# Logical Relations for Encryption*

Eijiro Sumii†
University of Tokyo
sumii@yl.is.s.u-tokyo.ac.jp

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 $\lambda$-calculus. We investigate the application of these tools in the security domain by defining a *cryptographic $\lambda$-calculus*—an extension of the standard simply typed $\lambda$-calculus with primitives for encryption, decryption, and key generation—and introducing syntactic logical relations (in the style of Pitts and Birkedal-Harper) for this calculus that can be used to prove behavioral equivalences between programs that use 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* [33] and its accompanying *logical relations* proof method—in the domain of security protocols. (By logical relations, we mean *syntactic* ones [6, 28]—that is, relations over syntactic expressions in a term model—throughout this paper.)

We begin by defining a *cryptographic $\lambda$-calculus*, an extension of the ordinary simply typed $\lambda$-calculus with primitives for encryption, decryption, and key generation. One can imagine a large family of different cryptographic $\lambda$-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. (Some issues

*An extended abstract appeared in *Proceedings of the 14th IEEE Computer Security Foundations Workshop, Keltic Lodge, Cape Breton, Nova Scotia, Canada, June 11-13, 2001.*

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

raised by introducing static types into the calculus will be discussed in Section 5 along with the details of the type system.) We now proceed in three steps:

1. We show how some simple security protocols can be modeled by expressions in the cryptographic $\lambda$-calculus. The essence of the encoding lies in representing each principal by (a pair of) the next message value it sends and/or a function representing its response to the next message it receives. Our main example is the Needham-Schroeder public-key protocol [24]. 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 [17].

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 $\lambda$-calculus, we can use the same language to reason about the attacker and the program.

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 observable respect. In particular, in its original form in the polymorphic $\lambda$-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 $\lambda$-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 explicitly quantifying 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 $i$ `mod` 2. In the standard informal notation, this protocol can be written as:

$$\begin{array}{lll} 1. \quad A \to B & : & \{i\}_z \\ 2. \quad B \to A & : & i \texttt{ mod } 2 \end{array}$$

In the cryptographic $\lambda$-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 $\lambda\{x\}_z.\, x$ `mod` 2 representing the principal B, which publishes $x$ `mod` 2 on receiving an integer $x$ encrypted by $z$. Let $p_i$ be an instance of the program with the secret integer being $i$.

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

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 "possible" attackers [8], who are 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 is 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 every 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 if and only if they are equal.
- Two pairs are related if and only if their elements are related.
- Two functions are related if and only if they return related values when applied to related arguments.
- Two values encrypted by a secret key $k$ are related if and only if they are related by $\varphi(k)$. Here, $\varphi$ is a *relation environment* mapping each secret key to the relation between values encrypted by that key. Intuitively, it specifies what values are encrypted by each secret key (e.g., 3 and 5 are encrypted by $z$ in the example above) and thereby keeps track of the correspondence between ciphertexts (e.g., between $\{3\}_z$ and $\{5\}_z$).

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 $\varphi$ such that the logical relation based on $\varphi$ 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 $\varphi$.
- 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.

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

Figure 1: Syntax

Thus, $p_3$ and $p_5$ are guaranteed to be behaviorally equivalent. In this way, the logical relation allows us 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 method of proving behavioral equivalence (which implies secrecy) of programs that use encryption.

The rest of this paper is structured as follows. Section 2 presents the syntax and intuitive semantics of the cryptographic $\lambda$-calculus and Section 3 demonstrates its use in encoding cryptographic protocols through examples. Section 4 shows its operational semantics, Section 5 gives a simple type system—a prerequisite for formalizing the logical relations—and Section 6 defines our logical relation and its variants to cope with more complex cases (such as "keys encrypting other keys" in security protocols). Section 7 discusses related work, and Section 8 future work.

## 2 Syntax and Intuitive Semantics

The cryptographic $\lambda$-calculus extends a standard $\lambda$-calculus with keys, fresh key generation, encryption, and decryption primitives. The formal syntax of the calculus is given in Figure 1. $\mathit{int_op}_n(e_1, \ldots, e_n)$ is the syntax of primitive operators for integer arithmetics like $\mathit{plus}_2(e_1, e_2)$. (The subscript denotes the arity of each operator. This is necessary for the type system in Section 5.) We adopt infix notations such as $e_1 + e_2$ for binary operations. We use a tuple with no elements (i.e., $n = 0$) to represent a dummy value, written $\langle\rangle$. $\mathtt{in}_i(\,)$ denotes the $i$-th injection ("tagging") into a disjoint sum.

A key $k$ is an element of the countably infinite set of keys $K$. The key generation form $\mathtt{new}\ x\ \mathtt{in}\ e$ 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 $\mathtt{new}\ x\ \mathtt{in}\ \mathtt{new}\ y\ \mathtt{in}\ \langle x, y\rangle$ generates two fresh keys and yields a 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 $\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4$ 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 $\mathtt{let}\ \{x\}_{k'} = \{1\}_k\ \mathtt{in}\ x + 2\ \mathtt{else}\ 0$ 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, $\mathtt{let}\ \{x\}_k = \{1\}_k\ \mathtt{in}\ x + 2\ \mathtt{else}\ 0$ gives 3 because the decryption succeeds.

Several abbreviations are used in examples. We write $\mathtt{true}$ for $\mathtt{in}_1(\langle\rangle)$, $\mathtt{false}$ for $\mathtt{in}_2(\langle\rangle)$, and $\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$ for $\mathtt{case}\ e\ \mathtt{of}\ \mathtt{in}_1(_) \Rightarrow e_1 \,[\!]\, \mathtt{in}_2(_) \Rightarrow e_2$, respectively, to represent booleans as a disjoint sum of dummy values. We use option values $\mathtt{Some}(e)$, abbreviating $\mathtt{in}_1(e)$, and $\mathtt{None}$, abbreviating $\mathtt{in}_2(\langle\rangle)$, to represent the return values of functions that may or may not actually return a meaningful value because of errors such as decryption failure. Finally, we use the pattern matching function syntax $\lambda\{x\}_{e_1}.\, e_2$ to abbreviate $\lambda y.\, \mathtt{let}\ \{x\}_{e_1} = y\ \mathtt{in}\ e_2\ \mathtt{else}\ \mathtt{None}$ (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 ciphertext not encrypted by $k$.

**Example 1.** The expression $\lambda\{x\}_k.\,\texttt{Some}(x\ \texttt{mod}\ 2)$, which is an abbreviation for $\lambda y.\,\texttt{let}\ \{x\}_k = y\ \texttt{in}\ \texttt{in}_1(x\ \texttt{mod}\ 2)\ \texttt{else}\ \texttt{in}_2(\langle\rangle)$, 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 $i$ and any $k' \neq k$.

**Example 2.** Let $p_i = \texttt{new}\ z\ \texttt{in}\ \langle\{i\}_z, \lambda\{x\}_z.\,\texttt{Some}(x\ \texttt{mod}\ 2)\rangle$ and $f = \lambda p.\,\#_2(p)\#_1(p)$. Then, $f(p_i)$ gives $\texttt{Some}(i\ \texttt{mod}\ 2)$. This can be seen as a run of an encoding of the following simple system, in which 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. 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 approximate 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 and used as encryption and decryption keys.

This encoding is somewhat tricky and not quite faithful to the real world: not only we are again assuming perfect encryption and taking no account of any algebraic or probabilistic properties of individual cryptography, but also it would be silly, in reality, to give an attacker the code of a function that includes any secret key, which a real attacker could presumably discover by disassembling the function. In addition, our encoding limits the capability of attackers: for instance, they cannot test equality of public keys. Nevertheless, it suffices for our goal in this paper of giving an account of a few major "benchmark" examples. (In situations where this encoding does *not* suffice, we could try to reinforce it: for instance, in order to reason about a protocol in which equality of public keys is an essential issue, we could give an attacker a public key like $\lambda x.\,\{x\}_k$ along with a function like $\lambda e.\,\texttt{let}\ \{_\}_k = e\langle\rangle\ \texttt{in true else false}$ to test whether another (encryption) key is equal to the previous one.)

# 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 $\lambda$-calculus has no built-in primitives for concurrency or communication, it can emulate a concurrent, communicating system by a reasonably straightforward encoding (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*).

More precise definitions of "correct" and "wrong" depend on the intention of each specific protocol, as we shall see below.

### 3.1 Encoding the Needham-Schroeder Public-Key Protocol

Consider the following system using the Needham-Schroeder public-key protocol [24] 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 key $N_B$, 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.

$$\begin{array}{lcl} 1. & B \to A & : & B \\ 2. & A \to B & : & \{N_A, A\}_{k_B} \\ 3. & B \to A & : & \{N_A, N_B\}_{k_A} \\ 4. & A \to B & : & \{N_B\}_{k_B} \\ 5. & B \to A & : & \{i\}_{N_B} \end{array}$$

Let us encode this system as an expression of the cryptographic $\lambda$-calculus. (The result is necessarily somewhat complex, because several actions implicitly assumed in the informal definition above—such as checks on the identity 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 a key for E. (This key for E is necessary for encoding an attack to the protocol as we shall see soon, in which E is one of the clients of A and therefore A must know the key for E.)

$$\begin{array}{l} \texttt{new}\ z_A\ \texttt{in}\ \texttt{new}\ z_B\ \texttt{in}\ \texttt{new}\ z_E\ \texttt{in} \\ \langle \lambda x.\, \{x\}_{z_A}, \lambda x.\, \{x\}_{z_B}, z_E, \ldots, \ldots \rangle \end{array}$$

Let us now encode B as the fourth element of the tuple. B starts by publishing its own name $B$, which we encode as a pair of $B$ and a function representing $B$'s next action, which we'll come back to in a moment. (We assume that names of principles are just integers—like IP addresses, for example—for the sake of simplicity.) The difference from the previous expression is underlined.

$$\begin{array}{l} \texttt{new}\ z_A\ \texttt{in}\ \texttt{new}\ z_B\ \texttt{in}\ \texttt{new}\ z_E\ \texttt{in} \\ \langle \lambda x.\, \{x\}_{z_A}, \lambda x.\, \{x\}_{z_B}, z_E, \underline{\langle B, \ldots \rangle}, \ldots \rangle \end{array}$$

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.

$$\begin{array}{l} \texttt{new}\ z_A\ \texttt{in}\ \texttt{new}\ z_B\ \texttt{in}\ \texttt{new}\ z_E\ \texttt{in} \\ \langle \lambda x.\, \{x\}_{z_A}, \lambda x.\, \{x\}_{z_B}, z_E, \langle B, \ldots \rangle, \\ \underline{\lambda X.\, \texttt{new}\ N_A\ \texttt{in}\ \langle \{\langle N_A, A \rangle\}_{z_x}, \ldots \rangle} \rangle \end{array}$$

Here, $z_x$ abbreviates $\texttt{if } X = A \texttt{ then } z_A \texttt{ else if } X = B \texttt{ then } z_B \texttt{ else } z_E$. The next action of B is to receive the pair of $N_A$ and a name $A'$ encrypted by $z_B$, check that $A' = A$, encrypt the pair of $N_A$ and a freshly generated nonce $N_B$ with $z_A$, and publish it:

$$\begin{array}{l}\texttt{new } z_A \texttt{ in new } z_B \texttt{ in new } z_E \texttt{ in}\\ \langle \lambda x.\, \{x\}_{z_A}, \lambda x.\, \{x\}_{z_B}, z_E,\\ \ \langle B, \underline{\lambda\{\langle N_A, A\rangle\}_{z_B}.\texttt{new } N_B \texttt{ in } \texttt{Some}(\langle\{\langle N_A, N_B\rangle\}_{z_A}, \ldots\rangle)}\rangle,\\ \ \lambda X.\texttt{new } N_A \texttt{ in } \langle\{\langle N_A, A\rangle\}_{z_x}, \ldots\rangle\rangle\end{array}$$

Here, $\lambda\{\langle N_A, A\rangle\}_{z_B}. \ldots$ abbreviates a "pattern matching" expression $\lambda y.\texttt{let } \{p\}_{z_B} = y \texttt{ in } (\texttt{if } \#_2(p) = A \texttt{ then } (\lambda N_A. \ldots)\#_1(p) \texttt{ else None}) \texttt{ else None}$. We will use similar abbreviations throughout this paper. Next, A receives the pair of a nonce $N'_A$ and $N_B$, checks that $N'_A = N_A$, encrypts $N_B$ with $z_B$, and publishes it:

$$\begin{array}{l}\texttt{new } z_A \texttt{ in new } z_B \texttt{ in new } z_E \texttt{ in}\\ \langle \lambda x.\, \{x\}_{z_A}, \lambda x.\, \{x\}_{z_B}, z_E,\\ \ \langle B, \lambda\{\langle N_A, A\rangle\}_{z_B}.\texttt{new } N_B \texttt{ in } \texttt{Some}(\langle\{\langle N_A, N_B\rangle\}_{z_A}, \ldots\rangle)\rangle,\\ \ \langle \lambda X.\texttt{new } N_A \texttt{ in } \langle\{\langle N_A, A\rangle\}_{z_x},\\ \qquad\qquad \underline{\lambda\{N_A, N_x\}_{z_A}.\texttt{Some}(\{N_x\}_{z_B})}\rangle\rangle\end{array}$$

Last, B receives a nonce $N'_B$ encrypted by $z_B$, checks that $N'_B = N_B$, encrypts $i$ with $N_B$, and publishes it:

$$\begin{array}{l}\texttt{new } z_A \texttt{ in new } z_B \texttt{ in new } z_E \texttt{ in}\\ \langle \lambda x.\, \{x\}_{z_A}, \lambda x.\, \{x\}_{z_B}, z_E,\\ \ \langle B, \lambda\{\langle N_A, A\rangle\}_{z_B}.\texttt{new } N_B \texttt{ in}\\ \qquad \texttt{Some}(\langle\{\langle N_A, N_B\rangle\}_{z_A}, \underline{\lambda\{N_B\}_{z_B}.\texttt{Some}(\{i\}_{N_B})}\rangle)\rangle,\\ \ \langle \lambda X.\texttt{new } N_A \texttt{ in } \langle\{\langle N_A, A\rangle\}_{z_x},\\ \qquad\qquad \lambda\{N_A, N_x\}_{z_A}.\texttt{Some}(\{N_x\}_{z_x})\rangle\rangle\end{array}$$

Let $NS_i$ be the expression above.

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.

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

Indeed, $\mathit{Good}(NS_i)$ 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 while interacting with B [17]. The attack goes as follows. (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 $N_A$, pairs it with $A$, encrypts it with E's public key, and sends it to E. (2) E encrypts the pair of $N_A$ and $A$ with B's public key and sends it to B, pretending to be A. (3,3′) 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 E's public key and sends it to E. (4) E encrypts $N_B$ with B's public key and sends it to B, pretending to be A. (5) B encrypts $i$ with $N_B$ and sends it to A, but E intercepts and decrypts it.

$$\begin{array}{lll} 1. & B \to E(A) & : \; B \\ 1'. & E \to A & : \; E \\ 2'. & A \to E & : \; \{N_A, A\}_{k_E} \\ 2. & E(A) \to B & : \; \{N_A, A\}_{k_B} \\ 3, 3'. & B \to A & : \; \{N_A, N_B\}_{k_A} \\ 4'. & A \to E & : \; \{N_B\}_{k_E} \\ 4. & E(A) \to B & : \; \{N_B\}_{k_B} \\ 5. & B \to E(A) & : \; \{i\}_{N_B} \end{array}$$

This attack can be expressed in the cryptographic $\lambda$-calculus by the following function *Evil*.

$$\begin{array}{l} \lambda p.\, \texttt{let}\ \langle B, c_B \rangle = \#_4(p)\ \texttt{in} \\ \quad \texttt{let}\ \langle \{\langle N_A, A\rangle\}_{\#_3(p)}, c_A\rangle = \#_5(p)E\ \texttt{in} \\ \quad \texttt{let}\ \texttt{Some}(\langle m, c'_B\rangle) = c_B(\#_2(p)\langle N_A, A\rangle)\ \texttt{in} \\ \quad \texttt{let}\ \texttt{Some}(\{N_B\}_{\#_3(p)}) = c_A m\ \texttt{in} \\ \quad \texttt{let}\ \texttt{Some}(\{i\}_{N_B}) = c'_B(\#_2(p)N_B)\ \texttt{in} \\ \quad \texttt{Some}(i) \end{array}$$

$\mathit{Evil}(\mathit{NS}_i)$ indeed evaluates to $\texttt{Some}(i)$, which leaks the secret. In other words, if $i \neq j$, then $\mathit{Evil}(\mathit{NS}_i)$ and $\mathit{Evil}(\mathit{NS}_j)$ evaluate to different observable values, which shows that $\mathit{NS}_i$ and $\mathit{NS}_j$ are not behaviorally equivalent.

Of course, the example in this section is not the only use of the Needham-Schroeder protocol. Some uses are easy to reason about within our framework while others are not: e.g., it is straightforward to extend the example above with other clients besides just B (it suffices to duplicate the encoding of B, just replacing the name $B$ with another), but changing the constant integer $i$ to non-constant data like $\texttt{new}\ x\ \texttt{in}\ x$ leads to a challenge in even *defining* how to state the secrecy of such data. Indeed, this is the reason why we introduced the 5th message $\{i\}_{N_B}$ into the protocol—so that we can state the secrecy of $N_B$ via the secrecy of $i$.

### 3.2 Encoding 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 [17]. (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 key $N_B$, 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.

$$\begin{array}{lll} 1. & B \to A & : \; B \\ 2. & A \to B & : \; \{N_A, A\}_{k_B} \\ 3. & B \to A & : \; \{N_A, N_B, \underline{B}\}_{k_A} \\ 4. & A \to B & : \; \{N_B\}_{k_B} \\ 5. & B \to A & : \; \{i\}_{N_B} \end{array}$$

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

$$
\begin{array}{l}
\texttt{new}\ z_A\ \texttt{in}\ \texttt{new}\ z_B\ \texttt{in}\ \texttt{new}\ z_E\ \texttt{in} \\
\langle \lambda x.\, \{x\}_{z_A}, \lambda x.\, \{x\}_{z_B}, z_E, \\
\ \langle B, \lambda \{\langle N_A, A\rangle\}_{z_B}.\, \texttt{new}\ N_B\ \texttt{in} \\
\qquad \texttt{Some}(\langle \{\langle N_A, N_B, \underline{B}\rangle\}_{z_A},\ \lambda \{N_B\}_{z_B}.\, \texttt{Some}(\{i\}_{N_B})\rangle)\rangle, \\
\ \langle \lambda X.\, \texttt{new}\ N_A\ \texttt{in}\ \langle \{\langle N_A, A\rangle\}_{z_x}, \\
\qquad\qquad\qquad \lambda \{N_A, N_x, \underline{X}\}_{z_A}.\, \texttt{Some}(\{N_x\}_{z_x})\rangle\rangle\rangle
\end{array}
$$

Let $NS'_i$ be the expression above.

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

$$
\begin{array}{l}
\lambda p.\, \texttt{let}\ \langle B, c_B\rangle = \#_4(p)\ \texttt{in} \\
\quad \texttt{let}\ \langle \{\langle N_A, A\rangle\}_{\#_3(p)}, c_A\rangle = \#_5(p)E\ \texttt{in} \\
\quad \texttt{let}\ \texttt{Some}(\langle m, c'_B\rangle) = c_B(\#_2(p)\langle N_A, A\rangle)\ \texttt{in} \\
\quad \texttt{let}\ \texttt{Some}(\{N_B\}_{\#_3(p)}) = c_A m\ \texttt{in} \\
\quad \texttt{let}\ \texttt{Some}(\{i\}_{N_B}) = c'_B(\#_2(p)N_B)\ \texttt{in} \\
\quad \texttt{Some}(i)
\end{array}
$$

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, $\mathit{Evil}(NS'_i)$ reduces to *None* for any $i$ and fails to leak the secret. In Section 6, we formally prove this secrecy property with respect to all possible attackers using our logical relation.

### 3.3 Encoding the ffgg Protocol

The ffgg protocol [20] is an artificial protocol with an intentional flaw. It tries to communicate a secret name $M$ in a rather strange manner. The point is that the protocol *does* keep the name secret as long as just one process runs for each principal, but fails to keep the secret *only* when more than one processes run for one of the principals! Although the cryptographic $\lambda$-calculus is sequential, it is actually expressive enough to encode this so-called "necessarily parallel attack" by interleaving.

To see this, let us encode the following exchange between 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 whether this $N_2$ is equal to the previous one which B already knows, and just lets $x$ be the second element of the tuple and $y$ be the third. (4) B tuples $x$, $y$ and $N_1$, encrypts them with $k$, and sends them to A with $N_1$ and $x$.

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

This system can be encoded as the following expression $\mathit{ffgg}_M$.

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

The attack on 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 a parallel copy B′ of 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.

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

This attack can be encoded as the following function, which takes the expression above as a parameter $p$.

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

This function indeed reveals the secret value $M$ in the expression $\mathit{ffgg}_M$ above. Note that the function representing the principal B did not have to be replicated explicitly, because functions in $\lambda$-calculus can be applied any number of times. In this way, our framework can express the so-called "necessarily parallel attack" without any extra treatment.

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.

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

$$
\frac{}{(s)i \Downarrow (s)i} \qquad \frac{(s_0)e_1 \Downarrow (s_1)i_1 \quad \ldots \quad (s_{n-1})e_n \Downarrow (s_n)i_n \quad \mathit{int_op}_n(i_1, \ldots, i_n) = j}{(s_0)\mathit{int_op}_n(e_1, \ldots, e_n) \Downarrow (s_n)j}
$$

$$
\frac{}{(s)\lambda x.\, e \Downarrow (s)\lambda x.\, e} \qquad \frac{(s)e_1 \Downarrow (s_1)\lambda x.\, e \quad (s_1)e_2 \Downarrow (s_2)v \quad (s_2)[v/x]e \Downarrow V}{(s)e_1 e_2 \Downarrow V}
$$

$$
\frac{(s_0)e_1 \Downarrow (s_1)v_1 \quad \ldots \quad (s_{n-1})e_n \Downarrow (s_n)v_n}{(s_0)\langle e_1, \ldots, e_n \rangle \Downarrow (s_n)\langle v_1, \ldots, v_n \rangle} \qquad \frac{(s)e \Downarrow (s')\langle \ldots, v_i, \ldots \rangle}{(s)\#_i(e) \Downarrow (s')v_i}
$$

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

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

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

$$
\frac{(s_0)e_1 \Downarrow (s_1)i_1 \quad \ldots \quad (s_{j-1})e_j \Downarrow \mathit{Error}}{(s_0)\mathit{int_op}_n(e_1, \ldots, e_n) \Downarrow \mathit{Error}} \qquad \frac{\begin{array}{c}(s_0)e_1 \Downarrow (s_1)v_1 \quad \ldots \quad (s_{n-1})e_n \Downarrow (s_n)v_n \\ v_i \text{ is not of the form } j \text{ for some } 1 \leq i \leq n\end{array}}{(s_0)\mathit{int_op}_n(e_1, \ldots, e_n) \Downarrow \mathit{Error}}
$$

$$
\frac{(s)e_1 \Downarrow \mathit{Error}}{(s)e_1 e_2 \Downarrow \mathit{Error}} \qquad \frac{(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow \mathit{Error}}{(s)e_1 e_2 \Downarrow \mathit{Error}} \qquad \frac{\begin{array}{c}(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow (s_2)v_2 \\ v_1 \text{ is not of the form } \lambda x.\, e\end{array}}{(s)e_1 e_2 \Downarrow \mathit{Error}}
$$

$$
\frac{(s_0)e_1 \Downarrow (s_1)v_1 \quad \ldots \quad (s_{i-1})e_i \Downarrow \mathit{Error}}{(s_0)\langle e_1, \ldots, e_n \rangle \Downarrow \mathit{Error}} \qquad \frac{(s)e \Downarrow \mathit{Error}}{(s)\#_i(e) \Downarrow \mathit{Error}} \qquad \frac{\begin{array}{c}(s)e \Downarrow (s')v \\ v \text{ is not of the form } \langle \ldots, v_i, \ldots \rangle\end{array}}{(s)\#_i(e) \Downarrow \mathit{Error}}
$$

$$
\frac{(s)e \Downarrow \mathit{Error}}{(s)\mathtt{in}_i(e) \Downarrow \mathit{Error}} \qquad \frac{(s)e \Downarrow \mathit{Error}}{(s)\mathtt{case}\ e\ \mathtt{of}\ \mathtt{in}_1(x_1) \Rightarrow e_1 \,[\!]\, \ldots \,[\!]\, \mathtt{in}_n(x_n) \Rightarrow e_n \Downarrow \mathit{Error}}
$$

$$
\frac{(s)e \Downarrow (s')v \quad v \text{ is not of the form } \mathtt{in}_i(v_i) \text{ for any } 1 \leq i \leq n}{(s)\mathtt{case}\ e\ \mathtt{of}\ \mathtt{in}_1(x_1) \Rightarrow e_1 \,[\!]\, \ldots \,[\!]\, \mathtt{in}_n(x_n) \Rightarrow e_n \Downarrow \mathit{Error}}
$$

$$
\frac{(s)e_1 \Downarrow \mathit{Error}}{(s)\{e_1\}_{e_2} \Downarrow \mathit{Error}} \qquad \frac{(s)e_1 \Downarrow (s_1)v \quad (s_1)e_2 \Downarrow \mathit{Error}}{(s)\{e_1\}_{e_2} \Downarrow \mathit{Error}} \qquad \frac{\begin{array}{c}(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow (s_2)v_2 \\ v_2 \text{ is not of the form } k\end{array}}{(s)\{e_1\}_{e_2} \Downarrow \mathit{Error}}
$$

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

$$
\frac{\begin{array}{c}(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow (s_2)v_2 \\ v_1 \text{ is not of the form } k\end{array}}{(s)\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 \Downarrow \mathit{Error}} \qquad \frac{\begin{array}{c}(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow (s_2)v_2 \\ v_2 \text{ is not of the form } \{v\}_k\end{array}}{(s)\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 \Downarrow \mathit{Error}}
$$

Figure 2: Semantics

# 4 Operational Semantics

In this section and the two that follow, we present the cryptographic $\lambda$-calculus, its type system, and the logical relations proof technique in detail.

The semantics of the calculus is defined by an *evaluation* relation mapping terms to results. For the ordinary $\lambda$-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 $\lambda$-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 $s \cup s'$ when $s \cap s' = \emptyset$, and undefined otherwise.) This is the only rule that increases the set of keys. In the rules 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 $\theta$ 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 $\theta'$ from $s_1'$ to $s_2'$ such that $v_2 = (\theta \uplus \theta')v_1$.

**Proof.** See the Appendix. □

**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 $\theta'$ from $s_1'$ to $s_2'$ such that $v_2 = \theta' v_1$.

**Proof.** Let $s_1 = s_2 = s$ and $\theta = id$ in Theorem 3. □

# 5 Type System

In this section, we define a simple type system for the cryptographic $\lambda$-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, in which the fundamental definition of the logical relations proceeds by induction on types.

In addition to the values found in the ordinary $\lambda$-calculus, the cryptographic $\lambda$-calculus has keys and ciphertexts. Therefore, besides the usual arrow, product, and sum types of the simply typed $\lambda$-calculus, we introduce a key type $\texttt{key}[\tau]$, whose elements are keys that can be used to

$$\tau \quad ::= \quad \texttt{int} \;\mid\; \tau_1 \to \tau_2 \;\mid\; \tau_1 \times \cdots \times \tau_n \;\mid\; \tau_1 + \cdots + \tau_n \;\mid\; \texttt{key}[\tau] \;\mid\; \texttt{bits}[\tau]$$

$$\frac{}{\Gamma, \Delta \vdash i : \texttt{int}}(\text{Const}) \quad \frac{\Gamma, \Delta \vdash e_1 : \texttt{int} \quad \ldots \quad \Gamma, \Delta \vdash e_n : \texttt{int}}{\Gamma, \Delta \vdash \mathit{int_op}_n(e_1, \ldots, e_n) : \texttt{int}}(\text{Op})$$

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

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

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

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

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

$$\frac{\Gamma, \Delta \vdash e_1 : \tau \quad \Gamma, \Delta \vdash e_2 : \texttt{key}[\tau]}{\Gamma, \Delta \vdash \{e_1\}_{e_2} : \texttt{bits}[\tau]}(\text{Enc})$$

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

Figure 3: Type System

encrypt values of type $\tau$, and a ciphertext type $\texttt{bits}[\tau]$, whose elements are ciphertexts containing a plaintext value of type $\tau$. 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, in order to guarantee type safety, we do not need to annotate both key types and ciphertext types with their underlying plaintext types. However, 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 $\Gamma$ for variables and the type environment $\Delta$ for keys, the expression $e$ has the type $\tau$, i.e., $e$ evaluates to a value of type $\tau$." 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 $\Delta$ for keys is used in the rule (Key) in the same way the type environment $\Gamma$ for variables is used in the rule (Var). For the sake of readability, we often write $\texttt{bool}$ for $\texttt{unit} + \texttt{unit}$ and $\texttt{option}[\tau]$ for $\tau + \texttt{unit}$, where $\texttt{unit}$ 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 there exist some $v$ and $\Delta'$ such that $V = (s \uplus s')v$ and $\emptyset, \Delta \uplus \Delta' \vdash v : \tau$ for $s' = dom(\Delta')$.

**Proof.** See the Appendix. □

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

**Proof.** Immediate from Theorem 5. □

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 $\lambda$-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. Moreover, there exists a specific kind of attacks—so-called "type attacks"—whose *essence* is to deceive principals into confusing values of different types.

Nevertheless, 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 type attacks, they are either (1) actually well-typed in the present type system, which does not distinguish nonces from keys, or (2) easily prevented using standard dynamic type checking techniques (see e.g. [12] for details).

# 6 Logical Relations for Encryption

Recall the family of expressions $p_i$ from Example 2:

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

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 $\lambda$-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 $\tau$," if and only if both of the following conditions hold:

- $\emptyset, \emptyset \vdash e : \tau$ and $\emptyset, \emptyset \vdash e' : \tau$
- For any $f$ with $\emptyset, \emptyset \vdash f : \tau \to \texttt{bool}$, there exist some $s$ and $s'$ such that one of the following conditions holds:
  - $(\emptyset) f e \Downarrow (s)\texttt{true}$ and $(\emptyset) f e' \Downarrow (s')\texttt{true}$
  - $(\emptyset) f e \Downarrow (s)\texttt{false}$ and $(\emptyset) f e' \Downarrow (s')\texttt{false}$

Essentially, this says that two expressions $e$ and $e'$ yield the same result under any observer function $f$. Although this 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 $\tau$ and any context $C[\,]$ of type $\texttt{bool}$ with a hole of type $\tau$. 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 $f e_0 = f e_0'$ 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 powerful enough to prove secrecy properties of realistic programs, such as (the encoding of) the improved Needham-Schroeder public-key protocol in Section 3. The others are extensions of the basic logical relation, the second for addressing the issue of "a key encrypting another key" (as in Needham-Schroeder) and the third 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 behaves 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 cannot be shown that the third elements $\lambda\{x\}_z.\, \texttt{Some}(x)$ and $\lambda\{x\}_z.\, \texttt{Some}(\texttt{not}(x))$ are "equivalent" in this context without knowing (1) the fact that (the key bound to) $z$ is kept secret throughout the whole programs and (2) the relation between values encrypted by $z$, that is, $\{\texttt{true}\}_z$ in $e$ corresponds to $\{\texttt{false}\}_z$ in $e'$ and $\{\texttt{false}\}_z$ to $\{\texttt{true}\}_z$. (Recall the correspondence between $\{3\}_z$ and $\{5\}_z$ in the example in Section 1.)

$$
\begin{array}{lcl}
\varphi \vdash^{\mathrm{val}}_{s,s'} i \sim i' : \mathtt{int} & \iff & i = i' \\
\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{ and } \mathit{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{ and } k \in s \cap s' \text{ and } k \notin \mathit{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 either} \\
 & & k \in \mathit{dom}(\varphi) \text{ and } k \in s \cap s' \text{ and } (v, v') \in \varphi(k), \text{ or else} \\
 & & k \notin \mathit{dom}(\varphi) \text{ and } 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{ and } \mathit{dom}(\psi) \subseteq t \cap t'
\end{array}
$$

Figure 4: Basic Logical Relation

Thus, we generalize $\vdash e \approx e' : \tau$ to the *logical relation* $\varphi \vdash e \sim e' : \tau$, in which the parameter $\varphi$ 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 $\varphi$, the family of relations $\varphi \vdash e \sim e' : \tau$ is defined by induction on $\tau$ as follows:

- Two functions are related if and only if they map any related arguments to related results.
- Two pairs are related if and only if their corresponding elements are related.
- Two tagged values are related if and only if their tags are equal and their bodies are related.
- Two keys are related if and only if they are identical and not secret. Here, the set of secret keys is identified with the domain of $\varphi$ (see below).
- Two ciphertexts $\{v\}_k$ and $\{v'\}_{k'}$ are related if and only if $k = k'$ 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. In other words, $\varphi$ represents the restriction on the attackers' knowledge that each $k \in \mathit{dom}(\varphi)$ is unknown to them and, furthermore, for each $(v, v') \in \varphi(k)$, the ciphertexts $\{v\}_k$ and $\{v'\}_k$ are indistinguishable to the attackers. (See Section 8 for some discussion of the issue of equality for ciphertexts.)

As for expressions, arbitrary expressions are related if and only if 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. $\varphi \vdash^{\mathrm{val}}_{s,s'} v \sim v' : \tau$ and $\varphi \vdash^{\mathrm{exp}}_{s,s'} e \sim e' : \tau$ 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.

Strictly speaking, the relation environment $\varphi$ should take $s$, $s'$ and a (partial) mapping $\Delta$ from keys to types as parameters. Then, for each $k \in s \cap s'$, $\varphi(k)$ is a relation on two values $v$, $v'$ of type $\Delta(k)$ such that $\mathit{Keys}(v) \subseteq s$ and $\mathit{Keys}(v') \subseteq s'$. In order to simplify the notations, however, we omit $s$, $s'$ and $\Delta$ since they are obvious from the context.

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

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

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

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

**Proof.** See the Appendix. □

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

**Proof.** See the Appendix. □

### 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 $\varphi$. However, $\varphi$ gives us 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 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 $\varphi$ 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 $\psi$ 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 $\varphi^{\psi}_{s,s'}(k) = \{(k', k') \mid \psi^{\chi}_{t,t'}(k') = \{(3,5)\}$ for any $t, t'$ and $\chi\}$. Accordingly, we extend the definition of the logical relation for ciphertext types to:

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

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 $\varphi$ in $\varphi^{\psi}_{s,s'}(k)$ is monotonic with respect to extension of $s$, $s'$, and $\psi$. Intuitively, this 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 $\varphi$ 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 $\psi$ and $\chi$ with $dom(\psi) \subseteq s \cap s'$ and $dom(\chi) \subseteq t \cap t'$. Technically, this condition is needed in the proof of Lemma 20 (weakening of logical relation) when $\tau$ is a ciphertext type. We refer to this condition as "$\varphi$ 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

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

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

**Example 13.** Let us see how to prove the correctness of the encoding in Section 3.2 of the improved 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.)

$$\begin{array}{l}\mathtt{new}\ z_A\ \mathtt{in}\ \mathtt{new}\ z_B\ \mathtt{in}\ \mathtt{new}\ z_E\ \mathtt{in} \\ \langle \lambda x.\, \{x\}_{z_A}, \lambda x.\, \{x\}_{z_B}, z_E, \\ \ \langle B, \lambda\{\underline{\mathtt{in}_1}(\langle N_A, A\rangle)\}_{z_B}.\, \mathtt{new}\ N_B\ \mathtt{in} \\ \qquad \mathtt{Some}(\langle\{\langle N_A, N_B, B\rangle\}_{z_A}, \lambda\{\underline{\mathtt{in}_2}(N_B)\}_{z_B}.\, \mathtt{Some}(\{i\}_{N_B})\rangle)\rangle, \\ \ \langle \lambda X.\, \mathtt{new}\ N_A\ \mathtt{in}\ \langle\{\underline{\mathtt{in}_1}(\langle N_A, A\rangle)\}_{z_x}, \\ \qquad\qquad\qquad \lambda\{N_A, N_x, X\}_{z_A}.\, \mathtt{Some}(\{\underline{\mathtt{in}_2}(N_x)\}_{z_x})\rangle\rangle\end{array}$$

Call this expression $NS''_i$. It can be given the type

$$\begin{array}{l}(\tau_1 \to \mathtt{bits}[\tau_1]) \times (\tau_2 \to \mathtt{bits}[\tau_2]) \times \mathtt{key}[\tau_2] \times \\ (\mathtt{nam} \times (\mathtt{bits}[\tau_2] \to \mathtt{option}[\mathtt{bits}[\tau_1] \times \\ \qquad\qquad (\mathtt{bits}[\tau_2] \to \mathtt{option}[\mathtt{bits}[\mathtt{int}]])]) \times \\ (\mathtt{nam} \to (\mathtt{bits}[\tau_2] \times (\mathtt{bits}[\tau_1] \to \mathtt{option}[\mathtt{bits}[\tau_2]]))))\end{array}$$

where $\mathtt{nam}$ is actually just $\mathtt{int}$ 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 $\sigma$. Call this type $\tau$.

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

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

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 $\mathit{Good}(NS''_i)$ 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 $\varphi$ 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.

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

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 $\varphi$ 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

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

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

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 [13, 18, 19, 21, etc.]. Of these, the spi-calculus [4] is one of the most powerful; it comes equipped with useful techniques such as bisimulation [3, 7] for proving behavioral equivalences and static typing for guaranteeing secrecy [1] and authenticity [11]. 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 simply to explore how 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, $\lambda$-calculus offers an attractive starting point, since it is in this setting that relational parametricity is best understood. Of course, the cost of this choice is that we depend on the ability of the $\lambda$-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 $\lambda$-term. Any attacks that we discover for the encoded term must be reality-checked against the original process (cf. the false attack on the encoding of the ffgg protocol in Section 3). However, if the encoding of a process *can* be proved secure, then the process itself should also be secure, at least against our notion of attackers (cf. the correctness proof of the improved Needham-Schroeder public-key protocol in Section 3 and the discussion in Section 8).

Formalizing and proving secrecy as non-interference—i.e., equivalence between instances of a program with different 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, 34, 38], among others.

There have also been many proposals for using techniques from programming languages—in particular, static typing—to guarantee security of programs. For example, Heintze and Riecke [14] proposed $\lambda$-calculus with type-based 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. Other work in this line includes [15, 16, 31, 32, 35]. 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.

Originally, logical relations were developed in the domain of denotational semantics for the purpose of establishing various kinds of correspondence in mathematical models of typed $\lambda$-calculi (see [22, Section 8], for example). In this setting, defining or using a logical relation requires establishing or understanding the denotational model(s) on which the logical relation is defined. In addition, the soundness of such relational reasoning (with respect to the operational semantics) depends on that of the denotational model. We circumvented these issues by adopting the approach of *syntactic* logical relations [6, 28], i.e., (a variant of) logical relations based on a term model of a language.

Since the cryptographic $\lambda$-calculus has a key generation primitive, we must be able to reason about generative names. For this purpose, we adapted Stark's work on $\lambda$-calculus with name generation [36] in formulating both the semantics in Section 4 and the logical relation in Section 6.1. A technical difference of our adaptation from Stark's original is that he introduced bijections while we rely on $\alpha$-conversion in order to manage the possible differences between names generated by each of the related terms. In addition, the combination of the logical relation for name generation and that for type abstraction [33] gave rise to a new problem—namely, how to specify fresh keys that have not yet been generated. This issue is critical when a fresh key is encrypted by another key, which is often the case in programs exchanging keys. We addressed this problem in Section 6.2 by extending the logical relation in a non-trivial way. The same technique would also apply for other purposes such as treating "references to references" in establishing logical relations for ML-like references [30].

Harper and Lillibridge [personal communication, July 2000] have independently developed a *typed seal calculus* that is closely related to our cryptographic $\lambda$-calculus. Their work mainly focuses on encoding *sealing* [23] 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

**Completeness of Logical Relations.** As a method of proving contextual equivalence, it is natural to wonder: can we generalize the logical relations in Section 6, so that the generalization becomes *complete* with respect to contextual equivalence? That is, can all contextually equivalent expressions be logically related? In general, however, it is rather difficult to obtain complete logical relations. Even for the polymorphic $\lambda$-calculus, the (syntactic) logical relation is incomplete [27] for existential types. (The original paper [27] attributes this to the presence of recursion, but a similar counter-example exists even without recursion [personal communication, June 2000].) Establishing a complete logical relation for encryption would first require more investigation of complete logical relations for type abstraction.

**Primitive Values and Primitive Operations** Besides its functional and cryptographic core, our calculus provides only integers and integer operations. It is straightforward to introduce other primitives such as booleans and strings in a similar way, i.e., by adding (1) expressions and values for the primitives, (2) evaluation rules mapping these expressions to values, (3) types for the primitive values and typing rules for the expressions, and (4) an equality relation between those primitive values as a base case of our logical relations.

It remains to see whether and how we can introduce more fundamental extensions such as public-key cryptography as primitives (cf. [2]).

**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, the 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; introducing recursive *types* breaks the very definition of the logical relations. This problem is of concern because it suggests that our approach would not be *sound* with respect to attacks that rely on recursion. (If, indeed, there are any such attacks in reality: observe that, for each particular $\lambda$-term, if there exists an attack that uses recursion to reveal a secret within a finite amount of time, then the same attack should also be possible without using recursion.) We expect that this limitation can be removed by incorporating the theory of logical relation for $\lambda$-calculus with recursive functions and/or recursive types (e.g., [6]).

**Equality for Ciphertexts.** In our calculus, we did not introduce any construct to test ciphertexts for equality. This lack of equality for ciphertexts can be a weakness of our development in this paper for the same reason as the lack of recursion may be. For example, $\mathtt{new}\ x\ \mathtt{in}\ \{3\}_x$ and $\mathtt{new}\ x\ \mathtt{in}\ \{5\}_x$ are equivalent in our calculus, but an attacker may discover the difference just by comparing the ciphertexts as bit strings. Using non-deterministic encryption (a.k.a. random encryption and probabilistic encryption) for implementation does not solve this problem: for instance, $\langle \{123\}_k, \{123\}_k \rangle$ and $\mathtt{let}\ x = \{123\}_k\ \mathtt{in}\ \langle x, x \rangle$ would be inequivalent under non-deterministic encryption, but they *are* equivalent in the present calculus.

The issue of ciphertext equality is closely related to that of polymorphic equality in the standard theory of relational parametricity for type abstraction [39, Section 3.4]. The solution would also be similar—i.e., require the corresponding relation $\varphi(k)$ to respect equality—but it remains to see what effects it will have on reasoning about information hiding by encryption.

**State and Linearity.** Although real programs often have some kind of state or linearity (in the sense of linear logic that some of the "resources" which they offer can be exploited only once), 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 we impose the constraint that the function $\lambda x. \ldots$ is used linearly (i.e., applied only once). A similar example can be given 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 $\lambda$-calculus with state or linearity [5, 30].

Moreover, pursuing this direction of adapting logical relations for linearity and state might lead to a theory of relational parametricity for process calculi with some form of information hiding, such as polymorphic $\pi$-calculus [37] and spi-calculus [4]. We conjecture that, in combination with a big-step, evaluation semantics of process calculi [29], this approach might lead to a more systematic, structural method than the bisimulation-based techniques that have been explored in the past [7, 25] for proving equivalence between concurrent programs with information hiding.

**Beyond Secrecy.** We were able to prove secrecy properties of security protocols by means of logical relations because (1) our logical relations are a means of proving (contextual) equivalence

and (2) equivalence leads to secrecy via non-interference. A natural question is whether we might be able to prove security properties *other* than secrecy—such as authenticity, anonymity, etc.—in a similar fashion via logical relations. The general idea would be to prove the equivalence between (encodings of) a real system and an ideal one whose "security" is obvious (cf. [10], for instance); further study is needed to see what kinds of problems can be addressed in this manner using our framework.

**Type Abstraction via Encryption.** While we have focused here on adapting the theory of type abstraction into encryption, it is also interesting to think of using the techniques 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 type 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 for such a translation [26, Section 4], but proved nothing about it. The results in the present paper—in particular, the logical relations in Section 6—may help improve our understanding of this issue.

## Acknowledgements

We would like to thank 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 at the University of Pennsylvania, as well as the anonymous CSFW and JCS 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] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999. Preliminary version appeared in *Theoretical Aspects of Computer Software*, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 1281, pp. 611–638, 1997.

[2] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 104–115, 2001.

[3] Martín Abadi and Andrew D. Gordon. A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, 5:267–303, 1998. Preliminary version appeared in *7th European Symposium on Programming*, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 1381, pp. 12–26, 1998.

[4] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999. Preliminary version appeared in *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pp. 36–47, 1997.

[5] Gavin M. Bierman, Andrew M. Pitts, and Claudio 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.

[6] Lars Birkedal and Robert Harper. Relational interpretations of recursive types in an operational setting. *Information and Computation*, 155(1–2):3–63, 1999. Summary appeared in *Theoretical Aspects of Computer Software*, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 1281, pp. 458–490, 1997.

[7] Michele Boreale, Rocco De Nicola, and Rosario Pugliese. Proof techniques for cryptographic processes. *SIAM Journal on Computing*, 31(3):947–986, 2002. Preliminary version appeared in *14th Annual IEEE Symposium on Logic in Computer Science*, pp. 157–166, 1999.

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

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

[10] Antonio Durante, Riccardo Focardi, and Roberto Gorrieri. A compiler for analysing cryptographic protocols using non-interference. *ACM Transactions on Software Engineering and Methodology*, 9(4):488–528, 2000.

[11] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop*, pages 145–159, 2001.

[12] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. In *13th IEEE Computer Security Foundations Workshop*, pages 255–268, 2000.

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

[14] Nevin Heintze and Jon 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.

[15] Matthew Hennessy and James Riely. Information flow vs. resource access in the asynchronous pi-calculus. In *Automata, Languages and Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 415–427. Springer-Verlag, 2000.

[16] Kohei Honda and Nobuko Yoshida. A uniform framework for secure information flow. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–92, 2002.

[17] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[18] Catherine 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.

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

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

[21] Jonathan K. Millen. CSFW home page. `http://www2.csl.sri.com/csfw/`, 2004.

[22] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

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

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

[25] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–586, 2000. Extended abstract appeared in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997, pp. 531–584.

[26] Benjamin C. Pierce and Eijiro Sumii. Relating cryptography and polymorphism, 2000. Draft. `http://www.cis.upenn.edu/~sumii/pub/`.

[27] Andrew M. Pitts. Existential types: Logical relations and operational equivalence. In *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, 1998.

[28] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000. Preliminary version appeared in *HOOTS II Second Workshop on Higher-Order Operational Techniques in Semantics*, *Electronic Notes in Theoretical Computer Science*, vol. 10, 1998.

[29] Andrew M. Pitts and Joshua R. X. Ross. Process calculus based upon evaluation to committed form. *Theoretical Computer Science*, 195:155–182, 1998. Preliminary version appeared in *CONCUR'96: Concurrency Theory*, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 1119, pp. 18–33, 1996.

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

[31] François Pottier. A simple view of type-secure information flow in the $\pi$-calculus. In *15th IEEE Computer Security Foundations Workshop*, pages 320–330, 2002.

[32] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 319–330, 2002.

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

[34] Peter Y. A. Ryan and Steve A. Schneider. Process algebra and non-interference. In *12th IEEE Computer Security Foundations Workshop*, pages 214–227, 1999.

[35] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, 1998.

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

[37] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.

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

[39] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

# Appendix

For the sake of conciseness, we write $\mathcal{R}^{\mathrm{val}}_{s,s'}(\tau)\varphi$ and $\mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$, respectively, for the sets $\{(v,v') \mid \varphi \vdash^{\mathrm{val}}_{s,s'} v \sim v' : \tau\}$ and $\{(e,e') \mid \varphi \vdash^{\mathrm{exp}}_{s,s'} e \sim e' : \tau\}$. That is, $(v,v') \in \mathcal{R}^{\mathrm{val}}_{s,s'}(\tau)\varphi$ means $\varphi \vdash^{\mathrm{val}}_{s,s'} v \sim v' : \tau$ and $(e,e') \in \mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$ means $\varphi \vdash^{\mathrm{exp}}_{s,s'} e \sim e' : \tau$.

In the proofs, we use the following lemmas.

### Lemmas about Evaluation

**Lemma 14 (Monotonic Increase of Key Set).** If $(s)e \Downarrow (s')v$, then $s \subseteq s'$. If $s \supseteq \mathit{Keys}(e)$ furthermore, then $s' \supseteq \mathit{Keys}(v)$.

**Proof.** By induction on the derivation of $(s)e \Downarrow (s')v$. □

**Lemma 15 (Evaluation of Value).** $(s)v \Downarrow (s)v$ for any $s$ and $v$. In addition, if $(s)v \Downarrow V$, then $V = (s)v$.

**Proof.** Immediately follows by induction on the structure of $v$. □

**Lemma 16 (Weakening of Key Set).** If $(s)e \Downarrow (s')v$, then $(s \uplus t)e \Downarrow (s' \uplus t)v$ for any $t$ with $t \cap s = \emptyset$ and $t \cap s' = \emptyset$.

**Proof.** By induction on the derivation of $(s)e \Downarrow (s')v$. □

### Lemmas about Typing

**Lemma 17 (Weakening of Type Judgment).** If $\Gamma, \Delta \vdash e : \tau$, then $\Gamma \uplus \Gamma', \Delta \uplus \Delta' \vdash e : \tau$ for any $\Gamma'$ and $\Delta'$ with $\mathit{dom}(\Gamma) \cap \mathit{dom}(\Gamma') = \emptyset$ and $\mathit{dom}(\Delta) \cap \mathit{dom}(\Delta') = \emptyset$.

**Proof.** By induction on the derivation of $\Gamma, \Delta \vdash e : \tau$. □

**Lemma 18 (Substitution Lemma).** Let $\Gamma = \{\tilde{x} \mapsto \tilde{\tau}\}$. If $\Gamma, \Delta \vdash e : \tau$ and $\emptyset, \Delta \vdash \tilde{v} : \tilde{\tau}$, then $\Gamma, \Delta \vdash [\tilde{v}/\tilde{x}]e : \tau$.

**Proof.** By induction on the derivation of $\Gamma, \Delta \vdash e : \tau$. We use Lemma 17 when the last rule used for the derivation is (Var). □

### Lemmas about Logical Relation

**Lemma 19 (Coincidence of Logical Relations).** $\mathcal{R}^{\mathrm{val}}_{s,s'}(\tau)\varphi \subseteq \mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$ for any $s$, $s'$, $\tau$, $\varphi$.

**Proof.** Immediately follows from the definition of $\mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$ and from Lemma 15. □

**Lemma 20 (Weakening of Logical Relation).** Suppose $s \cap s_0 = \emptyset$ and $s' \cap s'_0 = \emptyset$ where $dom(\varphi) \subseteq s \cap s'$ and $dom(\varphi_0) \subseteq s_0 \cap s'_0$. Then, $\mathcal{R}^{\mathrm{val}}_{s,s'}(\tau)\varphi \subseteq \mathcal{R}^{\mathrm{val}}_{s \uplus s_0, s' \uplus s'_0}(\tau)(\varphi \uplus \varphi_0)$ and $\mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi \subseteq \mathcal{R}^{\mathrm{exp}}_{s \uplus s_0, s' \uplus s'_0}(\tau)(\varphi \uplus \varphi_0)$.

**Proof.** By induction on the structure of $\tau$. In the proof of the latter half, note that, by Lemma 16, if $(s)e \Downarrow (s \uplus t)v$ and $(s')e' \Downarrow (s' \uplus t')v'$, then $(s \uplus s_0)e \Downarrow (s \uplus s_0 \uplus t)v$ and $(s' \uplus s'_0)e' \Downarrow (s' \uplus s'_0 \uplus t')v'$. □

### Proof of Theorem 3

By induction on the derivation of $(s_1)e \Downarrow (s_1 \uplus s'_1)v_1$. The latter half of Lemma 14 is used when the last rule used for the derivation is either (1) the evaluation rule for function application or (2) the evaluation rule for decryption. □

### Proof of Theorem 5

By induction on the derivation of $(s)[\tilde{v}/\tilde{x}]e \Downarrow V$. We perform case analysis on the form of $e$. We show the following four cases (the other cases are similar).

**Case** $e = x$**.** By (Var), $x = x_i$ and $\tau = \tau_i$ for some $i$. Thus, by Lemma 15, we have $V = (s)v_i$. So the theorem follows by letting $v = v_i$ and $\Delta' = \emptyset$.

**Case** $e = \lambda x.\, e_0$**.** By the evaluation rule for $\lambda$-abstraction, we have $V = (s)\lambda x.\, [\tilde{v}/\tilde{x}]e_0$. Meanwhile, by (Abs), $\tau$ is of the form $\sigma_1 \to \sigma_2$ and we have $\Gamma \uplus \{x \mapsto \sigma_1\}, \Delta \vdash e_0 : \sigma_2$. Therefore, by Lemma 18, we have $\Gamma \uplus \{x \mapsto \sigma_1\}, \Delta \vdash [\tilde{v}/\tilde{x}]e_0 : \sigma_2$. Then, by Lemma (Abs), we have $\Gamma, \Delta \vdash \lambda x.\, [\tilde{v}/\tilde{x}]e_0 : \sigma_1 \to \sigma_2$. Thus, the theorem follows by letting $v = \lambda x.\, [\tilde{v}/\tilde{x}]e_0$ and $\Delta' = \emptyset$.

**Case** $e = \texttt{new}\ x\ \texttt{in}\ e_0$**.** By the evaluation rule for key generation, we have $(s \uplus \{k\})[k/x][\tilde{v}/\tilde{x}]e_0 \Downarrow V$ for some $k$. Meanwhile, by (Key), we have $\emptyset, \Delta \uplus \{k \mapsto \tau'\} \vdash k : \texttt{key}[\tau']$. In addition, by Lemma 17, we have $\emptyset, \Delta \uplus \{k \mapsto \tau'\} \vdash \tilde{v} : \tilde{\tau}$, Furthermore, by (New) we have $\Gamma \uplus \{x \mapsto \texttt{key}[\tau']\}, \Delta \vdash e_0 : \tau$, so by Lemma 17 we have $\Gamma \uplus \{x \mapsto \texttt{key}[\tau']\}, \Delta \uplus \{k \mapsto \tau'\} \vdash e_0 : \tau$. Therefore, by the induction hypothesis, there exist some $v'_0$ and $\Delta_0$ such that $V = (s \uplus \{k\} \uplus s_0)v'_0$ and $\emptyset, \Delta \uplus \{k \mapsto \tau'\} \uplus \Delta_0 \vdash v'_0 : \tau$ for $s_0 = dom(\Delta_0)$. Thus, the theorem follows by letting $v = v'_0$ and $\Delta' = \{k \mapsto \tau'\} \uplus \Delta_0$.

**Case** $e = (\texttt{let}\ \{x\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4)$**.** By (Dec), we have $\Gamma, \Delta \vdash e_1 : \texttt{key}[\tau']$, $\Gamma, \Delta \vdash e_2 : \texttt{bits}[\tau']$, $\Gamma \uplus \{x \mapsto \tau'\}, \Delta \vdash e_3 : \tau$, and $\Gamma, \Delta \vdash e_4 : \tau$ for some $\tau'$. In addition, by the evaluation rules for decryption, we have $(s)[\tilde{v}/\tilde{x}]e_1 \Downarrow V_1$ for some $V_1$. Therefore, by the induction hypothesis, there exist some $v'_1$ and $\Delta_1$ such that $V_1 = (s \uplus s_1)v'_1$ and $\emptyset, \Delta \uplus \Delta_1 \vdash v'_1 : \texttt{key}[\tau']$ for $s_1 = dom(\Delta_1)$. Then, since $v'_1$ is a value of a key type, $v'_1$ is of the form $k_1$ by (Key).

Furthermore, by the evaluation rules for decryption, we have $(s \uplus s_1)[\tilde{v}/\tilde{x}]e_2 \Downarrow V_2$ for some $V_2$. Meanwhile, by Lemma 17, we have $\Gamma, \Delta \uplus \Delta_1 \vdash e_2 : \mathtt{bits}[\tau']$. Furthermore, by Lemma 17, we have $\emptyset, \Delta \uplus \Delta_1 \vdash \tilde{v} : \tilde{\tau}$. Therefore, by the induction hypothesis, there exist some $v_2'$ and $\Delta_2$ such that $V_2 = (s \uplus s_1 \uplus s_2)v_2'$ and $\emptyset, \Delta \uplus \Delta_1 \uplus \Delta_2 \vdash v_2' : \mathtt{bits}[\tau']$ for $s_2 = dom(\Delta_2)$. Then, since $v_2'$ is a value of a ciphertext type, $v_2'$ is of the form $\{v'\}_{k_2}$ and $\emptyset, \Delta \uplus \Delta_1 \uplus \Delta_2 \vdash v' : \tau'$ by (Enc).

Now we perform the following case analysis.

**Sub-case** $k_1 = k_2$**.** Let $k_1 = k_2 = k$. By the evaluation rules for decryption, we have $(s \uplus s_1 \uplus s_2)[v'/x][\tilde{v}/\tilde{x}]e_3 \Downarrow V$. In addition, by Lemma 17, we have $\Gamma \uplus \{x \mapsto \tau'\}, \Delta \uplus \Delta_1 \uplus \Delta_2 \vdash e_3 : \tau$. Furthermore, by Lemma 17, we have $\emptyset, \Delta \uplus \Delta_1 \uplus \Delta_2 \vdash \tilde{v} : \tilde{\tau}$. Therefore, by the induction hypothesis, there exist some $v_3'$ and $\Delta_3$ such that $V = (s \uplus s_1 \uplus s_2 \uplus s_3)v_3'$ and $\emptyset, \Delta \uplus \Delta_1 \uplus \Delta_2 \uplus \Delta_3 \vdash v_3' : \tau$ for $s_3 = dom(\Delta_3)$. Thus, the theorem follows by letting $v = v_3'$ and $\Delta' = \Delta_1 \uplus \Delta_2 \uplus \Delta_3$.

**Sub-case** $k_1 \neq k_2$**.** By the evaluation rules for decryption, we have $(s \uplus s_1 \uplus s_2)[\tilde{v}/\tilde{x}]e_4 \Downarrow V$. In addition, by Lemma 17, we have $\Gamma, \Delta \uplus \Delta_1 \uplus \Delta_2 \vdash e_4 : \tau$. Furthermore, by Lemma 17, we have $\emptyset, \Delta \uplus \Delta_1 \uplus \Delta_2 \vdash \tilde{v} : \tilde{\tau}$. Therefore, by the induction hypothesis, there exist some $v_4'$ and $\Delta_4$ such that $V = (s \uplus s_1 \uplus s_2 \uplus s_4)v_4'$ and $\emptyset, \Delta \uplus \Delta_1 \uplus \Delta_2 \uplus \Delta_4 \vdash v_4' : \tau$ for $s_4 = dom(\Delta_4)$. Thus, the theorem follows by letting $v = v_4'$ and $\Delta' = \Delta_1 \uplus \Delta_2 \uplus \Delta_4$. □

## Proof of Theorem 10

By induction on the structure of $e$. We perform case analysis on the form of $e$. We show the following four cases (the other cases are similar).

**Case** $e = x$**.** By (Var), we have $x = x_i$ and $\tau = \tau_i$ for some $i$. Thus, by Lemma 19, we have $([\tilde{v}/\tilde{x}]e, [\tilde{v}'/\tilde{x}]e) = (v_i, v_i') \in \mathcal{R}^{\mathrm{val}}_{s,s'}(\tau)\varphi \subseteq \mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau_i)\varphi$.

**Case** $e = \lambda x.\, e_0$**.** By (Abs), $\tau$ is of the form $\sigma_1 \to \sigma_2$ and we have $\Gamma \uplus \{x \mapsto \sigma_1\}, \Delta \vdash e_0 : \sigma_2$. Assume $(v, v') \in \mathcal{R}^{\mathrm{val}}_{s \uplus t, s' \uplus t'}(\sigma_1)(\varphi \uplus \psi)$ with $dom(\psi) \subseteq t \cap t'$. By Lemma 20, we have $(\tilde{v}, \tilde{v}') \in \mathcal{R}^{\mathrm{val}}_{s \uplus t, s \uplus t'}(\tilde{\tau})$ $(\varphi \uplus \psi)$. Therefore, by the induction hypothesis, we have $([v/x][\tilde{v}/\tilde{x}]e_0, [v'/x][\tilde{v}'/\tilde{x}]e_0) \in \mathcal{R}^{\mathrm{exp}}_{s \uplus t, s' \uplus t'}$ $(\sigma_2)\varphi \uplus \psi$. Thus, by the definition of $\mathcal{R}^{\mathrm{val}}_{s,s'}(\sigma_1 \to \sigma_2)\varphi$ and by Lemma 19, we have $([\tilde{v}/\tilde{x}]e, [\tilde{v}'/\tilde{x}]e) = (\lambda x.\, [\tilde{v}/\tilde{x}]e_0, \lambda x.\, [\tilde{v}'/\tilde{x}]e_0) \in \mathcal{R}^{\mathrm{val}}_{s,s'}(\sigma_1 \to \sigma_2)\varphi \subseteq \mathcal{R}^{\mathrm{exp}}_{s,s'}(\sigma_1 \to \sigma_2)\varphi = \mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$.

**Case** $e = \mathtt{new}\ x\ \mathtt{in}\ e_0$**.** By (New), we have $\Gamma \uplus \{x \mapsto \mathtt{key}[\sigma]\}, \Delta \vdash e_0 : \tau$ for some $\sigma$. By Lemma 20, we have $(\tilde{v}, \tilde{v}') \in \mathcal{R}^{\mathrm{val}}_{s \uplus \{k\}, s' \uplus \{k\}}(\tilde{\tau})\varphi$ for any $k \notin s \cup s'$. In addition, by the definition of $\mathcal{R}^{\mathrm{val}}_{s \uplus \{k\}, s' \uplus \{k\}}(\mathtt{key}[\sigma])\varphi$, we have $(k, k) \in \mathcal{R}^{\mathrm{val}}_{s \uplus \{k\}, s' \uplus \{k\}}(\mathtt{key}[\sigma])\varphi$. Therefore, by the induction hypothesis, we have $([k/x][\tilde{v}/\tilde{x}]e_0, [k/x][\tilde{v}'/\tilde{x}]e_0) \in \mathcal{R}^{\mathrm{exp}}_{s \uplus \{k\}, s' \uplus \{k\}}(\tau)\varphi$. Thus, by the definition of $\mathcal{R}^{\mathrm{exp}}_{s \uplus \{k\}, s' \uplus \{k\}}(\tau)\varphi$, there exist some $t_0$, $w_0$, $t_0'$, $w_0'$, and $\psi_0$ with $dom(\psi_0) \subseteq t_0 \cap t_0'$ such that $(s \uplus \{k\})[k/x][\tilde{v}/\tilde{x}]e_0 \Downarrow (s \uplus \{k\} \uplus t_0)w_0$, $(s' \uplus \{k\})[k/x][\tilde{v}'/\tilde{x}]e_0 \Downarrow (s' \uplus \{k\} \uplus t_0')w_0'$, and $(w_0, w_0') \in \mathcal{R}^{\mathrm{val}}_{s \uplus \{k\} \uplus t_0, s' \uplus \{k\} \uplus t_0'}(\tau)(\varphi \uplus \psi_0)$. Then, by the evaluation rule for key generation, we have $(s)\mathtt{new}\ x\ \mathtt{in}\ [\tilde{v}/\tilde{x}]e_0 \Downarrow (s \uplus \{k\} \uplus t_0)w_0$ and $(s')\mathtt{new}\ x\ \mathtt{in}\ [\tilde{v}'/\tilde{x}]e_0 \Downarrow (s' \uplus \{k\} \uplus t_0')w_0'$. Thus, by letting $t = \{k\} \uplus t_0$, $v = w_0$, $t' = \{k\} \uplus t_0'$, $v' = w_0'$, and $\psi = \psi_0$ in the definition of $\mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$, we have $([\tilde{v}/\tilde{x}]e, [\tilde{v}'/\tilde{x}]e) = (\mathtt{new}\ x\ \mathtt{in}\ [\tilde{v}/\tilde{x}]e_0, \mathtt{new}\ x\ \mathtt{in}\ [\tilde{v}'/\tilde{x}]e_0) \in \mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$.

**Case** $e = (\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4)$**.** By (Dec), we have $\Gamma, \Delta \vdash e_1 : \mathtt{key}[\sigma], \Gamma, \Delta \vdash e_2 : \mathtt{bits}[\sigma]$, $\Gamma \uplus \{x \mapsto \sigma\}, \Delta \vdash e_3 : \tau$, and $\Gamma, \Delta \vdash e_4 : \tau$ for some $\sigma$. Then, by the induction hypothesis, we have $([\tilde{v}/\tilde{x}]e_1, [\tilde{v}'/\tilde{x}]e_1) \in \mathcal{R}^{\mathrm{exp}}_{s,s'}(\mathtt{key}[\sigma])\varphi$. Thus, by the definition of $\mathcal{R}^{\mathrm{exp}}_{s,s'}(\mathtt{key}[\sigma])\varphi$, there exist some $t_1$, $w_1$, $t_1'$, $w_1'$, and $\psi_1$ with $dom(\psi_1) \subseteq t_1 \cap t_1'$ such that $(s)[\tilde{v}/\tilde{x}]e_1 \Downarrow (s \uplus t_1)w_1$, $(s')[\tilde{v}'/\tilde{x}]e_1 \Downarrow (s' \uplus t_1')w_1'$, and $(w_1, w_1') \in \mathcal{R}^{\mathrm{val}}_{s \uplus t_1, s' \uplus t_1'}(\mathtt{key}[\sigma])(\varphi \uplus \psi_1)$. Then, by the definition of $\mathcal{R}^{\mathrm{val}}_{s \uplus t_1, s' \uplus t_1'}(\mathtt{key}[\sigma])(\varphi \uplus \psi_1)$, we have $w_1 = w_1'$ is of the form $k_1$ where $k_1 \in (s \uplus t_1) \cap (s' \uplus t_1')$ and $k_1 \notin dom(\varphi \uplus \psi_1)$.

Furthermore, by the induction hypothesis, we have $([\tilde{v}/\tilde{x}]e_2, [\tilde{v}'/\tilde{x}]e_2) \in \mathcal{R}^{\mathrm{exp}}_{s\uplus t_1, s'\uplus t'_1}(\mathtt{bits}[\sigma])(\varphi \uplus \psi_1)$. Thus, by the definition of $\mathcal{R}^{\mathrm{exp}}_{s\uplus t_1, s'\uplus t'_1}(\mathtt{bits}[\sigma])(\varphi \uplus \psi_1)$, there exist some $t_2$, $w_2, t'_2$, $w'_2$, and $\psi_2$ with $dom(\psi_2) \subseteq t_2 \cap t'_2$ such that $(s \uplus t_1)[\tilde{v}/\tilde{x}]e_2 \Downarrow (s \uplus t_1 \uplus t_2)w_2$, $(s' \uplus t'_1)[\tilde{v}'/\tilde{x}]e_2 \Downarrow (s' \uplus t'_1 \uplus t'_2)w'_2$, and $(w_2, w'_2) \in \mathcal{R}^{\mathrm{val}}_{s\uplus t_1\uplus t_2, s'\uplus t'_1\uplus t'_2}(\mathtt{bits}[\sigma])(\varphi \uplus \psi_1 \uplus \psi_2)$. Then, by the definition of $\mathcal{R}^{\mathrm{val}}_{s\uplus t_1\uplus t_2, s\uplus t'_1\uplus t'_2}(\mathtt{bits}[\sigma])(\varphi \uplus \psi_1 \uplus \psi_2)$, $w_2$ and $w'_2$ are respectively of the form $\{w\}_{k_2}$ and $\{w'\}_{k_2}$ where $k_2 \in (s \uplus t_1 \uplus t_2) \cap (s' \uplus t'_1 \uplus t'_2)$.

Now we perform the following case analysis.

**Sub-case** $k_1 = k_2$**.** Since $k_1 \notin dom(\varphi \uplus \psi_1 \uplus \psi_2)$, we have $(w, w') \in \mathcal{R}^{\mathrm{val}}_{s\uplus t_1\uplus t_2, s'\uplus t'_1\uplus t'_2}(\sigma)(\varphi \uplus \psi_1 \uplus \psi_2)$. In addition, by Lemma 20, we have $(\tilde{v}, \tilde{v}') \in \mathcal{R}^{\mathrm{val}}_{s\uplus t_1\uplus t_2, s'\uplus t'_1\uplus t'_2}(\tilde{\tau})(\varphi \uplus \psi_1 \uplus \psi_2)$. Then, by the induction hypothesis, we have $([w/x][\tilde{v}/\tilde{x}]e_3, [w'/x][\tilde{v}'/\tilde{x}]e_3) \in \mathcal{R}^{\mathrm{exp}}_{s\uplus t_1\uplus t_2, s'\uplus t'_1\uplus t'_2}(\tau)(\varphi \uplus \psi_1 \uplus \psi_2)$. Thus, by the definition of $\mathcal{R}^{\mathrm{exp}}_{s\uplus t_1\uplus t_2, s'\uplus t'_1\uplus t'_2}(\tau)(\varphi \uplus \psi_1 \uplus \psi_2)$, there exist some $t_3$, $w_3, t'_3$, $w'_3$, and $\psi_3$ with $dom(\psi_3) \subseteq t_3 \cap t'_3$ such that $(s \uplus t_1 \uplus t_2)[w/x][\tilde{v}/\tilde{x}]e_3 \Downarrow (s \uplus t_1 \uplus t_2 \uplus t_3)w_3$, $(s' \uplus t'_1 \uplus t'_2)[w'/x][\tilde{v}'/\tilde{x}]e_3 \Downarrow (s' \uplus t'_1 \uplus t'_2 \uplus t'_3)w'_3$, and $(w_3, w'_3) \in \mathcal{R}^{\mathrm{val}}_{s\uplus t_1\uplus t_2\uplus t_3, s'\uplus t'_1\uplus t'_2\uplus t'_3}(\tau)(\varphi \uplus \psi_1 \uplus \psi_2 \uplus \psi_3)$. Therefore, by the evaluation rules for decryption, $(s)\mathtt{let}\ \{x\}_{[\tilde{v}/\tilde{x}]e_1} = [\tilde{v}/\tilde{x}]e_2\ \mathtt{in}\ [\tilde{v}/\tilde{x}]e_3\ \mathtt{else}\ [\tilde{v}/\tilde{x}]e_4 \Downarrow (s \uplus t_1 \uplus t_2 \uplus t_3)w_3$ and $(s')\mathtt{let}\ \{x\}_{[\tilde{v}'/\tilde{x}]e_1} = [\tilde{v}'/\tilde{x}]e_2\ \mathtt{in}\ [\tilde{v}'/\tilde{x}]e_3\ \mathtt{else}\ [\tilde{v}'/\tilde{x}]e_4 \Downarrow (s' \uplus t'_1 \uplus t'_2 \uplus t'_3)w'_3$. Thus, by letting $t = t_1 \uplus t_2 \uplus t_3$, $v = w_3$, $t' = t'_1 \uplus t'_2 \uplus t'_3$, $v' = w'_3$, and $\psi = \psi_1 \uplus \psi_2 \uplus \psi_3$ in the definition of $\mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$, we have $([\tilde{v}/\tilde{x}]e, [\tilde{v}'/\tilde{x}]e) = (\mathtt{let}\ \{x\}_{[\tilde{v}/\tilde{x}]e_1} = [\tilde{v}/\tilde{x}]e_2\ \mathtt{in}\ [\tilde{v}/\tilde{x}]e_3\ \mathtt{else}\ [\tilde{v}/\tilde{x}]e_4, \mathtt{let}\ \{x\}_{[\tilde{v}'/\tilde{x}]e_1} = [\tilde{v}'/\tilde{x}]e_2\ \mathtt{in}\ [\tilde{v}'/\tilde{x}]e_3\ \mathtt{else}\ [\tilde{v}'/\tilde{x}]e_4) \in \mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$.

**Sub-case** $k_1 \neq k_2$**.** By Lemma 20, we have $(\tilde{v}, \tilde{v}') \in \mathcal{R}^{\mathrm{val}}_{s\uplus t_1\uplus t_2, s'\uplus t'_1\uplus t'_2}(\tilde{\tau})(\varphi \uplus \psi_1 \uplus \psi_2)$. Then, by the induction hypothesis, we have $([\tilde{v}/\tilde{x}]e_4, [\tilde{v}'/\tilde{x}]e_4) \in \mathcal{R}^{\mathrm{exp}}_{s\uplus t_1\uplus t_2, s'\uplus t'_1\uplus t'_2}(\tau)(\varphi \uplus \psi_1 \uplus \psi_2)$. Thus, by the definition of $\mathcal{R}^{\mathrm{exp}}_{s\uplus t_1\uplus t_2, s'\uplus t'_1\uplus t'_2}(\tau)(\varphi \uplus \psi_1 \uplus \psi_2)$, there exist some $t_4$, $w_4, t'_4$, $w'_4$, and $\psi_4$ with $dom(\psi_4) \subseteq t_4 \cap t'_4$ such that $(s \uplus t_1 \uplus t_2)[\tilde{v}/\tilde{x}]e_4 \Downarrow (s \uplus t_1 \uplus t_2 \uplus t_4)w_4$, $(s' \uplus t'_1 \uplus t'_2)[\tilde{v}'/\tilde{x}]e_4 \Downarrow (s' \uplus t'_1 \uplus t'_2 \uplus t'_4)w'_4$ and $(w_4, w'_4) \in \mathcal{R}^{\mathrm{val}}_{s\uplus t_1\uplus t_2\uplus t_4, s'\uplus t'_1\uplus t'_2\uplus t'_4}(\tau)(\varphi \uplus \psi_1 \uplus \psi_2 \uplus \psi_4)$. Therefore, by the evaluation rules for decryption, we have $(s)\mathtt{let}\ \{x\}_{[\tilde{v}/\tilde{x}]e_1} = [\tilde{v}/\tilde{x}]e_2\ \mathtt{in}\ [\tilde{v}/\tilde{x}]e_4\ \mathtt{else}\ [\tilde{v}/\tilde{x}]e_4 \Downarrow (s \uplus t_1 \uplus t_2 \uplus t_4)w_4$ and $(s')\mathtt{let}\ \{x\}_{[\tilde{v}'/\tilde{x}]e_1} = [\tilde{v}'/\tilde{x}]e_2\ \mathtt{in}\ [\tilde{v}'/\tilde{x}]e_4\ \mathtt{else}\ [\tilde{v}'/\tilde{x}]e_4 \Downarrow (s \uplus t'_1 \uplus t'_2 \uplus t'_4)w'_4$. Thus, by letting $t = t_1 \uplus t_2 \uplus t_4$, $v = w_4$, $t' = t'_1 \uplus t'_2 \uplus t'_4$, $v' = w'_4$, and $\psi = \psi_1 \uplus \psi_2 \uplus \psi_4$ in the definition of $\mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$, we have $([\tilde{v}/\tilde{x}]e, [\tilde{v}'/\tilde{x}]e) = (\mathtt{let}\ \{x\}_{[\tilde{v}/\tilde{x}]e_1} = [\tilde{v}/\tilde{x}]e_2\ \mathtt{in}\ [\tilde{v}/\tilde{x}]e_4\ \mathtt{else}\ [\tilde{v}/\tilde{x}]e_4, \mathtt{let}\ \{x\}_{[\tilde{v}'/\tilde{x}]e_1} = [\tilde{v}'/\tilde{x}]e_2\ \mathtt{in}\ [\tilde{v}'/\tilde{x}]e_4\ \mathtt{else}\ [\tilde{v}'/\tilde{x}]e_4) \in \mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$. □

## Proof of Corollary 11

By Theorem 10, we have $(f, f) \in \mathcal{R}^{\mathrm{exp}}_{\emptyset,\emptyset}(\tau \to \mathtt{bool})\emptyset$ for any $f$ with $\emptyset, \emptyset \vdash f : \tau \to \mathtt{bool}$. Then, by the definition of $\mathcal{R}^{\mathrm{exp}}_{\emptyset,\emptyset}(\tau \to \mathtt{bool})\emptyset$, there exist some $s, w, s', w', \varphi$ with $dom(\varphi) \subseteq s \cap s'$ such that $(\emptyset)f \Downarrow (s)w$, $(\emptyset)f \Downarrow (s')w'$, and $(w, w') \in \mathcal{R}^{\mathrm{val}}_{s,s'}(\tau \to \mathtt{bool})\varphi$. Then, by the definition of $\mathcal{R}^{\mathrm{val}}_{s,s'}(\tau \to \mathtt{bool})\varphi$, $w$ and $w'$ are respectively of the form $\lambda x.\, e_0$ and $\lambda x.\, e'_0$ where $([v/x]e_0, [v'/x]e'_0) \in \mathcal{R}^{\mathrm{exp}}_{s\uplus t, s'\uplus t'}(\mathtt{bool})(\varphi \uplus \psi)$ for any $v, v', t, t', \psi$ with $dom(\psi) \subseteq t \cap t'$ such that $(v, v') \in \mathcal{R}^{\mathrm{val}}_{s\uplus t, s'\uplus t'}(\tau)(\varphi \uplus \psi)$.

Meanwhile, by Lemma 20, we have $(e, e') \in \mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$. Then, by the definition of $\mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$, there exist some $t, v, t', v', \psi$ with $dom(\psi) \subseteq t \cap t'$ such that $(s)e \Downarrow (s \uplus t)v$, $(s')e' \Downarrow (s' \uplus t')v'$, and $(v, v') \in \mathcal{R}^{\mathrm{val}}_{s\uplus t, s'\uplus t'}(\tau)(\varphi \uplus \psi)$.

Therefore, $([v/x]e_0, [v'/x]e'_0) \in \mathcal{R}^{\mathrm{exp}}_{s\uplus t, s'\uplus t'}(\mathtt{bool})(\varphi \uplus \psi)$. Then, by the definition of $\mathcal{R}^{\mathrm{exp}}_{s\uplus t, s'\uplus t'}(\mathtt{bool})(\varphi \uplus \psi)$, there exist some $s_0, w_0, s'_0, w'_0, \varphi_0$ with $dom(\varphi_0) \subseteq s_0 \cap s'_0$ such that $(s \uplus t)[v/x]e_0 \Downarrow (s \uplus t \uplus s_0)w_0$, $(s' \uplus t')[v'/x]e'_0 \Downarrow (s' \uplus t' \uplus s'_0)w_0$, and $(w_0, w'_0) \in \mathcal{R}^{\mathrm{val}}_{s\uplus t\uplus s_0, s'\uplus t'\uplus s'_0}(\mathtt{bool})(\varphi \uplus \psi \uplus \varphi_0)$. Then, by the definition of $\mathcal{R}^{\mathrm{val}}_{s\uplus t\uplus s_0, s'\uplus t'\uplus s'_0}(\mathtt{bool})(\varphi \uplus \psi \uplus \varphi_0)$, we have $w_0 = w'_0 = \mathtt{true}$ or $w_0 = w'_0 = \mathtt{false}$.

By the way, by the evaluation rules for function application, we have $(\emptyset)fe \Downarrow (s \uplus t \uplus s_0)w_0$ and $(\emptyset)fe' \Downarrow (s' \uplus t' \uplus s'_0)w'_0$. Thus, by Definition 7, we have $\vdash e \approx e' : \tau$. □