2013 IEEE 26th Computer Security Foundations Symposium

# Probabilistic Point-to-Point Information Leakage

Tom Chothia*, Yusuke Kawamoto*, Chris Novakovic* and David Parker*
*School of Computer Science
University of Birmingham, Birmingham, UK

***Abstract*—The outputs of a program that processes secret data may reveal information about the values of these secrets. This paper develops an information leakage model that can measure the leakage between arbitrary points in a probabilistic program. Our aim is to create a model of information leakage that makes it convenient to measure specific leaks, and provide a tool that may be used to investigate a program's information security. To make our leakage model precise, we base our work on a simple probabilistic, imperative language in which secret values may be specified at any point in the program; other points in the program may then be marked as potential sites of information leakage. We extend our leakage model to address both non-terminating programs (with potentially infinite numbers of secret and observable values) and user input. Finally, we show how statistical approximation techniques can be used to estimate our leakage measure in real-world Java programs.**

***Keywords*-information leakage; probabilistic language; non-termination**

## I. Introduction

This paper presents a framework that can be used to measure information leaks between arbitrary points in a program. Our motivation is to allow a programmer or analyst examining source code to verify that the values stored in particular variables at particular times cannot be deduced from the values that will be visible to an attacker. Our framework introduces two annotations: secret, for marking the values stored in selected variables as *secret* at particular points in the program, and observe, for identifying variables as potential leakage sites due to an attacker's ability to *observe* their values at those points. After an analyst annotates their secret and observable variables, our framework measures how much information can be deduced about the secret values from the observable values.

As a motivating example, we consider the following hypothetical fragment of source code from a card game:

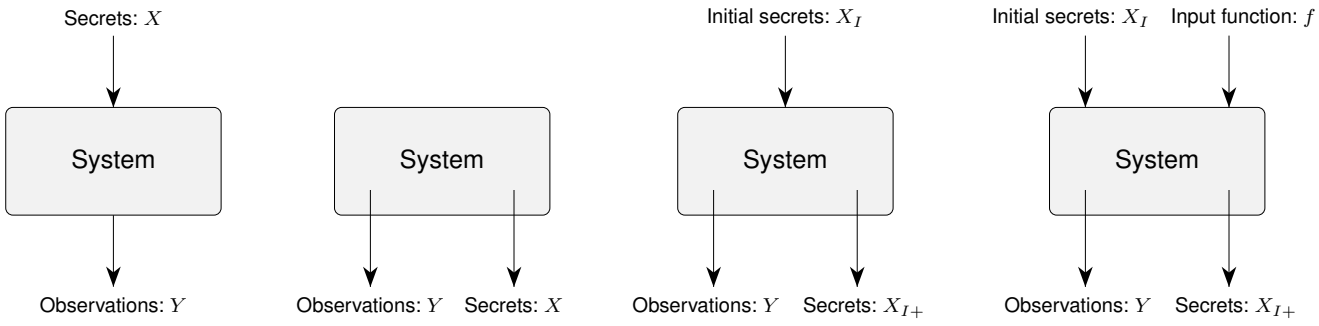
```
...
Card theirCard = deck.drawCard();
observe(theirCard);
socket.write(theirCard);
...
Card myCard = deck.drawCard();
secret(myCard);
...
determineWinner();
```

In this example, a `Card` object (`theirCard`) is drawn from the deck and sent over an insecure socket to a remote opposing player; since this is a site of a potential information leak, the object is marked as observable. Later, another `Card` object (`myCard`) is drawn from the deck, and the winner of the game then depends on this object in some way (e.g., on its face value); therefore, it is marked as secret. In this example, we wish to measure how much information an opponent learns about `myCard` by observing `theirCard` (rather than the randomness of `myCard` in isolation). The best possible implementation of this program provides the opponent observing `theirCard` with only a very small amount of information about the face value of `myCard` (i.e., that it is not the face value of `theirCard`). However, this may not be the case in poor implementations: if the deck is shuffled with a weak pseudorandom number generator (PRNG), the face value of `myCard` may be predictable from that of `theirCard`; in an even poorer implementation, the deck itself may be stored inside the `Card` object of `theirCard` somehow and the future face value of `myCard` is leaked entirely when `theirCard` is sent over the socket. A similar example is a program that generates a nonce that is exposed to a potential attacker, and then uses the same PRNG to generate a session key; the programmer would want to ensure that the session key cannot be guessed from the value of the nonce.

In this sense, our aim is to create a model of information leakage that makes it convenient to measure leaks between given points in a program, rather than a system that guarantees freedom from all types of leakage between variables at all times. Our model may not detect information leaks that occur between variables that have not been annotated, but we believe that in most cases the programmer understands their code best, and therefore that the ability to detect leaks at selected points will provide a more practical, targeted mechanism than one that attempts to detect all possible leaks (and may also return many false positives).

A complication of introducing these annotations is that they can occur anywhere (e.g., in loop bodies) and in any order, so it is important to precisely define the leakage being measured. To provide the theoretical base for a tool that measures such leakage, we present a formal framework in which secret information and potential sites of leakage can be annotated. Probability is an important aspect of many security protocols and systems, so we base our framework on a simple probabilistic, imperative language that we call CH-IMP. The language includes primitives for declaring new variables in a particular scope, assigning new values to variables, and annotating the values stored in selected variables as either secret or observable at that particular point during execution. We note that both the secret and observe commands operate on the variable only when the command is evaluated, and do not indicate that *all* values ever stored in the variable should be


DOI 10.1109/CSF.2013.20




(a) Classic model (b) Our model (c) Our model, with initial secrets (d) Our model, with initial secrets and user input

Fig. 1. An overview of the classic information leakage model and our leakage model (including variations on our leakage model).

treated as secret or observable for the duration of the program. Therefore, the values we wish to keep secret may be stored in variables that are declared at the start of the program, or may be values that result from complex computations performed during the program's execution.

We define the semantics for a CH-IMP program in terms of a discrete-time Markov chain (DTMC) augmented with information about occurrences of secret and observable values. The DTMC can then be used to calculate the joint probability distribution of the secret and observable values, and therefore any of the standard measures of information leakage, such as mutual information, min-entropy leakage, and guessing entropy. To explain the intuition behind our model of leakage, we define the "worst possible semantics" for a program that allows an attacker to observe the value of every variable annotated with the secret command, ignoring all other outputs. Our framework reveals how much an attacker learns about the output of a program executed with this worst possible semantics by observing the output of the program when executed with CH-IMP's standard semantics.

Our framework moves away from the standard model of measuring information leakage, where secret inputs are declared before a program executes and publicly observable outputs are produced upon termination (as depicted in Fig. 1a). Instead, the framework permits secret values and observable outputs to occur at arbitrary points (e.g., inside loops or in blocks of code that are only reached with a small probability). Since our model produces both secret and observable values as a side-effect of program execution (see Fig. 1b), our systems are not formal information-theoretic channels. Although for any terminating program there exists an equivalent program that defines all of its secrets before execution and outputs the observations upon termination, real-world programs are sufficiently complex that rewriting them in this way becomes difficult or impractical. Our model is general enough to also allow secret values to be defined *before* execution (see Fig. 1c), which we will later formalise in Section III-B. In this case, the secret values present upon termination of the program are those defined before *or* during execution, and our measure of information leakage is what can be learnt about these values from the observations.

In Section V, we extend the model further to include a function that supplies inputs to a program from a finite range, based on the previously-observed values (Fig. 1d). The initial secret values cannot be affected by the input function, but the secret values present upon termination of the program can (e.g., if the program uses an attacker-controlled input as a secret value at a later point, our extended model considers the attacker to have learnt the secret value). We show that, when deciding whether there exists *any* input function that causes a program to leak information, it is sufficient to test the program with the input function that produces all possible inputs with equal probability.

Our framework also allows us to define the information leakage of programs that do not terminate and therefore may produce an infinite list of values as their output. We define the leakage of non-terminating programs to be the limit of the leakage as the number of observations tends to infinity. We show this limit exists, and can be calculated by taking the limit of the finite leakage measurements; we also give an exact expression of its value using Lebesgue integration [1]. Our contribution here is to provide a meaningful method of reasoning about leakage from non-terminating programs that may have an infinite number of secret and observable values.

We provide a prototype implementation of our semantics in OCaml that is suitable for checking for information leaks in programs of low to moderate complexity. We present an example that shows how this tool can be used to calculate meaningful measures of information leakage.

To allow our framework to scale, we apply statistical approximation techniques [2] to estimate the measure of information leakage. These techniques provide a confidence interval for an estimation of information leakage from trial runs of a program. We have developed a tool, LeakWatch, which repeatedly executes Java classes annotated with "secret" and "observe" commands and estimates our framework's leakage measure. We show that, for small programs, the OCaml and LeakWatch implementations of the framework produce com-

parable results and that statistical approximation can be used to estimate meaningful measures of information leakage in large programs written in real-world programming languages.

The rest of the paper is organised as follows. In Section II, we define the CH-IMP syntax and motivate our leakage model. In Section III, we define the semantics of our framework using DTMCs and formalise the corresponding notion of information leakage using probability spaces and mutual information. We then extend our model to address non-terminating programs in Section IV by considering programs that produce an unbounded number of both secret and observable values. In Section V, we extend our model further to include external inputs. We discuss implementations of our framework in Sections VI and VII, and conclude in Section VIII.

A version of this paper with full proofs can be found at [3].

### A. Related Work

Information leakage in deterministic programs has been investigated previously (e.g., [4], [5], [6], [7], [8]). While related to our work, much of this previous research does not directly investigate the detection of information leaks in probabilistic programs. A number of probabilistic languages have been presented previously (e.g., [9], [10]), but most were designed for reasoning under uncertainty, which is inappropriate for our work where the entire state space must be known. A related probabilistic language is described in [11]. This work uses operational and denotational semantics to describe the behaviour of the language that, while intuitive, make it more difficult to measure the leakage of non-terminating programs; this complexity is not present in CH-IMP's semantics.

Using information theory to measure leakage in probabilistic systems is also common (e.g., [12], [13]), but most of this work is based on models that measure leakage from secrets declared at the start of a program to outputs observed upon termination; our model of information leakage is more complex. McIver & Morgan [14] propose a method of formalising sequential, non-deterministic programs probabilistically; their work preserves the secrecy of high-security variables throughout execution of the program, rather than just protecting their initial values. We note that our model *intentionally* does not consider the overwriting of secret variables to be the cause of an information leak. More recent work by McIver et al. [15] develops the model further. Askarov & Sabelfield's [16] information leakage model allows for the safe public disclosure of previously-secret values during program execution, but is qualitative rather than quantitative. Alvim et al. [17] propose the use of channels with memory and feedback to model an attacker with the ability to influence the distribution on secret values in terminating interactive programs; the attacker in our model does not have this ability, and studies the information leakage from non-terminating programs. Theoretical models of information leakage in non-terminating programs have been proposed before: O'Neill et al. [18] propose one such model, but it focuses on protecting the strategies of high-level users of the system rather than the values stored in particular high-security variables at specific moments.

Tools that quantitatively measure information leakage in simple languages have been designed previously. QIF [19] is a related tool capable of measuring leakage from a single high-security input to a single low-security output in a language similar to CH-IMP, but a probability distribution on the values of the high-security input must be defined before the program is executed; CH-IMP can measure the leakage from secret values to observable values at arbitrary points during execution. QUAIL [20] measures information leakage from non-terminating programs and uses a probabilistic language syntactically similar to CH-IMP, but relies on a different information leakage model: all variables must be declared before the system executes.

Various practical frameworks have previously been created for measuring information leakage in real-world source code: like CH-IMP, Jif [21] uses source code-level annotations to detect leaks in large Java projects, although it is based on a qualitative information leakage model, and cannot calculate the severity of leaks it detects. Flowcheck [22], a Valgrind-based information leakage tool for C, is able to operate on large software projects, but at the expense of the programmer being able to annotate the source code easily, which is a specific objective of our work.

## II. Information Leakage: Motivation and Model

In this section, we formally define the syntax of the CH-IMP language, and motivate our leakage model with some example programs that subtly leak information.

### A. The CH-IMP Language

The goal of our work is to formally define an information leakage model that can detect and measure information leaks between arbitrary points in programs. Our motivation is to make it as convenient as possible for programmers to utilise this model in their own programs, so we allow variables in CH-IMP to be identified as secret or observable at specific points in a program with minimal annotation. The syntax of the language is defined as follows.

*Definition 1:* A CH-IMP program is a command $C$ conforming to the grammar

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

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

We only consider CH-IMP programs that are well-formed; i.e., ones where variables are declared only once and are not

accessed out of scope. We provide the formal semantics for CH-IMP in Section III-B. The first five production rules are conventional: new declares a new variable $V$ and assigns to it an integer value according to the probability distribution $\rho$ (e.g., $\rho = \{\ 0 \mapsto 0.5,\ 1 \mapsto 0.5\ \}$ could simulate the tossing of a fair coin); assignments, conditional statements, loops and sequential composition are also standard constructs. start and end are not used directly, but are inserted by the semantic rules to create and destroy levels of variable scope (e.g., when evaluated, "while ($B$) { $C$ }" is expanded to "while ($B$) { start; $C$; end }").

### B. Leakage Model: Examples

The measurement of information leakage from secret values declared at the start of a program to public values at its end is an established and well-understood concept (e.g., [7], [12], [6]), as is detecting information leaks to public variables at any location within a program (e.g., [21]). Allowing both secret and observable values to occur at arbitrary points in programs causes subtleties that must be addressed before we can fully define our model of information leakage.

We model an attacker that has access to both the source code of a program and the observable values produced by the program. The attacker cannot identify which observe command caused a particular observation beyond what can be deduced from the source code and the observable values. We note that a more powerful attacker that *does* have this ability can be modelled by adding a unique identifier to each observable value.

Our model measures how much information is learnt (i.e., the reduction in uncertainty) about the program's secret values from its observable values. We do not measure the randomness or predictability of the secrets in isolation; our goal is to answer the question "is it safe to allow an attacker to observe the values of variables annotated with the observe command?". Likewise, marking values dependent on each other as secret does not increase the overall uncertainty about the secret values, and thus does not increase the overall leakage.

Our first example demonstrates that we must measure the information leakage from secret values to *all* observable values, rather than each individual observable value:

```
new rand := { 0 ↦ 0.5, 1 ↦ 0.5 };
observe rand;
new sec := { 0 ↦ 0.5, 1 ↦ 0.5 };
secret sec;
new out := sec xor rand;
observe out;
```

In this program, the attacker observes two values: the value of *rand* on line 2, and the value of *out* on line 6. The secret annotation on line 4 indicates the programmer's concern about possible information leaks regarding the value of *sec* at that point. In this case, the two observed values together leak the secret value, but individually they leak no information to the attacker. This example shows that we must measure the information leakage to the list of all observed values, including those that are observed before other variables are annotated as secret.

Secondly, we consider the possible leakage from sets of secrets:

```
new sec1 := { 0 ↦ 0.5, 1 ↦ 0.5 };
new sec2 := { 0 ↦ 0.5, 1 ↦ 0.5 };
secret sec1;
secret sec2;
new out := sec1 xor sec2;
observe out;
```

While this program does not leak any information about the secret values of *sec1* or *sec2*, an attacker does learn whether the values are equal; therefore, some information is leaked. This example shows that we must consider the leakage from the set of all secret values, rather than measuring the leakage from each secret value individually.

Thirdly, we consider a secret annotation occurring inside a loop:

```
new result := 0; new i := 0; new sec := 0;
while (i < 4) {
    observe result;
    sec := { 1 ↦ 0.0625, ..., 16 ↦ 0.0625 };
    secret sec;
    if (i == 2) {
        result := sec;
    }
    i := i + 1;
}
```

This program marks different variables' values as secret and observable inside a loop, and leaks *sec*'s third value via *result* on the fourth iteration of the loop (thus, 4 bits of information are leaked). Again, this example shows that the set of all observable values must be considered when calculating the possible leakage of any secret value, and not just the observable values that occur in a particular iteration of the loop. It also shows that, when a variable contains different secret values at different times during execution, we must measure the leakage from the set of all secret values that have been stored in that variable.

Our fourth example contains a situation in which no variable's value may be marked as secret:

```
new rand := { 0 ↦ 0.5, 1 ↦ 0.5 };
if (rand == 1) {
    new sec := { 0 ↦ 0.5, 1 ↦ 0.5 };
    secret sec;
    observe sec;
} else {
    observe rand;
}
```

If an attacker observes 1 as the output of this program, they know the secret value of *sec* is 1; however, if the attacker observes the output 0, there is a $\frac{1}{3}$ probability that *sec*'s secret value is 0 and a $\frac{2}{3}$ probability that no variable's value was annotated as secret. If we restrict our measurement of the leakage to only those cases in which the variable's value

is marked as secret, we would conclude that this program always leaks the value, which would be an overestimation. Therefore, in programs where a variable's value may or may not be marked as secret, our model states that the program leaks some information if an attacker can identify whether a variable is annotated as secret, even if they do not learn anything else about the value.

Similarly, we state that the attacker learns information if they discover the order or number of secret declarations, even if they learn nothing about the values themselves. For example, we consider the following program to leak information:

```
new x := { 0 ↦ 0.5, 1 ↦ 0.5 };
new y := { 0 ↦ 0.5, 1 ↦ 0.5 };
observe y;
if (y == 1) {
    new z := { 0 ↦ 0.5, 1 ↦ 0.5 };
    secret x; secret z;
} else {
    secret x;
}
```

In this program, the attacker learns whether only $x$ or both $x$ and $z$ were annotated as secret, but learns nothing about the value of either variable. We consider this to be an information leak, since the fact that a program is processing some confidential data, and what the program considers confidential for a particular execution, may reasonably be sensitive information.

These definitions of what is and is not an information leak are enforced by the semantic rules in Section III-B. We note that it is possible to define a sound leakage model that does not consider the order or number of secret declarations as confidential by making minor changes to our semantics; however, we have chosen the most conservative leakage model on the grounds that a user may consider the order and number of secrets a program possesses to be secret information in its own right.

## III. Information Leakage for the CH-IMP Language

We now formally define the semantics of CH-IMP and the corresponding model of information leakage outlined in Section II. We will define the semantics in terms of discrete-time Markov chains, so we begin with some brief background material on this topic.

### A. Probability Spaces and Discrete-Time Markov Chains

*1) Probability Spaces:* Non-terminating programs may produce an infinite number of observable values; therefore, we cannot represent the probability of observing values as a simple mapping from particular values to a number between 0 and 1. Instead we must represent the probability of observing values as a *probability space*, which is a triple $(\Omega, \mathcal{B}, P)$.

Here, $\Omega$ is the set of all possible *events* (i.e., secret and observable values), which is often infinite. $\mathcal{B}$ is a *$\sigma$-algebra* over the set $\Omega$, which is a set $\mathcal{B} \subseteq 2^\Omega$ of subsets of $\Omega$ that contains $\emptyset$ and is closed under complement and countable unions. For a set $\mathcal{G} \subseteq 2^\Omega$, we say that $\sigma$-algebra $\mathcal{B}$ is *generated* by $\mathcal{G}$ if it is the smallest $\sigma$-algebra containing $\mathcal{G}$. $P : \mathcal{B} \to [0,1]$ is a *probability measure* over $(\Omega, \mathcal{B})$; for each member of $\mathcal{B}$ it gives the probability of an event from that set occurring. We call $(\Omega, \mathcal{B})$ a *measurable space* and sets $b \in \mathcal{B}$ are said to be *measurable*.

Let $(\Omega_X, \mathcal{B}_X)$ be another measurable space. A *random variable* $X$ defined on $(\Omega, \mathcal{B})$ and taking values in $(\Omega_X, \mathcal{B}_X)$ is a function $X : \Omega \to \Omega_X$ such that, for each $b_X \in \mathcal{B}_X$, $X^{-1}(b_X) \in \mathcal{B}$ where $X^{-1}(b_X) = \{\omega \in \Omega \mid X(\omega) \in b_X\}$.

This means that the random variable $X$ has an associated probability distribution $P_X : \mathcal{B}_X \to [0,1]$ giving the probability of each $b_X \in \mathcal{B}_X$:

$$P_X(b_X) = P(X^{-1}(b_X)).$$

Finally, for two (probability) measures $P_1$ and $P_2$, we say that $P_1$ is *absolutely continuous* with respect to $P_2$, if $P_1(b) = 0$ for every set $b$ for which $P_2(b) = 0$.

*2) Discrete-Time Markov Chains:* We use a *discrete-time Markov chain* (DTMC) to represent the possible executions of a CH-IMP program, and then to generate the probability spaces that represent a program's secret and observable values. A DTMC is a tuple $\mathcal{D} = (S, \bar{s}, \mathbf{P})$ where:

- $S$ is a (countable) set of states;
- $\bar{s} \in S$ is an initial state;
- $\mathbf{P} : S \times S \to [0,1]$ is a transition probability matrix such that $\sum_{s' \in S} \mathbf{P}(s, s') = 1$ for all $s \in S$.

The matrix $\mathbf{P}$ gives the probability $\mathbf{P}(s, s')$ of making a transition between any pair of states $s, s' \in S$. A *path* of $\mathcal{D}$ is a (finite or infinite) sequence of states $\omega = s_0 s_1 s_2 \ldots$ such that $s_0 = \bar{s}$ and $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \geq 0$. The set of all infinite paths of $\mathcal{D}$ is denoted $\Omega_\mathcal{D}$ and we can build (following [23]) a probability space over $\Omega_\mathcal{D}$ as follows. For a finite path $\pi = s_0 \ldots s_n$, we assign probability $\mathbf{P}(\pi) \stackrel{\text{def}}{=} \prod_{i=0}^{n-1} \mathbf{P}(s_i, s_{i+1})$. The *basic cylinder* $Cyl(\pi)$ consists of all infinite paths starting with $\pi$. We then define a probability space $(\Omega_\mathcal{D}, \mathcal{B}_\mathcal{D}, P_\mathcal{D})$ over the infinite paths $\Omega_\mathcal{D}$ of $\mathcal{D}$, where:

- $\mathcal{B}_\mathcal{D} \subseteq 2^{\Omega_\mathcal{D}}$ is the $\sigma$-algebra generated by the set of basic cylinders $\{Cyl(\pi) \mid \pi \text{ is a finite path in } \mathcal{D}\}$;
- $P_\mathcal{D}$ is the unique measure with $P_\mathcal{D}(Cyl(\pi)) = \mathbf{P}(\pi)$ for all finite paths $\pi$.

The probability measure $P_\mathcal{D}$ allows us to define the probability of certain (measurable) events of interest in the system being modelled by the DTMC. We will also define random variables on this probability space.

### B. CH-IMP Semantics

We define the semantics of a CH-IMP program $\mathcal{P}$ as a discrete-time Markov chain $\mathcal{D}$. The states of $\mathcal{D}$ describe the current status of the program, annotated with additional information needed to compute information leakage; they are of the form $(C, \sigma, \mathcal{S}, \mathcal{O})$, where:

- $C$ is the list of commands to be executed (i.e., an expression derived from the CH-IMP grammar);

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

$$\textit{(Secrecy)}\ \frac{}{(\mathsf{secret}\ V; C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \sigma, \mathcal{S} :: (V \mapsto [\![V]\!]\sigma), \mathcal{O})}$$

$$\textit{(Observation)}\ \frac{}{(\mathsf{observe}\ V; C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \sigma, \mathcal{S}, \mathcal{O} :: [\![V]\!]\sigma)}$$

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

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

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

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

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

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

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

Fig. 2. The information leakage semantics of CH-IMP. These rules define the transition probability matrix for the discrete-time Markov chain described in Section III-B.

- $\sigma$ is a list of variable scopes (consisting of mappings from variable names to values), with the narrowest scope at the start of the list;
- $\mathcal{S}$ is a list of mappings from variable names to secret values at particular points during execution (as declared with the secret command);
- $\mathcal{O}$ is a list of values that have been marked as observable (with observe commands) up to this point in the program's execution.

The transition probability matrix of the DTMC ($\mathbf{P}$) is defined according to the semantic rules in Fig. 2. $s \xrightarrow{p} s'$ denotes the existence of a transition from state $s$ to state $s'$ with probability $p$, and $\sigma \oplus \{V \mapsto n\}$ denotes the replacement of the mapping for $V$ with $V \mapsto n$ in the narrowest scope in $\sigma$ already containing a mapping for $V$.

The declaration rule adds a new variable mapping to the current scope according to the probability distribution $\rho$. The assignment rule uses the $\oplus$ operator to overwrite the mapping for the variable in the narrowest possible scope. The rules for if and while are standard, with the addition of start and end to signify the creation and destruction of a level of scope respectively: start adds an empty mapping to the start of the list of mappings that will contain all new declarations in the current scope, and end removes the leftmost mapping from the list, unbinding its variables and potentially reducing the number of states (since two states may only differ by the variables bound in their narrowest scopes).

The key new rules of our semantics are those for secret and observe. The "observe $V$" command signifies that the value of the variable $V$ is observable by the attacker, so the semantic rule adds the value of $V$ to the end of the list of observable values $\mathcal{O}$. We note that only the value is added to the list: nothing is added that indicates the variable name or the position in the program of this particular command; therefore, the attacker cannot learn this information (per Section II-B). A stronger attacker model, in which the attacker could learn the variable name or location of the command in the program, could be encoded in our framework by inserting an additional observation that revealed this information.

The "secret V" command signifies that the value of $V$ is to be treated as secret at this point during execution. Its semantic rule therefore records both the name of the variable and its value at this point. As these are stored in an ordered list, our model also considers the order in which secret annotations occur to be confidential. If this were instead a set of mappings from variables to ordered lists, our model would not consider the ordering of secret commands operating on different variables to be confidential, and if it were simply a set of mappings, the ordering of secret commands would not be confidential at all. We have constructed the most conservative model, since it provides the strongest security guarantees.

By inspecting the semantic rules in Fig. 2, we see that at most one rule can be applied to any state $s$ and that the sum of the outgoing transitions for any such rule is always 1. Since

our definition of a DTMC insists on transition probabilities summing to 1 for *all* states, we add a self-loop (i.e., set $\mathbf{P}(s,s) = 1$) for any (terminating) state $s$ that matches no semantic rules. Alternatively, we could adapt our definition of the probability space over DTMC paths to range over both infinite paths and finite paths ending such a terminating state.

The initial state of $\mathcal{D}$ is $(\mathcal{P}, \langle\{\}\rangle, \langle\rangle, \langle\rangle)$ and we will assume that the state space, denoted $S$, is the set of all possible states that can be reached from this state by applying the rules in Fig. 2; this gives a set of (infinite) paths $\Omega_{\mathcal{D}} \subseteq S^{\omega}$ and a probability space $(\Omega_{\mathcal{D}}, \mathcal{B}_{\mathcal{D}}, P_{\mathcal{D}})$ over $\Omega_{\mathcal{D}}$.

We can also adapt our semantics to allow for the possibility of defining *initial* secret values (i.e., before $\mathcal{P}$ is executed) according to some probability distribution $P_{XI}$. We do so by, for each valuation of variables $x_i$ in the support of $P_{XI}$, adding a new state $\bar{s}_i = (\mathcal{P}, \langle\{x_i\}\rangle, \langle x_i\rangle, \langle\rangle)$ to $S$ and adding a transition from the original initial state $\bar{s}$ to $\bar{s}_i$ with probability $\mathbf{P}(\bar{s}, \bar{s}_i) = P_{XI}(x_i)$.

### C. Leakage Measurement

To quantify the information leakage from the program's secret values to its observable values, we use the information-theoretic measure of *mutual information* [24]. This is a common measure of information leakage: the mutual information of the probability distributions on the secret values and on the observable values tells us how much information an attacker learns about the secret values by inspecting the observable values. (For a discussion of the precise security guarantees provided by mutual information and other leakage measures, we refer the reader to the literature; e.g., [25], [26].)

We use $X$ and $Y$ to refer to secret and observable values, respectively. We denote by $D_X$ the set of all possible secret values (including a unique value to indicate that no secret values have been seen) and by $D_Y$ the set of all possible observable values. In accordance with the presentation of the semantics in Section III-B, $D_X$ comprises all possible sequences of mappings from a variable name to an (integer) value and $D_Y$ comprises sequences of (integer) values. In this section, we consider *finite* scenarios: we assume that any execution of the program results in a finite number of secret and observable values, and that these always occur within a finite number of steps of execution. In later sections, we will relax this requirement.

The probability of a program's list of secret value mappings being $x \in D_X$ will be denoted $P_X(x)$ and the probability of it producing the observable values $y \in D_Y$ will be $P_Y(y)$. More precisely (and to aid the more general definitions later in the paper), mutual information is defined in terms of a pair of random variables: one for the secret values and one for the observable values. Thus, we define random variables $X$ and $Y$ on the probability space over paths in the DTMC $\mathcal{D}$. For secret values, the function $X : \Omega_{\mathcal{D}} \to D_X$ projects any path $\omega \in \Omega_{\mathcal{D}}$ onto a list of secret value mappings ($\mathcal{S}$ in the state tuple). For observable values, the function $Y : \Omega_{\mathcal{D}} \to D_Y$ projects any path $\omega \in \Omega_{\mathcal{D}}$ onto the finite sequence of observable (integer) values ($\mathcal{O}$ in the state tuple).

As mentioned in Section III-A, these random variables induce the probability distributions $P_X$ and $P_Y$ over $D_X$ and $D_Y$, respectively. We can also define the joint probability distribution $P_{XY}$ over $D_X \times D_Y$. The mutual information between secret values and observable values is given by the equation:

$$I(X;Y) = \sum_{x \in D_X, y \in D_Y} P_{XY}(x,y) \log_2 \left( \frac{P_{XY}(x,y)}{P_X(x) P_Y(y)} \right).$$

Note that $P_X(x) = \Sigma_y P_{XY}(x,y)$ and $P_Y(y) = \Sigma_x P_{XY}(x,y)$, so the mutual information can be calculated from the joint probability distribution $P_{XY}$ alone. Under the finiteness assumptions that we impose in this section, the joint probability distribution $P_{XY}$ can be determined from an analysis of the *transient probabilities* of $\mathcal{D}$, which give the probability of being in each possible state at a particular time step $n$. In this section, we require that all secret and observe commands always occur within some finite number of steps of execution, say $n_f$. We can therefore compute $P_{XY}(x,y)$ for each $x, y$ by summing the transient probabilities at step $n_f$ for all states $(C, \sigma, \mathcal{S}, \mathcal{O})$ of the DTMC in which $x$ matches $\mathcal{S}$ and $y$ matches $\mathcal{O}$.

A related measure is the *conditional entropy* of $X$ given $Y$, which describes the attacker's uncertainty about the secret values when the observable values are known. It is given by the equation:

$$H(X|Y) = \sum_{x \in D_X, y \in D_Y} P_{XY}(x,y) \log_2 \left( \frac{P_Y(y)}{P_{XY}(x,y)} \right).$$

Another popular measure of information leakage is the *min-entropy leakage* of the secret values after inspecting the observable values [25]; this gives a measure of how difficult it is for the attacker to guess the secret values in one attempt. It is defined as:

$$\log_2 \sum_{y \in D_Y} \max_{x \in D_X} P_{XY}(x,y) - \log_2 \max_{x \in D_X} \sum_{y \in D_Y} P_{XY}(x,y).$$

Like mutual information, conditional entropy and min-entropy leakage can both be calculated from the joint probability distribution, so we can also compute these measures for finite, terminating programs.

### D. Intuition: The Worst Possible Program Semantics

The measure of leakage given above defines what an attacker learns about the stream of secret values by observing the stream of observable values. To give some intuition about what this actually means, we introduce the idea of the "worst possible" semantics for a program that simply leaks all information marked as secret to the attacker: states are of the form $(C, \sigma, \mathcal{S})$, the observe command is replaced with a skip (no-op) command, and the secret command is replaced with one that exposes the variable name and its secret value to the attacker (thus the attacker can observe $\mathcal{S}$). A program evaluated with this "worst possible" semantics leaks all information about every occurrence of the secret command. Let $W$ be the random variable representing the secret values

generated by these semantics. Our measure of leakage can be understood as the amount of information an attacker learns about the output a program when it is executed with this "worst possible" semantics by observing the output of the program when it is executed with the semantics of Fig. 2; i.e., $I(X;Y) = I(W;Y)$. This trivially holds because $X$ and $W$ are defined in the same way.

## IV. Calculating the Leakage of Non-Terminating Programs

### A. Leakage over Infinitely Many Observable Values

Since programs do not necessarily terminate, we now extend our leakage model to programs that may produce an infinitely-long list of observable values. In what follows, we use "non-terminating" specifically to refer to such programs (rather than those that may continue indefinitely, but only generate a finite number of observable values). Our contribution in this section is to formalise a way in which we can reason about information leakage for such non-terminating programs. For now, we continue to assume that the number of secret values is finite; in Section IV-C, we also relax this requirement.

We approximate mutual information for an infinite number of observable values by considering bounded versions of it. We do so by restricting our attention to the first $n$ observable values and the corresponding secret values.

We define $\Omega_{\mathcal{D}}^{(n)}$ to be the set of paths in the DTMC for a program that have been pruned after the first $n$ observable values, i.e., every path in $\Omega_{\mathcal{D}}^{(n)}$ either has exactly $n$ observable values or has fewer than $n$ observable values and terminates, or diverges silently. We note that $\Omega_{\mathcal{D}}^{(n)}$ may include an arbitrary number of paths that may be arbitrarily long. We denote the sets of secret values and observable values in the paths of $\Omega_{\mathcal{D}}^{(n)}$ as $D_X^{(n)}$ and $D_Y^{(n)}$ respectively.

Let $P_X^{(n)}(x)$ and $P_Y^{(n)}(y)$ denote the probabilities of secret values being equal to $x \in D_X^{(n)}$ and observable values being equal to $y \in D_Y^{(n)}$ respectively. As before, we write $P_{XY}^{(n)}$ for the joint probability distribution. We can now define the leakage that occurs after $n$ observations as:

$$I(X^{(n)};Y^{(n)}) = \sum_{x \in D_X^{(n)},\, y \in D_Y^{(n)}} P_{XY}^{(n)}(x,y)\, \log_2 \left( \frac{P_{XY}^{(n)}(x,y)}{P_X^{(n)}(x) P_Y^{(n)}(y)} \right)$$

where $X^{(n)}$ and $Y^{(n)}$ denote the random variables corresponding to $P_X^{(n)}$ and $P_Y^{(n)}$. Notice that, since we only consider $n$ observations, and we restrict the number of secret values to be finite, the sets of values $D_X^{(n)}$ and $D_Y^{(n)}$ are also finite for any given $n$.

Next, we define the leakage from a non-terminating program to be the limit of the above expression as $n$ tends to infinity.

*Definition 2:* The leakage from a non-terminating program with a finite number of secret values and a possibly infinite number of observable values is:

$$\lim_{n\to\infty} I(X^{(n)};Y^{(n)}) = \lim_{n\to\infty} \sum_{x \in D_X^{(n)},\, y \in D_Y^{(n)}} P_{XY}^{(n)}(x,y)\, \log_2 \left( \frac{P_{XY}^{(n)}(x,y)}{P_X^{(n)}(x) P_Y^{(n)}(y)} \right).$$

The intuition here is that the leakage from a non-terminating program is defined as the amount of information an attacker learns about the secret values by inspecting the observable values forever. While it may not be practical for a real attacker to learn this exact amount of information, our definition indicates that a patient attacker will be able to learn an arbitrary approximation of it. We note that this does not allow the attacker to observe the time at which the observations take place; we discuss an extension that permits timed observations in Section IV-D.

Due to our assumption of a finite number of secret values, the limit in Definition 2 always exists, and the leakage is well-defined. Formally, we state this as follows:

*Theorem 1:* The leakage from a non-terminating program with a finite number of secret values and a possibly infinite number of observable values (as defined in Definition 2) exists; i.e., $I(X^{(n)};Y^{(n)})$ converges.

Note that $I(\lim_{n\to\infty} X^{(n)}; \lim_{n\to\infty} Y^{(n)})$ is not defined, so this proof is not trivial. To prove Theorem 1, we use the following lemma:

*Lemma 1:* For all $n$, $I(X^{(n+1)};Y^{(n+1)}) \geq I(X^{(n)};Y^{(n)})$.

*Proof of Theorem 1:* We note that $I(X^{(n)};Y^{(n)}) \leq I(X^{(n+1)};Y^{(n+1)})$ for any $n$ and that $I(X^{(n)};Y^{(n)}) \leq \log_2 |D_X|$, where $|D_X|$ is the (finite) number of secret values. Therefore $I(X^{(n)};Y^{(n)})$ is non-decreasing and bounded from above, and so it converges. ∎

### B. An Exact Expression for Non-Terminating Leakage

Theorem 1 reveals our notion of non-terminating leakage to be well-defined, and taking the limit provides a method of calculating it. However, we believe that it is also interesting to find an exact expression that defines its value.

Firstly, we note that it does *not* suffice to use a definition expressed in terms of (Riemann) integration:

$$\int_{D_X} \int_{D_Y} \log_2 \left( \frac{P_{XY}(x,y)}{P_X(x) P_Y(y)} \right) P_{XY}(x,y)\; dy\; dx$$

since the probability distributions $P_X$ and $P_Y$ are not continuous functions on the real numbers, but defined in terms of infinite sequences of integers; therefore, this equation does not define a value that exists. Instead, we consider the more general notion of mutual information [1] defined using Lebesgue integration as follows. Riemann integration works by partitioning the domain of a function, and here the domain of the probability distributions (the observable values) is not

continuous. Lebesgue integration, however, works by partitioning the range of a function, so using Lebesgue integration we can find an exact expression for leakage from non-terminating programs.

Recall, from Section III-C, that we defined random variables $X$ and $Y$ on the probability space over (infinite) paths of the discrete-time Markov chain semantics of a CH-IMP program. These induced the probability distributions $P_X$ and $P_Y$ over the sets $D_X$ and $D_Y$, giving the probability of each possible list of secret value mappings $x \in D_X$ and list of observable values $y \in D_Y$.

Previously, we imposed restrictions of finiteness on $D_X$ and $D_Y$ (which ensured that $P_X$ and $P_Y$ were discrete probability distributions and allowed us to express mutual information as a finite summation). However, by relaxing this restriction (and allowing $D_Y$ to contain infinite sequences of observable values), we see that $X$ and $Y$ remain well-defined random variables, with corresponding probability measures $P_X$ and $P_Y$. We can also define the joint probability distribution $P_{XY}$ and the product distribution $P_X \times P_Y$ of $P_X$ and $P_Y$.

To find the fraction $\frac{P^{(n)}_{XY}(x,y)}{P^{(n)}_X(x)P^{(n)}_Y(y)}$ as $n$ tends to infinity, we use the Radon Nikodym Theorem (see e.g., [27]), which states that for Lebesgue integration:

*Theorem 2 (Radon Nikodym Theorem):* Given two $\sigma$-finite measures $\nu$ and $\mu$, such that $\nu$ is absolutely continuous with respect to $\mu$, there exists a function $f$ such that for any $A$: $\nu(A) = \int_A f d\mu$. We write $f$ as $\frac{d\nu}{d\mu}$.

Since $P^{(n)}_{XY}(x,y)$ is absolutely continuous with respect to $P^{(n)}_X(x)P^{(n)}_Y(y)$, we know that $\frac{dP_{XY}}{d(P_X \times P_Y)}$ is defined. We can then give an exact expression for the leakage from non-terminating programs:

*Theorem 3:* The leakage $\lim_{n\to\infty} I(X^{(n)}; Y^{(n)})$ from non-terminating programs as defined in Definition 2 equals:

$$I(X;Y) = \int \log_2 \left( \frac{dP_{XY}}{d(P_X \times P_Y)} \right) dP_{XY}.$$

*Proof Sketch:* Gray ([1], p. 80, Lemma 5.2.3) proves that if $P$ is absolutely continuous with respect to $M$ then $D(P||M) = \int \log_2 \frac{dP}{dM}(\omega) dP(\omega)$, where $D$ is the relative entropy. As $P_{XY}$ is absolutely continuous with respect to $P_X \times P_Y$, we obtain $D(P_{XY}||P_X \times P_Y) = I(X;Y)$, thus the result follows.

### C. Leakage over Infinitely Many Secret Values

We now consider non-terminating programs that produce an infinite number of secret values. Clearly, an infinite amount of secret information could be leaked by such a program, so the leakage as defined by Definition 2 may tend to infinity. It is more useful to consider a measure of the *rate* of leakage, or, more precisely, the amount of information leaked per secret value.

*Definition 3:* The rate of leakage from a non-terminating program with a potentially infinite number of secret and observable values is defined as:

$$\lim_{n\to\infty} \sum_{x,y} \frac{1}{\text{secrets}(x)} P^{(n)}_{XY}(x,y) \log_2 \left( \frac{P^{(n)}_{XY}(x,y)}{P^{(n)}_X(x)P^{(n)}_Y(y)} \right)$$

where $n$ is the number of observable values and $\text{secrets}(x)$ is the number of individual secret values defined in $x$ (unless $x$ is the single value that indicates that there were no secret values, in which case $\text{secrets}(x) = 1$).[1]

This gives the average amount of information learned by an attacker per secret command. We note that this limit is not guaranteed to exist. For instance, a program might leak secret values from progressively larger domains and so the value might tend to infinity, or a program could alternately leak secret values and then not leak secret values, causing the leakage rate to oscillate rather than converge. However, we argue that for programs where this leakage rate does not converge, there is no natural definition of information leakage; for programs where this rate does converge, it provides a natural measure of leakage for non-terminating programs.

### D. Detecting Time-Based Leakage

We do not model an attacker that is capable of measuring time, and we therefore do not consider time-based information leaks. We could model an attacker's ability to observe the passing of time by appending a symbol $\tau$ to the list of observations at each time step, e.g.:

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

One side-effect of this addition is that the $n$ that tends to infinity for non-terminating processes would equal both the number of observable values and the number of steps taken by the DTMC, therefore simplifying our framework. However, it would also lead to a very strong attacker model and such a simple discrete model of time may be unrealistic. If a realistic model of a program's execution time and the observable power of the attacker existed, it could be represented in our framework in this way.

## V. External Inputs to a Program

We extend our model to account for external inputs to a program by defining an "input $V : R$" command that takes an (integer) value as input from a finite range $R$ and binds it to the variable $V$. To represent the selection of particular inputs, we introduce a *probabilistic input strategy* function $f$, which takes the observable values encountered at this execution step and returns a probability distribution $\rho$ on the values $R$. For a fixed input strategy $f$, the resulting semantics for a CH-IMP program is obtained by combining the following semantic rule with those in Fig. 2:

[1]When there are no secret values, $P^{(n)}_{XY}(x,y) = P^{(n)}_X(x)P^{(n)}_Y(y)$; therefore, the leakage rate is zero.

$$\frac{f(\mathcal{O}) \to \rho}{(\textsf{input}\ V : R; C, o :: \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{\rho(n)} (C, (\{V \mapsto n\} \cup o) :: \sigma, \mathcal{S}, \mathcal{O} :: n_i)}$$

A program without an input strategy can be imagined as a (partially-observable) Markov decision process. Combining this with a particular input strategy (sometimes referred to as a policy, or adversary in the context of Markov decision processes) induces a discrete-time Markov chain equivalent to the one defined by our semantics above. Our use of input strategies is similar to that of O'Neill et al. [18], who define a language with non-probabilistic *user strategies* and adapt Volpano et al.'s [4] information leakage type system to support these user strategies.

An attacker can observe and possibly control the inputs, so they are considered to be observable values in our model; therefore, the semantic rule above adds the input to the list of observable values, with an $i$ subscript to identify it as an input. We also note that secret values defined after the input occurs may depend on the value of the input. Here, a leak does exist; this case may, for instance, correspond to a program in which the user must choose a secret key, and the attacker, via a particular sequence of inputs, forces the user to choose a particular key, "revealing" its value to the attacker. As discussed in Section I, this cannot be modelled as a formal information-theoretic channel. Alvim et al. [17] present an alternative model of interactive leakage that uses channels with feedback to make it possible to model the system as an information-theoretic channel.

This extension allows us to calculate the leakage from a program with a particular input strategy. Two obvious questions that arise are "does there exist an input strategy that leads to an information leak?" and "what function maximises the leakage?". To answer the first question we show that, when establishing whether a leak exists, it is sufficient to execute a program with the strategy that chooses all inputs uniformly:

*Theorem 4:* For any program, there exists an input strategy that makes the program leak information if, and only if, the program leaks information for an input strategy that chooses inputs with a uniform distribution.

*Proof Sketch:* We note that the choice of an input strategy may affect the probability distribution on the observable values and inputs ($P_Y$) and the probability distribution on the secret values ($P_X$), but it does not affect the conditional probability of the secret values given a particular sequence of inputs and observable values ($P_{X|Y}$). Let $P^f$ denote the probability distributions produced by executing a program with input strategy $f$. If the mutual information is zero for the uniform input strategy $f_U$, then $P_X^{f_U}$ and $P_Y^{f_U}$ are independent and therefore $\forall x, y P_{X|Y}^{f_U}(x|y) = P_X^{f_U}(x)$. Since $P_{X|Y}^{f} = P_{X|Y}^{f'}$ for all strategies $f, f'$, then, if the mutual information is zero for the uniform input strategy, $\forall f, x, y\ P_{X|Y}^{f}(x|y) = P_{X|Y}^{f_U}(x|y) = P_X^{f_U}(x)$ and therefore $P_X^f$ is independent of $P_Y^f$, and so the mutual information is zero.

A question that naturally arises from this is "what input strategy maximises the leakage?". However we show that, for non-terminating programs, such an input strategy may not exist:

*Proposition 1:* There exists a program such that, for any input strategy $f$ (that results in the random variables on secret values $X^f$ and on observable values and inputs $Y^f$), there exists an input strategy $f+$ (with random variables on secret values $X^{f+}$ and on observable values and inputs $Y^{f+}$) such that $I(X^{f+}; Y^{f+}) > I(X^f; Y^f)$.

*Proof:* Consider the following program:

```
new result := 0; new i := 0; new sec := { 1 ↦ 0.5, 2 ↦ 0.5 };
secret sec;
while (i == 0) {
    new z := { 1 ↦ 0.5, 2 ↦ 0.5 };
    if (z == 2) { result := sec; }
    input i : [ 0, 1 ];
}
observe result;
```

Each time the attacker inputs 0, the probability of the secret being copied into the variable *result* is increased, but never becomes 1. When the attacker inputs 1, the program terminates and allows the attacker to observe the outcome of the execution. In this example, any given strategy can be improved by inputting more 0s before inputting 1, so there is no input strategy for this program that leaks the most possible information. ■

A possible extension of this work is to define the maximum leakage as the supremium of the leakage caused by all possible input strategies. We leave the investigation of this as future work. O'Neill et al. [18] allow for user input strategies that must be kept secret from the attacker; another extension of this work would be to adapt their methods to add secret strategies to our framework.

## VI. Implementation

We have implemented the semantics of CH-IMP in an interpreter written in OCaml; our interpreter, along with a CH-IMP implementation of the Dining Cryptographers protocol [28] with an arbitrary number of participants, the motivating examples in Section II-B and other programs, can be found at [29].

An interface to the interpreter reads source code in CH-IMP syntax from a file and parses and executes it, thereby constructing the DTMC described in Section III-A. It then collapses the DTMC to the initial and terminating states, and saves it in a machine-readable form. The interface can then be used to list the variables whose values were marked as secret during the program's execution, and calculate the information leakage from the values of any given subset of them to the observable

```
new m := 8192;
new a := 4801;
new c := 83;
new seed := [ 0 .. 8191 ];

new rand1 := (a * seed + c) mod m;
new card1 := rand1 mod 52;
observe card1;

new rand2 := (a * rand1 + c) mod m;
new card2 := (card1 + 1 + (rand2 mod 51)) mod 52;
secret card2;
```

Fig. 3. A CH-IMP implementation of a random playing card selector using a linear congruential generator. "[ $m$ .. $n$ ]" is syntactic sugar for the uniform distribution ranging over the integers $(m, \ldots, n)$.

values. The tool can calculate the mutual information, min-entropy leakage and conditional entropy of the selected secret values and observable values.

### A. Example: A Pseudorandom Number Generator

Linear congruential generators (LCGs) are widely used for generating pseudorandomness. An LCG $L$ is defined by $L_n \equiv (a \cdot L_{n-1} + c) \bmod m$, where $m$ (the modulus), $a$ (the multiplier) and $c$ (the increment) are parameters chosen by the programmer, and $L_0$ is the LCG's seed value. The choice of $m$, $a$ and $c$ controls the quality of the pseudorandomness provided by the LCG: good values maximise the period of the LCG and thus make the sequence of generated numbers less predictable, while bad values shorten the period and make the sequence more predictable.

Fig. 3 presents a CH-IMP implementation of a random playing card selector. It answers the following question: if playing cards are chosen randomly from a deck according to an LCG with $m = 8192$, $a = 4801$ and $c = 83$ and dealt to each player, how much information does a player learn about the face value of the next card dealt to their opponent based on the face value of the card they were just dealt? In this case, the answer is $\approx 2.5$ bits, which constitutes a significant leak; these values of $m$, $a$ and $c$ therefore produce low-quality pseudorandomness. By making the minor modification $a = 4805$ the leakage is reduced to $\approx 0.03$ bits, underlining the importance of choosing appropriate parameters for an LCG.

### B. Performance

Our OCaml implementation of CH-IMP's semantics performs well on programs of any length provided that the corresponding DTMC remains relatively small: the playing card selector example above, which has a state space of $3+2^{13}\times 7 = 57347$, terminates in under a second on a modest desktop computer. The time required to analyse a CH-IMP program in this way can be exponential in the number of secret values, observable values and variables in scope: cases where the size of the DTMC explodes exponentially take much longer to terminate, as commands must be evaluated in more (and larger) states; nevertheless, our implementation provides a working prototype of the semantics, gives us confidence in its correctness, and allows for the precise computation of leakage from programs with low to medium complexity.

## VII. From Computation to Estimation of Leakage

Given these limitations of the OCaml implementation, along with the difficulty of expressing complex algorithms in CH-IMP syntax, it would be ideal to *a*) accurately measure information leakage through means other than precise computation, and *b*) measure leakage from programs written in a widely-used, real-world programming language. We have therefore developed a second tool, LeakWatch, that estimates the information leakage from secret to observable values in Java programs. Programmers identify the secret and observable values in their programs by calling methods exposed by the LeakWatch API. LeakWatch repeatedly executes a program's main method and collects the lists of secret and observable values defined during each execution, then estimates the joint probability distribution of the secret and observable values (rather than computing it precisely, as in Section III-C). This approximated distribution can then be used to estimate the leakage from the secret values to the observable values, as described in our earlier work on statistical approximation of mutual information [2]. Other tools, e.g., Weka [30], contain loosely similar functionality, but do not calculate confidence intervals for their estimates or test for compatibility with zero leakage. LeakWatch does, and is thus more reliably able to identify statistically significant information leaks in source code.

To demonstrate how LeakWatch can be used to find information leaks in larger, more complex programs, we use it to estimate the information leakage from several real-world Java examples. The examples encompass a number of the models in Fig. 1, rather than only the classic leakage model shown in Fig. 1a. Table I shows how LeakWatch performs on these examples, in terms of the approximate number of executions required for the leakage estimation to "stabilise" (i.e., for successive estimations to differ by fewer than 0.01 bits) to the nearest 1,000 executions, and the total time required to produce this estimation. For instance, LeakWatch estimates the information leakage in `ChimpLCG`, a Java reimplementation of the example in Fig. 3, to be $\approx 2.5$ bits after 50,000 executions in 21 seconds, thus demonstrating the viability of our statistical estimation technique.

The Java source code for all of these examples, along with the tool itself, can be downloaded from [32]. Further examples of the underlying estimation technique being applied to real-world scenarios can be found in [2], [33].

### A. Example: Visual Basic's `Rnd()` Function

The API for the Visual Basic family of programming languages exposes a `Rnd()` function that uses an LCG to return pseudorandom numbers. Visual Basic uses the LCG parameters $m = 2^{24}$, $a = 1140671485$ and $c = 12820163$; these values are too large to be used in the OCaml implementation presented in Section VI, particularly that of $m$: since the value of $L_0$ must fall between 0 and $m$, a precise computation of the leakage from such a program would require the generation and traversal of a DTMC with $3 + 2^{24} \times 7 \approx 117$ million states, which is impractical. In a Java reimplementation of

| Program name | Description | Observable(s) | Secret(s) | Executions | Time (min) |
|---|---|---|---|---|---|
| `ChimpLCG` | Reimplementation of CH-IMP playing card selector (see Section VI-A) | Opponent's card | Our card | 50,000 | 0.3 |
| `VisualBasicLCG` | Playing card selector example with emulation of Visual Basic's `Rnd()` function | Opponent's card | Our card | 60,000 | 0.4 |
| `BloomFilter` | Leakage about Bloom filter set membership based on its internal state | Bloom filter's internal bit array | 1 integer | 4,000 | 0.2 |
| | | | 2 integers | 20,000 | 0.9 |
| | | | 3 integers | 44,000 | 2.0 |
| | | | 4 integers | 88,000 | 4.2 |
| | | | 5 integers | 234,000 | 11.6 |
| | | | 6 integers | 513,000 | 24.4 |
| `SDM` | Information learnt about contents of Secure Data Manager [31] databases | Size of encrypted database | Number of accounts in database | 23,000 | 4.4 |

TABLE I
BENCHMARKING LEAKWATCH'S PERFORMANCE WHEN ESTIMATING LEAKAGE FROM SEVERAL JAVA PROGRAMS.

the example in Fig. 3 with Visual Basic's LCG parameters, LeakWatch reveals a leak of $\approx 0.93$ bits from the face value of the first chosen card to the face value of the second.

### B. Example: Internal State of a Bloom Filter

A Bloom filter is a probabilistic data structure used to test for set membership. The filter's internal state is a bit array of length $l$, with all bits initially set to 0; when elements are "added" to the filter, some of these bits are set to 1 to indicate that element's membership of the set. The precise bits that are set is determined by the output of a hash function $H$ that takes the element's value as input and and outputs the indeces of $k$ bits whose values should be set to 1. Membership of the set can then be tested by checking whether the $k$ bits of the filter corresponding to the test value are set to 1.

Since Bloom filters are probabilistic, it is possible (and indeed common, for small values of $l$) for collisions to occur in the output of $H$. Thus, the filter could mistakenly identify an element as a member of the set if all of its corresponding $k$ bits were set to 1 by other additions to the filter, resulting in a false positive. The filter's internal state therefore leaks a variable amount of information about the values of the elements added to it depending on the size of $l$ and $k$ and the number of elements added.

In this example, $n$ integers between 0 and 7 inclusive are added to an existing Java implementation of a Bloom filter [34] with parameters $l = 16$ and $k = 2$. LeakWatch demonstrates that, as $n$ increases, the average amount of information leaked about each integer added to the filter decreases as more collisions occur and the probability of encountering false positives increases (see Table II).

| Integers added | Leakage (bits) | Leakage per integer (bits) |
|---|---|---|
| 1 | 2.80 | 2.8 |
| 2 | 5.04 | 2.5 |
| 3 | 6.20 | 2.1 |
| 4 | 6.64 | 1.6 |
| 5 | 6.84 | 1.4 |
| 6 | 6.86 | 1.1 |

TABLE II
THE AMOUNT OF INFORMATION LEAKED ABOUT THE INTEGERS ADDED TO A BLOOM FILTER FROM THE FILTER'S INTERNAL STATE.

### C. Example: A Password Manager

Password management software is commonly used to securely store databases of sensitive account details, such as login names and passwords for web sites. To protect the database, the password manager encrypts it with a user-specified passphrase before writing it to a file on disk; to retrieve the database at a later time, the user must load the encrypted file and re-enter the passphrase for decryption. Care must be taken when designing password managers to ensure that information about the database is not inadvertently leaked when it is encrypted. Secure Data Manager [31], a free password manager written in Java, contains such a flaw: databases are serialised and DES-encrypted under a user-specified key but are not padded before being written to disk, so the size of the encrypted file leaks information about the number of accounts stored in the unencrypted database. LeakWatch indicates that an attacker can guess the approximate number of accounts in the database by observing the size of the encrypted file; this can be verified by randomly inserting between 1 and 32 accounts into an empty database, then annotating the number of accounts as a secret value and the size of the encrypted database (truncated to the nearest 100 bytes) as an observable value. The size of the leakage is estimated to be 4.72 out of 5 bits. This is a security risk: attackers will therefore target larger encrypted files known to be written by Secure Data Manager since they are more likely to contain larger numbers of account details. This example demonstrates how software developers can use LeakWatch to detect new information leaks in their own software as part of their development flow.

## VIII. CONCLUSION

We have presented a framework that can be used to measure information leaks between arbitrary points in a program. To do so, we have introduced CH-IMP, a language that allows variables' values to be annotated as either secret or observable

by an attacker, and that provides a mechanism for quantifying information leaks from secret to observable values. The language is underlied by a coherent model of information leakage for terminating programs, which we extended to non-terminating programs by taking the limit of the leakage for mutual information, and extended further to programs that produce potentially infinite numbers of secret and observable values. These methods could equally be applied to min-entropy leakage, and we plan to do so in further work. We also modelled attacker input strategies in our framework and showed that, when we wish to discover if a leak exists for any possible input strategy, it is sufficient to test a program with a uniform input strategy that ignores the observable values.

We have built a prototype implementation of our semantics in OCaml that can be used to measure information leakage from programs of low to medium complexity; we extended this to real-world programs by developing a tool, LeakWatch, that uses statistical approximation to estimate leakage from Java programs. In both cases, the secret and observable values occur at arbitrary points throughout the program; therefore, our methods of quantifying leakage were only achievable using the leakage model presented in this paper.

## ACKNOWLEDGEMENT

This work was supported by EPSRC Research Grant EP/J009075/1. We also wish to thank Olga Maleva for her helpful advice.

## REFERENCES

[1] R. Gray, *Entropy and Information Theory*, 1st ed. Springer, 1991.
[2] K. Chatzikokolakis, T. Chothia, and A. Guha, "Statistical Measurement of Information Leakage," in *The 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010)*, ser. LNCS, 2010, pp. 390–404.
[3] http://www.cs.bham.ac.uk/research/projects/infotools/.
[4] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2–3, pp. 167–187, Jan. 1996.
[5] D. Clark, S. Hunt, and P. Malacaria, "Quantified interference for a while language," *Electron. Notes Theor. Comput. Sci.*, vol. 112, pp. 149–166, 2005.
[6] ——, "A static analysis for quantifying information flow in a simple imperative language," *Journal of Computer Security*, vol. 15, no. 3, pp. 321–371, 2007.
[7] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic Discovery and Quantification of Information Leaks," in *Proceedings of the 30th IEEE Symposium on Security and Privacy*. Oakland, California, USA: IEEE Computer Society, May 2009, pp. 141–153.
[8] A. Almeida Matos and G. Boudol, "On declassification and the non-disclosure policy," *Journal of Computer Security*, vol. 17, no. 5, pp. 549–597, 2009.
[9] A. Dekhtyar and V. S. Subrahmanian, "Hybrid Probabilistic Programs," *Journal of Logic Programming*, vol. 3, no. 2, pp. 187–250, Jun. 2000.
[10] A. Pfeffer, "IBAL: A Probabilistic Rational Programming Language," in *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*. Seattle, Washington, USA: Morgan Kaufmann, Aug. 2001, pp. 733–740.
[11] D. Kozen, "Semantics of probabilistic programs," *Journal of Computer and System Sciences*, vol. 22, no. 3, pp. 328–350, 1981.
[12] K. Chatzikokolakis, C. Palamidessi, and P. Panangaden, "Anonymity Protocols as Noisy Channels," *Information and Computation*, vol. 206, no. 2–4, pp. 378–401, 2008.
[13] F. Biondi, A. Legay, P. Malacaria, and A. Wasowski, "Quantifying information leakage of randomized protocols," in *Proc. VMCAI'13*, 2013, pp. 68–87.
[14] A. McIver and C. Morgan, "Programming methodology," A. McIver and C. Morgan, Eds. New York, NY, USA: Springer-Verlag New York, Inc., 2003, ch. A probabilistic approach to information hiding, pp. 441–460. [Online]. Available: http://dl.acm.org/citation.cfm?id=766951.766972
[15] A. McIver, L. Meinicke, and C. Morgan, "Security, probability and nearly fair coins in the cryptographers' cafe;," in *Proceedings of the 2nd World Congress on Formal Methods*, ser. FM '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 41–71.
[16] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 2007, pp. 207–221.
[17] M. S. Alvim, M. E. Andrés, and C. Palamidessi, "Quantitative information flow in interactive systems," *Journal of Computer Security*, vol. 20, no. 1, pp. 3–50, 2012.
[18] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *Computer Security Foundations Workshop, 2006. 19th IEEE*, 2006.
[19] C. Mu and D. Clark, "A tool: quantitative analyser for programs," in *Proceedings of the 8th International Conference on Quantitative Evaluation of Systems (QEST 2011)*, 2011, pp. 145–146.
[20] F. Biondi, A. Legay, L. Traonouez, and A. Wasowski, "QUAIL: A Quantitative Security Analyzer for Imperative Code," in *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013)*, 2013.
[21] A. C. Myers and B. Liskov, "Complete, Safe Information Flow with Decentralized Labels," in *Proceedings of the 1998 IEEE Symposium on Security and Privacy*. Oakland, California, USA: IEEE Computer Society, May 1998, pp. 186–197.
[22] S. McCamant and M. D. Ernst, "Quantitative Information Flow as Network Flow Capacity," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008)*. Tucson, Arizona, USA: ACM Press, Jun. 2008, pp. 193–205.
[23] J. Kemeny, J. Snell, and A. Knapp, *Denumerable Markov Chains*, 2nd ed. Springer-Verlag, 1976.
[24] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed. Wiley-Interscience, 2006.
[25] G. Smith, "On the foundations of quantitative information flow," in *FOSSACS '09: Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 288–302.
[26] ——, "Quantifying Information Flow Using Min-Entropy," in *Proceedings of the 8th International Conference on Quantitative Evaluation of Systems (QEST 2011)*, 2011, pp. 159–167.
[27] A. C. Zaanen, *Introduction to Operator Theory in Riesz spaces*. Springer, 1996.
[28] D. Chaum, "The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability," in *Journal of Cryptology*, 1988, pp. 65–75.
[29] http://www.cs.bham.ac.uk/research/projects/infotools/chimp/.
[30] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software," *SIGKDD Explorations*, vol. 11, no. 1, pp. 10–18, 2009.
[31] "Secure Data Manager," http://sdm.sourceforge.net.
[32] http://www.cs.bham.ac.uk/research/projects/infotools/leakwatch/.
[33] T. Chothia, Y. Kawamoto, and C. Novakovic, "A Tool for Estimating Information Leakage," in *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013)*, 2013.
[34] "Bloom filters in Java," http://www.javamex.com/tutorials/collections/bloom_filter.shtml.