Math. Struct. in Comp. Science (2015), *vol.* 25, *pp.* 207–258. © Cambridge University Press 2014
doi:10.1017/S0960129513000595 First published online 10 November 2014

Quantification of integrity[†]

MICHAEL R. CLARKSON[‡] and FRED B. SCHNEIDER[§]

[‡]*Department of Computer Science, Cornell University, Ithaca, NY, 14853, USA*
Email: clarkson@cs.cornell.edu
[§]*Department of Computer Science, Cornell University, Ithaca, NY, 14853, USA*
Email: fbs@cs.cornell.edu

Received 12 January 2011; revised 1 July 2012

Three integrity measures are introduced: contamination, channel suppression and program suppression. Contamination is a measure of how much untrusted information reaches trusted outputs; it is the dual of leakage, which is a measure of information-flow confidentiality. Channel suppression is a measure of how much information about inputs to a noisy channel is missing from the channel outputs. And program suppression is a measure of how much information about the correct output of a program is lost because of attacker influence and implementation errors. Program and channel suppression do not have interesting confidentiality duals. As a case study, a quantitative relationship between integrity, confidentiality and database privacy is examined.

1. Introduction

Many integrity requirements for computer systems are qualitative, but quantitative requirements can also be valuable. For example, a system might be permitted to combine data from trusted and untrusted sensors if the untrusted sensors cannot corrupt the result too much. And noise could be added to a database, thereby hiding sensitive information, if the resulting anonymized database still contains enough uncorrupted information to be useful for statistical analysis. Yet methods for quantification of *corruption* – that is, damage to integrity – have received little attention to date, whereas quantification of information leakage has been a topic of research for over 20 years (Denning 1982; Millen 1987).

We take two notions of corruption as points of departure:

— Bad information that is present in program outputs.
— Good information that is missing from program outputs.

The first leads us to a measure that we call 'contamination'. The second leads us to two measures that we collectively refer to as 'suppression'.

Figure 1 depicts contamination and suppression in terms of fluid flowing into and out of a bucket. An untrusted user adds black (bad) fluid; a trusted user adds white (good) fluid. Fluid destined to the trusted user exits the bucket through a pipe on the right. That fluid is grey, because it is a mixture of black and white. Quantity c of contamination

[†] Supported in part by ONR grant N00014-09-1-0652, AFOSR grant F9550-06-0019, NSF grants 0430161, 0964409 and CCF-0424422 (TRUST), and a gift from Microsoft Corporation.

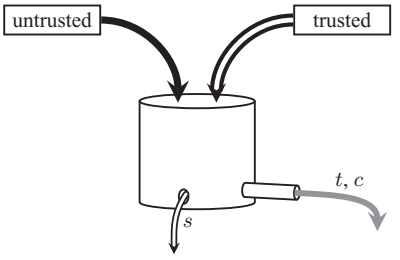

Fig. 1. Contamination, suppression and transmission.

is the amount of black fluid in that mixture, and quantity t, which we name 'transmission', is the amount of white fluid in the mixture. It might be possible to partially or fully separate the grey fluid back into its black and white components, depending upon how thoroughly they have been mixed in the bucket: the black fluid might dissolve so completely in the white that it is impossible to filter out again, or the black fluid might be insoluble in the white hence easy to filter. The bucket also has a hole near its bottom. Fluid escaping through that hole is not seen by any users. Quantity s of white fluid that escapes, which is shown in the figure, is the amount of suppression. Some black fluid might also escape through the hole, but that quantity is not depicted, because we are not interested in the amount of black fluid missing from the output pipe.

Contamination is closely related to *taint analysis* (Livshits and Lam 2005; Newsome and Song 2005; Suh *et al.* 2004; Wall *et al.* 1996; Xu *et al.* 2006), which tracks information flow from untrusted (tainted) inputs to outputs that are supposed to be trusted (untainted). Such flow results in what we call *contamination* of the trusted outputs. We might be willing to deem a program secure if it allows only a limited amount of contamination, but taint analysis would deem the same program to be insecure. So quantification of contamination would be useful.

Flow between untrusted and trusted objects was first studied by Biba (1977), who identified a duality between the models of integrity and confidentiality. The confidentiality dual to contamination is *leakage*, which is information flow from secret inputs to public outputs. Previous work has developed measures of leakage based on information theory (Clark *et al.* 2005b) and on beliefs (Clarkson *et al.* 2009). This paper adapts those measures to contamination. Through the Biba duality, we obtain a measure for corruption from a measure for leakage.

Suppression is connected to the program correctness, which is often phrased in terms of *specifications* and *implementations*. For a given input, an implementation should produce an output $\mathtt{o_I}$ that conveys an output $\mathtt{o_S}$ permitted by a specification. However, $\mathtt{o_I}$ and $\mathtt{o_S}$ need not be identical: an implementation might output all the bits in the binary representation of $\mathtt{o_S}$ but in a reverse order, or it might output $\mathtt{o_S}$ `xor` `k`, where `k` is a known constant. It suffices that knowledgeable users can recover $\mathtt{o_S}$ from $\mathtt{o_I}$.

The output of an incorrect implementation would fail to fully convey value $\mathtt{o_S}$. For example, an implementation might output only the first few bits of $\mathtt{o_S}$; or it might output $\mathtt{o_S}$ with probability p and output garbage with probability $1 - p$; or it might output $\mathtt{o_S}$ `xor` `u`, where `u` is an untrusted input. In each case, we say that *specification-violating*

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

suppression of information about the correct output value has occurred. Throughout this paper, we use a programming notation to write specifications (as well as implementations), so henceforth we use the more succinct term *program suppression* instead of *specification-violating suppression.*

Users might be willing to employ an implementation that produces sufficient information about the correct output value, hence exhibits little program suppression, even though a traditional verification methodology would deem the implementation to be incorrect. So quantification of program suppression would be useful.

The *echo* specification '`o := t`' gives rise to an important special case of program suppression. This specification stipulates that output `o` should be the value of input `t`, similar to the Unix `echo` command. For the echo specification, program suppression simplifies to the information-theoretic model of communication channels (Shannon 1948), in which a message is sent through a noisy channel. The receiver cannot observe the sender's inputs or the noise but must attempt to determine what message was sent. Sometimes, the receiver cannot recover the message or recovers an incorrect message. A noisy channel, for example, could be modelled by implementation '`o := t xor u`', in which noise `u` supplied as untrusted input by the attacker causes information about `t` to be lost. This loss of information represents *echo-specification violating suppression*, which for succinctness we henceforth call *channel suppression.*

This paper shows how to use information theory to quantify suppression, including how to quantify the attacker's influence on suppression. We start with channel suppression, then we generalize to program suppression. Applying the Biba duality to suppression yields no interesting confidentiality dual (see Section 4.2.4). So the classical duality of confidentiality and integrity was, in retrospect, incomplete.

We might wonder whether contamination generalizes suppression, or vice versa, but neither does. Consider the following three program statements, which take in trusted input `t` and untrusted input `u`, and produce trusted output `o`. Suppose that these programs are potential implementations of echo specification '`o := t`':

— `o := (t, u)`, where `(t, u)` denotes the pair whose components are `t` and `u`. This program exhibits contamination, because trusted output contains information derived from untrusted input `u`. The program does not exhibit suppression, because its output contains all the information about the value of `t`. A user of this program's output might filter out and ignore contaminant `u`, but that's irrelevant: in quantifying contamination, we are concerned only with measuring the amount of untrusted information in the output, not with what the user does with the output[†].

— `o := t xor n`, where the value of `n` is randomly generated by the program. This program exhibits suppression, because information about the correct output is lost. Suppression concerns that loss; suppression is not concerned with the presence of a

[†] Our definition of contamination is therefore consistent with Perl's *taint mode*, in which using the tainted data to affect the outside world is prohibited. Passing pair (t, u) to a system call that writes a file would be prohibited by Perl, because u is tainted.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

contaminant. In fact, this program cannot exhibit contamination, because it has no untrusted inputs.

— `o := t xor u`. This program exhibits contamination, because untrusted input `u` affects trusted output. This program also exhibits program suppression, because the noise of `u` causes information about the correct output to be lost.

So, although contamination and suppression both are kinds of corruption, they are distinct phenomena.

To illustrate our theory, we use it with two existing bodies of research. First, we revisit the work on *database privacy*. Databases that contain information about individuals are sometimes published in an anonymized form to enable statistical analysis. The goal is to protect the privacy of individuals yet still provide useful data for analysis. Mechanisms for anonymization suppress information – that is, integrity is sacrificed for confidentiality. Using our measure for channel suppression along with a measure for leakage, we are able to make this intuition precise and to analyse database privacy conditions from the literature.

Second, we revisit work on *belief-based information flow* (Clarkson *et al.* 2005, 2009). We give belief-based definitions of contamination and suppression. We also reexamine the relationship between the information-theoretic and belief-based approaches to quantifying information flow. We show that, for individual executions of a program, the belief-based definition is equivalent to an information-theoretic definition. And we show that, in expectation over all executions, the belief-based definition is a natural generalization of an information-theoretic definition.

We proceed as follows. Basic notions from information theory are used throughout the paper; Section 2 provides these definitions. Models for quantifying contamination and suppression are given in Sections 3 and 4. Database privacy is analysed in Section 5. Belief-based integrity is examined in Section 6. Related work is discussed in Sections 7 and 8. Some calculations are delayed from the main body to Appendix A, and all proofs appear in Appendix B.

This paper revises and expands a CSF 2010 paper (Clarkson and Schneider 2010), including the addition of (i) an improved model combining integrity and confidentiality, (ii) new results about database privacy and (iii) proofs, which were absent from the earlier paper.

2. Information theory review

This section reviews basic definitions from information theory used in the paper. More details can be found in any introductory text (e.g., Cover and Thomas (1991) and Jones (1979)). Readers familiar with this material might still want to scan this section to become acquainted with our notation.

The *self-information* (or simply *information*) $I(x)$ conveyed by a single event x that occurs with probability $\Pr(x)$ is defined as follows:

$$I(x) \triangleq -\log \Pr(x). \tag{1}$$

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

The base of the logarithm determines the unit of measurement for information. We assume base 2 for all logarithms, so the unit of measurement is bits. In effect, $I(x)$ quantifies how surprising event x is:

— The information conveyed by an event that is certain (i.e., an event with probability 1) is 0 – such an event is completely unsurprising.
— As the probability of an event approaches 0, the information conveyed by it approaches infinity, because the event becomes infinitely surprising. The quantity of information conveyed by an impossible event (i.e., probability 0) is undefined.
— If $\Pr(x) > \Pr(y)$, then $I(x) < I(y)$, because the occurrence of event x is less surprising than event y.
— If x and y are independent events, the information conveyed by the occurrence of both x and y is $I(x) + I(y)$. The surprise of x occurring is unaffected by whether y occurs, and vice versa.

The *conditional information* $I(x|y)$ conveyed by event x, given that event y has occurred, is the information conveyed by an event with conditional probability $\Pr(x|y)$:

$$I(x|y) \triangleq -\log \Pr(x|y). \tag{2}$$

The *mutual information* $I(x, y)$ between events x and y is the quantity of information that the two events have in common – that is, the information about x conveyed by y, or symmetrically, the amount of information about y conveyed by x. So, we would expect the following equalities to hold:

$$I(x, y) = I(x) - I(x|y) \tag{3}$$

$$= I(y) - I(y|x). \tag{4}$$

In Equation (3), $I(x)$ is how much information could possibly be obtained about x, and $I(x|y)$ is the amount remaining to obtain after observing y; the difference between these two quantities is the amount actually obtained about x by observing the occurrence of y. Equation (4) is symmetric. Since $\Pr(x, y) = \Pr(x|y) \cdot \Pr(y)$, mutual information $I(x, y)$ can also be expressed as follows:

$$\begin{aligned} I(x, y) &= I(x) - I(x|y) \\ &= -\log \Pr(x) + \log \Pr(x|y) \\ &\triangleq -\log \frac{\Pr(x)\Pr(y)}{\Pr(x, y)}. \end{aligned} \tag{5}$$

We take that last expression as the definition of mutual information between events. Note that if x and y are independent, their mutual information is 0.

Generalizing to distributions of events, the mutual information $\mathcal{I}(X, Y)$ between two distributions X and Y is the expected amount of information between all events $x \in X$ and $y \in Y$:

$$\begin{aligned} \mathcal{I}(X, Y) &\triangleq \mathsf{E}[I(X, Y)] \\ &= -\sum_{x,y} \Pr(x, y) \log \frac{\Pr(x)\Pr(y)}{\Pr(x, y)}. \end{aligned} \tag{6}$$

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

This definition requires a joint distribution of which X and Y are marginals. Operator E denotes expectation. By convention, $0 \log 0 = 0$ in this summation (and throughout this paper). Again note that if X and Y are independent, their mutual information is 0.

To conclude our development, let IN be a distribution of inputs to a channel; and OUT, of outputs. Then $\mathcal{I}(IN, OUT)$ is the expected amount of information that can be obtained about the inputs by observing the outputs.

Finally, an extension to mutual information will turn out to be useful. Sometimes auxiliary knowledge about channels is available – for example, a channel might be known to be noisier during daytime than during night. *Conditional mutual information* can model such knowledge. The conditional mutual information $I(x, y \mid z)$ – note that comma binds tighter than bar in this notation – between events x and y given the occurrence of auxiliary event z is defined like $I(x, y)$, but with all probabilities conditioned on z:

$$I(x, y \mid z) \triangleq -\log \frac{\Pr(x|z)\Pr(y|z)}{\Pr(x, y \mid z)}. \tag{7}$$

And conditional mutual information $\mathcal{I}(X, Y \mid Z)$ between distributions X and Y, given distribution Z, is again an expectation (and again requires a joint distribution of which X, Y and Z are marginals):

$$\begin{aligned} \mathcal{I}(X, Y \mid Z) &\triangleq \mathsf{E}[I(X, Y \mid Z)] \\ &= -\sum_{x,y,z} \Pr(x, y, z) \log \frac{\Pr(x|z)\Pr(y|z)}{\Pr(x, y \mid z)}. \end{aligned} \tag{8}$$

The *Shannon entropy* (or simply *entropy*) $\mathcal{H}(X)$ of a distribution X is the expected self-information[†] conveyed by the events of X:

$$\begin{aligned} \mathcal{H}(X) &\triangleq \mathsf{E}[I(X)] \\ &= -\sum_{x \in X} \Pr(x) \log \Pr(x). \end{aligned} \tag{9}$$

Entropy is always at least 0 and is maximized by uniform distributions. For example, the entropy of the uniform distribution of a space of 2^{32} events is 32 bits – the same number of bits as required to store a 32-bit integer.

The *joint entropy* $\mathcal{H}(X, Y)$ of two distributions X and Y is the expected amount of information conveyed by the occurrence of an event from their joint distribution:

$$\mathcal{H}(X, Y) = -\sum_{x \in X, y \in Y} \Pr(x, y) \log \Pr(x, y).$$

If X and Y are independent, joint entropy $\mathcal{H}(X, Y)$ is simply $\mathcal{H}(X) + \mathcal{H}(Y)$. If they are instead dependent, observing one might yield information about the other. The *conditional entropy* $\mathcal{H}(X|Y)$ is the expected amount of information conveyed by the occurrence of an

[†] For consistency, a better notation for the entropy of X might be $\mathcal{I}(X)$ – cf. Definition (6), where $\mathsf{E}[I(X, Y)]$ is equated with $\mathcal{I}(X, Y)$. But $\mathcal{H}(X)$ is the traditional notation for entropy.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

event from X given knowledge of what event from Y has occurred:

$$\mathcal{H}(X|Y) \triangleq -\sum_{x\in X, y\in Y} \Pr(x,y)\log\Pr(x|y). \tag{10}$$

An equivalent formulation of conditional entropy can be obtained by conditioning first on a single event, then taking an expectation over all events:

$$\mathcal{H}(X|y) = -\sum_{x\in X} \Pr(x|y)\log\Pr(x|y),$$
$$\mathcal{H}(X|Y) = \mathsf{E}_{y\in Y}[\mathcal{H}(X|y)]. \tag{11}$$

The joint entropy of X and Y can also be expressed as the amount of information obtained by observing X, then observing Y given knowledge of X; or vice versa:

$$\mathcal{H}(X,Y) = \mathcal{H}(X) + \mathcal{H}(Y|X) = \mathcal{H}(Y) + \mathcal{H}(X|Y).$$

Mutual information is related to entropy:

$$\mathcal{I}(X,Y) = \mathcal{H}(X) - \mathcal{H}(X|Y) = \mathcal{H}(Y) - \mathcal{H}(Y|X). \tag{12}$$

$\mathcal{H}(X)$ is how much (expected) information could be obtained about X, and $\mathcal{H}(X|Y)$ is the amount remaining to obtain after observing Y. The difference between these two quantities is the amount obtained about X by observing Y.

3. Quantification of contamination

Three agents are involved in our model of program execution: a system, a user and an attacker. The *system* executes the program, which has variables categorized as *input*, *output*, or *internal*. Input variables may only be read by the system, output variables may only be written by the system and internal variables may be read and written by the system but may not be observed by any agent except the system. The *user* and the *attacker* supply inputs by writing the initial values of input variables. These agents receive outputs by reading the final values of output variables. The attacker is untrusted, whereas the user is trusted.

Our goal is to quantify the information from untrusted inputs that contaminates trusted outputs. This goal generalizes taint analysis, which just determines whether any information from untrusted inputs contaminates trusted outputs. We accomplish our goal by quantifying the information the user learns about untrusted inputs by observing trusted inputs and outputs.

Informal definition: Contamination *is the amount of information a user learns about untrusted inputs by observing trusted inputs and outputs.*

Our use of terms 'learning' and 'observation' might suggest leakage of secret information. This is deliberate. We seek a definition of integrity that is dual to confidentiality. As we show in Section 3.4, our approach turns out to be dual to the technique of Clark *et al.* (2005b, 2007) for quantifying leakage[†].

[†] Readers familiar with Clark *et al.* (2005b, 2007) will be unsurprised by our final definition of expected contamination in Equation (18) and by the development leading up to it. We present the full development

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

Fig. 2. The contamination model.

The definition of contamination engenders two restrictions on the user's access to variables. First, the user may not directly read untrusted inputs. Otherwise, we would be quantifying something trivial – the amount of information the user learns about untrusted inputs by observing untrusted inputs. Second, the user may not read untrusted outputs, because we are interested only in the information the user learns from trusted outputs. In addition to these restrictions, we do not allow the user to write untrusted inputs. So the user may access only the trusted variables. Similarly, the attacker may access only the untrusted variables[‡]. These access restrictions agree with the Biba integrity model (Biba 1977): they prohibit 'reading up' (the user cannot read untrusted information) and 'writing down' (the attacker cannot write trusted information). The resulting communication model for contamination is depicted in Figure 2.

3.1. *Contamination in single executions*

One goal of information theory is to explain the behaviour of *channels*. A program, like a channel, accepts inputs and produces outputs. So, information flow can be quantified by modelling a program as a channel and using information theory to derive the amount of information transmitted over the channel[§].

A channel's inputs are characterized by a probability distribution of individual *input events*. Channels might be noisy and introduce randomness into *output events*, so a channel's outputs are also characterized by a probability distribution. Let t_{in}, u_{in} and t_{out} denote trusted input, untrusted input and trusted output events. (Each event may comprise the values of several input or output variables.) We assume a joint probability distribution of these events, and we let T_{in}, U_{in} and T_{out} denote the marginal probability distributions of trusted inputs, untrusted inputs and trusted outputs. Distribution T_{out} could alternatively be defined in terms of T_{in}, U_{in}, and some representation of the channel – for example, if the channel is represented as a probabilistic program, the denotational semantics of that program describes how to calculate T_{out} (Kozen 1981).

because it illuminates each step through the lens of integrity (rather than confidentiality), thus increasing confidence in our definitions. It also makes this paper self-contained.

‡ Flows from trusted to untrusted need not be prohibited. The attacker could be allowed to read trusted inputs or outputs, and the user could be allowed to write untrusted inputs. An attacker who reads trusted inputs might adaptively choose untrusted inputs to increase contamination; the joint probability distribution on inputs will characterize this adaptivity.

§ A consequence of using information theory to quantify information flow is that computational constraints on attackers are ignored; the security of cryptographic primitives such as encryption and hash functions therefore cannot be adequately characterized (Laud 2001; Backes 2005; Volpano 2000). Nonetheless, information theory is widely used to quantify information flow (see Section 7.1).

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

Mutual information characterizes the quantity of information that can be learned about channel inputs by observing outputs. $I(u_{in}, t_{out})$ denotes the mutual information between events u_{in} and t_{out} – that is, the amount of information either event conveys about the other. Note that $I(\cdot,\cdot)$ is mutual information between single events, not the more familiar mutual information between distributions of events. $I(u_{in}, t_{out} \mid t_{in})$ denotes the mutual information between events u_{in} and t_{out}, conditioned on the occurrence of event t_{in}.

The quantity $\mathcal{C}_1$ of contamination of trusted outputs by untrusted inputs in a single execution, given the trusted inputs, is defined as follows:

$$\mathcal{C}_1 \triangleq I(u_{in}, t_{out} \mid t_{in}). \tag{13}$$

(The subscript 1 is a mnemonic for 'single'.)

Consider the following program:

$$\mathtt{o_T} := \mathtt{i_U}\ \mathtt{xor}\ \mathtt{j_T} \tag{14}$$

Suppose that variables $\mathtt{o_T}$, $\mathtt{i_U}$ and $\mathtt{j_T}$ are one-bit trusted output, untrusted input and trusted input, respectively, and that the values of $\mathtt{i_U}$ and $\mathtt{j_T}$ are chosen uniformly at random. Intuitively, the user should be able to infer the value of $\mathtt{i_U}$ by observing $\mathtt{j_T}$ and $\mathtt{o_T}$, hence there is 1 bit of contamination. And according to Definition (13) of $\mathcal{C}_1$, the quantity of contamination caused by program (14) is indeed 1 bit. For example, the calculation of $I(\mathtt{i_U} = 0, \mathtt{o_T} = 1 \mid \mathtt{j_T} = 1)$ proceeds as follows:

$$\begin{aligned} I(\mathtt{i_U} = 0, \mathtt{o_T} = 1 \mid \mathtt{j_T} = 1) &= -\log \frac{\Pr(\mathtt{i_U} = 0 \mid \mathtt{j_T} = 1)\Pr(\mathtt{o_T} = 1 \mid \mathtt{j_T} = 1)}{\Pr(\mathtt{i_U} = 0, \mathtt{o_T} = 1 \mid \mathtt{j_T} = 1)} \\ &= -\log \frac{(1/2)(1/2)}{1/2} \\ &= 1. \end{aligned}$$

And calculating $I(\mathtt{i_U} = a, \mathtt{o_T} = b \mid \mathtt{j_T} = c)$ for any a, b and c such that $b = a\ \mathtt{xor}\ c$ would yield the same contamination of 1 bit. If $b \neq a\ \mathtt{xor}\ c$, then the calculation would yield an undefined quantity because of division by zero. This result is sensible, because such a relationship among a, b and c is impossible with program (14).

Having defined the exact quantity of contamination in a single execution, we can extend that definition to characterize any statistic of contamination. For example, we might wish to quantify the maximum possible contamination, so that we can evaluate the worst possible influence an attacker could have. This quantity is straightforward to define: the maximum contamination resulting from any input, or any distribution of inputs, is

$$\max_{T_{in}, U_{in}} (\mathcal{C}_1),$$

where $\mathcal{C}_1$ (13) depends upon distributions T_{in} and U_{in}. If we instead wish to quantify the maximum possible contamination for a particular trusted input, the definition can be specialized to $\max_{U_{in}}(\mathcal{C}_1)$, where T_{in} is no longer quantified. In the rest of this section, we investigate two other definitions of contamination that also build upon $\mathcal{C}_1$.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

3.2. *Contamination in sequences of executions*

Given $\mathcal{C}_1$, which provides a means to quantify contamination for single executions, we can quantify the contamination over a sequence of single executions. As an example, consider the following program, where operator & denotes bitwise AND:

$$\mathtt{o_T} := \mathtt{i_U} \,\&\, \mathtt{j_T} \tag{15}$$

Suppose that the attacker chooses a value for untrusted input $\mathtt{i_U}$ and that the user is allowed to execute the program multiple times. The user chooses a potentially new value for trusted input $\mathtt{j_T}$ in each execution, but the single value for $\mathtt{i_U}$ is used throughout. Also, suppose that all variables are k bits and that $\mathtt{i_U}$ is chosen uniformly at random. Intuitively, the contamination from this program in a single execution is the number of bits of $\mathtt{j_T}$ that are set to 1. Thus, a user that supplies `0x0001` for $\mathtt{j_T}$ learns[†] the least significant bit of $\mathtt{i_U}$ (so there is 1 bit of contamination); `0x0003` yields the two least significant bits (2 bits of contamination), etc. But when a user executes the program twice, supplying first `0x0001` then `0x0003`, the user learns a total of only 2 bits, not 3 ($= 1 + 2$). Directly summing $\mathcal{C}_1$ for each execution provides only an inexact upper bound on the contamination.

To calculate the exact amount of contamination for a sequence of executions, note the following. The untrusted input is chosen randomly at the beginning of the sequence. Each successive execution enables the user to refine knowledge of that untrusted input. So each successive calculation of contamination should use an updated distribution of untrusted inputs, embodying the user's refined knowledge about the particular untrusted input chosen at the beginning of the sequence[‡]. Let U^ℓ be a random variable representing the user's accumulated knowledge in execution ℓ about the untrusted input event, and let t^ℓ_{out} and t^ℓ_{in} be the trusted input and output events in that execution. The distribution of $U^{\ell+1}$ is defined in terms of the distribution of U^ℓ:

$$\Pr(U^{\ell+1} = u_{in}) = \Pr(U^\ell = u_{in} \mid t^\ell_{out}, t^\ell_{in}). \tag{16}$$

So, the updated distribution is obtained simply by conditioning on the trusted input and output. This conditioning is repeated after each execution.

We thus obtain the following formula for the total contamination $\vec{\mathcal{C}}$ in a sequence of executions:

$$\vec{\mathcal{C}} = \sum_\ell I(u^\ell_{in}, t^\ell_{out} \mid t^\ell_{in}),$$

where u^ℓ_{in} is the untrusted input event in execution ℓ, and mutual information $I(\cdot)$ is calculated according to distribution U^ℓ on untrusted inputs.

Returning to program (15), initial distribution U^1 on $\mathtt{i_U}$ is uniform. But distribution U^2, obtained by supplying `0x0001` as the first input, is uniform over $\mathtt{i_U}$ that have the same least significant bit as $\mathtt{j_T}$. Thus, the user learns only one additional bit by supplying

[†] Recall that contamination is the amount of information a user learns about untrusted input by observing trusted input and output.

[‡] Readers familiar with the use of beliefs in quantification of information flow will recognize this distribution as representing a belief; we discuss this matter further in Section 6.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

`0x0003` in the second execution. The total contamination according to $\vec{\mathcal{C}}$ is exactly 2 bits for the sequence – which is what our intuition suggested.

3.3. *Contamination in expectation*

$\mathcal{C}_1$ quantifies contamination in a single execution. It could be used at runtime by an execution monitor (Schneider 2000) to constrain how much contamination occurs during a given program execution. We might, however, be interested in how much contamination occurs on average over all executions of a program – a quantity that might be conservatively bounded by a static analysis. We now turn attention to that quantity.

The expected quantity $\mathcal{C}$ of contamination of trusted outputs by untrusted inputs, given the trusted inputs, is the expected value of $\mathcal{C}_1$:

$$\mathcal{C} = \mathsf{E}[\mathcal{C}_1]. \tag{17}$$

$\mathsf{E}[\mathcal{C}_1]$ can be rewritten as the mutual information $\mathcal{I}(U_{in},\ T_{out} \mid T_{in})$ between distributions U_{in} and T_{out}, conditioned on observation of T_{in}. That yields our definition of expected contamination:[†]

$$\mathcal{C} \triangleq \mathcal{I}(U_{in}, T_{out} \mid T_{in}). \tag{18}$$

Definition (18) of $\mathcal{C}$ yields an operational interpretation of contamination. In information theory, the *capacity* of a channel is the maximum quantity of information, over all distributions of inputs, that the channel can transmit. Shannon (1948) proved that channel capacity enjoys an operational interpretation in terms of coding theory: a channel's capacity is the highest rate, in bits per channel use, at which information can be sent over the channel with arbitrarily low probability of error. Therefore, the maximum quantity of contamination should also be the highest rate at which the attacker can contaminate the user. We leave investigation of this interpretation as future work.

3.4. *Leakage*

Clark *et al.* (2005b, 2007) define quantity $\mathcal{L}$ of expected leakage from secret inputs to public outputs, given knowledge of public inputs, as follows:

$$\mathcal{L} \triangleq \mathcal{I}(S_{in}, P_{out} \mid P_{in}). \tag{19}$$

Distributions S_{in}, P_{out} and P_{in} are on secret inputs, public outputs and public inputs, respectively. The corresponding definition for quantity $\mathcal{L}_1$ of leakage in a single execution is

$$\mathcal{L}_1 \triangleq I(s_{in}, p_{out} \mid p_{in}), \tag{20}$$

where events s_{in}, p_{out} and p_{in} unsurprisingly are secret input, public output and public input.

Replacing 'untrusted' with 'secret' and 'trusted' with 'public' in Equation (18) yields Equation (19); the same is true of Equations (13) and (20). Contamination and leakage

[†] The equality of Equations (17) and (18) follows from the definitions of $\mathcal{C}_1$ (13) and $\mathcal{I}$ (8).

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

are therefore information-flow duals: their definitions are the same, except the ordering of security levels is reversed. For example, the definition of $\mathcal{C}$ conditions on T_{in}, which represents inputs provided by a user with a high security level (because the user is cleared to provide trusted inputs); whereas the definition of leakage conditions on P_{in}, which represents inputs provided by a user with a low security level (because the user is not cleared to read secret inputs). So Biba's qualitative duality for confidentiality and integrity (Biba 1977) extends to these quantitative models[†].

4. Quantification of suppression

We begin by discussing channel suppression, then we generalize channel suppression to program suppression[‡].

4.1. *Channel suppression*

To quantify channel suppression, we refine our model of program execution by replacing the user with two agents, a *sender* and *receiver*. The receiver, by observing the program's outputs, attempts to determine the inputs provided by the sender. For example, the sender might be a database, and the program might construct a web page using queries to the database; the receiver attempts to reconstruct information in the database from the incomplete information in the web page. Information that cannot be reconstructed has been suppressed.

Informal definition: Channel suppression *is the amount of information a receiver fails to learn about trusted inputs by observing trusted outputs.*

As with contamination, the program receives trusted inputs as the initial values of variables and produces trusted outputs as the final values of variables. But now the sender writes the initial values of trusted inputs, and the receiver reads the final values of trusted outputs. These are the only ways that the sender and receiver may access variables. We continue to model an attacker, who attempts to interfere with trusted outputs by writing the initial values of untrusted inputs. The attacker still may access only the untrusted variables[§]. This communication model for channel suppression is depicted in Figure 3.

We first define channel suppression for single executions. As with our model of contamination, let t_{in} and t_{out} be trusted input and trusted output events. Since $I(t_{in}, t_{out})$

[†] We expect the duality between contamination and leakage would extend to other models, too. For example, the dual of *attacker influence on leakage* (Heusser and Malacaria 2010) would seem to be attacker influence on contamination. That latter quantity could be defined in the same way we define attacker-controlled suppression in Section 4.2.2.

[‡] Recall (from Section 1) that both kinds of suppression characterize information lost because a specification is violated. There might be other kinds of suppression besides specification violating. We leave investigation of them as future work.

[§] As with contamination, flows from trusted to untrusted need not be prohibited. The attacker could be allowed to read trusted inputs or outputs, and the sender could be allowed to write untrusted inputs. An attacker who reads trusted inputs might adaptively choose untrusted inputs to increase suppression; the joint probability distribution on inputs can characterize this adaptivity.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

Fig. 3. The channel suppression model.

is the quantity of information obtained about trusted inputs by observing trusted outputs, $I(t_{in}, t_{out})$ is defined to be the quantity $\mathcal{CT}_1$ of *channel transmission* from the sender to the receiver in a single execution:

$$\mathcal{CT}_1 \triangleq I(t_{in}, t_{out}). \tag{21}$$

$I(t_{in}|t_{out})$ denotes the information conveyed by the occurrence of event t_{in}, conditioned on observation of the occurrence of t_{out}. Using Equation (3), we rewrite the right-hand side of Equation (21):

$$\mathcal{CT}_1 = I(t_{in}) - I(t_{in}|t_{out}). \tag{22}$$

$I(t_{in})$ is the quantity of information that the receiver could learn about the trusted input, and $I(t_{in}|t_{out})$ is what remains to be learned after the receiver observes the trusted output. So, $I(t_{in}|t_{out})$ is the quantity of information that failed to be transmitted†. Therefore, $I(t_{in}|t_{out})$ is the quantity $\mathcal{CS}_1$ of *channel suppression* in a single execution:

$$\mathcal{CS}_1 \triangleq I(t_{in}|t_{out}). \tag{23}$$

Although untrusted input u_{in} does not directly appear in Equations (21) or (23), $\mathcal{CT}_1$ and $\mathcal{CS}_1$ do not ignore the attacker's influence on channel suppression: trusted output t_{out}, which does appear, can depend on u_{in}. Also, recall that Definition (13) of contamination $\mathcal{C}_1$ conditions on t_{in}; Equations (21) and (23) do not, because the receiver cannot directly observe trusted input – unlike the user, who could in the contamination model.

We next define channel suppression in expectation. $\mathcal{I}(T_{in}, T_{out})$ denotes the mutual information between distributions T_{in} and T_{out}, and $\mathcal{H}(T_{in}|T_{out})$ denotes the entropy of distribution T_{in}, conditioned on observation of T_{out}. (As before, T_{in} and T_{out} are marginal probability distributions of trusted inputs and trusted outputs, based on an underlying joint distribution.) By taking the expectation of $\mathcal{CT}_1$ and $\mathcal{CS}_1$, we obtain the expected quantities of channel transmission $\mathcal{CT}$ and channel suppression $\mathcal{CS}$:‡

$$\mathcal{CT} \triangleq \mathcal{I}(T_{in}, T_{out}), \tag{24}$$

$$\mathcal{CS} \triangleq \mathcal{H}(T_{in}|T_{out}). \tag{25}$$

† Alternatively, the right-hand side of Equation (21) could be rewritten with Equation (4) as $I(t_{out}) - I(t_{out}|t_{in})$. Perhaps this formula could also yield a measure for integrity, were we interested in backwards execution of programs – that is, computing inputs from outputs.

‡ Equation (24) follows directly from Equations (6) and (21). Similarly, Equation (25) follows from (10) and (23). Note that expected channel suppression $\mathcal{CS}$ is defined using entropy $\mathcal{H}$, not using mutual information $\mathcal{I}$, even though channel suppression $\mathcal{CS}_1$ is defined using self-information I. This notational quirk is inherited from information theory and occurs because entropy – not mutual information – is the expectation of self-information (see footnote †).

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

These definitions account for the attacker's influence on channel transmission and channel suppression, because distribution T_{out} depends on the attacker's distribution U_{in} on untrusted inputs. Also, these definitions should yield an operational interpretation in terms of coding theory; we leave that interpretation as future work†.

As an example, consider the following program:

$$\mathtt{o_T} := \mathtt{i_T}\ \mathtt{xor}\ \mathtt{rnd}(1) \tag{26}$$

Variables $\mathtt{i_T}$ and $\mathtt{o_T}$ are one-bit trusted input and output variables. Program expression $\mathtt{rnd}(x)$ returns x uniformly random bits. Suppose that trusted input distribution T_{in} is uniform on $\{0,1\}$. Then channel transmission $\mathcal{CT}$ is 0 bits and channel suppression $\mathcal{CS}$ is 1 bit. These quantities are intuitively sensible: because of the bit of random noise added by the program, the receiver cannot learn anything about $\mathtt{i_T}$ by observing $\mathtt{o_T}$.

4.1.1. *Attacker-controlled channel suppression.* An attacker might be able to influence the quantity of channel suppression by maliciously choosing inputs, as in the following program:

$$\mathtt{o_T} := \mathtt{i_T}\ \mathtt{xor}\ \mathtt{j_U} \tag{27}$$

Variable $\mathtt{j_U}$ is a one-bit untrusted input. Suppose that untrusted input distribution U_{in} is uniform. Then program (27) exhibits the same behaviour as program (26): 0 bits of channel transmission and 1 bit of channel suppression. But the source of that channel suppression is different. For program (26), the source is program randomness; for program (27), it is the attacker. We now develop definitions that distinguish these two sources of suppression.

Let $\mathcal{CS}_P$ denote the quantity of channel suppression attributable solely to the program – that is, the quantity that would occur if the attacker's input were known to the receiver:

$$\mathcal{CS}_P \triangleq \mathcal{H}(T_{in} \mid T_{out}, U_{in}). \tag{28}$$

This definition differs from Definition (25) of channel suppression $\mathcal{CS}$ only by the additional conditioning on U_{in}, which has the effect of accounting for the attacker's untrusted inputs. Any remaining channel suppression must come solely from the program.

Define the quantity $\mathcal{CS}_A$ of channel suppression under the attacker's control as the difference between the maximum amount of channel suppression caused by the attacker's choice of U_{in} and the minimum (which need not be 0 because of channel suppression attributable solely to the program):

$$\mathcal{CS}_A \triangleq \max_{U_{in}}(\mathcal{CS}) - \min_{U_{in}}(\mathcal{CS}). \tag{29}$$

($\mathcal{CS}$ is a function of T_{out}, which is a function of U_{in}, so quantifying over U_{in} is sensible.)

For program (26), quantity $\mathcal{CS}_P$ of program-controlled channel suppression is 1 bit, and quantity $\mathcal{CS}_A$ of attacker-controlled channel suppression is 0 bits. The converse holds for program (27), which exhibits 0 bits of program-controlled channel suppression and 1 bit of attacker-controlled channel suppression.

† The basis of that interpretation would be the capacity of the channel from trusted inputs to trusted outputs (see Section 3.3).

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

The following program exhibits both attacker- and program-controlled channel suppression:

$$\mathtt{o2}_\mathtt{T} := \mathtt{i2}_\mathtt{T}\ \mathtt{xor}\ \mathtt{j2}_\mathtt{U}\ \mathtt{xor}\ \mathtt{rnd(1)} \tag{30}$$

All variables in program (30) are two-bit. One bit of program-controlled channel suppression $\mathcal{CS}_P$ is caused by the `xor` with `rnd(1)`. But the attacker controls the rest of the channel suppression. If the attacker chooses $\mathtt{j2}_\mathtt{U}$ uniformly at random, the channel suppression is maximized and equal to 2 bits; whereas if the attacker makes $\mathtt{j2}_\mathtt{U}$ a constant (e.g., always '00'), the channel suppression is the minimal 1 bit caused by `rnd(1)`. Calculating $\mathcal{CS}_A$ yields 1 $(= 2 - 1)$ bit of attacker-controlled channel suppression.

4.1.2. *Error-correcting codes.* An *error-correcting code* adds redundant information to a message so that information loss can be detected and corrected. One of the simplest error-correcting codes is the *repetition code* R_n (Adámek 1991), which adds redundancy by repeating a message n times to form a *code-word*. For example, R_3 would encode message 1 as code-word 111. The code-word is sent over a noisy channel, which might corrupt the code-word; the receiver reads this possibly corrupted *word* from the channel. For example, the sender might send code-word 111, yet the receiver could receive word 101. To decode the received word, the receiver can employ *nearest-neighbour decoding*: the nearest neighbour of a word w is a code-word c that is closest to w by the *Hamming distance*. (The nearest neighbour is not necessarily unique for some codes, in which case an arbitrary nearest neighbour is chosen.) Treating words as vectors, Hamming distance $d(w, x)$ between words n-bit words $w = w_1 w_2 \ldots w_n$ and $x = x_1 x_2 \ldots x_n$ is the number of positions i at which $w_i \neq x_i$. For the repetition code, nearest-neighbour decoding means that a word is decoded to the symbol that occurs most frequently in the word. For example, word 101 would be decoded to code-word 111, thus to message 1; but word 001 would be decoded to message 0.

Consider the following program `BSC`, which models the *binary symmetric channel* studied in information theory:

$$\mathtt{BSC}: \quad \mathtt{w} := \mathtt{m}\ \mathtt{xor}\ \mathtt{rnd}_p\mathtt{(n)}$$

Variable `m`, which contains a message, is an `n`-bit trusted input, and variable `w`, which contains a word, is an `n`-bit trusted output. Expression $\mathtt{rnd}_p\mathtt{(x)}$, in which p is a constant, returns `x` independent, random bits. Each bit is distributed such that 0 occurs with probability p and 1 occurs with probability $1 - p$. (So `rnd(x)`, used in program (26), abbreviates $\mathtt{rnd}_{0.5}(x)$.) Thus, each bit of input `m` has probability $1 - p$ of being flipped in output `w`.

Suppose that $\mathtt{n} = 1$ and that the distribution of trusted input `m` is uniform. Then, the probability that `BSC` outputs `w` such that $\mathtt{w} = \mathtt{m}$ holds is p. Using Definitions (23) and (25), we can calculate the channel suppression from `BSC`, both in single executions and in expectation:

$$\mathcal{CS}_1 = -\log p, \tag{31}$$

$$\mathcal{CS} = -(p \log p + (1 - p) \log(1 - p)). \tag{32}$$

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

Next, suppose that the sender and receiver employ repetition code R_3 with program BSC. The sender encodes a one-bit input m into three bits and provides those as input to BSC (so now n = 3). The receiver gets a three-bit output and decodes it to bit w. Denote this composed program as R_3(BSC). The probability that w = m holds is now $p^3 + 3p^2(1-p)$, which can be derived by a simple argument[†]. Denote that probability as q. Substituting q for p in Equations (31) and (32), we can calculate the channel suppression from R_3(BSC):

$$\mathcal{CS}_1 = -\log q,$$
$$\mathcal{CS} = -(q \log q + (1-q)\log(1-q)).$$

Whenever $p > \frac{1}{2}$, we have that

$$-\log q \leqslant -\log p,$$
$$-(q \log q + (1-q)\log(1-q)) \leqslant -(p \log p + (1-p)\log(1-p)).$$

So for any channel at least slightly biased toward correct transmission, the channel suppression from R_3(BSC) is less than the channel suppression from BSC, both in single executions and in expectation. We conclude that repetition code R_3 improves channel transmission. Although this conclusion is unsurprising, it illustrates that our theory of channel suppression suffices to re-derive a well-known fact from the coding theory.

4.1.3. *Channel suppression versus contamination.* Recall program (27), restated here:

$$\mathtt{o_T} := \mathtt{i_T}\ \mathtt{xor}\ \mathtt{j_U}$$

This program is essentially the same as program '`o := t xor u`' from Section 1. We previously analysed program (27) and determined that it exhibits 1 bit of channel suppression if T_{in} and U_{in} are uniform distributions on $\{0,1\}$. We can also analyse the program for contamination: $\mathtt{i_T}$ is supplied by a user, and $\mathtt{o_T}$ is observed by that same user. Calculating $\mathcal{C}$ yields a contamination of 1 bit, indicating that the user learns all the (untrusted) information in $\mathtt{j_U}$. So this program exhibits both contamination and channel suppression, as we argued in Section 1.

You might wonder how a program with a one-bit output can exhibit both 1 bit of contamination and 1 bit of channel suppression. The answer is that the contamination and suppression models differ in what is observable: if an agent can observe both $\mathtt{i_T}$ and $\mathtt{o_T}$, then the agent can deduce $\mathtt{j_U}$, yielding 1 bit of contamination. But if the agent can observe only $\mathtt{o_T}$, then the agent cannot deduce $\mathtt{i_T}$, yielding 1 bit of suppression. Suppression concerns loss of trusted information (here 1 bit of trusted information is lost), whereas contamination concerns injection of untrusted information (here 1 bit of untrusted information is injected).

Also, recall program (26), restated here:

$$\mathtt{o_T} := \mathtt{i_T}\ \mathtt{xor}\ \mathtt{rnd}(1)$$

[†] Decoded output w equals input m if exactly zero or one bits are flipped during transmission. Each bit is transmitted correctly with probability p and flipped with probability $1-p$. The probability that zero bits are flipped is thus p^3; the probability that a particular bit is flipped is $p^2(1-p)$; and there are three possible single bits that could be flipped. So the total probability of correct decoding is $p^3 + 3p^2(1-p)$.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

Fig. 4. The declassifier model.

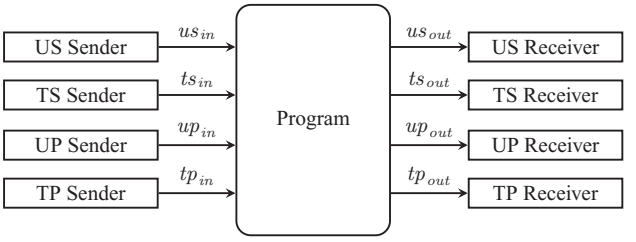

Fig. 5. The full model of channel suppression and leakage.

This program is essentially the same as program '`o := t xor n`' from Section 1. We previously determined that program (26) exhibits 1 bit of channel suppression. Because there are no untrusted inputs, quantity $\mathcal{C}$ of contamination is 0. So this program exhibits only channel suppression, as we argued in Section 1[†].

4.1.4. *Declassifiers.* Recall that leakage (20) in a single execution is the quantity $\mathcal{L}_1$ of information flow from secret input to public output. Leakage can be prevented by employing channel suppression. Consider a *declassifier* that accepts a trusted secret input ts_{in} and produces a trusted public output tp_{out}, as shown in Figure 4. The declassifier's task is to selectively release some secret information and suppress the rest. Whatever information is not leaked by the declassifier ought to have been suppressed.

That intuition is made formal by the following proposition. $I(ts_{in})$ denotes the self-information of event ts_{in}.

Proposition 1. *In the declassifier model,* $\mathcal{L}_1 + \mathcal{CS}_1 = I(ts_{in})$.

So for a given probability distribution of high inputs, leakage plus channel suppression is a constant in the declassifier model. Any information that enters the declassifier via ts_{in} must leave via tp_{out} or be obscured. Confidentiality is obtained by eroding integrity, and vice versa. Any security condition for declassifiers – we discuss some in Section 5 – that requires a minimum amount of confidentiality thereby restricts the maximum amount of integrity. And any utility condition that requires a minimum amount of integrity thereby restricts the maximum amount of confidentiality.

4.1.5. *Channel suppression and leakage.* The declassifier model of Section 4.1.4 included only one kind of input (trusted and secret) and one kind of output (trusted and public). More generally, programs might use all four combinations of {trusted, untrusted} × {secret,public} as inputs and outputs. That full model is depicted in Figure 5, which

[†] These arguments implicitly assume that random number generator `rnd(·)` is trusted. Untrusted generators could also be modelled, but we do not pursue that here.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

includes untrusted secret input us_{in}; trusted secret input ts_{in}; untrusted public input up_{in}; trusted public input tp_{in}; and corresponding outputs us_{out}, ts_{out}, up_{out} and tp_{out}. Each input or output is sent or received by a distinct agent. For example, ts_{in} is provided by an agent named 'TS Sender' in the figure, who is cleared to learn secret information and is trusted to provide high-integrity information. Similarly, up_{out} is received by an agent named 'UP Receiver' in the figure, who is cleared to learn only public information and does not require high-integrity information. Note that none of the receivers may directly read any of the senders' inputs; instead, the receivers access only the outputs of the program.

In this full model, we naturally would not expect Proposition 1 to directly hold, because information might be both leaked and suppressed simultaneously – that is, ts_{in} might flow to up_{out}. Nonetheless, in both models, any information that enters the program must leave or be obscured: any trusted inputs must be transmitted or suppressed, and any secret inputs must be leaked or kept hidden.

To formalize that intuition, first we define some events. As usual, let s_{in} denote a secret input event, p_{out} a public output event, p_{in} a public input event, t_{in} a trusted input event, and t_{out} a trusted output event. In the full model, each of these events is the joint occurrence of two finer-grained events:

— s_{in} is the joint event (ts_{in}, us_{in}) of both trusted and untrusted secret input.
— p_{out} is (tp_{out}, up_{out}).
— p_{in} is (tp_{in}, up_{in}).
— t_{in} is (tp_{in}, ts_{in}).
— t_{out} is (tp_{out}, ts_{out}).

Continuing with the formalization, define quantity $\mathcal{K}_1$ of secret information kept hidden in a single execution as follows:

$$\mathcal{K}_1 = I(s_{in} \mid p_{out}, p_{in}). \tag{33}$$

$\mathcal{K}_1$ is the amount of uncertainty remaining about secret inputs after observation of public outputs and inputs; the more uncertainty, the more information is kept hidden.

Finally, we formalize the intuition that any information entering the program must leave or be obscured. The sum of (i) the quantity of information that leaves the program ($\mathcal{CT}_1$ and $\mathcal{L}_1$) and (ii) that is obscured by the program ($\mathcal{CS}_1$ and $\mathcal{K}_1$) equals the sum of the quantity of information that enters via both trusted input and secret input.

Proposition 2. *In the full model,* $\mathcal{CS}_1 + \mathcal{L}_1 + \mathcal{CT}_1 + \mathcal{K}_1 = I(t_{in}) + I(s_{in})$.

Since the declassifier model is a special case of the full model, it is natural to expect that Proposition 1 would be a special case of Proposition 2. And it is:

Corollary 1. *Proposition 1 follows directly from specializing Proposition 2 to the declassifier model.*

Thus, the full model generalizes the declassifier model.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

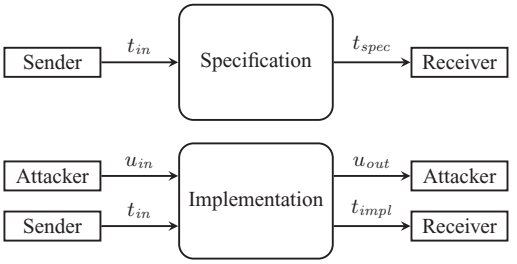

Fig. 6. The program suppression model.

4.2. *Program suppression*

We now generalize the idea of suppression from communication channels to the program correctness. Consider a *specification*, depicted in the top part of Figure 6: the specification receives a trusted input t_{in} from the sender and produces a *correct*, trusted output t_{spec} for the receiver. This idealized program does not interact with the attacker. But in the real world, an *implementation* that does interact with the attacker would be used to realize the specification. The implementation receives trusted input t_{in} from the sender and untrusted input u_{in} from the attacker; the implementation then produces untrusted output u_{out} for the attacker and trusted output t_{impl} for the receiver. A *correct implementation* would always produce the correct t_{spec} – that is, t_{impl} would equal t_{spec}. Incorrect implementations thus produce incorrect outputs, in part because they enable the attacker to influence the output.

In this model, the receiver observes t_{impl} but is interested in t_{spec}. So the extent to which t_{impl} informs the receiver about t_{spec} determines how much integrity the implementation has with respect to the specification. We can quantify this extent with information theory: *program transmission* is the amount of information that can be learned about t_{spec} by observing t_{impl}. Likewise, *program suppression* is the amount of information that t_{impl} fails to convey about t_{spec}.

Informal definition: Program suppression *is the amount of information a receiver fails to learn about the specification's trusted output by observing the implementation's trusted output*.

Let T_{spec} be the distribution on the specification's trusted outputs, and let T_{impl} be the distribution on the implementation's trusted outputs. These output distributions depend on trusted input distribution T_{in}, untrusted input distribution U_{in} (only for T_{impl}) and on the programs' semantics. Moreover, T_{spec} and T_{impl} are based on the same underlying trusted input – that is, the specification and the implementation are assumed to be executed with the same trusted input. We require T_{spec} to be a function of its input (non-functional specifications are discussed in Section 4.2.3):

$$\mathcal{H}(T_{spec}|T_{in}) = 0. \tag{34}$$

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

The definitions of program transmission and program suppression in single executions ($\mathcal{PT}_1$ and $\mathcal{PS}_1$) and in expectation ($\mathcal{PT}$ and $\mathcal{PS}$) are then as follows:

$$\mathcal{PT}_1 \triangleq I(t_{spec}, t_{impl}), \tag{35}$$

$$\mathcal{PS}_1 \triangleq I(t_{spec} | t_{impl}), \tag{36}$$

$$\mathcal{PT} \triangleq \mathcal{I}(T_{spec}, T_{impl}), \tag{37}$$

$$\mathcal{PS} \triangleq \mathcal{H}(T_{spec} | T_{impl}). \tag{38}$$

The rationale for these definitions remains unchanged from our development of channel transmission and suppression. Note that the attacker's influence is being incorporated, because T_{impl} can depend on U_{in}.

Channel transmission and suppression can now be seen as instances of program transmission and suppression for the echo specification, which stipulates that t_{spec} equal t_{in}. (This specification is deterministic and therefore satisfies Equation (34).) In Section 4.1, the output of the channel is called t_{out}, hence t_{impl} equals t_{out}. Given these equalities, we have that $T_{spec} = T_{in}$ and $T_{impl} = T_{out}$. Making these substitutions in the above definitions yields the definitions of channel transmission and channel suppression in single executions ($\mathcal{CT}_1$ and $\mathcal{CS}_1$) and in expectation ($\mathcal{CT}$ and $\mathcal{CS}$).

4.2.1. *Examples of program suppression.* Consider the following specification `SumSpec` for computing the sum of array `a`, which contains `m` elements indexed from 0 to `m` − 1:

```
SumSpec :  for (i = 0; i < m; i++)
             { s := s+a[i]; }
```

Assume throughout that `s` is initially 0.

Programmers frequently introduce off-by-one errors into loop guards. Such an error is exhibited by implementation `UnderSum`, which omits array element `a[0]`:

```
UnderSum :  for (i = 1; i < m; i++)
              { s := s+a[i]; }
```

Conversely, implementation `OverSum` adds `a[m]`, which is not an element of `a`:

```
OverSum :  for (i = 0; i <= m; i++)
             { s := s+a[i]; }
```

Suppose that array elements `a[0]..a[m-1]` are identically, independently distributed according to a binomial distribution with parameters n and p. Let $Bin(n, p)$ denote this distribution[†]. We consider elements `a[0]..a[m-1]` to be properly initialized and therefore trusted.

[†] A *binomial distribution* models the probability of the number of successes obtained in a series of n experiments, each of which succeeds with probability p. We choose this distribution because it enjoys a convenient summation property: if $X \sim Bin(n_x, p)$ and $Y \sim Bin(n_y, p)$, then $X + Y \sim Bin(n_x + n_y, p)$, where $Z \sim \mathcal{D}$ denotes that random variable Z is distributed according to distribution $\mathcal{D}$. Also, this distribution illustrates that our theory is not limited to uniform distributions.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

However, `a[m]` is not an element of the array, so it might have been initialized by the attacker; we therefore consider `a[m]` to be untrusted. To add as much entropy as possible to the sum, suppose that the attacker chooses `a[m]` to be uniformly distributed on integer interval $[0, 2^j - 1]$; let $Unif(0, 2^j - 1)$ denote this distribution.

`UnderSum` exhibits the following quantity $\mathcal{PS}_{\texttt{US}}$ of program suppression:

$$\mathcal{PS}_{\texttt{US}} = \sum_{\substack{s' \in Bin(n,p),\\ i \in Bin(n(m-1),p)}} \Pr(s')\Pr(i) \log \Pr(s'). \tag{39}$$

(The full calculation of $\mathcal{PS}_{\texttt{US}}$, as well as the calculations for Equations (40) and (41) below, appears in Appendix B.) So if $\texttt{m} = 10$, $n = 1$ and $p = 0.5$, then $\mathcal{PS}_{\texttt{US}}$ is 1 bit. This quantity is intuitively sensible: the implementation omits array element `a[0]`, which is distributed according to $Bin(1, 0.5)$, and the entropy of that distribution is 1 bit (because it assigns probability 0.5 to each of two values, 0 and 1). Moreover, this analysis suggests that `UnderSum` always exhibits program suppression equal to the entropy of the distribution on `a[0]`:

$$\mathcal{PS} = \mathcal{H}(Bin(n, p)). \tag{40}$$

Indeed, it is straightforward to reduce Equation (39) to Equation (40). Hence, `UnderSum` suppresses exactly the information about the omitted array element.

`OverSum` exhibits a different quantity $\mathcal{PS}_{\texttt{OS}}$ of program suppression:

$$\mathcal{PS}_{\texttt{OS}} = \sum_{\substack{s \in Bin(mn,p),\\ i' \in Unif(0,2^j-1)}} 2^{-j} \Pr(s) \log \frac{2^{-j} \Pr(s)}{\Pr(s + i')}. \tag{41}$$

Now if $\texttt{m} = 10$, $n = 1$, $p = 0.5$ and $j = 1$, then $\mathcal{PS}_{\texttt{OS}}$ is about 0.93 bits. Note that for this choice of parameters, all the array elements, including `a[m]`, are uniformly distributed on $\{0, 1\}$. The 1 bit of randomness added by the attacker through `a[m]` suppresses nearly 1 bit of information from the sum. The program suppression is not fully 1 bit because there are corner-case values that completely determine what the summands are – for example, if the sum is 0, then all array elements are 0 and the attacker's input is 0. If m were to increase while holding the other parameters constant, $\mathcal{PS}_{\texttt{OS}}$ would approach 1, because such corner cases occur with decreasing probability. So in the limit, the attacker can exploit memory location `a[m]` to suppress a single array element[†]. If j were to increase while the other parameters were held constant, $\mathcal{PS}_{\texttt{OS}}$ would approach 2.7 bits ($\approx \mathcal{H}(Bin(10, 0.5))$, because the noise added by the attacker would drown out the correct sum almost completely.

As another example of program suppression, consider the following specification:

$$\texttt{o}_{\texttt{T}} := 42$$

[†] This kind of analysis might be used to provide a mathematical explanation of why *failure-oblivious computing* (FOC) (Rinard *et al.* 2004) is successful at increasing software robustness. FOC rewrites out-of-bounds array reads to return strategically-chosen values that enable software to survive memory errors. Perhaps the choice of values could be understood as minimizing program suppression; we leave further investigation as future work.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

This specification represents a constant function: T_{spec} is the distribution assigning probability 1 to output 42. So quantity $\mathcal{PS}$ of program suppression is 0 bits, because the entropy of T_{spec} is 0 regardless of whether it is conditioned on T_{impl}, hence regardless of the implementation. Therefore, no implementation of a constant function exhibits program suppression.

As a final example, consider the following specification, in which $\texttt{lst}_\texttt{T}$ is a list of trusted values:

$$\texttt{o}_\texttt{T} := \mathit{sort}(\texttt{lst}_\texttt{T})$$

Implementation $\texttt{o}_\texttt{T} := \texttt{lst}_\texttt{T}$ exhibits no program suppression with respect to this specification, even though the implementation does not sort the list, because all the information needed to compute the correct output is contained in the implementation's output. Program suppression is information-theoretic, not functional. We leave exploration of more functional notions of suppression as future work.

4.2.2. *Attacker-controlled program suppression.* Attackers might influence the quantity of program suppression through choice of untrusted inputs. `OverSum` is one example. Other examples include the following:

— A search engine models a set of web pages as a graph in which nodes are pages and edges are links. Query results are ordered in part based on the number of incoming edges to each page in the graph. An attacker with control over some web pages creates many links to a particular page, causing it to be returned earlier in the query results. The correct ordering of query results has been suppressed by the attacker's influence on the web.
— An implementation contains a buffer overflow vulnerability that the attacker exploits by crafting an input containing malicious code. That code is executed and produces arbitrary new outputs that are unrelated to the specification. Those outputs cause the implementation to behave in malicious ways, such as deleting files, participating in a botnet, etc. The correct behaviour of the program has been suppressed, perhaps entirely, by the attacker's malware.

The equations defining attacker-controlled program suppression are simple adaptations of those defining attacker-controlled channel suppression in Section 4.1.1. The quantity $\mathcal{PS}_P$ of program suppression attributable solely to the implementation is the quantity of program suppression, but conditioned on knowledge of the attacker's untrusted inputs:

$$\mathcal{PS}_P \triangleq \mathcal{H}(T_{spec} \,|\, T_{impl}, U_{in}).$$

The quantity $\mathcal{PS}_A$ of program suppression under the attacker's control is the difference between the maximum amount of program suppression caused by the attacker's choice of U_{in} and the minimum:

$$\mathcal{PS}_A \triangleq \max_{U_{in}}(\mathcal{PS}) - \min_{U_{in}}(\mathcal{PS}).$$

Consider `OverSum`: the quantity of program suppression $\mathcal{PS}_P$ attributable solely to the implementation is 0 bits, because given knowledge of the attacker's untrusted input, no suppression occurs. But without that knowledge suppression does occur, as we calculated

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

above. Recall that as m increases, the quantity of program suppression approaches 1 bit. So, the quantity of program suppression $\mathcal{PS}_A$ under the attacker's control approaches 1 bit.

4.2.3. *Non-functional specifications.* Consider eliminating our requirement (34) that specifications be functional. We might instead allow probabilistic specifications, such as '$\mathtt{o_T} := \mathtt{rnd}(1)$'. It stipulates that the output must be 0 or 1, and that each output must occur with probability $\frac{1}{2}$. There is no correct output according to this specification; instead, there is a correct distribution on outputs. Program suppression should be the amount of information the receiver fails to learn about that correct distribution – rather than about a correct output – by observing the implementation.

To quantify that suppression with entropy, as we have done so far, it seems we would need an extra level of distributions: a probability distribution on a probability distribution on outputs. So far, we have modelled only discrete probability distributions, which have finite support. But there are infinitely many probability distributions on outputs, so it seems we would need to upgrade our model with continuous probability distributions and *differential entropy* (the continuous analogue of entropy).

Alternatively, we might quantify suppression with *relative entropy*, which measures the divergence between distributions. The relative entropy $D(Y \parallel X)$ between distributions Y and X is defined as follows:

$$D(Y \parallel X) \triangleq \sum_x \Pr_Y(x) \log \frac{\Pr_Y(x)}{\Pr_X(x)}, \tag{42}$$

where $\Pr_Z(z)$ denotes the probability of event z according to distribution Z. In the coding theory, $D(Y \parallel X)$ quantifies the inefficiency of a code that results from assuming that a distribution is X when in reality it is Y. By analogy, program suppression could be defined as $D(T_{spec} \parallel T_{impl})$, which is the inefficiency of assuming that the distribution on outputs is T_{impl}, as observed by the receiver, instead of T_{spec}, as stipulated by the specification.

Further justification for this definition of suppression could be obtained by rewriting the definition of relative entropy as follows:

$$\begin{aligned} D(Y \parallel X) &= \sum_x \Pr_Y(x) \log \frac{\Pr_Y(x)}{\Pr_X(x)} \\ &= \mathsf{E}_{x \in Y} \left[\log \frac{\Pr_Y(x)}{\Pr_X(x)} \right] \\ &= \mathsf{E}_{x \in Y} \left[\log \Pr_Y(x) - \log \Pr_X(x) \right] \\ &= \mathsf{E}_{x \in Y} \left[I_X(x) - I_Y(x) \right], \end{aligned}$$

where $I_Z(z)$ denotes the self-information (1) of event z according to distribution Z, and $\mathsf{E}_{z \in Z}[f(z)]$ denotes the expectation of $f(z)$ with respect to distribution Z. Term $I_X(x) - I_Y(x)$ is the additional surprise that results from assuming that a distribution is X when in reality it is Y. Thus, $D(T_{spec} \parallel T_{impl})$ is the expected additional surprise resulting from assuming that the distribution on outputs is T_{impl}, as observed by the receiver, instead of

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

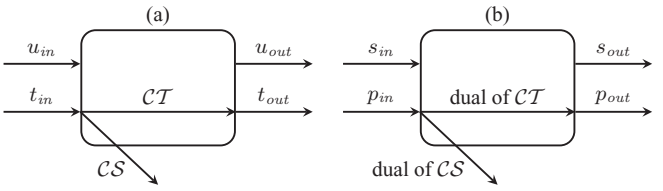

Fig. 7. (a) The channel suppression model, (b) dual of the channel suppression model.

T_{spec}, as stipulated by the specification. The greater that surprise, the more information has been suppressed by the implementation.

We might also allow nondeterministic specifications, such as "$o_T := 0 \,[]\, 1$". It stipulates that the output must be 0 or 1, but nothing more. There is no correct output according to this specification; instead, there is a correct set of outputs. We might even allow specifications that contain both probabilistic choice and nondeterministic choice. It could be possible to handle such specifications with the use of Dempster–Shafer belief functions (Shafer 1976), which assign probability to sets of possibilities.

We leave further investigation of non-functional specifications as future work.

4.2.4. *Duality*. Program suppression is the amount of information the implementation's trusted output fails to reveal about the trusted output that is correct according to the specification. Applying the Biba duality, the confidentiality dual of program suppression would be the amount of information that the implementation's public output fails to reveal about the public output that is correct according to the specification. For confidentiality, this flow is uninteresting: the amount of information that flows, or fails to flow, to public outputs does not characterize how a program leaks or hides secret information. So, there does not seem to be an interesting dual to suppression.

The lack of interesting duality is especially apparent in the case of the echo specification – that is, for channel suppression. Figure 7(a) depicts channel transmission and channel suppression. Channel transmission $\mathcal{CT}$ is information that flows from t_{in} to t_{out}, whereas channel suppression $\mathcal{CS}$ is information that flows in from t_{in} and is dropped by the program. Figure 7(b) depicts their duals. The dual of $\mathcal{CT}$ is information that flows from public inputs p_{in} to public outputs p_{out}. Since that flow does not involve secret inputs s_{in}, it is not interesting from the perspective of confidentiality. Likewise, the dual of $\mathcal{CS}$ is information that flows in from p_{in} and is dropped by the program. That flow does not involve secret inputs s_{in}, so is not interesting from the perspective of confidentiality.

Other notions of integrity also lack obvious confidentiality duals – for example, the Clark–Wilson (1987) integrity policy for commercial organizations, based on well-formed transactions and verification procedures. Apparently, the Biba duality goes only so far.

4.2.5. *Suppression versus availability*. Suppose that a program suppresses information during an execution; the program has corrupted its output and damaged integrity. But the receiver cannot acquire that suppressed information, at least not during that execution, so the program could also be said to exhibit compromised availability. Viewed through this lens, information integrity and information availability seem to be essentially the

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

Fig. 8. The anonymizer model.

same. So we cannot argue that suppression is about only one. Perhaps it is really about both. We suspect interesting relationships – perhaps even new dualities – are still to be found between availability and integrity.

However, *system availability* seems to be different than information integrity. System availability is generally concerned with reachability and timely response, not with quality of information. For example, execution of copies of a service on multiple machines improves system availability but potentially introduces program suppression: the different copies might provide different responses to the same request, and extracting a correct response from them might not be possible. Conversely, error-correcting codes defend against channel suppression but do not improve system availability – if a channel goes down (e.g., a wire is cut), a code cannot restore communication. So suppression is not about system availability.

5. Application: database privacy

Many organizations possess large collections of information about individuals – for example, government census data, hospital medical records, online product reviews, etc. Sharing that information with other organizations, including the public, enables research, facilitates decision making, and provides transparency. However, these collections typically contain *sensitive* information about individuals (e.g., medical diagnoses such as 'Alice suffers from hallucinations'). Publishing that sensitive information violates the *privacy* of individuals if they have not given consent. So organizations anonymize databases before publishing them, expecting that individuals' privacy is thereby maintained.

Algorithms that anonymize collections of information have been widely studied (Fung *et al.* 2010). The basic setup is shown in Figure 8. An *anonymizer* receives the contents of a database as input and produces an anonymized database as output; this output is made public[†]. The anonymizer must reveal some sensitive information about individuals, otherwise its output would be useless. But the anonymizer must also hide some of that information to protect privacy. So, there is an inherent tradeoff between the privacy and utility of the anonymized database.

That tradeoff can be quantified using leakage and suppression. Suppose the anonymizer receives input database d and produces anonymized database a as output. Input d is trusted because it originates with the assumedly trustworthy organization, and output a

[†] Note that the anonymized database might involve different domains than the original data, perhaps statistics (e.g., counts, sums, or averages) computed from individuals' information. We do not restrict our consideration to any particular statistics here.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

is trusted because it is computed by the assumedly trustworthy anonymizer. And input d is secret because it contains sensitive information, whereas output a is public because it is assumedly anonymized. Let p be a projection of d that contains exactly the non-sensitive information in the database. By the definition of quantity $\mathcal{L}_1$ of leakage (20), the amount L of sensitive information revealed by the anonymizer is $I(d, a \mid p)$. That amount is equal to quantity $\mathcal{CT}_1$ of channel transmission (21), because L is the amount of sensitive information transmitted by the anonymizer[†]. Similarly, the amount S of sensitive information hidden by the anonymizer is $I(d \mid a, p)$, which is the same as quantity $\mathcal{CS}_1$ of channel suppression (23) (except again for conditioning on p)[‡].

The amount of leakage L plus the amount of suppression S is a constant that depends on the distribution of database content. That tradeoff is expressed by the following corollary of Proposition 1.

Corollary 2. $L + S = I(d|p)$.

This is sensible – whatever the anonymizer does not suppress, it leaks. Designers of anonymizers thus have the opportunity to choose a point along this tradeoff between leakage and suppression[§].

Contamination is not relevant to this model of anonymizers: there is no information provided by an attacker as input, hence contamination must be zero. However, we could generalize the model to include attacker input – for example, attackers could contribute information to the database before anonymization. Beyond possibly contaminating the output of the anonymizer, that contribution might cause the anonymizer to leak more information that is sensitive. Hence, the attacker could influence the amount of information leaked by the anonymizer. This attack is another example of attacker-controlled suppression, which we discussed in Sections 4.1.1 and 4.2.2. However, for the analysis that follows, we will not allow attacker input to anonymizers. We leave further investigation of that to future work.

Many security conditions have been developed for anonymizers to characterize how well they protect privacy. In what follows, we apply our quantitative frameworks for integrity and confidentiality to some popular security conditions: k-anonymity (Samarati and Sweeney 1998), ℓ-diversity (Machanavajjhala *et al.* 2007), γ-amplification (Evfimievski *et al.* 2003) and ϵ-differential privacy (Dwork 2006). Rather than examine particular anonymization algorithms that enforce these conditions, we examine the conditions themselves, so that our results are applicable to all algorithms that achieve a given condition. We generalize each security condition to apply to information flow in programs, rather than the special case of anonymizers. And we offer an information-theoretic

[†] Note that Equation (21) does not condition on any information, because it measures transmission of the entire input. Here we condition on p to exclude it from measurement, because it is not sensitive.

[‡] Were we to use program suppression to analyse the anonymizer, the specification would be "$a := d$", which simplifies to channel suppression.

[§] This quantitative analysis could have been done with the confidentiality model alone. A confidentiality-only analysis would have shown that some information is leaked, and some is not. The contribution of the analysis here is to show that the information not leaked is the same as the information suppressed. This equality validates the integrity model.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

characterization of the generalized security condition in terms of how much sensitive information is suppressed or leaked. Those characterizations yield a quantitative basis for comparison of the security conditions.

5.1. *k-anonymity*

Samarati and Sweeney (1998) propose *k-anonymity*, a security condition for anonymizers that requires every individual to be anonymous within some set of k individuals[†]. For example, suppose that a database contains only gender and birth date. If Alice were born on 26 November, 1865, then to satisfy k-anonymity at least $k-1$ other females born that day must appear in the database. If fewer than $k-1$ appear, the data must be changed in some way. For that, Samarati and Sweeney propose *generalization*, which hierarchically replaces attribute values with less specific values. For example, Alice's birth date might be replaced by November 1865, by 1865, or even by 18**. Generalization enhances confidentiality by blurring attributes, but it diminishes the information conveyed – that is, generalization corrupts integrity. That tradeoff is unsurprising in light of Corollary 2.

Sweeney (2002a) quantifies the integrity of data with a *precision* metric. That metric has no obvious information-theoretic interpretation, in contrast to our metrics for leakage and suppression[‡]. As an example, consider generalization of birth dates. Assume that a program takes as input a birth date that is known to be chosen uniformly[§] at random from the year 1865. According to our definitions, if the program outputs the entire input date, it leaks about 8.5 bits and suppresses 0 bits. If the program outputs just the month and year, it leaks about 3.6 bits and suppresses about 4.9 bits. And if the program outputs just the year, it leaks 0 bits and suppresses about 8.5 bits[¶].

Adapting k-anonymity to information flow, we propose that the public output of a program must correspond to at least k possible secret inputs.

Definition: *A program S satisfies* k-anonymity *iff for every output o that program S can produce, there exist k inputs $i_1, \ldots, i_k$, such that the output of S on each of those inputs is o.*

The effect of this security condition is to make it impossible for the attacker to become certain of input by observing output. Hence, inputs are anonymous within a set of size k.

There is a similarity between our adaptation of k-anonymity and *possibilistic information flow* security conditions (Joshi and Leino 2000; Mantel 2000; McCullough 1987; McLean 1996; Smith and Volpano 1998). Those conditions typically require the set of possible outputs of a program to be independent of the secret input. The attacker, when observing an output, thus cannot be certain which of the secret inputs produced it. So every input is

[†] Samarati and Sweeney separately continued inquiry into k-anonymity (Samarati 2001; Sweeney 2002b,a). Our discussion ignores the issue of *quasi-identifiers*, which are part of the original definitions but are not relevant to our purposes.

[‡] Sweeney also uses the term 'suppression' but defines it differently than we do. She uses it to mean the complete removal of an individual's information from the output.

[§] Birth dates are, in reality, probably not uniformly distributed (Murphy 1996).

[¶] The entropy of a uniform distribution of the days in a year is about 8.5 bits, and the entropy of a uniform distribution of the months in a year is about 3.6 bits.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

anonymous within the set of all inputs[†]. But with k-anonymity, this condition is weakened to anonymity within a set of size k.

It is well known (Gray 1990; McLean 1990; Sabelfeld and Sands 2001; Volpano and Smith 1999) that possibilistic information-flow security conditions are vulnerable to attacks, including attacks based on the probabilistic behaviour of the program and on the probabilistic choice of inputs. The essential problem is that even though all (or a set of) inputs might be possible, some inputs might be more likely than others. For example, suppose that when program S produces an output o, the input is 99% likely to have been i_1, but that inputs $i_2, \ldots, i_k$, are also possible. Then the attacker can be relatively certain that the input was i_1, hence information leaks even if S satisfies k-anonymous information flow. There is no bound on the amount of leakage that might occur with k-anonymous information flow, nor is there any bound on the amount of channel suppression, because the posterior input distribution might be arbitrarily skewed.

By the same reasoning, k-anonymity is vulnerable to attacks. Fung *et al.* (2010) argue that k-anonymity protects against *record linkage* attacks but not *attribute linkage*. Record linkage occurs when an attacker can identify which record in a published database corresponds to an individual; attribute linkage occurs when an attacker can infer an individual's sensitive information from a published database. Machanavajjhala *et al.* (2007) demonstrate two attribute linkage attacks that succeed against k-anonymity: homogeneity attacks[‡] and background knowledge attacks. The latter is a kind of probabilistic attack. The essential problem is that even if an individual is anonymous within a record set of size k, one record could be more likely than the others to correspond to the individual; that record could leak information about the individual. So k-anonymity does not guarantee an upper bound on the amount of leakage, hence it does not guarantee any amount of channel suppression. The next security condition we examine, ℓ-diversity, addresses these issues.

5.2. ℓ-diversity

The *principle of ℓ-diversity* (Machanavajjhala *et al.* 2007) is that published data should not only make every individual's sensitive information appear to have at least ℓ possible values, but that each of those values should have roughly equal probability. This principle blunts homogeneity attacks as well as background-knowledge attacks, which depend on some sensitive values having higher probability than the rest.

Machanavajjhala *et al.* (2007) give an instantiation of the ℓ-diversity principle based on entropy, as follows. Define a *block* to be a set of records in which each record corresponds to an individual and in which every individual has the same values for non-sensitive attributes. For example, a block might contain all the records corresponding to individuals whose birth date is 18** and whose favourite pet is a cat. However, individuals

[†] This characterization assumes that the entire input is secret. If inputs comprise secret and non-secret components, then every input is instead anonymous within the set of all inputs having the same non-secret component.

[‡] Samarati (2001) also demonstrates homogeneity attacks.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

in the block may (indeed, should) have different values for their sensitive attributes. We can construct an *empirical probability distribution* of sensitive attributes in the block by taking their relative frequencies. For example, given the following block, the distribution would assign probability 0.5 to cancer and 0.25 to both heart disease and influenza:

Non-sensitive		*Sensitive*
Birth date	Favourite pet	Diagnosis
18**	cat	cancer
18**	cat	cancer
18**	cat	heart disease
18**	cat	influenza

For each such empirical distribution B constructed from a block of published data, *entropy ℓ-diversity* requires that $\mathcal{H}(B) \geqslant \log \ell$ holds, where $\mathcal{H}(B)$ denotes the entropy of B.[†] Applying this definition, we have that the block above is at most 1.5-diverse. A semantic interpretation of entropy ℓ-diversity (which is a syntactic condition) is that if an attacker knows that an individual is in the block, but knows nothing more, then the attacker has at least $\log \ell$ bits of uncertainty about the individual's sensitive attribute.

More generally, consider any block with empirical distribution B that satisfies entropy ℓ-diversity. The entropy of a uniform distribution of ℓ events is $\log \ell$. So if $\mathcal{H}(B) \geqslant \log \ell$, we have that B is at least as uncertain as a distribution of sensitive information in which the information has at least ℓ possible values, all of which are equally likely. Hence, entropy ℓ-diversity is an instantiation of the ℓ-diversity principle.

To measure the *utility* of ℓ-diverse published data – that is, how useful the data are for studying the characteristics of a population – Kifer and Gehrke (2006) and Machanavajjhala *et al.* (2007) use relative entropy (42), also known as *Kullback–Leibler divergence*.

Let B be an empirical distribution of sensitive attributes, as constructed above from anonymized data[‡]. And let R be an empirical distribution similarly constructed from the original (unanonymized) data. The utility measure of Kifer and Gehrke (2006) and Machanavajjhala *et al.* (2007) is the relative entropy of B to R. If B and R are the same distribution, meaning utility is maximal, their relative entropy is zero. And the less alike B and R are, the higher their relative entropy. So we call this metric *anti-utility*.

Definition: *The* anti-utility *of B with respect to R is $D(R \parallel B)$.*

To adapt anti-utility to information flow, we propose the substitution of program inputs and outputs for blocks. We treat program inputs and outputs as unstructured, rather than having rows and columns like blocks. For the unanonymized block, which is input to the anonymizer, we substitute trusted input event t_{in}. Likewise, we substitute trusted output event t_{out} for the anonymized block, which is output from the anonymizer. Distribution

[†] The definition of entropy ℓ-diversity originates with Øhrn and Ohno-Machado (1999).

[‡] We simplify their definition here. They define B as the *maximum entropy distribution* with respect to empirical distributions calculated from several published data sets.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

R should be that which would result if no anonymization occurred – that is, if the program simply echoed its input to its output. Thus for R, we substitute the distribution that assigns probability 1 to t_{in}; we denote that distribution simply as t_{in}. Distribution B should be that which results from observing the outcome of anonymization – that is, the actual output of the program. Thus for B, we substitute distribution $T_{in}|t_{out}$ of trusted inputs conditioned on observation of trusted output event t_{out}. Let T'_{in} denote distribution $T_{in}|t_{out}$.

The equivalent of anti-utility $D(R \parallel B)$ is thus $D(t_{in} \parallel T'_{in})$ in our information-flow adaptation. That quantity turns out to be exactly channel suppression $\mathcal{CS}_1$.

Theorem 1. $D(t_{in} \parallel T'_{in}) = \mathcal{CS}_1$.

Our metric $\mathcal{CS}_1$ for quantification of integrity is thus essentially the same as an existing metric for utility in database privacy. This similarity is sensible, because the less suppression data suffers, the more useful it is.

We can also adapt entropy ℓ-diversity to information flow. Recall that entropy ℓ-diversity semantically stipulates that, after observing output, the attacker has at least $\log \ell$ bits of uncertainty about the individual's sensitive attribute. For the sensitive attribute, we substitute trusted input event t_{in}; the observed output is t_{out}; and the remaining uncertainty about t_{in} after observing t_{out} is $I(t_{in} \mid t_{out})$. These substitutions lead to the following definition:

Definition: *A program S satisfies* entropy ℓ-diversity *iff for all t_{in}, and for all t_{out} produced by S, $I(t_{in} \mid t_{out}) \geqslant \log \ell$ holds.*

It is straightforward to show that entropy ℓ-diversity guarantees a lower bound on the quantity of channel suppression exhibited by a program.

Proposition 3. *A program S satisfies entropy ℓ-diversity iff for all t_{in}, and for all t_{out} produced by S, $\mathcal{CS}_1 \geqslant \log \ell$ holds.*

This bound is an improvement upon k-anonymity, which did not guarantee any suppression of inputs.

However, entropy ℓ-diversity does not directly impose an upper bound on the amount of information that may be transmitted: although $\log \ell$ bits are suppressed, many more bits might be transmitted. We turn to a stronger security condition, next, that addresses this problem.

5.3. *γ-amplification*

Suppose anonymizer A_c produces anonymized database a with 90% probability when given an original database containing the fact that Alice has cancer. Also suppose that A_c, with 1% probability, produces a from a database containing the fact that Alice has no diseases. When a is published, an attacker can infer that Alice likely has cancer. The anonymizer thus transmits information to the attacker. Moreover, this vulnerability is independent of whether a satisfies k-anonymity or ℓ-diversity – at issue is the probabilistic behaviour of the anonymizer, not whether individual outputs satisfy certain properties.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

Evfimievski *et al.* (2003) propose a security condition for anonymizers[†] that they name γ-amplification. It prevents attacks like the one above by bounding the amount by which the anonymizer can amplify the posterior probability of some inputs versus others.

Definition: *An anonymizer A satisfies* γ-amplification *(or is* γ-amplifying*) iff for all databases d and d', and for all anonymized databases a,* $\Pr(A(d) = a) \leqslant \gamma \cdot \Pr(A(d') = a)$ *holds.*

Notice that γ must be at least 1, because d and d' can be swapped. An anonymizer A_r that ignored its input and produced output by sampling from a fixed distribution, hence offering maximal privacy and minimal utility, would be 1-amplifying. As γ increases, the definition permits anonymizers to leak more sensitive information. Anonymizer A_c, described above, is at least 90-amplifying.

Amplification is straightforward to reformulate in terms of information flow. We just change databases to inputs and anonymized databases to outputs.

Definition: *A program S satisfies* γ-amplification *(or is* γ-amplifying*) iff for all inputs i and i', and for all outputs o,* $\Pr(S(i) = o) \leqslant \gamma \cdot \Pr(S(i') = o)$ *holds.*

We now show that γ-amplification, unlike k-anonymity or ℓ-diversity, yields an upper bound on the amount of information transmitted by a program. No execution of a program that is at most γ-amplifying can leak more than $\log \gamma$ bits of sensitive information.

Theorem 2. *A program S satisfies γ-amplification iff for all distributions T_{in} on inputs, all inputs i, and all outputs o, $|\mathcal{CT}_1(i, o)| \leqslant \log \gamma$ holds.*

(Recall that channel transmission $\mathcal{CT}_1$ (21) is based on distribution T_{in}.)

Anonymizer A_r, which is at most 1-amplifying, leaks zero bits of information according to this theorem. That is sensible, because A_r ignores its input. More generally, as γ increases, programs may leak more information, hence suppress less. Security parameter γ thus characterizes the quantitative information flow of a program.

However, γ-amplification does not characterize what specific information from the input may be leaked. Consider an anonymizer that is 2-amplifying, hence can leak at most one bit of information in any execution. That bit might be any bit from the input. Suppose that the anonymizer is maliciously crafted such that the bit always reveals whether Alice is HIV-positive. Alice will not be satisfied by this anonymizer, even with its low γ. Anonymizers need to protect the information of individuals. We turn next to a security condition that is designed for that goal.

5.4. *ϵ-differential privacy*

It is reasonable to desire a security condition that forbids any violation of individuals' privacy. However, Dwork (2006) shows that it is impossible for an anonymizer to satisfy such a condition if the anonymizer must also provide some utility. For example, if the anonymizer must reveal the most frequent medical diagnosis in a database, an attacker who knows that Alice's diagnosis is the same as the most frequent diagnosis will succeed

[†] Their definition is for the more general case of randomized operators. We have specialized it to anonymizers.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

in violating Alice's privacy[†]. Perfect protection of privacy is thus impossible, so individuals might prefer to withhold their information from databases. That is problematic for analysts who want to study those data.

To address this problem, Dwork *et al.* (2006) propose a security condition that is now called *differential privacy* (Dwork 2006). It stipulates that the likelihood of violating an individual's privacy should not be affected by whether the individual's information is included in a database – so the individual might as well contribute information. To make this intuition formal, define two databases to *differ in at most one individual* if they contain the same information except that one database includes an individual but the other does not. Let an anonymizer A take a database d as input and produce an anonymized data set $A(d)$ as output. Dwork (2006) defines differential privacy essentially as follows:

Definition: *An anonymizer A satisfies* ϵ-differential privacy *iff for all databases d and d' that differ in at most one individual, and for all predicates P on anonymized databases,* $\Pr(P(A(d))) \leqslant e^{\epsilon} \cdot \Pr(P(A(d')))$ *holds.*

Think of predicate P as a query run on an anonymized database[‡]. If the probability the query holds is significantly affected by whether the individual is in the database, then the query violates the individual's privacy. Differential privacy thus requires a query to hold with about the same probability regardless of whether the individual's information is included in the database. The e^{ϵ} factor quantifies how close the probabilities are.

Notice that ϵ-differential privacy is mathematically very similar to γ-amplification. Differential privacy adds the restriction that input databases must differ in at most one individual, whereas amplification quantifies over all inputs. And differential privacy considers arbitrary predicates P on outputs, whereas amplification considers just the class of equality predicates. Finally, differential privacy moves its security parameter into an exponent, enabling a simple additive result: the composition of an anonymizer that is ϵ_1-differentially private with an anonymizer that is ϵ_2-differentially private yields an anonymizer that is $(\epsilon_1 + \epsilon_2)$-differentially private (Dwork *et al.* 2010).

To reformulate differential privacy in terms of information flow, let $i \approx i'$ denote that program input i differs in at most one individual from input i'. For example, if i and i' are arrays of individuals, then $i \approx i'$ could mean that i and i' contain the same individuals except that one array contains an additional individual.

Definition: *A program S satisfies* ϵ-differential privacy *iff for all inputs i and i' such that $i \approx i'$, and for all predicates P on outputs,* $\Pr(P(S(i))) \leqslant e^{\epsilon} \cdot \Pr(P(S(i')))$ *holds.*

This definition is essentially a noninterference (Goguen and Meseguer 1982) condition: similar inputs must produce similar outputs. Although the similarity relations here are not

[†] The proof (Dwork 2006) of this impossibility result shows that there always exists a piece of background knowledge that, together with the information the querier learns from the anonymized result, results in a violation of privacy.

[‡] The intuition we give for A and P as an anonymizer and query is appropriate for the *non-interactive* model of database privacy (Dwork 2006), in which a database is anonymized, published and the public performs queries on it. In the *interactive* model (*op. cit.*), a curator interposes between the database and the public. In that model, an appropriate intuition is that A is an anonymized query, and P is a characteristic predicate identifying sets of anonymized values.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

the traditional equality of secret inputs and public outputs, researchers have studied such relaxations before (Barthe *et al.* 2004; Giacobazzi and Mastroeni 2004). So differential privacy links the field of privacy with information flow.

We now show that differential privacy bounds the amount of information leaked about an individual. First, we state a security condition that bounds the quantity of information transmitted. Given an input i, let $i \setminus \{x\}$ denote input i with individual x removed. For example, if inputs are arrays of individuals, then $i \setminus \{x\}$ could be array i without the array element containing x.

Definition: *A program S satisfies* ϵ-individual transmission *iff for all inputs i, all individuals x, all distributions T_{in} on inputs, all predicates Q on inputs and all outputs o, if the receiver is given $i \setminus \{x\}$, then* $|\mathcal{CT}_1(Q(i), o)| \leqslant \epsilon$.

(Recall that channel transmission $\mathcal{CT}_1$ (21) is based on distribution T_{in} and program S.) Think of predicate $Q(i)$ as a privacy-violating fact about individual x. Input i is chosen, and the attacker (who is the receiver) is given the entire input except for x. The attacker might then infer some information about x – for example, if the rest of the input contains only cancer patients, the attacker might surmise that x has cancer. Program S is run with input i, producing output o. If S satisfies ϵ-individual transmission, the attacker learns almost no new information from o about whether Q holds of x.

The previous two definitions are equivalent up to a constant factor, which simply accounts for the discrepancy between logarithm bases in the two definitions:

Theorem 3. *A program S satisfies ϵ-differential privacy iff S satisfies* $(\epsilon \cdot \log_2 e)$*-individual transmission.*

This result establishes information-theoretic bounds on leakage for differential privacy. Differential privacy is equivalent to transmitting almost no information about an individual x from database d beyond what is transmitted by database $d \setminus \{x\}$ without the individual. Thus, by Corollary 2, differential privacy is equivalent to suppressing almost all information about an individual. Note that $d \setminus \{x\}$ might inherently leak information about x. For example, if the attacker knows that x was a candidate for inclusion in the database, and if all the medical diagnoses in the database are of cancer, then the attacker can deduce that x likely has cancer.

Our goal with Theorem 3 was to provide an exact characterization of differential privacy, as it has previously been defined in the literature, using our quantitative theory of integrity. It turned out that this exact characterization was in terms of channel suppression $\mathcal{CT}_1$ in a single execution. The fact that we could do so illustrates the usefulness of our theory, because it was able to exactly express known definitions from the literature.

5.5. *Summary*

We adapted four security conditions from the database privacy literature to information flow in programs. Each condition offers a different guarantee on the amount of sensitive information that might be transmitted and suppressed. Sensitive information that is transmitted has been leaked, whereas sensitive information that is suppressed has been hidden.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

— k-anonymity provides no guarantee about the amount of sensitive information that is transmitted or suppressed.
— ℓ-diversity guarantees a lower bound on the amount of suppression, so attackers cannot learn all the sensitive information.
— γ-amplification guarantees an upper bound on the amount of transmission, thereby ensuring that attackers learn only a limited amount of sensitive information.
— ϵ-differential privacy guarantees that the additional amount of transmission about an individual, after the attacker is informed of the database without that individual, is nearly zero.

The first three conditions thus offer increasing security as quantified by our metrics. Differential privacy is similar to γ-amplification but guarantees individuals' privacy.

6. Application: beliefs

In our definitions of contamination and suppression, inputs are chosen according to probability distributions, and those distributions are assumed to be known by all agents. However, that assumption could be wrong – for example, with contamination, the user could believe that the attacker chooses untrusted inputs by sampling a distribution D, but the attacker might actually sample from another distribution D'. The quantity of contamination would then need be defined in terms of both distributions.

Clarkson *et al.* (2005, 2009) show how to quantify leakage from secret inputs to public outputs when agents have incorrect beliefs about the inputs. And since leakage is dual to contamination, that belief-based approach ought to work for quantifying contamination. We show that it does, next, as well as adapt it to suppression. For both contamination and suppression, the belief-based approach turns out to generalize the information-theoretic approach used so far in this paper.

6.1. *Contamination and beliefs*

A belief is a statement an agent makes about the state of the world, accompanied by some measure of how certain the agent is about the truthfulness of the statement[†]. Here, we define a *belief* to be a probability distribution of untrusted inputs. The state of the world is the actual untrusted input event, and the probability distribution characterizes the agent's (un)certainty. Note that the object of the belief is the actual input event, not the distribution of that event.

The user has a *prebelief* U_{in} about untrusted input event u_{in}. Recall that u_{in} is unobservable by the user. The user instead observes the trusted input and output, employing them to refine U_{in} to a *postbelief* U'_{in} about u_{in}. Unless the user's prebelief assigns probability 1 to u_{in}, the prebelief is *inaccurate*.

To quantify inaccuracy, we stipulate a function Δ such that $\Delta(X \twoheadrightarrow Y)$ is the inaccuracy of belief X about *reality* Y, where Y is also a distribution. Intuitively, $\Delta(X \twoheadrightarrow Y)$ is the

[†] See Halpern (2003) for a comprehensive treatment of belief representations.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

distance from the belief to reality. In previous work (Clarkson *et al.* 2009),[†] we showed that relative entropy (42) can successfully instantiate Δ:

$$\Delta(X \twoheadrightarrow Y) \triangleq D(Y \parallel X). \tag{43}$$

Since reality, in our model of contamination, is always a distribution that assigns probability 1 to event u_{in}, we can simplify our notation and definition. Let $\Delta(X \twoheadrightarrow x)$ be the inaccuracy of belief X about event x:

$$\Delta(X \twoheadrightarrow x) \triangleq -\log \Pr(X = x). \tag{44}$$

Equation (44) follows from (43) by setting Y to be a distribution that assigns probability 1 to event x. This simplified definition is equivalent to self-information – that is,

$$\Delta(X \twoheadrightarrow x) = I(x), \tag{45}$$

where the probability of x in the calculation of self-information $I(x)$ is specified by X.

Quantity $\mathcal{C}_B$ of contamination of beliefs is the improvement in accuracy of the user's belief, because the more accurate the belief becomes, the more untrusted information the user has learned:

$$\mathcal{C}_B \triangleq \Delta(U_{in} \twoheadrightarrow u_{in}) - \Delta(U'_{in} \twoheadrightarrow u_{in}). \tag{46}$$

In previous work (Clarkson *et al.* 2009), we defined an *experiment protocol* for calculating a postbelief from a prebelief and a probabilistic program semantics. That protocol turns out to be equivalent to calculating U'_{in} according to Equation (16): U'_{in} equals U_{in} conditioned on t_{in} and t_{out}.

The quantity of contamination according to $\mathcal{C}_B$ equals the quantity of contamination according to $\mathcal{C}_1$ (13).

Theorem 4. $\mathcal{C}_B = \mathcal{C}_1$.

Thus belief-based quantification is equivalent to mutual information-based quantification on single executions.

Moreover, Theorem 4 holds in expectation if the user knows the distribution the attacker uses to choose u_{in}. To capture that knowledge, define prebelief U_{in} to be *congruent with the attacker's distribution* (henceforth, *attacker congruent*) if the attacker chooses u_{in} by sampling user prebelief U_{in}. Accuracy and attacker congruence are orthogonal: 'attacker congruent' means that the user's prebelief is identical to the attacker's input distribution, hence the user and attacker have the same uncertainty about the trusted input event before it is sampled; whereas 'accurate' means that the user's prebelief assigns probability 1 to the actual input event u_{in} sampled from the attacker's input distribution, hence the user and attacker have the same uncertainty about the trusted input event after it is sampled. So a prebelief can be

— accurate but attacker incongruent (e.g., the attacker chooses uniformly between values a and b for u_{in}, the actual input event u_{in} in the execution under consideration is a, and the prebelief assigns probability 1 to a), or

[†] Function $\Delta(X \twoheadrightarrow Y)$ is written $D(X \twoheadrightarrow Y)$ in Clarkson *et al.* (2009). We change notation from D to Δ here to avoid confusion with relative entropy $D(Y \parallel X)$.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

— inaccurate but attacker congruent (e.g., the attacker chooses uniformly between a and b, actual input event u_{in} is a, and the prebelief assigns probability 0.5 to both a and b), or
— inaccurate and attacker incongruent (e.g., the attacker chooses a with probability 1, hence actual input event u_{in} must be a, but the prebelief assigns probability 1 to b), or
— accurate and attacker congruent (e.g., the attacker chooses a with probability 1, hence actual input event u_{in} must be a, and the prebelief assigns probability 1 to a).

Note that if a prebelief is both accurate and attacker congruent, the attacker must choose some input event with probability 1.

Corollary 3. *U_{in} is attacker congruent implies $\mathsf{E}[\mathcal{C}_B] = \mathcal{C}$.*

Thus belief-based quantification generalizes mutual information-based quantification.

Corollary 3 can also be understood in terms of leakage by applying the duality of contamination $\mathcal{C}$ and leakage $\mathcal{L}$ (19). If the attacker's distribution S_{in} on secret inputs is congruent with the high security user's distribution, the expected quantity of leakage according to the belief-based approach equals the quantity of leakage according to the mutual information-based approach. So, Corollary 3 also establishes how belief-based and mutual information-based measures for confidentiality are related: the mutual information measure is a special case of the belief measure.

6.2. *Suppression and beliefs*

In our model of contamination, the user holds beliefs about untrusted inputs. To model channel suppression, we replaced the user with a sender and a receiver. So to model channel suppression with beliefs, we now regard the receiver as the agent who holds beliefs. The receiver's joint prebelief (T_{in}, U_{in}) characterizes the receiver's uncertainty about trusted input t_{in} supplied by the sender and untrusted input u_{in} supplied by the attacker. And the receiver's postbelief T'_{in} characterizes the receiver's uncertainty about the untrusted input after observing the trusted output, so T'_{in} equals T_{in} conditioned on t_{out}. The improvement in the accuracy of the receiver's belief is the quantity $\mathcal{CT}_B$ of belief-based channel transmission:

$$\mathcal{CT}_B \triangleq \Delta(T_{in} \twoheadrightarrow t_{in}) - \Delta(T'_{in} \twoheadrightarrow t_{in}). \tag{47}$$

Term $\Delta(T'_{in} \twoheadrightarrow t_{in})$ characterizes the remaining error in the receiver's postbelief, hence the quantity of information that the receiver did not learn about t_{in}. So $\Delta(T'_{in} \twoheadrightarrow t_{in})$ is the quantity $\mathcal{CS}_B$ of belief-based channel suppression:

$$\mathcal{CS}_B \triangleq \Delta(T'_{in} \twoheadrightarrow t_{in}). \tag{48}$$

Unsurprisingly, the following results, corresponding to those we obtained for contamination, hold. For the corollary, we extend the notion of a congruent prebelief: (T_{in}, U_{in}) is *(sender,attacker)-congruent* if inputs t_{in} and u_{in} are chosen by the sender and attacker by sampling distributions T_{in} and U_{in}, respectively.

Theorem 5. *$\mathcal{CT}_B = \mathcal{CT}_1$ and $\mathcal{CS}_B = \mathcal{CS}_1$.*

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

Corollary 4. *(T_{in}, U_{in}) is (sender,attacker)-congruent implies $\mathsf{E}[\mathcal{CT}_B] = \mathcal{CT}$ and $\mathsf{E}[\mathcal{CS}_B] = \mathcal{CS}$.*

Thus, the belief-based definition of channel suppression generalizes the mutual information-based definition.

Likewise, we can generalize belief-based channel suppression and transmission to program suppression and transmission. Let $T'_{spec} = T_{spec}|t_{impl}$. The following definitions of belief-based program transmission $\mathcal{PT}_B$ and belief-based program suppression $\mathcal{PS}_B$ are straightforward generalizations of Equations (47) and (48):

$$\mathcal{PT}_B \triangleq \Delta(T_{spec} \twoheadrightarrow t_{spec}) - \Delta(T'_{spec} \twoheadrightarrow t_{spec}), \tag{49}$$

$$\mathcal{PS}_B \triangleq \Delta(T'_{spec} \twoheadrightarrow t_{spec}). \tag{50}$$

We obtain the obvious result:

Corollary 5. *$\mathcal{PT}_B = \mathcal{PT}_1$ and $\mathcal{PS}_B = \mathcal{PS}_1$. Further, (T_{in}, U_{in}) is (sender,attacker)-congruent implies $\mathsf{E}[\mathcal{PT}_B] = \mathcal{PT}$ and $\mathsf{E}[\mathcal{PS}_B] = \mathcal{PS}$.*

So belief-based definitions again generalize mutual information-based definitions.

7. Related work

7.1. *Quantitative*

Research on quantification of information flow began with analysis of covert channels, and progress has been made from theoretical definitions to automated analyses (Backes *et al.* 2009; Clark *et al.* 2005a; Denning 1982; Gray 1991; Lowe 2002; McCamant and Ernst 2008). Quantification of integrity and corruption is a relatively new line of research.

Newsome *et al.* (2009) implement a dynamic analysis that automatically quantifies attacker *influence*, which is a specialization of channel capacity to deterministic programs in which all inputs are either under the control of the attacker or are fixed constants. Our definitions allow probabilistic programs, trusted inputs that are not under the control of the attacker, and arbitrary distributions on inputs and outputs.

Heusser and Malacaria (2009) quantify the information leaked by a database query. They model database queries as programs, which enables application of their general purpose, automated, static analysis of leakage for C programs. Their work does not address integrity or relate information flow to existing database-privacy security conditions.

Our result (Theorem 3) about differential privacy was presented at CSF 2010; other researchers have since reported relationships between quantitative information flow and differential privacy. Alvim *et al.* (2010) and Barthe and Köpf (2011) both prove upper bounds on the leakage of differentially private mechanisms. Neither of these works, however, provides an exact characterization of differential privacy – that is, a statement of the form 'mechanism $\mathcal{K}$ leaks $f(\epsilon)$ bits if and only if $\mathcal{K}$ satisfies ϵ-differential privacy',for some f. Our Theorem 3 does provide such a characterization.

It was natural for our investigation of quantification of integrity to begin with mutual information, because variants of Shannon entropy, including conditional entropy, mutual information and channel capacity, have long been popular as metrics for quantification of

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

information flow (Backes *et al.* 2009; Chatzikokolakis *et al.* 2008a; Chen and Malacaria 2009; Clark *et al.* 2002, 2005a, 2007; Denning 1982; Gray 1991; Heusser and Malacaria 2010; Malacaria 2007; McCamant and Ernst 2008). Indeed, Smith (2009) calls these the 'consensus definitions' of a metric for quantitative information flow. Other metrics have also been proposed, including

— guessing entropy (Backes *et al.* 2009; Köpf and Basin 2007),
— relative entropy (Clarkson *et al.* 2005, 2009; Hamadou *et al.* 2010),
— min-entropy (Backes *et al.* 2009; Braun *et al.* 2009; Hamadou *et al.* 2010; Smith 2009),
— Bayes risk (Braun *et al.* 2008, 2009; Chatzikokolakis *et al.* 2008b),
— maximum likelihood (Braun *et al.* 2008) and
— marginal guesswork (Köpf and Basin 2007).

As Backes *et al.* (2009) point out, 'which [metric] is appropriate depends on the given attack scenario'. Variants of Shannon entropy were appropriate for the scenarios we examined in this work. Change the scenario, and a different metric might well become appropriate.

7.2. *Qualitative*

Biba (1977) defines the *integrity problem* as the formulation of 'policies and mechanisms that provide a subsystem with the isolation necessary for protection from subversion.' He formulates several such policies, one of which (his *strict integrity policy*) is dual to the Bell–LaPadula confidentiality policy (Bell and LaPadula 1973). But if the motivating concern is guaranteeing that systems perform as their designers intended, correctness should also be considered a critical piece of the integrity puzzle. And our program suppression measure $\mathcal{PS}$ does incorporate correctness. Perhaps other quantitative notions of correctness, such as software testing metrics, could also be understood as quantitative measures of integrity.

Li *et al.* (2003) identify three classes of qualitative integrity policies: program correctness, noninterference and data invariant. The first two are quantified by our program suppression $\mathcal{PS}$ and contamination $\mathcal{C}$ metrics, respectively. The third class contains policies stipulating that data is 'precise or accurate, consistent, unmodified, or modified only in acceptable ways'. Their formal definition of such policies clarifies that data invariant policies are safety properties (Lamport 1985). Birgisson *et al.* (2010), building upon the work of Li *et al.*, identify two kinds of data invariants that they call value and predicate invariants. The former stipulates that data values do not change; the latter, that a predicate holds before and after execution. Birgisson *et al.* argue that the program correctness subsumes all these. (And it does.) They give an execution monitor that can enforce noninterference as well as value and predicate invariants.

Information-flow integrity policies seem to receive less attention than their confidentiality counterparts. For example, early versions of Jif (Myers 1999, there called JFlow) did not include integrity policies, and Flow Caml (Pottier and Simonet 2003) does not distinguish confidentiality from integrity but instead uses an arbitrary lattice of security levels. But work on securing information flows in distributed systems programmed in Jif led to an

appreciation for the role of information-flow integrity policies, because they were needed to 'protect security-critical information from damage by subverted hosts' (Zdancewic *et al.* 2001) – an instance of Biba's integrity problem. Securing information flows in the presence of declassification (when, e.g., secret information is reclassified as public) also turned out to require integrity policies, so that attackers could not gain control over what information is declassified (Zdancewic and Myers 2001). Integrity cannot be easily dismissed, even when confidentiality is the primary concern.

Several recent systems use integrity policies in interesting ways. Some Jif-derived languages and systems incorporate integrity policies for building secure distributed applications – for example, SIF (Chong *et al.* 2007b), Swift (Chong, Liu *et al.* 2007a) and Fabric (Liu *et al.* 2009). These policies enable a principal to specify fine-grained requirements on how information may be affected by other principals. Policies also drive automated partitioning of applications, in which computations can be assigned to principals who are sufficiently trusted to perform the computations. When no such principal exists, computations can be replicated and their results validated against each other to boost integrity. Flume (Krohn *et al.* 2007) – a system that integrates information flow with operating system abstractions such as processes, pipes and sockets – also incorporates integrity policies, preventing (e.g.) untrusted dynamically-loaded code from affecting information in the process that loads it. Airavat (Roy *et al.* 2010) integrates information flow with MapReduce (Dean and Ghemawat 2004) and differential privacy (Dwork 2006), providing confidentiality and integrity for MapReduce computations and automatically declassifying computation results if they do not violate differential privacy.

7.3. *Availability*

Availability is usually treated as a concern that is separate from confidentiality and integrity. However, Zheng and Myers (2005) give a noninterference condition that simultaneously characterizes all three. Li *et al.* (2003) also examine noninterference policies for availability. We are not aware of any work in quantitative information-flow availability, but metrics such as mean time to failure seem related.

8. Concluding remarks

When we began this work, we thought we could simply apply Biba's duality to obtain a quantitative model of integrity from previous work on quantitative confidentiality. We soon discovered that the resulting model, which we named contamination, was not the same as the classical information-theoretic model of quantitative integrity, which we have since named channel suppression. We later came to see that channel suppression could be generalized to characterize the program correctness, yielding another kind of quantitative integrity.

Are there other kinds of (quantitative) integrity waiting to be discovered? We suspect so. We have not dealt, for example, with the Clark–Wilson (1987) integrity policy, which

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

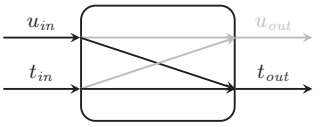

Fig. 9. Information-flow integrity in a program.

stipulates the use of trusted procedures to modify data. Nor have we dealt with database integrity constraints, which stipulate conditions that database records must satisfy.

We have not attempted to prove that contamination and suppression are sufficient to express all integrity properties, because we lack a formal definition of integrity[†]. But we gain some insight by reviewing the information-flow model we have used in this paper, depicted in Figure 9. The black arrows in this figure represent two kinds of integrity that we identified, contamination (flow from u_{in} to t_{out}) and channel suppression (attenuation of flow from t_{in} to t_{out}). The grey lines represent flows that are uninteresting from our security perspective: it does not matter how much trusted or untrusted information flows to untrusted outputs. Since these four arrows represent all possible flows, we conclude that contamination and channel suppression are the only interesting integrity properties in this information-flow model. However, other models almost certainly exist, and other kinds of integrity might have natural formulations there.

In our effort to measure integrity, we came to disentangle suppression from contamination. We also bridged a gap between database privacy and quantitative information-flow security. Here, Lord Kelvin had it right:

> When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the state of Science.
>
> —William Thomson, 1st Baron Kelvin[‡]

Acknowledgments

We thank Steve Chong, Tom Chothia, Johannes Gehrke, Mike Hicks, Andrew Myers, Pierangela Samarati and Latanya Sweeney for discussions about this work. We also thank Aslan Askarov, Steve Chong, Johannes Gehrke, Boris Köpf, Andrew Myers, Adam Smith and the anonymous CSF 2010 and MSCS reviewers for comments on drafts of this paper.

[†] The widely accepted informal definition of integrity seems to be 'prevention of unauthorized modification of information' (Voydock and Kent 1983; Clark and Wilson 1987; International Organization for Standardization 1989; Commission of the European Communities 1991; National Research Council 1991; International Organization for Standardization 2005).

[‡] From 'Electrical Units of Measurement', a lecture delivered at the Institution of Civil Engineers in London on May 3, 1883 and published in *Popular Lectures and Addresses*, 1:73, Macmillan, London, 1889. Quoted by Scripture (1892).

Appendix A. Calculations of Program Suppression

The following calculations support our analysis of program suppression examples in Section 4.2.1.

Calculation of `UnderSum` program suppression.

$$
\begin{aligned}
\mathcal{PS}_{\texttt{US}} &= \langle \text{definition (38)} \rangle \\
&\quad \mathcal{H}(T_{spec} \,|\, T_{impl}) \\
&= \langle \text{definition (10)} \rangle \\
&\quad \sum_{s \in T_{spec}, i \in T_{impl}} \Pr(s, i) \log \Pr(s|i) \\
&= \langle \text{definition of } T_{spec} \text{ for } \texttt{UnderSum} \text{ (see note at end of calculation)} \rangle \\
&\quad \sum_{s' \in Bin(n,p), i \in T_{impl}} \Pr(s', i) \log \Pr(s'|i) \\
&= \langle \text{definition of } T_{impl} \text{ for } \texttt{UnderSum} \rangle \\
&\quad \sum_{s' \in Bin(n,p), i \in Bin(n(m-1),p)} \Pr(s', i) \log \Pr(s'|i) \\
&= \langle s' \text{ is independent of } i\text{; this yields Equation (39)} \rangle \\
&\quad \sum_{s' \in Bin(n,p), i \in Bin(n(m-1),p)} \Pr(s')\Pr(i) \log \Pr(s') \\
&= \langle \text{distributivity} \rangle \\
&\quad \left(\sum_{s' \in Bin(n,p)} \Pr(s') \log \Pr(s') \right) \left(\sum_{i \in Bin(n(m-1),p)} \Pr(i) \right) \\
&= \langle \text{definition (9)} \rangle \\
&\quad \mathcal{H}(Bin(n,p)) \left(\sum_{i \in Bin(n(m-1),p)} \Pr(i) \right) \\
&= \langle \text{probability distribution must sum to 1} \rangle \\
&\quad \mathcal{H}(Bin(n,p)).
\end{aligned}
$$

Note: In the third step, we introduce bound variable s' such that $s = i + s'$. Variable s' represents array element `a[0]`. By the definitions of `UnderSum` and `SumSpec`, we have that $s = i +$ `a[0]` and `a[0]` $\sim Bin(n, p,)$. Hence $s' \sim Bin(n, p)$. □

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

Calculation of `OverSum` program suppression.

$$
\begin{aligned}
\mathcal{PS}_{\texttt{OS}} &= \langle \text{Definition (38)} \rangle \\
&\quad \mathcal{H}(T_{spec} \,|\, T_{impl}) \\
&= \langle \text{definition (10)} \rangle \\
&\quad \sum_{s \in T_{spec}, i \in T_{impl}} \Pr(s,i) \log \Pr(s|i) \\
&= \langle \text{definition of } T_{spec} \text{ for } \texttt{OverSum} \rangle \\
&\quad \sum_{s \in Bin(mn,p), i \in T_{impl}} \Pr(s,i) \log \Pr(s|i) \\
&= \langle \text{definition of } T_{impl} \text{ for } \texttt{OverSum} \text{ (see note at end of calculation)} \rangle \\
&\quad \sum_{s \in Bin(mn,p), i' \in Unif(0,2^j-1)} \Pr(s,i') \log \Pr(s|s+i') \\
&= \langle \text{definition of conditional probability} \rangle \\
&\quad \sum_{s \in Bin(mn,p), i' \in Unif(0,2^j-1)} \Pr(s,i') \log \frac{\Pr(s,s+i')}{\Pr(s+i')} \\
&= \langle s \text{ is independent of } i' \rangle \\
&\quad \sum_{s \in Bin(mn,p), i' \in Unif(0,2^j-1)} \Pr(s)\Pr(i') \log \frac{\Pr(s)\Pr(i')}{\Pr(s+i')} \\
&= \langle \Pr(i') = 2^{-j}\text{; this yields Equation (41).} \rangle \\
&\quad \sum_{s \in Bin(mn,p), i' \in Unif(0,2^j-1)} 2^{-j} \Pr(s) \log \frac{2^{-j} \Pr(s)}{\Pr(s+i')}
\end{aligned}
$$

Note. In the fourth step, we introduce bound variable i' such that $i = s + i'$. Variable i' represents memory location `a[m]`. By the definitions of `OverSum` and `SumSpec`, we have that $i = s + \texttt{a[m]}$ and $\texttt{a[m]} \sim Unif(0, 2^j - 1)$. Hence $i' \sim Unif(0, 2^j - 1)$. □

Appendix B. Proofs

In the main body, we used $I(\cdot)$ to denote information with respect to an implicit probability distribution; that distribution was always clear from context. In the proofs, it will sometimes be helpful to make the distribution explicit. So, we now let $I_X(\cdot)$ denote information according to distribution X – for example, $I_X(x)$ is the self-information of event x according to distribution X.

Proposition 1. *In the declassifier model,* $\mathcal{L}_1 + \mathcal{CS}_1 = I(ts_{in})$.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

Proof. By definition (20), $\mathcal{L}_1 = I(s_{in}, p_{out} \mid p_{in})$. Since the declassifier model does not include public inputs, $\mathcal{L}_1$ simplifies to $I(s_{in}, p_{out})$. In the declassifier model, ts_{in} is the (trusted) secret input and tp_{out} is the (trusted) public output. So $\mathcal{L}_1 = I(ts_{in}, tp_{out})$.

By definition (23), $\mathcal{CS}_1 = I(t_{in}|t_{out})$. In the declassifier model, ts_{in} is the trusted (secret) input and tp_{out} is the trusted (public) output. So $\mathcal{CS}_1 = I(ts_{in}|tp_{out})$. By Equation (3), $I(ts_{in}|tp_{out}) = I(ts_{in}) - I(ts_{in}, tp_{out})$, hence $\mathcal{CS}_1 = I(ts_{in}) - I(ts_{in}, tp_{out})$.

Therefore, $\mathcal{L}_1 + \mathcal{CS}_1 = I(ts_{in}, tp_{out}) + I(ts_{in}) - I(ts_{in}, tp_{out}) = I(ts_{in})$. □

Proposition 2. *In the full model,* $\mathcal{CS}_1 + \mathcal{L}_1 + \mathcal{CT}_1 + \mathcal{K}_1 = I(t_{in}) + I(s_{in})$.

Proof. By definitions (23), (20), (21) and (33), respectively,

— $\mathcal{CS}_1 = I(t_{in}|t_{out})$,
— $\mathcal{L}_1 = I(s_{in}, p_{out} \mid p_{in})$,
— $\mathcal{CT}_1 = I(t_{in}, t_{out})$ and
— $\mathcal{K}_1 = I(s_{in} \mid p_{out}, p_{in})$.

Since the receivers in the full model may not observe public inputs, $\mathcal{L}_1$ simplifies to $I(s_{in}, p_{out})$ and $\mathcal{K}_1$ simplifies to $I(s_{in} \mid p_{out})$. Now we calculate:

$$\begin{aligned}
\mathcal{CS}_1 + \mathcal{L}_1 + \mathcal{CT}_1 + \mathcal{K}_1 &= \langle \text{definitions and reasoning above} \rangle \\
&\quad I(t_{in}|t_{out}) + I(s_{in}, p_{out}) + I(t_{in}, t_{out}) + I(s_{in} \mid p_{out}) \\
&= \langle \text{regrouping terms} \rangle \\
&\quad \big(I(t_{in}|t_{out}) + I(t_{in}, t_{out})\big) + \big(I(s_{in}, p_{out}) + I(s_{in} \mid p_{out})\big) \\
&= \langle \text{Equation (3) and algebra, twice} \rangle \\
&\quad I(t_{in}) + I(s_{in}).
\end{aligned}$$

□

Corollary 1. *Proposition 1 follows directly from specializing Proposition 2 to the declassifier model.*

Proof. In the declassifier model, events s_{in} and t_{in} are simply ts_{in}, events p_{out} and t_{out} are simply tp_{out} and there is no public input event p_{in}. Thus we have that, in the declassifier model,

— $\mathcal{CS}_1 = I(ts_{in} \mid tp_{out})$,
— $\mathcal{L}_1 = I(ts_{in}, tp_{out})$,
— $\mathcal{CT}_1 = I(ts_{in}, tp_{out})$,
— $\mathcal{K}_1 = I(ts_{in} \mid tp_{out})$ and
— $I(t_{in}) = I(s_{in}) = I(ts_{in})$.

Starting with Proposition 2, we have that

$$\mathcal{CS}_1 + \mathcal{L}_1 + \mathcal{CT}_1 + \mathcal{K}_1 = I(t_{in}) + I(s_{in}).$$

We can rewrite Proposition 2 to specialize it to the declassifier model, using the above equalities to replace $\mathcal{CT}_1$, $\mathcal{K}_1$, $I(t_{in})$ and $I(s_{in})$:

$$\mathcal{CS}_1 + \mathcal{L}_1 + \mathcal{L}_1 + \mathcal{CS}_1 = I(ts_{in}) + I(ts_{in}).$$

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

And that simplifies to Proposition 1:

$$\mathcal{CS}_1 + \mathcal{L}_1 = I(ts_{in}).$$

□

Corollary 2. $L + S = I(d|p)$.

Proof. The proof is identical to that of Proposition 1, except that we use the following generalization of Equation (3) to conditional probabilities: $I(x, y \mid z) = I(x|z) - I(x \mid y, z)$. □

Theorem 1. $D(t_{in} \parallel T'_{in}) = \mathcal{CS}_1$.

Proof. By definition (42),

$$D(t_{in} \parallel T'_{in}) = \sum_i \Pr(t_{in} = i) \log \frac{\Pr(t_{in} = i)}{\Pr(T'_{in} = i)}.$$

Since point-mass distribution t_{in} assigns probability 1 to the event t_{in}, the summation collapses to just a single term, which is

$$\log \frac{1}{\Pr(T'_{in} = t_{in})}.$$

Applying a simple log identity, that term simplifies to $-\log \Pr(T'_{in} = t_{in})$. Now we calculate:

$$\begin{aligned} D(t_{in} \parallel T'_{in}) &= \langle \text{reasoning above} \rangle \\ &\quad -\log \Pr(T'_{in} = t_{in}) \\ &= \langle \text{by definition, } T'_{in} = T_{in}|t_{out} \rangle \\ &\quad -\log \Pr(T_{in} = t_{in} \mid t_{out}) \\ &= \langle \text{definition (2)} \rangle \\ &\quad I_{T_{in}}(t_{in}|t_{out}) \\ &= \langle \text{definition (21)} \rangle \\ &\quad \mathcal{CS}_1. \end{aligned}$$

Therefore $D(t_{in} \parallel T'_{in}) = \mathcal{CS}_1$. □

Proposition 3. *A program S satisfies entropy ℓ-diversity iff for all t_{in}, and for all t_{out} produced by S, $\mathcal{CS}_1 \geqslant \log \ell$ holds.*

Proof. By definition, S satisfies entropy ℓ-diversity iff $I(t_{in} \mid t_{out}) \geqslant \log \ell$ for all t_{in} and t_{out}. By definition (23), $\mathcal{CS}_1 = I(t_{in} \mid t_{out})$. Making that substitution, we have that S satisfies entropy ℓ-diversity iff $\mathcal{CS}_1 \geqslant \log \ell$ for all t_{in} and t_{out}. □

Theorem 2. *A program S satisfies γ-amplification iff for all distributions T_{in} on inputs, all inputs i, and all outputs o, it holds that $|\mathcal{CT}_1(i, o)| \leqslant \log \gamma$.*

Proof. The proof is essentially the same as the proof of Theorem 3 below. (That similarity is unsurprising, given the similarity of the definitions of γ-amplification and ϵ-differential privacy.)

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

First, we claim that a program S satisfies γ-amplification iff S satisfies γ-amplification semantic security, which is defined as follows.

Definition: *A program S satisfies* γ-amplification semantic security *iff for all input distributions D, all inputs i, and all outputs o, it holds that* $\Pr(D = i) \leqslant \gamma \cdot \Pr(D = i \mid S(D) = o)$.

The proof of that claim follows the same steps as the proof of the equivalence of differential privacy and its associated semantic security condition (Dwork *et al.* 2006).

Second, we claim that a program S satisfies γ-amplification semantic security iff S transmits at most $\log \gamma$ bits in any execution – that is, iff for all distributions T_{in} on inputs, all inputs i, and all outputs o, it holds that $|\mathcal{CT}_1(i, o)| \leqslant \log \gamma$. The proof of that claim follows the same steps as the proof below of the equivalence of ϵ-semantic security and $(\epsilon \cdot \log_2 e)$-individual transmission.

The theorem follows immediately from the preceding two claims. □

Theorem 3. *A program S satisfies ϵ-differential privacy iff S satisfies $(\epsilon \cdot \log_2 e)$-individual transmission.*

Proof. First, we state a definition of Dwork *et al.* (2006), adapted to information flow. In this definition, an *i-consistent distribution* is a distribution of inputs that assigns non-zero probability only to inputs of the form $i \cup \{x\}$ for an individual x, where $i \cup \{x\}$ denotes an input that contains all the individuals in i as well as individual x.

Definition: *A program S satisfies* ϵ-semantic security *iff for all inputs i, all i-consistent distributions T_{in}, all predicates Q on inputs, and all outputs o, if the attacker is informed of i, then, letting I' be a random variable distributed according to T_{in},*

$$\left| \ln \left(\frac{\Pr(Q(I'))}{\Pr(Q(I') \mid S(I') = o)} \right) \right| \leqslant \epsilon.$$

The probabilities in the inequality are with respect to T_{in} and S. Random variable I' describes a random choice of one more individual in addition to those individuals already in i. Note that our definition is simplification of the definition of Dwork *et al.*: we consider the privacy of only one individual rather than k individuals, and we assume a non-interactive program.

Second, we note that Dwork *et al.* (2006, claim 3) prove the equivalence of semantic security and differential privacy. Adapted to information flow, that result can be stated as follows.

Claim. *A program S satisfies ϵ-differential privacy iff S satisfies ϵ-semantic security.*

Finally, our own theorem is now simple to prove using that claim. We need only show that S satisfies ϵ-semantic security iff S satisfies $(\epsilon \cdot \log_2 e)$-individual transmission. Let i, Q, and o be arbitrary, and let T_{in} be i-consistent. Assume that the attacker is informed

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

of i.

$$
\begin{aligned}
& S \text{ satisfies } \epsilon\text{-semantic security} \\
&= \langle \text{definition of semantic security} \rangle \\
& \left| \ln \left(\frac{\Pr(Q(I'))}{\Pr(Q(I')|S(I') = o)} \right) \right| \leqslant \epsilon \\
&= \langle \text{log manipulation; } |x - y| = |-x + y| \rangle \\
& |-\ln \Pr(Q(I')) + \ln \Pr(Q(I')|S(I') = o)| \leqslant \epsilon \\
&= \langle \text{convert ln to } \log_2 \text{; definitions (1) and (2)} \rangle \\
& \left| \frac{I(Q(I'))}{\log_2 e} - \frac{I(Q(I')|S(I') = o)}{\log_2 e} \right| \leqslant \epsilon \\
&= \langle \text{simplify; Equation (22), substituting } Q(I') \text{ for } t_{in} \text{ and } o \text{ for } t_{out} \rangle \\
& |\mathcal{CT}_1(Q(I'), o)| \leqslant \epsilon \cdot \log_2 e \\
&= \langle \text{definition of individual transmission} \rangle \\
& S \text{ satisfies } (\epsilon \cdot \log_2 e)\text{-individual transmission.}
\end{aligned}
$$

□

Theorem 4. $\mathcal{C}_B = \mathcal{C}_1$.

Proof. By definition (46), $\mathcal{C}_B = \Delta(U_{in} \twoheadrightarrow u_{in}) - \Delta(U'_{in} \twoheadrightarrow u_{in})$. Applying Equation (45) twice to that equality, we have that $\mathcal{C}_B = I_{U_{in}}(u_{in}) - I_{U'_{in}}(u_{in})$. (For an explanation of subscripts U_{in} and U'_{in} on self-information I, see the first paragraph of this appendix.) Since untrusted inputs are independent of trusted inputs, $\Pr(u_{in}) = \Pr(u_{in}|t_{in})$, hence $I_{U_{in}}(u_{in}) = I_{U_{in}}(u_{in}|t_{in})$. Also, by Equation (16), $U'_{in} = U_{in} \mid t_{in}, t_{out}$. So we have that

$$I_{U'_{in}}(u_{in}) = I_{U_{in} \mid t_{in}, t_{out}}(u_{in}) = I_{U_{in}}(u_{in} \mid t_{in}, t_{out}).$$

Thus,

$$\mathcal{C}_B = I_{U_{in}}(u_{in}|t_{in}) - I_{U_{in}}(u_{in} \mid t_{in}, t_{out}).$$

For the rest of this proof, all self-information will be in terms of distribution U_{in}, so we cease writing that subscript on I. By definition (13), $\mathcal{C}_1 = I(u_{in}, t_{out} \mid t_{in})$. So, to show that $\mathcal{C}_1 = \mathcal{C}_B$, it suffices to show that

$$I(u_{in}, t_{out} \mid t_{in}) = I(u_{in}|t_{in}) - I(u_{in} \mid t_{in}, t_{out}).$$

The following lemma does just that. □

Lemma 1. $I(x, z \mid y) = I(x|y) - I(x \mid y, z)$.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

Proof. Intuitively, this lemma is the same as Equation (3), but with every term conditioned on y. Formally, we calculate, starting with the left-hand side of the lemma:

$$\begin{aligned} I(x, z \mid y) &= \langle \text{definition (7)} \rangle \\ &\quad -\log \frac{\Pr(x|y)\Pr(z|y)}{\Pr(x, z|y)} \\ &= \langle \text{definition } \Pr(a|b), \text{ twice} \rangle \\ &\quad -\log \frac{\Pr(x|y)\Pr(y, z)\Pr(y)}{\Pr(x, y, z)\Pr(y)} \\ &= \langle \text{simplification} \rangle \\ &\quad -\log \frac{\Pr(x|y)\Pr(y, z)}{\Pr(x, y, z)}. \end{aligned}$$

Similarly, we calculate, starting with the right-hand side of the lemma:

$$\begin{aligned} I(x|y) - I(x \mid y, z) &= \langle \text{definition (2), twice} \rangle \\ &\quad -\big(\log \Pr(x|y) - \log \Pr(x \mid y, z)\big) \\ &= \langle \log \text{ identity} \rangle \\ &\quad -\log \frac{\Pr(x|y)}{\Pr(x \mid y, z)} \\ &= \langle \text{definition } \Pr(a|b) \rangle \\ &\quad -\log \frac{\Pr(x|y)\Pr(y, z)}{\Pr(x, y, z)}. \end{aligned}$$

Both sides of the lemma turned out to equal the same formula. We therefore have that

$$I(x, z \mid y) = -\log \frac{\Pr(x|y)\Pr(y, z)}{\Pr(x, z, y)} = I(x|y) - I(x \mid y, z).$$

□

Corollary 3. *U_{in} is attacker congruent implies $\mathsf{E}[\mathcal{C}_B] = \mathcal{C}$.*

Proof. In the calculation of $\mathsf{E}[\mathcal{C}_B]$, the user's prebelief U_{in} could in general differ from the actual distribution R on untrusted inputs. Expectation $\mathsf{E}[\mathcal{C}_B]$ should be with respect to R, since R yields the actual probabilities that should be used as weights in the expectation's weighted average. However, by the assumption that U_{in} is attacker congruent, we have that $U_{in} = R$. Therefore, $\mathsf{E}[\mathcal{C}_B] = \mathsf{E}_{U_{in}}[\mathcal{C}_1] = \mathcal{C}$, where $\mathsf{E}_X[\cdot]$ denotes expectation with respect to distribution X, the first equality follows from Theorem 4 and the second equality follows from Equation (17). □

Theorem 5. *$\mathcal{CT}_B = \mathcal{CT}_1$ and $\mathcal{CS}_B = \mathcal{CS}_1$.*

Proof. In this proof, $I(\cdot)$ abbreviates $I_{T_{in}}(\cdot)$ – that is, if no subscript is present on I, the self-information is with respect to distribution T_{in}.

By definition, $\mathcal{CS}_B = \Delta(T'_{in} \twoheadrightarrow t_{in})$. By Equation (45), we have that $\Delta(T'_{in} \twoheadrightarrow t_{in}) = I_{T'_{in}}(t_{in})$. By the definition of T'_{in}, we have that $I_{T'_{in}}(t_{in}) = I(t_{in}|t_{out})$. And that last term is the definition of $\mathcal{CS}_1$ (23). Therefore, $\mathcal{CS}_B = \mathcal{CS}_1$.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

By definition,

$$\mathcal{CT}_B = \Delta(T_{in} \twoheadrightarrow t_{in}) - \Delta(T'_{in} \twoheadrightarrow t_{in}),$$

and by Equation (22),

$$\mathcal{CT}_1 = I(t_{in}) - I(t_{in}|t_{out}).$$

By the definitions of $\mathcal{CS}_B$ (48) and $\mathcal{CS}_1$ (23), we can rewrite those equalities as follows:

$$\mathcal{CT}_B = \Delta(T_{in} \twoheadrightarrow t_{in}) - \mathcal{CS}_B,$$
$$\mathcal{CT}_1 = I(t_{in}) - \mathcal{CS}_1.$$

By Equation (45), we have that $\Delta(T_{in} \twoheadrightarrow t_{in}) = I(t_{in})$. Therefore, since $\mathcal{CS}_B = \mathcal{CS}_1$, we have that $\mathcal{CT}_B = \mathcal{CT}_1$. □

Corollary 4. *(T_{in}, U_{in}) is (sender,attacker)-congruent implies $\mathsf{E}[\mathcal{CT}_B] = \mathcal{CT}$ and $\mathsf{E}[\mathcal{CS}_B] = \mathcal{CS}$.*

Proof. The proof technique is the same as in the proof of Corollary 3. In short, if (T_{in}, U_{in}) is (sender,attacker)-congruent, then Theorem 5 implies that $\mathsf{E}[\mathcal{CT}_B] = \mathsf{E}_{(T_{in},U_{in})}[\mathcal{CT}_1] = \mathcal{CT}$, and likewise for $\mathsf{E}[\mathcal{CS}_B]$ and $\mathcal{CS}$. □

Corollary 5. *$\mathcal{PT}_B = \mathcal{PT}_1$ and $\mathcal{PS}_B = \mathcal{PS}_1$. Further, (T_{in}, U_{in}) is (sender,attacker)-congruent implies $\mathsf{E}[\mathcal{PT}_B] = \mathcal{PT}$ and $\mathsf{E}[\mathcal{PS}_B] = \mathcal{PS}$.*

Proof. The proof techniques are the same as in the proofs of the corresponding statements in Theorem 5 and Corollary 4. One additional fact is needed: if (T_{in}, U_{in}) is (sender,attacker)-congruent, then T_{spec} and T_{impl} as calculated by the receiver are equal to those distributions as they would be calculated by an agent who knows the distributions used by both the sender and attacker. That fact holds because T_{spec} and T_{impl} are defined in terms of T_{in}, U_{in} and the program semantics – which is known to the receiver. □

References

Adámek, J. (1991) *Foundations of Coding*, John Wiley and Sons, New York.

Alvim, M. S., Chatzikokolakis, K., Degano, P. and Palamidessi, C. (2010) Differential privacy versus quantitative information flow, *Technical Report hal-00548214*, INRIA. Available at `http://hal.inria.fr/hal-00548214/en`.

Backes, M. (2005) Quantifying probabilistic information flow in computational reactive systems. In: *Proceedings European Symposium on Research in Computer Security* 336–354.

Backes, M., Köpf, B. and Rybalchenko, A. (2009) Automated discovery and quantification of information leaks. In: *Proceedings IEEE Symposium on Security and Privacy* 141–153.

Barthe, G., D'Argenio, P. R. and Rezk, T. (2004) Secure information flow by self-composition. In: *Proceedings IEEE Computer Security Foundations Workshop* 100–114.

Barthe, G. and Köpf, B. (2011) Information-theoretic bounds for differentially private mechanisms. In: *Proceedings IEEE Computer Security Foundations Symposium* 191–204.

Bell, D. E. and LaPadula, L. J. (1973) Secure computer systems: Mathematical foundations, *Technical Report 2547*, Volume I, MITRE Corporation.

Biba, K. (1977) Integrity considerations for secure computer systems, *Technical Report MTR-3153*, MITRE Corporation.

https://doi.org/10.1017/S0960129513000595 Published online by Cambridge University Press

Birgisson, A., Russo, A. and Sabelfeld, A. (2010) Unifying facets of information integrity. In: *Proceedings International Conference on Information Systems Security* 48–65.

Braun, C., Chatzikokolakis, K. and Palamidessi, C. (2008) Compositional methods for information-hiding. In: *Proceedings International Conference on Foundations of Software Science and Computation Structures* 443–457.

Braun, C., Chatzikokolakis, K. and Palamidessi, C. (2009) Quantitative notions of leakage for one-try attacks. In: *Proceedings Conference on Mathematical Foundations of Programming Semantics* 75–91.

Chatzikokolakis, K., Palamidessi, C. and Panangaden, P. (2008a) Anonymity protocols as noisy channels. *Information and Computation* **206** (2–4) 378–401.

Chatzikokolakis, K., Palamidessi, C. and Panangaden, P. (2008b) On the Bayes risk in information-hiding protocols. *Journal of Computer Security* **16** (5) 531–571.

Chen, H. and Malacaria, P. (2009) Quantifying maximal loss of anonymity in protocols. In: *ACM Symposium on Information, Computer and Communications Security* 206–217.

Chong, S., Liu, J., Myers, A. C., Qi, X., Vikram, K., Zheng, L. and Zheng, X. (2007a) Secure web applications via automatic partitioning. In: Proceedings ACM Symposium on Operating Systems Principles 31–44.

Chong, S., Vikram, K. and Myers, A. C. (2007b) SIF: Enforcing confidentiality and integrity in web applications. In: *Proceedings USENIX Security Symposium* 1–16.

Clark, D., Hunt, S. and Malacaria, P. (2002) Quantitative analysis of the leakage of confidential data. *Electronic Notes in Theoretical Computer Science* **59** (3) 1–14.

Clark, D., Hunt, S. and Malacaria, P. (2005a) Quantified interference for a while language. *Electronic Notes in Theoretical Computer Science* **112** 149–166.

Clark, D., Hunt, S. and Malacaria, P. (2005b) Quantitative information flow, relations and polymorphic types. *Journal of Logic and Computation* **18** (2) 181–199.

Clark, D., Hunt, S. and Malacaria, P. (2007) A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security* **15** (3) 321–371.

Clark, D. D. and Wilson, D. R. (1987) A comparison of commercial and military computer security policies. In: Proceedings IEEE Symposium on Security and Privacy 184–194.

Clarkson, M. R., Myers, A. C. and Schneider, F. B. (2005) Belief in information flow. In: Proceedings IEEE Computer Security Foundations Workshop 31–45.

Clarkson, M. R., Myers, A. C. and Schneider, F. B. (2009) Quantifying information flow with beliefs. *Journal of Computer Security* **17** (5) 655–701.

Clarkson, M. R. and Schneider, F. B. (2010) Quantification of integrity. In: *Proceedings IEEE Computer Security Foundations Symposium* 28–43.

Commission of the European Communities (1991) Information technology security evaluation criteria: Provisional harmonised criteria. Document COM(90) 314, Version 1.2.

Cover, T. M. and Thomas, J. A. (1991) *Elements of Information Theory*, John Wiley and Sons, New York.

Dean, J. and Ghemawat, S. (2004) MapReduce: Simplified data processing on large clusters. In: Proceedings USENIX Symposium on Operating System Design and Implementation 137–150.

Denning, D. (1982) *Cryptography and Data Security*, Addison-Wesley, Reading, Massachusetts.

Dwork, C. (2006) Differential privacy. In: *Proceedings International Colloquium on Automata, Languages and Programming* 1–12.

Dwork, C., McSherry, F., Nissim, K. and Smith, A. (2006) Calibrating noise to sensitivity in private data analysis. In: *Proceedings Theory of Cryptography Conference* 265–284.

Dwork, C., Naor, M., Pitassi, T. and Rothblum, G. N. (2010) Differential privacy under continual observation. In: *Proceedings ACM Symposium on Theory of Computing* 715–724.

Evfimievski, A., Gehrke, J. and Srikant, R. (2003) Limiting privacy breaches in privacy preserving data mining. In: *Proc. ACM Symposium on Principles of Database Systems* 211–222.

Fung, B. C. M., Wang, K., Chen, R. and Yu, P. S. (2010) Privacy-preserving data publishing: A survey on recent developments. *ACM Computing Surveys* **42** (4) 14:1–53.

Giacobazzi, R. and Mastroeni, I. (2004) Abstract non-interference. In: *Proceedings ACM Symposium on Principles of Programming Languages* 186–197.

Goguen, J. A. and Meseguer, J. (1982) Security policies and security models. In: *Proceedings IEEE Symposium on Security and Privacy* 11–20.

Gray, J. W., III (1990) Probabilistic interference. In: Proceedings IEEE Symposium on Security and Privacy 170–179.

Gray, J. W., III (1991) Toward a mathematical foundation for information flow security. In: *Proceedings IEEE Symposium on Security and Privacy* 21–35.

Halpern, J. Y. (2003) *Reasoning about Uncertainty*, MIT Press, Cambridge, Massachusetts.

Hamadou, S., Sassone, V. and Palamidessi, C. (2010) Reconciling belief and vulnerability in information flow. In: *Proceedings IEEE Symposium on Security and Privacy* 79–92.

Heusser, J. and Malacaria, P. (2009) Applied quantitative information flow and statistical databases. In: *Workshop on Formal Aspects in Security and Trust* 96–110.

Heusser, J. and Malacaria, P. (2010) Quantifying information leaks in software. In: *Annual Computer Security Applications Conference* 261–269.

International Organization for Standardization (1989) Information processing systems: Open systems interconnection – basic reference model, Part 2: Security architecture, ISO 7498-2.

International Organization for Standardization (2005) Common criteria for information technology security evaluation: Part 1: Introduction and general model, ISO 15408. CCMB-2006-09-001, Version 3.1, Revision 1. Available from `http://www.commoncriteriaportal.org`.

Jones, D. S. (1979) *Elementary Information Theory*, Clarendon Press, Oxford.

Joshi, R. and Leino, K. R. M. (2000) A semantic approach to secure information flow. *Science of Computer Programming* **37** 113–138.

Kifer, D. and Gehrke, J. (2006) Injecting utility into anonymized datasets. In: *Proceedings ACM Conference on Management of Data* 217–228.

Köpf, B. and Basin, D. (2007) An information-theoretic model for adaptive side-channel attacks. In: *Proceedings ACM Conference on Computer and Communications Security* 286–296.

Kozen, D. (1981) Semantics of probabilistic programs. *Journal of Computer and System Sciences* **22** 328–350.

Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E. and Morris, R. (2007) Information flow control for standard OS abstractions. In: Proceedings ACM Symposium on Operating Systems Principles 321–334.

Lamport, L. (1985) Basic concepts: Logical foundation. In: Distributed Systems: Methods and Tools for Specification, An Advanced Course. *Springer Lecture Notes in Computer Science* **190** 19–30.

Laud, P. (2001) Semantics and program analysis of computationally secure information flow. In: Proceedings European Symposium on Programming. *Springer Lecture Notes in Computer Science* **2028** 77–91.

Li, P., Mao, Y. and Zdancewic, S. (2003) Information integrity policies. In: *Workshop on Formal Aspects in Security and Trust* 53–70.

Liu, J., George, M. D., Vikram, K., Qi, X., Waye, L. and Myers, A. C. (2009) Fabric: A platform for secure distributed computation and storage. In: *Proceedings ACM Symposium on Operating Systems Principles* 321–334.

Livshits, V. B. and Lam, M. S. (2005) Finding security vulnerabilities in Java applications with static analysis. In: *Proceedings USENIX Security Symposium* 271–286.

Lowe, G. (2002) Quantifying information flow. In: *Proceedings IEEE Computer Security Foundations Workshop* 18–31.

Machanavajjhala, A., Kifer, D., Gehrke, J. and Venkitasubramaniam, M. (2007) ℓ-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data*. Available from http://dl.acm.org/citation.cfm?id=1217302.

Malacaria, P. (2007) Assessing security threats of looping constructs. In: *Proceedings ACM Symposium on Principles of Programming Languages* 225–235.

Mantel, H. (2000) Possibilistic definitions of security: An assembly kit. In: *Proc. IEEE Computer Security Foundations Workshop*, 185–199.

McCamant, S. and Ernst, M. D. (2008) Quantitative information flow as network capacity. In: *Proceedings ACM Conference on Programming Language Design and Implementation* 193–205.

McCullough, D. (1987) Specifications for multi-level security and a hook-up property. In: Proceedings IEEE Symposium on Security and Privacy 161–166.

McLean, J. (1990) Security models and information flow. In: Proceedings IEEE Symposium on Security and Privacy 180–189.

McLean, J. (1996) A general theory of composition for a class of 'possibilistic' properties. *IEEE Transactions on Software Engineering* **22** (1) 53–67.

Millen, J. (1987) Covert channel capacity. In: Proceedings IEEE Symposium on Security and Privacy 60–66.

Murphy, R. (1996) An analysis of the distribution of birthdays in a calendar year. Available at `http://www.panix.com/ murphy/bday.html`, accessed Dec. 29, 2009.

Myers, A. C. (1999) JFlow: Practical mostly-static information flow control. In: *Proceedings ACM Symposium on Principles of Programming Languages* 228–241.

National Research Council (1991) *Computers at Risk: Safe Computing in the Information Age*. National Academy Press, Washington, D.C.

Newsome, J., McCamant, S. and Song, D. (2009) Measuring channel capacity to distinguish undue influence. In: *Proceedings ACM Workshop on Programming Languages and Analysis for Security*. Available from `http://doi.acm.org/10.1145/1554339.1554349`.

Newsome, J. and Song, D. (2005) Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software. In: *Proceedings Symposium on Network and Distributed System Security*. Available from `http://www.isoc.org/isoc/conferences/ndss/05/proceedings/papers/taintcheck.pdf`.

Øhrn, A. and Ohno-Machado, L. (1999) Using Boolean reasoning to anonymize databases. *Artificial Intelligence in Medicine* **15** (3) 235–254.

Pottier, F. and Simonet, V. (2003) Information flow inference for ML. *ACM Transactions on Programming Languages and Systems* **25** (1) 117–158.

Rinard, M., Cadar, C., Dumitran, D., Roy, D. M., Leu, T. and Beebee, W. S., Jr. (2004) Enhancing server availability and security through failure-oblivious computing. In: Proceedings USENIX Symposium on Operating System Design and Implementation 303–316.

Roy, I., Setty, S. T. V., Kilzer, A., Shmatikov, V. and Witchel, E. (2010) Airavat: Security and privacy for MapReduce. In: *Proceedings USENIX Symposium on Networked Systems Design and Implementation* 297–312.

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

Samarati, P. (2001) Protecting respondents' identities in microdata release. *IEEE Transactions on Knowledge and Data Engineering* **13** (6) 1010–1027.

Samarati, P. and Sweeney, L. (1998) Protecting privacy when disclosing information: *k*-anonymity and its enforcement through generalization and suppression, *Technical Report SRI-CSL-98-04,* Computer Science Laboratory, SRI International. Available from `http://www.csl.sri.com/papers/sritr-98-04`.

Schneider, F. B. (2000) Enforceable security policies. *ACM Transactions on Information and System Security* **3** (1) 30–50.

Scripture, E. W. (1892) The need of psychological training. *Science* **19** (474) 127–128. Available from `http://www.jstor.org/stable/1766918`.

Shafer, G. (1976) *A Mathematical Theory of Evidence*, Princeton University Press, Princeton, NJ.

Shannon, C. E. (1948) A mathematical theory of communication. *Bell System Technical Journal* **27** 379–423 and 623–656.

Smith, G. (2009) On the foundations of quantitative information flow. In: *Proceedings Conference on Foundations of Software Science and Computation Structures* 288–302.

Smith, G. and Volpano, D. (1998) Secure information flow in a multi-threaded imperative language. In *Proceedings ACM Symposium on Principles of Programming Languages* 355–364.

Suh, G. E., Lee, J. W., Zhang, D. and Devedas, S. (2004) Secure program execution via dynamic information flow tracking. In: *Proceedings ACM Conference on Architectural Support for Programming Languages and Systems* 85–96.

Sweeney, L. (2002a) Achieving *k*-anonymity privacy protection using generalization and suppression. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems* **10** (5) 571–588.

Sweeney, L. (2002b) *k*-anonymity: A model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems* **10** (5) 557–570.

Volpano, D. (2000) Secure introduction of one-way functions. In: *Proceedings IEEE Computer Security Foundations Workshop* 246–254.

Volpano, D. and Smith, G. (1999) Probabilistic noninterference in a concurrent language. *Journal of Computer Security* **7** (2,3) 231–253.

Voydock, V. L. and Kent, S. T. (1983) Security mechanisms in high-level network protocols. *Computing Surveys* **15** (2) 135–171.

Wall, L., Christiansen, T. and Schwartz, R. L. (1996) *Programming Perl*, 2nd edition, O'Reilly, Sebastopol, California.

Xu, W., Bhatkar, S. and Sekar, R. (2006) Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In: *Proceedings USENIX Security Symposium* 121–136.

Zdancewic, S. and Myers, A. C. (2001) Robust declassification. In: *Proceedings IEEE Computer Security Foundations Workshop* 15–23.

Zdancewic, S., Zheng, L., Nystrom, N. and Myers, A. C. (2001) Untrusted hosts and confidentiality: Secure program partitioning. In: *Proceedings ACM Symposium on Operating Systems Principles* 1–14.

Zheng, L. and Myers, A. C. (2005) End-to-end availability policies and noninterference. In: Proceedings IEEE Computer Security Foundations Workshop 272–286.