

Confidential Safety via Correspondence Assertions^{*}

Radha Jagadeesan¹, Alan Jeffrey², Corin Pitcher¹, and James Riely¹

¹ School of Computing, DePaul University

² Bell Labs, Alcatel-Lucent

Abstract. We study confidential safety, a notion of secrecy that arises naturally in adversarial systems. We provide a formal definition in the context of a concurrent while-language. We provide techniques for establishing confidential safety, building on three ingredients: (1) We develop a novel view of correspondence assertions that focuses on their confidentiality (in contrast to the traditional view of correspondence assertions that uses their integrity to derive secrecy and authentication properties of protocols). (2) We establish the confidentiality of a correspondence assertion using techniques reminiscent of information flow, and incorporating cryptography using confounders. (3) We prove protocols are confidentially safe using techniques based on frame-bisimulation that exploit the non-interference properties of well-typed programs.

Keywords: Language based security, static analysis, confidentiality.

1 Introduction

The focus of this paper is on a notion of secrecy that arises naturally in adversarial systems. The general scenario is as follows: all agents, honest and otherwise, agree on a space of possible values. An honest agent chooses one of these values, and aims to make sure that this particular choice is not guessable by the adversary even in the context of a distributed protocol. We refer to this notion as *confidential safety*. The following examples are motivational.

Example 1. Alice is a ticket granting service. The principals Bob, Charlie, etc. first register with Alice. Following the standard network-as-opponent model, all such communication is monitored by Mallory. So, the identities of all principals are public and known to everyone. When a ticket is available, Alice chooses one of the registered principals and sends mail to the chosen principal using an mail server Sam.

Mallory is aware of the principals who are the possible targets of the mail from Alice. Our aim is to ensure that Mallory cannot reliably reduce this space of possibilities to a singleton and guess the identity of the recipient of the mail from Alice. □

^{*} Radha Jagadeesan and Corin Pitcher were supported by NSF 0916741. James Riely was supported by NSF Career 0347542.

Example 2. Consider a scenario with a server Sam and an associated accountability appliance Alice. Principals such as Bob and Charlie interact with Sam and perform operations (eg. database lookup and update). In order to establish a logging framework used in accountability, Sam records the transactions of interest with Alice who creates a log. Again, the opponent Mallory monitors all communication.

Mallory knows the set of all possible operations that could have been invoked by Bob on Sam. Our aim is to ensure that Mallory cannot reliably reduce this set to exactly guess the operations performed by Bob at the server Sam. \square

Our approach to establishing the confidential safety of a protocol is based on a novel variation on correspondence assertions. Consider the traditional use of correspondence assertions for authentication. Any agent (including dishonest ones) can begin an authentication transaction. Authentication is performed by honest agents who can perform end of a transaction. Such an end is correct if it has a matching begin; the matching of the end to its corresponding begin ensures that the honest agent has only performed a valid authentication. The goal of protocol design is to maximize the number of correct ends. The primary goal of static analysis in this setting is to make sure that *all* ends are correct.

The correctness of the authentication transaction can be viewed as arising from the integrity of the correspondence assertion. Cryptographic methods often play a crucial role in achieving this aim. In contrast, we achieve our aims by ensuring the confidentiality of the correspondence assertion.

We view the honest agent as performing the begin of the transaction and the attacker performs the end of the transaction, thus inverting the roles in the prior use of correspondence assertions. For example, when Alice sends a message to Bob via the server Sam, Alice performs `begin(Alice, Bob)`. What the attacker Mallory is trying to do is to deduce the target of the message from Alice. We model this by Mallory performing an `end(Alice, Bob)`. Thus, an honest agent uses begins to indicate transactions that are intended to be non-guessable by Mallory whereas Mallory uses ends to make an assertion about previous honest agent messages. The correctness of an end is as before — an end is correct if it has a matching begin. However, in contrast to the earlier uses of correspondence assertions, the goal of protocol design now is to minimize the number of correct ends. This ensures that the leakage of information from honest agents to attackers is minimized. The goal of static analysis is to make sure that number of correct opponent ends is zero, i.e., there is no leakage of information to the attacker.

The first major contribution of this paper is the definition of confidential safety of a protocol family using a novel view of correspondence assertions. While our precise statement is in the context of a first-order concurrent impera-

tive language, we feel that the form of our definitions can be easily adapted to different concurrent programming settings.

The second major contribution of this paper is an analysis to prove that a protocol is confidentially safe. Perforce, this analysis utilizes the specific infrastructure of our underlying programming language. The building block of our analysis is a type system developed along standard lines, blending elements from Abadi (1999) and Smith (2006). We provide proof-principles and sufficient conditions programs that are confidentially safe. The soundness of these proof principles are established using the non-interference properties satisfied by well-typed programs.

Rest of the paper. We informally develop our ideas in the next section. In Sections 4 and 6, we present the language and formally define confidential safety. We prove confidential safety for the mail example in Section 7. This proof relies on a typing system that establishes non-interference results. We describe the typing system in Section 5.

2 Informal examples

In this section, we illustrate the basic idea of confidential safety informally with the mail example.

Our programming model consists of first order imperative programs that communicate via a shared tuple space. The messages of honest agents indicate the intended receiver, and we assume that honest agents only take delivery of messages that are intended for them. We follow the standard network-as-opponent model. The adversary can view, duplicate, delete and create messages. We use symmetric key cryptography and assume a Dolev-Yao model for cryptography, rather than computational notions. An adversary can decrypt a ciphertext only if they have access to the encryption key.

In this example, Sam is the mail server. For each participating principal a , we assume there exists a shared key k_{AS} between a and Sam. Three additional principals are sufficient to illustrate the issue. Alice receives registrations from the other principals. In a separate thread, Alice nondeterministically chooses one of the registrants and sends a message to the chosen principal via Sam.

```
A: receive registrations
  | x = choose one of registered principals
    begin (A,x)
      out (x,msg-body) to S
B,C: register with A
```

The begin assertion has no operational effect. Rather, it indicates a confidentiality goal. The goal here is to hide the eventual recipient of the message.

A push mail server can be described as follows.

```
S: loop { inp (x,y) from z
          out (z,y) to x }
```

An opponent, Mallory, is able to deduce the recipient of the mail from Alice just by observing the target of the message from Sam. The system containing the code for the four principals thus violates confidential safety. This type of attack could be made much more difficult, if not impossible, by an onion-routing scheme in the network between Sam and the recipient of the message.

In our development of this example, we focus instead on a pull-based mail server; thus, each principal periodically queries Sam for mail. The first informal description of such a mail server follows.

```
S: loop { inp req from z
          case req of snd(x,y) => add (z,y) to mail[x]
                    rcv()    => out mail[x] to z }
```

The system containing this server is also not confidentially safe. The vulnerability arises from the ability of the Opponent to observe the contents of the message from Sam. Mallory can identify the recipient of Alice's message by looking for a non-empty response from the server for a `rcv` request, thus violating confidential safety.

The first attempt to fix this is to use cryptography to encrypt the message contents.

```
A: receive registrations
  | x = choose one of registered principals
  begin (A,x)
  out {snd(x,msg-body)}kAS to S
S: loop { inp req from z
          k = key for z
          case {req}k of snd(x,y) => add (z,y) to mail[x]
                        rcv()    => out {mail[x]}k to z }
```

This attempt still leaks information, but for more subtle reasons. The attack proceeds in two phases. (1) Mallory sends a `rcv` request to Sam, masquerading first as Bob and then as Charlie. This may be done in a run where these requests happen before Alice sends her first mail. Thus Mallory learns the value of the empty mail response when encrypted by Sam. (2) After this initial phase, Mallory is able to deduce when a `rcv` response from Sam is empty, by comparing the message to the ones previously captured.

This failure of confidential safety can be fixed by adding confounders (Abadi 1999) when encrypting.

```
S: loop { inp req from z
          k = key for z
```

```

case {req}k of snd(x,y) => add (z,y) to mail[x]
              rcv()    => (new n) out {mail[x],n}k to z

```

The key property that we demand is that with this use of confounders, no good process ever produces the same secret message twice. Thus, observing an encrypted message does not add to Mallory's ability to deduce facts about future messages. Later in the paper, we prove formally that this version of the protocol is indeed confidentially safe.

Variations that address hiding the identity of the sender of a message are also addressable in our framework, as are the protocols that aim to make both the sender and the receiver of a message non-guessable for Mallory.

In these examples, it is important that Alice's choice of recipient is nondeterministic; otherwise Mallory can predict Alice's choice. In our formal development, we do not include nondeterministic choice as an operator in the language. Instead we consider families (sets) of processes with similar behavior. To win, the opponent must be able to succeed against each member of the family. For example, we might have a family with two processes: one is the system in which Alice sends to Bob and the other is the system in which Alice sends to Charlie.

The confidentiality property that is declared in the code must include some information that the attacker cannot deduce. In the first examples above, confidentiality failed because too much information was sent in the clear. Confidentiality can also fail if the property is trivial. Consider a variant in which Alice sends mail to *all* registered participants:

```

A: receive registrations
  | for each x who is a registered principal
    begin (A,x)
      out {snd(x,msg-body)}kAS to S

```

This variant is not confidentially-safe. After Alice has sent a sufficient number of messages, Mallory can deduce that Alice has sent mail to each of the registered principals.

We develop our results using a concurrent imperative language. This allows us to easily treat programs which manipulate secret data and then perform public activities. Doing so in a process calculus requires explicit reasoning about the types of continuations. For simplicity, we also do not allow multiple threads to share the same store, and thus the code given above for Alice is not directly expressible. This technicality is easily relaxed at the expense of more baroque syntax or typing judgements.

3 Related work

Our definition of confidential safety is heavily influenced by the treatment of secrecy in adversarial systems by Halpern and O’Neill (2008) that builds on probabilistic non-interference Gray and Syverson (1992) and non-deducibility on strategies Wittbold and Johnson (1990). In particular, our basic definition can be seen as arising by eliding the probabilistic aspects of their analysis in Section 4.4 of Halpern and O’Neill (2008) on secrecy in adversarial systems. Rather than focussing on the secrecy of the choices made during the course of the entire strategy guiding an agent, our definitions focus on the secrecy of the initial choices made by the agent in the strategy.

The novelty of our work lies in the implementation of these ideas by combining a traditional type-based analysis of non-interference with a variant of correspondence assertions in the context of a specific programming language.

Correspondence assertions were introduced by Woo and Lam (1993). A static analysis to validate correspondence assertions was introduced in Gordon and Jeffrey (2003b). Honest agents do ends and these are statically validated by the effects in the type system that track, accumulate and discharge begins/ends. This led to type-based static validations of a variety of security protocols (Gordon and Jeffrey 2004, 2003a). As discussed in the introduction, in contrast to this line of work, we take a non-standard view of correspondence assertions. In our system, the adversary is the one trying to perform ends, whereas begins are used to indicate which pieces of information need to be non-guessable. In contrast to the methods used in the works cited above, our methods are influenced by those stemming from the study of information flow.

Information flow tries to make sure that untrusted programs do not leak information from high variables to low variables. In the classic framework of Denning, this is a two point lattice (Denning 1976). For a comprehensive survey of information flow research till 2003 see (Sabelfeld and Myers 2003). Volpano et al. (1996) initiated type-based analysis for information flow. Cryptography has been used to implement the program partitioning needed to realize information flow (Fournet et al. 2009).

Our type system builds on the secure information flow type system for singly-threaded imperative programs (Volpano et al. 1996), with cryptography (Smith 2006; Focardi and Centenaro 2008) and the spi-calculus type system for secrecy (Abadi 1999). Technically, our type system is very similar to that of Abadi (1999) and Focardi and Centenaro (2008). As in (Abadi 1999; Focardi and Centenaro 2008), our honest agents always encrypt using a confounder in order to disable implicit information flows that may arise because of unintended equality of ciphertexts. Thus, in contrast to Smith (2006) (but similar to (Abadi

1999; Focardi and Centenaro 2008)) we eschew the use of randomized cryptography and the associated computational/probabilistic considerations in the non-interference arguments.

4 A concurrent while language

We define the syntax and evaluation semantics for a simple concurrent while language. Several standard details are relegated to the appendix.

Syntax. We assume disjoint sets of variables, x, y, z , names, a, b, c, k, m, n references, p , and thread identifiers, t . We typically use names a, b, c for principals and k for keys.

Terms, commands and processes are defined as follows.

$M, N, L ::= m$	unit	(Base Val/Term)	$A, B ::= \text{skip}$	(Skip Cmd)
	(M, N)	(Product Val/Term)	$ A; B$	(Sequence Cmd)
	$\text{inl } M \mid \text{inr } M$	(Sum Val/Term)	$ p := M$	(Set Cmd)
	$\{M\}N$	(Cipher Val/Term)	$ \text{while } M A$	(While Cmd)
	x	(Variable Term)	$ \text{begin } M$	(Begin Cmd)
	p	(Reference Term)	$ \text{end } M$	(End Cmd)
	$M == N$	(Equals Term)	$ \text{out } M$	(Output Cmd)
	$\text{dec } M \text{ with } N$	(Decrypt Term)	$ \text{inp } (x); A$	(Input Cmd)
$P, Q ::= \mathbf{0}$		(Zero Proc)	$ \llbracket M \rrbracket N \text{ as } (x); A$	(Encrypt Cmd)
	$P \mid Q$	(Parallel Proc)	$ \text{split } M \text{ as } (x, y); A \dot{\wedge} B$	(Split Cmd)
	$(\nu m)P$	(New Proc)		
	$t.\Sigma/A$	(Thread Proc)	$ \text{case } M \text{ of } \text{inl } (x_1) => A_1$	
	$\text{begun } V$	(Begun Proc)	$\text{inr } (x_2) => A_2 \dot{\wedge} B$	(Case Cmd)
	$\text{ended } V$	(Ended Proc)		
	$\text{msg } V$	(Message Proc)		

A term is *closed* if it contains no variables and no references. A term is a *value*, V, W , if it is closed and contains no equality or decryption operators; that is, it can be constructed without the last four productions above.

In (Input Cmd) and (Encrypt Cmd), x is bound with scope A . In (Split Cmd), x and y are bound with scope A . In (Case Cmd), each x_i is bound with scope A_i . In (New Proc), m is bound with scope P .

For any syntax category, let fv return the set of free variables and let fn return the set of free names. We say that a variable, name or reference is *fresh* if it does not occur in the surrounding context, as understood by usage. We identify syntax up to renaming of bound variables and names and write $A\{V/x\}$ for the capture avoiding substitution of V for x in A . We assume similar notation for substitution of names for names and for substitution over other syntax categories.

A process P is *well formed* if no thread identifier occurs more than once in P . Henceforth we assume that all processes are well formed.

Evaluation. A *store* is a map from references to values. Let Σ range over stores. Write $\Sigma, p:V$ for store extension. The partial function $\Sigma(p)$ is defined to return the image of p in Σ , if it exists.

The evaluation relation on terms ($\Sigma/M \Downarrow V$) is defined in Figure 9, where $\Sigma/M \Downarrow$ is defined as $\Sigma/M \Downarrow V$ for some V .

The evaluation relation on processes ($P \rightarrow Q$) is defined in Figure 9. The definition uses contexts, which are defined as follows.

$$\mathbb{C} ::= [-] \mid (vm)\mathbb{C} \mid \mathbb{C} \mid P \mid P \mid \mathbb{C}$$

Let \rightarrow be the reflexive and transitive closure of $(\rightarrow) \cup (\equiv)$, where \equiv is a standard structural equivalence ($P \equiv Q$).

Discussion. Values include names, sums, products and ciphers. Terms additionally include destructors for equality tests and decryption. Evaluation of terms is total and side-effect free. Thus, decryption returns an option type. For example, encrypting V and then decrypting with the same key results in $\text{inl } V$.

Reference names are not values; they cannot be stored in a reference or communicated. Thread identifiers are not even terms; they are used only for bookkeeping.

The command language combines three standard sets of features. The first four command productions define a standard while language with assignment.

The final three productions in the syntax of commands include side effecting or partial constructors and destructors for the term language. The random encryption primitive creates a new confounder name, and so is included here. The destructors for products and sums may fail if the value has the wrong shape. We include a default case $\not\downarrow B$ so that execution does not get stuck.

The middle four productions, familiar from process calculi, provide correspondence assertions and inter-thread communication. The remainder of the process constructs can be found in the syntax of processes. For simplicity, we do not include commands to create new names or threads.

The *begin* and *end* commands simply accumulate as *begun* and *ended* processes; these have no operational affect, but are included to state expected invariants.

We use a single global channel, or tuple space, for asynchronous communication between processes. Output adds a value to the tuple space and input retrieves one. This communication model is very primitive. In examples, every message sent will include name for the sender and receiver, and good processes

will ignore messages that are not intended for themselves (they do this by simply doing an out with the message that they inadvertently received). We define a shorthand for this below. This process language is ill-suited for reasoning about progress, but is sufficient for reasoning about safety. Our results are robust and continue to hold in languages with stronger communication primitives. This communication model gives maximal power to opponents: an opponent may intercept any message and honest agents have no way to detect that a message has been intercepted.

Notational conventions. We adopt several notational conventions for examples. We use indentation and parentheses to group terms, commands and processes. We use tuples other than pairs.

We may write a random encryption command as a value. In this case $\dots \downarrow M \uparrow N \dots$ is implemented as $\downarrow M \uparrow N \text{ as } (x); \dots x \dots$, where x is fresh.

We write `true` for `inl unit` and `false` for `inr unit`. We write `if M then A1 else A2 ↯ B` to abbreviate `case M inl(x1) => A1 inr(x2) => A2 ↯ B` when neither x_i occurs in A_i .

We use lists, which can be defined using recursive sum and product types. We write `nil` for the empty list, and $M :: N$ for the list with head M and tail N .

We treat lists of pairs as maps. We write $M[N]$ for the map lookup function, which returns `inl(L)` if M is a list containing the pair (N, L) , and which returns `false` otherwise. We write $M[N \mapsto L]$ for the map update function, which returns a new map which is like M except for the pair with first element N , whose second element is now L .

We use pattern matching syntax for case constructs and for input. For example, `inp(m, x); A` is shorthand for the following, where y and z do not occur free in A and p is fresh (ie, does not appear in A or the surrounding context).

```

p:=true;
while p { inp(y);
  split y as (z,x);
  if z==m then p:=false; A
  else out y;
  ↯ out y }

```

The message is placed back on the network if it is not a pair whose first component is the name m .

We drop default cases that do nothing. That is, we write `split M as (x,y); A` to abbreviate `split M as (x,y); A ↯ skip`, and similarly for case and if.

We drop uninteresting thread identifiers and empty stores from threads, writing $A \mid \mathbf{0}$, as shorthand for $(t.\emptyset/A) \mid \mathbf{0}$.

5 Typing for Non-Interference

In this section we define a type system for secure information flow in the presence of encryption, and establish a non-interference result for well-typed processes.

Value types include secret types and the untrusted type Un . Values at secret types cannot be sent over the untrusted network directly. However, encryption yields ciphertext of an untrusted type Un that can be sent over the untrusted network.

The syntax of secret types, σ, ρ , types, τ , and typing environments, E , is as follows.

$$\begin{array}{ll}
 \sigma, \rho ::= SecretUn & \text{(Secret Untrusted Type)} \\
 | Key \sigma & \text{(Key Type)} \\
 | \sigma + \rho & \text{(Sum Type)} \\
 | \sigma \times \rho & \text{(Product Type)} \\
 \tau ::= \sigma | Un & \text{(Type)} \\
 E ::= \emptyset | E, x: \tau | E, m: \tau | E, t.p: \tau & \text{(Environment)}
 \end{array}$$

The type $Key \sigma$ is used to encrypt plaintext of type σ . The type $SecretUn$ is used to type terms that are untrusted but may be tainted by the information flow analysis. Sum and product types may include any secret type.

We only consider environments E where there is at most one occurrence of each variable, reference, or name. We treat the environment as unordered, and as a function from variables, names, and reference/thread pairs to types.

The judgement $E \vdash_t M : \tau$ states that a term M has type τ at thread t in environment E . When M contains no references, the thread t is irrelevant; in this case, we write $E \vdash M : \tau$ as a shorthand for $(\exists t) E \vdash_t M : \tau$.

The term typing rules are given in Figure 2. The typing rules can be broadly classified into two groups: rules for data at secret types τ ; and rules for data at the untrusted type Un .

The rules for data at secret types ensure that if any data at a secret type is read in a term, then the term's type is also secret. In particular, decryption with a secret key yields a sum type, which is in turn a secret, because successful decryption potentially leaks information about which secret key is used for decryption. Encryption with a secret key of type $Key \sigma$ requires plaintext of type $\sigma \times SecretUn$, where the $SecretUn$ component is the confounder.

The Un rules are used for handling non-secret data and for typing arbitrary opponent processes. A term of type Un can only read data from Un sources. The type Un cannot occur as a component of a secret type. For example, $Un \times Un$ is

not syntactically valid, so pairing of data at Un results in the type Un . However, a Un may be coerced to a $SecretUn$, turning it into a secret. This is important because the secret type σ for the plaintext in $Key \sigma$ does not range over Un , and so data of type Un must be coerced to $SecretUn$ before it is encrypted. This means that upon decryption, the resulting plaintext will still be at $SecretUn$, and can no longer be recovered at Un . Consequently, Un values cannot be encrypted with a secret key, transmitted across the insecure network, and recovered at Un after decryption.

This partitioning of Un from secret types is stricter than other spi-calculus typing systems, such as (Abadi 1999).

We now turn to typing commands. Following the approach of (Volpano et al. 1996) for typing imperative programs, we assign a level to a well-typed command. Command types γ are as follows.

$$\gamma ::= CmdUn \mid CmdSec \quad (\text{Command Type})$$

The type $CmdUn$ represents commands that may have written to references of type Un and $CmdSec$ represents commands that have only written to non- Un types. The command, store and process typing judgements are given in Figures 3 and 1.

As in (Volpano et al. 1996), the application of a branching construct for while loops, splitting a pair, or case analysis of a sum to a term of secret type requires the commands in each branch to be typed at level Sec . This prevents assignment to references of type Un . In addition, input and output commands are typed at level Un , so input and output cannot occur in branches that depend upon secret data.

Our command typing differs from the type system for a multi-threaded imperative language in (Smith and Volpano 1998) in that we permit while loops with secret guards, albeit with a constraint that the loop must always terminate.

The typing rule for `begin` is more restrictive than required for the results of this section. It is explained at the beginning of Section 6.

The following lemma guarantees that data used as a secret key is kept partitioned from untrusted data that may be known by the opponent. This property is key to establishing non-interference.

Lemma 3. *If $E \vdash_1 M : Un$ and $E \vdash_1 N : Key \sigma$ then $M \neq N$.* □

Finally, we note that processes using Un data can always be typechecked, yielding an opponent typability property.

Lemma 4 (Opponent Typability). *If the free variables, names, and reference/thread pairs of a process P are in the domain of an environment E , and the range of E is $\{Un\}$, then $E \vdash P$.* □

Non-Interference. We now establish a non-interference result that is in turn used to establish confidential safety in the sequel.

We handle non-interference in an imperative setting with ciphertext using an adaptation of the theory of framed bisimulation (Abadi and Gordon 1998). Framed bisimulation provides a sound technique for reasoning about equivalence of spi processes. The theory of a frame represents ciphertext released to an opponent, or at Un in our setting. Theories are ordered sequences of triples, defined by:

$$\theta ::= \emptyset \mid \theta, (\{V\}V', \{W\}W', \sigma)$$

Each component of the theory is required to be ciphertext, arising from encryption with secret keys. In order for a theory to be well-formed, the left hand ciphertext may not appear elsewhere in the theory paired with anything other than the right hand ciphertext (similarly for the right hand ciphertext). This is formalized as the judgment $E; \theta \vdash \diamond$. The definition is a mutual induction with the non-interference relation between terms, defined next.

(EMPTY WF THEORY)

$$\frac{}{E; \emptyset \vdash \diamond}$$

(NON-EMPTY WF THEORY)

$\{V_1\}V_2$ does not occur as LHS in θ

$\{V'_1\}V'_2$ does not occur as RHS in θ

$$\frac{E; \theta \vdash V_1 \sim V'_1 : \sigma \times SecretUn \quad E; \theta \vdash V_1 \sim V'_2 : Key \sigma}{E; \theta, (\{V_1\}V_2, \{V'_1\}V'_2, \sigma) \vdash \diamond}$$

The judgement $E; \theta \vdash_7 M \sim M' : \tau$, defined in Figure 4, states that terms are related by the non-interference relation \sim in the environment E and theory θ , where both terms have type τ at thread θ . When M and M' have no references, the thread θ is irrelevant, and we write $E; \theta \vdash M \sim M' : \tau$ as a shorthand for $(\exists \theta)E; \theta \vdash_7 M \sim M' : \tau$.

Roughly, the non-interference relation relates either two arbitrary terms of the same secret type σ , or two terms of Un if they are identical except for ciphertext subterms that are encrypted with a secret key. Ciphertext subterms are required to be declared in the theory θ to ensure consistent usage throughout a process, without leaking information that can be discovered by an opponent that compares ciphertexts seen on the insecure network.

Intuitively, stores are related by the non-interference relation if they only differ in the values at secret types, i.e., references of Un type have identical values. The rules are defined in Figure 6.

The rules for commands in Figure 5 are similar to those of terms in the sense that arbitrary well-typed commands with level Sec are related, and can

subsequently be coerced to be related at Un . The encryption command does not reference the theory θ because of its evaluation behavior which generates a fresh confounder. The remaining rules are for Un commands and require syntactic identity for outermost constructors. The rules for processes in Figure 6 are similar and straightforward.

Lemma 5 (Opponent Reflexivity). *If the free variables, names, and reference/thread pairs of a process P are in the domain of an environment E , and the range of E is $\{Un\}$, then $E; \emptyset \vdash P \sim P$.* \square

Lemma 6 (Initial Process Reflexivity). *If $E \vdash P$ and P contains no existing ciphertext in terms then $E; \emptyset \vdash P \sim P$.* \square

Note that substitution into the theory is not necessary because the theory has no free variables (only triples with ciphertext values).

Lemma 7 (Substitution). *If $E; \theta \vdash V \sim V' : \tau$ and $E, x: \tau; \theta \vdash P \sim P'$ then $E; \theta \vdash P\{V/x\} \sim P'\{V'/x\}$.* \square

In proving non-interference we use that evaluation of terms never gets stuck and always terminates (a property of the operational semantics due to default branches), and that evaluation of commands of type Sec never gets stuck and always terminates in $skip$ because while loops at Sec are constrained to terminate.

The non-interference theorem states that evaluation preserves \sim relationships. Since non-interference on processes permits the stores of each command to be related by \sim (as opposed to being syntactically identical) this result establishes non-interference for processes that differ only in the values of store references with non- Un type. In particular, this ensures that using differing values in references of secret key type will not lead to observable differences in references of Un type.

Theorem 8 (Non-Interference). *Suppose $dom(E)$ contains no variables. If*

- $E; \theta \vdash P \sim Q$, and
- $P \twoheadrightarrow (v\vec{m})P'$,

then there exist an environment E' such that $dom(E')$ contains no variables, a theory θ' , a process Q' and names \vec{n} such that

- $E, E'; \theta, \theta' \vdash P' \sim Q'$, and
- $Q \twoheadrightarrow (v\vec{n})Q'$. \square

6 Confidential safety

In this section we define confidential safety and illustrate its properties via several examples.

Write $P \in \mathbb{C}$ to mean $\mathbb{C}[\mathbf{0}] \equiv \mathbb{C}'[P]$ for some \mathbb{C}' .

Definition 9 (Safety). A program P is *safe* if whenever $P \rightarrow \mathbb{C}[\text{ended } V]$ then $\text{begun } V \in \mathbb{C}$. \square

Define function *erase* on commands to replace `begin` and `end` subcommands by `skip`. Define function $\text{erase}(P)$ on processes to erase commands and to replace `begun` and `ended` subprocesses by $\mathbf{0}$. So for example, $\text{erase}(\text{begun } m \mid \text{skip}; \text{end } m) = \mathbf{0} \mid \text{skip}; \text{skip}$.

Let \mathcal{P} range over sets (families) of processes. An *opponent* is a process that is typed in an environment where all identifiers are assigned type Un .

Definition 10 (Confidential safety). A process family \mathcal{P} is *confidentially safe for O* if whenever $P \mid O$ is safe for every $P \in \mathcal{P}$, then $\text{erase}(P) \mid O$ is safe for every $P \in \mathcal{P}$.

A process family \mathcal{P} is *confidentially safe (CS)* if it is confidentially safe for every begin-free opponent. \square

Intuitively, the definition says that the justification for an opponent end must not come from the process family. We say that a process is *begin-free* if it contains no `begin` or `begun`, and similarly for *end-free* processes. Note that any begin-free, safe opponent can never reach an end. For end-free process families, this leads to a simple alternative characterization.

Definition 11. A begin-free opponent O is *successful* against an end-free family \mathcal{P} if $P \mid O$ is safe for every $P \in \mathcal{P}$ and $P \mid O \rightarrow \mathbb{C}[\text{ended } V]$ for some $P \in \mathcal{P}$ and some \mathbb{C} and V . \square

Lemma 12. An end-free family \mathcal{P} is CS iff it has no successful opponents. \square

Example 13. (1) Any family of processes in which no process performs a `begin` is CS; this includes the empty set.

(2) The singleton family `begin a` is not CS. For example, the opponent `end a` is safe when run against `begin a`, but not when run against $\text{erase}(\text{begin a}) = \text{skip}$.

(3) The singleton family `begin a` \mid `begin b` is also not CS, as demonstrated by the successful opponents `end a`, `end b`, and `end a` \mid `end b`.

The first example above is unsurprising since the family claims no confidentiality. The second and third examples are also unsurprising, since these families produce the same begins in all runs. \square

In the definition of CS, the process family allows nondeterminism outside the system via initial conditions which are unknown to the opponent.

Example 14. The family consisting of the two process $\text{begin } a$ and $\text{begin } b$ is CS. This family has no successful opponents. Different begins occur in different runs, yet the opponent has no way to determine which begin has occurred. If our language included nondeterministic choice³ the singleton family $\text{begin } a \oplus \text{begin } b$ would be CS as well. \square

Confidentiality requires that there be something that the opponent cannot predict. In addition, it requires that this secret be maintained. In all of the examples so far, secrecy is trivial, since the processes communicate nothing, and thus the attacker has no information upon which to base an end. Moving toward more realistic examples, let us consider processes that announce their begin. This is similar to communication protocols in which the recipient of a message must be sent in plaintext.

Example 15. The family comprising the two processes $\text{begin } a; \text{ out } a$ and $\text{begin } b; \text{ out } b$ is not CS, as evidenced by the successful opponent is $\text{in } (x); \text{ end } x$. Assuming nondeterministic choice, the singleton $\text{begin } a; \text{ out } a \oplus \text{begin } b; \text{ out } b$ is also not CS. \square

In this example, we regain CS by adding encryption with a secret key.

Example 16. The family comprising the two processes $(vk) \text{ begin } a; \text{ out } \{a\}k$ and $(vk) \text{ begin } b; \text{ out } \{b\}k$ is CS. \square

In this case, encryption with a fresh key ensures that the attacker cannot distinguish the message containing a from that containing b . Frame bisimulation (Abadi and Gordon 1998) provides a general framework for reasoning about equivalence of processes using cryptography.

Although $(vk) \text{ out } \{a\}k$ is frame bisimilar to $(vk) \text{ out } \{b\}k$, one must be careful if messages may be sent more than once. $(vk) \text{ out } \{a\}k \text{ out } \{a\}k$ is not frame bisimilar to $(vk) \text{ out } \{a\}k \text{ out } \{b\}k$.

Example 17. The following variant family is not CS. The first process is $(vk) \text{ begin } a; \text{ out } \{a\}k \text{ out } \{b\}k$. The second process is $(vk) \text{ begin } b; \text{ out } \{b\}k \text{ out } \{b\}k$. A successful opponent need only receive all messages and then check for equality. If the two messages sent are identical, then the opponent can end a , otherwise the opponent can end b . \square

³ The standard encoding of $A \oplus B$ is $(vm)(\text{msg } m \mid \text{inp } (m) A \mid \text{inp } (m) B)$, where m is fresh. The encoding uses pattern matching, as described in Section 4. In isolation, this performs nondeterministic choice, but an opponent can easily disrupt this interpretation. When placed in parallel with $\text{inp } (x); \text{ out } x; \text{ out } x$, the process may perform both A and B .

To ensure that terms with different secrets are frame bisimilar, we require confounders (Abadi 1999).

In general, CS requires reasoning about information flow. In Section 7 we present examples using the type system for establishing non-interference described in Section 5.

7 Proving confidential safety

To establish that a process family is CS, it is not sufficient to show that the members of the family are pairwise related by the non-interference relation \sim . In addition, one must show that the begins performed by the processes differ using a secret. To simplify the presentation, we fix the location of the secret, insisting that every typable begin be a pair, whose first component has a secret type.

We say that two processes have *strongly different begins* if whenever $P \rightarrow (v\vec{m})(\text{begun } (m, V) \mid P')$ and $Q \rightarrow (v\vec{n})(\text{begun } (n, W) \mid Q')$ then $m \notin \vec{m}$ and $n \notin \vec{n}$ and $m \neq n$. Note that the typing rules guarantee that begun-free processes with strongly different begins will differ in a secret value.

Corollary 18. *A process family \mathcal{P} is confidentially safe if*

- for every $\{P, Q\} \subseteq \mathcal{P}$ we have $E; \emptyset \vdash P \sim Q$ for some E such that $\text{dom}(E)$ that contains no variables, and
- there exists $\{P, Q\} \subseteq \mathcal{P}$ such that P and Q contain no begun subprocesses and P and Q have strongly different begins.

PROOF. Noninterference guarantees that an opponent cannot distinguish members of the family. The fact that family members have strongly different begins ensures that any opponent that is successful against P will fail against Q (and vica versa). \square

We first consider a simple mail server with one sender and two possible recipients. The goal is to keep the message recipient confidential.

To improve readability, we use some notational conventions. We underline keywords and reserved words, write parameters in italics and write program variables in slanted font. To make patterns more readable, we use some shorthand. For requests sent to the server, let “get” represent “inl unit”, and let “snd x ” represent “inr x ”. For acknowledgements sent from the server, let “ok” represent “inl unit”, and let “mail x ” represent “inr x ”.

Server(s) is defined as follows, where the parameter s is the identity of the server. The definition uses three references. *keys* is a map from principal names

to encryption keys. msgs is a map from principal names to lists of messages. ack is the acknowledgement to be sent.

The parameter s is public, as is the variable x . The three references and all other variables are given secret types. The code sends an acknowledgement to every request, regardless of the result. The only useful information in an acknowledgement, then, is in the payload, which is encrypted using a secret key.

$$\text{Server}(s) \triangleq \text{msgs} := \emptyset;$$

$$\text{while true } \{ \text{inp } (x, s, \text{ctext});$$

$$\text{case keys}[x] \text{ of}$$

$$\text{false} \Rightarrow \text{skip}$$

$$\text{inl } k \Rightarrow \text{case } (\text{dec } \text{ctext with } k) \text{ of}$$

$$(\text{get}, _) \Rightarrow \text{ack} := \text{mail } \text{msgs}[x];$$

$$\text{msgs} := \text{msgs}[x \mapsto \text{nil}]$$

$$(\text{snd}(r, m), _) \Rightarrow \text{ack} := \text{ok};$$

$$\text{msgs} := \text{msgs}[r \mapsto [(x, m)]]$$

$$\text{out } (s, x, \text{ack}) \} (\text{keys}[x]) \}$$

The loop guard has type Un and thus the loop has type $CmdUn$. In the body of the loop, the decryption result is secret, and so the case over its contents must be typed as $CmdSec$. The case assigns the secret reference ack , which is then encrypted and sent out on the net using out . It is crucial both that the assignment to ack be secret and that the encrypted output be public. The output must occur outside the case construct. The typing rules allow $CmdSec; CmdUn$ to be treated as $CmdUn$. This is sound in our setting. Getting a similar result in a language such as the spi calculus requires explicit typing rules for continuations.

Define the simple mail system as

$$\text{System}(x) \triangleq (\nu k_{as})(\nu k_{bs})(\nu k_{cs}) \text{ keys}: [(a, k_{as}), (b, k_{bs}), (c, k_{cs})] / \text{Server}(s)$$

$$| \text{r}:x, m:m / \text{Sender}(a, s, k_{as})$$

$$| \text{Receiver}(b, s, k_{bs})$$

$$| \text{Receiver}(c, s, k_{cs})$$

where Sender and Receiver are defined below.

$\text{Sender}(a, s, k_{as})$ is defined as follows, where a is the name of the sender, s is the name of the server, and k_{as} is a secret key known only to the server and sender. a and s are public, whereas k_{as} is secret. The secret references m and r hold the message contents and the name of the recipient.

$$\text{Sender}(a, s, k_{as}) \triangleq \text{begin } (r, a, m);$$

$$\text{out } (a, s, \text{snd}(r, m)) \text{ } k_{as};$$

$$\text{inp } (s, a, \text{ctext});$$

$$\text{case } (\text{dec } \text{ctext of } k_{as}) \text{ of } (\text{ok}, _) \Rightarrow \text{skip}$$

$$(\text{mail } _, _) \Rightarrow \text{error}$$

$\text{Receiver}(r, s, k_{rs})$ is defined as follows, where r is the name of the receiver, s is the name of the server, and k_{rs} is a secret key known only to the server and receiver. r and s are public, whereas k_{rs} is secret.

$$\begin{aligned}
Receiver(r, s, k_{rs}) \triangleq & \text{ while true } \{ \text{ out } (r, s, \text{!get } \!k_{rs}); \\
& \text{ inp } (s, r, \text{c\textit{t}ext}); \\
& \text{ case (dec c\textit{t}ext of } k_{rs}) \text{ of} \\
& \quad (\text{ok, } _) \Rightarrow \text{ error} \\
& \quad (\text{mail } ms, _) \Rightarrow \\
& \quad \quad \text{ case } ms \text{ of } \underline{\text{nil}} \Rightarrow \text{ skip} \\
& \quad \quad [(x, m)] \Rightarrow \text{ skip //end}(r, x, m) \}
\end{aligned}$$

The receiver can safely perform and end in the case that they receive mail. For simplicity, we do not track legal effects in our type system, and thus the end will not type here. Allowing such programs to type is straightforward, however, following Gordon and Jeffrey (2004).

Proposition 19. *The process family $\{System(b), System(c)\}$ is CS.*

PROOF. The result follows from Corollary 18, since the processes have strongly different begins and $System(b) \sim System(c)$, with r typed as *SecretUn*. \square

8 Conclusion

We have studied a notion of secrecy that arises naturally in adversarial systems. Our notion of secrecy aims to capture the idea that the choice of a value from a globally known collection of values by an honest agent is unguessable by an adversary, even in the context of a distributed protocol. We called this notion confidential safety and provided a formal definition using a novel variation of correspondence assertions. We provided means to establish this property using a mixture of information flow and frame-bisimulation techniques.

In future work, rather than demanding guaranteed success, we intend to explore probabilistic settings to formalize the possibilities of the adversary succeeding with a reasonable chance of success. Such an extension would incorporate probabilistic adversaries and enable us to consider computational-style analysis that incorporate realistic models of randomized cryptography and resource bounds on the computation performed by the adversary.

Bibliography

- M. Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, 1999.
- M. Abadi and A. D. Gordon. A bisimulation method for cryptographic protocols. *Nordic J. of Computing*, 5(4):267–303, 1998.
- D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

- R. Focardi and M. Centenaro. Information flow security of multi-threaded distributed programs. In *PLAS*, pages 113–124. ACM, 2008.
- C. Fournet, G. L. Guernic, and T. Rezk. A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In *CCS*, pages 432–441. ACM, 2009.
- A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–520, 2003a.
- A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3-4):435–483, 2004.
- A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theor. Comput. Sci.*, 300(1-3):379–409, 2003b.
- I. Gray, J.W. and P. Syverson. A logical approach to multilevel security of probabilistic systems. pages 164–176, may 1992.
- J. Y. Halpern and K. R. O’Neill. Secrecy in multiagent systems. *ACM Trans. Inf. Syst. Secur.*, 12(1), 2008.
- A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
- G. Smith. Secure information flow with random assignment and encryption. In *FMSE*, pages 33–44. ACM, 2006.
- G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, pages 355–364. ACM Press, 1998.
- D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
- J. T. Wittbold and D. M. Johnson. Information flow in nondeterministic systems. *Security and Privacy, IEEE Symposium on*, 0:144, 1990. ISSN 1540-7993. doi: <http://doi.ieeecomputersociety.org/10.1109/RISP.1990.63846>.
- T. Y. C. Woo and S. S. Lam. A framework for distributed authorization. In *CCS*, pages 112–118, 1993.

$$\begin{array}{c}
\text{(STORE)} \\
\frac{\forall p \in \text{dom}(\Sigma) \exists \tau. E(t.p) = \tau \text{ and } E \vdash \Sigma(p) : \tau}{E \vdash \Sigma} \\
\begin{array}{cccccc}
\text{(CMD PRC)} & \text{(MSG PRC)} & \text{(BEGUN PRC)} & \text{(ENDED PRC)} & \text{(NEW PRC)} & \text{(PAR PRC)} \\
\frac{\text{(ZERO PRC)} \quad E \vdash \Sigma \quad E \vdash A : \gamma}{E \vdash \mathbf{0}} & \frac{E \vdash V : \text{Un}}{E \vdash \text{msg } V} & \frac{E \vdash V : \text{Un}}{E \vdash \text{begun } V} & \frac{E \vdash V : \text{Un}}{E \vdash \text{ended } V} & \frac{E, m : \tau \vdash P}{E \vdash (vm)P} & \frac{E \vdash P \quad E \vdash Q}{E \vdash P \mid Q}
\end{array}
\end{array}$$

Fig. 1. Store and Process Typing $(E \vdash \Sigma)$ $(E \vdash P)$

(COERCE UN TERM)	(VAR TERM)(NAME TERM)	(REF TERM)	(LEFT TERM)	(RIGHT TERM)
$\frac{E \vdash M : Un}{E \vdash M : SecretUn}$	$\frac{E(x) = \tau \quad E(m) = \tau}{E \vdash x : \tau \quad E \vdash m : \tau}$	$\frac{E(t.p) = \tau}{E \vdash p : \tau}$	$\frac{E \vdash M : \sigma}{E \vdash \text{inl } M : \sigma + \rho}$	$\frac{E \vdash M : \rho}{E \vdash \text{inr } M : \sigma + \rho}$
(UN LEFT TERM)	(UN RIGHT TERM)	(PRODUCT TERM)	(UN PRODUCT TERM)	
$\frac{E \vdash M : Un}{E \vdash \text{inl } M : Un}$	$\frac{E \vdash M : Un}{E \vdash \text{inr } M : Un}$	$\frac{E \vdash M : \sigma \quad E \vdash N : \rho}{E \vdash (M, N) : \sigma \times \rho}$	$\frac{E \vdash M : Un \quad E \vdash N : Un}{E \vdash (M, N) : Un}$	
(CIPHER TERM)	(UN CIPHER TERM)	(EQUALS TERM)	(UN EQUALS TERM)	
$\frac{E \vdash M : \sigma \times SecretUn \quad E \vdash N : Key \sigma}{E \vdash \{M\}N : Un}$	$\frac{E \vdash M : Un \quad E \vdash N : Un}{E \vdash \{M\}N : Un}$	$\frac{E \vdash M : \sigma \quad E \vdash N : \sigma}{E \vdash M = N : Bool}$	$\frac{E \vdash M : Un \quad E \vdash N : Un}{E \vdash M = N : Un}$	
(DECRYPT TERM)	(UN DECRYPT TERM)			
$\frac{E \vdash M : SecretUn \quad E \vdash N : Key \sigma}{E \vdash \text{dec } M \text{ with } N : (\sigma \times SecretUn) + SecretUn}$	$\frac{E \vdash M : Un \quad E \vdash N : Un}{E \vdash \text{dec } M \text{ with } N : Un}$			

Fig. 2. Term Typing ($E \vdash M : \tau$)

(COERCE SEC CMD)	(SEQUENCE CMD)	(ASSIGN CMD)	(UN ASSIGN CMD)
$\frac{E \vdash A : CmdSec}{E \vdash A : CmdUn}$	$\frac{E \vdash A : \gamma \quad E \vdash B : \gamma}{E \vdash A ; B : \gamma}$	$\frac{E(t.p) = \sigma \quad E \vdash M : \sigma}{E \vdash p := M : CmdSec}$	$\frac{E(t.p) = Un \quad E \vdash M : Un}{E \vdash p := M : CmdUn}$
(SKIP CMD)	(WHILE CMD)	(UN WHILE CMD)	(BEGIN CMD)
$\frac{E \vdash M : Bool \quad E \vdash A : CmdSec}{E \vdash \text{while } M A : CmdSec}$	$\frac{E \vdash M : Un \quad E \vdash A : CmdUn}{E \vdash \text{while } M A : CmdUn}$	$\frac{E \vdash M : \sigma \quad E \vdash N : \tau}{E \vdash \text{begin } (M, N) : \gamma}$	$\frac{E \vdash M : \tau}{E \vdash \text{end } M : \gamma}$
(ENCRYPT CMD)	(UN ENCRYPT CMD)		
$\frac{E \vdash M : \sigma \quad E \vdash N : Key \sigma}{E \vdash \llbracket M \rrbracket N \text{ as } (x); A : \gamma}$	$\frac{E, x : Un \vdash A : \gamma \quad E \vdash M : Un \quad E \vdash N : Un}{E \vdash \llbracket M \rrbracket N \text{ as } (x); A : \gamma}$		
(SPLIT CMD)	(UN SPLIT CMD)		
$\frac{E \vdash M : \sigma \times \rho \quad E, x : \sigma, y : \rho \vdash A : CmdSec \quad E \vdash B : CmdSec}{E \vdash \text{split } M \text{ as } (x, y); A \zeta B : CmdSec}$	$\frac{E \vdash M : Un \quad E, x : Un, y : Un \vdash A : \gamma \quad E \vdash B : \gamma}{E \vdash \text{split } M \text{ as } (x, y); A \zeta B : \gamma}$		
(CASE CMD)	(UN CASE CMD)		
$\frac{E \vdash M : \sigma + \rho \quad E, x_1 : \sigma \vdash A_1 : CmdSec \quad E, x_2 : \rho \vdash A_2 : CmdSec \quad E \vdash B : CmdSec}{E \vdash \text{case } M \text{ of inl } (x_1) \Rightarrow A_1 \text{ inr } (x_2) \Rightarrow A_2 \zeta B : CmdSec}$	$\frac{E \vdash M : Un \quad E, x_1 : Un \vdash A_1 : \gamma \quad E, x_2 : Un \vdash A_2 : \gamma \quad E \vdash B : \gamma}{E \vdash \text{case } M \text{ of inl } (x_1) \Rightarrow A_1 \text{ inr } (x_2) \Rightarrow A_2 \zeta B : \gamma}$		

Fig. 3. Command Typing ($E \vdash A : \gamma$)

$$\begin{array}{c}
\text{(SEC TERM REL)} \quad \text{(UN VAR REL)} \quad \text{(UN NAME REL)} \quad \text{(UN REF REL)} \\
\frac{E; \theta \vdash \diamond \quad E \Vdash M : \sigma \quad E \Vdash M' : \sigma}{E; \theta \Vdash M \sim M' : \sigma} \quad \frac{E(x) = Un \quad E; \theta \vdash \diamond}{E; \theta \Vdash x \sim x : Un} \quad \frac{E(m) = Un \quad E; \theta \vdash \diamond}{E; \theta \Vdash m \sim m : Un} \quad \frac{E(t.p) = Un \quad E; \theta \vdash \diamond}{E; \theta \Vdash p \sim p : Un} \\
\text{(UN LEFT REL)} \quad \text{(UN RIGHT REL)} \quad \text{(UN PRODUCT REL)} \\
\frac{E; \theta \Vdash M \sim M' : Un}{E; \theta \Vdash \text{inl } M \sim \text{inl } M' : Un} \quad \frac{E; \theta \Vdash M \sim M' : Un}{E; \theta \Vdash \text{inr } M \sim \text{inr } M' : Un} \quad \frac{E; \theta \Vdash M \sim M' : Un \quad E; \theta \Vdash N \sim N' : Un}{E; \theta \Vdash (M, N) \sim (M', N') : Un} \\
\text{(CIPHER REL)} \quad \text{(UN CIPHER REL)} \\
\frac{(\{M\}N, \{M'\}N', \sigma) \in \theta \quad E; \theta \vdash \diamond \quad E; \theta \Vdash M \sim M' : Un \quad E; \theta \Vdash N \sim N' : Un}{E; \theta \Vdash \{M\}N \sim \{M'\}N' : Un} \quad \frac{E; \theta \Vdash M \sim M' : Un \quad E; \theta \Vdash N \sim N' : Un}{E; \theta \Vdash \{M\}N \sim \{M'\}N' : Un} \\
\text{(UN EQUALS REL)} \quad \text{(UN DECRYPT REL)} \\
\frac{E; \theta \Vdash M \sim M' : Un \quad E; \theta \Vdash N \sim N' : Un}{E; \theta \Vdash M = N \sim M' = N' : Un} \quad \frac{E; \theta \Vdash M \sim M' : Un \quad E; \theta \Vdash N \sim N' : Un}{E; \theta \Vdash \text{dec } M \text{ with } N \sim \text{dec } M' \text{ with } N' : Un}
\end{array}$$

Fig. 4. Term Equivalence ($E; \theta \Vdash M \sim M' : \tau$)

$$\begin{array}{c}
\text{(SEC REL)} \quad \text{(COERCE SEC REL)} \\
\frac{E; \theta \vdash \diamond \quad E \Vdash A : \text{CmdSec} \quad E \Vdash A' : \text{CmdSec}}{E; \theta \Vdash A \sim A' : \text{CmdSec}} \quad \frac{E; \theta \Vdash A \sim A' : \text{CmdSec}}{E; \theta \Vdash A \sim A' : \text{CmdUn}} \\
\text{(UN SEQUENCE REL)} \quad \text{(UN ASSIGN REL)} \\
\frac{E; \theta \Vdash A \sim A' : \text{CmdUn} \quad E; \theta \Vdash B \sim B' : \text{CmdUn}}{E; \theta \Vdash A; B \sim A'; B' : \text{CmdUn}} \quad \frac{E(t.p) = Un \quad E; \theta \Vdash M \sim M' : \text{CmdUn}}{E; \theta \Vdash p := M \sim p := M' : \text{CmdUn}} \\
\text{(UN WHILE REL)} \quad \text{(UN BEGIN REL)} \quad \text{(UN END REL)} \\
\frac{E; \theta \Vdash M \sim M' : Un \quad E; \theta \Vdash A \sim A' : \text{CmdUn}}{E; \theta \Vdash \text{while } M A \sim \text{while } M' A' : \text{CmdUn}} \quad \frac{E; \theta \Vdash M \sim M' : \tau}{E; \theta \Vdash \text{begin } M \sim \text{begin } M' : \text{CmdUn}} \quad \frac{E; \theta \Vdash M \sim M' : \tau}{E; \theta \Vdash \text{end } M \sim \text{end } M' : \text{CmdUn}} \\
\text{(UN OUT REL)} \quad \text{(UN INP REL)} \\
\frac{E; \theta \Vdash M \sim M' : Un}{E; \theta \Vdash \text{out } M \sim \text{out } M' : \text{CmdUn}} \quad \frac{E, x:Un; \theta \Vdash A \sim A' : \text{CmdUn}}{E; \theta \Vdash \text{inp } (x); A \sim \text{inp } (x); A' : \text{CmdUn}} \\
\text{(ENCRYPT REL)} \\
\frac{E; \theta \Vdash M \sim M' : \sigma \quad E; \theta \Vdash N \sim N' : \text{Key } \sigma \quad E, x:Un; \theta \Vdash A \sim A' : \text{CmdUn}}{E; \theta \Vdash \mathbb{M} N \text{ as } (x); A \sim \mathbb{M}' N' \text{ as } (x); A' : \text{CmdUn}} \\
\text{(UN ENCRYPT REL)} \\
\frac{E; \theta \Vdash M \sim M' : Un \quad E; \theta \Vdash N \sim N' : Un \quad E, x:Un; \theta \Vdash A \sim A' : \text{CmdUn}}{E; \theta \Vdash \mathbb{M} N \text{ as } (x); A \sim \mathbb{M}' N' \text{ as } (x); A' : \text{CmdUn}} \\
\text{(UN SPLIT REL)} \\
\frac{E; \theta \Vdash M \sim M' : Un \quad E, x:Un, y:Un; \theta \Vdash A \sim A' : \text{CmdUn} \quad E; \theta \Vdash B \sim B' : \text{CmdUn}}{E; \theta \Vdash \text{split } M \text{ as } (x, y); A \mathbb{Z} B \sim \text{split } M' \text{ as } (x, y); A' \mathbb{Z} B' : \text{CmdUn}} \\
\text{(UN CASE REL)} \\
\frac{E; \theta \Vdash M \sim M' : Un \quad E, x_1:Un; \theta \Vdash A_1 \sim A'_1 : \text{CmdUn} \quad E, x_2:Un; \theta \Vdash A_2 \sim A'_2 : \text{CmdUn} \quad E; \theta \Vdash B \sim B' : \text{CmdUn}}{E; \theta \Vdash \text{case } M \text{ of } \text{inl}(x_1) \Rightarrow A_1 \text{ inr}(x_2) \Rightarrow A_2 \mathbb{Z} B \sim \text{case } M' \text{ of } \text{inl}(x_1) \Rightarrow A'_1 \text{ inr}(x_2) \Rightarrow A'_2 \mathbb{Z} B' : \text{CmdUn}}
\end{array}$$

Fig. 5. Command Equivalence ($E; \theta \Vdash A \sim A' : \gamma$)

$$\begin{array}{c}
\text{(STORE REL)} \\
\frac{dom(\Sigma) = dom(\Sigma') \quad (\forall p \in dom(\Sigma)) E; \theta \vdash_{\bar{\tau}} \Sigma(p) \sim \Sigma'(p) : E(t.p)}{E; \theta \vdash_{\bar{\tau}} \Sigma \sim \Sigma'} \\
\text{(CMD PRC REL)} \qquad \qquad \qquad \text{(MSG PRC REL)} \\
\frac{\text{(ZERO PRC REL)} E; \theta \vdash_{\bar{\tau}} \Sigma \sim \Sigma' \quad E; \theta \vdash_{\bar{\tau}} A \sim A' : \gamma \quad E; \theta \vdash V \sim V' : Un}{E; \theta \vdash \mathbf{0} \sim \mathbf{0} \quad E; \theta \vdash t.\Sigma/A \sim t.\Sigma'/A' \quad E; \theta \vdash \text{msg } V \sim \text{msg } V'} \\
\text{(BEGUN PRC REL)} \qquad \text{(ENDED PRC REL)} \qquad \text{(NEW PRC REL)} \qquad \text{(PAR PRC REL)} \\
\frac{E; \theta \vdash V \sim V' : \tau}{E; \theta \vdash \text{begun } V \sim \text{begun } V'} \quad \frac{E; \theta \vdash V \sim V' : \tau}{E; \theta \vdash \text{ended } V \sim \text{ended } V'} \quad \frac{m \notin fn(\theta) \quad E, m : \tau; \theta \vdash P \sim P'}{E; \theta \vdash (vm)P \sim (vm)P'} \quad \frac{E; \theta \vdash P \sim P' \quad E; \theta \vdash Q \sim Q'}{E; \theta \vdash P|Q \sim P'|Q'}
\end{array}$$

Fig. 6. Store and Process Equivalence $(E; \theta \vdash_{\bar{\tau}} \Sigma \sim \Sigma') \quad (E; \theta \vdash P \sim P')$

Structural equivalence is defined as the smallest equivalence relation over processes that satisfies the following.

$$\begin{array}{ll}
\mathbb{C}[t.\Sigma/\text{skip}] \equiv \mathbb{C}[\mathbf{0}] & \text{(Par Equiv)} \\
\mathbb{C}[P|\mathbf{0}] \equiv \mathbb{C}[P] & \text{(Zero Equiv)} \\
\mathbb{C}[P|Q] \equiv \mathbb{C}[Q|P] & \text{(Comm Equiv)} \\
\mathbb{C}[P|(Q|R)] \equiv \mathbb{C}[(P|Q)|R] & \text{(Trans Equiv)} \\
\mathbb{C}[(vm)(P|Q)] \equiv \mathbb{C}[(vm)P|Q] & \text{if } m \notin fn(Q) \quad \text{(New Par Equiv)} \\
\mathbb{C}[(vm)P] \equiv \mathbb{C}[P] & \text{if } m \notin fn(P) \quad \text{(New Equiv)} \\
\mathbb{C}[(vm)(vn)P] \equiv \mathbb{C}[(vn)(vm)P] & \text{(New New Equiv)}
\end{array}$$

Fig. 7. Structural equivalence

$$\begin{array}{lll}
\Sigma/m \Downarrow m & & \text{(Name Eval)} \\
\Sigma/\text{unit} \Downarrow \text{unit} & & \text{(Unit Eval)} \\
\Sigma/(M,N) \Downarrow (V,W) & \text{if } \Sigma/M \Downarrow V \text{ and } \Sigma/N \Downarrow W & \text{(Product Eval)} \\
\Sigma/\text{inl } M \Downarrow \text{inl } V & \text{if } \Sigma/M \Downarrow V & \text{(Sum Left Eval)} \\
\Sigma/\text{inr } M \Downarrow \text{inr } V & \text{if } \Sigma/M \Downarrow V & \text{(Sum Right Eval)} \\
\Sigma/\{M\}N \Downarrow \{V\}W & \text{if } \Sigma/M \Downarrow V \text{ and } \Sigma/N \Downarrow W & \text{(Cipher Eval)} \\
\Sigma/p \Downarrow \Sigma(p) & \text{if } \Sigma(p) \text{ defined} & \text{(Ref Eval)} \\
\Sigma/M==N \Downarrow \text{inl unit} & \text{if } \Sigma/M \Downarrow V \text{ and } \Sigma/N \Downarrow V & \text{(Equals True Eval)} \\
\Sigma/M==N \Downarrow \text{inr unit} & \text{otherwise if } \Sigma/M \Downarrow \text{ and } \Sigma/N \Downarrow & \text{(Equals False Eval)} \\
\Sigma/\text{dec } M \text{ with } N \Downarrow \text{inl } V & \text{if } \Sigma/M \Downarrow \{V\}W \text{ and } \Sigma/N \Downarrow W & \text{(Decrypt True Eval)} \\
\Sigma/\text{dec } M \text{ with } N \Downarrow \text{inr unit} & \text{otherwise if } \Sigma/M \Downarrow \text{ and } \Sigma/N \Downarrow & \text{(Decrypt False Eval)}
\end{array}$$

Fig. 8. Term evaluation $(\Sigma/M \Downarrow V)$

$\mathbb{C}[t.\Sigma/A; B] \rightarrow \mathbb{C}'[t.\Sigma'/A'; B]$	if $\mathbb{C}[t.\Sigma/A] \rightarrow \mathbb{C}'[t.\Sigma'/A']$ (Sequence Eval)
$\mathbb{C}[t.\Sigma/\text{skip}; B] \rightarrow \mathbb{C}[t.\Sigma/B]$	(Skip Eval)
$\mathbb{C}[t.\Sigma/p := M] \rightarrow \mathbb{C}[t.\Sigma, p:V/\text{skip}]$	if $\Sigma/M \Downarrow V$ (Set Eval)
$\mathbb{C}[t.\Sigma/\text{while } MA] \rightarrow \mathbb{C}[t.\Sigma/A; \text{while } MA]$	if $\Sigma/M \Downarrow \text{inl unit}$ (While True Eval)
$\mathbb{C}[t.\Sigma/\text{while } MA] \rightarrow \mathbb{C}[t.\Sigma/\text{skip}]$	otherwise if $\Sigma/M \Downarrow$ (While False Eval)
$\mathbb{C}[t.\Sigma/\text{begin } M] \rightarrow \mathbb{C}[\text{begun } V \mid t.\Sigma/\text{skip}]$	if $\Sigma/M \Downarrow V$ (Begin Eval)
$\mathbb{C}[t.\Sigma/\text{end } M] \rightarrow \mathbb{C}[\text{ended } V \mid t.\Sigma/\text{skip}]$	if $\Sigma/M \Downarrow V$ (End Eval)
$\mathbb{C}[t.\Sigma/\text{out } M] \rightarrow \mathbb{C}[\text{msg } V \mid t.\Sigma/\text{skip}]$	if $\Sigma/M \Downarrow V$ (Output Eval)
$\mathbb{C}[\text{msg } V \mid t.\Sigma/\text{inp } (x); A] \rightarrow \mathbb{C}[t.\Sigma/A\{V/x\}]$	(Input Eval)
$\mathbb{C}[t.\Sigma/\text{!}M \text{!}N \text{ as } (x); A] \rightarrow \mathbb{C}[(\nu m)t.\Sigma/A\{\text{!}(V,m)\text{!}W/x\}]$	if $\Sigma/M \Downarrow V$ and $\Sigma/N \Downarrow W$ and m fresh (Encrypt Eval)
$\mathbb{C}[t.\Sigma/\text{split } M \text{ as } (x, y); A \dot{\downarrow} B] \rightarrow \mathbb{C}[t.\Sigma/A\{V/x\}\{W/y\}]$	if $\Sigma/M \Downarrow (V, W)$ (Split Eval)
$\mathbb{C}[t.\Sigma/\text{split } M \text{ as } (x, y); A \dot{\downarrow} B] \rightarrow \mathbb{C}[t.\Sigma/B]$	otherwise if $\Sigma/M \Downarrow$ (Split Default Eval)
$\mathbb{C}[t.\Sigma/\text{case } M \text{ of inl } (x_1) \Rightarrow A_1 \text{ inr } (x_2) \Rightarrow A_2 \dot{\downarrow} B] \rightarrow \mathbb{C}[t.\Sigma/A_1\{V/x_1\}]$	if $\Sigma/M \Downarrow \text{inl } V$ (Case Left Eval)
$\mathbb{C}[t.\Sigma/\text{case } M \text{ of inl } (x_1) \Rightarrow A_1 \text{ inr } (x_2) \Rightarrow A_2 \dot{\downarrow} B] \rightarrow \mathbb{C}[t.\Sigma/A_2\{V/x_2\}]$	if $\Sigma/M \Downarrow \text{inr } V$ (Case Right Eval)
$\mathbb{C}[t.\Sigma/\text{case } M \text{ of inl } (x_1) \Rightarrow A_1 \text{ inr } (x_2) \Rightarrow A_2 \dot{\downarrow} B] \rightarrow \mathbb{C}[t.\Sigma/B]$	otherwise if $\Sigma/M \Downarrow$ (Case Default Eval)

Fig. 9. Process Evaluation ($P \rightarrow Q$)