Electronic Communications of the EASST
Volume ?? (2010)

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

18 pages

Guest Editors:
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

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 use of static analysis 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 taken from a known flaw in versions of OpenSSH distributed with the FreeBSD operating system.

Keywords:

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, will typically 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 quantifying the information released by a program, while taking into account its purpose and visible 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 use concepts from information theory to develop a measure of the information release of a program. We use static analysis to compute the value of this measure for a given program. 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 any 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 usernames and passwords. The examples are inspired by password checking programs used in different versions of OpenSSH on the FreeBSD operating system.

Contributions. This paper contributes to the theory of quantitative information flow analysis and is inspired by work on secure information flow (see e.g. [**?**]).

As mentioned before, we propose a measure of information release for programs, which takes into account program functionality as viewed by a particular attacker. In technical terms, functionality is described by the program's input/output model, or *functional model*. A given attacker will have a limited view of a program's functionality, depending on his or her access to particular resources, including various clients, servers and channels; we refer to this view as the *attacker model*. An attacker can observe the behaviour of a program for different inputs and draw various conclusions about the way it is implemented. More specifically, repeated observation of a program for different inputs will allow the attacker to deduce a *partition* on the space of program inputs.

We show that information release is a property of this partition and so establish an important theoretical link between quantitative analysis and qualitative analysis of information flow. Furthermore, we establish a general model, which can be adapted to different attacker types; the functional model of a program can be extracted directly from its operational semantics.

We present a flow-sensitive and termination-sensitive static analysis, which quantifies the amount of information released by a deterministic program with loops and outputs under a specific attacker model. We prove the correctness of the analysis. To the best of our knowledge, this paper is the first to present a static analysis that comprehensively accounts for the quantitative information flow in programs with output interaction in the presence of program divergence.

Preliminaries. A partial equivalence relation (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 $PER(\Omega)$. Similarly, $ER(\Omega)$ denotes the set of all equivalence relations over Ω. Let $R \in PER(\Omega)$ be a PER, the domain of definition of R is given by $dom(R) \triangleq \{\omega \in \Omega \,|\, \omega R \omega\}$, and for any $\omega \in dom(R)$, the equivalence class of ω is given by $[\omega]_R \triangleq \{\omega' \in \Omega \,|\, \omega R \omega'\}$. We denote by $[\Omega]_R \triangleq \{[\omega]_R \,|\, \omega \in dom(R)\}$ the set of all equivalence classes of R. The kernel of a function $f : \Omega \to \mathcal{V}$ is an equivalence relation over Ω, which relates every pair of elements $\omega, \omega' \in \Omega$ iff $f(\omega) = f(\omega')$.

A PER over Ω models information by its ability to distinguish, or not, the elements of the set Ω. 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 $\omega, \omega' \in \Omega$, $\sigma R' \sigma' \implies \sigma R \sigma'$. The intuition behind $R \sqsubseteq R'$ is that if R' cannot distinguish a pair, neither can R, by the contrapositive, R' distiguishes more than R and is thus more informative. The combination of information in R and R' is achieved via the operation $\sqcup$, where 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 $\forall R \in \mathcal{R}$, $\omega R \omega'$. It is clear that $PER(\Omega)$ is partially ordered by $\sqsubseteq$ and that $\sqcup$ is the corresponding *least upper bound* or *join* operation.

The operation $\sqcup$ does not preserve the domain of PERs, hence we define a domain preserving version, $\uplus$, as follows. Let $R \in PER(\Omega)$, and let $D \subseteq \Omega$. Define a domain completion operation on R, relative to D, to be $\mathcal{C}_D(R)$ such that $\forall \omega, \omega' \in \Omega$, $\omega\, \mathcal{C}_D(R)\, \omega'$ iff $\omega R \omega'$ or $\omega, \omega' \in dom(R) \backslash D$.

It is easy to see that $\mathcal{C}_D(R) \in PER(\Omega)$ since it the union of two disjoint PERs. Now let $R_1, R_2 \in PER(\Omega)$, and let $D = dom(R_1) \cup dom(R_2)$. Define the domain preserving join operation as $R \uplus R' \triangleq \mathcal{C}_D(R) \sqcup \mathcal{C}_D(R')$. The extension of $\uplus$ to sets is defined such that for any $\mathcal{R} \subseteq PER(\Omega)$, and $D = \bigcup_{R \in \mathcal{R}} dom(R)$, $\biguplus \mathcal{R} \triangleq \bigsqcup_{R \in \mathcal{R}} \mathcal{C}_D(R)$. Define the partial order relation, $\sqsubseteq$, relative to $\uplus$, on $PER(\Omega)$ so that for any $R, R' \in PER(\Omega)$, $R \sqsubseteq R' \iff R \uplus R' = R'$.

Lemma 1 *For any set Ω, the partially ordered set $\langle PER(\Omega), \sqsubseteq, \uplus \rangle$ is a complete lattice.*

Let Ω be a set. The map $\mu : \mathcal{P}(\Omega) \to [0,1]$ to the closed real interval $[0,1]$ is a *probability measure* over Ω if $\mu(\Omega) = 1$, and for any disjoint $X, Y \subseteq \Omega$, $\mu(X \cup Y) = \mu(X) + \mu(Y)$. If μ is a probability measure over Ω, and $X \subseteq \Omega$, then $\mu|X$ is a conditional probability measure (conditioned on X), where for any $Y \subseteq \Omega$, $\mu(Y|X) = \mu|X(Y) = \mu(Y \cap X)/\mu(X)$. A set $Y \subseteq \Omega$ is called an *event*. For singleton events, we shall write $\mu(\omega)$ instead of $\mu(\{\omega\})$, and $\mu(\omega|X)$ instead of $\mu(\{\omega\}|X)$ for the conditioned counterparts.

Let μ be a probability measure over Ω, the *entropy* due to this measure is defined as $\mathcal{H}(\mu) \triangleq -\sum_{\omega \in \Omega} \mu(\omega) \log \mu(\omega)$. Similarly, for a conditional probability measure $\mu|X$, the entropy is given by $\mathcal{H}(\mu|X) \triangleq -\sum_{\omega \in \Omega} \mu(\omega|X) \log \mu(\omega|X)$. Let $R \in PER(\Omega)$ be a PER. We define the family of conditional probability measures induced by R over μ by $\mu|R = (\mu|X)_{X \in [\Omega]_R}$. The entropy of $\mu|R$ is given by $\mathcal{H}(\mu|R) = \sum_{X \in [\Omega]_R} \mu(X) \mathcal{H}(\mu|X)$.

1.1 Syntax and Semantics of the *While* Language.

In this section we present the core imperative language, *While*, which has loops and input-output interaction. The syntax (Figure 1) and the operational semantics (Figure 2) of *While* 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, $\blacksquare$, are maps from variables, in **Var**, to values. The evaluation of the expression e at the state $\sigma \in \blacksquare$ is summarised as $\sigma(e)$. Expression evaluations are performed atomically and have no side-effect on state. The set of free variables of the expression e is denoted by $FV(e)$. The projection of the state σ to $Z \subseteq \mathbf{Var}$ is denoted by $\sigma_{\downarrow Z}$. A program action, ranged over by a, can either be an internal action τ, which is not observable ordinarily; or it can be an output action (via a *write* command), where the expression value can be observed. 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 trace of a *While* program P, starting from the state $\sigma \in \blacksquare$, is denoted by $\langle P, \sigma \rangle \xrightarrow{a_0} \langle P_1, \sigma_1 \rangle \xrightarrow{a_1} \cdots$, according to the operational semantics. The trace of P at the state σ is said to *terminate* if there is a natural number n such that $\langle P, \sigma \rangle \xrightarrow{a_0} \cdots \xrightarrow{a_{n-1}} \langle \cdot, \sigma' \rangle$, written also as $\langle P, \sigma \rangle \Downarrow \sigma'$; otherwise, the

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

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

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

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

Figure 2: *Operational Semantics of* While

trace is said to be *nonterminating*, and P *diverges* at σ.

1.2 Attacker Models

The information gained by an attacker through a program is determined by what the attacker can observe during the program's execution. Hence the analysis of secure information release is carried out with specific attackers in mind. We formalise the attacker's observational power as a rewrite of the labels of the standard transistion system of the program to an induced transistion system. This allows us to parametrise the static analysis with the specific attacker model 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 abreviate by $\xrightarrow{\alpha}_A$, the sequence of transitions $\xrightarrow{a_1}_A \xrightarrow{a_2}_A \ldots$ in T_A, where $\alpha = a_1, a_2, \ldots \in \mathcal{A}_A^*$.

To illustrate the attack model, we introduce an attacker A, which, in addition to its ability to observe all external action in the standard semantics, is also able to observe the passage of time by counting the number of the primitive program commands executed. The transition relation $\longrightarrow_A$ as seen by this attacker is defined, for any of the small-step command-configuration transition in $\longrightarrow$, as

$$\frac{\langle c, \sigma\rangle \xrightarrow{a} \langle c', \sigma'\rangle}{\langle c, \sigma\rangle \xrightarrow{\langle a, t+1\rangle}_A \langle c', \sigma'\rangle}\,[t] \qquad \frac{\langle c, \sigma\rangle \xrightarrow{a} \langle \cdot, \sigma'\rangle}{\langle c, \sigma\rangle \xrightarrow{\langle a, t+1\rangle}_A \langle c', \sigma'\rangle}\,[t] \tag{1}$$

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.

1.3 Modelling and Analysis Framework

We envisage an automated software platform for computing the information release of arbitrary *While* programs, and for comparing this quantity with a specification of information, given by a policy. The formal definition of information release is given in Section 3, while the means of computing information release for arbitrary programs through static analysis is given in Section 4; Information release policies are described in Section 5.

INSERT DIAGRAM

2 Quantifying Information Release

How should we quantify information flow in a program? To motivate our approach, we shall start by analysing a few simple programs to illustrate the basic idea. The analysis is performed in the context of a simple imperative *While* language, whose full syntax and semantics are presented in Section 1.1. The *write* construct in the language is for program *output*, such that for a given expression e, the statement `write` e prints the result of evaluating e to the program output. The attacker is assumed able to observe the output value. The precise definition of what the attacker observes under a given *attack model* is presented in Section **??**.

Suppose that the secret $h \in \{0,1,2,3\}$ is a parameter to the following four programs:

- $P_1 \triangleq \texttt{write}\, h - h$
- $P_2 \triangleq \texttt{write}\, h \mod 2$
- $P_3 \triangleq \texttt{if}\ (h \leq 1)\ \texttt{then}\, \texttt{write}\ 1\ \texttt{else}\, \texttt{write}\ 2$
- $P_4 \triangleq \texttt{write}\, h$.

It is intuitively clear that P_1 releases the least (no) information about h, whereas P_4 releases the greatest (all) information about the secret h. Can we capture this intuition in a quantitative sense? Firstly, being deterministic, we can model these programs as *functions* from the input space $H = \{0,1,2,3\}$ to an output space, which we can also take to be H in this example. So, P_1 induces a constant function, f_1 such that for all $h \in H$, $f_1(h) = 0$, whose kernel is given by κ_1, where for all $h,h' \in H$, $h\,\kappa_1\,h'$. Similarly, P_2 induces a function whose kernel is κ_2, defined such that $h\,\kappa_2\,h'$ iff $h = h' \mod 2$. For P_3, the kernel κ_3 of the induced function is defined such that $h\,\kappa_3\,h'$ iff $h,h' \in \{0,1\}$ or $h,h' \in \{2,3\}$. Finally, for P_4, the kernel of the induced funtion is κ_4, where $h\,\kappa_4\,h'$ iff $h = h'$. Figure 3 shows the partitioning of H by the kernels k_i, and how the programs P_i transform these partitions. For example, the arrow labelled P_4 shows that given the initial knowledge represented by κ_1, the attacker's final knowledge is modelled by κ_4. By following the arrows labelled P_2 and P_3, we obtain the transformation of the attackers knowledge from κ_1 via κ_2 to κ_4, which can be obtained by running the composed program $P_2;P_3$.

The information released by these programs can be described qualitatively as the equivalence relations κ_i, such that based on the output of the program P_i, the observer cannot distinguish between between a pair of inputs $h,h' \in H$ if they are related by κ_i. Thus, qualitatively, P_1 releases the least information because κ_1 is the least equivalence relation over H, relating (and therefore unable to distinguish) any pair of values in H. At the other extreme, P_4 releases the greatest

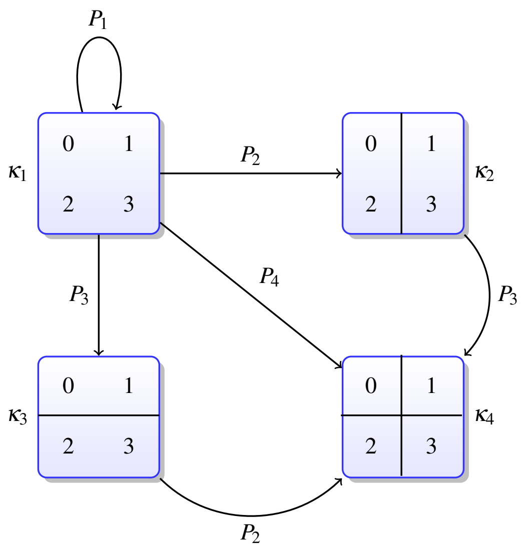

Figure 3: *How programs transform partitions of secret domain* $H = \{0,1,2,3\}$.

information because the equivalence relation κ_4 relates any value in H to itself only, and hence can distinguish every pair of different values in H. This partitioning behaviour has information-theoretic interpretation. Quantitatively, each partition of κ_i induces a conditional probability measure, the entropy of which charaterises the quantitative information gained - the uncertainty that remains by knowing that the secret lies within that partition. By taking the weighted measure of these entropies, indexed by the partitions of κ_i, we obtain the average uncertainty that remains after executing the program. The difference between the uncertainty before executing the program and the uncertainty afterwards gives us a measure of the quantitative information flow. As will be shown next, this explains the intuition about information flows of the programs P_i.

Let us assume that the attacker starts with only the knowledge that the secret lies within the set H, and suppose that the input value is chosen with uniform probability measure, μ, over H, so that for all $h \in H$, $\mu(h) = \frac{1}{4}$. In the case of P_1, the attacker's knowledge is unchanged, as demonstrated by κ_1, which relates all values in H. Consequently, κ_1 has a single partition H, which induces a probability measure $\mu(\cdot\,|\,H)$ (also written as $\mu|H$) conditioned on H, which is the same as the unconditioned measure $\mu(\cdot)$. Since, for all $h \in H$, $\mu(h\,|\,H) = \mu(h)$, the difference in entropy $\mathcal{H}(\mu) - \mathcal{H}(\mu|H) = 0$ explains the lack of information flow. For program P_2, there are two partitions $H_1^{\kappa_2} = \{1,3\}$ and $H_2^{\kappa_2} = \{0,2\}$ of κ_2. These induce the conditional probability measures $\mu|H_1^{\kappa_2}$ and $\mu|H_2^{\kappa_2}$, where $\mu(1\,|\,H_1^{\kappa_2}) = \mu(3\,|\,H_1^{\kappa_2}) = \frac{1}{2}$ and $\mu(0\,|\,H_1^{\kappa_2}) = \mu(2\,|\,H_1^{\kappa_2}) = 0$, and $\mu(1\,|\,H_2^{\kappa_2}) = \mu(3\,|\,H_2^{\kappa_2}) = 0$ and $\mu(0\,|\,H_2^{\kappa_2}) = \mu(2\,|\,H_2^{\kappa_2}) = \frac{1}{2}$. But, $\mu(H_1^{\kappa_2}) = \mu(H_1^{\kappa_2}) = \frac{1}{2}$, hence the average uncertainty that remains over H due to the execution of P_2 is $E_{\kappa_2} = \mu(H_1^{\kappa_2}) \times \mathcal{H}(\mu|H_1^{\kappa_2}) + \mu(H_2^{\kappa_2}) \times \mathcal{H}(\mu|H_2^{\kappa_2}) = 1$. Thus, the information released is $\mathcal{H}(\mu) - E_{\kappa_2} = 1$. Since, μ is uniform, the program P_2 halves the uncertainty remaining over H by revealing the parity of the 2-bit information space. Similarly, P_3 reveals one bit of information by halving the uncertainty, so that the attacker after observing the output knows whether the secret is less than or equal to 1, or not. Finally, for P_4, we have the set of partitions $H_i = \{i\}$, for $i = 0,1,2,3$. Thus for each i, we have $\mu|H_i$, where for all $h \in H$, $\mu(h\,|\,H_i) = 1$ if $h = i$ and $\mu(h\,|\,H_i) = 0$ otherwise. Since for any i, $\mathcal{H}(\mu|H_i) = 0$, the information released by P_4 is given by $\mathcal{H}(\mu) - \sum_i \mu(H_i) \times \mathcal{H}(\mu|H_i) = \mathcal{H}(\mu) = 2$. This result confirms the intuition that P_4 releases all information about the secret input.

From the preceeding we can observe informally that any two programs, which induce the same partitions on the set of inputs, will release the same quantitative information under a given assumption about the attacker's initial uncertainty. Secondly, although an the initial probability measure (the attacker's uncertainty) is required in the calculation of the information flow, the actual information flow is a property of the program, specifically, how it partitions its domain, and the particular choice of initial probability measure is merely a scaling factor, and the particular choice will not make the program less or more secure. These intuitions are formalised in the following theorem.

Theorem 1 *Let μ and μ' be probability measures over Ω, and let $R,R' \in PER(\Omega)$ be PERs over Ω. Then*

1. $R \sqsubseteq R' \implies \mathcal{H}(\mu|R') \leq \mathcal{H}(\mu|R)$.
2. $\mathcal{H}(\mu|R) = \dfrac{\mathcal{H}(\mu)}{\mathcal{H}(\mu')} \times \mathcal{H}(\mu'|R)$.

The first aspect of Theorem 1 establishes an important link between qualitative PER-based analysis of information flow, and quantitative characterisation of information release. The implication is that in the deterministic case, quantitative information release can be reduced to a PER-based analysis of information flow, which allows us to carry out the analysis parametrised by the input distribution. In practice, this means that we do not need the input distribution beforehand to analyse the program or system in question, and we can even check whether the implementation is safe according to a quantitative specification of the amount of information to be released by comparing PERs. Namely, if we have an input-output model of the system, whose kernel is R', and the analysis of the implementation shows that the kernel of the implementation is less than R', then the implementation is safe with respect to any desired quantitative information release, regardless of the initial assumption about the attacker's uncertainty. This is useful because the attacker's uncertainty may not be known.

3 Static Analysis

In this section we present the concrete static analysis of while programs with respect to an attacker model A. Since the attacker model is clear, we shall simply write the typing judgement $\Gamma_A \vdash c : R \Rightarrow R'$ as $c : R \Rightarrow R'$.

4 Information Flow Policies

We present a semantic definition of information flow policies, which characterises our intentional information release requirements. Given a lattice of information, an information flow policy is

$$\texttt{skip}: R \Rightarrow R \qquad z := e : R \Rightarrow R \qquad \texttt{read}\,x : R \Rightarrow R \qquad \texttt{write}\,e : R \Rightarrow R \sqcup e : id$$

$$\frac{c_1 : R \Rightarrow R' \quad c_2 : R' \Rightarrow R''}{c_1 ; c_2 : R \Rightarrow R''}$$

$$\frac{\begin{array}{l} \Sigma_1' = \{\sigma' \mid \sigma \in \Sigma, \sigma(b) = \mathbf{tt}, \langle c_1, \sigma\rangle \Downarrow \sigma'\} \quad \Sigma_2' = \{\sigma' \mid \sigma \in \Sigma, \sigma(b) = \mathbf{ff}, \langle c_2, \sigma\rangle \Downarrow \sigma'\} \\ \forall \sigma, \sigma' \in \Sigma \quad \sigma R' \sigma' \iff \begin{array}{lcl} \sigma(b) = \sigma'(b) = \mathbf{tt} & \Longrightarrow & obs(c_1,\sigma) = obs(c_1,\sigma') \\ \sigma(b) = \sigma'(b) = \mathbf{ff} & \Longrightarrow & obs(c_2,\sigma) = obs(c_2,\sigma') \\ \sigma(b) = \mathbf{tt}, \sigma'(b) = \mathbf{ff} & \Longrightarrow & obs(c_1,\sigma) = obs(c_2,\sigma') \end{array} \end{array}}{\texttt{if}\,(b)\,\texttt{then}\,c_1\,\texttt{else}\,c_2 : \langle \Sigma, R\rangle \to \langle \Sigma_1' \cup \Sigma_2', R \uplus \uparrow_{\mathbf{TVar}} R'\rangle}$$

$$\frac{\begin{array}{c} \texttt{if}\,(b)\,\texttt{then}\,c\,\texttt{else}\,\texttt{skip} : \langle \Sigma_i, R_i\rangle \to \langle \Sigma_{i+1}, R_{i+1}\rangle \\ \Sigma' = \{\sigma \in \bigcup_{i \geq 0} \Sigma_i \mid \sigma(b) = \mathbf{ff}\} \quad \forall \sigma, \sigma' \in \Sigma', \sigma R'' \sigma' \quad R' = \biguplus_{i\geq 0} R_i \uplus \uparrow_{\mathbf{TVar}} R'' \end{array}}{\texttt{while}\,(b)\,\texttt{do}\,c : \langle \Sigma, R\rangle \to \langle \Sigma', R'\rangle} \quad \begin{array}{l} \Sigma_0 = \Sigma \\ R_0 = R \end{array}$$

Figure 4: *Information Release Typing Rules*

a transformer, which sets upper bounds on the information transferred through a program to an observer. Since the secrets to be protected are stored in program states during computation, our information lattice is defined as partial equivalence relation over states.

Definition 1 (Information Release Policy) Let $\blacksquare$ be the set of program states and let $\mathcal{I} \triangleq PER(\blacksquare)$ be the set of all partial equivalence relations over $\blacksquare$. An information flow policy $R \Rightarrow R'$ is a transformer over the lattice $\mathcal{I}$ such that $R \sqsubseteq R'$.

The information release policy $R \Rightarrow R'$ allows the observer to gain at most the information R' if the observer has a prior information of at least R. Intuitively, the requirement $R \sqsubseteq R'$ means that information release policies can only increase the observer's knowledge. As an example, a policy that releases the parity of the secret contained in variable x may be defined as: $all \Rightarrow \mathrm{Par}_x$, where Par_x is the equivalence relation defined such that $\forall \sigma, \sigma' \in \blacksquare, \sigma\, \mathrm{Par}_x\, \sigma' \iff \sigma(x) = \sigma'(s)$ 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 can distinguish at most the parity of x after the release.

The information flow transformers $R \Rightarrow R'$ are also used as *security types* that can be assigned to programs to characterise their information release properties. The information release typing judgement $\Gamma_A \vdash c : R \Rightarrow R'$ (under the attacker model A) is valid iff $\forall \alpha \in \mathcal{A}_A^*$ and $\sigma_1, \sigma_2 \in \blacksquare$

$$\sigma_1 R' \sigma_2 \wedge \langle c, \sigma_1\rangle \xrightarrow{\alpha}_A \langle c', \sigma_1'\rangle \Longrightarrow \exists \langle c'', \sigma_2'\rangle \in \mathcal{S} : \langle c, \sigma_2\rangle \xrightarrow{\alpha}_A \langle c'', \sigma_2'\rangle \wedge \sigma_1 R \sigma_2. \tag{2}$$

The typing judgement $\Gamma_A \vdash c : R \Rightarrow R'$ means that 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. This semantic definition of information flow ties together the standard program semantics, the attacker's observational power, and the information release. Informally, (2) means that the information flow typing holds iff for any pair of initial states σ_1, σ_2 of c, which are indistinguishable under the released information R' are also indistinguishable under R, and any observation α that A can make of the partial execution of c under σ_1 can be made as well under σ_2. The first clause ensures that knowledge of A is rising monotonically, and the second ensures that the execution starting from σ_1 cannot be distinguished,based on the attacker's observation, from the one starting from σ_2. Since R' and R are symmetric, the same can be said of the observations made during the partial execution of c under σ_2.

A program P is said to satisfy a policy $R \Rightarrow R'$ under the attacker model A iff $\forall R_1 \in \mathcal{I}$ such that $\Gamma_A \vdash P : R_1 \Rightarrow R_2$, we have that $R_2 \sqsubseteq R_1 \sqcup R'$.

Authentication Policies. Let us start by considering the archetypal password authentication program in Figure 9, which models the key steps of the authentication process that we wish to consider. During these steps, the *user-supplied password* (u) is compared with a password (p), which has been previously stored[1] in the system and is known only to the legitimate user. These secrets (or their images) are then compared for equality: if there is a match, the user is authenticated, otherwise the authentication fails. In Figure 9, an output of 1 signals successful authentication, and an output of 2 signals authentication failure.

```
if ($u = p$) then
  write 1;        // authenticated
else
  write 2;        // not authenticated
```

Figure 5: *A Model of Authentication*

Thus, in an authentication program, the information that we wish to release is the equality or not of the stored and the user-supplied passwords. Let this program be P, and let its set of states be $\blacksquare$, then the equivalence relations over $\blacksquare$ describing this information release is $R_{auth} \in ER(\blacksquare)$ such that $\forall \sigma, \sigma' \in \blacksquare, \sigma\, R_{auth}\, \sigma' \iff \sigma(u = p) = \sigma'(u = p)$, which relates any pair of states that both agree on the equality or not of u and p. Hence, the desired authentication policy is a map $f_{auth}(R) \triangleq R \sqcup R_{auth}$, which declassifies R_{auth}. This definition is intuitive. For example, on one hand, if the attacker had no prior knowledge, then the attacker is allowed to gain $f_{auth}(all) = R_{auth}$, which is exactly the declassified information. On the other hand, if the attacker already knows the user-supplied password u, say by supplying a guess, the attacker's prior knowledge is the identity of u, $id_u \in ER(\blacksquare)$, defined such that $\forall \sigma, \sigma' \in \blacksquare, \sigma\, id_u\, \sigma' \iff \sigma(u) = \sigma'(u)$. In this case, the information that the policy allows the attacker to gain is $f_{auth}(id_u) = id_u \sqcup R_{auth}$. But we observe that $\sigma\, id_u \sqcup R_{auth}\, \sigma' \iff (\sigma(p) = \sigma(u) = \sigma'(p) = \sigma'(u)) \vee (\sigma(p) \neq \sigma(u) = \sigma'(u) \neq \sigma'(p))$. This means that, given the knowledge of u, the attacker does not gain more than the fact that the stored

[1] In many modern operating systems a password is not directly stored, but its image, which is usually a secure hash of the password itself. The authentication process involves checking the hash of the user-supplied password against the hash of the stored password. In Unix-based systems, salts are also used in order to make dictionary attacks less successful [?, ?, ?].

password p is *that* known u (that is $\sigma(p) = \sigma(u) = \sigma'(p) = \sigma'(u)$), or that p is not equal to that u (the $\sigma(p) \neq \sigma(u) = \sigma'(u) \neq \sigma'(p)$ part). The static analysis shows that the program of Figure 9 is secure, since $P : \langle \blacksquare, all\rangle \rightarrow \langle \blacksquare, R_{auth}\rangle$.

```
if (u = p) then
  write 1;      // authenticated
else
  write 2;      // not authenticated
write u;           // attack
```

Figure 6: *A rogue authentication program*

Now consider a rogue implementation of the authentication program of Figure 6, which reveals the user supplied password in addition to the authentication result. The implementation is clearly insecure, because, even without previously knowing the user supplied password, the attacker now learns what the password is when authentication succeeds, and learns what the password is not when authentication fails. The static analysis detects this, because the analysis of this rogue program, let's call it P_6, is $P_6 : \langle \blacksquare, all\rangle \rightarrow \langle \blacksquare, id_u \sqcup R_{auth}\rangle$. This program is rejected since $id_u \sqcup R_{auth} \not\sqsubseteq R_{auth}$ as required by the policy f_{auth}.

Now let us consider the quantitative policy for the password program. As demonstrated above, we only want to reveal whether u and p match or not, and not more. This corresponds to the quantitative information release due to the partitioning of $\blacksquare$ by R_{auth}. Thus, the desired quantitative policy is f defined such that for any given uncertainty μ over $\blacksquare$, $f(\mathcal{H}(\mu_\perp) - \mathcal{H}(\mu)) = \mathcal{H}(\mu_\perp) - \mathcal{H}(\mu|R_{auth})$.

Encryption Policies. Encryption is an important primitive that is foundational to the security of many systems. However, encryption functions do release information about the keys and messages. It is thus important to be able to characterise the information released by a crypto algorithm and to check whether a given implementation does not release more than is intended. To illustrate the approach consider a data backup program which, when given a message and a key computes the *xor* of the two and stores the result to a publicly observable place. Thus, given the key k and message m, the program computes the function $\boldsymbol{enc}(k,m) = k\,\boldsymbol{xor}\,m$. Thus, the program induces the equivalence relation R_{xor} on states such that $\forall \sigma, \sigma' \in \blacksquare, \sigma\, R_{xor}\, \sigma' \iff \sigma(\boldsymbol{enc}(k,m)) = \sigma'(\boldsymbol{enc}(k,m))$

5 Example: Password Timing Attacks

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

Figure 7: *Password-checking program, version 1.*

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

Figure 8: *Password-checking program, version 2.*

```
passed:=false ;
attempts :=0;
while (passed=false )
    read user ;
    if (member(user,U)) then
        read pw;
        if ( valid ( user ,pw)) then
            write 1
            passed:=true
        else
            attempts :=attempts+1
            delay (na∗attempts)
            write 2
    else
        attempts :=attempts+1
        delay (nb∗attempts)
        write 2
```

Figure 9: *Password-checking program, version 3.*

6 Conclusions and Future Work

...
- typechecking implementation
- how to ensure security of a plugin-based system? if you've verified code but not the plugin -¿¿¿¿ compositionality

A Proofs

Lemma 1 *For any set Ω, the partially ordered set $\langle PER(\Omega), \sqsubseteq^{+}, \sqcup^{+} \rangle$ is a complete lattice.*

Proof. The proof that $\sqsubseteq^{+}$ is a partial order is straightforward. We shall now show that for any $\mathcal{R} \subseteq PER(\Omega)$ and $R' \in PER(\Omega)$ such that for all $R \in \mathcal{R}, R \sqsubseteq R'$, we have that $\bigsqcup^{+} R \sqsubseteq^{+} R'$. It is clear from the definition that if $D_2 = dom(R_2)$ and $R_1 \sqsubseteq^{+} R_2$ then $dom(R_1) \subseteq D_2$ and $\mathcal{C}_{D_2}(R_2) = R_2 = R_1 \sqcup^{+} R_2 = \mathcal{C}_{D_2}(R_1) \sqcup R_2$. Now let $D = dom(R')$. Hence, by definition $\bigsqcup^{+} \mathcal{R} \sqcup^{+} R' = \bigsqcup_{R \in \mathcal{R}} \mathcal{C}_D(R) \sqcup R' = R'$, since we know that for all $R \in \mathcal{R}, R \sqsubseteq^{+} R'$. Hence, $\bigsqcup^{+} \mathcal{R} \sqcup^{+} R' = R'$, which implies $\bigsqcup^{+} \mathcal{R} \sqsubseteq^{+} R'$, showing the desired property. □

Lemma 2 *Let $\pi(X) = \{X_j \mid j \in J\}$ be a partitioning of the set X, not necessarily covering X, such that $\bigcup_{j \in J} X_j \subseteq X$, and let μ be a probability measure. We have that $\mu(X)\mathcal{H}(\mu|X) \leq \sum_{j \in J} \mu(X_j)\mathcal{H}(\mu|X_j)$.*

Proof. Let $Y = \bigcup_{j \in J} X_j$ and let $Z = X \backslash Y$. Define $f : X \to \pi(X) \cup \varnothing$ such that for any $x \in X, f(x) = X_j$ if $x \in X_j$ and $f(x) = \varnothing$ otherwise. Thus, because of the partitioning of X, for any $x, x' \in X$ such that $x \notin f(x')$, we have that $\mu(x|f(x')) = 0$ since $\mu(\varnothing) = 0$ and $\{x\} \cap f(x') = \varnothing$. Therefore, we have that $\sum_{j \in J} \mu(X_j)\mathcal{H}(\mu|X_j) = -\sum_{j \in J} \mu(X_j) \sum_{x \in X_j} \mu(x|X_j) \log(\mu(x|X_j) = -\sum_{x \in Y} \mu(f(x))\mu(x|f(x)) \log(\mu(x|f(x))$. Furthermore, since for any $x \in Z, \mu(x|f(x)) = 0$, we have that $\sum_{j \in J} \mu(X_j)\mathcal{H}(\mu|X_j) = -\sum_{x \in X} \mu(f(x))\mu(x|f(x)) \log(\mu($
Now take any $x \in X$, we observe that $0 \leq \mu(f(x)) \leq \mu(X) \leq 1$ since $f(x) \subseteq X$ and μ is a probability measure. By the same token, $0 \leq \mu(x|X) \leq \mu(x|f(x)) \leq 1$ by the definition of conditioning. Hence, $\log(\mu(x|X)) \leq \log(\mu(x|f(x))) \leq 0$. Furthermore, we observe that if $f(x) = \varnothing$, $\mu(f(x))\mu(x|f(x)) = 0$ and $\mu(f(x))\mu(x|f(x)) = \mu(x) = \mu(X)\mu(x|X)$ otherwise. Hence, $\mu(X)\mu(x|X) \log(\mu(x|X)) \leq \mu(f(x))\mu(x|f(x)) \log(\mu(x|f(x)))$ Therefore, $-\sum_{x \in X} \mu(X)\mu(x|X) \log(\mu(x|X)) \geq -\sum_{x \in X} \mu(f(x))\mu(x|f(x)) \log(\mu(x|f(x)))$. That is, $\mu(X)\mathcal{H}(\mu|X) \geq \sum_{j \in J} \mu(X_j)\mathcal{H}(\mu|X_j)$. □

Theorem 1 *Let μ and μ' be probability measures over Ω, and let $R, R' \in PER(\Omega)$ be PERs over Ω. Then*

1. $R \sqsubseteq R' \implies \mathcal{H}(\mu|R') \leq \mathcal{H}(\mu|R)$.
2. $\mathcal{H}(\mu|R) = \dfrac{\mathcal{H}(\mu)}{\mathcal{H}(\mu')} \times \mathcal{H}(\mu'|R)$.

Proof. 1. The proof follows immediately from Lemma 2 by induction on the equivalence classes of R, since $R \sqsubseteq R'$ implies that each equivalence class X of R is partitioned by a maximal set $\{X_j \mid j \in J\} \subseteq [\Omega]_{R'}$ of equivalence classes of R' such that $\bigcup_{j \in J} X_j \subseteq X$.

2.

□

Proof.

1. This is clear from the definition.

2. Since $R' \sqsubseteq R''$ then $R'' = R' \uplus R''$ and hence $\Sigma'' = dom(R'') = dom(R'') \cup dom(R')$. Define $\overline{R'}$ such that $\forall \sigma, \sigma' \in \blacksquare, \sigma \, \overline{R'} \, \sigma' \iff \sigma, \sigma' \in \Sigma'' \backslash dom(R')$. Since $\Sigma'' = dom(R'')$ then $\mathcal{C}_{\Sigma''}(R'') = R''$. Thus, $R'' = R' \uplus R'' = \mathcal{C}_{\Sigma''}(R') \sqcup R'' = (R' \sqcup R'') \cup (R'' \sqcup \overline{R'})$. Hence, $R'' \sqcup R = R \sqcup R' \sqcup R''$ because $\overline{R'} \sqcup R = \varnothing$ since $dom(R) \subseteq dom(R')$, and, R' and $\overline{R'}$ are disjoint by definition. Since $R'' \sqcup R = R \sqcup R' \sqcup R''$, then $R \sqcup R' \sqsubseteq R \sqcup R''$.

3. The symmetry of $\uparrow_Z R$ is clear from the definition. For transitivity, suppose $\sigma \uparrow_Z R \, \sigma'$ and $\sigma' \uparrow_Z R \, \sigma''$ hold. Then there exist two sequences of states $\sigma_1, \ldots, \sigma_n \in \blacksquare$ and $\sigma'_1, \ldots, \sigma'_m \in \blacksquare$ such that for all $i = 1, \ldots, n-1$ and $j = 1, \ldots, m-1$ there exist $\sigma^A_i, \sigma^B_j \in dom(R)$ such that $\sigma_i, \sigma_{i+1} \in havocZ([\sigma^A_i]_R)$ and $\sigma'_j, \sigma'_{j+1} \in havocZ([\sigma^B_j]_R)$ and $\sigma = \sigma_1$ and $\sigma' = \sigma_n = \sigma'_1$ and $\sigma'' = \sigma'_m$. Thus, transitivity of $\uparrow_Z R$ is clear by concatenating the two sequences of states.

4. The extensivity of $havocZ(\cdot)$ is clear from the definition, that is for any $\Sigma \subseteq \blacksquare$, we have that $\Sigma \subseteq havocZ(\Sigma)$. Hence, $[\sigma]_{\uparrow_Z R} \subseteq havocZ([\sigma]_{\uparrow_Z R})$. Now take any $\sigma' \in havocZ([\sigma]_{\uparrow_Z R})$, then there exists $\sigma'' \in [\sigma]_{\uparrow_Z R}$ such that for all $y \in \mathbf{Var} \backslash Z, \sigma'(y) = \sigma''(y)$. Since $\sigma'' \in [\sigma]_{\uparrow_Z R}$ then $\sigma \uparrow_Z R \, \sigma''$ holds. It is thus clear from the definition of $\uparrow_Z R$ that $\sigma'' \uparrow_Z R \, \sigma'$ holds since σ' is be obtained from σ'' possibly by modifying values of variables in Z. Transitivity of $\uparrow_Z R$ means that $\sigma \uparrow_Z R \, \sigma'$ also holds and hence $\sigma' \in [\sigma]_{\uparrow_Z R}$, which means that $havocZ([\sigma]_{\uparrow_Z R}) \subseteq [\sigma]_{\uparrow_Z R}$.

5. We shall start by showing that $\uparrow_X \uparrow_Y R = \uparrow_{X \cup Y} R$. Let $Z = X \cup Y$. Then from the definition of $\uparrow_X(\cdot)$ we have that for any $\sigma, \sigma' \in \blacksquare$, $\sigma \uparrow_X \uparrow_Y R \, \sigma'$ iff there exist sequences $\sigma_1, \ldots, \sigma_n \in \blacksquare$ and $\tau_1, \ldots, \tau_{n-1} \in dom(\uparrow_Y R)$, such that $\sigma = \sigma_1$ and $\sigma' = \sigma_n$ and for all i, $1 \leq i \leq n-1$ implies $\sigma_i, \sigma_{i+1} \in havocX([\tau_i]_{\uparrow_Y R}) = havocZ([\tau_i]_{\uparrow_Y R})$ since by (**??**) $[\tau_i]_{\uparrow_Y R} = havocY([\tau_i]_{\uparrow_Y R})$. Hence, for all i, $1 \leq i \leq n-1$ there exist $\sigma'_i, \sigma'_{i+1} \in [\tau_i]_{\uparrow_Y R}$ such that $\sigma_i \in havocZ(\{\sigma'_i\})$ and $\sigma_{i+1} \in havocZ(\{\sigma'_{i+1}\})$ and since σ'_i and σ'_{i+1} are related by $\uparrow_Y R$ then by definition there exist sequences $\sigma^i_1, \ldots, \sigma^i_{m_i} \in \blacksquare$ and $\tau^i_1, \ldots, \tau^i_{m_i - 1} \in dom(R)$ such that $\sigma'_i = \sigma^i_1$ and $\sigma'_{i+1} = \sigma^i_{m_i}$ and $\forall j, 1 \leq j \leq m_{i-1} - 1 \implies \sigma^i_j, \sigma^i_{j+1} \in havocY([\tau^i_j]_R)$. Since $\sigma'_i \in havocY([\tau^i_1]_R)$ and $\sigma_i \in havocZ(\{\sigma'_i\})$ hence $\sigma_i \in havocZ([\tau^i_1]_R)$. Similarly, $\sigma_{i+1} \in havocZ([\tau^i_{m_i - 1}]_R)$. Hence for any i, $1 \leq i \leq n-1$ we obtain the sequences $\sigma_i, \sigma^i_1, \ldots, \sigma^i_{m_i - 1}, \sigma_{i+1} \in \blacksquare$ and $\tau^i_1, \ldots, \tau^i_{m_i - 1} \in dom(R)$ such that $\sigma_i, \sigma^i_1 \in havocZ([\tau^i_1]_R)$ and $\sigma_{i+1}, \sigma^i_{m_i - 1} \in havocZ([\tau^i_{m_1 - 1}]_R)$ and for all $j, 1 \leq j \leq m_{i-1} - 1 \implies \sigma^i_j, \sigma^i_{j+1} \in havocY([\tau^i_j]_R) \subseteq havocZ([\tau^i_j]_R)$. Since $\sigma = \sigma_1$ and $\sigma' = \sigma_n$, hence by definition, $\sigma \uparrow_Z R \, \sigma'$.

   The reverse implication is straightforward because by definition $\sigma \uparrow_Z R \, \sigma'$ holds iff $\exists \sigma_1, \ldots, \sigma_n \in \blacksquare$ and $\tau_1, \ldots, \tau_{n-1} \in dom(R)$ and $\sigma = \sigma_1, \sigma' = \sigma_n$ such that for all $i, i \leq i \leq n-1 \implies \sigma_i, \sigma_{i+1} \in havocZ([\tau_i]_R)$. Now, since for any $\tau \in dom(R), [\tau]_R \subseteq [\tau]_{\uparrow_Y R}$ and $dom(R) \subseteq dom(\uparrow_Y R)$ then $\tau_1, \ldots, \tau_{n-1} \in dom(\uparrow_Y R)$ and hence by replacing $[\tau_i]_R$ above with $[\tau_i]_{\uparrow_Y R}$ for all i, we obtain $\sigma_i, \sigma_{i+1} \in havocX(havocY([\tau_i]_{\uparrow_Y R})) = havocX([\tau_i]_{\uparrow_Y R})$ by applying (**??**). Hence, $\sigma \uparrow_X \uparrow_Y R \, \sigma'$ holds.

   Since $\uparrow_X \uparrow_Y R = \uparrow_{X \cup Y} R$, then the fact that set union is commutative means that $\uparrow_X \uparrow_Y R = \uparrow_{X \cup Y} R = \uparrow_{Y \cup X} R = \uparrow_Y \uparrow_X R$.

6. Let $\Sigma = dom(\uparrow_Z R) \cup dom(\uparrow_Z R')$. Now define the PERs $\overline{\uparrow_Z R}$ and $\overline{\uparrow_Z R'}$ such that $\forall \sigma, \sigma' \in \blacksquare, \sigma \, \overline{\uparrow_Z R} \, \sigma' \iff \sigma, \sigma' \in \Sigma \backslash dom(\uparrow_Z R)$ and $\sigma \, \overline{\uparrow_Z R'} \, \sigma' \iff \sigma, \sigma' \in \Sigma \backslash dom(\uparrow_Z R')$. The PERs

$\overline{\uparrow_Z R}$ and $\overline{\uparrow_Z R'}$ both have only one partition, which respectively are the sets $\Sigma_1 = \Sigma \backslash dom(\uparrow_Z R)$ and $\Sigma_2 = \Sigma \backslash dom(\uparrow_Z R')$. Therefore, by (**??**) we have that $havocZ(\Sigma_1) = \Sigma_1$ and $havocZ(\Sigma_2) = \Sigma_2$ since by (**??**) we know that $dom(\uparrow_Z R) = havocZ(dom(\uparrow_Z R))$ and $dom(\uparrow_Z R') = havocZ(dom(\uparrow_Z R'))$ and hence by (**??**) that $\Sigma = havocZ(\Sigma)$, since $havocZ(\cdot)$ is idempotent. That is, $\uparrow_Z\left(\overline{\uparrow_Z R}\right) = \overline{\uparrow_Z R}$ and $\uparrow_Z\left(\overline{\uparrow_Z R'}\right) = \overline{\uparrow_Z R'}$. By definition $\mathcal{C}_\Sigma(\uparrow_Z R) = \uparrow_Z R \cup \overline{\uparrow_Z R}$, and hence by applying (**??**), we know that for any $\sigma \in dom(\mathcal{C}_\Sigma(\uparrow_Z R))$, $[\sigma]_{\mathcal{C}_\Sigma(\uparrow_Z R)} = havocZ([\sigma]_{\mathcal{C}_\Sigma(\uparrow_Z R)})$, because $[\sigma]_{\uparrow_Z R} = havocZ([\sigma]_{\uparrow_Z R})$ and $[\sigma]_{\overline{\uparrow_Z R}} = havocZ([\sigma]_{\overline{\uparrow_Z R}})$. Therefore, for any $\sigma, \sigma' \in \blacksquare$, $\sigma \uparrow_Z \mathcal{C}_\Sigma(\uparrow_Z R)\, \sigma'$ iff $\exists \sigma_1, \ldots, \sigma_n \in \blacksquare$, $\sigma'_1, \ldots, \sigma'_{n-1} \in dom(\mathcal{C}_\Sigma(\uparrow_Z R))$, such that $\sigma = \sigma_1, \sigma' = \sigma_n$ and $\forall i,\ i \leq i \leq n-1 \implies \sigma_i, \sigma_{i+1} \in havocZ([\sigma'_i]_{\mathcal{C}_\Sigma(\uparrow_Z R)}) = [\sigma'_i]_{\mathcal{C}_\Sigma(\uparrow_Z R)}$. Hence, we have that $\uparrow_Z(\mathcal{C}_\Sigma(\uparrow_Z R)) = \mathcal{C}_\Sigma(\uparrow_Z R)$. Similarly, we obtain $\uparrow_Z(\mathcal{C}_\Sigma(\uparrow_Z R')) = \mathcal{C}_\Sigma(\uparrow_Z R')$. Since by definition, $\uparrow_Z R \uplus \uparrow_Z R' = \mathcal{C}_\Sigma(\uparrow_Z R) \sqcup \mathcal{C}_\Sigma(\uparrow_Z R')$, hence we obtain $\uparrow_Z(\uparrow_Z R \uplus \uparrow_Z R') = \uparrow_Z R \uplus \uparrow_Z R'$ by applying (**??**).

7. Take any $\sigma_1, \sigma_2 \in \Sigma = dom(R)$ such that $(\sigma_1, \sigma_2) \notin R$, then it is clear that $[\sigma_1]_R \cap [\sigma_2]_R = \varnothing$, since the states belong to different partitions of R. Hence, $havoc\mathbf{TVar}([\sigma_1]_R) \cap havoc\mathbf{TVar}([\sigma_2]_R) = \varnothing$. This is so because if there exist $\sigma \in [\sigma_1]_R$ and $\sigma' \in [\sigma_2]_R$ such that $havoc\mathbf{TVar}(\{\sigma\}) \cap havoc\mathbf{TVar}(\{\sigma'\}) \neq \varnothing$, then $\sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}}$ and therefore $\sigma_{\downarrow X} = \sigma'_{\downarrow X}$. This means that $\sigma(e) = \sigma'(e)$ since the two states both agree on the values of all the free variables of e, and hence $\sigma' \in [\sigma]_R = [\sigma_1]_R$, which violates our initial assumption that $[\sigma_1]_R$ and $[\sigma_2]_R$ are disjoint. This means by the definition of $\uparrow_{\mathbf{TVar}} R$, that $\sigma \uparrow_{\mathbf{TVar}} R\, \sigma'$ iff there exists $\sigma'' \in dom(R)$ such that $\sigma, \sigma' \in havoc\mathbf{TVar}([\sigma'']_R)$. Thus, we have that for any $\sigma'' \in dom(R)$, $[\sigma'']_{\uparrow_{\mathbf{TVar}} R} = havoc\mathbf{TVar}([\sigma'']_R)$. Since for any $\sigma_1, \sigma_2 \in dom(R), (\sigma_1, \sigma_2) \notin R \implies havoc\mathbf{TVar}([\sigma_1]_R) \cap havoc\mathbf{TVar}([\sigma_2]_R) = [\sigma_1]_{\uparrow_{\mathbf{TVar}} R} \cap [\sigma_2]_{\uparrow_{\mathbf{TVar}} R} = \varnothing \implies (\sigma_1, \sigma_2) \notin \uparrow_{\mathbf{TVar}} R$, it follows by the contrapositive that for all $\sigma, \sigma' \in dom(R) = \Sigma, \sigma \uparrow_{\mathbf{TVar}} R\, \sigma' \implies \sigma R \sigma' \implies \sigma(e) = \sigma'(e)$.

□

We shall now show that the information flow and semantic correctness of the static analysis. Before that, we need to show a property of information flow due to the sequential composition of programs.

Proposition 1 *Let $P = P_1;P_2$ be a* While *program, whose set of states is $\blacksquare$. Define the PER $\lfloor P_1 \bullet P_2 \rfloor$ to be $\forall \sigma, \sigma' \in \blacksquare, \sigma \lfloor P_1 \bullet P_2 \rfloor \sigma'$ iff $\sigma \lfloor P_1 \rfloor \sigma'$ and $\langle P_1, \sigma \rangle \Downarrow \sigma_1, \langle P_1, \sigma' \rangle \Downarrow \sigma'_1$ implies $\sigma_1 \lfloor P_2 \rfloor \sigma'_1$. Then we have $\lfloor P \rfloor \sqsubseteq \lfloor P_1 \bullet P_2 \rfloor$.*

Proof. Take any $\sigma, \sigma' \in \blacksquare$ such that $\sigma \lfloor P_1 \bullet P_2 \rfloor \sigma'$ holds. Then, $\sigma \lfloor P_1 \rfloor \sigma'$ holds, which by Definition **??** means that either P_1 terminates under both states σ and σ' or that it diverges under both states, and that the attacker makes the same observation on the traces of the two states, that is, $obs(P_1, \sigma) = obs(P_1, \sigma')$.

In the first case, suppose that P_1 diverges under both σ and σ', then we have $obs(P_1, \sigma) = obs(P_1, \sigma') = obs(P, \sigma) = obs(P, \sigma')$ since the trailing subprogram P_2 of P cannot be executed due to the divergence of P_1, and hence $\sigma \lfloor P \rfloor \sigma'$ holds.

Now suppose P_1 terminates under both σ and σ', then $\sigma \lfloor P_1 \bullet P_2 \rfloor \sigma'$ implies that $\sigma \lfloor P_1 \rfloor \sigma'$ holds, and there exist $\sigma_1, \sigma'_1 \in \blacksquare$ such that $\langle P_1, \sigma \rangle \Downarrow \sigma_1$ and $\langle P_1, \sigma' \rangle \Downarrow \sigma'_1$ and $\sigma_1 \lfloor P_2 \rfloor \sigma'_1$ holds. Thus, $obs(P_1, \sigma) = obs(P_1, \sigma')$ and $obs(P_2, \sigma_1) = obs(P_2, \sigma'_1)$, and therefore, $obs(P, \sigma) = obs(P, \sigma')$, which means that $\sigma \lfloor P \rfloor \sigma'$ holds. Thus, $\lfloor P \rfloor \sqsubseteq \lfloor P_1 \bullet P_2 \rfloor$. □

Proof. The proof proceeds by exhaustion of the analysis rules.

- The proof for the case when P is the `skip` statement is straightforward.
- The proof for the case when P is $z := e$ is straightforward.
- Suppose P is $\texttt{write}\, e$. Then $\Sigma' = \Sigma = \{\sigma \mid \sigma \in \Sigma, \langle \texttt{write}\, e, \sigma \rangle \Downarrow \sigma\}$. Let R_1 be defined such that $\sigma R_1 \sigma'$ iff $\sigma, \sigma' \in \Sigma, \sigma(e) = \sigma'(e)$. Since $\Uparrow_{\mathbf{TVar}} R = R$ then it is clear, by applying (**??**) and (**??**) of proposition **??**, that $\Uparrow_{\mathbf{TVar}} R' = R'$ since $R' = R \uplus \Uparrow_{\mathbf{TVar}} R_1$. It now remains to show that $R_\Sigma \sqcup \lfloor \texttt{write}\, e \rfloor = R_\Sigma \sqcup R_1 \sqsubseteq R' \sqcup R_\Sigma$, that is $\forall \sigma, \sigma' \in \Sigma, \sigma R' \sigma' \implies \sigma(e) = \sigma'(e)$. Let $X = FV(e) \cap \mathbf{TVar}$, since $\mathbf{TVar}$ variables are assigned before use, and $\mathbf{IVar}$ variables are fixed throughout program execution, then $\forall \sigma, \sigma' \in \Sigma, \sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma_{\downarrow X} = \sigma'_{\downarrow X}$. That is, the values of the X projection of states is a function of the P's formal parameter since *While* is deterministic. It therefore follows from (**??**) of proposition **??** that $\forall \sigma, \sigma' \in \Sigma, \sigma R' \sigma' \implies \sigma(e) = \sigma'(e)$.
- Now suppose that P is $c_1; c_2$, such that $c_1 : \langle \Sigma, R \rangle \to \langle \Sigma_1, R_1 \rangle$ and $c_2 : \langle \Sigma_1, R_1 \rangle \to \langle \Sigma', R' \rangle$. Then by applying the induction hypothesis to the derivation of c_1 and c_2 we have that $\Sigma_1 = \{\sigma' \mid \sigma \in \Sigma, \langle c_1, \sigma \rangle \Downarrow \sigma'\}$ and $\Sigma' = \{\sigma' \mid \sigma \in \Sigma_1, \langle c_2, \sigma \rangle \Downarrow \sigma'\}$ and also that $\Uparrow_{\mathbf{TVar}} R = R$ and $\Uparrow_{\mathbf{TVar}} R' = R'$ and that $\lfloor c_1 \rfloor \sqcup R_\Sigma \sqsubseteq R_1 \sqcup R_\Sigma$ and $\lfloor c_2 \rfloor \sqcup R_{\Sigma_1} \sqsubseteq R' \sqcup R_{\Sigma_1}$ where $\sigma R_{\Sigma_1} \sigma' \iff \sigma, \sigma' \in R_{\Sigma_1}$. It is clear that $\Sigma' = \{\sigma' \mid \sigma \in \Sigma, \langle c_1; c_2, \sigma \rangle \Downarrow \sigma'\}$.

  It now remains to show that $\lfloor c_1; c_2 \rfloor \sqcup R_\Sigma \sqsubseteq R' \sqcup R_\Sigma$. Now define $\lfloor c_1 \bullet c_2 \rfloor$ to be such that $\forall \sigma, \sigma' \in \blacksquare, \sigma \lfloor c_1 \bullet c_2 \rfloor \sigma'$ iff $\sigma \lfloor c_1 \rfloor \sigma'$ and $\langle c_1, \sigma \rangle \Downarrow \sigma_1, \langle c_1, \sigma' \rangle \Downarrow \sigma'_1 \implies \sigma_1 \lfloor c_2 \rfloor \sigma'_1$. Furthermore, define $\blacksquare_{\Downarrow} = \{\sigma_A \in \blacksquare \mid \langle c_1, \sigma_A \rangle \Downarrow \sigma_B\}$ to be the set of all states under which c_1 terminates, and define $f : \blacksquare_{\Downarrow} \to \blacksquare$ such that for all $\sigma_A \in \blacksquare_{\Downarrow}$, $f(\sigma_A) = \sigma_B$ iff $\langle c_1, \sigma_A \rangle \Downarrow \sigma_B$ to represent the transformation of states by c_1. Now let $R_0 \in PER(\blacksquare)$ be defined such that $\sigma R_0 \sigma'$ iff $\sigma, \sigma' \in \blacksquare \backslash \blacksquare_{\Downarrow}$ or $\sigma, \sigma' \in \blacksquare_{\Downarrow} \implies f(\sigma) \lfloor c_2 \rfloor f(\sigma')$. It is easy to see that $\lfloor c_1 \bullet c_2 \rfloor = \lfloor c_1 \rfloor \sqcup R_0$.

  We know from the analysis rules that $R \trianglerighteq R_1 \trianglerighteq R'$ since R' is computed by taking a join $\uplus$ of R_1 with some other PERs, and similarly, R_1 from R. Hence, $R_\Sigma \sqcup R_1 \sqsubseteq R_\Sigma \sqcup R'$, by applying (**??**) of proposition **??**, since $\Sigma \subseteq dom(R) \subseteq dom(R_1)$. Hence, $\lfloor c_1 \rfloor \sqcup R_\Sigma \sqsubseteq R' \sqcup R_\Sigma$ by combining this fact with the inductive hypothesis. Now suppose that $\sigma (R_\Sigma \sqcup R') \sigma'$ holds, then $\sigma, \sigma' \in \Sigma$ and $\sigma R' \sigma'$ holds. Furthermore, suppose that $\sigma, \sigma' \in \blacksquare_{\Downarrow}$, then we know that $f(\sigma) \in havoc\mathbf{TVar}(\{\sigma\})$ and $f(\sigma') \in havoc\mathbf{TVar}(\{\sigma'\})$ since c_1 does not modify the $\mathbf{IVar}$ projection of states. Hence, $\sigma R' f(\sigma)$ and $\sigma' R' f(\sigma')$ hold by applying (**??**) of proposition **??**, since $\Uparrow_{\mathbf{TVar}} R' = R'$ by the induction hypothesis. Therefore, $f(\sigma) R' f(\sigma')$ holds by the transitivity of R' since $\sigma R' \sigma'$ holds. Hence, $\forall \sigma, \sigma' \in \blacksquare_{\Downarrow} \cap \Sigma, \sigma R' \sigma' \implies f(\sigma) (R' \sqcup R_{\Sigma_1}) f(\sigma') \implies f(\sigma) \lfloor c_2 \rfloor f(\sigma')$ since $f(\sigma), f(\sigma') \in R_{\Sigma_1}$, and $\lfloor c_2 \rfloor \sqcup R_{\Sigma_1} \sqsubseteq R' \sqcup R_{\Sigma_1}$ by the induction hypothesis. Furthermore, $\forall \sigma, \sigma' \in (\blacksquare \backslash \blacksquare_{\Downarrow}) \cap \Sigma, \sigma R' \sigma' \implies \sigma \lfloor c_1 \rfloor \sigma'$ since we have shown above that $\lfloor c_1 \rfloor \sqcup R_\Sigma \sqsubseteq R' \sqcup R_\Sigma$. Hence, we obtain that $R_\Sigma \sqcup R_0 \sqsubseteq R_\Sigma \sqcup R' \implies R_\Sigma \sqcup R_0 \sqcup \lfloor c_1 \rfloor \sqsubseteq R_\Sigma \sqcup R' \sqcup \lfloor c_1 \rfloor = R_\Sigma \sqcup R'$. Now since $\lfloor c_1 \bullet c_2 \rfloor = \lfloor c_1 \rfloor \sqcup R_0$, then $R_\Sigma \sqcup \lfloor c_1 \bullet c_2 \rfloor \sqsubseteq R_\Sigma \sqcup R' \implies R_\Sigma \sqcup \lfloor c_1; c_2 \rfloor \sqsubseteq R_\Sigma \sqcup R'$ by applying proposition 1.

- Let P be `if` (b) `then` c_1 `else` c_2. Then it is clear from the definition that $\Sigma' = \{\sigma' \,|\, \sigma \in \Sigma, \langle$ `if` (b) `then` c_1 `else` Now define R_1 such that $\forall \sigma, \sigma' \in \Sigma, \sigma\, R_1\, \sigma' \iff \sigma(b) = \sigma'(b) = \mathbf{tt} \implies obs(c_1, \sigma) = obs(c_1, \sigma') \wedge \sigma(b) = \sigma'(b) = \mathbf{ff} \implies obs(c_2, \sigma) = obs(c_2, \sigma') \wedge \sigma(b) = \mathbf{tt}, \sigma'(b) = \mathbf{ff} \implies obs(c_1, \sigma) = obs(c_2, \sigma')$. Since $\uparrow_{\mathbf{TVar}} R = R$ and $R' = R \uplus \uparrow_{\mathbf{TVar}} R_1$, then we have by applying (**??**) and (**??**) of proposition **??** that $\uparrow_{\mathbf{TVar}} R' = R'$. It now remains to show that $R_\Sigma \sqcup \lfloor$`if` (b) `then` c_1 `else` $c_2 \rfloor \sqsubseteq R_\Sigma \sqcup R'$. It is clear that $R_\Sigma \sqcup \lfloor$`if` (b) `then` c_1 `else` $c_2\rfloor = R_1$, and we know that $R_1 \trianglelefteq R'$ and $dom(R_\Sigma) = dom(R_1)$. Thus, by applying (**??**) of proposition **??** we have that $R_\Sigma \sqcup \lfloor$`if` (b) `then` c_1 `else` $c_2 \rfloor \sqsubseteq R_\Sigma \sqcup R'$.

- Let P be `while` (b) `do` c and let $R_0 = R$ and $\Sigma_0 = \Sigma$ such that for all $i \geq 0$, `if` (b) `then` c `else` `skip`: $\langle \Sigma_i, R_i\rangle \rightarrow \langle \Sigma_{i+1}, R_{i+1}\rangle$. Then we have that $\Sigma' = \{\sigma \in \bigcup_{i\geq 0} \Sigma_i \,|\, \sigma(b) = \mathbf{ff}\}$. It is clear that $\Sigma' = \{\sigma' \,|\, \sigma \in \Sigma, \langle$`while` (b) `do` $c, \sigma\rangle \Downarrow \sigma'\}$. Now define $R'' \in PER(\blacksquare)$ such that $\forall \sigma, \sigma' \in \blacksquare, \sigma\, R''\, \sigma' \iff \sigma, \sigma' \in \Sigma'$. Then we have that $R' = \biguplus_{i\geq 0} R_i \uplus \uparrow_{\mathbf{TVar}} R''$. It is thus clear from (**??**) of proposition **??** that $\uparrow_{\mathbf{TVar}} R' = R'$. It now remains to be shown that $R_\Sigma \sqcup \lfloor$`while` (b) `do` $c\rfloor \sqsubseteq R_\Sigma \sqcup R'$.

  Now define $C_0 \triangleq$ `if` (b) `then` c `else` `skip` and for all $i \geq 1$, define $C_i \triangleq C_{i-1}; C_0$. Furthermore, let $\Sigma'' = dom(R')$ and take any $\sigma, \sigma' \in \Sigma$ such that $\sigma\, R'\, \sigma'$ holds. Then, by definition, we have that for all $i \geq 0$, $\sigma\, \mathcal{C}_{\Sigma''}(R_i)\, \sigma'$ and $\sigma\, \mathcal{C}_{\Sigma''}(\uparrow_{\mathbf{TVar}} R'')\, \sigma'$ hold. We first observe that R' relates any pair of states only if `while` (b) `do` c terminates under both states or diverges under both states. That is, can distinguish terminating traces from diverging ones. To see why, let $\Sigma_A = \{\sigma \in \Sigma \,|\, \langle$`while` (b) `do` $c, \sigma\rangle \Downarrow \sigma'\}$ be the subset of Σ under which `while` (b) `do` c terminates and let $\Sigma_B = \Sigma \backslash \Sigma_A$ be the subset of Σ under which it diverges. Since P does not modify the **IVar** projection of states, we know that $\Sigma_A \subseteq havoc\mathbf{TVar}(\Sigma')$. Furthemore, $\Sigma_B \cap havoc\mathbf{TVar}(\Sigma') = \varnothing$. But $havoc\mathbf{TVar}(\Sigma') = dom(\uparrow_{\mathbf{TVar}} R'')$ by applying (**??**) of proposition **??**, and since $\Sigma_B \subseteq \Sigma''$ we know also that $\Sigma_B \subseteq \Sigma'' \backslash dom(\uparrow_{\mathbf{TVar}} R'')$. This means that for any $\sigma, \sigma' \in \Sigma, \sigma\, \mathcal{C}_{\Sigma''}(R'')\, \sigma' \implies \sigma, \sigma' \in \Sigma_A$ or $\sigma, \sigma' \in \Sigma_B$ by the definition of $\mathcal{C}_{\Sigma''}(R'')$. Thus, any pair of states in Σ that are related by R' have the property that they both lead to the termination of P or they both lead to its divergence. This leaves us with only two cases to consider whenever a pair of states in Σ are related by R', namely the diverging and the terminating cases.

  Now suppose that $\sigma, \sigma' \in \Sigma$ and $\sigma\, R'\, \sigma$ holds and `while` (b) `do` c terminates under both σ and σ'. Then for all $i \geq 0$, we have that $obs(C_i, \sigma) =$ there exist $i, j \in \mathbb{N}$, such that $\langle C_i, \sigma\rangle \Downarrow \sigma_1$ and $\langle C_j, \sigma'\rangle \Downarrow \sigma_2$

  $\sigma_1 \uparrow_{\mathbf{TVar}} R'\, \sigma' \implies \sigma_1,$.

  Now suppose that `while` (b) `do` c terminates under σ and σ' ... TBC.

□