation and therefore indirectly to the instrumented semantics[1].

1 Introduction

This paper discusses a static program analysis that can be used to check that the validity of data is only promoted to higher levels of trust in a conscious and controlled fashion.

It is important to stress that the purpose of the analyses is *not* to improve run-time performance, but to give warnings to the programmer whenever untrustworthy data are being unduly trusted.

In the rest of the paper we try to motivate the need for a trust analysis. We give an instrumented semantics for a simple first order language with pointers, in effect keeping track of the trustworthiness of data at run-time. Then an abstract interpretation is presented, approximating the analysis statically. Finally, in order to gain separate analysis of separate program modules as well as better time complexity, a constraint based analysis is presented. The constraint based analysis is proved to be a safe approximation of the abstract interpretation.

2 Motivation

Many computer systems handle information of various levels of trustworthiness. Whereas the contents of the company database can usually be trusted, the input gathered via a

***B**asic **R**esearch **i**n **C**omputer **S**cience,Dept. of Comp. Sci., University of Aarhus, Denmark, Centre of the Danish National Research Foundation.

[1]References should point to [8] instead to this report.

modem, or from a part-time secretary may not be trusted as much, and data validation and authentication routines must ensure the validity of data before it is promoted to a higher level of trust and entered into the database.

That there is a need for some method to control the propagation of trust in real-life computer programs is witnessed for example by the security hole recently found in the Unix `sendmail` program [1]. `Sendmail` is the mail forwarding program running on the majority of Unix machines on the Internet. The security hole allowed one to give the program a certain devious input (in an e-mail message) that would result in having arbitrary commands executed on the machine with superuser privileges. Had an analysis like the one described in this paper been run on the `sendmail` sources it is likely that such a breach in security could have been noticed in advance. See below.

As an example of the kind of analysis envisioned, Perl [9] implements "taint" checks at run-time to help ensure that untrustworthy values are not put in places (such as a process' user-id) where only trusted data should go. This "tainting" is very closely related to the instrumented semantics given below.

We aim at finding a *static* program analysis, i.e. an analysis run only once when a program is compiled, such that the programmer is warned if and when data is promoted from untrustworthy to trustworthy in an uncontrolled fashion. Clearly there will be a need to promote data from untrusted to trusted, but with the envisioned analysis we can guarantee that the promotion takes place in an explicit and conscious way.

In [3, 4] Denning and Denning present a flow analysis for what they call "secure information flow". Their analysis in a sense solves the dual of the problem attacked in this paper. Their aim is to prevent privileged information from leaking out of a trusted computer system, whereas "trust analysis" aims at preventing untrustworthy information from entering into a trusted computer system.

2.1 The Sendmail example

Inside the `sendmail` C code there is a routine, `deliver()`, that delivers an e-mail message to an address:

```
void deliver(MSG m, ADR a, ...) {
   ...
   setuid(a.uid);
   ...
}
```

For some addresses, the `uid` field makes no sense and is uninitialized. In current sources, the `ADR` structure contains a bit that should be set just when the `uid` field is valid, and this bit is tested in several places at run-time before the `uid` field is used. The security hole existed because the programmer had forgotten to insert enough of these checks and consequently, under certain circumstances one was able to circumvent the checks and gain superuser privileges.

With a trust analysis, a reasonable choice is to make the `setuid()` system-call accept only trusted values, as it sets the user-id of the current process. This forces `a.uid` to be a trusted value for compilation of `deliver()` to succeed. One would then have just *one* place, namely in a validation procedure, where the value of an address' `uid` field is promoted to trusted.

```
ADR validate_address(ADR a) {
```

```
    ADR a1;
    ... some validation, fill in appropriate parts of a1.
    ... we may now trust the contents of a.uid.
    a1.uid = trust(a.uid);
    return a1;
}
```

The trust analyzer will now be able to ensure the programmer that only trusted values are passed to `setuid()`. And all the run-time checks on the validity bit are no longer needed as the trust checks are wholly static.

3 The `While` language

Since a large part of security conscious programs today are written in C, a stripped down imperative C-like language with pointers is explored. The abstract syntax for the language is defined by the following BNF:

$$
\begin{array}{rcl}
I & ::= & \textit{variable names} \\
P & ::= & \mathsf{deref}\ P \mid I \\
E & ::= & P \mid E + E \mid \ldots \mid \textit{const} \mid \mathsf{addr}\ I \mid \mathsf{trust}\ E \mid \mathsf{distrust}\ E \\
S & ::= & \mathsf{while}\ E\ \mathsf{do}\ S \mid S; S \mid P\ := \ E
\end{array}
$$

Informally, I denotes identifiers, P denotes pointer expressions, E denotes arithmetic and boolean expressions and S denotes statements. Initially the language included first order procedures, but due to lack of space and since they can be added on in a straightforward way they have been left out. How to do this is briefly discussed in Section 7.

We assume programs are *strongly typed* (i.e. like in Pascal), but leave out type declarations such as int or bool as the only thing that matters for our purpose is whether a variable contains a pointer or a scalar (non-pointer) value.

Deref dereferences pointers. If-statements can be emulated by while loops. This saves a syntactic construct.

Notation: The following conventions are used for meta-syntactic variables: i ranges over identifiers I; e, e_1 and e_2 range over expressions E; p range over pointer expressions P and s, s_1 and s_2 range over statements S.

4 Instrumented Semantics

In order to keep track the trustworthiness of values at run-time, we give an instrumented semantics that associate each value with a flag telling whether the value can be trusted or not. This is to be taken as the *definition* of the desired analysis.

Below are the definitions of the semantic domains. *Addr* is the set of possible addresses in memory. The set of possible program values, *Val*, includes at least integers, booleans and addresses. Environments (*Env*) map identifiers to addresses. Note that environments are assumed to be *injective*.

By strong typing we can assume that trust _ is applied to scalar values only. This will be important for the constraint generation analysis. *Notation:* The memory $M[v/a]$ is as M except that the address a is mapped to the value v, and similarly for environments.

$$
\begin{array}{rcl}
Tr & = & \{\bot, \top\} \\
Val_I & = & Val \times Tr \\
Mem_I & = & Addr \rightarrow Val_I \\
\mathcal{E}_I & : & E \rightarrow Env \rightarrow Mem_I \rightarrow Val_I \\
\mathcal{S}_I & : & S \rightarrow Env \rightarrow Mem_I \rightarrow Tr \rightarrow Mem_I \\
addr_I & : & P \rightarrow Env \rightarrow Mem_I \rightarrow Addr \\
M_I & \in & Mem_I
\end{array}
$$

We equip the set *Tr* with a total ordering ($\leq$) such that $\bot \leq \top$ in order to make it a lattice. The least upper bound operation on this lattice will be denoted by $\vee$, which will also be used to denote the *lub* of environments by point-wise extension. The idea is that $\bot$ corresponds to trusted data, and $\top$ corresponds to untrusted data. *Notation:* $\langle \cdot, \cdot \rangle$ forms Cartesian products and π_n is the n'th projection. t ranges over *Tr* and v over *Val*.

$$
\begin{array}{rcl}
\mathcal{E}_I\ i\ A\ M_I & = & M_I(A(i)) \\
\mathcal{E}_I\ [\![\mathsf{addr}\ i]\!]\ A\ M_I & = & \langle A(i),\ \bot \rangle \\
\mathcal{E}_I\ [\![\mathsf{deref}\ p]\!]\ A\ M_I & = & \mathsf{let}\ \langle v, t \rangle = \mathcal{E}_I\ p\ A\ M_I\ \mathsf{in} \\
 & & \quad \langle \pi_1(M_I(v)),\ t \vee \pi_2(M_I(v)) \rangle \\
\mathcal{E}_I\ [\![e_1 + e_2]\!]\ A\ M_I & = & (\mathcal{E}_I\ e_1\ A\ M_I) \hat{+} (\mathcal{E}_I\ e_2\ A\ M_I) \\
\mathcal{E}_I\ [\![\mathsf{trust}\ e]\!]\ A\ M_I & = & \langle \mathcal{E}_I\ e\ A\ M_I,\ \bot \rangle \\
\mathcal{E}_I\ [\![\mathsf{distrust}\ e]\!]\ A\ M_I & = & \langle \mathcal{E}_I\ e\ A\ M_I,\ \top \rangle \\
\mathcal{E}_I\ const\ A\ M_I & = & \langle const,\ \bot \rangle \\
\langle v_1, t_1 \rangle \hat{+} \langle v_2, t_2 \rangle & = & \langle v_1 + v_2, t_1 \vee t_2 \rangle
\end{array}
$$

The last parameter to $\mathcal{S}_I$ is used in connection with while loops, the reason being that if the condition in the loop cannot be trusted, then all variables assigned in the loop can no longer be trusted as they may depend on the number of iterations taken.

$$
\begin{array}{rcl}
addr_I\ i\ A\ M_I & = & A(i) \\
addr_I\ [\![\mathsf{deref}\ p]\!]\ A\ M_I & = & \pi_1(M_I(addr_I\ p\ A\ M_I))
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S}_I\ [\![\mathsf{while}\ e\ \mathsf{do}\ s]\!]\ A\ M_I\ t & = & \mathsf{let}\ \langle v, t' \rangle = \mathcal{E}_I\ e\ A\ M_I\ \mathsf{in} \\
 & & \quad \mathsf{if}\ v\ \mathsf{then} \\
 & & \quad\quad \mathcal{S}_I\ [\![\mathsf{while}\ e\ \mathsf{do}\ s]\!]\ A\ (\mathcal{S}_I\ s\ A\ M_I\ (t \vee t'))\ t \\
 & & \quad \mathsf{else}\ M_I \\
\mathcal{S}_I\ [\![p\ :=\ e]\!]\ A\ M_I\ t & = & \mathsf{let}\ \langle v, t' \rangle = \mathcal{E}_I\ e\ A\ M_I\ \mathsf{in} \\
 & & \quad M_I[\langle v,\ t \vee t' \rangle / (addr_I\ p\ A\ M_I)] \\
\mathcal{S}_I\ [\![s_1; s_2]\!]\ A\ M_I\ t & = & \mathcal{S}_I\ s_2\ A\ (\mathcal{S}_I\ s_1\ A\ M_I\ t)\ t
\end{array}
$$

5 Abstract Interpretation

The instrumented semantics has the drawback that it propagates the trust of variables only at run-time. Below is presented an abstract interpretation [2] of the language computing an approximation to the trust tags and not the actual values.

Since the actual values are not known during the abstract interpretation neither are the addresses, hence environments and memories are collapsed into abstract environments mapping identifiers directly to "trust signatures". *Notation:* 2^I denotes the set of subsets of I.

$$\begin{array}{rcl} Val_A & = & Tr \cup 2^I \\ Env_A & = & I \to Val_A \\ \mathcal{E}_A & : & E \to Env_A \to Val_A \\ \mathcal{S}_A & : & S \to Env_A \to Tr \to Env_A \\ addr_A & : & P \to Env_A \to 2^I \\ asg & : & Val_A \to Env_A \to Val_A \to Env_A \\ M_A & \in & Mem_I \\ v & \in & Val_A \\ A_A & \in & Env_A \end{array}$$

We extend the total ordering on *Tr* to a partial ordering on Val_A such that

$$\forall v \in Val_A : \perp \leq v \leq \top \text{ and } a,\, b \in 2^I \Rightarrow (a \leq b \iff a \subseteq b).$$

This makes Val_A a complete lattice, and for any finite collection of programs, finite as well. $\vee$ is used for least upper bound on this lattice too.

The idea is that $\perp$ corresponds to trusted scalars. A set of identifiers corresponds to a trusted pointer that may point to any of the variables mentioned in the set. $\top$ corresponds to untrusted values of any kind. Letting abstract environments map identifiers to sets of identifiers, instead of keeping both information about the pointer and the data pointed to in the abstract environment, is done to handle pointer aliasing. *Notation:* For brevity, define $A_A(\top) = \top$, and for $a \subseteq I$ let $A_A(a) = \bigcup\{A_A(i) \mid i \in a\}$.

$$\begin{array}{rcl} \mathcal{E}_A\ i\ A_A & = & A_A(i) \\ \mathcal{E}_A\ [\![\mathsf{addr}\ i]\!]\ A_A & = & \{i\} \\ \mathcal{E}_A\ [\![\mathsf{deref}\ p]\!]\ A_A & = & \bigvee A_A(\mathcal{E}_A\ p\ A_A) \\ \mathcal{E}_A\ [\![e_1 + e_2]\!]\ A_A & = & (\mathcal{E}_A\ e_1\ A_A) \vee (\mathcal{E}_A\ e_2\ A_A) \\ \mathcal{E}_A\ [\![\mathsf{trust}\ e]\!]\ A_A & = & \perp \\ \mathcal{E}_A\ [\![\mathsf{distrust}\ e]\!]\ A_A & = & \top \\ \mathcal{E}_A\ const\ A_A & = & \perp \end{array}$$

The auxiliary *asg* function monotonically assigns a new trust value to a set of identifiers in an abstract environment. $addr_A\ p\ A_A$ yields the set of variables that might be assigned to when p is the left hand side of an assignment. *Notation:* $dom(M)$ denotes the domain of the map M.

$$\begin{array}{rcl} asg\ t\ A_A\ \top & = & \{(i \mapsto \top) \mid i \in dom(A_A)\} \\ asg\ t\ A_A\ s & = & \{(i \mapsto A_A(i) \vee t) \mid i \in s\} \\ & & \cup\{(i \mapsto A_A(i)) \mid i \in dom(A_A) \setminus s\} \\ addr_A\ i\ A_A & = & \{i\} \\ addr_A\ [\![\mathsf{deref}\ p]\!]\ A_A & = & \bigvee A_A(addr_A\ p\ A_A) \end{array}$$

$$
\begin{array}{rcl}
\mathcal{S}_A\ [\![\mathsf{while}\ e\ \mathsf{do}\ s]\!]\ A_A\ t & = & \mathsf{let}\ A'_A = \mathcal{S}_A\ s\ A_A\ (t \vee \mathcal{E}_A\ e\ A_A) \\
 & & \mathsf{in\ if}\ A'_A \leq A_A\ \mathsf{then}\ A_A\ \mathsf{else}\ \mathcal{S}_A\ [\![\mathsf{while}\ e\ \mathsf{do}\ s]\!]\ A'_A\ t \\
\mathcal{S}_A\ [\![p\ :=\ e]\!]\ A_A\ t & = & asg\ (\mathcal{E}_A\ e\ A_A \vee t)\ A_A\ (addr_A\ p\ A_A) \\
\mathcal{S}_A\ [\![s_1; s_2]\!]\ A_A\ t & = & \mathcal{S}_A\ s_2\ (\mathcal{S}_A\ s_1\ A_A\ t)\ t
\end{array}
$$

To relate the instrumented and abstract semantics an ordering between instrumented and abstract values is defined relative to an environment:

$$A \vdash \langle v, t \rangle \sqsubseteq a$$

if and only if $a = \bot \Rightarrow t = \bot$ and $a \subseteq I \Rightarrow (t = \bot\ and\ v \in A(a))$.

Informally, the first implication means that if the abstract semantics says that a value is a trustworthy scalar then indeed it is marked trusted in the instrumented semantics. The second implication means that if the abstract semantics thinks a value is a pointer to one of the variables in a set a then by the instrumented semantics the value is indeed trustworthy and is a pointer to one of the variables in the set a.

The relation is extended to relate combined instrumented memories and environments with abstract environments like this:

$$M_I \circ A \sqsubseteq A_A$$

if and only if $dom(M_I \circ A) = dom(A_A)$ and $A \vdash (M_I \circ A)(i) \sqsubseteq A_A(i)$ for all variables $i \in dom(A_A)$

We relate the abstract interpretation to the instrumented semantics in the following way:

Proposition 1 (Safety) *If a statement is executed in an environment A and a memory M_I by the instrumented semantics, and the abstract environment A_A is a safe approximation of A and M_I then the result of the abstract interpretation is a safe approximation of the memory resulting from the instrumented semantics. Formally: If*

$$\mathcal{S}_I\ s\ A\ M_I\ t\ =\ M',\ \ M_I \circ A \sqsubseteq A_A,\ \ \mathcal{S}_A\ s\ A_A\ t_A\ =\ A'\ \ and\ \ t \leq t_A$$

then $M' \circ A \sqsubseteq A'$.

Proof. See Appendix A.

The abstract interpretation terminates. It is clear that $\mathcal{E}_A$ terminates as it is defined inductively in the (finite) structure of expressions, and no fixpoints are computed. The only possibility for $\mathcal{S}_A$ to diverge would be in the while case where a fixpoint is computed, but by Lemma 4 the fixpoint is computed of a monotone function over a lattice of finite height, hence the fixpoint can be found in finite time by iteration.

If we let n denote the number of distinct variables used in a program, let l denote the number of statements and expressions, and let m denote the greatest depth of while-loop nests in the program, the number of least upper bound operations on Val_A executed by the abstract interpretation will be in $O((n+l)^{2m})$. In the worst case, the least upper bound operation on Val_A can be computed in $O(n)$ time. This sounds worse than it really is. For ordinary programs m will be a small constant, and the complexity of analyzing a while-loop is at most $O(n_b^2)$ times the complexity of analyzing the loop body. Here n_b is the number of *pointer* variables occurring in the body of the loop.

If procedures are added to the language, fixpoints need to be computed for each procedure call, hence the time complexity will be even worse in that case.

Apart from the time complexity, the main drawback of the abstract interpretation analysis is that it needs the world to be closed; that is, the analysis cannot be run for each program module separately. In the next section a separable constraint based analysis is presented.

6 Constraint Generation

- Or else, what follows?
- Bloody constraint!...

William Shakespeare: Henry V, Act II, Scene 4.

The constraint generator is going to associate three constraint variables to each program variable. A solution to the generated set of constraints will assign an appropriate trust value for the program variable to one of these constraint variables.

The constraint analysis constructs constraints from any sequence of statements. This is more general than simply allowing for separate analysis of individual functions, since any sequence of statements can be (partly) analyzed out of context. This might for example be useful with an advanced module system like the Beta fragment system [7].

For the purpose of this article, a program consists of a top fragment that includes zero or more fragments which may again include smaller fragments and so on. The inclusion ordering of the fragments form a directed acyclic graph (DAG), as a single fragment may be included more than once, but we disallow circular dependencies.

Fragments are supposed to be analyzed in a bottom-up fashion, first analyzing the leaf fragments that include no other fragments, then analyzing fragments that include only leaf fragments and so on. In effect, the fragments are treated in reverse topological order.

The domains used in the definition of the constraint generation analysis are defined below:

$$
\begin{array}{rcl}
V & ::= & I \mid \nabla I \mid \Delta I \\
\delta, \eta & : & V \to V \\
N & : & P \to V \\
Ct & = & (V \cup Tr) \times V \\
C & = & 2^{Ct} \\
\mathcal{E}_S & : & E \to C \times V \\
\mathcal{S}_S & : & S \to C \times 2^V \\
G & \in & V
\end{array}
$$

V is the set of constraint variables. For an identifier i, Δi, and ∇i are simply constraint variables. The intuition is that whereas i will hold the trustworthiness of the value of the program variable i, Δi will hold the trust of all the values reachable by dereferencing i any number of times. Constraint variables ∇i are used to hold the trust of addr terms.

G is a special constraint variable corresponding to the global trustworthiness of a memory. That is, if a value is assigned to the target of an untrusted pointer then that value could end up anywhere, and the trustworthiness of the entire memory is corrupted.

The pair $\langle s, t\rangle \in Ct$ codes the constraint $s \leq t$. For readability we write $\{s \leq t\}$ for such a constraint and $\{s = t\}$ as an abbreviation for $\{s \leq t, t \leq s\}$. The generated constraints will be of the form $\{variable\ or\ constant\ \leq\ variable\}$ over the two element lattice $\{\bot, \top\}$, hence they can be solved by simple constraint propagation in linear time. The existence of a solution is guaranteed since assigning $\top$ to all constraint variables will satisfy the generated constraints.

We assume that any set of constraints include the constraints $\{i \leq \Delta i\}$ for all identifiers i.

The function δ on V "dereferences" constraint variables:

$$\begin{array}{rcl} \delta\ \nabla i & = & i \\ \delta\ i & = & \Delta i \\ \delta\ \Delta i & = & \Delta i \end{array}$$

The function η "safely" takes the address of a constraint variable.

$$\begin{array}{rcl} \eta\ \nabla i & = & \nabla i \\ \eta\ i & = & \nabla i \\ \eta\ \Delta i & = & \Delta i \end{array}$$

The map N generates constraint variables from pointer expressions P:

$$\begin{array}{rcl} N\ i & = & i \\ N[\![\mathsf{deref}\ p]\!] & = & \delta N(p) \end{array}$$

$\mathcal{E}_S$ generates constraints for expressions together with the variable corresponding to the given expression. In each case n denotes a freshly created constraint variable.

$$\begin{array}{rcl} \mathcal{E}_S\ i & = & \langle \emptyset, i \rangle \\ \mathcal{E}_S[\![\mathsf{addr}\ i]\!] & = & \langle \emptyset, \nabla i \rangle \\ \mathcal{E}_S[\![\mathsf{deref}\ p]\!] & = & \mathsf{let}\ \langle c, v \rangle = \mathcal{E}_S\ p \\ & & \mathsf{in}\ \langle c, \delta v \rangle \\ \mathcal{E}_S\ [\![e_1 + e_2]\!] & = & \mathsf{let}\ \langle c_1, v_1 \rangle = \mathcal{E}_S\ e_1 \\ & & \quad \langle c_2, v_2 \rangle = \mathcal{E}_S\ e_2 \\ & & \mathsf{in}\ \langle c_1 \cup c_2 \cup \{v_1 \leq n,\ v_2 \leq n\},\ n \rangle \\ \mathcal{E}_S[\![\mathsf{trust}\ e]\!] & = & \langle \emptyset, n \rangle \\ \mathcal{E}_S[\![\mathsf{distrust}\ e]\!] & = & \langle \{\top \leq n\}, n \rangle \\ \mathcal{E}_S\ const & = & \langle \emptyset, n \rangle \end{array}$$

$\mathcal{S}_S$ generates constraints for statements. The second part of the result is the set of constraint variables corresponding to variables assigned to within the statement. This is used to generate additional constraints for while-loops such that variables assigned to in the loop body are trusted only if the condition of the loop is.

$$\begin{array}{rcl} \mathcal{S}_S[\![\mathsf{while}\ e\ \mathsf{do}\ s]\!] & = & \mathsf{let}\ \langle c_e, v \rangle = \mathcal{E}_S\ e \\ & & \quad \langle c_s, a \rangle = \mathcal{S}_S\ s \\ & & \mathsf{in}\ \langle c_s \cup c_e \cup \{v \leq x \mid x \in a\},\ a \rangle \end{array}$$

$$
\begin{array}{rcl}
\mathcal{S}_S[\![p \ := \ e]\!] & = & \mathsf{let}\ \langle c_e, v\rangle = \mathcal{E}_S\ e \\
& & \mathsf{in}\ \langle c_e \cup \{v \leq N(p),\, \delta N(p) = \delta v, \eta N(p) \leq G\},\, \{N(p)\}\rangle \\
\mathcal{S}_S[\![s_1; s_2]\!] & = & \mathsf{let}\ \langle c_1, a_1\rangle = \mathcal{S}_S\ s_1 \\
& & \quad \langle c_2, a_2\rangle = \mathcal{S}_S\ s_2 \\
& & \mathsf{in}\ \langle c_1 \cup c_2,\, a_1 \cup a_2\rangle
\end{array}
$$

A solution to the generated constraints (called a *model*) is a map m giving values to the constraint variables such that the constraints c are fulfilled, this is written $m \models c$. Formally: $m \models c$ if and only if

$$\forall \langle s, t\rangle \in c: \ m(s) \leq m(t).$$

It is clear that if $m \models c_1 \cup c_2$ then $m \models c_1$ and $m \models c_2$.

We will consider only a subset of all possible models for a set of constraints, namely so-called *coherent* models. A model m is coherent if it satisfies

$$m(a) \leq m(b) \ \Rightarrow \ m(\delta a) \leq m(\delta b).$$

It is clear that the model that assigns $\top$ to all variables is a coherent model, hence the existence of a coherent model is assured.

Coherent models and abstract environments can be related to each other in the following way: We write $A_A \sqsubseteq m$ if and only if

$$A_A(i) = \top \ \Rightarrow \ m(i) = \top$$

and

$$a \in A_A(i) \ \Rightarrow \ m(\Delta i) = m(a),$$

or, alternatively $m(G) = \top$.

An intuitive view of the above is that in order for a model to be a safe approximation of an abstract environment, it must assign conservative trust-values to all variables, and if a pointer p can point to a number of variables then the constraint variable Δp must be equated to the trust-values of all these variables.

The constraint generation analysis is related to the abstract interpretation by the following safety statement:

Proposition 2 *If*

$$
\begin{array}{rcl}
\langle c, v\rangle & = & \mathcal{S}_S\ s, \\
m & \models & c, \text{ and } m \text{ is coherent} \\
A_A & \sqsubseteq & m, \\
\forall x \in v & : & t \leq m(x), \\
A'_A & = & \mathcal{S}_A\ s\ A_A\ t
\end{array}
$$

then $A'_A \sqsubseteq m$.

Proof. See Appendix B.

The constraint generation analysis is strictly weaker than the abstract interpretation in the sense that more variables are treated as untrusted, as is demonstrated by the following example:

Program	New constraint
p := addr j	$\{\nabla j \leq p,\, \Delta p = j\}$
p := addr i	$\{\nabla i \leq p,\, \Delta p = i\}$
i := distrust 8	$\{\top \leq i\}$
k := deref p	$\{\Delta p \leq k,\, \Delta k = \Delta p\}$

Remember that the following constraint is implicitly assumed: $\{p \ \leq \ \Delta p\}$. In the abstract interpretation, only i will be marked untrusted at the end, whereas in the constraint analysis the trust of i and j are linked by equality since p may point to both[2].

Generating the constraints for a program of size n takes $O(n^2)$ time in the worst case assuming that the addition of a single constraint can be done in constant time. The constraints, being of such simple nature, may be solved by value propagation in linear time in the number of constraints. All in all constraint generation and (partial) solving can be done in quadratic time in the size of the program fragment.

7 Extensions

By treating arrays as one logical variable, the analysis is able to handle arrays as well as scalar data. This means that the analysis cannot know that some elements of an array are trusted and some are not. Either all elements are trusted or none are. This tradeoff is necessary for the abstract and constraint analyses since they are unable to compute actual offsets in the array. This tradeoff in accuracy is the same as encountered in set-based analysis [5].

Records or structs can be handled by treating each field of the record as a separate variable.

Extending the language with first order procedures is simple enough. The abstract interpretation will simply model the procedure calls directly and compute fixed points in case of recursion. The constraint generation will first compute constraints for the body of a procedure and for each call add constraints matching formal and actual parameters. By copying the constraints generated for the body we can achieve a polyvariant analysis such that a particular call of the procedure with an untrustworthy argument does not influence other calls of that procedure.

A "check for trusted value" construct that will raise an error when an untrusted value is given as parameter is easily added to the language, but makes the semantics larger and a bit more complicated as it has to deal with abnormal termination. The relation between the instrumented and abstract interpretation must state that if the instrumented semantics says that a program will fail then the abstract interpretation will too. Extending the constraint generation analysis with the "check" construct means that there will only be a model for the generated constraints if all checks are met.

Extending the analysis to languages with higher order functions while still catering for pointers and mutable data seems to be more complicated and is left for future research.

The concept of trust can be extended to multiple levels of trust, so that instead of a binary lattice of trust values, a lattice with longer chains was used. For the instrumented semantics and the constraint generation, this is a straightforward generalization. For the

[2] As remarked by one of the referees, it might be possible to detect some of these situations as p is *dead* after the first assignment, so one might remove the constraints added in the first line from the final constraints and thereby get a better solution. This effect might also be achieved by removing assignments to dead variables before trust analysis.

abstract interpretation, the abstract domain is changed such that all “very trusted” pointers are below the “lesser trusted” pointers all of which are below $\top$.

8 Conclusion

We have argued that the analysis of the trustworthiness of data is a useful program analysis in security conscious settings, and we have given two static analyses for this purpose, one based on abstract interpretation, and another constraint analysis that facilitates separate analysis of program modules at the cost of slightly less accuracy.

The analyses have been proved safe with respect to an instrumented semantics that has served as the definition of the goal of the analysis.

The main contribution of this paper is thought to be the introduction of the concept of trust analysis, and the application of it to a language with pointers and mutable data

Currently, work is in progress together with Jens Palsberg to formulate trust analysis for a higher order language with polymorphic functions in terms of a type inference system. There are some similarities between binding-time analysis [6] and trust analysis in this case, but there are also significant differences. Most notably, in binding-time analysis: if an argument is used by a function that expects a dynamic argument, the argument itself has to be marked dynamic, and the “dynamicness” propagates back through the argument. Not so in trust analysis. There the argument can be “lifted” from trusted to untrusted in that place without affecting other parts of the program.

Acknowledgments: The author wants to thank Jens Palsberg, Peter D. Mosses and Neil D. Jones for reading earlier drafts of this paper and giving useful comments. Also the anonymous referees provided useful feedback.

A Safety of Abstract Interpretation

Fact 1 *If $a \leq b$ and $b \in s \subseteq Val_A$ then $a \leq \bigvee s$.*

Fact 2 *If $A \vdash v \sqsubseteq a$ and $a \leq b$ then $A \vdash v \sqsubseteq b$.*

Lemma 1 *If $M_I \circ A \sqsubseteq A_A$ then $A \vdash \ \mathcal{E}_I\ e\ A\ M_I \sqsubseteq \mathcal{E}_A\ e\ A_A$.*

Proof. By structural induction on e. We proceed with a case analysis:

- $e = i$: Show $A \vdash \ M_I(A(i)) \sqsubseteq A_A(i)$ which follows from the definition of $\sqsubseteq$.
- $e = [\![\mathsf{addr}\ i]\!]$: Show $A \vdash \ \langle A(i), \bot\rangle \ \sqsubseteq \{i\}$, and clearly $A(i) \in A(\{i\})$.
- $e = [\![\mathsf{deref}\ p]\!]$: Let $\langle v, t\rangle = \mathcal{E}_I\ p\ A\ M_I$ and $a = \mathcal{E}_A\ p\ A_A$, show:

$$A \vdash \langle \pi_1(M_I(v)), t \vee \pi_2(M_I(v))\rangle \ \sqsubseteq \bigvee A_A(a).$$

  By induction, $A \vdash \langle v, t\rangle \ \sqsubseteq a$. By strong typing we can assume that v is indeed a pointer and that either $a = \top$ or $a \subseteq I$. In the first case the desired inequality holds trivially. In the second case we know that $v \in A(a)$, and also that $t = \bot$. Assume that $v = A(a_0),\ a_0 \in a$. As $M_I \circ A \sqsubseteq A_A$, $A \vdash M_I(v) \sqsubseteq A_A(a_0)$, and using Fact 1 and Fact 2 we get the result.

- $e = [\![e_1 + e_2]\!]$: By induction,

$$A \vdash \ (\langle v_1, t_1\rangle \ = \mathcal{E}_I\ e_1\ A\ M_I) \sqsubseteq \mathcal{E}_A\ e_1\ A_A = a_1,$$

$$A \vdash (\langle v_2, t_2\rangle \ = \mathcal{E}_I \ e_2 \ A \ M_I) \sqsubseteq \mathcal{E}_A \ e_2 \ A_A = a_2.$$

Show $A \vdash \ \langle v_1 + v_2, t_1 \vee t_2\rangle \ \sqsubseteq \ a_1 \vee a_2$. If one of $\{a_1, a_2\}$ is $\top$, the result is trivial. If they are both $\bot$, both t_1 and t_2 must be too.

- $e = [\![\mathsf{trust}\ e']\!]$: Show $A \vdash \langle \mathcal{E}_I \ e' \ A \ M_I, \ \bot\rangle \sqsubseteq \bot$. This follows directly from the definition of $\sqsubseteq$.
- $e = [\![\mathsf{distrust}\ e']\!]$: Trivial from the definitions.
- $e = const$: Show $A \vdash \langle const, \bot\rangle \ \sqsubseteq \ \bot$, which is trivial.

Lemma 2 *If $M_I \circ A \sqsubseteq A_A$ then*

$$A \vdash \langle addr_I \ p \ A \ M_I, \ \bot\rangle \sqsubseteq addr_A \ p \ A_A$$

Proof. By structural induction in p.

- $p = i$: Show $A \vdash \langle A(i), \bot\rangle \sqsubseteq \{i\}$ which follows directly from the definition of $\sqsubseteq$.
- $p = [\![\mathsf{deref}\ p']\!]$: Show $A \vdash \langle \pi_1(M_I(addr_I \ p' \ A \ M_I)), \bot\rangle \sqsubseteq \bigvee A_A(addr_A \ p' \ A_A)$. By induction: $A \vdash \langle addr_I \ p' \ A \ M_I, \ \bot\rangle \sqsubseteq addr_A \ p' \ A_A$. If $addr_A \ p' \ A_A \ = \ \top$ the result is trivial. If $addr_A \ p' \ A_A = s \subseteq I$ then there is an identifier $a_0 \in s$ such that $A(a_0) = addr_I \ p' \ A \ M_I$. Thus $A \vdash M_I(A(a_0)) \sqsubseteq A_A(a_0)$ by the assumption that $M_I \circ A \sqsubseteq A_A$, and via Fact 1 and 2 the result follows.

Lemma 3 *$\mathcal{E}_A$ is monotone in its second argument:*

$$A_A \leq A'_A \ \Rightarrow \ \mathcal{E}_A \ e \ A_A \leq \mathcal{E}_A \ e \ A'_A.$$

Proof. Trivial by structural induction in e.

Lemma 4 *$\mathcal{S}_A$ is monotone in its second argument:*

$$A_A \leq A'_A \ \Rightarrow \ \mathcal{S}_A \ s \ A_A \ t \leq \mathcal{S}_A \ s \ A'_A \ t$$

Proof. By structural induction in s.

Proof of Proposition 1 (Safety). We want to prove the following: If

$$\mathcal{S}_I \ s \ A \ M_I \ t \ = \ M', \ \ M_I \circ A \sqsubseteq A_A, \ \ \mathcal{S}_A \ s \ A_A \ t_A \ = \ A'_A \ \ and \ \ t \leq t_A$$

then $M' \circ A \sqsubseteq A'_A$.

The proof is by induction in the number of calls of $\mathcal{S}_I$. We proceed by a case analysis of the syntax of s:

- $s = [\![\mathsf{while}\ e\ \mathsf{do}\ s']\!]$: By Lemma 1, monotonicity of $\mathcal{E}_A$ and Fact 2 we know that $A \vdash \mathcal{E}_I \ e \ A \ M_I \sqsubseteq \mathcal{E}_A \ e \ A_A$.

  If v is false (in the definition of $\mathcal{S}_I$) the result follows from monotonicity.

  Otherwise, let $\langle v, t'\rangle \ = \ \mathcal{E}_I \ e \ A \ M_I, \ M'' \ = \ \mathcal{S}_I \ s' \ A \ M_I \ (t \vee t')$ and $A''_A \ = \ \mathcal{S}_A \ s' \ A_A \ (t_A \vee \mathcal{E}_A \ e \ A_A)$. By the above fact on e we can apply induction and get $M'' \circ A \sqsubseteq A''_A$.

  Now we have $M' = \mathcal{S}_I \ [\![\mathsf{while}\ e\ \mathsf{do}\ s']\!] \ A \ M'' \ t$ and

  $$A'_A = \mathcal{S}_A \ [\![\mathsf{while}\ e\ \mathsf{do}\ s']\!] \ A_A \ t_A = \mathcal{S}_A \ [\![\mathsf{while}\ e\ \mathsf{do}\ s']\!] \ A''_A \ t_A.$$

  By induction we get $M' \circ A \sqsubseteq A'_A$.

- $s = [\![p\ :=\ e]\!]$: Let $\langle v, t' \rangle = \mathcal{E}_I\ e\ A\ M_I$, $v_A = \mathcal{E}_A\ e\ A_A$, $a = addr_I\ p\ A\ M_I$ and $a_A = addr_A\ p\ A_A$. We need to show:

$$M_I[\langle v, t \vee t' \rangle / a] \circ A \sqsubseteq asg\ (v_A \vee t_A)\ A_A\ a_A.$$

  If $a_A = \top$ then this follows directly from the definition of asg. Otherwise by Lemma 2 we have $A \vdash \langle a, \bot \rangle \sqsubseteq a_A$, hence there exists an $a_0 \in a_A$ such that $A(a_0) = a$. It is enough to ensure the inequality at a_0 since this is the only point where the left hand side is different from $M_I \circ A$ and asg is clearly monotone in the second argument so by Fact 2 the inequality holds automatically everywhere else. Evaluating we get:

$$(asg\ (v_A \vee t_A)\ A_A\ a_A)(a_0) = (v_A \vee t_A \vee A_A(a_0)).$$

  and

$$(M_I[\langle v, t \vee t' \rangle / a] \circ A)(a_0) = \langle v, t \vee t' \rangle.$$

  By Lemma 1 we know that $A \vdash \langle v, t' \rangle \sqsubseteq v_A$. All that remains to show is: $A \vdash \langle v, t \vee t' \rangle \sqsubseteq v_A \vee t_A \vee A_A(a_0)$ which follows from Fact 1.

- $s = [\![s_1; s_2]\!]$: This case follows immediately by two applications of induction.

B Safety of Constraint Generation

Lemma 5 (Addresses) *If m is a coherent model, $m(G) = \bot$ and $A_A \sqsubseteq m$ then these two implications hold:*

$$a \in addr_A\ p\ A_A \subseteq I \ \Rightarrow\ m(a) = m(N(p))$$

and

$$addr_A\ p\ A_A = \top \ \Rightarrow\ \top = m(\eta N(p)) \leq m(N(p)).$$

Proof. By structural induction in p.

- $p = i$: $addr_A\ i\ A_A = \{i\}$ and $m(i) = m(N(p)) = m(i)$.
- $p = [\![\mathsf{deref}\ p']\!]$: Note that $m(N(p')) \leq m(\eta N(p))$. First assume $a \in addr_A\ p\ A_A = \bigvee A_A(addr_A\ p'\ A_A) \subseteq I$. By induction, $b \in addr_A\ p'\ A_A \Rightarrow m(b) = m(N(p'))$. Since m is coherent, $m(\delta b) = m(\delta N(p')) = m(N(p))$. Also, as $A_A \sqsubseteq m$: $m(\delta b) = m(\Delta b) = m(a)$. Combining the equalities we get the desired result.

  Secondly, suppose $\bigvee A_A(addr_A\ p'\ A_A) = \top$. Either $addr_A\ p'\ A_A = \top$ in which case induction yields $\top = m(N(p')) \leq m(\delta N(p')) = m(N(p))$, or there is some $b_0 \in addr_A\ p'\ A_A$ such that $A_A(b_0) = \top$. Since m is a safe approximation of A_A this means $m(b_0) = \top$. By induction $m(b) = m(N(p'))$ for all $b \in addr_A\ p'\ A_A$ so we get $\top = m(b_0) = m(N(p')) \leq m(N(p))$ which is the required result.

Lemma 6 (Expressions) *The constraints generated for expressions safely approximate the abstract interpretation of expressions.*

Suppose $\langle c, v \rangle = \mathcal{E}_S\ e$, m is a coherent model of c, $A_A \sqsubseteq m$ and $a = \mathcal{E}_A\ e\ A_A$ then the following implications hold:

$$a = \top \ \Rightarrow\ m(v) = \top$$

and

$$a_0 \in a \subseteq I \ \Rightarrow\ m(a_0) = m(\delta v).$$

Proof. By structural induction in e.

- $e = i$: $\mathcal{E}_A\ i\ A_A = A_A(i)$ and $\langle c, v\rangle = \langle \emptyset, i\rangle$ by definition. If $A_A(i) = \top$ then $m(i) = m(v) = \top$ as $A_A \sqsubseteq m$. If $a_0 \in A_A(i)$ then $m(\Delta i) = m(\delta i) = m(a_0)$ by the same reason.
- $e = [\![\mathsf{addr}\ i]\!]$: $\langle c, v\rangle = \langle \emptyset, \nabla i\rangle$ and $a = \{i\}$. What is required to prove thus is $m(a_0) = m(\delta v) = m(i)$ for $a_0 \in \{i\}$ which is clear.
- $e = [\![\mathsf{deref}\ p]\!]$: $\langle c, v_p\rangle = \mathcal{E}_S\ p$, $v = \delta v_p$ and $a = \bigvee A_A(\mathcal{E}_A\ p\ A_A)$.

  If $a = \top$ then either $\mathcal{E}_A\ p\ A_A = \top$ and by induction $\top = m(v_p) \leq m(\delta v_p) = m(v)$, or $\mathcal{E}_A\ p\ A_A \subseteq I$ in which case there is some $a_0 \in \mathcal{E}_A\ p\ A_A$ such that $A_A(a_0) = \top$. As $A_A \sqsubseteq m$ this means that $m(a_0) = \top$. By induction $\top = m(a_0) = m(\delta v_p) = m(v)$.

  If $a_0 \in a \subseteq I$ then we must show $m(a_0) = m(\delta v)$. By induction $m(a_0') = m(\delta v_p)$ for all $a_0' \in \mathcal{E}_A\ p\ A_A \subseteq I$. $a_0 = A_A(a_0')$ for some such a_0' thus since $A_A \sqsubseteq m$, $m(\Delta a_0') = m(a_0)$ and since m is coherent:

  $$m(a_0') = m(\delta v_p) = m(v) \Rightarrow m(a_0) = m(\delta a_0') = m(\delta v).$$

- $e = [\![e_1 + e_2]\!]$: Let $\langle c_1, v_1\rangle = \mathcal{E}_S\ e_1$ and $\langle c_2, v_2\rangle = \mathcal{E}_S\ e_2$. We have $c = c_1 \cup c_2 \cup \{v_1 \leq v, v_2 \leq v\}$ and by induction the implications hold for the two subexpressions. Suppose $a = \top$: This means that $\mathcal{E}_A\ e_j\ A_A = \top$ for some $j \in \{1, 2\}$ and by induction this means that $m(v_j) = \top$ and by definition of c we get $m(v) = \top$.

  By strong typing, the abstract value for the expression must be either $\top$ or $\bot$ so this concludes the case.
- $e = [\![\mathsf{trust}\ e']\!]$: We have $a = \bot$ so the implications hold vacuously.
- $e = [\![\mathsf{distrust}\ e']\!]$: We have $a = \top$ and $c = \{\top \leq v\}$ hence $m(v) = \top$ as required.
- $e = const$: We have $a = \bot$ so the implications hold vacuously.

Proof of Proposition 2. We want to prove the following: If

$$\begin{array}{rcll} \langle c, v\rangle & = & \mathcal{S}_S\ s, & (1)\\ m & \models & c,\ and\ m\ is\ coherent & (2)\\ A_A & \sqsubseteq & m, & (3)\\ \forall x \in v & : & t \leq m(x), & (4)\\ A_A' & = & \mathcal{S}_A\ s\ A_A\ t & (5) \end{array}$$

then $A_A' \sqsubseteq m$.

We proceed by induction in the number of calls to $\mathcal{S}_A$. If $m(G) = \top$ then the final inequality holds regardless of A_A', so assume $m(G) = \bot$. A case analysis follows:

- $s = [\![\mathsf{while}\ e\ \mathsf{do}\ s']\!]$: Let $\langle c_e, v_e\rangle = \mathcal{E}_S\ e$ and $\langle c_s, v_s\rangle = \mathcal{S}_S\ s'$. By definition of c: $x \in v_s \Rightarrow m(v_e) \leq m(x)$ and by Lemma 6 $\mathcal{E}_A\ e\ A_A = \top \Rightarrow m(v_e) = \top$ thus by (4) $\forall x \in v_s : t \vee \mathcal{E}_A\ e\ A_A \leq m(x)$. We can now apply induction on s' and get $A_A' = \mathcal{S}_A\ s'\ A_A\ (t \vee \mathcal{E}_A\ e\ A_A) \sqsubseteq m$. If this is the same as A_A we are done. Otherwise we apply induction once more and get the result.
- $s = [\![p\ :=\ e]\!]$: If $addr_A\ p\ A_A = \top$ then by Lemma 5, $\top = m(\eta N(p)) \leq m(G)$ so in that case $A_A' \sqsubseteq m$ by definition of $\sqsubseteq$.

  Now suppose $a_0 \in a = addr_A\ p\ A_A \subseteq I$. A_A' differs from A_A only on the set a by definition of asg. Let $\langle c_e, v_e\rangle = \mathcal{E}_S\ e$ and $a_e = \mathcal{E}_A\ e\ A_A$.

If $A'_A(a_0) = t \vee A_A(a_0) \vee a_e = \top$ we must show $m(a_0) = \top$. By (4) $t \leq m(N(p)) = m(a_0)$ where that last equality comes from Lemma 5. By (3) $A_A(a_0) = \top \Rightarrow m(a_0) = \top$. By Lemma 6 $a_e = \top \Rightarrow m(v_e) = \top$, and by definition of c, $m(v_e) \leq m(v) = m(N(p)) = m(a_0)$, using Lemma 5 last. For $A'_A(a_0)$ to be $\top$ at least one of the parts of the above disjunction must be $\top$ (by definition of the Val_A lattice) and by the inequalities, $m(a_0) = \top$ in all cases.

If $A'_A(a_0) = t \vee A_A(a_0) \vee a_e \subseteq I$ then we must show that $a' \in A'_A(a_0) \Rightarrow m(\Delta a_0) = m(a')$. a' cannot belong to t as $t \in Tr$. If $a' \in A_A(a_0)$ then (3) secures the result. Otherwise, if $a' \in a_e$ then by Lemma 6 $m(a') = m(\delta v_e) = m(\delta N(p))$ where the last equality stems from the definition of c. By Lemma 5 and coherence $m(\delta N(p)) = m(\delta a_0) = m(\Delta a_0)$.

- $s = [\![s_1; s_2]\!]$: Let $A''_A = \mathcal{S}_A\ s_1\ A_A\ t$ and $\langle c_1, v_1 \rangle = \mathcal{S}_S\ s_1$. Now $c_1 \subseteq c$ and $v_1 \subseteq v$ by definition of $\mathcal{S}_S$, so by induction we get $A''_A \sqsubseteq m$. With this and equivalent considerations as above we can apply induction to A''_A and s_2 and get $A'_A \sqsubseteq m$ as required.

References

[1] CERT Advisory 94:12 Sendmail Vulnerability. Technical report, CERT, 1994. URL: ftp://ftp.cert.org/.

[2] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[3] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–242, May 1976.

[4] D. E. Denning and P. J. Denning. Certifications of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–512, July 1977.

[5] N. Heintze. Set-Based Analysis of ML Programs. Technical Report CMU-CS-93-193, CMU School of Computer Science, 1993. URL: ftp://reports.adm.cs.cmu.edu/usr/anon/1993/CMU-CS-93-193.ps.

[6] F. Henglein and C. Mossin. Polymorphic Binding-Time Analysis. In D. Sannella, editor, *Proceedings of the 1994 European Symposium on Programming (ESOP'94)*, volume 788 of LNCS, pages 287–301. Springer-Verlag, April 1994.

[7] J. L. Knudsen, M. Löfgren, O. L. Madsen, and B. Magnusson. *Object Oriented Environments: The Mjølner Approach*. Prentice-Hall, 1993. ISBN 0-13-009291-6.

[8] P. Ørbæk. Can you Trust your Data? In P. D. Mosses, editor, *Proceedings of the TAPSOFT/FASE'95 Conference*, volume 915 of *Springer Lecture Notes in Computer Science*, pages 575–590, Aarhus, Denmark, May 1995. Springer-Verlag. URL: ftp://ftp.daimi.aau.dk/pub/empl/poe/index.html.

[9] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly and Associates, 1991.

Recent Publications in the BRICS Report Series

RS-95-24 Peter Ørbæk. *Can you Trust your Data?* April 1995. 15 pp. Appears in Mosses, Nielsen, and Schwartzbach, editors, *Theory and Practice of Software Development. 6th International Joint Conference CAAP/FASE*, TAPSOFT '95 Proceedings, LNCS 915, 1995, pages 575–590.

RS-95-23 Allan Cheng and Mogens Nielsen. *Open Maps (at) Work*. April 1995. 33 pp.

RS-95-22 Anna Ingólfsdóttir. *A Semantic Theory for Value–Passing Processes, Late Approach, Part II: A Behavioural Semantics and Full Abstractness*. April 1995. 33 pp.

RS-95-21 Jesper G. Henriksen, Ole J. L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders B. Sandholm. *MONA: Monadic Second-Order Logic in Practice*. May 1995. 17 pp.

RS-95-20 Anders Kock. *The Constructive Lift Monad*. March 1995. 18 pp.

RS-95-19 François Laroussinie and Kim G. Larsen. *Compositional Model Checking of Real Time Systems*. March 1995. 20 pp.

RS-95-18 Allan Cheng. *Complexity Results for Model Checking*. February 1995. 18pp.

RS-95-17 Jari Koistinen, Nils Klarlund, and Michael I. Schwartzbach. *Design Architectures through Category Constraints*. February 1995. 19 pp.

RS-95-16 Dany Breslauer and Ramesh Hariharan. *Optimal Parallel Construction of Minimal Suffix and Factor Automata*. February 1995. 9 pp.

RS-95-15 Devdatt P. Dubhashi, Grammati E. Pantziou, Paul G. Spirakis, and Christos D. Zaroliagis. *The Fourth Moment in Luby's Distribution*. February 1995. 10 pp. To appear in *Theoretical Computer Science*.

RS-95-14 Devdatt P. Dubhashi. *Inclusion–Exclusion*(3) *Implies Inclusion–Exclusion*(n). February 1995. 6 pp.