Electronic Communications of the EASST
Volume 35 (2010)

Proceedings of the
10th International Workshop on
Automated Verification of Critical Systems
(AVoCS 2010)

Static Analysis of Information Release in Interactive Programs

Adedayo O. Adetoye and Nikolaos Papanikolaou

15 pages

Guest Editors: Jens Bendisposto, Michael Leuschel, Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Static Analysis of Information Release in Interactive Programs

Adedayo O. Adetoye and Nikolaos Papanikolaou

International Digital Laboratory, WMG, University of Warwick, UK

Abstract: In this paper we present a model for analysing information release (or leakage) in programs written in a simple imperative language. We present the semantics of the language, an attacker model, and the notion of an information release policy. Our key contribution is the static analysis technique to compute information release of programs and to verify it against a policy. We demonstrate our approach by analysing information released to an attacker by faulty password checking programs; our example is inspired by a known flaw in versions of OpenSSH distributed with various Unix, Linux, and OpenBSD operating systems.

Keywords: Secure Information Release, Static Analysis, Program Verification.

1 Introduction

It is often inevitable, during the course of program execution, for sensitive information to be leaked to the environment; in the presence of an attacker, such leakage — henceforth *information release* — can be catastrophic, or at the very least damaging, to the parties with whom the data is concerned. Ensuring that information release is minimal is a critical requirement in a variety of applications; this is true, for instance, with authentication, encryption and statistical analysis software. General purpose applications infected by malicious code, or malware, may seek to release much more information than expected by the user; this is also the case with Trojan horses (think of a tax-return calculator that releases private financial information to an unauthorised observer). What the user generally expects in these applications is that the amount of information release does not exceed what is absolutely necessary for normal operation.

Therefore it is highly necessary to have means of controlling the information released by a program, while taking into account its purpose and functionality, namely, how it transforms its inputs to publicly observable output. The problem we are then concerned with is how to check whether the program does not release more than is specified. In other words, we seek a way of checking that a given program conforms to an *information release policy*.

In this paper we present static analysis techniques to measure the information released by a program, both qualitatively and quantitatively (using information theory), and develop a policy model whereby users' information release requirements may be specified. The intention is that, by comparing the information actually released by the program with a specification of its expected information release, as stated in a policy, we can judge whether the program has secure information flow and reject insecure implementations.

We demonstrate our approach by investigating attacks on password-checking programs, where timing delays can give clues to potential attackers about the validity of user log-in names and passwords. The examples are inspired by password checking programs used in different versions of OpenSSH on various Unix, Linux, and OpenBSD operating system.

Contributions. In this paper we present a general static analysis technique, parametrised by attacker models, for the verification of secure information flow in interactive programs. This includes a concrete static analysis technique for *While* programs under a "standard attacker" model, which can observe program outputs as prescribed by the standard operational semantics.

Our analysis is both flow-sensitive and termination-sensitive, accounting for sequencing of programs as well as correctly dealing with information release in the face of program divergence. We present a qualitative policy framework, whereby users may enforce secure information release requirements on programs. We also demonstrate how the qualitative policies can be described quantitatively, with examples. We show a limitation of the quantitative technique.

The overall architecture and framework is described in Section 2, while the static analysis of information release for arbitrary *While* programs is presented in Section 3. Information release policies are described in Section 4. We illustrate our analysis and enforcement technique by considering examples which exploit design, implementation and configuration flaws in password authentication programs in Section 5. The examples are motivated by flaws in version of the OpenSSH authentication module. Section 6 shows how the qualitative PER-based policies of Section 4 may also be expressed quantitatively using information theory. Section 7 concludes and looks at areas of future work.

Related Work. This paper describes a technique for the analysis of secure information release in computer programs, which is an established field [10]. This paper extends our earlier work [2], on the lattice-based formulation of secure information flow under various attacker models, with a concrete static analysis methodology. We are evaluating ways of efficiently implementing the proposed technique, which may benefit from minimal model generation results of [3]. In [1] a theorem-proving approach is presented that deals with noninterference properties of programs. The implementation for real languages is still further away, but we envisage applying the technique in the context of low-level code such as assembly or virtual machine languages [8, 14, 4].

2 Analysis and Enforcement Framework

Our approach forms the basis of a framework for analysing the security of programs. In particular, we envisage the technique of static analysis described in this paper as being implemented in a software tool, possibly a kernel module for various operating systems, which computes the information release of programs during installation or prior to that in a proof-carrying code setting [9]. A user would supply (or be supplied with, by a trusted source) an information release policy to this tool, and if a program fails to satisfy the requirement of the policy, its execution would be prevented and the user warned.

We have so far developed the theory of information release for programs expressed in a simple, but typical, imperative language. The static analysis rules described in this paper could be adapted to different languages and generalised to account for different types of attacker. As part of a long-term research programme, we will be targeting the analysis of low-level code for system software and applications running on mobile devices.

The analysis technique we are proposing would be used in a system architecture comprising: a set of users , programs assumed to be potentially hostile (until verified otherwise), an execution environment, a set of information release policies, a static analyser. Users are assumed to have legitimate uses for the programs in the system. The environment in which programs are executed is assumed to include attackers, potentially waiting locally or on other networks for the program to disclose confidential or sensitive information to them. That is, we consider the malicious code scenario, where the program may contain spyware or design or implementation flaws that can be used to reveal sensitive data that are ordinarily accessible only to the user and the programs.

Information release policies may be published by authors of programs; additionally, users can define their own information release policies for programs they use, in order to specify their expectations of information release. Most importantly, users control the application of the static analysis tool to programs they execute in order to check that their policies are satisfied. On one hand, if a program fails the analysis, this is an indication that it may contain exploitable flaws. On the other hand, programs that pass verification are provably secure against the attacker model used in the verification.

3 Static Analysis

In this section we present a static analysis of information flow in *While* programs.

3.1 Syntax and Semantics of the *While* Language.

In this section we present the core imperative language, *While*, which has loops and input-output interactions. Its syntax (Figure 1) and the operational semantics (Figure 2) are largely familiar.

$$c ::= \ \texttt{skip} \mid z := e \mid \texttt{read}\, z \mid \texttt{write}\, e \mid c;c \mid \texttt{if}\,(b)\,\texttt{then}\, c\, \texttt{else}\, c \mid \texttt{while}\,(b)\,\texttt{do}\, c$$

Figure 1: *The While Language*

In the language, expressions are either boolean-valued (with values taken from $\mathbb{B} \triangleq \{\mathbf{tt}, \mathbf{ff}\}$), or integer-valued (taken from $\mathbb{Z}$). Program states, Σ, are maps from variables to values. The evaluation of the expression e at the state $\sigma \in \Sigma$ is summarised as $\sigma(e)$. Expression evaluations are performed atomically and have no side-effect on state. A program action, ranged over by a, can either be an internal action τ, which is not observable ordinarily; or it can be an input action through the *read* command; or it can be an output action via the *write* command, where the expression value can be observed. Secret program inputs are treated as parameters. The operational semantics is specified through transition relations between expression configurations $(\langle e, \sigma\rangle \xrightarrow{\tau} \langle \sigma(e), \sigma\rangle)$ and command configurations $(\langle c, \sigma\rangle \xrightarrow{a} \langle c', \sigma'\rangle)$. A special *terminal command configuration*, $\langle \cdot, \sigma\rangle$, indicates termination in the state σ. The set of all command configurations, including the terminal command configuration is denoted by $\mathcal{S}$.

We adopt the *relational semantics* definition of [6], where program semantics is modelled as a relation $\langle\cdot\rangle \subset \Sigma_\infty \times \Sigma_\infty$ over the extended state space $\Sigma_\infty = \Sigma \cup \{\infty\}$, which is obtained by adding a special "looping state" ∞ to Σ. Thus, for any program c, $\sigma\ \langle c\rangle\ \sigma'$ holds if there exists a terminating state $\sigma' \in \Sigma$ of c when it is executed at the initial state of $\sigma \in \Sigma$; otherwise $\sigma\ \langle c\rangle\ \infty$ asserts the divergence of c under σ. Additionally, no program can exit the "looping state", so that

$\infty \langle c \rangle \infty$ always holds. Furthermore, we assume that $\langle c,\infty\rangle \xrightarrow{\tau} \langle c,\infty\rangle$. The angelic relational semantics $\langle\cdot\rangle_{\downarrow}$ restricts the domain and range of $\langle\cdot\rangle$ to Σ. The operators ; and $\cup$, when used with relations, are respectively the standard relational composition and union operators.

$$\langle \texttt{skip},\sigma\rangle \xrightarrow{\tau} \langle\cdot,\sigma\rangle \qquad \langle z := e,\sigma\rangle \xrightarrow{\tau} \langle\cdot,\sigma[z\mapsto\sigma(e)]\rangle \qquad \langle \texttt{read}\, z,\sigma\rangle \xrightarrow{\boldsymbol{in}(n)} \langle\cdot,\sigma[z\mapsto n]\rangle$$

$$\langle \texttt{write}\, e,\sigma\rangle \xrightarrow{\boldsymbol{out}(\sigma(e))} \langle\cdot,\sigma\rangle \qquad \frac{\langle c_1,\sigma\rangle \xrightarrow{a} \langle\cdot,\sigma'\rangle}{\langle c_1;c_2,\sigma\rangle \xrightarrow{a} \langle c_2,\sigma'\rangle} \qquad \frac{\langle c_1,\sigma\rangle \xrightarrow{a} \langle c_1',\sigma'\rangle}{\langle c_1;c_2,\sigma\rangle \xrightarrow{a} \langle c_1';c_2,\sigma'\rangle}$$

$$\frac{\langle b,\sigma\rangle \xrightarrow{\tau} \langle \mathbf{tt},\sigma\rangle \quad \langle c_1,\sigma\rangle \xrightarrow{a} \langle c_1',\sigma'\rangle}{\langle \texttt{if}\,(b)\,\texttt{then}\, c_1\,\texttt{else}\, c_2,\sigma\rangle \xrightarrow{a} \langle c_1',\sigma'\rangle} \qquad \frac{\langle b,\sigma\rangle \xrightarrow{\tau} \langle \mathbf{ff},\sigma\rangle \quad \langle c_2,\sigma\rangle \xrightarrow{a} \langle c_2',\sigma'\rangle}{\langle \texttt{if}\,(b)\,\texttt{then}\, c_1\,\texttt{else}\, c_2,\sigma\rangle \xrightarrow{a} \langle c_2',\sigma'\rangle}$$

$$\frac{\langle b,\sigma\rangle \xrightarrow{\tau} \langle \mathbf{ff},\sigma\rangle}{\langle \texttt{while}\,(b)\,\texttt{do}\, c,\sigma\rangle \xrightarrow{\tau} \langle\cdot,\sigma\rangle} \qquad \frac{\langle b,\sigma\rangle \xrightarrow{\tau} \langle \mathbf{tt},\sigma\rangle \quad \langle c,\sigma\rangle \xrightarrow{a} \langle c',\sigma'\rangle}{\langle \texttt{while}\,(b)\,\texttt{do}\, c,\sigma\rangle \xrightarrow{a} \langle c';\texttt{while}\,(b)\,\texttt{do}\, c,\sigma'\rangle}$$

Figure 2: *Operational Semantics of While*

Preliminaries. Partial Equivalence Relations (PERs) have been used to model information [7, 11]. A PER over a set Ω is a symmetric and transitive binary relation. If, in addition, the PER is reflexive over Ω, then it is an equivalence relation over that set. For any given set Ω, we denote the set of all PERs over Ω to be $PER(\Omega)$. Let $R \in PER(\Omega)$ be a PER, the domain of definition of R is given by $dom(R) \triangleq \{\omega \in \Omega \mid \omega\, R\, \omega\}$, and for any $\omega \in dom(R)$, the equivalence class of ω is given by $[\omega]_R \triangleq \{\omega' \in \Omega \mid \omega\, R\, \omega'\}$. We denote by $[\Omega]_R \triangleq \{[\omega]_R \mid \omega \in dom(R)\}$ the set of all equivalence classes of R.

A PER over Ω models information by its ability to distinguish, or not, the elements of the set Ω [12]. Two elements of Ω are said to be indistinguishable (lack of knowledge) by a PER if they are related by that PER, otherwise the PER distinguishes (has knowledge about) them. Let $R,R' \in PER(\Omega)$ be PERs, R' is said to be more informative than R, written $R \sqsubseteq R'$, iff for every $\omega,\omega' \in \Omega$, $\omega\, R'\, \omega' \implies \omega\, R\, \omega'$. The intuition behind $R \sqsubseteq R'$ is that if R' cannot distinguish a pair, neither can R; and thus by the contrapositive, R' distinguishes more than R, making R' more informative. In order to combine the information in two PERs R and R', we define the lattice join operation $\sqcup$ over PERs, such that for all $\omega,\omega' \in \Omega, \omega\,(R \sqcup R)\,\omega'$ iff $\omega\, R\, \omega'$ and $\omega\, R'\, \omega'$. It is clear that $R \sqsubseteq R' \iff R \sqcup R' = R'$. The extension of $\sqcup$ to sets is defined in the usual way, such that for any $\mathcal{R} \subseteq PER(\Omega)$, $\omega \bigsqcup \mathcal{R}\, \omega'$ iff $\forall R \in \mathcal{R}$, $\omega\, R\, \omega'$. For any set Ω, $PER(\Omega)$ is a complete lattice. We also note the general property that the union $R \cup R'$ of disjoint PERs is also a PER.

We define the identity (id) and the all (all) equivalence relations over Ω such that for all $\omega,\omega' \in \Omega$, $\omega\, all\, \omega'$ holds; and $\omega\, id\, \omega'$ holds iff $\omega = \omega'$. For any *While* expression e of type t, where $[\![t]\!]$ is the set of all t-values, and $\phi \in PER([\![t]\!])$, define $e : \phi \in PER(\Sigma)$ to be the PER over program states defined such that $\forall \sigma,\sigma' \in \Sigma, \sigma\, e : \phi\, \sigma'$ iff $\sigma(e)\,\phi\,\sigma'(e)$. Let $rng(R)$ be the range of the relation R, we define the operator $\bullet$, which composes a relation and a PER, and is defined for any PER R over Σ such that $\forall \sigma, \sigma' \in \Sigma, \sigma\ \langle c \rangle \bullet R\ \sigma'$ iff $\exists \sigma_1, \sigma_2 \in rng(\langle c \rangle_{\downarrow})$ s.t. $\sigma\ \langle c \rangle\ \sigma_1 \wedge \sigma'\ \langle c \rangle\ \sigma_2 \wedge \sigma_1\, R\, \sigma_2 \vee (\sigma\ \langle c \rangle\ \infty \wedge \sigma'\ \langle c \rangle\ \infty)$. Since c is deterministic, and hence $\langle c \rangle$ is a function, the relation $\langle c \rangle \bullet R$ is a PER, and it mirrors in the domain of $\langle c \rangle$, the partitioning of the range of $\langle c \rangle$ by R. Additionally, $\langle c \rangle \bullet R$ partitions initial states of c that lead to divergence from those under which c terminates.

The map $\mu : \mathcal{P}(\Omega) \to [0,1]$ is a *probability measure* over Ω if $\mu(\Omega) = 1$, and for any disjoint $X, Y \subseteq \Omega$, $\mu(X \cup Y) = \mu(X) + \mu(Y)$. For singleton events $\{\omega\} \subseteq \Omega$, we write $\mu(\omega)$ instead of $\mu(\{\omega\})$. Given the probability measure μ over the space Ω, the entropy of the space due to μ is given by $\mathcal{H}(\mu) = -\sum_{\omega \in \Omega} \mu(\omega) \log_2(\mu(\omega))$.

3.2 Attacker Models

The information gained by an attacker through a program is determined by what the attacker can observe during the program's execution. We refer to what the attacker can see about a program's execution as the attacker's *observational power*. Therefore, the analysis of secure information release is carried out relative to a specific attacker, modelled by the attacker's observational power. We formalise the attacker's observational power as a rewrite of the labels of the standard transition system of the program to an induced transition system. This allows us to parametrise the static analysis with the specific attacker models, against which the analysis is secure. Let $T = \langle \mathcal{S}, \longrightarrow, \mathcal{A} \rangle$ be the labelled transition system of a program in the concrete semantics, then the observational power of an attacker A over this program induces another transition system $T_A = \langle \mathcal{S}, \longrightarrow_A, \mathcal{A}_A \rangle$, where $\mathcal{A}_A$ is the set of actions that can be observed by A, and $\longrightarrow_A \subseteq \mathcal{S} \times \mathcal{A}_A \times \mathcal{S}$ is the transition relation as seen by A. Typically, $\longrightarrow_A$ is defined as rewrite rules over $\longrightarrow$. As usual, $\mathcal{A}_A^*$ is the Kleene closure of $\mathcal{A}_A$, and we abbreviate by $\xrightarrow{\alpha}_A$, the sequence of transitions $\xrightarrow{a_1}_A \xrightarrow{a_2}_A \ldots$ in T_A, where $\alpha = a_1, a_2, \ldots \in \mathcal{A}_A^*$.

We consider a standard attacker A_S, which is able to observe the output values of *write* statements. This attacker cannot ordinarily observe input actions or the values read during input, which allows us to model input actions (such as read from files), which are not visible to the attacker. However, what the attacker knows about inputs is modelled directly in our information flow definition ($R \Rightarrow R'$, introduced in Section 4). This is reasonable, since the attacker's prior knowledge is external to the program semantics, and is only a parameter to our analysis of information flow. Thus, $\longrightarrow_{A_S}$ rewrites all labels of the transition relation $\longrightarrow$ of the standard operational semantics to τ, except for output labels $\boldsymbol{out}(v)$, which are left unchanged.

3.3 Information Release Typing Rules

We now present the concrete static analysis of *While* programs with respect to an attacker model A. Firstly, we define an equivalence relation $\equiv_c^A$ over states, which models the information released to the attacker A by observing the execution of c as follows: $\forall \sigma, \sigma' \in \Sigma$, $\sigma \equiv_c^A \sigma'$ iff $\forall \alpha, \alpha' \in \mathcal{A}_A^*$

$$\begin{array}{rcl} \exists \langle \cdot, \sigma_1 \rangle \in \mathcal{S} \wedge \langle c, \sigma \rangle \xrightarrow{\alpha}_A \langle \cdot, \sigma_1 \rangle & \Longleftrightarrow & \exists \langle \cdot, \sigma_2 \rangle \in \mathcal{S} \wedge \langle c, \sigma' \rangle \xrightarrow{\alpha}_A \langle \cdot, \sigma_2 \rangle \quad \& \\ \exists \langle c', \sigma_1 \rangle \in \mathcal{S} \wedge \langle c, \sigma \rangle \xrightarrow{\alpha'}_A \langle c', \sigma_1 \rangle & \Longleftrightarrow & \exists \langle c'', \sigma_2 \rangle \in \mathcal{S} \wedge \langle c, \sigma' \rangle \xrightarrow{\alpha'}_A \langle c'', \sigma_2 \rangle \end{array} \tag{1}$$

It is clear that $\equiv_c^A$ relates any pair of states that lead to executions of c, which are observationally equivalent as far as the attacker A can tell, and it captures semantically, the information that A can gain about the initial states of c. The definition of $\equiv_c^A$ is *termination-sensitive*, distinguishing between terminating and non-terminating executions of c. It is easy to see that $\equiv_c^A$ is an equivalence relation over states.

Our static analysis is defined as a type system, parametrised by an attacker model. The typing derivation for a program c, under the attacker model A, captures how A's knowledge changes due to information released by c and is written in the form $\Gamma_A \vdash c : (R_X, R) \Rightarrow (R_Y, R')$. The typing environment Γ_A makes explicit the fact that the analysis is with respect to the attacker model A. The PER R stands for our assumption about A's initial knowledge and the PER R' is the information released to the attacker, which includes the attacker's initial knowledge (that is, $R \sqsubseteq R'$). We use R_X and R_Y to model how c transforms program states, linking the analysis of information flow to the program semantics. We assume $R_X \subseteq \Sigma_\infty \times \Sigma_\infty$ is a function, which maps an initial set of states of interest to the set of states prior to the execution of c. Then, $R_Y \subseteq \Sigma_\infty \times \Sigma_\infty$ is also a function, which maps the initial set of states to the states after the execution of c, and is simply obtained by the composition $R_Y = R_X;\langle c\rangle$. Formally, the type judgement $\Gamma_A \vdash c : (R_X, R) \Rightarrow (R_Y, R')$ is valid if $R_Y = R_X;\langle c\rangle$ and $\forall \sigma, \sigma' \in \Sigma$,

$$\sigma R' \sigma' \implies \sigma R \sigma' \wedge \left(\exists \sigma_1, \sigma_2 \in rng(R_X) : \sigma R_X \sigma_1 \wedge \sigma' R_X \sigma_2 \implies \sigma_1 \equiv_c^A \sigma_2\right). \quad (2)$$

The judgement $\Gamma_A \vdash c : (R_X, R) \Rightarrow (R_Y, R')$ characterises the information released by c to the attacker A, which refines A's knowledge over the initial set of states in $dom(R_X)$. For full program analysis, we will typically take R_X to be the identity relation over Σ. Thus, under the assumption of initial information R that the attacker A might have, A can gain at most the information R' by observing the execution of c which can be computed as $R' = (R_X \bullet \equiv_c^A) \sqcup R$. This semantic definition of information flow ties together the standard program semantics, the attacker's observational power, and the information release. Note that the clause $\sigma R' \sigma' \implies \sigma R \sigma'$ in (2) ensures that the attacker's knowledge is monotonically increasing.

We present the analysis rules for the standard attacker model A_S in Figure 3. The rules also apply to other attacker models, such as A_T (introduced in Section 5), which are simple rewrites of the transition relation under A_S. Since the attacker model is clear, we shall simply write the typing judgement $\Gamma_{A_S} \vdash c : (R_X, R) \Rightarrow (R_Y, R')$ as $c : (R_X, R) \Rightarrow (R_Y, R')$, and $\equiv_c^{A_S}$ as $\equiv_c$.

The analysis rules for *skip*, *assignment* and *read* statements do not change the attacker's knowledge, and hence do not ordinarily release information because the attacker model cannot directly learn anything about the inputs by observing their execution. The rule for *write* statement shows that the attacker gains information about the expression e, by partitioning the input space so that all states in each class evaluates e to an identical value. The composition rule, [COMP], shows how to compose the analysis of sequential statements. The rule [SUB] says that we can safely weaken our assumptions about attacker's prior knowledge and strengthen the result of the analysis of the attacker's final knowledge. The rule for *if* statement combines the information released by the execution of the conditional statement with the attacker's prior knowledge. Finally, the *while* rule, computes a fixed point of the information released by unrolling the *while* statement to an equivalent one-step execution, and applying the rule until a fixed point is reached. Since the analysis rules are to be applied in a concrete static analysis tool, we assume that the set of states Σ considered is finite, so that there exists a unique n for the least fixed point. A more general definition, of the *while* fixed point, which copes with a countably infinite set of

$$\frac{}{\texttt{skip} : (R_X, R) \Rightarrow (R_X, R)} \qquad \frac{}{z := e : (R_X, R) \Rightarrow (R_X ; \langle z := e \rangle, R)}$$

$$\frac{}{\texttt{read}\, x : (R_X, R) \Rightarrow (R_X ; \langle \texttt{read}\, x \rangle, R)} \qquad \frac{}{\texttt{write}\, e : (R_X, R) \Rightarrow (R_X, R \sqcup (R_X \bullet e : id))}$$

$$[\text{COMP}]\frac{c_1 : (R_X, R) \Rightarrow (R_Y, R') \quad c_2 : (R_Y, R') \Rightarrow (R_Z, R'')}{c_1 ; c_2 : (R_X, R) \Rightarrow (R_Z, R'')} \quad \begin{array}{l} R_Y = R_X ; \langle c_1 \rangle \\ R_Z = R_Y ; \langle c_2 \rangle \end{array}$$

$$[\text{SUB}]\frac{c : (R_X, R_1) \Rightarrow (R_Y, R_2) \quad R_0 \sqsubseteq R_1 \quad R_2 \sqsubseteq R_3}{c : (R_X, R_0) \Rightarrow (R_Y, R_3)} \quad R_Y = R_X ; \langle c \rangle$$

$$\frac{c = \texttt{if}\,(b)\,\texttt{then}\, c_1\, \texttt{else}\, c_2}{\texttt{if}\,(b)\,\texttt{then}\, c_1\, \texttt{else}\, c_2 : (R_X, R) \Rightarrow (R_Y, R \sqcup (R_X \bullet \equiv_c))} \quad R_Y = R_X ; \langle c \rangle$$

$$\frac{\begin{array}{c} \texttt{if}\,(b)\,\texttt{then}\, c\, \texttt{else}\, \texttt{skip} : (R_{X_i}, R_i) \Rightarrow (R'_{X_i}, R_{i+1}) \quad R_{X_{i+1}} = R_{X_i} \cup R'_{X_i} \\ R'_{X_n} \triangleq \{(\sigma, \infty), (\sigma_1, \sigma_2) \mid (\sigma, \sigma') \in R_{X_n}, \sigma'(b) = \mathbf{tt} \vee \sigma' = \infty, (\sigma_1, \sigma_2) \in R_{X_n}, \sigma_2(b) = \mathbf{ff}\} \\ \sigma\, R'_n \sigma' \iff (\exists \sigma_1, \sigma_2 \in rng(\langle R'_{X_n} \rangle_{\downarrow}) \wedge \sigma\, R'_{X_n}\, \sigma_1 \wedge \sigma'\, R'_{X_n}\, \sigma_2) \vee (\sigma\, R'_{X_n}\, \infty \wedge \sigma'\, R'_{X_n}\, \infty) \end{array}}{\texttt{while}\,(b)\,\texttt{do}\, c : (R_{X_0}, R_0) \Rightarrow (R'_{X_n}, R_n \sqcup R'_n)} \quad \begin{array}{l} R_{X_n} = R_{X_{n+1}} \\ R_n = R_{n+1} \end{array}$$

Figure 3: *Information Release Typing Rules*

states would be $(\bigcup_{i \in \mathbb{N}} R_{X_i}, \bigsqcup_{i \in \mathbb{N}} R_i)$. At the fixed point, the *while* statement diverges at states that evaluate the guard b to **tt**, hence those states are replaced by the "looping state" ∞, which cannot cause further information flow in subsequent statements. Furthermore, R'_n partitions the initial states between those that lead to divergence of the *while* and those that do not.

Theorem 1 (Correctness) *For any program P, and attacker A_S's initial knowledge R. Let $R_X \subseteq \Sigma_\infty \times \Sigma_\infty$ be a strict total function over the finite extended state space Σ_∞ of P and let $\sigma\, |R_X| \sigma' \iff \sigma, \sigma' \in \{\sigma_1 \in dom(R_X) \mid \exists \sigma_2 \in rng(\langle R_X \rangle_{\downarrow}) \wedge \sigma_1\, R_X\, \sigma_2\}$. Then the type derivation $\Gamma_{A_S} \vdash P : (R_X, R) \Rightarrow (R'_X, R')$ has the following properties*

1. $R'_X = R_X ; \langle P \rangle$,

2. $(R_X \bullet \equiv_P^{A_S}) \sqcup R \sqsubseteq |R_X| \sqcup R'$.

Theorem 1 establishes the correctness of the analysis by expressing a safety property of the analysis. In particular, if we choose R_X to be the identity relation over the state space of P, then we can see that the result of the analysis of information release R', under any assumption of the attacker's prior knowledge is always at least as great as the combination of the attacker's prior knowledge and the actual information $(R_X \bullet \equiv_P^{A_S}) = \equiv_P^{A_S}$ released by the program P. That is, $\equiv_P^{A_S} \sqcup R \sqsubseteq R'$.

4 Information Flow Policies

Our objective is to ensure that a program that requires (legitimate) access to confidential data does not release more information than is intended. We now present a semantic definition of information flow policies, which characterises our intentional information release requirements. Generically, we view information release policies such that, given a lattice of information, the policy sets upper bounds on the information transferred through a program to an observer. Our information release policies fall under the *what* dimension of information declassification, which considers what information is released by a system. Other dimensions of declassification include the *who*, *when*, and *where* dimensions [12].

After information release, the final knowledge of the observer is dependent on the observer's initial knowledge. In this paper, we consider the case where the observer's final knowledge is simply computed by taking the lattice join of the initial knowledge and the information release. Schematically, if $K_i \in \mathcal{I}$ is the initial knowledge of the observer, and $R \in \mathcal{I}$ is the intended information release, both taken from some underlying lattice of information $\mathcal{I}$, then, in this scheme, the final knowledge K_f of the observer is computed simply as $K_f = K_i \sqcup R$. More generally though, we consider a class of information release policies, $\multimapdotbothA$, which are maps from some initial knowledge of the observer to a final one. Since information release causes knowledge to increase, the only requirement in this more general case is that the final knowledge is at least as much as the initial one before receiving additional information. We define such a class of policies below. Because the secrets to be protected are stored in program states during computation, in this paper, our information lattice is defined as partial equivalence relation over states.

Definition 1 (Enforcement of PER-based Information Release Policies) Let Σ be the set of program states and let $\mathcal{I} \triangleq PER(\Sigma)$ be the set of all partial equivalence relations over Σ such that $R, R' \in \mathcal{I}$. An information release policy, $R \multimapdotbothA R'$, is a transformer over the lattice $\mathcal{I}$ such that $R \sqsubseteq R'$.

A program P is said to satisfy the policy $R \multimapdotbothA R'$ (under the attacker model A) if the typing judgement $\Gamma_A \vdash P : (id, R) \Rightarrow (R_Y, R'')$ holds, and we have that $R'' \sqsubseteq R'$.

The information release policy $R \multimapdotbothA R'$ allows the observer to gain *at most* the information R' if the observer has a prior information R_A such that $R \sqsubseteq R_A \sqsubseteq R'$. Intuitively, the requirement $R \sqsubseteq R'$ means that information release policies can only increase the observer's knowledge; although, it may dissalow information gain in case of the noninterference [5] policy: $R \multimapdotbothA R$. As an example, a policy that releases at most the parity of the secret contained in variable x may be defined as: $all \multimapdotbothA \mathrm{Par}_x$, where Par_x is the equivalence relation defined such that $\forall \sigma, \sigma' \in \Sigma, \sigma\, \mathrm{Par}_x\, \sigma' \iff \sigma(x) = \sigma'(x) \mod 2$. This says that if the observer has no prior information (i.e. cannot distinguish any pair of states since $\sigma\, all\, \sigma'$ holds for all states), then the observer is allowed to be able to distinguish at most the parity of x after the release.

The second part of Definition 1 shows how to enforce the information release policy $R \multimapdotbothA R'$. We start by the verification, through static analysis, of the program to determine the level of information that it might release to the observer. The verification is based on the assumption R, of the attacker's initial knowledge. The program is deemed secure if the analysis result R'', which represents the information that the program might release is below R' (the upper bound on information flow allowed by the policy). Because we specified id in $\Gamma_A \vdash P : (id, R) \Rightarrow (R_Y, R'')$, the analysis is carried out over the total state space of P. Because of the ordering property PERs, which means that $R_1 \sqsubseteq R_2 \implies dom(R_2) \subseteq dom(R_1)$, we need not use the identity relation id over Σ, rather, we can restrict this to the identity relation over the actual space of inputs to P; or, in the case that $dom(R)$ is smaller than that input space, over $dom(R)$. This results in some analysis efficiencies.

5 Example: Password Timing Attacks

In this section we consider two password-checking programs, expressed in the *While* language. This example is motivated by potential timing attacks that may be mounted against versions of the OpenSSH with PAM (Pluggable Authentication Module) support distributed with various Linux operating systems. In summary, depending on the behaviour (timing delay) of the authentication module on invalid username-password combinations, the attacker may be able to infer further information on whether a user exists or not, in addition to whether the supplied password matches the valid user's password or not. Note that the timing delays are usually added to failed authentication steps to reduce the effectiveness of dictionary attacks, however, wrongly-implemented, can be exploited as we demonstrate in the following analyses.

The standard observational model A_S cannot observe time delays in program execution. Hence, for the examples below, we shall introduce an attacker model, A_T, which can observe the passage of time, or, more precisely, can model various delays during program execution by counting the number of primitive commands executed. It can also observe *read* prompts, when the program accepts either the username or password. The attacker model A_T extends the standard observational capability of A_S by introducing a capability to count the number of primitive commands executed. The transition relation $\longrightarrow_{A_T}$ as seen by the attacker A_T is defined, for any of the small-step command-configuration transition in $\longrightarrow_{A_S}$, as

$$\frac{\langle c,\sigma\rangle \xrightarrow{a}_{A_S} \langle c',\sigma'\rangle}{\langle c,\sigma\rangle \xrightarrow{\langle a,t+1\rangle}_{A_T} \langle c',\sigma'\rangle}[t] \qquad \frac{\langle c,\sigma\rangle \xrightarrow{a}_{A_S} \langle \cdot,\sigma'\rangle}{\langle c,\sigma\rangle \xrightarrow{\langle a,t+1\rangle}_{A_T} \langle \cdot,\sigma'\rangle}[t] \qquad \frac{\langle \mathtt{read}\,x,\sigma\rangle \xrightarrow{a}_{A_S} \langle \cdot,\sigma'\rangle}{\langle \mathtt{read}\,x,\sigma\rangle \xrightarrow{\langle \boldsymbol{in},t+1\rangle}_{A_T} \langle \cdot,\sigma'\rangle}[t] \quad (3)$$

Thus, if the program makes a small step transition in the standard semantics at the "time" t, the attacker observes the increment of counter t by 1, in addition to the action a performed in the standard semantics. This capability constitutes the basis of the timing attacks demonstrated below.

In a password authentication program we *have to* release the information that a user with the correct password is valid (the case when authentication succeeds) and that a valid user with the wrong password as well as invalid users regardless of the password are invalid (the case when authentication fails). What we do not want to do is to further distinguish the cases between a valid user with wrong password and a non-existent user. The intended information release policy can be formalised as follows.

Let $\mathcal{U}$ be the set of all possible users, regardless of whether they exist or not on the target system, and let $U \subseteq \mathcal{U}$ be the valid ones, that exist on the target system. For each valid user $u \in U$, let p_u be the user's password. Similarly, let $\mathcal{P}$ be the set of all possible passwords, of which a subset of it is the set of legitimate users' passwords. The set of valid users, with valid passwords is thus $V = \{(u, p_u) \mid u \in U, p_u \in \mathcal{P}\}$. Hence, we can define our username-password state space to be $\Sigma = \mathcal{U} \times \mathcal{P}$, and the equivalence relation which models precisely the information we intend to release is R_v where $\forall (u,p),(u',p') \in \Sigma, (u,p) R_v (u',p') \iff (u,p),(u',p') \in V \vee (u,p),(u',p') \in \Sigma \backslash V$. This equivalence relation only distinguishes legitimate users with correct passwords from the rest of the world, and no more. Hence, our intended information release policy would be $all \twoheadrightarrow R_v$, which allows the observer, which has no prior information, to gain the information R_v (as required by a genuine authentication system).

Now consider the program on the left-hand-side of Figure 4. This program accepts both the username and password at the beginning, and then outputs the value 1 on a successful authorisation, or:

- either produces a time delay[1] of `na` units and outputs the value 2 to indicate an unsuccessful attempt (the case of an invalid password),

- or produces a time delay of `nb` units and outputs the value 2 to indicate an unsuccessful attempt (the case of an invalid username).

Clearly the values of `na` and `nb` are significant: if `na=nb` then an attacker observing time delays will not be able to distinguish whether a delay has been caused by an incorrect username or an incorrect password. If the attacker only observes program output he or she will not be able to distinguish these cases, since both write out the same "error message" value of 2. Our static analysis, under the attacker model A_T, is able to distinguish between the case when `na=nb`, and the case when `na≠nb`. Let the program be P_A, which sets `na=nb`. Then applying the analysis rule to P_A, under the attacker model A_T, gives $\Gamma_{A_T} \vdash P_A : (id, all) \Rightarrow (R_Y, R_v)$ on the one hand[2]. On the other hand, now consider another implementation P_B where `na≠nb`. We obtain the type analysis $\Gamma_{A_T} \vdash P_B : (id, all) \Rightarrow (R_Y, R'_v)$, where $\forall (u,p),(u',p') \in \Sigma, (u,p) R'_v (u',p') \iff (u,p),(u',p') \in V, (u,p),(u',p') \in V', (u,p),(u',p') \in \Sigma \backslash (V \cup V')$, and where $V' = (U \times \mathcal{P}) \backslash V$ is the set of valid users with invalid passwords. Clearly, since $R'_v \not\sqsubseteq R_v$,(in fact, $R_v \sqsubset R'_v$), P_B does not satisfy our information release policy, and should therefore be rejected. Generally, the evaluation of expressions is not observable, and therefore the evaluation of the *member* and *valid* expressions conditions is not visible to the attacker in these examples.

The password checking program on the right hand side of Figure 4 is differently structured to that of the left, in that it first accepts only the username at the start and directly checks whether it is valid or not, producing a delay `nb` in the latter case before reporting a failure. However, the fact that the user is prompted to enter the password in the case that the username exists, and is not in the case that the user does not exist already reveals information on the existence or not of the specified user, even without further interaction. The static analysis of this program P_C is $\Gamma_{A_T} \vdash P_C : (id, all) \Rightarrow (R_Y, R'_v)$, where $\forall (u,p),(u',p') \in \Sigma, (u,p) R'_v (u',p') \iff (u,p),(u',p') \in V, (u,p),(u',p') \in V', (u,p),(u',p') \in \Sigma \backslash (V \cup V')$. This is exactly the same information released by the program P_B where the fact that `na` differs from `nb` helps the attacker to distinguish the

[1] The **delay** `n` function may be implemented in the *While* language by looping over a skip statement `n` times, which can be differentiated by the attacker model A_T for different values of `n`.

[2] The relation R_Y maps all initial states to a final state that contains the supplied username and password values.

```
read user;                        read user;
read pw;                          if (member(user,U)) then
if (member(user,U)) then            read pw;
  if (valid(user,pw)) then          if (valid(user,pw)) then
    write 1                           write 1
  else                              else
    delay na                          delay na
    write 2                           write 2
else                              else
  delay nb                          delay nb
  write 2                           write 2
```

Figure 4: *Password-checking programs (Version 1 on the left, and Version 2 on the right).*

case between non-existent user and a valid user with invalid password. However, in the case of P_C, the same information is released regardless of the equality or not of `na` and `nb`.

The lesson learned from these analyses is that even non-malicious program can contain subtle bugs or design flaws which violate information security policies and must therefore be checked against such unintended information leakage. However, our approach is perfect for the malicious code scenario, where apart from the possibility of unintentional information leakage, malicious information release can be detected through static verification of programs. In the case of the program on the left hand side of Figure 4, the implementation is correct, but the configuration (i.e. how the values of `na` and `nb` are set relative to each other) can expose the timing flaw, which our analysis detects. However, in the case of the right hand side program, it is an implementation or design flaw to check the existence of the user before proceeding to prompt for a password.

6 Quantifying the Information Release

The analysis presented in this paper shows that deterministic programs may be viewed as agents that release information by partitioning their input domains. Policies are then controls, which specify to what extent a program is allowed to partition this domain. When the analysis is furnished with a probability measure, which represents the attacker's uncertainty over the input space, our qualitative PER-based policy specification actually dictates the maximum quantitative information, in an information-theoretic [13] sense, that the program in question is allowed to release. Because of the determinism, any sort of probability distribution observed in the output behaviour of the program is induced by the probability distribution of the input space, and hence any reduction in the uncertainty of the attacker over the entropy of the input space obtained by observing program execution is, in fact, as a result of the refinement of the partitioning of the input space caused by the program. Hence, when we specify the policy $R \twoheadrightarrow R'$, where $R \sqsubseteq R'$ are equivalence relations, we are effectively also specifying an upper bound on the quantitative information that we allow to be released. Specifically, given a probability distribution μ over the input space Σ, the equivalence relation R over Σ describes what the attacker is assumed to know before the execution of the program, and the qualitative information R can be quantified as $\mathcal{H}(\mu|R)$ (see Definition 2), which measures the entropy of the input space under the distribution μ, subject to the partitioning of the input space by R. Similarly, under the policy $R \twoheadrightarrow R'$, where we allow the attacker to refine its knowledge about the input space from R up to a maximum of R', the policy effectively specifies the minimum entropy $\mathcal{H}(\mu|R')$ over the input space that the attacker is allowed to reach. Thus, under any given probability distribution of the input space, the policy $R \twoheadrightarrow R'$ specifies an upper bound on the quantitative information that we allow to be released: this is a maximum allowable reduction in entropy $\mathcal{H}(\mu|R \twoheadrightarrow R')$ given below.

Definition 2 (Quantifying Information Release) Let μ be a probability measure over the set Σ and let $R, R' \in PER(\Sigma)$ be equivalence relations over Σ such that $R \sqsubseteq R'$. Define the entropy of the space Σ, under μ, subject to the partitioning of R as

$$\mathcal{H}(\mu|R) = \mathcal{H}(\mu) - \sum_{X \in [\Sigma]_R} \mu(X)\log_2(\mu(X)).$$

Define the entropy reduction over the space Σ under $R \twoheadrightarrow R'$ as

$$\mathcal{H}(\mu|R \twoheadrightarrow R') = \mathcal{H}(\mu|R) - \mathcal{H}(\mu|R').$$

The definition of $\mathcal{H}(\mu|R)$ takes away from the entropy of μ, the entropy of the space of the equivalence classes of R. Since Σ is assumed finite, recall that by the *finite additivity* property of μ, we may compute $\mu(X)$, for any equivalence class $X \in [\Sigma]_R$ of R, as $\mu(X) = \sum_{\sigma \in X} \mu(\sigma)$. Now, the definition $\mathcal{H}(\mu|R \twoheadrightarrow R')$ of quantitative information release is reasonable. For example, the policy $all \twoheadrightarrow id$, which on the one hand allows the attacker to gain all information about the input space quantitatively removes all the uncertainty of the attacker, because for any initial uncertainty as modelled by the measure μ over the input space $\mathcal{H}(\mu|all \twoheadrightarrow id) = \mathcal{H}(\mu)$. On the other hand, the *non-interference* [5] policy, $all \twoheadrightarrow all$, which prevents the attacker from refining its knowledge through information release has the property that $\mathcal{H}(\mu|all \twoheadrightarrow all) = 0$.

For the password authentication example of Section 5, the desired quantitative information release under the assumption of the attacker's initial probability distribution μ is $\mathcal{H}(\mu|all \twoheadrightarrow R_v)$. Like under the qualitative PER-based policy, the programs P_B and P_C do not satisfy the quantitative information release requirement either under any assumption of μ. This is be because $R_v \sqsubset R'_v$ and $R_v \sqsubset R''_v$ and we can easily show that for any μ, and PERs R_A and R_B, $R_A \sqsubseteq R_B \implies \mathcal{H}(\mu|R_B) \leq \mathcal{H}(\mu|R)$. However, because the reverse implication does not necessarily hold, this can lead to a false sense of security. In particular, because the entropy measure only uses the probability distributions, and not the space itself, the values may not reflect which element has become more likely as a result of information release. This is clear because, for example, permutation of probability measures over the space will leave the entropy measure unaffected. To have more control over *about what elements* of the input space information is released, we advocate using qualitative policies, rather than only quantitative ones. Let us illustrate this with a final example.

Now consider the following four programs, which processes the input parameter $h \in \{0,1,2,3\}$, which is a secret:

- $P_1 \triangleq \texttt{write}\, h - h$

- $P_2 \triangleq \texttt{write}\, h \mod 2$
- $P_3 \triangleq \texttt{if}\,(h \leq 1)\,\texttt{then}\,\texttt{write}\,1\,\texttt{else}\,\texttt{write}\,2$
- $P_4 \triangleq \texttt{write}\, h$.

Let us model the state space of these programs by $H = \{0,1,2,3\}$, where $n \in H$ models the state where the value of variable h is n. Our analysis shows the following results: $\Gamma_{A_S} \vdash P_1 : (id, all) \Rightarrow (R_Y, all)$, $\Gamma_{A_S} \vdash P_2 : (id, all) \Rightarrow (R_Y, \kappa)$, $\Gamma_{A_S} \vdash P_3 : (id, all) \Rightarrow (R_Y, \kappa')$, $\Gamma_{A_S} \vdash P_4 : (id, all) \Rightarrow (R_Y, id)$, as depicted in Figure 5, where R_Y maps all states to the input value of h. The equivalence relations are defined as follows: $\forall h, h' \in H$, $h\, \kappa\, h'$ iff $h = h' \mod 2$, $h\, \kappa'\, h'$ iff $h, h' \in \{0,1\}$ or $h, h' \in \{2,3\}$. The arrows in Figure 5 describe how the respective programs transform the partition of their domains. For example, the arrow labelled P_4 shows that given the initial knowledge represented by *all*, the attacker's final knowledge is modelled by *id*. By following the arrows labelled P_2 and P_3, we obtain the transformation of the attackers knowledge from *all* via κ to *id*, which can be obtained by running the composed program $P_2; P_3$. Not all possible arrows are shown.

Now suppose that we wish to release, at most, the parity of the secret h. The desired qualitative policy would be $all \twoheadrightarrow \kappa$, which releases the parity of h. Clearly P_1 and P_2 satisfy this policy, but P_3 and P_4 do not because $\kappa' \not\sqsubseteq \kappa$ and $id \not\sqsubseteq \kappa$. Now, let us take a uniform probability measure μ over h, such that $\forall h \in H, \mu(h) = \frac{1}{4}$. The desired quantitative information release is $\mathcal{H}(\mu | all \twoheadrightarrow \kappa) = 1$: which allows 1-bit information over the space H to be released, since we are effectively halving the uncertainty over the whole space, which is 2 bits. So, quantitatively, we have for P_1 and P_2, the information release $\mathcal{H}(\mu | all \twoheadrightarrow all) = 0$ and $\mathcal{H}(\mu | all \twoheadrightarrow \kappa) = 1$, which satisfies our requirement as usual. However, for P_3, we have also that $\mathcal{H}(\mu | all \twoheadrightarrow \kappa') = 1$, which satisfies our quantitative release requirement, but releases information other than the parity of h. This is a class of the probability permutation problem, where elements with the same probabilities in different equivalence classes are swapped between the equivalence classes of the PER. This leads to a different PER but leaves the entropy measure intact.

7 Conclusions and Future Work

In this paper we have presented a general static analysis technique for the verification of information release by programs written in a simple imperative language. The analysis technique is defined parametric to an attackers' observational power. By using various observational powers, we can move the analysis from the standard semantics to other non-standard semantics, allowing us to model aspects of the system that may be implicit in the design, or that are specific to certain implementation environments, for example, multi-user environments where a program may be interacted with locally or across a network - with different behaviours. To illustrate the use of various attacker's observational power model, we presented an attacker which can count instruction execution, allowing this attacker to mount a "timing" attack on the system.

We have demonstrated the value of such an attacker model through the analysis of password checking programs, which are inspired by the corresponding code in the OpenSSH with PAM implementation found in various UNIX systems, including versions of OpenBSD, and Linux.

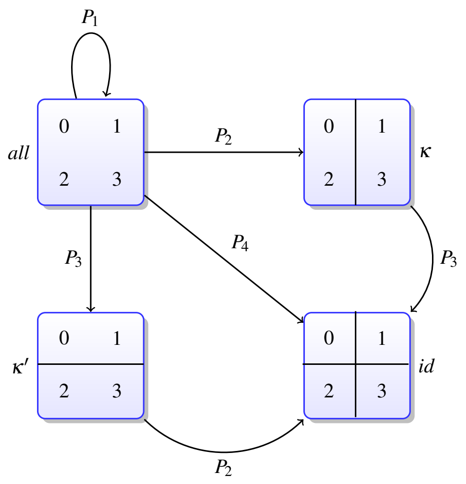

Figure 5: *How programs transform partitions of secret domain* $H = \{0,1,2,3\}$.

The work presented here forms the basis of a wider programme to analyse information release of operating system-level programs and code for mobile devices. Building on the ideas presented here, we expect to be able to implement static code checking against information release policies: the first step will be to incorporate the rules for static analysis into a type checker for *While*-like programs. Extensions of the language will be considered, including constructs for procedure invocation, object-oriented programming, and other features. If the verification of programs against information release policies is to be done at operating system level (with the type checker implemented as a kernel module, for example), then the analysis of binary executables may be necessary, since the original source code may not be to hand. In this case, the static analysis needs to be adapted for low-level language constructs. For mobile devices it may be sufficient to apply the analysis to the instruction set of a virtual machine such as the JVM or the Dalvik executable format of the Android platform, and this is a direction for further investigation. We find the approach of [4] very interesting. There is significant interest in the Android platform for mobile devices, and there is an opportunity to study information release in this setting.

Finally, in this paper we have considered specific attacker models (A_S and A_T) - we hope to be able to perform the analysis for attackers with different observational capabilities. While for the password-checking problem it seems our model is sufficient, there are other possibilities to consider, and we hope to do so in the light of further examples and case studies.

Bibliography

[1] Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Proc. 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer-Verlag, 2005.

[2] A. O. Adetoye and Atta Badii. A policy model for secure information flow. In Pierpaolo

Degano and Luca Viganò, editors, *ARSPA-WITS*, volume 5511 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.

[3] A. Bouajjani, Jean-Claude Fernandez, and Nicholas Halbwachs. Minimal model generation. In Edmund M. Clarke and Robert P. Kurshan, editors, *Proceedings of Computer-Aided Verification (CAV '90)*, volume 531 of *LNCS*, pages 197–203, Berlin, Germany, June 1991. Springer.

[4] Avik Chaudhuri. Language-based security on android. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 1–7, New York, NY, USA, 2009. ACM.

[5] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 11–20, Oakland, CA, April 1982. IEEE Computer Society Press.

[6] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1-3):113–138, 2000.

[7] J. Landauer and T. Redmond. A lattice of information. In *Proceedings of the Computer Security Foundations Workshop VI (CSFW '93)*, pages 65–70, Washington - Brussels - Tokyo, June 1993. IEEE.

[8] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talx86: A realistic typed assembly language. In *In Second Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[9] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[10] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[11] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001.

[12] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 2007.

[13] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[14] Dachuan Yu and Nayeem Islam. A typed assembly language for confidentiality. In Peter Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 162–179. Springer, 2006.