June 2001

Logical Relations for Encryption (Extended Abstract)

Eijiro Sumii
University of Pennsylvania

Benjamin C. Pierce
University of Pennsylvania, bcpierce@cis.upenn.edu

Logical Relations for Encryption (Extended Abstract)

Abstract
The theory of *relational parametricity* and its *logical relations* proof technique are powerful tools for reasoning about information hiding in the polymorphic λ-calculus. We investigate the application of these tools in the security domain by defining a *cryptographic λ-calculus* -- an extension of the standard simply typed λ-calculus with primitives for encryption, decryption, and key generation -- and introducing logical relations for this calculus that can be used to prove behavioral equivalences between programs that rely on encryption.

We illustrate the framework by encoding some simple security protocols, including the Needham-Schroeder public-key protocol. We give a natural account of the well-known attack on the original protocol and a straightforward proof that the improved variant of the protocol is secure.

Logical Relations for Encryption (Extended Abstract)*

Eijiro Sumii†
University of Tokyo
sumii@saul.cis.upenn.edu

Benjamin C. Pierce
University of Pennsylvania
bcpierce@cis.upenn.edu

Abstract

The theory of relational parametricity *and its* logical relations *proof technique are powerful tools for reasoning about information hiding in the polymorphic λ-calculus. We investigate the application of these tools in the security domain by defining a* cryptographic λ-calculus*—an extension of the standard simply typed λ-calculus with primitives for encryption, decryption, and key generation—and introducing logical relations for this calculus that can be used to prove behavioral equivalences between programs that rely on encryption.*

We illustrate the framework by encoding some simple security protocols, including the Needham-Schroeder public-key protocol. We give a natural account of the well-known attack on the original protocol and a straightforward proof that the improved variant of the protocol is secure.

1 Introduction

Information hiding is a central concern in both programming languages and computer security. In the security community, encryption is the fundamental means of hiding information from outsiders. In programming languages, mechanisms such as abstract data types, modules, and parametric polymorphism play an analogous role. Each community has developed a rich set of mathematical tools for reasoning about the hiding of information in applications built using its chosen primitives. Given the intuitive similarity of the notions of hiding in the two domains, it is natural to wonder whether some of these techniques can be transferred from the programming language setting and applied to security problems, or vice versa.

As a first step in this direction, we investigate the application of one well established tool from the theory of programming languages—the concept of *relational parametricity* [23] and its accompanying *logical relations* proof method—in the domain of security protocols.

We begin by defining a *cryptographic λ-calculus*, an extension of the ordinary simply typed λ-calculus with primitives for encryption, decryption and key generation. (One can imagine a large family of different cryptographic λ-calculi, each based on a different set of encryption primitives. For the present study, we use the simplest member of this family—the one where the primitives are assumed to provide *perfect* shared-key encryption.) This calculus offers a suitable mix of structures for our investigation: encryption primitives, since our goal is to reason about programs from the security domain, together with the type structure on which logical relations are built. We now proceed in three steps:

1. We show how some simple security protocols can be modeled by expressions in the cryptographic λ-calculus. The essence of the encoding lies in regarding principals as pairs of the message values they send and functions representing new principals waiting for their next message. Our main example is the Needham-Schroeder public-key protocol [19]. The encoding of this protocol gives a clear account both of the well-known attack on the original protocol and of the resilience of the improved variant of this protocol to the same attack [14].

2. We formalize desired secrecy properties in terms of *behavioral equivalence*. Suppose, for instance, that we would like to prove that a program keeps some integer secret against all possible attacks. Let p_i be an instance of the program with the secret integer being i. If we encode each attacker as a function f that takes the program as an argument and returns an observable value (a boolean, say), then we want to show the equality $f(p_i) = f(p_j)$ for $i \neq j$. Since such a function is itself an expression in the cryptographic λ-calculus, we can use the same language to reason about the attacker and the program.

*A full version including the proofs of the theorems is available at: http://www.yl.is.s.u-tokyo.ac.jp/~sumii/pub/infohide2.ps.gz

†The present work was carried out while the first author was visiting the University of Pennsylvania.

3. We introduce a proof technique for behavioral equivalence based on *logical relations*. The technique gives a method of "relating" (in a formal sense) two programs that differ only in their secrets and that behave equivalently in every other respect. In particular, in its original form in the polymorphic λ-calculus, it gives a method of showing behavioral equivalence between different implementations of the same abstract type—so-called relational parametricity. We adapt the same ideas to the cryptographic λ-calculus, which enables us to prove the equivalence of p_i and p_j from the point of view of an arbitrary attacker f, without quantifying explicitly over all such attackers.

To illustrate these ideas, let us consider a simple system with two principals A and B sharing a secret key z, where A encrypts a secret integer i with z and sends it to B, and B replies by returning $i \bmod 2$. In the standard informal notation, this protocol can be written as:

$$\begin{array}{lll} 1. & A \rightarrow B & : \{i\}_z \\ 2. & B \rightarrow A & : i \bmod 2 \end{array}$$

In the cryptographic λ-calculus, this system can be encoded as a pair p of the ciphertext $\{i\}_z$, which represents the principal A sending the integer i encrypted with z, and a function $\lambda\{x\}_z.\, x \bmod 2$ representing the principal B, which publishes $x \bmod 2$ on receiving an integer x encrypted by z. Let p_i be an instance of the program with the secret integer being i.

$$p_i \quad = \quad \texttt{new}\ z\ \texttt{in}\ \langle \{i\}_z,\ \lambda\{x\}_z.\, x \bmod 2 \rangle$$

Here, the key generation construct new z in ... guarantees that z is *fresh*, that is, different from any other keys and unknown to the attacker.

The network, scheduler, and attackers for this system are encoded as functions operating on this pair. We assume a standard model of a "possible" attacker [8], who is able to intercept, forge, and forward messages, encrypt and decrypt them with any keys known to the attacker, and—in addition—schedule processes arbitrarily. (The last point is not usually emphasized, but generally assumed by considering any possible scheduling when verifying protocols.) In short, it has full control of the network and process scheduler—or, to put it extremely, "the attacker *is* the network and scheduler." Then, the properties to prove are: (i) the system accomplishes its goal under a "good" network/scheduler and (ii) the system does not leak its secret to any possible attacker.

The "good" network/scheduler for this system can be represented as a function f that takes a pair p as an argument and applies its second element $\#_2(p)$, which is a function representing a principal receiving a message, to its first element $\#_1(p)$, which is a value representing a principal sending the message.

$$f \quad = \quad \lambda p.\, \#_2(p)\#_1(p)$$

The correctness of this network/scheduler can be checked by applying f to p, which yields $i \bmod 2$ as expected. (Of course, this network/scheduler is designed to work with this particular system only. It is also possible to encode a generic network/scheduler that will work with a range of protocols, by including the intended receiver's name in each message and delivering messages accordingly.)

On the other hand, the property that the system keeps the concrete value of i secret against any possible attacker, can be stated as a claim of *behavioral equivalence* between, say, p_3 and p_5. That is, $f(p_3)$ and $f(p_5)$ give the same result for any function f returning an observable value.

Why is this? The point here is that p_3 and p_5 differ only in the concrete values of the secret integers and behave equivalently in any other respect. That is, the correspondence between values encrypted by a secret key—i.e., the integers 3 and 5 encrypted by the key z—is preserved by every part of both programs. Indeed, the first elements of the pairs are $\{3\}_z$ and $\{5\}_z$, and the second elements of the pairs are functions that, given the arguments $\{3\}_z$ and $\{5\}_z$, return the same value 1. Since the key z itself is kept secret, no other values can be encrypted by it.

Our logical relation formalizes and generalizes this argument. It demonstrates behavioral equivalence between two programs which differ only in the concrete values of their secrets, i.e., the values encrypted by secret keys. It is defined as follows:

- Two integers are related iff they are equal.
- Two pairs are related iff their elements are related.
- Two functions are related iff they return related values when applied to related arguments.
- Two values encrypted by a secret key k are related iff they are related by $\varphi(k)$, where the *relation environment* φ gives the relations between values encrypted by each secret key.

The soundness theorem for the logical relation now tells us that, in order to prove behavioral equivalence of two programs, it suffices to find some φ such that the logical relation based on φ relates the programs we are interested in.

For instance, in the example above, take $\varphi(z) = \{(3,5)\}$. Then:

- The first elements of p_3 and p_5 are related by the definition of φ.
- The second elements of p_3 and p_5 are related since they return related values (i.e., 1) for any related arguments (i.e., $\{3\}_z$ and $\{5\}_z$).

- Therefore, the pairs p_3 and p_5 are related since their elements are related.

Thus, p_3 and p_5 are guaranteed to be behaviorally equivalent.

In this way, the logical relation allows is to prove the behavioral equivalence of programs—which amounts to proving secrecy—in a compositional manner.

The contributions of this work are twofold: theoretically, it clarifies the intuitive similarity between two forms of information hiding in different domains, namely, encryption in security systems and type abstraction in programming languages; practically, it offers a systematic method of proving secrecy properties of programs using encryption.

The structure of the rest of this paper is as follows. Section 2 presents the syntax and intuitive semantics of the cryptographic λ-calculus and Section 3 demonstrates the use of our framework through examples. Section 4 shows the operational semantics, Section 5 gives a simple type system—which is prerequisite for formalizing the logical relation—and Section 6 defines the logical relation and its variants. Section 7 discusses related work and Section 8 mentions future work.

2 Syntax and Intuitive Semantics

The cryptographic λ-calculus extends a standard λ-calculus with keys, fresh key generation, encryption, and decryption primitives. The formal syntax of the calculus is given in Figure 1. A key k is an element of the countably infinite set of keys K. The key generation form $\texttt{new}\ x\ \texttt{in}\ e$ generates a fresh key, binds it to the variable x, and evaluates the expression e (in which x may appear free). For example, the program $\texttt{new}\ x\ \texttt{in}\ \texttt{new}\ y\ \texttt{in}\ \langle x, y\rangle$ generates two fresh keys and gives the pair of them. The encryption expression $\{e_1\}_{e_2}$ encrypts the value obtained by evaluating the expression e_1 with the key obtained by evaluating the expression e_2. The decryption form $\texttt{let}\ \{x\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4$ attempts to decrypt the ciphertext obtained by evaluating e_2, using the key obtained by evaluating e_1. If the decryption succeeds, it binds the plaintext thus obtained to the variable x and evaluates the expression e_3. If the decryption fails, it instead evaluates e_4. For example, the program $\texttt{let}\ \{x\}_{k'} = \{1\}_k\ \texttt{in}\ x + 2\ \texttt{else}\ 0$ encrypts the integer 1 with the key k, tries to decrypt it with another key k'—which fails because the keys are different—and therefore gives 0. On the other hand, the program $\texttt{let}\ \{x\}_k = \{1\}_k\ \texttt{in}\ x + 2\ \texttt{else}\ 0$ gives 3 because the decryption succeeds.

Several abbreviations are used in examples. We use a tuple with no elements (i.e., $n = 0$) to represent a dummy value, written $\langle\rangle$. We write `true` for $\texttt{in}_1(\langle\rangle)$, `false` for $\texttt{in}_2(\langle\rangle)$, and $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ for $\texttt{case}\ e\ \texttt{of}\ \texttt{in}_1(\_) \Rightarrow e_1 \parallel \texttt{in}_2(\_) \Rightarrow e_2$, respectively, to represent booleans as a disjoint sum of dummy values.[1] We use the "option values" $\texttt{Some}(e)$, abbreviating $\texttt{in}_1(e)$, and `None`, abbreviating $\texttt{in}_2(\langle\rangle)$, to represent the return values of functions that may or may not actually return a meaningful value because of errors such as decryption failure. We use the "pattern matching" function syntax $\lambda\{x\}_{e_1}.\, e_2$ to abbreviate $\lambda y.\, \texttt{let}\ \{x\}_{e_1} = y\ \texttt{in}\ e_2\ \texttt{else None}$ (where y does not appear free in e_1 and e_2), representing functions accepting arguments encrypted by a particular key only. For example, the function $\lambda\{x\}_k.\, \texttt{Some}(x + 1)$ returns $\texttt{Some}(i + 1)$ when applied to an integer i encrypted by the key k, and `None` for any argument not encrypted by k. Finally, we use integer arithmetic in the examples, although we have not included it in the formal definition of the calculus.

Example 1. The expression $\lambda\{x\}_k.\, \texttt{Some}(x \bmod 2)$, which is an abbreviation of the expression $\lambda y.\, \texttt{let}\ \{x\}_k = y\ \texttt{in}\ \texttt{in}_1(x \bmod 2)\ \texttt{else}\ \texttt{in}_2(\langle\rangle)$, represents a function f that accepts an integer x encrypted by the key k and returns its remainder when divided by 2, with the tag `Some` to denote success. For instance, the application $f(\{3\}_k)$ gives the result $\texttt{Some}(1)$ while $f(\{i\}_{k'})$ returns `None` for any $k' \neq k$ and any i.

Example 2. For some integer i, let $p_i = \texttt{new}\ z\ \texttt{in}\ \langle\{i\}_z, \lambda\{x\}_z.\, \texttt{Some}(x \bmod 2)\rangle$ and $f = \lambda p.\, \#_2(p)\#_1(p)$. Then, $f(p_i)$ gives $\texttt{Some}(i \bmod 2)$. This can be seen as a run of an encoding of the following simple system, where two principals A and B share a key z (to be precise, a key bound to the variable z): first, A encrypts and sends i; then, B receives and decrypts it, and publishes its remainder when divided by 2. Here, the function f plays the role of a "good" network and scheduler for this system.

Our cryptographic primitives directly model only shared-key encryption, but they can also be used to obtain the effect of public-key encryption: for any key k, the encryption function $\lambda x.\, \{x\}_k$ and decryption function $\lambda\{x\}_k.\, \texttt{Some}(x)$ can be passed around by themselves and used as encryption and decryption keys.

This trick raises one subtle issue: when we pass the function $\lambda x.\, \{x\}_k$ to the attacker to use as an encryption key, we are implicitly assuming that the attacker is unable to "disassemble" this function and discover the embedded key k. Although this is true if the attacker is itself a λ-context, in general it would be foolish to send out code strings with embedded secret keys. In practice, we need to choose a different *representation* for this function—e.g., as an integer constituting the public part of a public/private key pair, or as a *pointer* that the attacker can use to call a function physically located in the principal—a representation that can be used by the attacker only for the purposes we intend, and

[1] $\texttt{in}_i()$ denotes the i-th injection ("tagging") into a disjoint sum.

$$e ::= x \mid \lambda x.e \mid e_1 e_2 \mid \langle e_1, \ldots, e_n \rangle \mid \#_i(e) \mid \mathtt{in}_i(e) \mid \mathtt{case}\ e\ \mathtt{of}\ \mathtt{in}_1(x_1) \Rightarrow e_1 \,[\!]\, \ldots \,[\!]\, \mathtt{in}_n(x_n) \Rightarrow e_n \mid k \mid \mathtt{new}\ x\ \mathtt{in}\ e \mid \{e_1\}_{e_2} \mid \mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4$$

Figure 1. Syntax

that does not give away information we desire to keep secret. Our framework abstracts away from such choices of representation, simply assuming that a suitable one can be found.

3 Applications

Now we demonstrate the use of our framework on some larger examples, in which concurrent principals communicate with one another by using encryption. Although the cryptographic λ-calculus has no built-in primitives for concurrency or communication, it can emulate a concurrent, communicating system by encoding it as follows. (Recall the example in Section 1.)

- The system as a whole is encoded as a tuple of the processes and their public keys (if any).
- An output process is encoded as the message itself.
- An input process is encoded as a function receiving a message.
- A network/scheduler/attacker for the system is encoded as a function that applies the input functions to the output messages in a certain order, possibly manipulating the messages using the keys that it knows.

Then, the following two properties are desired in general.

- Under a "correct" network and scheduler, i.e., a function applying appropriate messages to appropriate functions in some appropriate order, the program gives some correct result (*soundness*).
- Under *any* possible attacker, the program does not do anything "wrong" such as leaking a secret (*safety*).

3.1 A Program Based on the Needham-Schroeder Public-Key Protocol

Consider the following system using the Needham-Schroeder public-key protocol [19] in a network with a server A, a client B, and an attacker E. (1) B sends its own name B to A. (2) A generates a fresh nonce N_a, pairs it with its own name A, encrypts it with B's public key, and sends it to B. (3) B generates a fresh nonce N_b, pairs it with N_a, encrypts it with A's public key, and sends it to A. (4) A encrypts N_b with B's public key and sends it to B. (5) B encrypts some secret integer i with N_b and sends it to A.[2]

$$\begin{array}{lll} 1. & B \to A & : \ B \\ 2. & A \to B & : \ \{N_a, A\}_{k_b} \\ 3. & B \to A & : \ \{N_a, N_b\}_{k_a} \\ 4. & A \to B & : \ \{N_b\}_{k_b} \\ 5. & B \to A & : \ \{i\}_{N_b} \end{array}$$

Let us encode this system as an expression of the cryptographic λ-calculus. (The result is somewhat complex, because several actions implicitly assumed in the informal definition above—such as identity check of names and keys—are made explicit in the encoding process.)

Recall that we encode such a concurrent, communicating system as a tuple of the principals and the public keys. So we begin the encoding by generating the system's keys and publishing their public portions, that is, A's encryption key, B's encryption key, and E's key:

$$\begin{array}{l} \mathtt{new}\ z_a\ \mathtt{in}\ \mathtt{new}\ z_b\ \mathtt{in}\ \mathtt{new}\ z_e\ \mathtt{in} \\ \langle \lambda x.\{x\}_{z_a}, \lambda x.\{x\}_{z_b}, z_e, \ldots, \ldots \rangle \end{array}$$

Let us now encode B as the fourth element of the tuple. B publishes its own name B. (We assume that names are just integers for the sake of simplicity.) The difference from the previous expression is underlined.

$$\begin{array}{l} \mathtt{new}\ z_a\ \mathtt{in}\ \mathtt{new}\ z_b\ \mathtt{in}\ \mathtt{new}\ z_e\ \mathtt{in} \\ \langle \lambda x.\{x\}_{z_a}, \lambda x.\{x\}_{z_b}, z_e, \underline{\langle B, \ldots \rangle}, \ldots \rangle \end{array}$$

Next, let us encode A as the fifth element of the tuple. A receives a name X, encrypts the pair of a freshly generated nonce N_a and its own name A with X's key, and publishes it.

$$\begin{array}{l} \mathtt{new}\ z_a\ \mathtt{in}\ \mathtt{new}\ z_b\ \mathtt{in}\ \mathtt{new}\ z_e\ \mathtt{in} \\ \langle \lambda x.\{x\}_{z_a}, \lambda x.\{x\}_{z_b}, z_e, \langle B, \ldots \rangle, \\ \underline{\lambda X.\mathtt{new}\ N_a\ \mathtt{in}\ \langle \{\langle N_a, A \rangle\}_{z_x}, \ldots \rangle} \rangle \end{array}$$

Here, z_x abbreviates $\mathtt{if}\ X = A\ \mathtt{then}\ z_a\ \mathtt{else}\ \mathtt{if}\ X = B\ \mathtt{then}\ z_b\ \mathtt{else}\ z_e$. The next action of B is to receive the pair of N_a and a name A' encrypted by z_b, assert $A' = A$, encrypt the pair of N_a and a freshly generated nonce N_b with z_a, and publish it:

$$\begin{array}{l} \mathtt{new}\ z_a\ \mathtt{in}\ \mathtt{new}\ z_b\ \mathtt{in}\ \mathtt{new}\ z_e\ \mathtt{in} \\ \langle \lambda x.\{x\}_{z_a}, \lambda x.\{x\}_{z_b}, z_e, \\ \underline{\langle B, \lambda \{\langle N_a, A \rangle\}_{z_b}.\mathtt{new}\ N_b\ \mathtt{in}\ \mathtt{Some}(\langle \{\langle N_a, N_b \rangle\}_{z_a}, \ldots \rangle)\rangle}, \\ \underline{\lambda X.\mathtt{new}\ N_a\ \mathtt{in}\ \langle \{\langle N_a, A \rangle\}_{z_x}, \ldots \rangle} \rangle \end{array}$$

[2] A reader familiar with security protocols may think this fifth message problematic in practice, because it uses a random nonce as a secret key. We introduced it just for a technical purpose: to state the secrecy of the nonce N_b in terms of the secrecy of the integer i, which is easier to deal with.

Here, $\lambda\{\langle N_a, A\rangle\}_{z_b}. \ldots$ abbreviates $\lambda y.\,\texttt{let}\ \{p\}_{z_b} = y\ \texttt{in}$ $(\texttt{if}\ \#_2(p) = A\ \texttt{then}\ (\lambda N_a. \ldots)\#_1(p)\ \texttt{else None})\ \texttt{else None}$. Next, A receives the pair of a nonce N'_a and N_b, asserts $N'_a = N_a$, encrypts N_b with z_b, and publishes it:

$$\begin{array}{l}\texttt{new}\ z_a\ \texttt{in new}\ z_b\ \texttt{in new}\ z_e\ \texttt{in}\\ \langle \lambda x.\{x\}_{z_a}, \lambda x.\{x\}_{z_b}, z_e,\\ \ \langle B, \lambda\{\langle N_a, A\rangle\}_{z_b}.\texttt{new}\ N_b\ \texttt{in Some}(\langle\{\langle N_a, N_b\rangle\}_{z_a}, \ldots\rangle)\rangle,\\ \ \langle \lambda X.\texttt{new}\ N_a\ \texttt{in}\ \langle\{\langle N_a, A\rangle\}_{z_x},\\ \qquad\qquad \underline{\lambda\{N_a, N_x\}_{z_a}.\texttt{Some}(\{N_x\}_{z_b})}\rangle\rangle\rangle\end{array}$$

Here, $\lambda\{N_a, N_x\}_{z_a}. \ldots$ abbreviates $\lambda y.\,\texttt{let}\ \{p\}_{z_a} = y\ \texttt{in}$ $(\texttt{let}\ \{\_\}_{\#_1(p)} = \{0\}_{N_a}\ \texttt{in}\ (\lambda N_x. \ldots)\#_2(p)\ \texttt{else None})$ $\texttt{else None}$. Last, B receives a nonce N'_b encrypted by z_b, asserts $N'_b = N_b$, encrypts i with N_b, and publishes it:

$$\begin{array}{l}\texttt{new}\ z_a\ \texttt{in new}\ z_b\ \texttt{in new}\ z_e\ \texttt{in}\\ \langle \lambda x.\{x\}_{z_a}, \lambda x.\{x\}_{z_b}, z_e,\\ \ \langle B, \lambda\{\langle N_a, A\rangle\}_{z_b}.\texttt{new}\ N_b\ \texttt{in}\\ \qquad \texttt{Some}(\langle\{\langle N_a, N_b\rangle\}_{z_a}, \underline{\lambda\{N_b\}_{z_b}.\texttt{Some}(\{i\}_{N_b})}\rangle)\rangle,\\ \ \langle \lambda X.\texttt{new}\ N_a\ \texttt{in}\ \langle\{\langle N_a, A\rangle\}_{z_x},\\ \qquad\qquad \lambda\{N_a, N_x\}_{z_a}.\texttt{Some}(\{N_x\}_{z_x})\rangle\rangle\rangle\end{array}$$

Let NS_i be the expression above. Here, $\lambda\{N_b\}_{z_b}. \ldots$ abbreviates $\lambda y.\,\texttt{let}\ \{N'_b\}_{z_b} = y\ \texttt{in}\ (\texttt{let}\ \{\_\}_{N'_b} = \{0\}_{N_b}\ \texttt{in}$ $\ldots\ \texttt{else None})\ \texttt{else None}$.

A correct run of this system can be expressed by evaluation of this expression under the following function *Good*, which represents a "good" network/scheduler for this system.

$$\begin{array}{l}\lambda p.\,\texttt{let}\ \langle B, c_b\rangle = \#_4(p)\ \texttt{in}\\ \quad \texttt{let}\ \langle m_1, c_a\rangle = \#_5(p)B\ \texttt{in}\\ \quad \texttt{let Some}(\langle m_2, c'_b\rangle) = c_b m_1\ \texttt{in}\\ \quad \texttt{let Some}(m_3) = c_a m_2\ \texttt{in}\\ \quad \texttt{let Some}(m_4) = c'_b m_3\ \texttt{in}\\ \quad \texttt{Some}(m_4)\end{array}$$

Here, $\texttt{let}\ \langle X, c_b\rangle = \ldots\ \texttt{in}\ \ldots$ etc. are syntactic sugar that tries pattern matching and returns None if it fails, as are the abbreviations above. $Good(NS_i)$ indeed evaluates to $\texttt{Some}(\{i\}_{N_b})$ for some fresh N_b, which means a successful execution of the system.

It is well known that a use of the Needham-Schroeder public-key protocol such as the system above—namely, letting the server A accept a request not only from the friendly client B but also from the malicious attacker E—is vulnerable to the following man-in-the-middle attack, which allows E to impersonate A with respect to B [14]. (1) B sends its own name B to A, but E intercepts it. (1′) E sends its own name E to A. (2′) A generates a fresh nonce N_a, pair it with A, encrypt it with E's public key, and send it to E. (2) E encrypts the pair of N_a and A with B's public key and send it to B, pretending to be A. (3,3′) B generates a fresh nonce N_b, pair it with N_a, encrypt it with A's public key, and send it to A. (4′) A encrypts N_b with E's public key and send it to E. (4) E encrypts N_b with B's public key and send it to B, pretending to be A. (5) B encrypts i with N_b and send it to A, but E intercepts and decrypts it.

$$\begin{array}{lcll}1. & B \to E(A) & : & B\\ 1'. & E \to A & : & E\\ 2'. & A \to E & : & \{N_a, A\}_{k_e}\\ 2. & E(A) \to B & : & \{N_a, A\}_{k_b}\\ 3, 3'. & B \to A & : & \{N_a, N_b\}_{k_a}\\ 4'. & A \to E & : & \{N_b\}_{k_e}\\ 4. & E(A) \to B & : & \{N_b\}_{k_b}\\ 5. & B \to E(A) & : & \{i\}_{N_b}\end{array}$$

This attack can be expressed in the cryptographic λ-calculus by the following function *Evil* for the expression NS_i above.

$$\begin{array}{l}\lambda p.\,\texttt{let}\ \langle B, c_b\rangle = \#_4(p)\ \texttt{in}\\ \quad \texttt{let}\ \langle\{\langle N_a, A\rangle\}_{\#_3(p)}, c_a\rangle = \#_5(p)E\ \texttt{in}\\ \quad \texttt{let Some}(\langle m, c'_b\rangle) = c_b(\#_2(p)\langle N_a, A\rangle)\ \texttt{in}\\ \quad \texttt{let Some}(\{N_b\}_{\#_3(p)}) = c_a m\ \texttt{in}\\ \quad \texttt{let Some}(\{i\}_{N_b}) = c'_b(\#_2(p)N_b)\ \texttt{in}\\ \quad \texttt{Some}(i)\end{array}$$

Again, $\texttt{let}\ \langle\{\langle N_a, A\rangle\}_{\#_3(p)}, c_a\rangle = \ldots\ \texttt{in}\ \ldots$ etc. are syntactic sugar for pattern matching. $Evil(NS_i)$ indeed evaluates to $\texttt{Some}(i)$, which leaks the secret. In other words, if $i \neq j$, then $Evil(NS_i)$ and $Evil(NS_j)$ evaluate to different observable values, so NS_i and NS_j are not behaviorally equivalent, and therefore non-interference fails.

3.2 A Program Based on the Improved Needham-Schroeder Public-Key Protocol

Consider the following variant of the system above, using an improved version of the Needham-Schroeder public-key protocol [14]. (The difference from the original version is underlined.) (1) B sends its own name B to A. (2) A generates a fresh nonce N_a, pair it with its own name A, encrypts it with B's public key, and sends it to B. (3) B generates a fresh nonce N_b, tuples it with N_a and $\underline{B}$, encrypts it with A's public key, and sends it to A. (4) A encrypts N_b with B's public key and sends it to B. (5) B encrypts some secret integer i with N_b and sends it to A.

$$\begin{array}{lcll}1. & B \to A & : & B\\ 2. & A \to B & : & \{N_a, A\}_{k_b}\\ 3. & B \to A & : & \{N_a, N_b, \underline{B}\}_{k_a}\\ 4. & A \to B & : & \{N_b\}_{k_b}\\ 5. & B \to A & : & \{i\}_{N_b}\end{array}$$

Following the same lines as the encoding of the original system, this improved system can be encoded as follows.

$$\begin{array}{l}\texttt{new } z_a \texttt{ in new } z_b \texttt{ in new } z_e \texttt{ in}\\ \langle \lambda x.\{x\}_{z_a}, \lambda x.\{x\}_{z_b}, z_e,\\ \ \langle B, \lambda\{\langle N_a, A\rangle\}_{z_b}.\texttt{new } N_b \texttt{ in}\\ \qquad \texttt{Some}(\langle\{\langle N_a, N_b, \underline{B}\rangle\}_{z_a}, \lambda\{N_b\}_{z_b}.\texttt{Some}(\{i\}_{N_b})\rangle)\rangle,\\ \ \langle \lambda X.\texttt{new } N_a \texttt{ in } \langle\{\langle N_a, A\rangle\}_{z_x},\\ \qquad\qquad \lambda\{N_a, N_x, \underline{X}\}_{z_a}.\texttt{Some}(\{N_x\}_{z_x})\rangle\rangle\rangle\end{array}$$

Let NS'_i be the expression above. Here, $\lambda\{N_a, N_x, \underline{X}\}_{z_a}. \ldots$ abbreviates $\lambda y.\texttt{let } \{p\}_{z_a} = y \texttt{ in } (\texttt{let } \{\_\}_{\#_1(p)} = \{0\}_{N_a} \texttt{ in } \underline{(\texttt{if } \#_3(p) = X \texttt{ then } (\lambda N_x.\ldots) \#_2(p) \texttt{ else None})}) \texttt{else None}) \texttt{else None}$.

How does this change prevent the attack? Recall that *Evil* was the following function.

$$\begin{array}{l}\lambda p.\texttt{let } \langle B, c_b\rangle = \#_4(p) \texttt{ in}\\ \quad \texttt{let } \langle\{\langle N_a, A\rangle\}_{\#_3(p)}, c_a\rangle = \#_5(p)E \texttt{ in}\\ \quad \texttt{let Some}(\langle m, c'_b\rangle) = c_b(\#_2(p)\langle N_a, A\rangle) \texttt{ in}\\ \quad \texttt{let Some}(\{N_b\}_{\#_3(p)}) = c_a m \texttt{ in}\\ \quad \texttt{let Some}(\{i\}_{N_b}) = c'_b(\#_2(p)N_b) \texttt{ in}\\ \quad \texttt{Some}(i)\end{array}$$

When the attacker forwards the message $m = \{\langle N_a, N_b, B\rangle\}_{z_b}$ (which is encrypted by B's secret key and cannot be decrypted by the attacker) from B to A, A tries to match B against $X = E$, which fails. Thus, $Evil(NS'_i)$ reduces to *None* for any i and fails to leak the secret. (In Section 6, we formally prove this secrecy property against any possible attackers, using our logical relation.)

3.3 A Program Based on the ffgg Protocol

The ffgg protocol is an artificial protocol with an intentional flaw, which is secure as long as only one process runs for each principal, but insecure when more than one process runs for a principal [17]. Although the cryptographic λ-calculus is sequential, it is actually expressive enough to encode this so-called "parallel attack" by interleaving.

To see this, let us encode the following system with two principals A and B using the ffgg protocol. (1) A sends its own name A to B. (2) B generates two fresh nonces N_1 and N_2 and sends them to A. (3) A tuples N_1, N_2, and some secret value M, encrypts them with a shared secret key k, and sends them to B. However, B does not check the identity of N_2 and lets x and y be the second and third elements of the tuple, respectively. (4) B tuples x, y and N_1, encrypts them with k, and sends them to A with N_1 and x.

$$\begin{array}{llll}1. & A \to B & : & A\\ 2. & B \to A & : & N_1, N_2\\ 3. & A \to B & : & \{N_1, N_2, M\}_k \text{ as } \{N_1, x, y\}_k\\ 4. & B \to A & : & N_1, x, \{x, y, N_1\}_k\end{array}$$

This system can be encoded as follows.

$$\begin{array}{l}\texttt{new } z \texttt{ in}\\ \langle\langle A, \lambda\langle N_1, N_2\rangle.\{\langle N_1, N_2, M\rangle\}_z\rangle,\\ \ \lambda A.\texttt{new } N_1 \texttt{ in new } N_2 \texttt{ in}\\ \ \langle\langle N_1, N_2\rangle, \lambda\{N_1, x, y\}_z.\langle N_1, x, \{\langle x, y, N_1\rangle\}_z\rangle\rangle\rangle\end{array}$$

Again, for the sake of brevity, we used syntactic sugar for pattern matching.

The attack to this system is as follows. (1) A sends its own name A to B. (1′) Pretending to be A, the attacker E sends A to another process B′ running for B. (2a) B generates two fresh nonces N_1 and N_2, and send them to A, but E intercepts them. (2′) B′ generates other two fresh nonces N'_1 and N'_2, and send them to A, but E again intercepts them. (2b) E sends N_1 and N'_1 to A, pretending to be B. (3) A tuples N_1, N'_1 and M, encrypts them with k, and sends them to B. (4) B tuples N'_1, M and N_1, encrypts them with k, and send them to A with N_1 and N'_1, but E intercepts them. (3′) E forwards the tuple of N'_1, M and N_1 encrypted by k to B′, pretending to be A. (4′) B′ tuples M, N_1 and N'_1, encrypts them with k, and send them to A with N'_1 and M, but E intercepts them.

$$\begin{array}{llll}1. & A \to B & : & A\\ 1'. & E(A) \to B' & : & A\\ 2a. & B \to E(A) & : & N_1, N_2\\ 2'. & B' \to E(A) & : & N'_1, N'_2\\ 2b. & E(B) \to A & : & N_1, N'_1\\ 3. & A \to B & : & \{N_1, N'_1, M\}_k\\ 4. & B \to E(A) & : & N_1, N'_1, \{N'_1, M, N_1\}_k\\ 3'. & E(A) \to B' & : & \{N'_1, M, N_1\}_k\\ 4'. & B' \to E(A) & : & N'_1, M, \{M, N_1, N'_1\}_k\end{array}$$

This attack can be encoded as the following function on the expression above.

$$\begin{array}{l}\lambda p.\texttt{let } \langle A, c_a\rangle = \#_1(p) \texttt{ in}\\ \quad \texttt{let } \langle\langle N_1, N_2\rangle, c_b\rangle = \#_2(p)A \texttt{ in}\\ \quad \texttt{let } \langle\langle N'_1, N'_2\rangle, c'_b\rangle = \#_2(p)A \texttt{ in}\\ \quad \texttt{let } m = c_a\langle N_1, N'_1\rangle \texttt{ in}\\ \quad \texttt{let } \langle N_1, N'_1, m'\rangle = c_b m \texttt{ in}\\ \quad \texttt{let } \langle N'_1, M, m''\rangle = c'_b m' \texttt{ in}\\ \quad \texttt{Some}(M)\end{array}$$

This function indeed reveals the secret value M in the expression above. Note that the function representing the principal B did not have to be replicated explicitly, because functions in λ-calculus can be applied any times by default.

By the way, in this encoding, there actually exists an even simpler function which leaks the secret.

$$\begin{array}{l}\lambda p.\texttt{let } \langle A, c_a\rangle = \#_1(p) \texttt{ in}\\ \quad \texttt{let } \langle\langle N_1, N_2\rangle, c_b\rangle = \#_2(p)A \texttt{ in}\\ \quad \texttt{let } m = c_a\langle N_1, N_2\rangle \texttt{ in}\\ \quad \texttt{let } \langle N_1, N_2, m'\rangle = c_b m \texttt{ in}\\ \quad \texttt{let } \langle N_2, M, m''\rangle = c_b m' \texttt{ in}\\ \quad \texttt{Some}(M)\end{array}$$

This attack is usually considered impossible in reality, because it applies the "continuation" function c_b twice, which means exploiting one state of (a process running for) the principal B more than once. This kind of false attacks could perhaps be excluded in our framework by using *linear* types for continuation functions like c_b. See Section 8 for details.

4 Operational Semantics

In this section and the two that follow, we present the cryptographic λ-calculus, its type system, and the logical relations proof technique more formally.

The semantics of the calculus is defined by an *evaluation* relation mapping terms to results. For the ordinary λ-calculus, the evaluation relation has the form $e \Downarrow v$, read "evaluation of the (closed) expression e yields the value v." However, since the cryptographic λ-calculus includes a primitive for key generation, we need to represent "the set of keys generated so far" in some rigorous fashion. We do this by annotating the evaluation relation with a set s, representing the keys that have already been used when evaluation begins, and a set s', representing the keys that have been used when evaluation finishes. To be precise, we define the relation $(s)e \Downarrow V$ where V is either of the form $(s')v$ or *Error* (signalling a run-time type error). We maintain the invariant that $(s)e \Downarrow (s')v$ implies $s \subseteq s'$, that is, $s' \setminus s$ is the set of fresh keys generated during the evaluation of e. The evaluation relation is defined inductively by the rules in Figure 2.

Most of the evaluation rules are standard and straightforward; we explain just a few important points. In the rule for key generation, k is guaranteed to be "freshly generated" because $s \uplus \{k\}$ is defined and therefore $k \notin s$. (Here, $s \uplus s'$ is defined as $s \cup s'$ if $s \cap s' = \emptyset$, and undefined otherwise.) This is the rule that increases the set of keys. In the rule for decryption, we first evaluate e_1 to obtain the decryption key k_1, then e_2 is evaluated to obtain a ciphertext of the form $\{v\}_{k_2}$. If e_1 does not evaluate to a key or e_2 does not evaluate to a ciphertext, then a type error occurs. Otherwise, if the two keys match ($k_1 = k_2$), the body e_3 is evaluated, with x bound to the decrypted plaintext v. Otherwise, the `else` clause e_4 is evaluated.

The following theorem and corollary state that the result of evaluating an expression is unique, modulo the names of freshly generated keys. (We write $Keys(e)$ for the set of keys syntactically appearing in e.)

Theorem 3. Let $s_1 \supseteq Keys(e)$ and let θ be a one-to-one substitution from s_1 to another set of keys s_2. If $(s_1)e \Downarrow (s_1 \uplus s_1')v_1$ and $(s_2)\theta e \Downarrow V$, then V has the form $(s_2 \uplus s_2')v_2$ and there exists some one-to-one substitution θ' from s_1' to s_2' such that $v_2 = (\theta \uplus \theta')v_1$.

Corollary 4 (Uniqueness of Evaluation Result). Let $s \supseteq Keys(e)$. If $(s)e \Downarrow (s \uplus s_1')v_1$ and $(s)e \Downarrow V$, then V has the form $(s \uplus s_2')v_2$ and there exists some one-to-one substitution θ' from s_1' to s_2' such that $v_2 = \theta' v_1$.

5 Type System

In this section, we define a simple type system for the cryptographic λ-calculus. Types in this setting play not only the traditional role of guaranteeing the absence of run-time type errors (a well-typed term cannot evaluate to *Error*), but, more importantly, provide a framework for the reasoning method we consider in the next section. (The fundamental definition of the logical relation proceeds by induction on types.)

In addition to the values found in the ordinary λ-calculus, the cryptographic λ-calculus has keys and ciphertexts. Therefore, besides the usual function, product, and sum types of the simply typed λ-calculus, we introduce a key type $\texttt{key}[\tau]$, whose elements are keys that can be used to encrypt values of type τ, and a ciphertext type $\texttt{bits}[\tau]$, whose elements are ciphertexts containing a plaintext value of type τ. Thus, keys of a given type cannot be used to encrypt values of different types, and ciphertexts of a given type cannot contain plaintext values of different types. This restriction is not particularly bothersome, since values of (finitely many) different types can always be injected into a common sum type. (Actually, to guarantee type safety, we do not need to annotate both key types and ciphertext types with their underlying plaintext types, but doing so simplifies the definition of the logical relations in Section 6.)

The typing judgment has the form $\Gamma, \Delta \vdash e : \tau$, read "under the type environment Γ for variables and the type environment Δ for keys, the expression e has the type τ, i.e., e evaluates to a value of type τ." The typing rules (which are straightforward) are given in Figure 3. Here, $f \uplus f'$ for two mappings f and f' is defined as $(f \uplus f')(x) = f(x)$ for $x \in dom(f)$ and $(f \uplus f')(y) = f'(y)$ for $y \in dom(f')$ if $dom(f) \cap dom(f') = \emptyset$, and undefined otherwise. Note that the type environment Δ for keys is used in the rule (Key) in the same way the type environment Γ for variables is used in the rule (Var). For the sake of readability, we often write `bool` for `unit` + `unit` and $\texttt{option}[\tau]$ for τ + `unit`, where `unit` is the type of a tuple with no elements.

In what follows, we often abbreviate a sequence of the form $X_1, \ldots, X_n$ as $\tilde{X}$ and a proposition of the form $\bigwedge_{1 \le j \le m} P(Y_{1j}, \ldots, Y_{nj})$ as $P(\tilde{Y}_1, \ldots, \tilde{Y}_n)$. For example, $\tilde{k} \in \tilde{s}$ abbreviates $(k_1 \in s_1) \wedge \ldots \wedge (k_n \in s_n)$.

The following theorem and corollary state that the evaluation of a well-typed program never causes a type error.

Theorem 5. Suppose $\Gamma, \Delta \vdash e : \tau$ and $\emptyset, \Delta \vdash \tilde{v} : \tilde{\tau}$ for $\Gamma = \{\tilde{x} \mapsto \tilde{\tau}\}$. If $(s)[\tilde{v}/\tilde{x}]e \Downarrow V$ for $s = dom(\Delta)$, then

$$
\begin{array}{rcl}
v & ::= & \lambda x.e \mid \langle v_1, \ldots, v_n \rangle \mid \mathtt{in}_i(v) \mid k \mid \{v\}_k \\
V & ::= & (s)v \mid \mathit{Error}
\end{array}
$$

$$(s)\lambda x.e \Downarrow (s)\lambda x.e \qquad \frac{(s)e_1 \Downarrow (s_1)\lambda x.e \quad (s_1)e_2 \Downarrow (s_2)v \quad (s_2)[v/x]e \Downarrow V}{(s)e_1e_2 \Downarrow V} \qquad \frac{(s_0)e_1 \Downarrow (s_1)v_1 \quad \ldots \quad (s_{n-1})e_n \Downarrow (s_n)v_n}{(s_0)\langle e_1, \ldots, e_n \rangle \Downarrow (s_n)\langle v_1, \ldots, v_n \rangle}$$

$$\frac{(s)e \Downarrow (s')\langle \ldots, v_i, \ldots \rangle}{(s)\#_i(e) \Downarrow (s')v_i} \qquad \frac{(s)e \Downarrow (s')v}{(s)\mathtt{in}_i(e) \Downarrow (s')\mathtt{in}_i(v)} \qquad \frac{(s)e \Downarrow (s')\mathtt{in}_i(v) \quad (s')[v/x_i]e_i \Downarrow V}{(s)\mathtt{case}\ e\ \mathtt{of}\ \mathtt{in}_1(x_1) \Rightarrow e_1 \parallel \ldots \parallel \mathtt{in}_n(x_n) \Rightarrow e_n \Downarrow V}$$

$$(s)k \Downarrow (s)k \qquad \frac{(s \uplus \{k\})[k/x]e \Downarrow V}{(s)\mathtt{new}\ x\ \mathtt{in}\ e \Downarrow V} \qquad \frac{(s)e_1 \Downarrow (s_1)v \quad (s_1)e_2 \Downarrow (s_2)k}{(s)\{e_1\}_{e_2} \Downarrow (s_2)\{v\}_k}$$

$$\frac{(s)e_1 \Downarrow (s_1)k_1 \quad (s_1)e_2 \Downarrow (s_2)\{v\}_{k_2} \quad (s_2)[v/x]e_3 \Downarrow V \quad k_1 = k_2}{(s)\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 \Downarrow V}$$

$$\frac{(s)e_1 \Downarrow (s_1)k_1 \quad (s_1)e_2 \Downarrow (s_2)\{v\}_{k_2} \quad (s_2)e_4 \Downarrow V \quad k_1 \neq k_2}{(s)\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 \Downarrow V}$$

$$\frac{(s)e_1 \Downarrow \mathit{Error}}{(s)e_1e_2 \Downarrow \mathit{Error}} \qquad \frac{(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow \mathit{Error}}{(s)e_1e_2 \Downarrow \mathit{Error}} \qquad \frac{(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow (s_2)v_2 \quad v_1 \neq \lambda x.e}{(s)e_1e_2 \Downarrow \mathit{Error}}$$

$$\frac{(s_0)e_1 \Downarrow (s_1)v_1 \quad \ldots \quad (s_{i-1})e_i \Downarrow \mathit{Error}}{(s_0)\langle e_1, \ldots, e_n \rangle \Downarrow \mathit{Error}} \qquad \frac{(s)e \Downarrow \mathit{Error}}{(s)\#_i(e) \Downarrow \mathit{Error}} \qquad \frac{(s)e \Downarrow (s')v \quad v \neq \langle \ldots, v_i, \ldots \rangle}{(s)\#_i(e) \Downarrow \mathit{Error}} \qquad \frac{(s)e \Downarrow \mathit{Error}}{(s)\mathtt{in}_i(e) \Downarrow \mathit{Error}}$$

$$\frac{(s)e \Downarrow \mathit{Error}}{(s)\mathtt{case}\ e\ \mathtt{of}\ \mathtt{in}_1(x_1) \Rightarrow e_1 \parallel \ldots \parallel \mathtt{in}_n(x_n) \Rightarrow e_n \Downarrow \mathit{Error}}$$

$$\frac{(s)e \Downarrow (s')v \quad v \neq \mathtt{in}_1(v_1), \ldots, \mathtt{in}_n(v_n)}{(s)\mathtt{case}\ e\ \mathtt{of}\ \mathtt{in}_1(x_1) \Rightarrow e_1 \parallel \ldots \parallel \mathtt{in}_n(x_n) \Rightarrow e_n \Downarrow \mathit{Error}}$$

$$\frac{(s)e_1 \Downarrow \mathit{Error}}{(s)\{e_1\}_{e_2} \Downarrow \mathit{Error}} \qquad \frac{(s)e_1 \Downarrow (s_1)v \quad (s_1)e_2 \Downarrow \mathit{Error}}{(s)\{e_1\}_{e_2} \Downarrow \mathit{Error}} \qquad \frac{(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow (s_2)v_2 \quad v_2 \neq k}{(s)\{e_1\}_{e_2} \Downarrow \mathit{Error}}$$

$$\frac{(s)e_1 \Downarrow \mathit{Error}}{(s)\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 \Downarrow \mathit{Error}} \qquad \frac{(s)e_1 \Downarrow (s_1)v \quad (s_1)e_2 \Downarrow \mathit{Error}}{(s)\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 \Downarrow \mathit{Error}}$$

$$\frac{(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow (s_2)v_2 \quad v_1 \neq k}{(s)\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 \Downarrow \mathit{Error}} \qquad \frac{(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow (s_2)v_2 \quad v_2 \neq \{v\}_k}{(s)\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 \Downarrow \mathit{Error}}$$

Figure 2. Semantics

$$\tau ::= \tau_1 \to \tau_2 \mid \tau_1 \times \cdots \times \tau_n \mid \tau_1 + \cdots + \tau_n \mid \mathtt{key}[\tau] \mid \mathtt{bits}[\tau]$$

$$\Gamma, \Delta \vdash x : \Gamma(x) \text{ (Var)} \quad \frac{\Gamma \uplus \{x \mapsto \tau_1\}, \Delta \vdash e : \tau_2}{\Gamma, \Delta \vdash \lambda x.\, e : \tau_1 \to \tau_2}\text{(Abs)} \quad \frac{\Gamma, \Delta \vdash e_1 : \tau' \to \tau \quad \Gamma, \Delta \vdash e_2 : \tau'}{\Gamma, \Delta \vdash e_1 e_2 : \tau}\text{(App)}$$

$$\frac{\Gamma, \Delta \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma, \Delta \vdash e_n : \tau_n}{\Gamma, \Delta \vdash \langle e_1, \ldots, e_n \rangle : \tau_1 \times \cdots \times \tau_n}\text{(Pair)} \quad \frac{\Gamma, \Delta \vdash e : \tau_1 \times \cdots \times \tau_i \times \cdots \times \tau_n}{\Gamma, \Delta \vdash \#_i(e) : \tau_i}\text{(Proj)}$$

$$\frac{\Gamma, \Delta \vdash e : \tau_i}{\Gamma, \Delta \vdash \mathtt{in}_i(e) : \tau_1 + \cdots + \tau_i + \cdots + \tau_n}\text{(In)}$$

$$\frac{\Gamma, \Delta \vdash e : \tau_1 + \cdots + \tau_n \quad \Gamma \uplus \{x_1 \mapsto \tau_1\}, \Delta \vdash e_1 : \tau \quad \ldots \quad \Gamma \uplus \{x_n \mapsto \tau_n\}, \Delta \vdash e_n : \tau}{\Gamma, \Delta \vdash \mathtt{case}\ e\ \mathtt{of}\ \mathtt{in}_1(x_1) \Rightarrow e_1 \parallel \ldots \parallel \mathtt{in}_n(x_n) \Rightarrow e_n : \tau}\text{(Case)}$$

$$\Gamma, \Delta \vdash k : \mathtt{key}[\Delta(k)] \text{ (Key)} \quad \frac{\Gamma \uplus \{x \mapsto \mathtt{key}[\tau']\}, \Delta \vdash e : \tau}{\Gamma, \Delta \vdash \mathtt{new}\ x\ \mathtt{in}\ e : \tau}\text{(New)} \quad \frac{\Gamma, \Delta \vdash e_1 : \tau \quad \Gamma, \Delta \vdash e_2 : \mathtt{key}[\tau]}{\Gamma, \Delta \vdash \{e_1\}_{e_2} : \mathtt{bits}[\tau]}\text{(Enc)}$$

$$\frac{\Gamma, \Delta \vdash e_1 : \mathtt{key}[\tau'] \quad \Gamma, \Delta \vdash e_2 : \mathtt{bits}[\tau'] \quad \Gamma \uplus \{x \mapsto \tau'\}, \Delta \vdash e_3 : \tau \quad \Gamma, \Delta \vdash e_4 : \tau}{\Gamma, \Delta \vdash \mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 : \tau}\text{(Dec)}$$

Figure 3. Type System

there exist some v and Δ' such that $V = (s \uplus s')v$ and $\emptyset, \Delta \uplus \Delta' \vdash v : \tau$ for $s' = dom(\Delta')$.

Corollary 6 (Type Safety). If $\emptyset, \emptyset \vdash e : \tau$, then $(\emptyset)e \not\Downarrow \mathit{Error}$.

One subtle point deserves mention, concerning the relation between types and the modeling of security protocols. Since we intend to represent both principals and attackers as terms of the cryptographic λ-calculus, if we restrict our attention to only well-typed terms, we seem to run the risk of artificially (and unrealistically) restricting the power of the attackers we can model. In particular, since the calculus under this type system is strongly normalizing (i.e., every well-typed program terminates), the attackers are not Turing-complete.

However, we believe that the present simple type system is flexible enough to allow typical attacks: indeed, all of the attacks we have seen so far are well-typed in the type system. As for so-called "type attacks," which make principals confuse values of different types, they are either (1) actually well-typed in the present type system, which does not distinguish nonces from keys, or (2) easily prevented by standard dynamic type checking.

6 Logical Relations for Encryption

Recall the family of expressions p_i from Example 2:

$$p_i = \mathtt{new}\ z\ \mathtt{in}\ \langle \{i\}_z, \lambda\{x\}_z.\, \mathtt{Some}(x \bmod 2) \rangle$$

Suppose we want to argue that each p_i keeps its concrete value of i secret from any possible attacker. Intuitively, this is so because the only capabilities p_i provides to an attacker (at least, if that attacker can be represented as an expression of the cryptographic λ-calculus) are a ciphertext encrypting i under a key that the attacker cannot learn plus a function that will return just the least significant bit of a number encrypted with this key.

The intuition that the concrete value of i is kept secret can be formulated more precisely as a *non-interference* condition: for any i and j such that $i \bmod 2 = j \bmod 2$ (i.e., such that the part of the information that we do allow p_i and p_j to reveal is the same), we want to prove that p_i and p_j are behaviorally equivalent, in the following sense.

Definition 7 (Extensional Equivalence). We say that $\vdash e \approx e' : \tau$, pronounced "the expressions e and e' are extensionally equivalent at type τ," iff, for any f with $\emptyset, \emptyset \vdash f : \tau \to \mathtt{bool}$, there exist some s and s' such that $(\emptyset)fe \Downarrow (s)\mathtt{true}$ and $(\emptyset)fe' \Downarrow (s')\mathtt{true}$ or $(\emptyset)fe \Downarrow (s)\mathtt{false}$ and $(\emptyset)fe' \Downarrow (s')\mathtt{false}$.

Although extensional equivalence is defined for closed expressions only, it can be used to prove the more general property of *contextual equivalence* for open expressions as follows. Take any expressions e and e' of type τ and any context $C[\,]$ of type $\mathtt{bool}$ with a hole of type τ. Let $\tilde{x}$ be the free variables of e and e', and let $f = \lambda x_0.\, C[x_0\tilde{x}]$, $e_0 = \lambda\tilde{x}.\, e$, and $e_0' = \lambda\tilde{x}.\, e'$. Then $fe_0 = fe_0'$ implies $C[e] = C[e']$. Thus, contextual equivalence of e and e' follows from extensional equivalence of e_0 and e_0'.

In the following subsections, we define three variants of the logical relation proof technique for extensional equivalence. The first one shows the basic ideas, but it is not pow-

erful enough to prove secrecy properties of realistic programs, such as (the encoding of the system based on) the improved Needham-Schroeder public-key protocol in Section 3. The second and third are extensions of the basic logical relation, one for addressing the issue of "a key encrypting another key" and the other for accommodating discrepancies in the number of keys used in the programs being compared.

6.1 Basic Logical Relation

Extensional equivalence is difficult to prove directly because it involves a quantification over all functions f of type $\tau \to \texttt{bool}$, which are infinitely many in general. Instead, we would like prove it in a compositional manner, by showing that each part of two programs behave equivalently. However, this approach will not suffice to prove any interesting case of extensional equivalence if we do not consider the correspondence between ciphertexts. Consider, for example, the expressions $e = \texttt{new}\ z\ \texttt{in}\ \langle\{\texttt{true}\}_z, \{\texttt{false}\}_z, \lambda\{x\}_z.\texttt{Some}(x)\rangle$ and $e' = \texttt{new}\ z\ \texttt{in}\ \langle\{\texttt{false}\}_z, \{\texttt{true}\}_z, \lambda\{x\}_z.\texttt{Some}(\texttt{not}(x))\rangle$. Although these tuples are equivalent, it can not be shown, say, that the third elements $\lambda\{x\}_k.\texttt{Some}(x)$ and $\lambda\{x\}_k.\texttt{Some}(\texttt{not}(x))$ are equivalent for any freshly generated key k, without knowing (1) the fact that k is kept secret throughout the whole programs and (2) the relation between values encrypted by k.

Thus, we generalize $\vdash e \approx e' : \tau$ to the *logical relation* $\varphi \vdash e \sim e' : \tau$, where the parameter φ is a *relation environment*: a mapping from keys to relations, associating to each secret key k a relation $\varphi(k)$ between the values that may be encrypted by k. Given φ, the family of relations $\varphi \vdash e \sim e' : \tau$ is defined by induction on τ as follows:

- Two functions are related iff they map any related arguments to related results.
- Two pairs are related iff their corresponding elements are related.
- Two tagged values are related iff their tags are equal and their bodies are related.
- Two keys are related iff they are identical and not secret. Here, the set of secret keys is identified with the domain of φ.
- Two ciphertexts $\{v\}_k$ and $\{v'\}_{k'}$ are related iff $k = k'$ and either:
  - k is secret and $(v, v') \in \varphi(k)$, or else
  - k is not secret and v and v' are related.

Intuitively, $\varphi \vdash v \sim v' : \tau$ means "under any possible attackers, the values v and v' behave equivalently and furthermore preserve the invariant that values encrypted by any secret key k are related by $\varphi(k)$." It is this invariant which makes the logical relation work at all: as is often the case in inductive proofs, requiring this extra condition helps us in proving the final goal, i.e., extensional equivalence. Note that, in the definition above, secret keys are not related even if they are identical, because if they were related, an attacker would be able to encrypt arbitrary values under the keys and break the invariance.

As for expressions, arbitrary expressions are related iff they evaluate to values that, in turn, are related under a relation environment extended with the fresh keys that were generated during evaluation. The formal definition of the logical relation is given in Figure 4. $\varphi \vdash^{\text{val}}_{s,s'} v \sim v' : \tau$ and $\varphi \vdash^{\text{exp}}_{s,s'} e \sim e' : \tau$ are logical relations for values and expressions, respectively. The sets s and s', respectively, denote the keys generated so far in the left and right hand sides.

Example 8. For the e and e' in the previous example, let $\tau = \texttt{bits[bool]} \times \texttt{bits[bool]} \times (\texttt{bits[bool]} \to \texttt{option[bool]})$. Then, $\emptyset \vdash^{\text{exp}}_{\emptyset,\emptyset} e \sim e' : \tau$. To prove this, let $t = t' = \{k\}$ and $\psi = \{k \mapsto \{(\texttt{true}, \texttt{false}), (\texttt{false}, \texttt{true})\}\}$ in the definition of $\emptyset \vdash^{\text{exp}}_{\emptyset,\emptyset} e \sim e' : \tau$.

Example 9. For the p_i in Example 2, let $\tau = \texttt{bits[int]} \times (\texttt{bits[int]} \to \texttt{option[int]})$. Then, $\emptyset \vdash^{\text{exp}}_{\emptyset,\emptyset} p_i \sim p_j : \tau$ for any i and j with $i \bmod 2 = j \bmod 2$. (Here, we define $\varphi \vdash^{\text{val}}_{s,s'} i \sim i' : \texttt{int} \iff i = i'$.) To prove this, let $t = t' = \{k\}$ and $\psi(k) = \{(i, j)\}$ in the definition of $\emptyset \vdash^{\text{exp}}_{\emptyset,\emptyset} p_i \sim p_j : \tau$.

The following theorem and corollary state that the logical relation indeed implies extensional equivalence.

Theorem 10. Let $\Gamma, \Delta \vdash e : \tau$ where $\Gamma = \{\tilde{x} \mapsto \tilde{\tau}\}$. Let furthermore $\varphi \vdash^{\text{val}}_{s,s'} \tilde{v} \sim \tilde{v}' : \tilde{\tau}$ where $dom(\varphi) \cap dom(\Delta) = \emptyset$ and $s, s' \supseteq dom(\varphi) \uplus dom(\Delta)$. Then, $\varphi \vdash^{\text{exp}}_{s,s'} [\tilde{v}/\tilde{x}]e \sim [\tilde{v}'/\tilde{x}]e : \tau$. That is, any expression is related to itself when its free variables are substituted with related values.

Corollary 11 (Soundness of Logical Relation). If $\emptyset \vdash^{\text{exp}}_{\emptyset,\emptyset} e \sim e' : \tau$, then $\vdash e \approx e' : \tau$.

6.2 Extended Logical Relation

In the basic logical relation above, a relation between values encrypted by each secret key k is given by the relation environment φ. However, φ gives up no information about the relations that should be associated with fresh keys that are still to be generated in the future. As a result, the basic logical relation technique fails to prove the equivalence of some important examples that are, in fact, equivalent: in particular, we cannot prove the security of the improved version of

$$
\begin{array}{lcl}
\varphi \vdash^{\mathrm{val}}_{s,s'} f \sim f' : \tau_1 \to \tau_2 & \iff & f = \lambda x.\, e \text{ and } f' = \lambda x.\, e' \text{ for some } x, e, e' \text{ such that} \\
 & & \varphi \uplus \psi \vdash^{\mathrm{exp}}_{s \uplus t, s' \uplus t'} [v/x]e \sim [v'/x]e' : \tau_2 \text{ for any } v, v', t, t', \psi \text{ such that} \\
 & & \varphi \uplus \psi \vdash^{\mathrm{val}}_{s \uplus t, s' \uplus t'} v \sim v' : \tau_1 \text{ with } dom(\psi) \subseteq t \cap t' \\
\varphi \vdash^{\mathrm{val}}_{s,s'} p \sim p' : \tau_1 \times \cdots \times \tau_n & \iff & p = \langle v_1, \ldots, v_n \rangle \text{ and } p' = \langle v'_1, \ldots, v'_n \rangle \text{ for some } \tilde{v}, \tilde{v}' \text{ such that} \\
 & & \varphi \vdash^{\mathrm{val}}_{s,s'} \tilde{v} \sim \tilde{v}' : \tilde{\tau} \\
\varphi \vdash^{\mathrm{val}}_{s,s'} t \sim t' : \tau_1 + \cdots + \tau_n & \iff & t = \mathtt{in}_i(v) \text{ and } t' = \mathtt{in}_i(v') \text{ for some } i, v, v' \text{ such that} \\
 & & \varphi \vdash^{\mathrm{val}}_{s,s'} v \sim v' : \tau_i \\
\varphi \vdash^{\mathrm{val}}_{s,s'} k \sim k' : \mathtt{key}[\tau] & \iff & k = k' \text{ where } k \in s \cap s' \text{ with } k \notin dom(\varphi) \\
\varphi \vdash^{\mathrm{val}}_{s,s'} c \sim c' : \mathtt{bits}[\tau] & \iff & c = \{v\}_k \text{ and } c' = \{v'\}_k \text{ for some } v, v', k \text{ such that} \\
 & & k \in dom(\varphi) \text{ with } k \in s \cap s' \text{ and } (v, v') \in \varphi(k), \text{ or} \\
 & & k \notin dom(\varphi) \text{ with } k \in s \cap s' \text{ and } \varphi \vdash^{\mathrm{val}}_{s,s'} v \sim v' : \tau \\
\varphi \vdash^{\mathrm{exp}}_{s,s'} e \sim e' : \tau & \iff & (s)e \Downarrow (s \uplus t)v \text{ and } (s')e' \Downarrow (s' \uplus t')v' \text{ for some } t, v, t', v', \psi \text{ such that} \\
 & & \varphi \uplus \psi \vdash^{\mathrm{val}}_{s \uplus t, s' \uplus t'} v \sim v' : \tau \text{ with } dom(\psi) \subseteq t \cap t'
\end{array}
$$

Figure 4. Basic Logical Relation

the Needham-Schroeder public-key protocol from Section 3.2.

For a simpler example showing where the proof technique goes wrong, consider a program $q_i = \mathtt{new}\ x\ \mathtt{in}\ \langle \lambda\_.\ \mathtt{new}\ y\ \mathtt{in}\ \{y\}_x,\ \lambda\{y'\}_x.\mathtt{Some}(\{i\}_{y'})\rangle$ for some secret integer i. Since the key x (to be precise, the key bound to the variable x) is kept secret, the key $y = y'$ is also kept secret, so i is kept secret. Therefore, q_3 and q_5, say, should be equivalent. But in order to prove this by using the basic logical relation above, we would have to give a relation between values encrypted by the key k bound to x. Since the key k' that will be bound to y is not yet determined, we cannot specify a relation like $\varphi(k) = \{(k', k')\}$. Thus, q_3 and q_5 cannot be related.

This problem can be addressed by refining the definition of the logical relation a little, i.e., parameterizing the relation environment φ with respect to sets s and s' of keys—representing the sets of keys that will have been generated at some point of interest in the future—as well as the relation environment ψ that will be in effect at that time. (The definition of "a relation environment parametrized by another relation environment" is recursive, but such entities can be constructed inductively, just as elements of a recursive type can be.) Then, in the example above, for instance, we can specify the needed relation as $\varphi^{\psi}_{s,s'}(k) = \{(k', k') \mid \psi_{t,t'}(k')\chi = \{(3,5)\} \text{ for any } t, t' \text{ and } \chi\}$. Accordingly, we extend the definition of the logical relation for ciphertext types to:

$$
\begin{array}{l}
\varphi \vdash^{\mathrm{val}}_{s,s'} c \sim c' : \mathtt{bits}[\tau] \iff \\
c = \{v\}_k \text{ and } c' = \{v'\}_k \text{ for some } v, v', k \text{ such that} \\
k \in dom(\varphi) \text{ with } k \in s \cap s' \text{ and } (v, v') \in \varphi^{\varphi}_{s,s'}(k), \text{ or} \\
k \notin dom(\varphi) \text{ with } k \in s \cap s' \text{ and } \varphi \vdash^{\mathrm{val}}_{s,s'} v \sim v' : \tau
\end{array}
$$

Interestingly, even after this extension, the propositions in Section 6.1 (and their proofs!) continue to hold *without change*—as long as we impose the condition that φ in $\varphi^{\psi}_{s,s'}(k)$ is monotonic with respect to extension of s, s', and ψ. Intuitively, the condition guarantees that values related once do not become unrelated as fresh keys are generated in the future. This is not the case if we take $\varphi^{\psi}_{s,s'}(k) = \{(k', k') \mid k' \notin s \cup s'\}$, for example. The monotonicity condition excludes such anomalies. Formally, we require that each φ satisfies

$$\varphi^{\psi}_{s,s'}(k) \subseteq \varphi^{\psi \uplus \chi}_{s \uplus t, s' \uplus t'}(k)$$

for any s, s', t and t' with $s \cap t = \emptyset$ and $s' \cap t' = \emptyset$, and for any ψ and χ with $dom(\psi) \subseteq s \cap s'$ and $dom(\chi) \subseteq t \cap t'$. We refer to this condition as "φ is monotonic."

Example 12. For the previous q_i, let $\tau = (\mathtt{unit} \to \mathtt{bits}[\mathtt{key}[\mathtt{int}]]) \times (\mathtt{bits}[\mathtt{key}[\mathtt{int}]] \to \mathtt{option}[\mathtt{bits}[\mathtt{int}]])$. Then, $\emptyset \vdash^{\mathrm{exp}}_{\emptyset,\emptyset} q_i \sim q_j : \tau$ for any i and j. To prove this, let $t = t' = \{k\}$ and

$$\psi^{\varphi}_{s,s'}(k) = \left\{(k', k') \mid \varphi^{\chi}_{t,t'}(k') = \{(i,j)\} \text{ for any } t, t' \text{ and } \chi\right\}$$

in the definition of $\emptyset \vdash^{\mathrm{exp}}_{\emptyset,\emptyset} q_i \sim q_j : \tau$. It is straightforward to check that ψ is monotonic. Hence $\vdash q_i \approx q_j : \tau$.

Example 13. Let us see how to prove the correctness of the system in Section 3.2, which is based on the improved version of the Needham-Schroeder public-key protocol, using the extended logical relation.

First, in order for the encoding NS'_i to be well-typed at all, values encrypted by the keys z_b and z_x need to be tagged. (The tags are underlined.)

$$
\begin{array}{l}
\mathtt{new}\ z_a\ \mathtt{in}\ \mathtt{new}\ z_b\ \mathtt{in}\ \mathtt{new}\ z_e\ \mathtt{in} \\
\langle \lambda x.\, \{x\}_{z_a}, \lambda x.\, \{x\}_{z_b}, z_e, \\
\ \langle B, \lambda\{\underline{\mathtt{in}_1}(\langle N_a, A\rangle)\}_{z_b}.\, \mathtt{new}\ N_b\ \mathtt{in} \\
\qquad \mathtt{Some}(\langle\{\langle N_a, N_b, B\rangle\}_{z_a},\ \lambda\{\underline{\mathtt{in}_2}(N_b)\}_{z_b}.\mathtt{Some}(\{i\}_{N_b})\rangle)\rangle, \\
\ \langle \lambda X.\, \mathtt{new}\ N_a\ \mathtt{in}\ \langle\{\underline{\mathtt{in}_1}(\langle N_a, A\rangle)\}_{z_x}, \\
\qquad\qquad \lambda\{N_a, N_x, X\}_{z_a}.\, \mathtt{Some}(\{\underline{\mathtt{in}_2}(N_x)\}_{z_x})\rangle\rangle\rangle
\end{array}
$$

Call this expression NS_i''. It can be given the type

$$(\tau_1 \to \mathtt{bits}[\tau_1]) \times (\tau_2 \to \mathtt{bits}[\tau_2]) \times \mathtt{key}[\tau_2] \times (\mathtt{nam} \times (\mathtt{bits}[\tau_2] \to \mathtt{option}[\mathtt{bits}[\tau_1] \times (\mathtt{bits}[\tau_2] \to \mathtt{option}[\mathtt{bits}[\mathtt{int}]])])) \times (\mathtt{nam} \to (\mathtt{bits}[\tau_2] \times (\mathtt{bits}[\tau_1] \to \mathtt{option}[\mathtt{bits}[\tau_2]])))$$

where nam is actually just int and

$$\begin{array}{rcl} \tau_1 & = & \mathtt{key}[\sigma] \times \mathtt{key}[\mathtt{int}] \times \mathtt{nam} \\ \tau_2 & = & \mathtt{key}[\sigma] \times \mathtt{nam} + \mathtt{key}[\mathtt{int}] \end{array}$$

for some σ. Call this type τ.

Now, NS_i'' and NS_j'' can be related (and are therefore extensionally equivalent) for any i and j by letting $t = t' = \{k_a, k_b, k_e\}$ and

$$\begin{array}{rcl} \psi^{\varphi}_{s,s'}(k_a) & = & \{(v,v') \mid \varphi \vdash^{\mathrm{val}}_{s,s'} v \sim v' : \tau_1\} \\ & \cup & \{(\langle N_a, N_b, B\rangle, \langle N_a, N_b, B\rangle) \mid \varphi^{\chi}_{t,t'}(N_a) = r \text{ and } \varphi^{\chi}_{t,t'}(N_b) = \{i,j\} \text{ for any } t, t' \text{ and } \chi\} \\ \psi^{\varphi}_{s,s'}(k_b) & = & \{(v,v') \mid \varphi \vdash^{\mathrm{val}}_{s,s'} v \sim v' : \tau_2\} \\ & \cup & \{(\mathtt{in}_1(\langle N_a, A\rangle), \mathtt{in}_1(\langle N_a, A\rangle)) \mid \varphi^{\chi}_{t,t'}(N_a) = r \text{ for any } t, t' \text{ and } \chi\} \\ & \cup & \{(\mathtt{in}_2(N_b), \mathtt{in}_2(N_b)) \mid \varphi^{\chi}_{t,t'}(N_b) = \{i,j\} \text{ for any } t, t' \text{ and } \chi\} \end{array}$$

for some r in the definition of $\emptyset \vdash^{\mathrm{exp}}_{\emptyset,\emptyset} NS_i'' \sim NS_j'' : \tau$.

It is straightforward, by the way, to check that $Good(NS_i'')$ evaluates to $\mathtt{Some}(\{i\}_{N_b})$ for some fresh N_b. So this system is indeed both safe (from attacks that can be modeled in our setting) and sound.

6.3 Another Extended Logical Relation

Another way of extending the logical relation is to let a relation environment φ map a *pair* of secret keys—rather than one secret key—to a relation between values encrypted by those keys. Consider, for example, the following two expressions.

$$\begin{array}{rcl} e & = & \mathtt{new}\ x\ \mathtt{in}\ \langle \{1\}_x, \{2\}_x, \\ & & \quad \lambda z.\, \mathtt{let}\ \{i\}_x = z\ \mathtt{in}\ \mathtt{Some}(x \bmod 2)\ \mathtt{else}\ \mathtt{None}\rangle \\ e' & = & \mathtt{new}\ x\ \mathtt{in}\ \mathtt{new}\ y\ \mathtt{in}\ \langle \{3\}_x, \{4\}_y, \\ & & \quad \lambda z.\, \mathtt{let}\ \{i\}_x = z\ \mathtt{in}\ \mathtt{Some}(i \bmod 2)\ \mathtt{else} \\ & & \qquad \mathtt{let}\ \{j\}_y = z\ \mathtt{in}\ \mathtt{Some}(j \bmod 2)\ \mathtt{else}\ \mathtt{None}\rangle \end{array}$$

They should be extensionally equivalent because, in both expressions, the keys x and y are kept secret, and therefore the only way to use the first and second elements of the tuples is to apply the third elements, which return the same value. However, this extensional equivalence cannot be proved by using either of the logical relations above, because the second elements are encrypted by different keys.

This problem can be solved by letting a relation environment φ take a *pair* of secret keys, like $\varphi(k_x, k_x) = \{(1,3)\}$ and $\varphi(k_x, k_y) = \{(2,4)\}$ for example, and extending the definition of the logical relation accordingly, letting

$$\begin{array}{l} \varphi \vdash^{\mathrm{val}}_{s,s'} c \sim c' : \mathtt{bits}[\tau] \iff \\ c = \{v\}_k \text{ and } c' = \{v'\}_{k'} \text{ for some } v, v', k, k' \text{ such that} \\ (k,k') \in dom(\varphi) \text{ with } (k,k') \in s \times s' \text{ and } (v,v') \in \varphi(k), \text{ or} \\ (k,k') \notin dom(\varphi) \text{ with } (k,k') \in s \times s' \text{ and } \varphi \vdash^{\mathrm{val}}_{s,s'} v \sim v' : \tau \end{array}$$

$$\begin{array}{l} \varphi \vdash^{\mathrm{val}}_{s,s'} k \sim k' : \mathtt{key}[\tau] \iff \\ k = k' \text{ where } (k,k) \in s \times s' \text{ with} \\ (k,k'') \notin dom(\varphi) \text{ and } (k'',k) \notin dom(\varphi) \text{ for any } k'' \end{array}$$

and so forth. Again, it is straightforward to adapt the results in Section 6.1 for this extension. (It may seem somewhat surprising that the results in Section 6.1 are so easily adapted to different definitions of logical relations. This stems from the fact that the proofs of the propositions do not actually depend on the internal structure of relation environments.)

7 Related Work

Numerous approaches to formal verification of security protocols have been explored in the literature [11, 13, 15, 16, etc.]. Of these, the spi-calculus [3] is one of the most powerful; it comes equipped with useful techniques such as bisimulation [2, 6] for proving behavioral equivalences and static typing for guaranteeing secrecy [1] and authenticity [10]. We are not in a position yet to claim that our approach is superior to the spi-calculus (or any other existing approach); rather, our goal has been to demonstrate that standard techniques for reasoning about type abstraction *can* be adapted to the task of reasoning about encryption, in particular about security protocols. For this study, λ-calculus offers a better starting point than name-passing process calculi, where relational parametricity does not actually work very well because of aliasing [21]. Of course, the cost of this choice is that we depend on the ability of the λ-calculus to encode communication and concurrency by function application and interleaving. Since this encoding is not fully abstract (processes are linear by default while functions are not), a process that is actually secure is not always encoded as a secure λ-term. Any attacks that we discover for the encoded term must be reality-checked against the original process (cf. the false attack on the program using the ffgg protocol in Section 3). However, if the encoding of a process *can* be proved secure, then the process itself is also secure (cf. the proof of the secrecy property of the program using the improved Needham-Schroeder public-key protocol in Section 3).

Formalizing and proving secrecy as non-interference—i.e., equivalence between instances of a program with dif-

ferent secret values—has been a popular approach both in the security community and in the programming language community. Non-interference reasoning in protocol verification can be found in [9, 24, 26], among others.

Since the cryptographic λ-calculus has a key generation primitive, we must be able to reason about generative names. We adopted Pitts and Stark's work on λ-calculus with name generation [25] in formulating both the semantics in Section 4 and the logical relation in Section 6.1.

Encryption is similar to type abstraction in that both restrict access to secrets (the former dynamically obfuscates their values, while the latter statically hide their types). To define the logical relation in Section 6.1, we also referred to the logical relation for the polymorphic λ-calculus, also known as the theory of relational parametricity [23]. While the latter assigns each type variable a relation between values implementing the abstract type, the former assigns each secret key a relation between values encrypted by the key.

There also exist many proposals for using techniques in programming languages—in particular, static typing—to guarantee security of programs. For example, Heintze and Riecke [12] proposed a typed λ-calculus with information flow control, and proved a non-interference property—that a value of high security does not leak to any context of low security—using a logical relation. Most of those approaches aim to statically exclude attackers coming into a system, rather than to dynamically protect a program from attackers outside the system. (An exception is the work cited above on static typing for secrecy and authenticity in spi-calculus.)

Lillibridge and Harper [personal communication, July 2000] have independently developed a *typed seal calculus* that is closely related to our cryptographic λ-calculus. Their work mainly focuses on encoding *sealing* [18] primitives in terms of other mechanisms such as exceptions and references (and vice versa), rather than establishing techniques for reasoning about secrecy properties of programs using sealing.

8 Future Work

Recursive Functions and Recursive Types It can be shown (from Theorem 10 and the definition of $\varphi \vdash^{\mathrm{exp}}_{s,s'} e \sim e' : \tau$) that under our simple type system, evaluation of a well-typed expression *always* terminates. Therefore, recursive functions cannot be written. Indeed, introducing recursive functions breaks the soundness proof of the logical relations. Also, introducing recursive types breaks the well-definedness of the logical relations. We expect that these limitations can be removed by incorporating the theory of logical relation for λ-calculus with recursive functions and/or recursive types (e.g., [5, 7]).

State and Linearity Although real programs often have some kind of state or linearity (in the sense of linear logic), our framework does not take them into account. Thus, it cannot prove the security of a program depending on them.

For example, consider an expression $p_i = \mathtt{new}\ z\ \mathtt{in}\ \lambda x.\ \mathtt{let}\ \{\_\}_z = x\ \mathtt{in}\ \mathtt{in}_1(i)\ \mathtt{else}\ \mathtt{in}_2(z)$ for some secret integer i. Although this program leaks the secret integer i under the attacker $f = \lambda p.\ \mathtt{let}\ \mathtt{in}_2(z) = p\{0\}_k\ \mathtt{in}\ \mathtt{let}\ \mathtt{in}_1(i) = p\{0\}_z\ \mathtt{in}\ \mathtt{Some}(i)$, it is actually secure if the function $\lambda x.\ \ldots$ is linear (i.e., applied only once). A similar example can also be given by using an ML-like reference cell.

Although we have not yet come across a realistic program whose security depends on its state or linearity in a crucial manner (maybe because such a "dangerous" design is avoided *a priori* by engineering practice?), we expect that this issue can be addressed, too, by incorporating the theory of logical relation for λ-calculus with state or linearity [4, 22].

Type Abstraction via Encryption Although we focused on adapting the theory of type abstraction into encryption, it is also interesting to think of using the technique of encryption for type abstraction. Specifically, it may be possible to implement type abstraction by means of encryption, in order to protect secrets not only from well-typed programs, but also from arbitrary attackers—in other words, to combine polymorphism with dynamic typing without losing the abstraction. That would enable us to write programs in a high-level language using type abstraction and translate them into a lower-level code using encryption. Then, the problem is whether and how such translation is possible, preserving the abstraction. In an earlier version of this work, we suggested one possibility of such translation [20, Section 4] but proved nothing about it. The results in the present paper—in particular, the logical relations in Section 6—would help better understanding of this issue.

Acknowledgements

We would like to thank many people including Martín Abadi, Naoki Kobayashi, the members of Akinori Yonezawa's group in the University of Tokyo, the members of the Logic and Computation Seminar—especially Andre Scedrov—and the Programming Language Club in the University of Pennsylvania, and anonymous reviewers for useful advice on earlier versions of this work.

This work was supported by the National Science Foundation under NSF Career grant CCR-9701826 and by the Japan Society for the Promotion of Science.

References

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[2] M. Abadi and A. D. Gordon. A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, 5:267–303, 1998.

[3] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[4] G. M. Bierman, A. M. Pitts, and C. V. Russo. Operational properties of Lily, a polymorphic linear lambda calculus with recursion. In *Higher Order Operational Techniques in Semantics*, volume 41 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2000.

[5] L. Birkedal and R. Harper. Relational interpretations of recursive types in an operational setting. *Information and Computation*, 155(1–2):3–63, 1999.

[6] M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes, 1999. Available at `ftp://rap.dsi.unifi.it/pub/papers/spi.ps.gz`. An extended and revised version of the paper that appeared in *14th Annual IEEE Symposium on Logic in Computer Science*, pp. 157–166.

[7] K. Crary and R. Harper. Syntactic logical relations over polymorphic and recursive types. Draft, 2000.

[8] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[9] A. Durante, R. Focardi, and R. Gorrieri. CVS: A compiler for the analysis of cryptographic protocols. In *12th IEEE Computer Security Foundations Workshop*, pages 203–212, 1999.

[10] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop*, 2001. To appear.

[11] N. Heintze and E. Clarke, editors. *Workshop on Formal Methods and Security Protocols*, 1999. `http://cm.bell-labs.com/cm/cs/who/nch/fmsp99/`.

[12] N. Heintze and J. G. Riecke. The slam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

[13] IEEE computer security foundations workshop. `http://www2.csl.sri.com/csfw/index.html`.

[14] G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[15] C. Meadows. Formal verification of cryptographic protocols: A survey. In *Advances in Cryptology – Asiacrypt '94*, volume 917 of *Lecture Notes in Computer Science*, pages 133–150. Springer-Verlag, 1995.

[16] C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In *DARPA Information Survivability Conference and Exposition*, pages 237–250. IEEE Computer Society, 2000.

[17] J. K. Millen. A necessarily parallel attack. In *Workshop on Formal Methods and Security Protocols*, 1999. Available at `http://www.cs.bell-labs.com/who/nch/fmsp99/program.html`.

[18] J. H. Morris Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.

[19] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[20] B. Pierce and E. Sumii. Relating cryptography and polymorphism, 2000. Manuscript. Available at `http://www.yl.is.s.u-tokyo.ac.jp/~sumii/pub/infohide.ps.gz`.

[21] B. C. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–586, 2000.

[22] A. Pitts and I. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998.

[23] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*, pages 513–523, 1983.

[24] P. Ryan and S. Schneider. Process algebra and noninterference. In *12th IEEE Computer Security Foundations Workshop*, pages 214–227, 1999.

[25] I. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, 1994. Available at `http://www.dcs.ed.ac.uk/home/stark/publications/thesis.html`.

[26] D. Volpano. Formalization and proof of secrecy properties. In *12th IEEE Computer Security Foundations Workshop*, pages 92–95, 1999.