Confidential Safety via Correspondence Assertions

Radha Jagadeesan
DePaul University

Alan Jeffrey
Bell Labs, Alcatel-Lucent

Corin Pitcher
DePaul University

James Riely
DePaul University

***Abstract*—We study a notion of secrecy that arises naturally in adversarial systems. Let all agents agree on a space of possible values. An honest agent chooses one of these values, and aims to make sure that this particular choice cannot be reliably guessed by an adversary, even in the context of a distributed protocol. An example is an agent that uses an honest mail server to send a message, wishing to keep the identity of the eventual recipient hidden from an adversary.**

We refer to this property as confidential safety. We provide a formal definition in the context of a concurrent while-language. We provide techniques for establishing confidential safety, building on three ingredients:

- **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).**
- **We establish the confidentiality of a correspondence assertion using techniques reminiscent of information flow, and incorporating cryptography using confounders.**
- **We prove protocols (such as instances of the mail server above) are confidentially safe using techniques based on frame-bisimulation that exploit the non-interference properties of well-typed programs.**

I. 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. □

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. □

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, in our current paper, 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. While our precise statement is in the context of a first-order concurrent imperative 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. We describe a type system using techniques that are familiar from work on secrecy and information flow. The type system develops along standard lines, blending elements from Abadi (1999) and Smith (2006). Well-typed programs satisfy non-interference properties that ensure that values guessable by the opponent are kept separate from the values that are not-guessable by the opponent. Our proofs of non-interference are based on establishing a contextual frame-bisimulation.

Rest of the paper. We informally develop our ideas in the next section. In Sections IV and V, we present the language and formally define confidential safety. We present a typing system to establish non-interference results in Section VI, and use it to prove confidential safety for the mail example in Section VII.

II. 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 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 $\mathtt{k_{AS}}$ 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 confidentially 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 an 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)}k_AS 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 the 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 replacing the encryption used in the above protocol by using confounders (Abadi 1999).

```
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 messages 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 confidentially 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)}k_AS to S
```

This variant 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 while 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.

III. Related work

Our definition of confidential safety is heavily influenced by the treatment of secrecy in adversarial systems by Halpern and O'Neill (2008). In particular, our basic definition can be seen as arising by eliding the probabilistic aspects of their analysis in Section 4.4. The novelty of our work lies in the adaptation of these ideas 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 `end`s 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).

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 `end`s, whereas `begin`s 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), the spi-calculus type system for secrecy (Abadi 1999), and many others. What we share with all these papers is the desire to partition the worlds of information that is guessable by the adversary (and publishable over the network) from that which is not. As in (Smith 2006) and (Abadi 1999), we use cryptography as a means to declassify non-guessable information so that it can be published over the network. For an analysis of the issues underlying declassification, we refer the reader to Sabelfeld and Sands (2005).

At a superficial level, our underlying paradigm of multi-threaded imperative programs with message passing (motivated for us by the examples of interest) differs from these papers (that range from multi-threaded imperative programs with shared memory to process algebras).

As in Abadi (1999), 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) we eschew the use of randomized cryptography and the associated computational/probabilistic considerations in the non-interference arguments.

The type system that we use as a basis of our analysis is very similar to that of Abadi (1999). The main technical difference, apart from the underlying paradigm, is that we use a relation inspired by frame-bisimulation to show a that programs that differ only on secret subcomponents cannot be distinguished by an adversary. This contextual version of non-interference generalizes the single program non-interference statements of Abadi.

Our use of frames and contextual-frame arguments is inspired by frame-bisimulation (Abadi and Gordon 1998).

IV. A CONCURRENT WHILE LANGUAGE

We define the syntax and evaluation semantics for a simple concurrent while language. The syntax categories include terms, values, commands and processes. We first present the formalities, then provide discussion. Finally, we describe abbreviations that are used in examples.

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 are defined as follows.

$$
\begin{array}{rll}
M,N,L & ::= m \mid \texttt{unit} & \text{(Base Val/Term)} \\
& \mid (M,N) & \text{(Product Val/Term)} \\
& \mid \texttt{inl}\ M \mid \texttt{inr}\ M & \text{(Sum Val/Term)} \\
& \mid \{M\}N & \text{(Cipher Val/Term)} \\
& \mid x & \text{(Variable Term)} \\
& \mid p & \text{(Reference Term)} \\
& \mid M{=}{=}N & \text{(Equals Term)} \\
& \mid \texttt{dec}\ M\ \texttt{with}\ N & \text{(Decrypt Term)}
\end{array}
$$

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.

Commands and processes are defined as follows.

$$
\begin{array}{rll}
A,B & ::= \texttt{skip} & \text{(Skip Cmd)} \\
& \mid A\texttt{;}\ B & \text{(Sequence Cmd)} \\
& \mid p := M & \text{(Set Cmd)} \\
& \mid \texttt{while}\ M\ A & \text{(While Cmd)} \\
& \mid \texttt{begin}\ M & \text{(Begin Cmd)} \\
& \mid \texttt{end}\ M & \text{(End Cmd)} \\
& \mid \texttt{out}\ M & \text{(Output Cmd)} \\
& \mid \texttt{inp}\ (x)\texttt{;}\ A & \text{(Input Cmd)} \\
& \mid \{\!|M|\!\}N\ \texttt{as}\ (x)\texttt{;}\ A & \text{(Encrypt Cmd)} \\
& \mid \texttt{split}\ M\ \texttt{as}\ (x,y)\texttt{;}\ A \not\downarrow B & \text{(Split Cmd)} \\
& \mid \texttt{case}\ M\ \texttt{of}\ \texttt{inl}(x_1)\ \texttt{=>}\ A_1\ \texttt{inr}(x_2)\ \texttt{=>}\ A_2 \not\downarrow B & \text{(Case Cmd)} \\
P,Q & ::= \mathbf{0} & \text{(Zero Proc)} \\
& \mid P \mid Q & \text{(Parallel Proc)} \\
& \mid (\nu m)P & \text{(New Proc)} \\
& \mid t.\Sigma/A & \text{(Thread Proc)} \\
& \mid \texttt{begun}\ V & \text{(Begun Proc)} \\
& \mid \texttt{ended}\ V & \text{(Ended Proc)} \\
& \mid \texttt{msg}\ V & \text{(Message Proc)}
\end{array}
$$

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 1, where $\Sigma/M \Downarrow$ is defined as $\Sigma/M \Downarrow V$ for some V.

The evaluation relation on processes ($P \to Q$) is defined in Figure 2. The definition uses contexts, which are defined as follows.

$$\mathbb{C} ::= [\text{–}] \mid (\nu m)\mathbb{C} \mid \mathbb{C} \mid P \mid P \mid \mathbb{C}$$

Let $\twoheadrightarrow$ be the reflexive and transitive closure of $(\to) \cup (\equiv)$, where $\equiv$ is a standard[1] 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 $\texttt{inl}\,V$.

[1] Structural equivalence is defined as the smallest equivalence relation over processes that satisfies the following. $\mathbb{C}[t.\Sigma/\texttt{skip}] \equiv \mathbb{C}[\mathbf{0}]$. $\mathbb{C}[P \mid \mathbf{0}] \equiv \mathbb{C}[P]$. $\mathbb{C}[P \mid Q] \equiv \mathbb{C}[Q \mid P]$. $\mathbb{C}[P \mid (Q \mid R)] \equiv \mathbb{C}[(P \mid Q) \mid R]$. $\mathbb{C}[(\nu m)(P \mid Q)] \equiv \mathbb{C}[((\nu m)P) \mid Q)]$ if $m \notin fn(Q)$. $\mathbb{C}[(\nu m)P] \equiv \mathbb{C}[P]$ if $m \notin fn(P)$. $\mathbb{C}[(\nu m)(\nu n)P] \equiv \mathbb{C}[(\nu n)(\nu m)P]$.

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

Figure 1. Term Evaluation ($\Sigma/M \Downarrow V$)

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

Figure 2. Process Evaluation ($P \to Q$)

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 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 $\ldots\ \{\!|M|\!\}N\ \ldots$ is implemented as $\{\!|M|\!\}N$ `as (`x`);` $\ldots\ x\ \ldots$, where x is fresh.

We write `true` for `inl unit` and `false` for `inr unit`. We write `if` M `then` A_1 `else` $A_2 \not\downarrow B$ to abbreviate `case` M `inl(`x_1`) => `A_1`inr(`x_2`) => `$A_2 \not\downarrow B$ 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`);` $A \not\downarrow$ `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 $(t.\emptyset/A) \mid \mathbf{0}$.

V. Conditional safety

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

Definition 3 (Safety). A program P is *safe* if whenever $P \twoheadrightarrow \mathbb{C}[\texttt{ended}\, V]$ then $\texttt{begun}\, V \in \mathbb{C}$. □

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

Let $\mathscr{P}$ range over sets (families) of processes.

An *opponent* is a process that contains no type but *Un*.

Definition 4 (Confidential safety). A process family $\mathscr{P}$ is *confidentially safe for* O if whenever $P \mid O$ is safe for every $P \in \mathscr{P}$, then $erase(P) \mid O$ is safe for every $P \in \mathscr{P}$.

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

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 5. A `begin`-free opponent O is *successful* against an `end`-free family $\mathscr{P}$ if $P \mid O$ is safe for every $P \in \mathscr{P}$ and $P \mid O \twoheadrightarrow \mathbb{C}[\texttt{ended}\, V]$ for some $P \in \mathscr{P}$ and some $\mathbb{C}$ and V. □

Lemma 6. *An* `end`*-free family* $\mathscr{P}$ *is* CS *if and only if it has no successful opponents.*

Proof. Immediate. □

To give a feel for the definitions, let us consider some trivial examples.

Example 7. Any family of processes in which no process performs a `begin` is CS; this includes the empty set. □

This is unsurprising since the family claims no confidentiality.

Example 8. 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 $erase(\texttt{begin a}) = \texttt{skip}$. □

Example 9. The singleton family `begin a | begin b` is also not CS, as demonstrated by the successful opponents `end a`, `end b`, and `end a | end b`. □

These results are also unsurprising, since these families produce the same `begin`s in all runs.

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

Example 10. The family consisting of the two process `begin a` and `begin b` is CS. This family has no successful opponents. Different `begin`s occur in different runs, yet the opponent has no way to determine which `begin` has occurred. If our language included nondeterministic choice[2]

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

the singleton family `begin a` $\oplus$ `begin b` would be CS as well. □

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 thus far, secrecy is trivial, since the processes communicate nothing.

The examples to now are all trivial. There is no communication 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 11. The family comprising the two processes `begin a; out a` and `begin b; out b` is not CS, as evidenced by the successful opponent is `in (x); end x`. Assuming nondeterministic choice, the singleton `begin a; out a` $\oplus$ `begin b; out b` is also not CS. □

Adding encryption with a secret key is enough to regain CS in this simple example.

Example 12. The family comprising the two processes (νk) `begin a; out {a}k` and (νk) `begin b; out {b}k` is CS. □

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 (νk) `out {a}k` is frame bisimilar to (νk) `out {b}k`, one must be careful if messages may be sent more than once. (νk) `out {a}k out {a}k` is not frame bisimilar to (νk) `out {a}k out {b}k`.

Example 13. The following variant family is not CS. The first process is (νk) `begin a; out {a}k out {b}k`. The second process is (νk) `begin b; out {b}k out {b}k` is CS. 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`. □

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

In general, CS requires reasoning about information flow. In the next section we present a type system that ensures that secret data is not leaked by good processes. In Section VII we present examples using this type system.

Unusually, our use of correspondence assertions does not require that good processes ever perform an `end`. To simplify the presentation, our typing system does not track legal `end`s. Tracking legal `end`s is solved by Gordon and Jeffrey (2002) using effects and `Ok` types, and their solution is easily adapted to our setting.

VI. 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.

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

The type $\mathit{Key}\ \sigma$ 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 3. 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 $\mathit{Key}\ \sigma$ requires plaintext of type $\sigma \times \mathit{SecretUn}$, 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, $\mathit{Un} \times \mathit{Un}$ 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 $\mathit{Key}\ \sigma$ does not range over *Un*, and so data of type *Un* must be coerced to *SecretUn* before it is encrypted.

(COERCE UN TERM) $\dfrac{E \vdash_t M : Un}{E \vdash_t M : SecretUn}$ (VAR TERM) $\dfrac{E(x) = \tau}{E \vdash_t x : \tau}$ (NAME TERM) $\dfrac{E(m) = \tau}{E \vdash_t m : \tau}$ (UNIT TERM) $\dfrac{}{E \vdash_t \mathtt{unit} : Un}$ (REF TERM) $\dfrac{E(t.p) = \tau}{E \vdash_t p : \tau}$ (LEFT TERM) $\dfrac{E \vdash_t M : \sigma}{E \vdash_t \mathtt{inl}\, M : \sigma + \rho}$ (RIGHT TERM) $\dfrac{E \vdash_t M : \rho}{E \vdash_t \mathtt{inr}\, M : \sigma + \rho}$

(UN LEFT TERM) $\dfrac{E \vdash_t M : Un}{E \vdash_t \mathtt{inl}\, M : Un}$ (UN RIGHT TERM) $\dfrac{E \vdash_t M : Un}{E \vdash_t \mathtt{inr}\, M : Un}$ (PRODUCT TERM) $\dfrac{E \vdash_t M : \sigma \quad E \vdash_t N : \rho}{E \vdash_t (M,N) : \sigma \times \rho}$ (UN PRODUCT TERM) $\dfrac{E \vdash_t M : Un \quad E \vdash_t N : Un}{E \vdash_t (M,N) : Un}$

(CIPHER TERM) $\dfrac{E \vdash_t M : \sigma \times SecretUn \quad E \vdash_t N : Key\,\sigma}{E \vdash_t \{M\}N : Un}$ (UN CIPHER TERM) $\dfrac{E \vdash_t M : Un \quad E \vdash_t N : Un}{E \vdash_t \{M\}N : Un}$ (EQUALS TERM) $\dfrac{E \vdash_t M : \sigma \quad E \vdash_t N : \sigma}{E \vdash_t M{=}{=}N : Bool}$ (UN EQUALS TERM) $\dfrac{E \vdash_t M : Un \quad E \vdash_t N : Un}{E \vdash_t M{=}{=}N : Un}$

(DECRYPT TERM) $\dfrac{E \vdash_t M : SecretUn \quad E \vdash_t N : Key\,\sigma}{E \vdash_t \mathtt{dec}\, M\, \mathtt{with}\, N : (\sigma \times SecretUn) + SecretUn}$ (UN DECRYPT TERM) $\dfrac{E \vdash_t M : Un \quad E \vdash_t N : Un}{E \vdash_t \mathtt{dec}\, M\, \mathtt{with}\, N : Un}$

Figure 3. Term Typing $(E \vdash_t M : \tau)$

(COERCE SEC CMD) $\dfrac{E \vdash_t A : CmdSec}{E \vdash_t A : CmdUn}$ (SKIP CMD) $\dfrac{}{E \vdash_t \mathtt{skip} : CmdSec}$ (SEQUENCE CMD) $\dfrac{E \vdash_t A : \gamma \quad E \vdash_t B : \gamma}{E \vdash_t A\,;\, B : \gamma}$ (ASSIGN CMD) $\dfrac{E(t.p) = \sigma \quad E \vdash_t M : \sigma}{E \vdash_t p := M : CmdSec}$ (UN ASSIGN CMD) $\dfrac{E(t.p) = Un \quad E \vdash_t M : Un}{E \vdash_t p := M : CmdUn}$

(WHILE CMD) $\dfrac{E \vdash_t M : Bool \quad E \vdash_t A : CmdSec \quad \text{loop always terminates}}{E \vdash_t \mathtt{while}\, M\, A : CmdSec}$ (UN WHILE CMD) $\dfrac{E \vdash_t M : Un \quad E \vdash_t A : CmdUn}{E \vdash_t \mathtt{while}\, M\, A : CmdUn}$ (BEGIN CMD) $\dfrac{E \vdash_t M : \sigma \quad E \vdash_t N : \tau}{E \vdash_t \mathtt{begin}\, (M,N) : \gamma}$ (END CMD) $\dfrac{E \vdash_t M : \tau}{E \vdash_t \mathtt{end}\, M : \gamma}$

(ENCRYPT CMD) $\dfrac{E \vdash_t M : \sigma \quad E \vdash_t N : Key\,\sigma \quad E,x{:}Un \vdash_t A : \gamma}{E \vdash_t \{\!|M|\!\}N\, \mathtt{as}\, (x)\,;\, A : \gamma}$ (UN ENCRYPT CMD) $\dfrac{E \vdash_t M : Un \quad E \vdash_t N : Un \quad E,x{:}Un \vdash_t A : \gamma}{E \vdash_t \{\!|M|\!\}N\, \mathtt{as}\, (x)\,;\, A : \gamma}$

(SPLIT CMD) $\dfrac{E \vdash_t M : \sigma \times \rho \quad E,x{:}\sigma,y{:}\rho \vdash_t A : CmdSec \quad E \vdash_t B : CmdSec}{E \vdash_t \mathtt{split}\, M\, \mathtt{as}\, (x,y)\,;\, A \,\mathtt{\not\downarrow}\, B : CmdSec}$

(UN SPLIT CMD) $\dfrac{E \vdash_t M : Un \quad E,x{:}Un,y{:}Un \vdash_t A : \gamma \quad E \vdash_t B : \gamma}{E \vdash_t \mathtt{split}\, M\, \mathtt{as}\, (x,y)\,;\, A \,\mathtt{\not\downarrow}\, B : \gamma}$

(CASE CMD) $\dfrac{E \vdash_t M : \sigma + \rho \quad E,x_1{:}\sigma \vdash_t A_1 : CmdSec \quad E,x_2{:}\rho \vdash_t A_2 : CmdSec \quad E \vdash_t B : CmdSec}{E \vdash_t \mathtt{case}\, M\, \mathtt{of}\, \mathtt{inl}(x_1) \Rightarrow A_1\, \mathtt{inr}(x_2) \Rightarrow A_2 \,\mathtt{\not\downarrow}\, B : CmdSec}$

(UN CASE CMD) $\dfrac{E \vdash_t M : Un \quad E,x_1{:}Un \vdash_t A_1 : \gamma \quad E,x_2{:}Un \vdash_t A_2 : \gamma \quad E \vdash_t B : \gamma}{E \vdash_t \mathtt{case}\, M\, \mathtt{of}\, \mathtt{inl}(x_1) \Rightarrow A_1\, \mathtt{inr}(x_2) \Rightarrow A_2 \,\mathtt{\not\downarrow}\, B : \gamma}$

Figure 4. Command Typing $(E \vdash_t A : \gamma)$

(STORE) $\dfrac{\forall p \in dom(\Sigma) \exists \tau. E(t.p) = \tau \text{ and } E \vdash_t \Sigma(p) : \tau}{E \vdash_t \Sigma}$

(ZERO PRC) $\dfrac{}{E \vdash \mathbf{0}}$ (CMD PRC) $\dfrac{E \vdash_t \Sigma \quad E \vdash_t A : \gamma}{E \vdash t.\Sigma/A}$ (MSG PRC) $\dfrac{E \vdash V : Un}{E \vdash \mathtt{msg}\, V}$ (BEGUN PRC) $\dfrac{E \vdash V : Un}{E \vdash \mathtt{begun}\, V}$ (ENDED PRC) $\dfrac{E \vdash V : Un}{E \vdash \mathtt{ended}\, V}$ (NEW PRC) $\dfrac{E,m{:}\tau \vdash P}{E \vdash (\nu m)P}$ (PAR PRC) $\dfrac{E \vdash P \quad E \vdash Q}{E \vdash P \mid Q}$

Figure 5. Store and Process Typing $(E \vdash_t \Sigma)$ $(E \vdash P)$

(SEC TERM REL) $\dfrac{E;\theta \vdash \diamond \quad E \vdash_t M : \sigma \quad E \vdash_t M' : \sigma}{E;\theta \vdash_t M \sim M' : \sigma}$ (UN VAR REL) $\dfrac{E(x) = Un \quad E;\theta \vdash \diamond}{E;\theta \vdash_t x \sim x : Un}$ (UN NAME REL) $\dfrac{E(m) = Un \quad E;\theta \vdash \diamond}{E;\theta \vdash_t m \sim m : Un}$ (UN REF REL) $\dfrac{E(t.p) = Un \quad E;\theta \vdash \diamond}{E;\theta \vdash_t p \sim p : Un}$

(UN LEFT REL) $\dfrac{E;\theta \vdash_t M \sim M' : Un}{E;\theta \vdash_t \mathtt{inl}\, M \sim \mathtt{inl}\, M' : Un}$ (UN RIGHT REL) $\dfrac{E;\theta \vdash_t M \sim M' : Un}{E;\theta \vdash_t \mathtt{inr}\, M \sim \mathtt{inr}\, M' : Un}$ (UN PRODUCT REL) $\dfrac{E;\theta \vdash_t M \sim M' : Un \quad E;\theta \vdash_t N \sim N' : Un}{E;\theta \vdash_t (M,N) \sim (M',N') : Un}$

(CIPHER REL) $\dfrac{(\{M\}N, \{M'\}N', \sigma) \in \theta \quad E;\theta \vdash \diamond}{E;\theta \vdash_t \{M\}N \sim \{M'\}N' : Un}$ (UN CIPHER REL) $\dfrac{E;\theta \vdash_t M \sim M' : Un \quad E;\theta \vdash_t N \sim N' : Un}{E;\theta \vdash_t \{M\}N \sim \{M'\}N' : Un}$

(UN EQUALS REL) $\dfrac{E;\theta \vdash_t M \sim M' : Un \quad E;\theta \vdash_t N \sim N' : Un}{E;\theta \vdash_t M{=}{=}N \sim M'{=}{=}N' : Un}$ (UN DECRYPT REL) $\dfrac{E;\theta \vdash_t M \sim M' : Un \quad E;\theta \vdash_t N \sim N' : Un}{E;\theta \vdash_t \mathtt{dec}\, M\, \mathtt{with}\, N \sim \mathtt{dec}\, M'\, \mathtt{with}\, N' : Un}$

Figure 6. Term Equivalence $(E;\theta \vdash_t M \sim M' : \tau)$

(SEC REL)
$$\frac{E;\theta \vdash \diamond \quad E \vdash_l A : CmdSec \quad E \vdash_l A' : CmdSec}{E;\theta \vdash_l A \sim A' : CmdSec}$$

(COERCE SEC REL)
$$\frac{E;\theta \vdash_l A \sim A' : CmdSec}{E;\theta \vdash_l A \sim A' : CmdUn}$$

(UN SEQUENCE REL)
$$\frac{E;\theta \vdash_l A \sim A' : CmdUn \quad E;\theta \vdash_l B \sim B' : CmdUn}{E;\theta \vdash_l A\,;\, B \sim A'\,;\, B' : CmdUn}$$

(UN ASSIGN REL)
$$\frac{E(t.p) = Un \quad E;\theta \vdash_l M \sim M' : CmdUn}{E;\theta \vdash_l p := M \sim p := M' : CmdUn}$$

(UN WHILE REL)
$$\frac{E;\theta \vdash_l M \sim M' : Un \quad E;\theta \vdash_l A \sim A' : CmdUn}{E;\theta \vdash_l \texttt{while}\, M\, A \sim \texttt{while}\, M'\, A' : CmdUn}$$

(UN BEGIN REL)
$$\frac{E;\theta \vdash_l M \sim M' : \tau}{E;\theta \vdash_l \texttt{begin}\, M \sim \texttt{begin}\, M' : CmdUn}$$

(UN END REL)
$$\frac{E;\theta \vdash_l M \sim M' : \tau}{E;\theta \vdash_l \texttt{end}\, M \sim \texttt{end}\, M' : CmdUn}$$

(UN OUT REL)
$$\frac{E;\theta \vdash_l M \sim M' : Un}{E;\theta \vdash_l \texttt{out}\, M \sim \texttt{out}\, M' : CmdUn}$$

(UN INP REL)
$$\frac{E,x:Un;\theta \vdash_l A \sim A' : CmdUn}{E;\theta \vdash_l \texttt{inp}\,(x)\,;\, A \sim \texttt{inp}\,(x)\,;\, A' : CmdUn}$$

(ENCRYPT REL)
$$\frac{E;\theta \vdash_l M \sim M' : \sigma \quad E;\theta \vdash_l N \sim N' : Key\,\sigma \quad E,x:Un;\theta \vdash_l A \sim A' : CmdUn}{E;\theta \vdash_l \{\!|M|\!\}N\, \texttt{as}\,(x)\,;\, A \sim \{\!|M'|\!\}N'\, \texttt{as}\,(x)\,;\, A' : CmdUn}$$

(UN ENCRYPT REL)
$$\frac{E;\theta \vdash_l M \sim M' : Un \quad E;\theta \vdash_l N \sim N' : Un \quad E,x:Un;\theta \vdash_l A \sim A' : CmdUn}{E;\theta \vdash_l \{\!|M|\!\}N\, \texttt{as}\,(x)\,;\, A \sim \{\!|M'|\!\}N'\, \texttt{as}\,(x)\,;\, A' : CmdUn}$$

(UN SPLIT REL)
$$\frac{E;\theta \vdash_l M \sim M' : Un \quad E,x:Un,y:Un;\theta \vdash_l A \sim A' : CmdUn \quad E;\theta \vdash_l B \sim B' : CmdUn}{E;\theta \vdash_l \texttt{split}\, M\, \texttt{as}\,(x,y)\,;\, A \nmid B \sim \texttt{split}\, M'\, \texttt{as}\,(x,y)\,;\, A' \nmid B') : CmdUn}$$

(UN CASE REL)
$$\frac{E;\theta \vdash_l M \sim M' : Un \quad E,x_1:Un;\theta \vdash_l A_1 \sim A_1' : CmdUn \quad E,x_2:Un;\theta \vdash_l A_2 \sim A_2' : CmdUn \quad E;\theta \vdash_l B \sim B' : CmdUn}{E;\theta \vdash_l \texttt{case}\, M\, \texttt{of}\, \texttt{inl}(x_1) \Rightarrow A_1\, \texttt{inr}(x_2) \Rightarrow A_2 \nmid B \sim \texttt{case}\, M'\, \texttt{of}\, \texttt{inl}(x_1) \Rightarrow A_1'\, \texttt{inr}(x_2) \Rightarrow A_2' \nmid B' : CmdUn}$$

Figure 7. Command Equivalence $(E;\theta \vdash_l A \sim A' : \gamma)$

(STORE REL)
$$\frac{dom(\Sigma) = dom(\Sigma') \quad (\forall p \in dom(\Sigma))E;\theta \vdash_l \Sigma(p) \sim \Sigma'(p) : E(t.p)}{E;\theta \vdash_l \Sigma \sim \Sigma'}$$

(ZERO PRC REL)
$$\frac{}{E;\theta \vdash \mathbf{0} \sim \mathbf{0}}$$

(CMD PRC REL)
$$\frac{E;\theta \vdash_l \Sigma \sim \Sigma' \quad E;\theta \vdash_l A \sim A' : \gamma}{E;\theta \vdash t.\Sigma/A \sim t.\Sigma'/A'}$$

(MSG PRC REL)
$$\frac{E;\theta \vdash V \sim V' : Un}{E;\theta \vdash \texttt{msg}\, V \sim \texttt{msg}\, V'}$$

(BEGUN PRC REL)
$$\frac{E;\theta \vdash V \sim V' : \tau}{E;\theta \vdash \texttt{begun}\, V \sim \texttt{begun}\, V'}$$

(ENDED PRC REL)
$$\frac{E;\theta \vdash V \sim V' : \tau}{E;\theta \vdash \texttt{ended}\, V \sim \texttt{ended}\, V'}$$

(NEW PRC REL)
$$\frac{m \notin fn(\theta) \quad E,m:\tau;\theta \vdash P \sim P'}{E;\theta \vdash (\nu m)P \sim (\nu m)P'}$$

(PAR PRC REL)
$$\frac{E;\theta \vdash P \sim P' \quad E;\theta \vdash Q \sim Q'}{E;\theta \vdash P \mid Q \sim P' \mid Q'}$$

Figure 8. Store and Process Equivalence $(E;\theta \vdash_l \Sigma \sim \Sigma')$ $(E;\theta \vdash P \sim P')$

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.

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

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 4 and 5.

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 V.

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 14. *If* $E \vdash_l M : Un$ *and* $E \vdash_l N : Key\,\sigma$ *then* $M \neq N$.□

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

Lemma 15 (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 mutually induction with the non-interference relation between terms, defined next.

(EMPTY WF THEORY)
$$\frac{}{E;\emptyset \vdash \diamond}$$
(NON-EMPTY WF THEORY)
$$\frac{\begin{array}{l}\{V_1\}V_2 \text{ does not occur as LHS in } \theta \\ \{V_1'\}V_2' \text{ does not occur as RHS in } \theta \\ E;\theta \vdash V_1 \sim V_1' : \sigma \times \mathit{SecretUn} \quad E;\theta \vdash V_1 \sim V_2' : \mathit{Key}\ \sigma\end{array}}{E;\theta,(\{V_1\}V_2,\{V_1'\}V_2',\sigma) \vdash \diamond}$$

The judgement $E;\theta \vdash_\tau M \sim M' : \tau$, defined in Figure 6, 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 $(\exists\theta)E;\theta \vdash_\tau M \sim M' : \tau$.

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 8.

The rules for commands in Figure 7 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 8 are similar and straightforward.

Lemma 16 (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* $E;\emptyset \vdash P \sim P$. □

Lemma 17 (Initial Process Reflexivity). *If* $E \vdash P$ *and P contains no existing ciphertext in terms then* $E;\emptyset \vdash P \sim P$. □

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

Lemma 18 (Substitution). *If* $E;\theta \vdash V \sim V' : \tau$ *and* $E, x:\tau;\theta \vdash P \sim P'$ *then* $E;\theta \vdash P\{^\tau/_x\} \sim P'\{^\tau/_x\}$. □

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

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

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*
- $Q \twoheadrightarrow (\nu\vec{n})Q'$. □

To see how the theory evolves in the course of evaluation of two processes related by the non-interference relation, consider the following definitions:

$$E \triangleq k_1 : \mathit{Key}\ \sigma, k_1' : \mathit{Key}\ \sigma, k_2 : \mathit{Key}\ \mathit{Key}\ \sigma, k_2' : \mathit{Key}\ \mathit{Key}\ \sigma$$
$$A \triangleq \{\!|x|\!\}k_2\ \texttt{as (y); out y}$$

$A' \triangleq \{\!|x|\!\}k_2'$ `as (y); out` y

The non-interference relation relates k_1 and k_1' since they have a secret type *Key* σ (for an unspecified secret type σ). In addition, A and A' are related as *Un* commands at any thread θ, and are thus related as processes when combined with an empty store for θ.

$$E;\emptyset \vdash k_1 \sim k_1' : \mathit{Key}\ \sigma \quad (1)$$

$$E, x : \mathit{Key}\ \sigma; \emptyset \vdash_\tau A \sim A' : \mathit{CmdUn} \quad (2)$$

$$E, x : \mathit{Key}\ \sigma; \emptyset \vdash t.\emptyset/A \sim t.\emptyset/A' \quad (3)$$

Applying substitution to (1) and (3) we deduce

$$E;\emptyset \vdash t.\emptyset/A\{^{k_1}/_x\} \sim t.\emptyset/A\{^{k_1'}/_x\}$$

The LHS and RHS evaluate by generating a fresh name m and pairing it with the plaintext (the keys k_1 and k_1'). Evaluation proceeds as follows.

$$t.\emptyset/A\{^{k_1}/_x\} \twoheadrightarrow (\nu m)t.\emptyset/\mathtt{out}\ W \quad \text{where } W \triangleq \{(k_1, m)\}k_2$$

$$t.\emptyset/A\{^{k_1'}/_x\} \twoheadrightarrow (\nu m)t.\emptyset/\mathtt{out}\ W' \quad \text{where } W' \triangleq \{(k_1', m)\}k_2'$$

In the non-interference proof we construct the following relation:

$$E, m : \mathit{SecretUn}; (W, W', \mathit{Key}\ \sigma) \vdash t.\emptyset/\mathtt{out}\ W \sim t.\emptyset/\mathtt{out}\ W'$$

Here the expressions W and W' are related at *Un*, because they are the result of encrypting with secret keys and their ciphertext is present in the singleton theory. Well-formedness of the theory is trivial by (1) and the fact that the theory is a singleton set, so there are no conflicts with other triples.

However, even if there was an existing theory θ, the well-formedness of the theory $(\theta, (W, W', \mathit{Key}\ \sigma))$ can still be established, precisely because of the fresh name m that appears in the underlying plaintext and the assumption of idealized encryption. Thus the theory represents a history of the ciphertext encrypted with secret keys, and released at *Un* type.

In addition to recording the history, the non-interference result uses the history to ensure that equality tests in expressions of *Un* type yield the same answer. Consider values V, W, V', W' in related equality tests $V{=}{=}W \sim V'{=}{=}W' : \mathit{Un}$ (in some unspecified environment and theory). We want to show that $V = W$ iff $V' = W'$. Here we have the following cases for V and W:

- If both V and W result from encryption with a secret key, both must be in the theory, and the well-formedness condition immediately yields that $V = W$ iff $V' = W'$.
- If V results from encryption with a secret key, and W result from encryption with a key at *Un*, we use the fact that there are no values (including keys) that type at both *Un* and a secret key type. Therefore the ciphertexts are encrypted with different keys, and we conclude that the ciphertexts must be distinct.
- If both V and W result from encryption with a key at *Un*, the plaintexts are at *Un*, then the plaintexts are identical except at ciphertext occurrences. An inductive argument is used to deal with such nested ciphertext occurrences.

VII. Proving conditional 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 of the previous section. In addition, one must show that the `begin`s performed by the processes differ using a secret. To simplify the presentation, we fix the location of the secret, insisting that the every typable `begin` be a pair, whose first component has a secret type.

We say that two processes have *strongly different begins* if whenever $P \twoheadrightarrow (\nu \vec{m})(\mathtt{begun}\ (m, V) \mid P')$ and $Q \twoheadrightarrow (\nu \vec{n})(\mathtt{begun}\ (n, W) \mid Q')$ 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 20. *A process family $\mathscr{P}$ is confidentially safe if*

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

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). □

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 readablity, 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.

```
Server(s) ≜
  msgs := ∅;
  while true
    inp (x,s,ctext);
    case keys[x] of
      false => skip
      inl k => case (dec ctext with k) of
        (get,_) =>
          ack  := mail msgs[x];
          msgs := msgs[x↦nil]
        (snd(r,m),_) =>
          ack  := ok;
          msgs := msgs[r↦[(x,m)]]
    out (s,x,{|ack|}(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

```
System(x) ≜ (νk_as)(νk_bs)(νk_cs)
  keys:[(a,k_as),(b,k_bs),(c,k_cs)]/Server(s)
  | r:x,m:m/Sender(a,s,k_as)
  | Receiver(b,s,k_bs)
  | Receiver(c,s,k_cs)
```

where *Sender* and *Receiver* are defined below.

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.

```
Sender(a,s,k_as) ≜
  begin (r,a,m);
  out (a,s,{|snd(r,m)|}k_as);
  inp (s,a,ctext);
  case (dec ctext of k_as) of
    (ok,_)      => skip
    (mail _,_)  => error
```

Receiver(r,s,k_{rs}) is defined as follows, where r is the name of the receiver, 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_{as} is secret.

```
Receiver(r,s,k_rs) ≜
  while true
    out (r,s,{|get|}k_rs);
    inp (s,r,ctext);
    case (dec ctext of k_rs) of
      (ok,_)      => error
      (mail ms,_) =>
        case ms of nil     => skip
                   [(x,m)] => skip //end(r,x,m)
```

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

PROOF. The result follows from Corollary 20, since the processes have strongly different `begin`s and $System(b) \sim System(c)$, with `r` typed as *SecretUn*. □

VIII. CONCLUSION

In this paper, 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 secrecy and provided a formal definition using a novel variation of correspondence assertions. We provided means to establish this property using a mixture of techniques from information flow and frame-bisimulation style arguments.

From the viewpoint of stochastic games, it can be argued we are being restrictive about when our adversaries are able to break the confidential safety of a protocol. Rather than demanding guaranteed success, the quantitative setting of stochastic games permits us to explore the possibilities of the adversary succeeding with a reasonable chance of success. On the one hand, this enriches the power of the adversary with probabilities as a means to estimate and analyze the events happening on the network. On the other hand, it also opens the possibility of considering computational-style analysis that incorporate realistic models of randomized cryptography and resource bounds on the computation performed by the adversary. We leave the investigation into these extensions to future work.

References

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.

C. Fournet, G. L. Guernic, and T. Rezk. A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In *ACM Conference on Computer and Communications Security*, pages 432–441. ACM, 2009.

A. D. Gordon and A. Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In *ISSS*, volume 2609 of *Lecture Notes in Computer Science*, pages 263–282. Springer, 2002.

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. Typing correspondence assertions for communication protocols. *Theor. Comput. Sci.*, 300(1-3):379–409, 2003b.

A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3-4):435–483, 2004.

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.

A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW*, pages 255–269. IEEE Computer Society, 2005.

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 *In Proc. ACM Symp. on Principles of Programming Languages*, 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.

T. Y. C. Woo and S. S. Lam. A framework for distributed authorization. In *ACM Conference on Computer and Communications Security*, pages 112–118, 1993.