Secure Information Flow with Random Assignment and Encryption

Geoffrey Smith Rafael Alpízar
School of Computing and Information Sciences
Florida International University
Miami, Florida 33199 USA
smithg@cis.fiu.edu, ralpi001@cis.fiu.edu

ABSTRACT

Type systems for secure information flow aim to prevent a program from leaking information from variables classified as H to variables classified as L. In this work we extend such a type system to address encryption and decryption; our intuition is that encrypting a H plaintext yields a L ciphertext. We argue that well-typed, polynomial-time programs in our system satisfy a computational probabilistic noninterference property, provided that the encryption scheme is IND-CCA secure. As a part of our proof, we first consider secure information flow in a language with a random assignment operator (but no encryption). We establish a result that may be of independent interest, namely, that well-typed, probabilistically total programs with random assignments satisfy probabilistic noninterference. We establish this result using a weak probabilistic bisimulation.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of Programs*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types*

General Terms

Security, Languages, Verification

Keywords

secure information flow, type systems, encryption, noninterference, reductions, IND-CPA, IND-CCA

1. INTRODUCTION

Secure information flow analysis aims to prevent untrusted programs from "leaking" the sensitive information that they manipulate. More precisely, if we classify variables as H (high) or L (low), then we wish to prevent information from H variables from flowing into L variables. (More generally, we may want a richer *lattice* of security levels.) In the Dennings' seminal paper [8], an expression is classified as H if it contains any H variables; otherwise, it is classified as L. The paper then establishes two basic restrictions on programs: first, to prevent *explicit flows*, a H expression may not be assigned to a L variable; second, to prevent *implicit flows*, an **if** or **while** command whose guard is H may not make *any* assignments to L variables. Much later, Volpano, Smith, and Irvine [24] showed that the Denning restrictions could be formulated as a type system and that they suffice to ensure *noninterference*, which says (roughly) that the final values of L variables are independent of the initial values of H variables. Since this early work, the area of secure information flow has been heavily studied, addressing issues such as nontermination, concurrency, and exceptions; see [19] for a comprehensive survey up to 2003.

Our goal here is to extend a type system for secure information flow with typing rules for symmetric encryption and decryption primitives. Intuitively, we expect that encrypting a H plaintext yields a L ciphertext, while decrypting a L (or H) ciphertext yields a H plaintext. So, if $\mathcal{E}$ and $\mathcal{D}$ denote encryption and decryption under a suitably chosen shared key, we would like to include typing rules that say that

- if e is H, then $\mathcal{E}(e)$ is L, and
- if e is either L or H, then $\mathcal{D}(e)$ is H.

The desirability of treating encryption within the type system is illustrated by recent work by Askarov and Sabelfeld [2] on implementing a "mental poker" protocol in Jif [18]. In their implementation, they are forced to use explicit declassification "casts" to tell the Jif type system to regard the encryption of a H expression as L. Such casts must all be analyzed by hand to decide whether they are safe. If, instead, the type system included rules as above, then such casts and hand analysis would be unnecessary.

A crucial question, of course, is whether our intuitive typing rules for encryption and decryption are *sound*. In fact, they are not sound unless the encryption scheme satisfies strong security properties. For example, if encryption is *deterministic* (for instance, a block cipher like AES in Electronic Code Book mode), then a well-typed program can efficiently leak a secret. For example, suppose that *secret* is a H n-bit variable and that *leak* and *mask* are L variables. Consider the following program, in which "|" denotes bitwise-or, and "$\gg 1$" denotes right shift by one bit:

FMSE'06, November 3, 2006, Alexandria, Virginia, USA.

```
leak := 0;
mask := 2^{n-1};
while mask ≠ 0 do (
    if E(secret | mask) = E(secret) then
        leak := leak | mask;
    mask := mask ≫ 1
)
```

Note that if $\mathcal{E}$ is a deterministic encryption function, then the test in the **if** command is true iff $secret \mid mask = secret$. And this is true iff the bit of *secret* specified by *mask* is 1. Hence this code copies *secret* into *leak*. Furthermore, under the Denning restrictions together with the above rule for typing $\mathcal{E}$, this program is well typed—the guards of the **while** and **if** are both L, which means that the nested assignments to the L variables *leak* and *mask* are allowed.

In the cryptographic community, it is well understood that deterministic encryption cannot give good security properties [6]. In fact one needs a *probabilistic* or *stateful* encryption algorithm, so that encrypting the same plaintext repeatedly yields (probably) different ciphertexts each time. Notice that if $\mathcal{E}$ were probabilistic in this way, the leaking program above would not work—then typically *all* of the tests in the **if** command would come out false, which means that *leak* would just end up with value 0.

We will use two strong properties that are used in the cryptographic community to define the security of symmetric encryption schemes, namely *IND-CPA* and *IND-CCA* security [6]. (These abbreviations stand for "indistinguishability under chosen-plaintext attack" and "indistinguishability under chosen-ciphertext attack".) However, even assuming strong security properties for the encryption scheme, it is still challenging to justify our new typing rules. The difficulty is that we cannot show *noninterference*, because the typing rule for encryption allows *some* information to flow from H to L. For example, suppose that h is H and l is L and consider the well-typed program

$$l \stackrel{?}{\leftarrow} \mathcal{E}(h)$$

which encrypts h and puts the result into l. (We write $\stackrel{?}{\leftarrow}$ here to reflect the fact that $\mathcal{E}$ is probabilistic.) *Probabilistic noninterference* requires that the probability distribution on the final values of L variables be *independent* of the initial values of H variables. That plainly cannot hold here, because any two distinct plaintexts must give rise to *disjoint* sets of possible ciphertexts—otherwise decryption would not be possible.

Instead we seek a *computational probabilistic noninterference* result that says that, if program c is well typed, then changes to the initial values of H variables lead to probability distributions on the final values of L variables that are *indistinguishable* to observers with limited computational resources. However, this property cannot hold for *all* well-typed programs, because there is a well-typed program that uses brute-force search to discover the implicit key K and then uses it to leak a secret h:

1. Pick a few L plaintexts and encrypt each with $\mathcal{E}$, producing a few L ciphertexts.

2. Go through all possible keys, searching for one that successfully decrypts each of the ciphertexts. Decryption is done not by calling the decryption primitive $\mathcal{D}$, but by directly implementing the underlying decryption algorithm. Eventually the key will be found, and it can be stored in a L variable.

3. Encrypt the H variable h using $\mathcal{E}$, producing a L ciphertext.

4. Decrypt the ciphertext using the key found in Step 2, and write the result into the L variable l.

```
t ←? {0,1};
if t = 0 then (
    while h = 1 do skip;
    l := 0
)
else (
    while h = 0 do skip;
    l := 1
)
```

Figure 1: A random assignment program

Of course, the running time of this program is exponential in the size of K. Our security property will apply only to programs that run in polynomial time in the size of K.

In carrying out our proofs, it turns out that we need to do secure information flow analysis on a simple imperative language extended with a *random assignment* command $x \stackrel{?}{\leftarrow} \mathcal{R}$, which assigns a randomly-chosen value to x according to probability distribution $\mathcal{R}$. Random assignment makes the language probabilistic, and hence similar to multi-threaded languages that have been previously investigated. But because the language is sequential, we do not have *races* between different threads, which greatly complicate secure information flow analysis and require more restrictive typing rules. For example, [21] achieves probabilistic noninterference on multi-threaded programs by requiring that a command whose running time depends on H variables cannot be followed sequentially by an assignment to a L variable.

Instead we propose to type random assignment programs using just the Denning restrictions, and we would like to show probabilistic noninterference. However, the possibility of nontermination raises complications. Consider for example the program in Figure 1, where h is H and l and t are L. (In the program, the random assignment $t \stackrel{?}{\leftarrow} \{0,1\}$ assigns either 0 or 1 to t, each with probability 1/2.) Under the typing rules that we will present in Section 3, this program is well typed. But, if $h = 0$, then this program terminates with $l = 0$ with probability 1/2 and fails to terminate with probability 1/2. And if $h = 1$, then it terminates with $l = 1$ with probability 1/2 and fails to terminate with probability 1/2. Thus this program does not satisfy probabilistic noninterference.

This example shows that we need to consider carefully the possibility of nontermination in our secure information-flow analysis. But, for our intended application to cryptography, this is not too big a problem because, as we argued above, we already need to assume that the program's running time is polynomial in the size of the key.

The rest of this paper is organized as follows. In Section 2, we discuss related work. Then, in Section 3, we define the

syntax, semantics, and type system of the random assignment language, proving that well-typed probabilistically-total programs satisfy probabilistic noninterference; the proof uses a weak probabilistic bisimulation as in [22]. We also develop techniques for bounding the effects of possible nonterminations. In Section 4, we extend the random assignment language with an encryption primitive and prove that well-typed programs running in polynomial time satisfy a computational probabilistic noninterference property, provided that the encryption scheme is IND-CPA secure. In Section 5, we further extend the language with a decryption primitive and again prove security, this time provided that the encryption scheme is IND-CCA secure. In Section 6, we briefly discuss how our type system might be applied in practice. Finally, Section 7 concludes.

2. RELATED WORK

Peeter Laud has pioneered the area of computationally-secure information-flow analysis in the presence of encryption. In his first works [13, 14], the analysis was not in the form of a type system. But his recent paper with Varmo Vene [16] is quite similar in theme to our paper. However, there are some significant differences. First, their type system does not address decryption, while ours does. Second, their language is richer in that it supports the generation and manipulation of keys. In contrast, we assume that the encryption and decryption operations use a single, implicitly generated key; in compensation, we are able to use a much simpler type system and soundness proof. A final, minor, difference is that they do not separately consider the analysis of programs that use random assignment but not encryption. (Such programs can, of course, be analyzed under their system, but their results imply only that such programs, if well-typed, satisfy a *computational secure information flow* property, but this is weaker than the *probabilistic noninterference* property that we show.)

More distantly related is the large body of recent work aimed at proving computational security properties of cryptographic protocols. An example is Warinschi's paper [25] that proves the computational soundness of the Needham-Schroeder-Lowe public key protocol, provided that the encryption scheme is IND-CCA secure. However, it should be noted that the problem addressed by such works is very different from the secure information-flow problem: protocol work considers distributed systems in the presence of an *active adversary*, whose behavior is unconstrained but that does not have direct access to certain secrets; in contrast, secure information-flow analysis considers untrusted programs that *do* have direct access to secret information and that must be prevented (using typing rules, for example) from *propagating* it improperly.

Like our work here, much of the work on cryptographic protocols aims to relate formal analysis techniques with computational security properties. For example, the pioneering work of Abadi and Rogaway [1] proves a computational justification for Dolev-Yao-style analysis of protocol messages under passive adversaries. More recently, Backes and Pfitzmann [4] establish secrecy properties for a rich Dolev-Yao-style cryptographic library.

Building on [4], Laud [15] presents a type system to ensure a computational secrecy property. This paper appears quite similar to ours (and to [16]), except that it considers a process calculus (similar to spi) rather than a general-purpose programming language. The process calculus is rich in that it includes constructs for message passing, symmetric and asymmetric encryption and decryption, key and nonce generation, and so forth. However, the type system of [15] has a major limitation: its rules for typing conditionals, (IfH) and (IfR), do not allow branching on secret data. In contrast, our rule (IF), which is standard in secure-information flow analysis, allows such branching (while of course preventing implicit flows). In the context of implementing protocols, branching on secret data may well be unnecessary, but it certainly cannot reasonably be disallowed in general-purpose programming.

In secure information-flow analysis, recently there has been a great deal of interest in approaches that relax noninterference to allow *some* leaks of information, while still preserving security (in some sense). For example, Volpano [23] shows a computational justification for typing rules for one-way hash functions; because hashing is deterministic, however, his typing rules need to be more restrictive than the rules we use here. Another notable work is Li and Zdancewic [17], which uses downgrading policies as security levels, so that the security level specifies what must be done to "sanitize" a piece of information. Yet another approach is to measure the leakage of information quantitatively; examples include Di Pierro, Hankin, and Wiklicky [9] and Clark, Hunt, and Malacaria [7]. For a general survey of recent work on incorporating declassification into secure information-flow analyses, see Sabelfeld and Sands [20].

$$
\begin{array}{llll}
(\textit{phrases}) & p & ::= & e \mid c \\
(\textit{expressions}) & e & ::= & x \mid n \mid e_1 + e_2 \mid \ldots \\
(\textit{commands}) & c & ::= & x := e \mid \\
 & & & x \stackrel{?}{\leftarrow} \mathcal{R} \mid \\
 & & & \textbf{skip} \mid \\
 & & & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \\
 & & & \textbf{while } e \textbf{ do } c \mid \\
 & & & c_1; c_2
\end{array}
$$

Figure 2: Language Syntax

3. RANDOM ASSIGNMENT

The language that we consider is the simple imperative language extended with a random assignment command. The language syntax is defined in Figure 2. In the syntax, metavariable x ranges over identifiers and n over integer literals. Integers are the only values; we use 0 for false and nonzero for true. The command $x \stackrel{?}{\leftarrow} \mathcal{R}$ is a random assignment; here $\mathcal{R}$ ranges over some set of probability distributions on the integers.

A program c is executed under a *memory* μ, which maps identifiers to integers. We assume that expressions are total and evaluated atomically, with $\mu(e)$ denoting the value of expression e in memory μ. A *configuration* is either a pair (c, μ) or simply a memory μ. In the first case, c is the command yet to be executed; in the second case, the command has terminated, yielding final memory μ.

Because of the random assignment command, the usual transition relation on configurations needs to be extended with probabilities—we write $C \xrightarrow{p} C'$ to indicate that the

$$\text{(UPDATE)}\quad \frac{x \in dom(\mu)}{(x := e, \mu) \xrightarrow{1} \mu[x := \mu(e)]}$$

$$\text{(CHOOSE)}\quad \frac{x \in dom(\mu)\ \ \mathcal{R}(v) > 0}{(x \xleftarrow{?} \mathcal{R}, \mu) \xrightarrow{\mathcal{R}(v)} \mu[x := v]}$$

$$\text{(NO-OP)}\quad (\mathbf{skip}, \mu) \xrightarrow{1} \mu$$

$$\text{(BRANCH)}\quad \frac{\mu(e) \neq 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu) \xrightarrow{1} (c_1, \mu)} \qquad \frac{\mu(e) = 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu) \xrightarrow{1} (c_2, \mu)}$$

$$\text{(LOOP)}\quad \frac{\mu(e) = 0}{(\mathbf{while}\ e\ \mathbf{do}\ c, \mu) \xrightarrow{1} \mu} \qquad \frac{\mu(e) \neq 0}{(\mathbf{while}\ e\ \mathbf{do}\ c, \mu) \xrightarrow{1} (c; \mathbf{while}\ e\ \mathbf{do}\ c, \mu)}$$

$$\text{(SEQUENCE)}\quad \frac{(c_1, \mu) \xrightarrow{p} \mu'}{(c_1; c_2, \mu) \xrightarrow{p} (c_2, \mu')} \qquad \frac{(c_1, \mu) \xrightarrow{p} (c_1', \mu')}{(c_1; c_2, \mu) \xrightarrow{p} (c_1'; c_2, \mu')}$$

$$\text{(STUTTER)}\quad \mu \xrightarrow{1} \mu$$

Figure 3: Structural Operational Semantics

probability of going from configuration C to configuration C' is p. The semantic rules are given in Figure 3. Rule STUTTER is added so that the set of configurations forms a discrete Markov chain [10].

Now we consider a type system for secure information flow. Our intuition is that random assignment does not lead to leaks of information, so we would like to use just the usual Denning restrictions. Here are the types we will use:

$$\begin{array}{lll} (\textit{data types}) & \tau & ::= L \mid H \\ (\textit{phrase types}) & \rho & ::= \tau \mid \tau\ \textit{var} \mid \tau\ \textit{cmd} \end{array}$$

Intuitively, τ *var* is the type of variables that store information of level τ, while τ *cmd* is the type of commands that assign only to variables of level τ or higher; this implies that command types obey a *contravariant* subtyping rule. Typing judgments have the form $\Gamma \vdash p : \rho$, where Γ is an *identifier typing* that maps each variable to a type of the form τ *var*. The typing and subtyping rules are given in Figures 4 and 5. The rules are the same as those in [24], except for the new rule RANDOM for random assignment.

We now explore the properties of the type system. First, we have the usual Simple Security, Confinement, and Subject Reduction properties:

LEMMA 3.1 (SIMPLE SECURITY). *If $\Gamma \vdash e : \tau$, then e contains only variables of level τ or lower.*

Proof. By induction on the structure of e. □

LEMMA 3.2 (CONFINEMENT). *If $\Gamma \vdash c : \tau\ cmd$, then c assigns only to variables of level τ or higher.*

$$\text{(R-VAL)}\quad \frac{\Gamma(x) = \tau\ var}{\Gamma \vdash x : \tau}$$

$$\text{(INT)}\quad \Gamma \vdash n : L$$

$$\text{(PLUS)}\quad \frac{\Gamma \vdash e_1 : \tau,\ \ \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 + e_2 : \tau}$$

$$\text{(ASSIGN)}\quad \frac{\Gamma(x) = \tau\ var,\ \ \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \tau\ cmd}$$

$$\text{(RANDOM)}\quad \frac{\Gamma(x) = \tau\ var}{\Gamma \vdash x \xleftarrow{?} \mathcal{R} : \tau\ cmd}$$

$$\text{(SKIP)}\quad \Gamma \vdash \mathbf{skip} : H\ cmd$$

$$\text{(IF)}\quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c_1 : \tau\ cmd \quad \Gamma \vdash c_2 : \tau\ cmd}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \tau\ cmd}$$

$$\text{(WHILE)}\quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c : \tau\ cmd}{\Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ c : \tau\ cmd}$$

$$\text{(COMPOSE)}\quad \frac{\Gamma \vdash c_1 : \tau\ cmd \quad \Gamma \vdash c_2 : \tau\ cmd}{\Gamma \vdash c_1; c_2\ :\ \tau\ cmd}$$

Figure 4: Typing rules

$$\text{(BASE)}\quad L \subseteq H$$

$$\text{(CMD)}\quad \frac{\tau' \subseteq \tau}{\tau\ cmd \subseteq \tau'\ cmd}$$

$$\text{(REFLEX)}\quad \rho \subseteq \rho$$

$$\text{(TRANS)}\quad \frac{\rho_1 \subseteq \rho_2,\ \ \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$$

$$\text{(SUBSUMP)}\quad \frac{\Gamma \vdash p : \rho_1,\ \ \rho_1 \subseteq \rho_2}{\Gamma \vdash p : \rho_2}$$

Figure 5: Subtyping rules

Proof. By induction on the structure of c. □

LEMMA 3.3 (SUBJECT REDUCTION). *If $\Gamma \vdash c : \tau$ cmd and $(c, \mu) \xrightarrow{p} (c', \mu')$ for some $p > 0$, then $\Gamma \vdash c' : \tau$ cmd.*

Proof. By induction on the structure of c. □

Now we want to argue that well-typed programs satisfy *probabilistic noninterference*, which asserts that the joint probability distribution of the final values of L variables is independent of the initial values of H variables. But, as noted in Section 1, the possibility of nontermination causes difficulties. Thus we argue this property only for programs that are *probabilistically total*, which means that they halt with probability 1 on all inputs.

DEFINITION 3.1. *Memories μ and ν are* L-equivalent*, written $\mu \sim_L \nu$, if μ and ν have the same domain and agree on the values of all L variables.*

Next we define L-equivalence on commands, using an approach similar to that used in [21]. First, adopt the convention that sequential composition associates to the *left*, so that, for $n \geq 1$, $c_1; c_2; c_3; \ldots; c_n$ represents

$$(\ldots((c_1; c_2); c_3); \ldots); c_n.$$

Notice that when a command of the form $c_1; c_2; c_3; \ldots; c_n$ is executed, the first execution step touches only c_1.

DEFINITION 3.2. *We say that well-typed commands c and d are L-equivalent, written $c \sim_L d$, if either*

1. *$c = d$, or*
2. *for some $n \geq 1$, c has form $c_1; c_2; c_3; \ldots; c_n$, d has form $d_1; c_2; c_3; \ldots; c_n$, and c_1 and d_1 both have type H cmd.*

Next we extend the notion of L-equivalence to configurations:

DEFINITION 3.3. *Configurations C and D are L-equivalent, written $C \sim_L D$, if any of the following four cases applies:*

1. *C has form (c, μ), D has form (d, ν), $c \sim_L d$, and $\mu \sim_L \nu$.*
2. *C has form (c, μ), D has form ν, c has type H cmd, and $\mu \sim_L \nu$.*
3. *C has form μ, D has form (d, ν), d has type H cmd, and $\mu \sim_L \nu$.*
4. *C has form μ, D has form ν, and $\mu \sim_L \nu$.*

It is easy to see that $\sim_L$ is an equivalence relation on the set of well-typed configurations.

Next, we recall the notion of *weak probabilistic bisimulation* as defined in [22]. Suppose that we have an equivalence relation $\approx$ on the states of a discrete Markov chain. Given two distinct equivalence classes Φ and Ψ and state $C \in \Phi$, we let $\mathcal{P}(C, \Phi, \Psi)$ denote the probability of starting at C, moving for 0 or more steps within Φ, and then entering Ψ.

DEFINITION 3.4. *Equivalence relation $\approx$ is a* weak probabilistic bisimulation *if, for all distinct equivalence classes Φ and Ψ, $\mathcal{P}(C, \Phi, \Psi)$ is independent of the choice of C.*

Now we come to our key theorem:

THEOREM 3.4. *On the set of well-typed, probabilistically total configurations, $\sim_L$ is a weak probabilistic bisimulation.*

Proof. Suppose that $C \sim_L D$, where C and D are well-typed, probabilistically total configurations. If we denote the equivalence class of C and D by Φ, then we must show that for any other equivalence class Ψ,

$$\mathcal{P}(C, \Phi, \Psi) = \mathcal{P}(D, \Phi, \Psi).$$

First, if either C or D is a terminated configuration, then the other is either terminated or contains a command of type H cmd. Hence (by Confinement and Subject Reduction) neither C nor D can ever leave their equivalence class. So $\mathcal{P}(C, \Phi, \Psi) = \mathcal{P}(D, \Phi, \Psi) = 0$.

If, instead, $C = (c, \mu)$ and $D = (d, \nu)$, then $c \sim_L d$ and $\mu \sim_L \nu$. We consider the two possibilities of Definition 3.2 in turn.

If $c \sim_L d$ by virtue of possibility 2 of Definition 3.2, then for some $n \geq 1$, c is of the form $c_1; c_2; c_3; \ldots; c_n$, d is of the form $d_1; c_2; c_3; \ldots; c_n$, and c_1 and d_1 both have type H cmd. Now, executing $(c_1; c_2; c_3; \ldots; c_n, \mu)$ for a step can lead to configurations of the form $(c_1'; c_2; c_3; \ldots; c_n, \mu')$. But since $c_1 : H$ cmd, we have (by Subject Reduction and Confinement) that $c_1' : H$ cmd and $\mu \sim_L \mu'$. Hence such a transition does not leave the original equivalence class. So executing (c, μ) must stay in the original equivalence class at least until reaching a configuration of the form $(c_2; c_3; \ldots; c_n, \mu')$ where $\mu \sim_L \mu'$. Furthermore, because (c, μ) is probabilistically total, such a configuration is reached with probability 1. The same reasoning applies to (d, ν). The conclusion is that both (c, μ) and (d, ν) stay in the same equivalence class until reaching a configuration of the form $(c_2; c_3; \ldots; c_n, \xi)$ where $\mu \sim_L \nu \sim_L \xi$, and both reach such a configuration with probability 1.

If $c \sim_L d$ is not by virtue of possibility 2 of Definition 3.2, then we must have that $c = d$. Write c and d in the *standard form* $c_1; c_2; \ldots; c_k$ for some $k \geq 1$, where c_1 is *not* a sequential composition. Because possibility 2 of Definition 3.2 does not hold, we also know that c_1 does not have type H cmd. Consider the possible forms of c_1:

1. Case **skip**. Impossible, since **skip** : H cmd.
2. Case $x := e$. By rule UPDATE (and rule SEQUENCE),

   $$(c_1; c_2; \ldots; c_k, \mu) \xrightarrow{1} (c_2; \ldots; c_k, \mu[x := \mu(e)])$$

   and

   $$(c_1; c_2; \ldots; c_k, \nu) \xrightarrow{1} (c_2; \ldots; c_k, \nu[x := \nu(e)]).$$

   Since c_1 does not have type H cmd, we must have $x : L$ var and $e : L$, by rule ASSIGN. Hence, by Simple Security, $\mu(e) = \nu(e)$, which implies that $\mu[x := \mu(e)] \sim_L \nu[x := \nu(e)]$.
3. Case $x \xleftarrow{?} \mathcal{R}$. By rule CHOOSE, for each v with $\mathcal{R}(v) > 0$,

   $$(c_1; c_2; \ldots; c_k, \mu) \xrightarrow{\mathcal{R}(v)} (c_2; \ldots; c_k, \mu[x := v])$$

   and

   $$(c_1; c_2; \ldots; c_k, \nu) \xrightarrow{\mathcal{R}(v)} (c_2; \ldots; c_k, \nu[x := v])$$

Hence both configurations go to the same equivalence classes with the same probabilities.

4. Case **if** e **then** c_{11} **else** c_{12}. Since c_1 does not have type H *cmd*, we must have $e : L$ by rule IF. Hence, $\mu(e) = \nu(e)$ by Simple Security, which means that the executions from μ and from ν choose the same branch.

5. Case **while** e **do** c_{11}. Similar to the **if** case.

□

COROLLARY 3.5. *Any well-typed, probabilistically total program c satisfies probabilistic noninterference.*

Proof. If c is a well-typed probabilistically total program and $\mu \sim_L \nu$, then $(c, \nu) \sim_L (c, \mu)$. Since $\sim_L$ is a weak probabilistic bisimulation, the two executions pass through the same equivalence classes with the same probabilities (though not necessarily at the same rate). Hence the probability distribution on the final values of L variables will be the same in both executions. □

What about programs that are not probabilistically total, like the program in Figure 1? As explained before, that program does not satisfy probabilistic noninterference. What goes wrong? Intuitively, that program is "trying" to set l to either 0 or 1, each with probability 1/2, regardless of the value of h. But there are infinite loops, which depend on the value of h, that sometimes prevent it from reaching an assignment to l. As a result, the probabilities of certain values of l are lowered, because the paths that would have led to them become infinite loops.

To make these ideas clearer, let us define for any well-typed program c a "stripped" version of c, denoted by $\lfloor c \rfloor$, that replaces all subcommands of type H *cmd* within c with **skip**. More precisely, define $\lfloor c \rfloor =$ **skip** if c has type H *cmd*; otherwise, define $\lfloor c \rfloor$ by

- $\lfloor x := e \rfloor = x := e$
- $\lfloor x \stackrel{?}{\leftarrow} \mathcal{R} \rfloor = x \stackrel{?}{\leftarrow} \mathcal{R}$
- $\lfloor$**if** e **then** c_1 **else** $c_2 \rfloor =$ **if** e **then** $\lfloor c_1 \rfloor$ **else** $\lfloor c_2 \rfloor$
- $\lfloor$**while** e **do** $c_1 \rfloor =$ **while** e **do** $\lfloor c_1 \rfloor$
- $\lfloor c_1; c_2 \rfloor = \lfloor c_1 \rfloor; \lfloor c_2 \rfloor$

For example, here is the stripped version of the program in Figure 1:

```
t ←? {0,1};
if t = 0 then (
    skip;
    l := 0
)
else (
    skip;
    l := 1
)
```

It is easy to see that $\lfloor c \rfloor$ contains no H variables. For example, if c is $x := e$ and c has type H *cmd*, then $\lfloor c \rfloor =$ **skip**. And if c does not have type H *cmd* then $\lfloor c \rfloor = x := e$. But in this case by rule ASSIGN we must have that x is a L variable and $e : L$, which implies (by Simple Security) that e contains only L variables.

The key effect of the stripping operation is to boost the probabilities of terminating with certain values of L variables by lowering the probability of nontermination. Consider the example above. When $h = 0$, the stripped program boosts the probability of terminating with $l = 1$ from 0 up to 1/2 by lowering the probability of nontermination from 1/2 down to 0; also, both the original and stripped program terminate with $l = 0$ with probability 1/2.

More precisely, the "boosting" property of $\lfloor c \rfloor$ says that, as far as the final values of L variables are concerned, $\lfloor c \rfloor$ can simulate whatever c can do, at least as quickly and with at least the same probability:

THEOREM 3.6. *If c is well typed and can terminate within at most n steps with its L variables having certain values, then $\lfloor c \rfloor$ can do the same, with probability at least as great.*

Proof. (Sketch.) The main idea is that if c is of the form $c_1; c_2$, where c_1 has type H *cmd*, then c_1 cannot assign to L variables, so replacing it by **skip** in $\lfloor c \rfloor$ has no effect on L variables, except possibly to increase the probabilities of the values that would result from executing c_2; this will happen if c_1 might go into an infinite loop.

This argument can be made formally using the approach of *probabilistic strong simulation* due to Jonsson and Larsen [11]. □

We will use these results about $\lfloor c \rfloor$ in the next section.

4. ENCRYPTION

In this section, we extend our language from Section 3 with a symmetric encryption primitive $\mathcal{E}$ and we extend the type system with a rule that says that the encryption of a H expression is L. We first recall the definitions of *symmetric encryption scheme* and *IND-CPA security* from [5, 6]:

DEFINITION 4.1. *A* symmetric encryption scheme $\mathcal{SE}$ *with security parameter k is a triple of algorithms $(\mathcal{K}, \mathcal{E}, \mathcal{D})$, where*

- *$\mathcal{K}$ is a randomized key-generation algorithm that generates a k-bit key; we write $K \stackrel{?}{\leftarrow} \mathcal{K}$*
- *$\mathcal{E}$ is a randomized encryption algorithm that takes a key and a plaintext and returns a ciphertext; we write $C \stackrel{?}{\leftarrow} \mathcal{E}_K(M)$.*
- *$\mathcal{D}$ is a deterministic decryption algorithm that takes a key and a ciphertext and returns a plaintext; we write $M := \mathcal{D}_K(C)$.*

We recall the notion of *IND-CPA security*, which stands for *indistinguishability under chosen-plaintext attack*. An adversary $\mathcal{A}$ is given an *LR oracle* of the form

$$\mathcal{E}_K(\mathrm{LR}(\cdot, \cdot, b)),$$

where K is a randomly generated key and b is an internal *selection bit*, which is either 0 or 1. When $\mathcal{A}$ sends a pair of equal-length messages (M_0, M_1) to the LR oracle, it selects either M_0 or M_1 according to the value of b, encrypts it using $\mathcal{E}_K$, and returns the ciphertext C to $\mathcal{A}$. Thus when $\mathcal{A}$ sends a sequence of pairs of messages to the LR oracle, it either gets back encryptions of the *left* messages (if $b = 0$) or else

encryptions of the *right* messages (if $b = 1$). $\mathcal{A}$'s challenge is to guess which of these two "worlds" it is in; it returns a bit with its guess of the value of b. (Intuitively, $\mathcal{A}$ will be unsuccessful if $\mathcal{SE}$ is strong, as $\mathcal{A}$ will be unable to glean any useful information from the ciphertexts.) Formally, $\mathcal{A}$ is executed in two *experiments*:

Experiment $\mathbf{Exp}^{\text{ind-cpa-1}}_{\mathcal{SE}}(\mathcal{A})$
  $K \xleftarrow{?} \mathcal{K}$;
  $d \xleftarrow{?} \mathcal{A}^{\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,1))}$;
  return d

and

Experiment $\mathbf{Exp}^{\text{ind-cpa-0}}_{\mathcal{SE}}(\mathcal{A})$
  $K \xleftarrow{?} \mathcal{K}$;
  $d \xleftarrow{?} \mathcal{A}^{\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,0))}$;
  return d

The *IND-CPA advantage* of $\mathcal{A}$ is defined as

$$\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(\mathcal{A}) = \Pr[\mathbf{Exp}^{\text{ind-cpa-1}}_{\mathcal{SE}}(\mathcal{A}) = 1] - \Pr[\mathbf{Exp}^{\text{ind-cpa-0}}_{\mathcal{SE}}(\mathcal{A}) = 1].$$

We say that $\mathcal{SE}$ is *IND-CPA secure* if no adversary $\mathcal{A}$ running in polynomial time in the security parameter k can achieve a non-negligible advantage. (As usual, $s(k)$ is *negligible* if for any positive polynomial $p(k)$, there exists k_0 such that $s(k) \leq \frac{1}{p(k)}$, for all $k \geq k_0$.)

Having recalled these preliminaries, we now extend our language from Section 3 to include symmetric encryption. We do not, however, include key generation and manipulation in our language. Instead, we assume that a single key K is generated before executing the program and is used implicitly in all encryption and decryption operations. Also, we do not yet include decryption, instead deferring it to Section 5.

One technical issue must be addressed before proceeding. An IND-CPA secure symmetric encryption scheme is typically *message-length revealing*, because the size of the ciphertext depends on the size of the plaintext. For example, under cipher-block chaining with random initial vector, an m-block plaintext gives an $(m + 1)$-block ciphertext. This property could be exploited to leak a secret efficiently:

if *secret* % 2 = 0 **then**
  $h :=<$ *some 1-block plaintext* $>$
else
  $h :=<$ *some 2-block plaintext* $>$;
$l \xleftarrow{?} \mathcal{E}(h)$;
if l *is 2 blocks long* **then**
  leak := 0
else
  leak := 1

This code leaks the last bit of *secret*, yet it is well typed (using our intended typing rule for encryption) if *secret* and h are H and l and *leak* are L.

Laud and Vene [16] address this difficulty by assuming that encryption is *length concealing*, as defined by Abadi and Rogaway [1]. Here we adopt an essentially equivalent restriction: we assume that all integer values in our language are n bits long, for some n (perhaps 128). This way, we never encrypt plaintexts of different sizes.

If n is the block size of a block cipher, and we use CBC\$ mode (cipher block chaining with random initial vector) or CTR\$ mode (counter mode with random initial vector) [6], then our ciphertexts will be two n-bit blocks long. We therefore adopt the following syntax for the encryption operation in our language:

$$(x, y) \xleftarrow{?} \mathcal{E}(e)$$

We add this as a new command, joining those in Figure 2. This command encrypts the n-bit value of e with the implicit key K, producing a $2n$-bit ciphertext; it then puts the first n bits into x and the second n bits into y.

Here is the typing rule for encryption, which we add to the rules in Figure 4:

$$\text{(ENCRYPT)}\quad \frac{\begin{array}{l}\Gamma(x) = \tau_1\ \mathit{var}\\ \Gamma(y) = \tau_2\ \mathit{var}\\ \Gamma \vdash e : H\\ \tau \subseteq \tau_1\\ \tau \subseteq \tau_2\end{array}}{\Gamma \vdash (x, y) \xleftarrow{?} \mathcal{E}(e) : \tau\ \mathit{cmd}}$$

This rule allows the encryption of a H expression e to be assigned to variables x and y, regardless of whether they are H or L. (To control implicit flows, however, the type of the command must record the minimum level of variable that is assigned to; this is the purpose of the last two hypotheses in the rule.)

We now wish to argue that extending the type system with rule ENCRYPT is *sound*. We begin by showing that no well-typed program can (with probability significantly greater than 1/2) efficiently leak a randomly-chosen, 1-bit H variable h into a L variable l, provided that encryption is IND-CPA secure.

More precisely, a *leaking adversary* $\mathcal{B}$ is a program containing a H variable h and a L variable l and other variables that can be typed arbitrarily. It is executed in the following experiment:

Experiment $\mathbf{Exp}^{\text{leak}}_{\mathcal{SE}}(\mathcal{B})$
  $K \xleftarrow{?} \mathcal{K}$;
  $h_0 \xleftarrow{?} \{0, 1\}$;
  $h := h_0$;
  initialize all other variables of $\mathcal{B}$ to 0;
  run $\mathcal{B}^{\mathcal{E}_K(\cdot)}$;
  if $l = h_0$ **then return** 1 **else return** 0

Here $h_0 \xleftarrow{?} \{0, 1\}$ assigns a random 1-bit integer to h_0. Variable h_0 must not occur in $\mathcal{B}$; it is used to record the initial value of h. Finally, the *leaking advantage* of $\mathcal{B}$ is defined as

$$\mathbf{Adv}^{\text{leak}}_{\mathcal{SE}}(\mathcal{B}) = 2 \cdot \Pr[\mathbf{Exp}^{\text{leak}}_{\mathcal{SE}}(\mathcal{B}) = 1] - 1.$$

(The leaking advantage is defined in this way to reflect the fact that $\mathcal{B}$ can trivially succeed with probability 1/2.)

THEOREM 4.1. *Given a well-typed leaking adversary $\mathcal{B}$ (in the random assignment language with encryption) that runs in polynomial time $p(k)$, there exists an IND-CPA adversary $\mathcal{A}$ such that*

$$\mathbf{Adv}^{\textit{ind-cpa}}_{\mathcal{SE}}(\mathcal{A}) \geq \frac{1}{2} \cdot \mathbf{Adv}^{\textit{leak}}_{\mathcal{SE}}(\mathcal{B}).$$

Moreover, $\mathcal{A}$ runs in $O(p(k))$ time.

Proof. Given leaking adversary $\mathcal{B}$, we construct an IND-CPA adversary $\mathcal{A}$ that runs $\mathcal{B}$ with a randomly-chosen 1-bit value of h. Whenever $\mathcal{B}$ makes a call $\mathcal{E}(e)$ to its encryption primitive, $\mathcal{A}$ passes $(0^n, e)$ to its LR oracle and returns the result to $\mathcal{B}$. When $\mathcal{B}$ terminates, $\mathcal{A}$ checks whether $\mathcal{B}$ has succeeded in leaking h to l; if so, $\mathcal{A}$ guesses that it is in world 1; if not, $\mathcal{A}$ guesses that it is in world 0. Also (for reasons that will be explained shortly) $\mathcal{A}$ limits $\mathcal{B}$'s execution to $p(k)$ steps, returning 0 if $\mathcal{B}$ does not terminate within that time. Formally:

Adversary $\mathcal{A}^{\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,b))}$
  $h_0 \xleftarrow{?} \{0,1\}$;
  $h := h_0$;
  initialize all other variables of $\mathcal{B}$ to 0;
  run $\mathcal{B}^{\mathcal{E}_K(\mathrm{LR}(0^n,\cdot,\cdot,b))}$ for at most $p(k)$ steps;
  if $\mathcal{B}$ did not terminate **then return** 0;
  if $l = h_0$ **then return** 1 **else return** 0

Note first that $\mathcal{A}$ runs in $O(p(k))$ time.

We now analyze $\mathcal{A}$'s IND-CPA advantage. First, note that if $\mathcal{A}$ is in world 1, then $\mathcal{B}$ is run faithfully—each encryption $\mathcal{E}(e)$ is converted to $\mathcal{E}_K(\mathrm{LR}(0^n, e, 1))$, which returns $\mathcal{E}_K(e)$. Also $p(k)$ steps are enough for $\mathcal{B}$ to terminate, by assumption. Hence $\mathcal{A}$ returns 1 precisely if $\mathcal{B}$ succeeds in its leaking experiment. Hence

$$\begin{aligned}&\Pr[\mathbf{Exp}^{\text{ind-cpa-1}}_{\mathcal{SE}}(\mathcal{A}) = 1]\\ &= \Pr[\mathbf{Exp}^{\text{leak}}_{\mathcal{SE}}(\mathcal{B}) = 1]\\ &= \tfrac{1}{2} \cdot \mathbf{Adv}^{\text{leak}}_{\mathcal{SE}}(\mathcal{B}) + \tfrac{1}{2}.\end{aligned}$$

Next, suppose that $\mathcal{A}$ is in world 0. In this case, $\mathcal{B}$ is *not* run faithfully—each encryption $\mathcal{E}(e)$ is converted to $\mathcal{E}_K(\mathrm{LR}(0^n, e, 0))$, which returns $\mathcal{E}_K(0^n)$. This has nothing to do with e—it just returns a $2n$-bit value from some fixed probability distribution (depending only on K). Hence in this case $\mathcal{B}$ is actually run as $\mathcal{B}^{\mathcal{E}_K(0^n)}$, a well-typed program in the random assignment language studied in Section 3.

A subtle point, however, is that $\mathcal{B}^{\mathcal{E}_K(0^n)}$ might run for *longer* than $\mathcal{B}^{\mathcal{E}_K(\cdot)}$; in fact it might not even be probabilistically total. (For instance, $\mathcal{B}$ might call $\mathcal{E}(0^n)$ and $\mathcal{E}(1^n)$ and then test whether the two ciphertexts are equal. In $\mathcal{B}^{\mathcal{E}_K(\cdot)}$, the test would certainly be false, but in $\mathcal{B}^{\mathcal{E}_K(0^n)}$, it would be true with a small nonzero probability.) This explains why $\mathcal{A}$ needs to run $\mathcal{B}$ for at most $p(k)$ steps.

More seriously, we cannot simply apply Corollary 3.5 to conclude that $\mathcal{B}^{\mathcal{E}_K(0^n)}$ satisfies probabilistic noninterference. Instead we use Theorem 3.6 to bound the probability that $\mathcal{A}$ returns 1 in world 0. Consider $\lfloor \mathcal{B}^{\mathcal{E}_K(0^n)} \rfloor$, the stripped unfaithful version of $\mathcal{B}$. Because it does not contain any H variables, we know that it trivially satisfies probabilistic noninterference. Hence there must exist probabilities p_0, p_1, p_2, and p_3 satisfying $p_0 + p_1 + p_2 + p_3 = 1$ such that, if $\lfloor \mathcal{B}^{\mathcal{E}_K(0^n)} \rfloor$ is run for $p(k)$ steps then, regardless of the initial value of h, $\lfloor \mathcal{B}^{\mathcal{E}_K(0^n)} \rfloor$

- terminates with $l = 0$ with probability p_0,
- terminates with $l = 1$ with probability p_1,
- terminates with l having some other value with probability p_2, and
- fails to terminate with probability p_3.

Now, the unstripped program $\mathcal{B}^{\mathcal{E}_K(0^n)}$ need not satisfy probabilistic noninterference, but by Theorem 3.6 the terminating probabilities can only go down. That is, there must exist nonnegative q_0, q_1, q_2, r_0, r_1, and r_2 such that, if $\mathcal{B}^{\mathcal{E}_K(0^n)}$ is run for $p(k)$ steps then, when $h = 0$, $\mathcal{B}^{\mathcal{E}_K(0^n)}$

- terminates with $l = 0$ with probability $p_0 - q_0$,
- terminates with $l = 1$ with probability $p_1 - q_1$,
- terminates with l having some other value with probability $p_2 - q_2$, and
- fails to terminate with probability $p_3 + q_0 + q_1 + q_2$

and, when $h = 1$, $\mathcal{B}^{\mathcal{E}_K(0^n)}$

- terminates with $l = 0$ with probability $p_0 - r_0$,
- terminates with $l = 1$ with probability $p_1 - r_1$,
- terminates with l having some other value with probability $p_2 - r_2$, and
- fails to terminate with probability $p_3 + r_0 + r_1 + r_2$.

We can now bound the probability that $\mathcal{A}$ returns 1 in world 0. When $h = 0$, $\mathcal{A}$ returns 1 with probability $p_0 - q_0$, and when $h = 1$, $\mathcal{A}$ returns 1 with probability $p_1 - r_1$. Hence in total $\mathcal{A}$ returns 1 in world 0 with probability

$$\frac{1}{2} \cdot (p_0 - q_0) + \frac{1}{2} \cdot (p_1 - r_1),$$

which is at most $1/2$, since $p_0 + p_1 \leq 1$ and q_0 and r_1 are nonnegative. Hence we have shown that

$$\Pr[\mathbf{Exp}^{\text{ind-cpa-0}}_{\mathcal{SE}}(\mathcal{A}) = 1] \leq \frac{1}{2}.$$

Now the theorem follows immediately from the definition of $\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(\mathcal{A})$. □

Corollary 4.2. *If $\mathcal{SE}$ is IND-CPA secure, then there is no polynomial-time, well-typed leaking adversary $\mathcal{B}$ that achieves a non-negligible advantage.*

Proof. Given such a $\mathcal{B}$ with non-negligible leaking advantage $s(k)$, then by Theorem 4.1 there exists polynomial-time $\mathcal{A}$ with IND-CPA advantage $s(k)/2$, contradicting the IND-CPA security of $\mathcal{SE}$. □

We can further apply Corollary 4.2 to argue a *computational probabilistic noninterference* property, similar to properties considered by Laud [14, 16] and Backes and Pfitzmann [3]. The idea is that if a well-typed, polynomial-time program c is run twice, under two L-equivalent memories μ and ν, then an adversary with access to the final values of the L variables of c cannot efficiently guess whether the initial memory was μ or ν with probability non-negligibly greater than $1/2$.

To this end, let us say that a *noninterference adversary* $\mathcal{N}$ for c, μ, and ν is a program that does not refer to the H variables of c and outputs its guess into a new variable g. It is executed in the following experiment, where h_0 is a new variable:

Experiment $\mathbf{Exp}^{\text{NI}}_{\mathcal{SE},c,\mu,\nu}(\mathcal{N})$
  $K \xleftarrow{?} \mathcal{K}$;
  $h_0 \xleftarrow{?} \{0,1\}$;
  if $h_0 = 0$ **then** initialize the variables of c to μ
    else initialize the variables of c to ν;
  c;
  $\mathcal{N}$;
  if $g = h_0$ **then return** 1 **else return** 0

Now the *noninterference advantage* of $\mathcal{N}$ is defined as

$$\mathbf{Adv}^{\text{NI}}_{\mathcal{SE},c,\mu,\nu}(\mathcal{N}) = 2 \cdot \Pr[\mathbf{Exp}^{\text{NI}}_{\mathcal{SE},c,\mu,\nu}(\mathcal{N}) = 1] - 1.$$

THEOREM 4.3. *If c is a well-typed, polynomial-time program and μ and ν are two L-equivalent memories, then no polynomial-time noninterference adversary $\mathcal{N}$ for c, μ, and ν can achieve a non-negligible noninterference advantage.*

Proof. Assuming that such a noninterference adversary $\mathcal{N}$ exists, we can build a leaking adversary $\mathcal{B}$ which uses two new variables h and l, of type H and L, respectively. $\mathcal{B}$ is defined as follows:

Adversary $\mathcal{B}$
  initialize the L variables of c according to μ and ν;
  if $h = 0$ **then**
    initialize the H variables of c according to μ
  else
    initialize the H variables of c according to ν;
  c;
  $\mathcal{N}$;
  $l := g$

Now, observe that $\mathcal{B}$ runs in polynomial time. Also, we can argue that $\mathcal{B}$ is well typed. First, the initialization code is well typed under rules ASSIGN and IF. (Notice that this would not be true if μ and ν were not L-equivalent, because then the initialization of the L variables of c would depend on h.) Next, c is well typed by hypothesis. Finally, we can argue that $\mathcal{N}$ is well typed. This does *not* follow by hypothesis—$\mathcal{N}$ should be thought of as a *passive adversary*, which cannot be assumed to be well typed. But because $\mathcal{N}$ cannot refer to the H variables of c, we can give all of its variables type L, which makes it *automatically* well typed under our rules. Finally, the leaking advantage of $\mathcal{B}$ is non-negligible, because it is the same as the noninterference advantage of $\mathcal{N}$. This contradicts Corollary 4.2. □

5. DECRYPTION

In this section, we further extend our language from Section 3 with a decryption operation $\mathcal{D}$, which decrypts under the implicit key K. As explained in Section 4, our encryption primitive $\mathcal{E}$ is assumed to take an n-bit plaintext to a $2n$-bit ciphertext. Hence our decryption primitive $\mathcal{D}$ takes two n-bit expressions as input (to represent the $2n$ bits of the ciphertext) and returns an n-bit plaintext. Because decryption is deterministic, we can model it simply as a new expression: $\mathcal{D}(e_1, e_2)$. We add this as a new expression, joining those in Figure 2.

Here is the typing rule for decryption, which we add to the rules in Figure 4:

$$\text{(DECRYPT)}\quad \frac{\Gamma \vdash e_1 : H \qquad \Gamma \vdash e_2 : H}{\Gamma \vdash \mathcal{D}(e_1, e_2) : H}$$

Notice that, because of subtyping, e_1 and e_2 can be either L or H. But, whatever they are, $\mathcal{D}(e_1, e_2)$ is always H.

With our extended language, we use exactly the same definition of leaking adversary $\mathcal{B}$ as before, except that the adversary can now perform decryption. The definitions of the experiment $\mathbf{Exp}^{\text{leak}}_{\mathcal{SE}}(\mathcal{B})$ and leaking advantage $\mathbf{Adv}^{\text{leak}}_{\mathcal{SE}}(\mathcal{B})$ are unchanged.

Intuitively, it does not seem that decryption should help a leaking adversary, since a decryption always has type H. However, we have so far been unable to adapt the proof of Theorem 4.1 to deal with decryption. But we are able to prove an analogous theorem in the case where the symmetric encryption scheme $\mathcal{SE}$ satisfies a stronger security property, namely *IND-CCA security*, which stands for *indistinguishability under chosen-ciphertext attack* [6].

Like an IND-CPA adversary, an IND-CCA adversary $\mathcal{A}$ has an LR-oracle $\mathcal{E}_K(\text{LR}(\cdot,\cdot,b))$ whose selection bit b it is trying to guess. But, unlike an IND-CPA adversary, an IND-CCA adversary also has a decryption oracle $\mathcal{D}_K(\cdot)$. To prevent $\mathcal{A}$ from winning trivially, it is forbidden from querying the decryption oracle on any ciphertext that it previously received from the LR-oracle. Aside from these differences, the definitions of $\mathbf{Exp}^{\text{ind-cca-1}}_{\mathcal{SE}}(\mathcal{A})$ and $\mathbf{Exp}^{\text{ind-cca-0}}_{\mathcal{SE}}(\mathcal{A})$, of the *IND-CCA advantage* $\mathbf{Adv}^{\text{ind-cca}}_{\mathcal{SE}}(\mathcal{A})$, and of *IND-CCA security* are just like the definitions in the IND-CPA case. (See [6] for details.)

We now show that our type system with both encryption and decryption is sound, assuming that $\mathcal{SE}$ is IND-CCA secure.

THEOREM 5.1. *Given a well-typed leaking adversary $\mathcal{B}$ (in the random assignment language with encryption and decryption) that runs in polynomial time $p(k)$, there exists an IND-CCA adversary $\mathcal{A}$ such that*

$$\mathbf{Adv}^{\textit{ind-cca}}_{\mathcal{SE}}(\mathcal{A}) \geq \frac{1}{2} \cdot \mathbf{Adv}^{\textit{leak}}_{\mathcal{SE}}(\mathcal{B}).$$

Moreover, $\mathcal{A}$ runs about as quickly as $\mathcal{B}$.

Proof. The proof is quite similar to that of Theorem 4.1. Given leaking adversary $\mathcal{B}$, we construct an IND-CCA adversary $\mathcal{A}$ that runs $\mathcal{B}$ with a randomly-chosen 1-bit value of h, for at most $p(k)$ steps, As before, whenever $\mathcal{B}$ makes a call $\mathcal{E}(m)$ to its encryption primitive, $\mathcal{A}$ passes $(0^n, m)$ to its LR oracle and returns the result, which is a two-block ciphertext (c_1, c_2), to $\mathcal{B}$. But $\mathcal{A}$ also remembers $((c_1, c_2), m)$ in a hash table. Whenever $\mathcal{B}$ makes a call $\mathcal{D}(c_1, c_2)$ to its decryption primitive, $\mathcal{A}$ first checks whether (c_1, c_2) is in the hash table. If so, it returns the corresponding plaintext m to $\mathcal{B}$. If not, it answers the query using its decryption oracle $\mathcal{D}_K(\cdot)$. As before, $\mathcal{A}$ guesses that it is in world 1 if $\mathcal{B}$ terminates within $p(k)$ steps and successfully leaks h; if not, then $\mathcal{A}$ guesses that it is in world 0.

Note that although there will be some cost associated with maintaining the hash table, $\mathcal{A}$ still runs about as quickly as $\mathcal{B}$. Also note that $\mathcal{A}$ never "cheats" by calling its decryption oracle on a ciphertext previously returned by its LR-oracle.

As before, when $\mathcal{A}$ is in world 1, it runs $\mathcal{B}$ faithfully. And when $\mathcal{A}$ is in world 0, it runs $\mathcal{B}$ as a program in the random assignment language of Section 3. (Note that the decryption primitive $\mathcal{D}$ can be viewed as an ordinary operation like $+$, since its typing rule conforms to the Simple Security lemma.) The calculation of $\mathbf{Adv}^{\text{ind-cca}}_{\mathcal{SE}}(\mathcal{A})$ follows as before. □

Using Theorem 5.1, we can obtain results analogous to Corollary 4.2 and Theorem 4.3, using exactly the same arguments.

6. DISCUSSION

This paper formally establishes that well-typed programs under our type system cannot leak H information. How might the type system, scaled to a more realistic language, be applied in practice? One situation that seems promising is in systems involving several components communicating over insecure channels that are subject to eavesdropping. We could model such channels as variables of type L, indicating that the information on them must be public. Then, if two components wish to share H information across a channel, the type system will require that they first encrypt the information before sending it. (In contrast, L information can be sent without encryption.) The type system would thus provide protection against sloppy programming of the components that inadvertently exposes H information to eavesdropping. More interestingly, the type soundness result ensures that there is no way for a malicious insider to deliberately leak H information through tricky programming. (One important caveat should be noted—an eavesdropper could presumably sense the *time* when information is put onto the channel, introducing the possibility of *timing channels*. Such channels are notoriously difficult to prevent, but their effectiveness can be limited by introducing random delays, as in the NRL Pump [12].)

7. CONCLUSION

In this paper, we first have shown that the standard Denning restrictions are sufficient to ensure probabilistic noninterference for probabilistically total programs with random assignment. We have also developed techniques for bounding the possible effects of nontermination. Then we have applied these techniques to show the soundness of simple typing rules for symmetric encryption and decryption primitives, showing that well-typed, polynomial-time programs using encryption and decryption satisfy computational probabilistic noninterference, provided that the encryption scheme is IND-CCA secure.

In future work, it would be interesting to determine whether IND-CCA secure encryption is really necessary here. It appears possible to construct pathological IND-CPA secure encryption schemes that allow well-typed programs that use both encryption and decryption to leak H variables, but it is unclear whether "real" IND-CPA secure schemes would allow similar attacks. Finally, it would be valuable to extend to a richer language, for example with public-key encryption, and to carry out case studies to explore the usefulness of the language and type system in developing realistic applications with provable security properties.

Acknowledgments

We are grateful to Michael Backes for shepherding this paper, and to the anonymous referees for helpful comments. This work was partially supported by the National Science Foundation under grant HRD-0317692.

8. REFERENCES

[1] M. Abadi and P. Rogaway. Reconciling two views of cryptography (The computational soundness of formal encryption). In *TCS '00: Proceedings of the IFIP International Conference on Theoretical Computer Science*, pages 3–22, Aug. 2000.

[2] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS 2005)*, pages 197–221, Sept. 2005.

[3] M. Backes and B. Pfitzmann. Computational probabilistic noninterference. In *Proceeding 7th ESORICS*, pages 1–23, 2002.

[4] M. Backes and B. Pfitzmann. Relating symbolic and cryptographic secrecy. In *Proceeding 26th IEEE Symposium on Security and Privacy*, Oakland, California, 2005.

[5] M. Bellare, E. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption: Analysis of DES modes of operation. In *Proceedings of the 38th Symposium on Foundations of Computer Science*, 1997.

[6] M. Bellare and P. Rogaway. Introduction to modern cryptography. At `http://www-cse.ucsd.edu/users/mihir/cse207/classnotes.html`, 2005.

[7] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. *Electronic Notes in Theoretical Computer Science*, 59(3), 2001.

[8] D. Denning and P. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[9] A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *Proceedings 15th IEEE Computer Security Foundations Workshop*, pages 1–17, Cape Breton, Nova Scotia, Canada, June 2002.

[10] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume I. John Wiley & Sons, Inc., Third edition, 1968.

[11] B. Jonsson and K. Larsen. Specification and refinement of probabilistic processes. In *Proc. 6th IEEE Symposium on Logic in Computer Science*, pages 266–277, 1991.

[12] M. H. Kang and I. S. Moskowitz. A pump for rapid, reliable secure communication. In *Proceedings of the 1st ACM Conference on Computer & Communications Security*, pages 119–129, Nov. 1993.

[13] P. Laud. Semantics and program analysis of computationally secure information flow. In *Proceedings 10th ESOP (European Symposium on Programming*, pages 77–91, 2001.

[14] P. Laud. Handling encryption in an analysis for secure information flow. In *Proceedings 12th ESOP (European Symposium on Programming*, pages 159–173, 2003.

[15] P. Laud. Secrecy types for a simulatable cryptographic library. In *Proceedings 12th CCS (ACM Conference on Computer and Communications Security)*, pages 26–35, 2005.

[16] P. Laud and V. Vene. A type system for computationally secure information flow. In *Proceedings of the 15th International Symposium on Fundamentals of Computational Theory*, volume 3623 of *Lecture Notes in Computer Science*, pages 365–377, Lübeck, Germany, 2005.

[17] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings 32nd Symposium on Principles of Programming Languages*, pages 158–170, Jan. 2005.

[18] A. C. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic. *Jif: Java + information flow*. Cornell University, 2004. Available at `http://www.cs.cornell.edu/jif/`.

[19] A. Sabelfeld and A. C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[20] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings 18th IEEE Computer Security Foundations Workshop*, June 2005.

[21] G. Smith. A new type system for secure information flow. In *Proceedings 14th IEEE Computer Security Foundations Workshop*, pages 115–125, Cape Breton, Nova Scotia, Canada, June 2001.

[22] G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proceedings 16th IEEE Computer Security Foundations Workshop*, pages 3–13, Pacific Grove, California, June 2003.

[23] D. Volpano. Secure introduction of one-way functions. In *Proceedings 13th IEEE Computer Security Foundations Workshop*, pages 246–254, Cambridge, UK, June 2000.

[24] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.

[25] B. Warinschi. A computational analysis of the Needham-Schroeder-(Lowe) protocol. In *Proceedings 16th IEEE Computer Security Foundations Workshop*, pages 248–262, Pacific Grove, California, June 2003.