Automatically Calculating Quantitative Integrity Measures for Imperative Programs

Tom Chothia[1], Chris Novakovic[1], and Rajiv Ranjan Singh[2]

[1] School of Computer Science, University of Birmingham, Birmingham, UK
[2] Dept. of Computer Science, Shyam Lal College (Eve), University of Delhi, India

Abstract. This paper presents a framework for calculating measures of data integrity for programs in a small imperative language. We develop a Markov chain semantics for our language which calculates Clarkson and Schneider's definitions of data contamination and suppression. These definitions are based on conditional mutual information and entropy; we present a result relating them to mutual information, which can be calculated by a number of existing tools. We extend a quantitative information flow tool (CH-IMP) to calculate these measures of integrity and demonstrate this tool with examples based on error correcting codes, the Dining Cryptographers protocol and the attempts by a number of banks to influence the Libor rate.

1 Introduction

Data integrity is an important issue for programs, but this does not mean that programs must guarantee that all good data is perfectly preserved, or that no bad data can affect a program's output in any way. It is often the case that a good service can be provided with a small but acceptable loss in the the integrity of data. Therefore, there is a need for a framework in which the level of integrity a system provides can be accurately measured.

Qualitative integrity has been well-studied (see e.g. the work of Birgisson et al. [3], who provide a unified framework for qualitative integrity policies), but quantitative integrity measures have received far less attention. An exception to this is the work of Clarkson and Schneider, which defines the data integrity measures of "contamination" and "suppression" [8,9]. In this paper, we extend these measures to a small imperative language, give this language a Markov chain semantics, and show how these measures of integrity can be calculated for programs with multiple trusted and untrusted inputs and outputs. We extend an existing quantitative information flow (QIF) tool, CH-IMP [6], to use this semantics to automatically calculate these measures.

Clarkson and Schneider's definitions are based on conditional mutual information and entropy. Existing QIF tools can calculate the mutual information between a system's secret values and observable outputs; we investigate whether mutual information can be used to calculate conditional mutual information. We find that for secrecy, in which the public inputs of a system must be unrelated to the secret inputs, conditional mutual information can be calculated using mutual information (and therefore that existing QIF tools can calculate conditional mutual information-based definitions of secrecy by simply appending a program's public inputs to its public outputs). However, for integrity, the trusted data is considered to be public, and so the untrusted data sent to a system may depend on it, so conditional mutual information must be calculated explicitly.

The semantics we present makes it possible to calculate the integrity between multiple trusted and untrusted inputs and outputs. In doing so, we pay particular attention to what data is recorded by the semantics, and thus the probability distributions used to calculate the integrity measures. These decisions define the attacker model: CH-IMP models an attacker that wishes to learn anything they can about the secret values (including, e.g., the order in which they occur) and can only observe certain outputs from the system. For integrity, we consider a model in which we are only concerned that data is correctly preserved as it flows between variables. Thus, because of the different attacker models, the calculation of quantitative integrity is not the dual of the calculation of quantitative secrecy.

The contributions of this paper are:

- An extension of the CH-IMP semantics that can calculate the integrity measures defined by Clarkson and Schneider for multiple trusted and untrusted inputs and outputs in a simple imperative language.
- Showing that, when trusted/public inputs are unrelated to untrusted/secure inputs, measures based on conditional mutual information can be calculated in terms of mutual information.
- A software tool — the first of its kind — that automatically quantifies measures of integrity, and example programs.

In Section 2 we review related work, including the integrity definitions of Clarkson and Schneider and the CH-IMP framework. In Section 3 we show that definitions based on conditional mutual information can be rewritten in terms of mutual information, and can therefore be calculated more easily. In Section 4 we extend the CH-IMP semantics to calculate quantitative integrity for imperative programs. We implement this semantics and give some example programs in Section 5. Finally, we conclude in Section 6.

Our tool and a number of example programs are available on our website [14].

2 Background

2.1 Information Theory

The entropy (see e.g. [10]) of a random variable X with a probability mass function p is defined as

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x)$$

and is the average number of bits required to describe the result of the random variable. This extends to joint distributions in the obvious way: $H(X,Y) = -\sum_{x\in X}\sum_{y\in Y} p(x,y)\log_2 p(x,y)$.

The conditional entropy gives the amount of information required to describe the value of a random variable X if the value of a second random variable Y is known:

$$H(X|Y) = \sum_{x\in X}\sum_{y\in Y} p(x,y)\log_2 \frac{p(x)}{p(x,y)}$$

A natual extension of this is to define $H(X|Y,Z) = \sum_{x,y,z} p(x,y,z)\log_2 \frac{p(x)}{p(x,y,z)}$.

Mutual information is a measure of the information that one random variable contains about another:

$$I(X;Y) = H(X) - H(X|Y) = \sum_{x\in X}\sum_{y\in Y} p(x,y)\log_2 \frac{p(x,y)}{p(x)p(y)}. \tag{1}$$

Finally, conditional mutual information tells us how much information a random variable X contains about another random variable Y, given knowledge of a third random variable Z:

$$I(X;Y|Z) = H(X|Z) - H(X|Y,Z). \tag{2}$$

2.2 Quantification of Information Leakage and CH-IMP

QIF systems measure how much information an attacker learns about the high-level secret values in a system by observing low-level public values. Clark et al. [7] propose a standard model in which a system has high and low inputs and outputs, as depicted in Fig. 1. The information leakage in the system is then defined using conditional mutual information:

$$\text{Leakage} = I(H_{in}; L_{out}|L_{in})$$

i.e., the information the system leaks is the amount of information that an attacker (who knows the public inputs to the system) can learn about the high-level secret inputs to a system by observing the low-level public outputs.

More recently, a number of QIF tools have been developed that calculate the leakage from programs of varying sizes and complexities (e.g., [6,2,12]). However these tools calculate the simpler leakage measure of $I(H_{in}; L_{out})$; i.e., they do not explicitly model the attacker's knowledge of low inputs to the system.

Another popular measure of QIF is min-entropy leakage [13], which is given by $\log_2(\Sigma_l \max_h p(h,l)) - \log_2(\max_h p(h))$. This measure describes a system's resistance to correct guesses of the secret value in a single attempt. In this paper, we use definitions based on mutual information, but much of this work also applies to min-entropy-based definitions of integrity.

In previous work [6], we generalised the standard model of information leakage to allow secrets and observables to occur at any point in a program, and

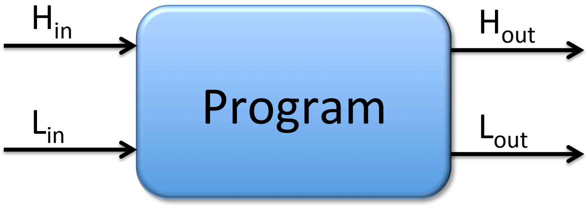

Fig. 1: Information leakage model of a program

$$
\begin{array}{rcl}
C & ::= & \textsf{new}\ V := \rho \\
 & \mid & V := \rho \\
 & \mid & \textsf{if}\ (B)\ \{\ C\ \}\ \textsf{else}\ \{\ C\ \} \\
 & \mid & \textsf{while}\ (B)\ \{\ C\ \} \\
 & \mid & C;\ C \\
 & \mid & \textsf{start} \\
 & \mid & \textsf{end} \\
 & \mid & \textsf{secret}\ V \\
 & \mid & \textsf{observe}\ V
\end{array}
$$

Fig. 2: The syntax of CH-IMP

we developed a tool (CH-IMP) to automatically calculate the leakage from a program.

The syntax of the CH-IMP language is given in Fig. 2. It is a simple imperative language with loops, variable declaration and assignment, and scope. It is a probabilistic language: the ρ in declaration and assignments is a probability distribution on expressions. The novel features of this language are the secret and observe commands,[3] which are used to tag high-level secret data and low-level observable data respectively. CH-IMP calculates how much information an attacker learns about the values tagged with secret by inspecting the values tagged with observe.

The semantics of CH-IMP are defined as a discrete-time Markov chain (DTMC) with states of the form $(C, \sigma, \mathcal{S}, \mathcal{O})$, where C is the program to be executed, σ is the current environment (mappings of variables to values), $\mathcal{S}$ are the secret variable mappings observed so far, and $\mathcal{O}$ is a list of the observed values. In order to support scope, an environment is a stack of sets of variable mappings, and the start and end commands push and pop elements onto and from this stack respectively.

[3] For simplicity, we write secret and observe as commands in the language but, as they have no effect on the state or control flow of a program, they may more accurately be considered annotations.

$$\frac{}{(\mathsf{new}\ V := \rho;\ C, o :: \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{\rho(n)} (C, (\{V \mapsto n\} \cup o) :: \sigma, \mathcal{S}, \mathcal{O})}$$

$$\frac{}{(\mathsf{secret}\ V;\ C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \sigma, \mathcal{S} :: (V \mapsto \llbracket V \rrbracket \sigma), \mathcal{O})}$$

$$\frac{}{(\mathsf{observe}\ V;\ C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \sigma, \mathcal{S}, \mathcal{O} :: \llbracket V \rrbracket \sigma)}$$

$$\frac{}{(V := \rho;\ C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{\rho(n)} (C, \sigma \oplus \{V \mapsto n\}, \mathcal{S}, \mathcal{O})}$$

$$\frac{\sigma(B) \to \mathbf{true}}{(\mathsf{if}\ (B)\ \{\ C_T\ \}\ \mathsf{else}\ \{\ C_F\ \};\ C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (\mathsf{start};\ C_T;\ \mathsf{end};\ C, \sigma, \mathcal{S}, \mathcal{O})}$$

$$\frac{\sigma(B) \to \mathbf{false}}{(\mathsf{if}\ (B)\ \{\ C_T\ \}\ \mathsf{else}\ \{\ C_F\ \};\ C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (\mathsf{start};\ C_F;\ \mathsf{end};\ C, \sigma, \mathcal{S}, \mathcal{O})}$$

$$\frac{\sigma(B) \to \mathbf{true}}{(\mathsf{while}\ (B)\ \{\ C_W\ \};\ C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (\mathsf{start};\ C_W;\ \mathsf{end};\ \mathsf{while}\ (B)\ \{\ C_W\ \};\ C, \sigma, \mathcal{S}, \mathcal{O})}$$

$$\frac{\sigma(B) \to \mathbf{false}}{(\mathsf{while}\ (B)\ \{\ C_W\ \};\ C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \sigma, \mathcal{S}, \mathcal{O})}$$

$$\frac{}{(\mathsf{start};\ C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \{\} :: \sigma, \mathcal{S}, \mathcal{O})}$$

$$\frac{}{(\mathsf{end};\ C, o :: \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \sigma, \mathcal{S}, \mathcal{O})}$$

Fig. 3: The semantic rules of CH-IMP.

The DTMC for a program $\mathcal{C}$ has the initial state $(C, \langle\rangle, \langle\rangle, \langle\rangle)$ and a probability transition matrix defined by the rules in Fig. 3. $s \xrightarrow{p} s'$ denotes the existence of a transition from state s to state s' with probability p, and $\sigma \oplus \{V \mapsto n\}$ denotes the replacement of the mapping for V with $V \mapsto n$ in the narrowest scope in σ already containing a mapping for V. The novel rules in the semantics are the rules for secret and observe; these rules record the high and low values that occur along a particular path of execution of the program.

A subtlety of the language is that it records variable mappings for secrets, but only the values of the public observables. This reflects our attacker model, in which the attacker wishes to learn *any* information about how the secret data is processed — in particular, if an attacker learns which secret variables are used (or in which order), but does not learn the values of those variables, then we

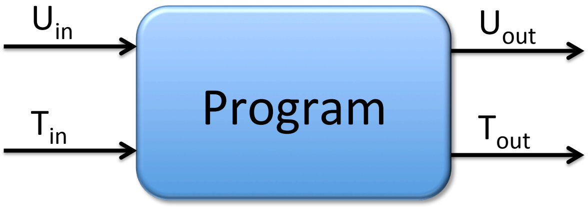

Fig. 4: Integrity measurement model of a program

consider that they have learnt some information about the secrets; the variable name must therefore be recorded in $\mathcal{S}$. However, the attacker only sees the values outputted by the program (not where or how these values were produced), so only the values of the observables are recorded in $\mathcal{O}$.

The final states of the DTMC of a terminating program give the probability distribution $p(\mathcal{S}, \mathcal{O})$ from which the information leakage measure $I(\mathcal{S}; \mathcal{O})$ can be calculated. The CH-IMP tool, source code and a range of examples are available at [14].

2.3 Quantification of Integrity

Clarkson and Schneider's work on the quantification of integrity [8,9] provides three definitions of integrity, although they do not address how to compute these definitions for programs. Their framework considers a program which has both trusted and untrusted inputs and outputs, as depicted in Fig. 4. Their intuition is that the untrusted data is bad data that an attacker may have added to the system, whereas the trusted input is the good data added by an honest party. A good system should minimise the effect that the untrusted inputs have on the trusted outputs, while at the same time preserving as much of the information from the trusted inputs as possible.

Clarkson and Schneider's first definition is *data contamination*, which they define as the amount of untrusted input data which can be learnt from the trusted output:

Definition 1 (Contamination). *The contamination in a system with untrusted input U_{in}, trusted input T_{in} and trusted output T_{out} equals $I(U_{in}; T_{out} | T_{in})$.*

We note that the measure $I(U_{in}; T_{out})$ would tell us how much information about the untrusted inputs is carried over to the trusted outputs; however, using this measure would produce misleading results in cases where the untrusted inputs were based on the trusted inputs. For instance, if the untrusted input was always an exact copy of the trusted input, and the trusted output was an exact copy of the trusted input, we would not consider that contamination was occurring.

The second measure Clarkson and Schneider define is *data suppression*; their intuition here is that the amount of data suppressed is the amount of data from the trusted inputs that the receiver cannot learn from the trusted outputs.

Definition 2 (Suppression). *The suppression in a system with trusted input T_{in} and trusted output T_{out} equals $H(T_{in}|T_{out})$.*

Their third measure, *program suppression*, compares a program's outputs to a specification of a program, and is beyond the scope of this paper. We note that none of these measures uses the untrusted outputs of a program, as these should not be taken into account when determining a program's integrity.

None of these measures make use of the untrusted output, which is included in Fig. 4 for completeness. This indicates that, when considering the integrity of a system, the value of any output of a system that we do not trust is irrelevant.

Clarkson and Schneider's work only provides the definitions of quantification of integrity; they do not address how the measures can be calculated for programs, and they only give single-line examples with single trusted and untrusted values in which the probability distributions on the trusted and untrusted values are clear, and so do not need to be calculated.

3 Using Mutual Information to Calculate Conditional Mutual Information and Entropy

CH-IMP, along with several other QIF tools (e.g. [6,2,12]), can calculate the mutual information between the input and output values of a program. In the next section, we consider how the models of integrity and secrecy differ for complex programs, but setting this aside it is natural to ask if these other tools can be used to calculate integrity measures directly; i.e., given a tool to calculate $I(X;Y)$, can we calculate contamination or suppression?

With regard to suppression ($H(T_{in}|T_{out})$), we note that Equation 1 gives us:

$$H(T_{in}|T_{out}) = H(T_{in}) - I(T_{in};T_{out}).$$

If the distribution of the trusted inputs is known, which we would expect, then $H(T_{in})$ can be calculated directly; we could then tag the trusted inputs to a program as secrets and the trusted outputs as observables, run the tool to calculate $I(T_{in};T_{out})$, and subtract this from $H(T_{in})$ to calculate suppression.

Contamination ($I(U_i;T_o|T_i)$) presents us with more of a problem. However, we note that:

$$\begin{aligned}
I(U_i;T_o|T_i) &= H(U_i|T_i) - H(U_i|T_o,T_i) && \text{by Equation 2} \\
&= H(U_i) + H(U_i|T_i) - H(U_i|T_o,T_i) - H(U_i) && \text{+ and - } H(U_i) \\
&= H(U_i) - H(U_i|T_o,T_i) - (H(U_i) - H(U_i|T_i)) && \text{rearrange terms} \\
&= I(U_i;T_o,T_i) - I(U_i;T_i) && \text{by Equation 1.}
\end{aligned}$$

If the untrusted inputs U_i and the trusted inputs T_i are completely unrelated then $I(U_i;T_i) = 0$, which in turn means that $I(U_{in},T_{out}|T_{in}) = I(U_{in};T_{in},T_{out})$;

$$
\begin{array}{rl}
C ::= & \mathsf{new}\ V := \rho \\
\mid & V := \rho \\
\mid & \mathsf{if}\ (B)\ \{\ C\ \}\ \mathsf{else}\ \{\ C\ \} \\
\mid & \mathsf{while}\ (B)\ \{\ C\ \} \\
\mid & C;\ C \\
\mid & \mathsf{start} \\
\mid & \mathsf{end} \\
\mid & \mathsf{untrustedin}\ V \\
\mid & \mathsf{trustedin}\ V \\
\mid & \mathsf{trustedout}\ V
\end{array}
$$

Fig. 5: The syntax of CH-IMP with integrity quantification (CH-IMP-IQ)

i.e., contamination in this case could be calculated by appending the trusted inputs to the trusted outputs and calculating the mutual information between this and the untrusted inputs. We state this as a theorem:

Theorem 1. *Contamination* $= I(U_i; T_o|T_i) = I(U_i; T_i, T_o) + I(U_i; T_i)$*, and, in the case that the untrusted inputs are independent of the trusted inputs, contamination* $= I(U_i; T_i, T_o)$*.*

However, the attacker's untrusted inputs may depend on the trusted inputs. There is no assumption of secrecy for the trusted values, and attacks on the integrity of a system may rely on an attacker sending carefully-crafted untrusted inputs based on the trusted inputs; e.g., the attacker might pick an input which, when combined with a trusted input, causes an overflow. In this case, $I(U_i; T_i, T_o)$ overestimates contamination by an amount equal to the amount of information about the untrusted inputs contained in the trusted inputs.

When calculating information leakage from a system, on the other hand, we explicitly require that the attacker does not know the value of the secret inputs ($High_{in}$), and therefore the attacker cannot pick low values (Low_{in}) based on these; i.e., $I(High_{in}; Low_{in}) = 0$. So, for secrecy, the conditional form with low inputs can be calculated directly by existing QIF tools:

Corollary 1. *When calculating information leakage, for which we have* $I(H_{in}; L_{in}) = 0$*, the standard measure of information leakage,* $I(H_{in}; L_{out}|L_{in})$ *is equal to* $I(H_{in}; L_{in}, L_{out})$*.*

Thus, existing QIF tools that calculate $I(High_{in}; Low_{out})$ can calculate the conditional mutual information measurement of leakage by simply appending the low-level inputs to the low-level outputs used to calculate leakage.

4 A Language for Integrity Checking

CH-IMP-IQ syntax. The syntax of CH-IMP with integrity quantification (CH-IMP-IQ) is given in Fig. 5. V ranges over variable names, B ranges over

Boolean expressions (i.e., evaluating to one of $\{\mathbf{true}, \mathbf{false}\}$), and ρ ranges over probability distributions on arithmetic expressions: variables, integers, or the result of evaluating two variables or integers with one of the standard arithmetic operations $\{+, \text{-}, *, /, \mathsf{mod}, \mathsf{xor}\}$.

The key new commands are untrustedin, trustedin and trustedout. These are used to label variables; we note that there is no label for untrusted outputs, as these are not needed to calculate contamination or suppression. An input in CH-IMP-IQ is not concrete; instead, it is selected from a probability distribution. For instance, Clarkson and Schneider use a one-line example program in their paper — $o_T := i_T$ xor j_U — in which o_T is the trusted output, i_T is the trusted input, and j_U is the untrusted input (i_T and j_U are chosen to be 0 or 1 uniformly). This example is equivalent to the following CH-IMP-IQ program:

```
new $i_T$ := { 0 ↦ 0.5, 1 ↦ 0.5 };
trustedin $i_T$;
new $j_U$ := { 0 ↦ 0.5, 1 ↦ 0.5 };
untrustedin $j_U$;
new $o_T$ := $i_T$ xor $j_U$;
trustedout $o_T$;
```

The CH-IMP-IQ model. The CH-IMP-IQ semantics calculates the probability distributions on the (un)trusted inputs and outputs. Furthermore, we would like to be able to analyse programs with many (un)trusted inputs and outputs, so we must consider which probability distributions should be calculated, and in particular whether we should consider the values of the variables (as CH-IMP does with observables) or the distribution on the mapping of variable names to values (as CH-IMP does with secrets). We illustrate this with the following example program (we use the shorthand trusted[in|out] new $V := \rho$ to mean new $V := \rho$; trusted[in|out] V):

```
trustedin new in1 := { 0 ↦ 0.5, 1 ↦ 0.5 };
trustedin new in2 := { 0 ↦ 0.5, 1 ↦ 0.5 };
new coin := { 0 ↦ 0.5, 1 ↦ 0.5 };
if (coin = 1) {
    trustedout new out1 = in1+1;
    trustedout new out2 = in2+1;
} else {
    trustedout new out2 = in2+1;
    trustedout new out1 = in1+1;
}
```

This program has two trusted inputs and it adds 1 to each of them. The order in which the inputs are incremented and marked as trusted outputs is decided by a random coin flip. We argue that this program does not suppress the integrity of its inputs: the order in which the outputs are declared should not matter, so the suppression of this program should be 0. However, if we consider only the values of the valuables marked as trusted outputs, we obtain the probability

$$\frac{}{(\mathsf{trustedin}\ V;\ C, \sigma, \mathcal{T}_{in}, \mathcal{U}_{in}, \mathcal{T}_{out}) \xrightarrow{1} (C, \sigma, \mathcal{T}_{in} \oplus (V \mapsto \mathcal{T}_{in}(V) :: [\![V]\!]\sigma), \mathcal{U}_{in}, \mathcal{T}_{out})}$$

$$\frac{}{(\mathsf{untrustedin}\ V;\ C, \sigma, \mathcal{T}_{in}, \mathcal{U}_{in}, \mathcal{T}_{out}) \xrightarrow{1} (C, \sigma, \mathcal{T}_{in}, \mathcal{U}_{in} \oplus (V \mapsto \mathcal{U}_{in}(V) :: [\![V]\!]\sigma), \mathcal{T}_{out})}$$

$$\frac{}{(\mathsf{trustedout}\ V;\ C, \sigma, \mathcal{T}_{in}, \mathcal{U}_{in}, \mathcal{T}_{out}) \xrightarrow{1} (C, \sigma, \mathcal{T}_{in}, \mathcal{U}_{in}, \mathcal{T}_{out} \oplus (V \mapsto \mathcal{T}_{out}(V) :: [\![V]\!]\sigma))}$$

Fig. 6: The integrity quantification semantics of CH-IMP-IQ

transition matrix on the left-hand side below, which implies a suppression value greater than 0.

	11	12	21	22
00	1	0	0	0
01	0	0.5	0.5	0
10	0	0.5	0.5	0
11	0	0	0	1

	{out1=1, out2=1}	{out1=1, out2=2}	{out1=2, out2=1}	{out1=2, out2=2}
{in1=0,in2=0}	1	0	0	0
{in1=0,in2=1}	0	1	0	0
{in1=1,in2=0}	0	0	1	0
{in1=1,in2=1}	0	0	0	1

To capture the behaviour we want, we need to consider mappings from the variables to the values taken by those variables as the elements of the probability distribution, as in the matrix on the right-hand side. This means that the model of integrity in CH-IMP-IQ is not *exactly* the dual of the model of secrecy in CH-IMP, because the different systems use different attacker models. For secrecy, we consider an attacker that only sees the output values of a program, and who would like to learn *any* information about the secrets, including the order in which they occur. For integrity, the observer should know which values relate to which trusted variables, and is only interested in the integrity of these values.

CH-IMP-IQ semantics. The semantics of a CH-IMP-IQ program $\mathcal{C}$ is defined as a DTMC (as with CH-IMP programs) with states describing the current path of execution, annotated with the information needed to compute integrity. They are of the form $(C, \sigma, \mathcal{T}_{in}, \mathcal{U}_{in}, \mathcal{T}_{out})$, where C, σ are the commands to be executed and the program state respectively, $\mathcal{T}_{in}$ is a set of mappings from variables to lists of values that have been marked as trusted inputs (indicated with the trustedin command), $\mathcal{T}_{out}$ is a set of mappings from variables to lists of values that have been marked as trusted outputs (indicated with the trustedout command), and $\mathcal{U}_{in}$ is a set of mappings from variable names to lists of values that have been marked as untrusted inputs (indicated with the untrustedin command).

The semantic rules for the new commands are given in Fig. 6; they track the values of the variables marked as containing trusted input, untrusted input and trusted output by appending their current values to the lists $\mathcal{T}_{in}$, $\mathcal{U}_{in}$ and $\mathcal{T}_{out}$ respectively. The rules for the remaining CH-IMP commands are unchanged from Fig. 3. The DTMC for a program $\mathcal{C}$ has the initial state $(C, \langle\rangle, \{\}, \{\}, \{\})$, and a probability transition matrix defined by these rules.

Calculating integrity. The final states of a terminating program[4] give us the probability distribution $p(t_{in}, u_{in}, t_{out})$. To calculate suppression, we calculate $p(t_{in}, t_{out}) = \sum_{u \in U_{in}} p(t_{in}, u, t_{out})$ and $p(t_{in}) = \sum_{t \in T_{out}} p(t_{in}, t)$, then, using the formula for conditional entropy:

$$H(T_{in}|T_{out}) = \sum_{t_i \in T_{in}} \sum_{t_o \in T_{out}} p(t_i, t_o) \log_2 \frac{p(t_i)}{p(t_i, t_o)}.$$

To calculate contamination, we calculate $p(u_i, t_i) = \sum_{t \in T_{out}} p(t_{in}, u_i, t)$ and then use Bayes' Theorem to expand the definition of conditional mutual information to give:

$$I(U_{in}, T_{out}|T_{in}) = \sum_{t_i \in T_{in}} \sum_{t_o \in T_{out}} \sum_{u_i \in U_{in}} p(u_i, t_o, t_i) \log_2 \frac{p(t_i)p(u_i, t_o, t_i)}{p(u_i, t_i)p(t_o, t_i)}.$$

5 Implementation and Examples

We have implemented the CH-IMP-IQ semantics to calculate contamination and suppression using approximately 1,000 lines of ML code (about two-thirds of which are shared with the CH-IMP implementation). This implementation calculates the measures precisely, modulo any rounding errors made by ML's floating point arithmetic.

The implementation builds the DTMC defined by the semantics, so its run-time is proportional to the state space. While the exact run-time will depend on the program being analysed, for a typical example the tool can analyse programs with 2^{12} states in seconds, and programs with 2^{22} states overnight. The source code and all of the examples from this section are available at [14].

Error correcting code. Our first example considers two simple error correction strategies. We first consider a program that broadcasts a 2 × 2 matrix in which each cell is either 1 or 0 with a probability of 0.5. However we assume that, for each cell, there is a probability that an error will occur and that the cell's value will be incremented by 1. In this case, the trusted inputs are the original cell values, the untrusted inputs are the occurrences of the errors, and the trusted outputs are the (possibly incremented) cell values.

The values of contamination and suppression for this program, with different probabilities of error, are graphed in Fig. 7(a) (where the dashed line is suppression and the solid line is contamination). We see that both the contamination and suppression values increase as the probability of an error increases, but as an error becomes more certain (and therefore predictable), they both decrease. This is because if we can be certain that an error will take place, we can treat a 2 as a 1, and a 1 as a 0, and the original information in the matrix is maintained.

[4] We only consider terminating programs in this paper; however, simpler methods than the ones we presented in [6] could be used to extend our definitions to non-terminating programs.

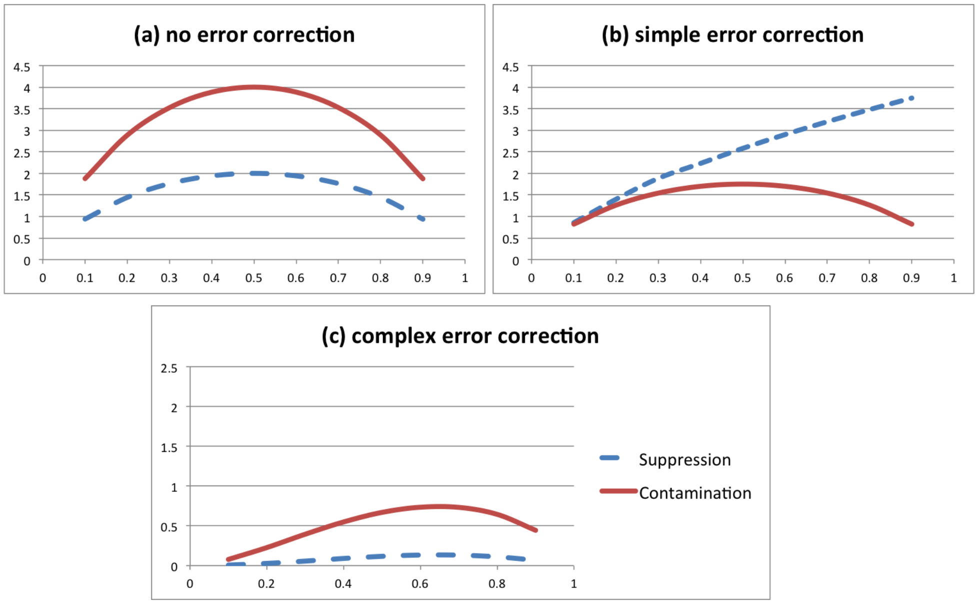

Fig. 7: Integrity measures for the cell errors

In this example, a 2 as the final value in one of the cells clearly indicates that an error has occurred; we could therefore attempt to correct for this error by replacing all 2s with 1s.

Running CH-IMP-IQ again, for a range of different probabilities of error, we find the values of suppression and contamination graphed in Fig. 7(b). We find that this error correction has decreased the contamination; i.e., the error value has a smaller effect on the result. However, the suppression value has increased, and indicates that now almost no useful data is received for high error rates. This is because the output of the system now contains less information: previously a cell value of 2 indicated that the original value was 1 and that an error had occurred. This error correction could therefore be considered counterproductive.

Finally, we consider a more complex error correction method that calculates the sum for each row or column, and then uses these sums to check for errors. After the errors, we add code that tries to correct the cell values based on the checksums. This code checks the post-error cell values and a correction is applied if one of the cells does not match the sum for the row and column. Again, the suppression and contamination measures for this code are graphed in Fig. 7(c) (note the different scale for this graph). Errors now only go uncorrected if they occur in pairs, so both suppression and contamination are much lower but still non-zero. We see that the information integrity of the data actually increases for high-error probabilities, again because errors become predictable and the information is preserved.

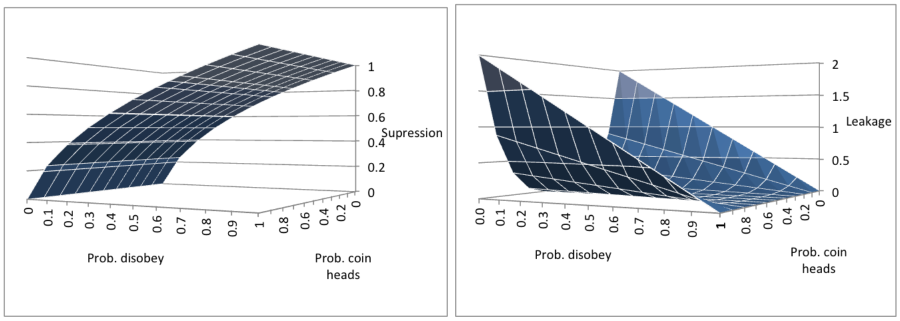

Fig. 8: Comparison of information leakage (via biased coins) and integrity loss (via payer disobedience) in the Dining Cryptographers protocol

The Dining Cryptographers protocol [4] is a popular example in the information flow literature. The protocol is motivated by the story of three cryptographers who, at the end of a meal, discover that the bill has been paid. They suspect that it was either anonymously paid by one of them, or that it was paid by the NSA. Not wanting to accept money from an organisation that may be performing mass surveillance, they need a protocol that allows one of them to communicate the fact that they paid, without revealing their identity. The protocol runs as follows: each adjacent pair of cryptographers flips a coin that only they see. The cryptographers therefore each see two coins, and they then publicly announce whether the coins agree or disagree — except for the payer (if any), who negates their answer. As the cryptographers are essentially computing the XOR of the coins, if there is an odd number of "disagree" announcements then one of the cryptographers paid, but if there is an even number of "disagree" announcements then the NSA paid.

The secrecy of the payer's identity in this protocol relies on the fairness of the coins. A popular example for QIF tools is to show that secrecy of the payer's identity decreases as the coins become more biased; however, the effect that biased coins have on the *integrity* of the protocol has not been studied. A CH-IMP-IQ model of this protocol is available on our website [14]. In this model, a variable *nsapaid* indicates whether the NSA is the payer; we tag this variable as the trusted input. The trusted outputs are the announcements by the cryptographers, and the untrusted inputs are the (possibly biased) coins. We can also calculate secrecy for this program using CH-IMP. Previous analysis of this protocol has shown that biased coins lead to the payer's identity being leaked, so we also consider how probabilistically disobeying the protocol may protect the payer: we introduce a probabilistic variable *obey*, and if this equals zero the payer will disobey the protocol and announce the true results of their coins.

We tested the secrecy and integrity of this system for a range of biased coins and probabilities of the payer disobeying the protocol; the results are shown in the surface plots in Fig. 8. The left-hand subfigure shows the suppression (a suppression of 0 bits means that the payer's identity is always correctly communicated). We see that making the coins biased has no effect on this value; i.e., biased coins do not affect integrity. On the other hand, as the probability of the payer disobeying the protocol increases, the amount of information suppressed rapidly increases.

The right-hand subfigure shows the amount of information leaked about the payer's identity when the probability of disobeying the protocol is zero: a small bias has little effect, but the leakage increases exponentially as the bias increases, leading to a total loss of anonymity for completely biased coins. We also see that disobeying the protocol increases the payer's anonymity, but only at a linear rate. This suggests that probabilistically disobeying the protocol is a poor defence for the payer.

Banks Fixing Libor. Our final example is inspired by the attempts of certain banks to influence the London Interbank Offered Rate (Libor) [1]. Libor is an estimate of the rate at which London banks lend money to each other. This measure is calculated by asking each bank to estimate the rate it would expect and averaging the responses. The rate affects payments on mortgages, student loans and other financial products; thus, any bank that can anticipate its value can make a lot of money. Problems with Libor were first spotted in 2008 [11] and it became evident in 2012 that some banks were constantly trying to fix Libor by under- or over-reporting their estimates [1].

We model a number of different strategies that a corrupt bank could use to fix Libor. Our model makes a number of simplifying assumptions: we assume there is a true rate, modelled as a CH-IMP-IQ variable *rate* (the trusted input) with an integer value selected uniformly between 1 and 4. We also assume that there are four banks that will estimate this value either exactly, or ± 1, with equal probability. Each of the four banks report their estimated rates, which can then be used to calculate Libor (the trusted output).

Perhaps the biggest simplifying assumption we make is that there is only one dishonest bank: one of the four banks will be randomly chosen as the corrupt bank, and a random variable will be used to decide if the bank will try to push the rate up or down; this variable is the untrusted input to the system. We write four different CH-IMP-IQ programs to reflect four ways in which the bank might try to affect the rate (all of which are available on our website):

1. No fixing: the bank does not make any changes, giving us a baseline to see how well the process estimates the true rate.
2. Maximum fixing: the bank will either set its rate to 1 or 4 depending on whether it is trying to increase or lower the rate.
3. ± 1: the bank will change its estimated rate by just 1 point within the range 1 to 4.
4. Average-based: the bank will follow the ± 1 strategy except when this would make the banks reported rate more than 1 point away from the average of the other bank's rates, in which case the corrupt bank will report its true rate.

As well as calculating the integrity of the reported rate, we declare the identity of the corrupt bank as secret and the reported rates as observable, and calculate how much information about the identity of the corrupt bank could be computed from the rates. The results are shown in the following table:

	No fixing	Max. fixing	± 1	Average-based
Suppression	0.487	1.083	0.687	0.648
Contamination	0	0.789	0.222	0.171
Leakage	0	1.32	0.153	0.108

We see that the "maximum fixing" strategy allows the bank to have a large effect on the rate, but the high leakage value indicates that the bank's actions will be obvious. The "± 1" strategy has a smaller effect and a smaller leakage value, indicating that it would be harder to spot than the "maximum fixing" strategy. The "average-based" strategy avoids the cases where the corrupt bank's estimate is suspiciously high or low, so we see a much lower leakage, with only a slightly smaller effect on the rate.

The strategy the dishonest banks actually employed was closest to the "± 1" strategy. In 2008, the Wall Street Journal spotted that banks were reporting rates that, at times, seemed too high or too low [11], and started the investigation into the Libor-rigging scandal. Our analysis suggests that if the banks had instead followed the "average-based" strategy it would have been much harder for reporters to have spotted their actions, and only have had a minor decrease in the affect they had over Libor.

6 Conclusion and Further Work

We have presented a framework that makes it possible to calculate quantitative measures of integrity from imperative programs. We have extended Clarkson and Schneider's definitions of data contamination and suppression to a semantics for a small imperative language, and we have implemented this language to produce the only currently available tool that can be used to quantify integrity.

As further work, we are interested in developing Clarkson and Schneider's definition of integrity with respect to specifications for imperative programs. This would involve a framework in which a user could write a specification program in CH-IMP-IQ, along with a more complex implementation program. The integrity of the implementation program would then be checked against the specification. While we have used Shannon entropy measures in this paper, most of this work would also apply to min-entropy-based measures, and as min-entropy leakage has proven to be a good measure of information leakage [13], it is worth investigating whether it also provides a good measure of integrity.

Due to their complexity and typical state space size, it would be difficult to extend the semantics we present in this paper to real programming languages, such as Java. However, in other work, we have investigated how statistical methods can be used to estimate information leakage measures from large, complex Java programs [5]; we would also like to investigate whether similar techniques can be used to estimate integrity in Java and other languages.

References

1. BBC: Libor scandal: Seven banks face us questioning. BBC News, 16 August (2012)
2. Biondi, F., Legay, A., Traonouez, L., Wasowski, A.: QUAIL: A Quantitative Security Analyzer for Imperative Code. In: Proc. of the 25th International Conference on Computer Aided Verification (CAV 2013) (2013)
3. Birgisson, A., Russo, A., Sabelfeld, A.: Unifying facets of information integrity. In: International Conference on Information Systems Security (ICISS) (2010)
4. Chaum, D.: The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. In: Journal of Cryptology. pp. 65–75 (1988)
5. Chothia, T., Kawamoto, Y., Novakovic, C.: Leakwatch: Estimating information leakage from java programs. In: European Symposium on Research in Computer Security (ESORICS) (2014)
6. Chothia, T., Kawamoto, Y., Novakovic, C., Parker, D.: Probabilistic Point-to-Point Information Leakage. In: Proc. of the 26th IEEE Computer Security Foundations Symposium (CSF 2013). pp. 193–205. IEEE Computer Society (Jun 2013)
7. Clark, D., Hunt, S., Malacaria, P.: Quantified interference for a while language. Electron. Notes Theor. Comput. Sci. 112, 149–166 (2005)
8. Clarkson, M.R., Schneider, F.B.: Quantification of integrity. In: Computer Security Foundations Symposium (CSF), 2010 23rd IEEE. pp. 28–43. IEEE (2010)
9. Clarkson, M.R., Schneider, F.B.: Quantification of integrity. Mathematical Structures in Computer Science. To appear (2014)
10. Cover, T.M., Thomas, J.A.: Elements of information theory. Wiley (2012)
11. Mollenkamp, C., Whitehouse, M.: Study casts doubt on key rate. The Wall Street Journal, 29 May (2008)
12. Mu, C., Clark, D.: A tool: quantitative analyser for programs. In: Proc.of the 8th Conference on Quantitative Evaluation of Systems (QEST) (2011)
13. Smith, G.: On the Foundations of Quantitative Information Flow. In: Proc. FOSSACS. pp. 288–302 (2009)
14. University of Birmingham: CH-IMP-IQ, `http://www.cs.bham.ac.uk/research/projects/infotools/chimp/iq`