2010 23rd IEEE Computer Security Foundations Symposium

Quantification of Integrity

Michael R. Clarkson Fred B. Schneider
Department of Computer Science
Cornell University
Ithaca, NY, USA
{*clarkson, fbs*}*@cs.cornell.edu*

***Abstract*: Two kinds of integrity measures—contamination and suppression—are introduced. Contamination measures how much untrusted information reaches trusted outputs; it is the dual of information-flow confidentiality. Suppression measures how much information is lost from outputs; it does not have a confidentiality dual. Two forms of suppression are considered: programs and channels. Program suppression measures how much information about the correct output of a program is lost because of attacker influence and implementation errors. Channel suppression measures how much information about inputs to a noisy channel is missing from channel outputs. The relationship between quantitative integrity, confidentiality, and database privacy is examined.**

***Keywords*: Integrity, quantitative information flow, information theory, database privacy**

I. Introduction

Many integrity requirements for computer systems are qualitative, but quantitative requirements can also be valuable. For example, a system might combine data from trusted and untrusted sensors if the untrusted sensors cannot corrupt the result too much. As another example, we might add noise to a database, thereby protecting privacy, 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 twenty years [1], [2].

To quantify corruption, a formal definition of "integrity" is required. We know of no such widely accepted definition, although the widely accepted informal definition seems to be "prevention of unauthorized modification of information" [3]–[8]. So we take two distinct notions of information modification as points of departure: taint analysis and program correctness. These lead to two distinct measures of corruption that we name "contamination" and "suppression."

Contamination generalizes *taint analysis* [9]–[13], which tracks information flow from *tainted* inputs to outputs that are supposed to be *untainted*—or, using alternative terminology, from *untrusted* inputs to outputs that are supposed to be *trusted*. This flow results in *contamination* of the trusted outputs. Trusted outputs are not supposed to be influenced by untrusted information, so contamination corrupts integrity. We might be willing to deem a program secure if it allows only limited contamination, even though 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 [14], who identified a duality between 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 [15] and on beliefs [16]. This paper adapts those measures to contamination.[1] Through the Biba duality, we obtain a measure for corruption from a measure for leakage.

Suppression, our other measure for corruption, is derived from program correctness. For a given input, a correct implementation should produce an output o permitted by a specification. The output might be permitted to differ from o provided the output conveys the same information as o. An implementation might, for example, produce all the bits in the binary representation of o but in reverse order. Or the implementation might produce o `xor` k, where k is a known constant. Any knowledgeable user of these implementations could recover o from the implementation's output.

Conversely, the output of an incorrect implementation would fail to convey all the information about o. For example, a (somewhat) incorrect implementation might output only the first few bits of o; or it might output o with probability p and output garbage with probability $1-p$; or it might output o `xor` u, where u is an untrusted input. In each case, we say that *program suppression* of information about the correct output has occurred. Implementation outputs are not supposed to omit information about correct outputs, so program suppression corrupts integrity. Yet we might be willing to use an implementation that produces sufficient information about the correct output, hence exhibits little program suppression, even though a traditional verification methodology (e.g., Hoare logic) 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 form of program suppression that we call *channel*

[1] Newsome et al. [17] adapt the same information-theoretic metric to measure what they call *quantitative influence* (cf. §VI).

978-0-7695-4082-5/10 $26.00 © 2010 IEEE
DOI 10.1109/CSF.2010.10

suppression. The echo specification stipulates that a correct output `o` is the value of input `t`, similar to the Unix `echo` command. For the echo specification, our model of program suppression simplifies to the information-theoretic model of communication channels [18], 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. For example, a noisy channel could be modeled by implementation "`o := t xor u`", in which noise supplied by the attacker in untrusted input `u` causes information about correct output `t` to be lost. The channel thus damages the integrity of information.

With programs and channels, *suppression* occurs when information is lost. 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, giving both information-theoretic and belief-based definitions. Applying the Biba duality to these definitions yields no meaningful confidentiality dual. So the classical duality of confidentiality and integrity was, in retrospect, incomplete.

We might suspect that contamination generalizes suppression, or vice versa, but this is not the case. Consider the following three program statements, which take `t` as trusted input and `u` as untrusted input, and produce `o` as trusted output. Suppose that these statements 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 (only) contamination, because trusted output contains information derived from untrusted input. A user of this program's output might filter out and ignore contaminant `u`, but that's irrelevant: the contaminant damages integrity just by its presence, as in taint analysis. Contamination is concerned only with measuring the amount of contaminant in the output—not what the user does with the output. This program does not exhibit program suppression, because its output contains all the information about the value of `t`.
- `o := t xor n`, where `n` is randomly generated noise. This program exhibits (only) program suppression, because information about the correct output is lost. Suppression concerns that loss; suppression is not concerned with the presence of contaminant. In fact, this program cannot exhibit contamination, because it has no untrusted inputs.
- `o := t xor u`. This program exhibits contamination, because untrusted information 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.

In addition to introducing measures for corruption, this paper examines 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.

Finally, we revisit work on database privacy. Databases that contain information about individuals are sometimes anonymized and published to enable statistical analysis. The goal is to protect the privacy of individuals, while still providing 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 analyze database privacy conditions from the literature.

We proceed as follows. Models for quantifying contamination and suppression are given in §II and §III. Belief-based definitions and their relationship to information-theoretic definitions are examined in §IV. Database privacy is analyzed in §V. Related work is discussed in §VI, and §VII concludes. All proofs appear in the accompanying technical report [19].

Basic notions from information theory (e.g., entropy and mutual information) are used throughout the paper. Their definitions can be found in the accompanying technical report [19] or in any introductory text [20], [21].

II. Quantification of Contamination

Three agents are involved in our model of program execution: a system, a user, and an attacker.[2] The *system* executes the program, which has variables categorized as *input*, *output*, or *internal*. Input variables may only be read by the program, output variables may only be written by the program, and internal variables may be read and written but may never be observed by any agent except the system itself. 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 about untrusted inputs that contaminates trusted outputs. This goal generalizes taint analysis, which just determines whether any information about 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:

[2]Although we phrase our theory in terms of programs, other notions of computation could be used. All we require is that there are inputs, outputs, and a way to derive output distributions from input distributions.

Figure 1. Contamination model

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 quantification of confidentiality. The resemblance is deliberate, because we seek a definition of integrity that is dual to confidentiality. In particular, our approach is dual to the technique of Clark et al. [15], [22] for quantifying leakage.[3]

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.[4] These access restrictions agree with the Biba integrity model [14]: 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 1.

A. Contamination in Single Executions

Information theory explains the behavior of *channels*. A channel, like a program, accepts inputs and produces outputs. So information flow can be quantified by modeling 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.) Also let T_{in}, U_{in}, and T_{out} denote probability distributions on trusted inputs, untrusted inputs, and trusted outputs.[5]

Mutual information characterizes the quantity of information that can be learned about channel inputs by observing outputs. Let $I(u_{in}, t_{out})$ denote the mutual information between events u_{in} and t_{out}—that is, the amount of information either event conveys about the other.[6] Note that $I(\cdot,\cdot)$ is mutual information between single events, not the more familiar mutual information between distributions of events. Let $I(u_{in}, t_{out} \mid t_{in})$ denote 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:

$$\mathcal{C}_1 \triangleq I(u_{in}, t_{out} \mid t_{in}). \tag{1}$$

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

Consider the following program:

$$\mathtt{o_T} := \mathtt{i_U}\ \mathtt{xor}\ \mathtt{j_T} \tag{2}$$

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 (1) of $\mathcal{C}_1$, the quantity of contamination caused by program (2) 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 (2).

[3]Hence, readers familiar with that work will be unsurprised by our final definition of expected contamination in equation (6), and perhaps unsurprised by the development leading up to it. But we present the full development 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.

[4]Flows from trusted to untrusted need not be prohibited. So the attacker could be allowed to read trusted inputs or outputs, and the user could be allowed to write untrusted inputs. But for simplicity, we don't consider those flows in this paper.

[5]Distribution T_{out} could 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} [23].

[6]Some readers might be more familiar with $I(\cdot\,;\cdot)$ as a notation for mutual information. We use a comma, rather than a semi-colon, to emphasize that the notation is symmetric.

B. 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:

$$\mathtt{o_T} := \mathtt{i_U} \,\&\, \mathtt{j_T} \tag{3}$$

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[7] the least significant bit of $\mathtt{i_U}$ (so there is 1 bit of contamination); `0x0011` yields the two least significant bits (2 bits of contamination), etc. But if a user executes the program twice, supplying first `0x0001` then `0x0011`, 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.[8] Let U_j be a random variable representing the user's accumulated knowledge in execution j about the untrusted input event, and let t^j_{out} and t^j_{in} be the trusted input and output events in that execution. The distribution of U_{j+1} is defined in terms of the distribution of U_j:

$$\Pr(U_{j+1} = u_{in}) = \Pr(U_j = u_{in} \,|\, t^j_{out}, t^j_{in}). \tag{4}$$

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:

$$\vec{\mathcal{C}} = \sum_j I(u^j_{in}, t^j_{out} \,|\, t^j_{in}),$$

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

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

[8]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 §IV.

Returning to program (3), 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 1 bit by supplying `0x0011` 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.

C. Contamination in Expectation

$\mathcal{C}_1$ quantifies contamination in a single execution. It could be used at runtime by an execution monitor [24] to constrain how much contamination occurs during 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 our 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 $\mathcal{C}_1$:

$$\mathcal{C} = \mathsf{E}[\mathcal{C}_1]. \tag{5}$$

The right-hand side of equation (5) can be rewritten as the mutual information $\mathcal{I}(U_{in}, T_{out} \,|\, T_{in})$ between distributions U_{in} and T_{out}, conditioned on observation of T_{in}. That yields our definition of expected contamination:

$$\mathcal{C} \triangleq \mathcal{I}(U_{in}, T_{out} \,|\, T_{in}). \tag{6}$$

Definition (6) 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 [18] 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.

D. Leakage

Clark et al. [15], [22] define quantity $\mathcal{L}$ of leakage from secret inputs S_{in} to public outputs P_{out}, given public inputs P_{in}, as follows:

$$\mathcal{L} \triangleq \mathcal{I}(S_{in}, P_{out} \,|\, P_{in}). \tag{7}$$

Replacing "untrusted" with "secret" and "trusted" with "public" in equation (6) yields equation (7). Contamination and leakage 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

Figure 2. Channel suppression model

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 [14] extends to these quantitative models.

III. Quantification of Suppression

A. 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. The program models a channel in which inputs are messages, and the receiver attempts to determine what messages were sent. 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.

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. For simplicity, the attacker is not allowed to read trusted inputs or outputs. This communication model for channel suppression is depicted in figure 2.

We first define channel suppression for single executions. As in our model of contamination, let t_{in} and t_{out} be trusted input and trusted output events. Since $I(t_{in}, t_{out})$ is the quantity of information obtained about trusted inputs by observing trusted outputs, $I(t_{in}, t_{out})$ is the quantity $\mathcal{CT}_1$ of *channel transmission* from the sender to the receiver in a single execution:

$$\mathcal{CT}_1 \triangleq I(t_{in}, t_{out}). \tag{8}$$

Let $I(t_{in}|t_{out})$ denote the information conveyed by the occurrence of event t_{in}, conditioned on observation of the occurrence of t_{out}. We can rewrite the right-hand side of equation (8):

$$\mathcal{CT}_1 = I(t_{in}) - I(t_{in}|t_{out}). \tag{9}$$

Quantity $I(t_{in})$ is the total 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.[9] Therefore, $I(t_{in}|t_{out})$ is the quantity $\mathcal{CS}_1$ of *channel suppression* in a single execution:

$$\mathcal{CS}_1 \triangleq I(t_{in}|t_{out}). \tag{10}$$

Although untrusted input u_{in} does not directly appear in equations (8) or (10), they 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 the definition (1) of contamination $\mathcal{C}_1$ conditions on u_{in}; equations (8) and (10) do not because the receiver cannot directly observe untrusted input—unlike the user, who could in the contamination model.

We next define channel suppression in expectation. Let $\mathcal{I}(T_{in}, T_{out})$ denote the mutual information between distributions T_{in} and T_{out}, and let $\mathcal{H}(T_{in}|T_{out})$ denote the entropy of distribution T_{in}, conditioned on observation of T_{out}. (As before, T_{in} and T_{out} are probability distributions on trusted inputs and trusted outputs.) 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 $\mathcal{CS}$:[10]

$$\mathcal{CT} \triangleq \mathcal{I}(T_{in}, T_{out}), \tag{11}$$

$$\mathcal{CS} \triangleq \mathcal{H}(T_{in}|T_{out}). \tag{12}$$

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.[11]

As an example, consider the following program:

$$\mathtt{o_T} := \mathtt{i_T}\ \mathtt{xor}\ \mathtt{rnd}(1) \tag{13}$$

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

[9] Alternatively, the right-hand side of equation (8) could be rewritten 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.

[10] 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.

[11] The basis of that interpretation would be the capacity of the channel from trusted inputs to trusted outputs (cf. §II-C).

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 $\mathtt{i}_\mathtt{T}$ by observing $\mathtt{o}_\mathtt{T}$.

Similarly, consider the following program:

$$\mathtt{o}_\mathtt{T} := \mathtt{i}_\mathtt{T} \texttt{ xor } \mathtt{j}_\mathtt{U} \tag{14}$$

Variable $\mathtt{j}_\mathtt{U}$ is a one-bit untrusted input. Suppose that untrusted input distribution U_{in} is uniform. Then program (14) exhibits the same behavior as program (13): 0 bits of channel transmission and 1 bit of channel suppression. Because of the bit of random noise added by the attacker, the receiver cannot learn anything about $\mathtt{i}_\mathtt{T}$ by observing $\mathtt{o}_\mathtt{T}$.

Programs (13) and (14) both cause 1 bit of channel suppression, but the source of that channel suppression is different. For program (13), the source is program randomness; for program (14), it is the attacker. We develop definitions that distinguish between these sources, next.

1) Attacker-controlled channel 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:

$$\mathcal{CS}_P \triangleq \mathcal{H}(T_{in} \,|\, T_{out}, U_{in}). \tag{15}$$

This definition differs from definition (12) 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 of channel suppression $\mathcal{CS}_A$ 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):

$$\mathcal{CS}_A \triangleq \max_{U_{in}}(\mathcal{CS}) - \min_{U_{in}}(\mathcal{CS}). \tag{16}$$

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

For program (13), 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 (14), which exhibits 0 bits of program-controlled channel suppression and 1 bit of attacker-controlled channel suppression.

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

$$\mathtt{o2}_\mathtt{T} := \mathtt{i2}_\mathtt{T} \texttt{ xor } \mathtt{i2}_\mathtt{U} \texttt{ xor } \mathtt{rnd}(1) \tag{17}$$

All variables in program (17) are two-bit. One bit of program-controlled channel suppression $\mathcal{CS}_P$ is caused by the xor with $\mathtt{rnd}(1)$. But the attacker controls the rest of the channel suppression. If the attacker chooses $\mathtt{i2}_\mathtt{U}$ uniformly at random, the channel suppression is maximized and equal to 2 bits; whereas if the attacker makes $\mathtt{i2}_\mathtt{U}$ a constant (e.g., always "00"), the channel suppression is the minimal 1 bit caused by $\mathtt{rnd}(1)$. Calculating $\mathcal{CS}_A$ yields 1 $(= 2 - 1)$ bit of attacker-controlled channel suppression.

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 [25], 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-neighbor decoding*: the nearest neighbor of a word w is a code-word c that is closest to w by the *Hamming distance*. (The nearest neighbor is not necessarily unique for some codes, in which case an arbitrary nearest neighbor is chosen.) Treating words as vectors, Hamming distance $d(w, x)$ between words w and x is the number of positions i at which $w_i \neq x_i$. For the repetition code, nearest-neighbor 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:

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

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 $\mathtt{rnd}(\mathtt{x})$, used in program (13), 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. So quantity $\mathcal{CS}_1$ of channel suppression is $-\log p$. 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 $R_3(\mathtt{BSC})$. The probability that $\mathtt{w} = \mathtt{m}$ holds is now $p^3 + 3p^2(1-p)$, which can be derived by a simple argument. (See the accompanying technical report [19].) The amount of channel suppression $\mathcal{CS}_1$ is thus $-\log(p^3 + 3p^2(1-p))$, which is less than $-\log p$ for any $p > \frac{1}{2}$.

So for any $p > \frac{1}{2}$ (i.e., for any channel at least slightly biased toward correct transmission) the channel suppression

from $R_3(\texttt{BSC})$ is less than the channel suppression from BSC. 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 coding theory.

3) Channel suppression vs. contamination: Recall program (14), restated here:

$$\texttt{o}_\texttt{T} := \texttt{i}_\texttt{T}\ \texttt{xor}\ \texttt{j}_\texttt{U}$$

This program is essentially the same as program "o := t xor u" from §I. We previously analyzed program (14) 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 analyze the program for contamination: $\texttt{j}_\texttt{T}$ is supplied by a user, and $\texttt{o}_\texttt{T}$ 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 $\texttt{i}_\texttt{U}$. So this program exhibits both contamination and channel suppression, as we argued in §I.

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 contamination concerns injection of information (here 1 bit of untrusted information is injected), whereas suppression concerns loss of information (here 1 bit of trusted information is lost).

Also, recall program (13), restated here:

$$\texttt{o}_\texttt{T} := \texttt{i}_\texttt{T}\ \texttt{xor}\ \texttt{rnd}(1)$$

This program is essentially the same as program "o := t xor n" from §I. We previously determined that program (13) 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 §I.[12]

4) Channel suppression and leakage: Recall that leakage (7) is the quantity $\mathcal{L}$ of information flow from secret inputs to public outputs. Leakage can be prevented by employing channel suppression. Consider a *declassifier* that accepts trusted, secret inputs and produces trusted, public outputs. 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. Let s denote a secret input event and let p denote a public output event. Let $I(s)$ denote the self-information of event s. Let $\mathcal{L}_1$ denote the leakage in a single execution of the declassifier and be defined as $I(s, p)$; this definition follows from equation (7) by removing the conditioning on P_{in} (since the declassifier has no public inputs) and by removing the expectation over all executions from the definition of mutual information $\mathcal{I}$.

[12]These arguments implicitly assume that random number generator $\texttt{rnd}(\cdot)$ is trusted. Untrusted generators could also be modeled, but we don't pursue that here.

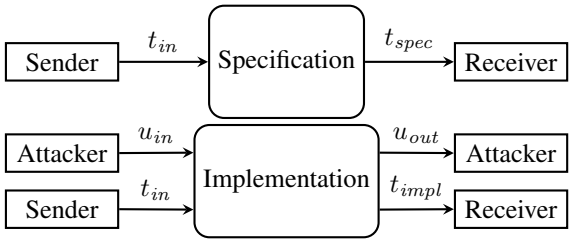

Figure 3. Program suppression model

Proposition 1. $\mathcal{L}_1 + \mathcal{CS}_1 = I(s)$.

So for a given probability distribution of high inputs, leakage plus channel suppression is a constant. Confidentiality is obtained by eroding integrity, and vice versa. Any security condition for declassifiers—we discuss some in §V—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.

B. Program Suppression

We now generalize the idea of suppression from communication channels to program correctness. Consider a *specification* program, depicted in figure 3: 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* program 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}.

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 must be executed with the same trusted input. We require T_{spec} to be a (deterministic) function of its input:

$$\mathcal{H}(T_{spec}|T_{in}) = 0. \tag{18}$$

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:

$$\mathcal{PT}_1 \triangleq I(t_{spec}, t_{impl}), \tag{19}$$

$$\mathcal{PS}_1 \triangleq I(t_{spec}|t_{impl}), \tag{20}$$

$$\mathcal{PT} \triangleq \mathcal{I}(T_{spec}, T_{impl}), \tag{21}$$

$$\mathcal{PS} \triangleq \mathcal{H}(T_{spec}|T_{impl}). \tag{22}$$

The rationale for these definitions remains unchanged from our development of channel transmission and suppression. Note that the attacker's influence is accounted for 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 (18).) In §III-A, 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 $\mathcal{CS}$).

Before turning to more compelling examples, we consider the following specification as a corner case:

$$\mathtt{o_T} := 42$$

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.

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. Implementation `UnderSum` exhibits such an error by omitting 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 each array element `a[i]` is identically, independently distributed according to a binomial distribution with parameters n and p. Let $Bin(n, p)$ denote this distribution.[13] Also suppose that the value found in `a[m]` is uniformly distributed on integer interval $[0, 2^j - 1]$; let $Unif(0, 2^j - 1)$ denote this distribution. We consider elements `a[0]`..`a[m-1]` to be properly initialized according to their binomial distribution and therefore to be trusted. But `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.

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

$$\mathcal{PS}_{\mathtt{US}} = \sum_{\substack{s' \in Bin(n,p),\\ i \in Bin(n(m-1),p)}} \Pr(s')\Pr(i)\log\Pr(s'). \tag{23}$$

(The full calculation of $\mathcal{PS}_{\mathtt{US}}$, as well as the calculations for equations (24) and (25) below, appears in the accompanying technical report [19].) So if $\mathtt{m} = 10$, $n = 1$, and $p = 0.5$, then $\mathcal{PS}_{\mathtt{US}}$ 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]`:

$$\mathcal{PS} = \mathcal{H}(Bin(n, p)). \tag{24}$$

Indeed, it is straightforward to reduce equation (23) to equation (24). Hence, `UnderSum` suppresses exactly the information about the omitted array element.

[13] 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 $\mathcal{D}$—and because it illustrates that our theory is not limited to uniform distributions.

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

$$\mathcal{PS}_{\texttt{OS}} = \sum_{\substack{s \in Bin(mn,p),\\ i' \in Unif(0,2^j-1)}} 2^{-j} \Pr(s) \log \frac{2^{-j} \Pr(s)}{\Pr(s+i')}. \quad (25)$$

Now if $\texttt{m} = 10$, $n = 1$, $p = 0.5$, and $j = 1$, then $\mathcal{PS}_{\texttt{OS}}$ is about 0.93 bits. The 1 bit of randomness added by the attacker through `a[m]`, which is uniformly distributed on $\{0, 1\}$, 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. As m increases, $\mathcal{PS}_{\texttt{OS}}$ approaches 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.[14]

2) Probabilistic specifications: Equation (18) requires specifications to be deterministic. Consider eliminating that requirement and allowing probabilistic specifications—for example, "$\texttt{o}_\texttt{T} := \texttt{rnd}(1)$". This specification 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. And 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. When quantifying suppression of correct outputs, we needed a probability distribution on outputs to model the receiver's uncertainty. To quantify suppression of correct distributions, we would need an extra level of distributions: a probability distribution on a probability distribution on outputs. So far, we have modeled 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). We leave this mathematical upgrade as future work.

3) 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 a dual to suppression.

Other notions of integrity also lack obvious confidentiality duals—for example, the Clark–Wilson [4] integrity policy for commercial organizations, based on well-formed transactions and verification procedures. Apparently, the Biba duality goes only so far.

4) Suppression vs. availability: If a program (or channel) suppresses all its input, the receiver gains no information. It might at first seem as though the program has made the correct output unavailable, so we might be tempted to conclude that suppression measures availability rather than integrity. However, availability is usually concerned with timely response—not with quality of information—whereas suppression is concerned with quality, not timeliness. Furthermore, techniques typically employed to prevent suppression differ from those for improving availability. For example, error-correcting codes defend against (channel) suppression, but they do not improve availability—if a channel goes down (e.g., a wire is cut), a code cannot restore communication. And replication improves availability but potentially introduces (program) suppression, because different replicas might provide different responses and combining those responses might yield incorrect output. Hashes and digital signatures are therefore used in conjunction with replication to increase integrity by ensuring correct output.

Complete absence of information could be viewed as complete confidentiality, complete loss of integrity, or complete unavailability. Thus some quantitative relaxation of "complete absence" could yield quantitative characterizations of confidentiality, integrity, or availability. So there might be some interesting relationships—perhaps even new dualities—still to be found.

IV. Integrity and Beliefs

In our models of contamination and suppression, inputs are chosen according to probability distributions. For example, the user assumes that untrusted inputs are chosen according to distribution U_{in} in our model of contamination. But the user could be wrong—the inputs could be chosen according to a different distribution; a calculation of information flow would then need to incorporate both distributions.

Clarkson et al. [16], [27] show how to quantify leakage from secret inputs to public outputs when attackers have (possibly 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.

[14] This kind of analysis might be used to provide a mathematical explanation of why *failure-oblivious computing* (FOC) [26] 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.

A. Contamination and Beliefs

Define a *belief* to be a probability distribution of untrusted inputs. 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, enabling the user 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 D such that $D(X \twoheadrightarrow Y)$ is the inaccuracy of belief X about reality Y, where Y is also a distribution. Intuitively, $D(X \twoheadrightarrow Y)$ is the distance from the belief to reality. In previous work [16], we showed that *relative entropy* can successfully instantiate D:

$$D(X \twoheadrightarrow Y) \triangleq \sum_x \Pr(Y = x) \log \frac{\Pr(Y = x)}{\Pr(X = x)}. \tag{26}$$

The right-hand side of this definition is the relative entropy[15] between Y and X.

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

$$D(X \twoheadrightarrow x) \triangleq -\log \Pr(X = x). \tag{27}$$

Equation (27) follows from (26) by setting Y to be a distribution that assigns probability 1 to event x. This simplified definition is equivalent to self-information—that is,

$$D(X \twoheadrightarrow x) = I(x), \tag{28}$$

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:

$$\mathcal{C}_B \triangleq D(U_{in} \twoheadrightarrow u_{in}) - D(U'_{in} \twoheadrightarrow u_{in}). \tag{29}$$

In previous work [16], 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 (4): U'_{in} equals U_{in} conditioned on t_{in} and t_{out}. Furthermore, the quantity of contamination according to $\mathcal{C}_B$ equals the quantity of contamination according to $\mathcal{C}_1$ (1).

Theorem 1. $\mathcal{C}_B = \mathcal{C}_1$.

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

[15]The traditional notation for the relative entropy between Y and X is $D(Y \parallel X)$, but we use notation $D(X \twoheadrightarrow Y)$ to emphasize the asymmetry between the two distributions. Also, we abuse notation by treating distributions as random variables in the probability terms.

Moreover, define belief U_{in} to be *correct* if the attacker chooses u_{in} by sampling user prebelief U_{in}—that is, if the user is correct about how untrusted inputs are chosen. Then applying the expectation operator to both sides of theorem 1, we have that the expected quantity of contamination of beliefs equals the expected quantity of contamination according to $\mathcal{C}$ (6).

Corollary 1. *U_{in} is correct implies $\mathsf{E}[\mathcal{C}_B] = \mathcal{C}$.*

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

Corollary 1 can also be understood in terms of leakage by applying the duality of contamination $\mathcal{C}$ and leakage $\mathcal{L}$ (7). If the attacker's distribution S_{in} on secret inputs is correct, 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 1 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.

B. 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:

$$\mathcal{CT}_B \triangleq D(T_{in} \twoheadrightarrow t_{in}) - D(T'_{in} \twoheadrightarrow t_{in}). \tag{30}$$

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

$$\mathcal{CS}_B \triangleq D(T'_{in} \twoheadrightarrow t_{in}). \tag{31}$$

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

Theorem 2. *$\mathcal{CT}_B = \mathcal{CT}_1$ and $\mathcal{CS}_B = \mathcal{CS}_1$.*

Corollary 2. *(T_{in}, U_{in}) is correct implies $\mathsf{E}[\mathcal{CT}_B] = \mathcal{CT}$ and $\mathsf{E}[\mathcal{CS}_B] = \mathcal{CS}$.*

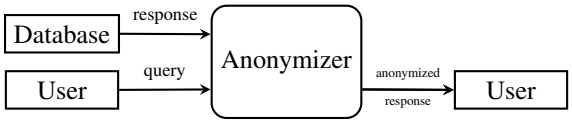

Figure 4. Anonymizer model

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 (30) and (31):

$$\mathcal{PT}_B \triangleq D(T_{spec} \twoheadrightarrow t_{spec}) - D(T'_{spec} \twoheadrightarrow t_{spec}), \quad (32)$$

$$\mathcal{PS}_B \triangleq D(T'_{spec} \twoheadrightarrow t_{spec}). \quad (33)$$

We obtain the obvious result:

Corollary 3. *$\mathcal{PT}_B = \mathcal{PT}_1$ and $\mathcal{PS}_B = \mathcal{PS}_1$. Further, (T_{in}, U_{in}) is correct 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.

V. Case Study: Database Privacy

Databases that contain information about individuals sometimes must respond to queries in a way that protects the privacy of those individuals. Such databases often will employ an *anonymizer* to suppress information about individuals. We model the anonymizer as a program that receives two inputs, as depicted in figure 4. The first input is the user's *query*. The second input is a *response* computed by the database with the user's query. Both inputs are trusted by the anonymizer and by the user. The response contains information from the database—perhaps even its entire contents—so the response is secret. The query, however, is public because it contains no sensitive information about individuals. The anonymizer produces an *anonymized response* as output.[16] The anonymized response is trusted by the user and is public, because it (presumably) has been anonymized. Although the query and responses might involve statistics (e.g., sums or averages) computed from individuals' information, we do not restrict our consideration to any particular statistics. Our model is agnostic about the domains of queries and data.

The user attempts to learn secret information about individuals through queries. The anonymizer should leak some information to the user; otherwise, interacting with it would be pointless. And the anonymizer acts as a noisy communication channel, where the database is the sender and the user is the receiver.[17] The anonymizer suppresses some information from this channel's outputs to protect privacy. By proposition 1, the amount of leakage plus the amount of channel suppression is a constant that depends on the distribution of database content. This is sensible—whatever the anonymizer doesn't suppress, it leaks.

The quantitative frameworks we have developed for integrity and confidentiality yield a nuanced characterization of database privacy. We demonstrate this by analyzing two popular security conditions, k-anonymity [28] and ℓ-diversity [29]. For each, we are able to offer an information-theoretic characterization of the security condition.

A. k-anonymity

Sweeney [28] proposes *k-anonymity*, a security condition for anonymizers, which requires every individual to be anonymous within some set (of individuals) of size at least k. For example, if Alice was born Nov. 26, 1865, and if gender and birth date are both published, then at least $k-1$ other females born that day must be included in the published data. If the original database does not contain at least that many individuals, the data must be changed in some way to satisfy k-anonymity. Sweeney proposes *generalization*, which hierarchically replaces attributes values with less specific values. For example, Alice's birth date might be replaced by Nov. 1865, by 1865, or even by 18**. Generalization improves confidentiality by obscuring identities, but it diminishes the information conveyed—that is, generalization corrupts integrity. That tradeoff is unsurprising in light of proposition 1. Sweeney quantifies the integrity of generalized data with a *precision* metric that is based on the generalization hierarchy and the domains used in it.

Adapting Sweeney's insight to information flow, we could imagine requiring that the public output of a program corresponds to at least k possible secret inputs. This requirement would make any particular input be anonymous within a set of size k. We have the tools to analyze how generalization affects our notions of leakage and channel suppression.[18] As an example, consider generalization of birth dates. Assume that birth dates are uniformly distributed within a given year—for example, 1865.[19] Then, according to our definitions, a program that outputs the entire input date leaks about 8.5 bits and suppresses 0 bits; a program that outputs just the month and year leaks about 3.6 bits and suppresses about 4.9 bits; and a program that outputs just the year leaks 0 bits and suppresses about 8.5 bits.

[16] The anonymizer might also produce some output about the anonymization it just performed, and this output might be stored in the database and used during future anonymizations.

[17] Alternatively, we could use program suppression to model anonymizers instead of channel suppression. The anonymizer would be an implementation program; the specification would be the query evaluator. Channel suppression is simpler—it does not require modeling the query evaluator.

[18] Sweeney defines "suppression" differently than we do; she uses it to mean the complete removal of an individual's information from the output.

[19] Birth dates are, in reality, probably not uniformly distributed [30].

Moreover, leakage and channel suppression enable an information-theoretic understanding of generalization. Channel suppression quantifies how much information is lost because of generalization, whereas Sweeney's precision metric has no obvious information-theoretic interpretation. And leakage quantifies how much information is released despite generalization, whereas k-anonymity makes no guarantees on how much information might be leaked. For example, suppose that published data includes a medical diagnosis and a favorite pet. If it is known that Alice's favorite pet is a cat and that the rest of the individuals in the population are highly unlikely to have a cat as a favorite pet, then Alice's medical diagnosis could be inferred with high probability. Thus information about Alice would be leaked despite k-anonymity. As another example, if a program's output could have been caused by any one of k possible inputs, but one of those inputs is much more probable than the rest, then information about the input would be leaked despite k-anonymity. These kinds of leakage—made possible by the attacker's background knowledge—were discovered by Machanavajjhala et al. [29], who invented a new criterion, ℓ-diversity. We turn to that, next.

B. ℓ-diversity

The *principle of ℓ-diversity* [29] 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 background knowledge attacks, which depend on some sensitive values having significantly higher probability than the rest.

Machanavajjhala et al. [29] give an instantiation of the ℓ-diversity principle based on entropy, as follows. Define a *block* to be a set of tuples in which each tuple 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 tuples corresponding to individuals whose birth date is 18** and whose favorite pet is a cat. However, individuals 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	Favorite pet	Diagnosis
18**	cat	cancer
18**	cat	cancer
18**	cat	heart disease
18**	cat	influenza

For each such probability distribution B constructed from a block of published data, *entropy ℓ-diversity* requires that $\mathcal{H}(B) \geq \log \ell$ holds, where $\mathcal{H}(B)$ denotes the entropy of distribution B.[20] Applying this definition, we have that the block above is 1.5-diverse. Notice that it is not 2-diverse because the two most frequent sensitive values (either cancer and heart disease, or cancer and influenza) do not occur with roughly equal probability—cancer is twice as likely as the other diagnoses.

More generally, consider any block with distribution B that satisfies entropy ℓ-diversity. The entropy of a uniform distribution of ℓ events is $\log \ell$. So if $\mathcal{H}(B) \geq \log \ell$, 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.

We now recast entropy ℓ-diversity in terms of information flow. In the example above, B is the distribution on the diagnosis of an arbitrary patient that results from observing the block. More generally, B is the distribution on trusted (secret) inputs that results from observing a block, which is a trusted (public) output, under the assumption that the observer's initial distribution T_{in} on inputs is uniform. (Were it not uniform, B would be a function of the block's empirical distribution and the observer's initial distribution.) Hence, $B = T_{in}|t_{out}$. And since $\mathcal{H}(B) \geq \log \ell$, we have that $\mathcal{H}(T_{in}|t_{out}) \geq \log \ell$ for any t_{out} produced by the anonymizer. We can use this fact to obtain a bound on the anonymizer's channel suppression:

$$\begin{aligned}
\mathcal{CS} &= \langle \text{equation (12)} \rangle \\
&\quad \mathcal{H}(T_{in}|T_{out}) \\
&= \langle \mathcal{H}(X|Y) = \mathsf{E}_{y \in Y}[\mathcal{H}(X|y)] \rangle \\
&\quad \mathsf{E}_{t_{out} \in T_{out}}[\mathcal{H}(T_{in}|t_{out})] \\
&\geq \langle \text{fact above} \rangle \\
&\quad \mathsf{E}_{t_{out} \in T_{out}}[\log \ell] \\
&= \langle \text{expectation of constant} \rangle \\
&\quad \log \ell.
\end{aligned}$$

So we have that $\mathcal{CS} \geq \log \ell$. As a straightforward consequence of its definition, entropy ℓ-diversity therefore enforces a bound on channel suppression.

Interpreting that bound, suppose that an individual is in the block from which B was constructed, and suppose that T_{in} is uniform—meaning that the individual is equally likely to have any value for his sensitive attributes. Then B yields the probability distribution on that individual's sensitive attributes that results from observing the published block. Entropy ℓ-diversity requires at least $\log \ell$ bits of uncertainty in that distribution. So at least $\log \ell$ bits of information are suppressed about the individual's sensitive attributes.

However, entropy ℓ-diversity does not directly place a bound on the amount of information that may be transmitted;

[20] The definition of entropy ℓ-diversity originates with Øhrn and Ohno-Machado [31].

beyond the $\log \ell$ bits that are suppressed, there might be many bits that are transmitted about an individual. For example, there might be

- a lot of information about the individual (e.g., an entire DNA sequence) already present in the input, little of which is suppressed; or
- a lot of background knowledge about the individual already possessed by the user, enabling inference of a lot of information from the output.

To measure the *utility* of published data—that is, how useful the data is for studying the characteristics of a population—Machanavajjhala et al. [29] and Kifer and Gehrke [32] use an information-theoretic metric called *Kullback-Leibler divergence*. This metric is another name for relative entropy D (26). Let B be an empirical probability distribution of sensitive attributes, as constructed above from anonymized data.[21] And let R be an empirical distribution similarly constructed from the original (non-anonymous) data. Their utility measure is the relative entropy of B to R—that is, $D(B \twoheadrightarrow R)$. Notice that the best possible utility is 0, meaning that B and R are the same distribution, and that the higher the utility is, the less the distributions are alike. So we call this metric *anti-utility*.

Again recasting in terms of information flow, note that anti-utility is the distance between two distributions: an empirical distribution of trusted inputs, after observing trusted outputs; and an empirical distribution of trusted inputs. Were we to ignore the "empirical" part of that characterization, we could say that anti-utility is $D(T_{in}|t_{out} \twoheadrightarrow T_{in})$, which is the expectation of $D(T_{in}|t_{out} \twoheadrightarrow t_{in})$ with respect to t_{in}. That latter quantity is $\mathcal{CS}_B$ (31), because $T'_{in} = T_{in}|t_{out}$. And by corollary 2, expected belief-based channel suppression $\mathsf{E}[\mathcal{CS}_B]$ is equal to information-theoretic channel suppression $\mathcal{CS}$. So anti-utility would be the quantity of channel suppression if we used real, instead of empirical, distributions.[22] This equivalence is sensible, because the less suppression data suffers, the more useful it is.

VI. Related Work

Research on quantification of information flow began with analysis of covert channels, and progress has been made from theoretical definitions to automated analyses [33]–[37]. Quantification of integrity and corruption is a relatively new line of research.

Newsome, McCamant, and Song [17] implement a dynamic analysis that automatically quantifies attacker influence in real-world programs. They quantify the influence an attacker can exert over the execution of a program as the logarithm of the size of the set of possible outputs. This quantity is the same as our contamination $\mathcal{C}_1$ in a single execution, assuming that programs are deterministic and that all inputs are either under the control of the attacker or are fixed constants. But our definition of $\mathcal{C}_1$ allows probabilistic programs, trusted inputs that are not under the control of the attacker, and arbitrary distributions on inputs and outputs.

Heusser and Malacaria [38] 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.

Biba [14] 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 (termed the "strict integrity policy") is dual to the Bell–LaPadula confidentiality policy [39]. But since Biba's motivating concern was guaranteeing that systems perform as their designers intended, correctness is also a critical piece of the integrity puzzle. Our program suppression measure $\mathcal{PS}$ addresses correctness; perhaps other quantitative notions of correctness, such as software testing metrics, could also be understood as addressing quantitative integrity.

Information-flow integrity policies have sometimes received less attention than their confidentiality counterparts. For example, early versions of Jif [40] (then called JFlow) did not include integrity policies, and Flow Caml [41] 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" [42]—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 [43]. So integrity cannot be easily dismissed, even when confidentiality is the primary concern.

Several recent systems use integrity policies in interesting ways. Jif-derived languages and systems [44]–[46] for building secure distributed applications incorporate integrity policies, enabling principals to specify fine-grained requirements on how their information may be affected by other principals. These policies drive automated partitioning of applications, in which computations can be assigned to principals who are sufficiently trusted to perform the computations. When no such principals exist, computations can be replicated and their results validated against each other to boost integrity. Flume [47]—a system that integrates information flow with operating system abstractions such as processes, pipes, and sockets—also incorporates

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

[22] Definitions of anti-utility [29], [32] use empirical distributions because they deal with concrete databases and anonymizations.

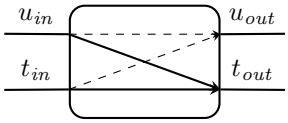

Figure 5. Information-flow integrity in a program

integrity policies, preventing (e.g.) untrusted dynamically-loaded code from affecting information in the process that loads it. Airavat [48] integrates information flow with MapReduce [49] and differential privacy [50], providing confidentiality and integrity for MapReduce computations and automatically declassifying computation results if they do not violate differential privacy.

VII. Concluding Remarks

When we began this work, we thought we could simply apply Biba's confidentiality–integrity 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 same as the classical information-theoretic model of quantitative integrity, which we named channel suppression. We later discovered that channel suppression could be generalized to characterize 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 [4] integrity policy, which 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 cannot even attempt to prove that contamination and suppression are sufficient to express all integrity properties, because we lack a formal definition of integrity. But we can gain some insight by reviewing the information-flow model we have used in this paper, depicted in figure 5. The solid 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 dashed arrows 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. Other kinds of integrity must exist outside it.

Finally, our work exemplifies how measurement can drive research, even in computer security. In an effort to measure integrity, we came to disentangle suppression from contamination. We also bridged a gap between database privacy and quantitative information-flow security. 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 meagre [*sic*] 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[23]

Acknowledgments

Supported in part by ONR grant N00014-09-1-0652, AFOSR grant F9550-06-0019, National Science Foundation grants 0430161 and CCF-0424422 (TRUST), and a gift from Microsoft Corporation.

We thank Steve Chong and Andrew Myers for discussions about this work. We also thank Aslan Askarov, Steve Chong, Johannes Gehrke, Boris Köpf, Andrew Myers, and the anonymous CSF reviewers for comments on a draft of this paper.

References

[1] D. Denning, *Cryptography and Data Security*. Reading, Massachusetts: Addison-Wesley, 1982.

[2] J. Millen, "Covert channel capacity," in *Proc. IEEE Symposium on Security and Privacy*, Apr. 1987, pp. 60–66.

[3] V. L. Voydock and S. T. Kent, "Security mechanisms in high-level network protocols," *Computing Surveys*, vol. 15, no. 2, Jun. 1983.

[4] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies," in *Proc. IEEE Symposium on Security and Privacy*, Apr. 1987, pp. 184–194.

[5] International Organization for Standardization, "Information processing systems: Open systems interconnection—basic reference model, Part 2: Security architecture, ISO 7498-2," 1989.

[6] Commission of the European Communities (ECSC, EEC, EAEC), "Information Technology Security Evaluation Criteria: Provisional harmonised criteria," Jun. 1991, document COM(90) 314, Version 1.2.

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

[8] "Common Criteria for Information Technology Security Evaluation: Part 1: Introduction and general model," Sep. 2005, cCMB-2006-09-001, Version 3.1, Revision 1. Available from www.commoncriteriaportal.org.

[9] L. Wall, T. Christiansen, and R. L. Schwartz, *Programming Perl*, 2nd ed. Sebastopol, California: O'Reilly, 1996.

[23] 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 in [51].

[10] G. E. Suh, J. W. Lee, D. Zhang, and S. Devedas, "Secure program execution via dynamic information flow tracking," in *Proc. ACM Conference on Architectural Support for Programming Languages and Systems*, Oct. 2004, pp. 85–96.

[11] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *Proc. USENIX Security Symposium*, Aug. 2005, pp. 271–286.

[12] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software," in *Proc. Symposium on Network and Distributed System Security*, Feb. 2005, available from http://www.isoc.org/isoc/conferences/ndss/05/proceedings/papers/taintcheck.pdf.

[13] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in *Proc. USENIX Security Symposium*, Aug. 2006, pp. 121–136.

[14] K. Biba, "Integrity considerations for secure computer systems," MITRE Corporation, Tech. Rep. MTR-3153, Apr. 1977.

[15] D. Clark, S. Hunt, and P. Malacaria, "Quantitative information flow, relations and polymorphic types," *Journal of Logic and Computation*, vol. 18, no. 2, pp. 181–199, 2005.

[16] M. R. Clarkson, A. C. Myers, and F. B. Schneider, "Quantifying information flow with beliefs," *Journal of Computer Security*, vol. 17, no. 5, pp. 655–701, 2009.

[17] J. Newsome, S. McCamant, and D. Song, "Measuring channel capacity to distinguish undue influence," in *Proc. ACM Workshop on Programming Languages and Analysis for Security*, Jun. 2009, available from http://doi.acm.org/10.1145/1554339.1554349.

[18] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423, 623–656, 1948.

[19] M. R. Clarkson and F. B. Schneider, "Quantification of integrity," Cornell University Computing and Information Science Technical Report, http://hdl.handle.net/1813/14470, May 2010.

[20] D. S. Jones, *Elementary Information Theory*. Oxford: Clarendon Press, 1979.

[21] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. New York: John Wiley & Sons, 1991.

[22] D. Clark, S. Hunt, and P. Malacaria, "A static analysis for quantifying information flow in a simple imperative language," *Journal of Computer Security*, vol. 15, no. 3, pp. 321–371, 2007.

[23] D. Kozen, "Semantics of probabilistic programs," *Journal of Computer and System Sciences*, vol. 22, pp. 328–350, 1981.

[24] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security*, vol. 3, no. 1, pp. 30–50, 2000.

[25] J. Adámek, *Foundations of Coding*. New York: John Wiley and Sons, 1991.

[26] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr., "Enhancing server availability and security through failure-oblivious computing," in *Proc. USENIX Symposium on Operating System Design and Implementation*, Dec. 2004, pp. 303–316.

[27] M. R. Clarkson, A. C. Myers, and F. B. Schneider, "Belief in information flow," in *Proc. IEEE Computer Security Foundations Workshop*, Jun. 2005, pp. 31–45.

[28] L. Sweeney, "Achieving k-anonymity privacy protection using generalization and suppression," *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, vol. 10, no. 5, pp. 571–588, 2002.

[29] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam, "ℓ-diversity: Privacy beyond k-anonymity," *ACM Transactions on Knowledge Discovery from Data*, vol. 1, no. 1, 2007.

[30] R. Murphy, "An analysis of the distribution of birthdays in a calendar year," http://www.panix.com/~murphy/bday.html, accessed Dec. 29, 2009.

[31] A. Øhrn and L. Ohno-Machado, "Using Boolean reasoning to anonymize databases," *Artificial Intelligence in Medicine*, vol. 15, no. 3, pp. 235–254, 1999.

[32] D. Kifer and J. Gehrke, "Injecting utility into anonymized datasets," in *Proc. ACM Conference on Management of Data*, Jun. 2006, pp. 217–228.

[33] J. W. Gray, III, "Toward a mathematical foundation for information flow security," in *Proc. IEEE Symposium on Security and Privacy*, May 1991, pp. 21–35.

[34] G. Lowe, "Quantifying information flow," in *Proc. IEEE Computer Security Foundations Workshop*, Jun. 2002, pp. 18–31.

[35] D. Clark, S. Hunt, and P. Malacaria, "Quantified interference for a while language," *Electronic Notes in Theoretical Computer Science*, vol. 112, pp. 149–166, Jan. 2005.

[36] S. McCamant and M. D. Ernst, "Quantitative information flow as network capacity," in *Proc. ACM Conference on Programming Language Design and Implementation*, Jun. 2008, pp. 193–205.

[37] M. Backes, B. Köpf, and A. Rybalchenko, "Automated discovery and quantification of information leaks," in *Proc. IEEE Symposium on Security and Privacy*, May 2009, pp. 141–153.

[38] J. Heusser and P. Malacaria, "Applied quantitative information flow and statistical databases," in *Workshop on Formal Aspects in Security and Trust*, Nov. 2009.

[39] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," MITRE Corporation, Tech. Rep. 2547, Volume I, Mar. 1973.

[40] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proc. ACM Symposium on Principles of Programming Languages*, 1999, pp. 228–241.

[41] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 1, pp. 117–158, Jan. 2003.

[42] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Untrusted hosts and confidentiality: Secure program partitioning," in *Proc. ACM Symposium on Operating Systems Principles*, Oct. 2001, pp. 1–14.

[43] S. Zdancewic and A. C. Myers, "Robust declassification," in *Proc. IEEE Computer Security Foundations Workshop*, Jun. 2001, pp. 15–23.

[44] S. Chong, K. Vikram, and A. C. Myers, "SIF: Enforcing confidentiality and integrity in web applications," in *Proc. USENIX Security Symposium*, Aug. 2007, pp. 1–16.

[45] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web applications via automatic partitioning," in *Proc. ACM Symposium on Operating Systems Principles*, Oct. 2007, pp. 31–44.

[46] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers, "Fabric: A platform for secure distributed computation and storage," in *Proc. ACM Symposium on Operating Systems Principles*, Oct. 2009, pp. 321–334.

[47] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard OS abstractions," in *Proc. ACM Symposium on Operating Systems Principles*, Oct. 2007, pp. 321–334.

[48] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and privacy for MapReduce," in *Proc. USENIX Symposium on Networked Systems Design and Implementation*, Apr. 2010, pp. 297–312.

[49] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. USENIX Symposium on Operating System Design and Implementation*, Dec. 2004, pp. 137–150.

[50] C. Dwork, "Differential privacy," in *Proc. International Colloquium on Automata, Languages and Programming*, Jul. 2006, pp. 1–12.

[51] E. W. Scripture, "The need of psychological training," *Science*, vol. 19, no. 474, pp. 127–128, Mar. 1892, available from http://www.jstor.org/stable/1766918.