Secrecy Despite Compromise: Types, Cryptography, and the Pi-Calculus

Andrew D. Gordon[1] and Alan Jeffrey[2⋆]

[1] Microsoft Research
[2] DePaul University and Bell Labs, Lucent Technologies

Abstract. A realistic threat model for cryptographic protocols or for language-based security should include a dynamically growing population of principals (or security levels), some of which may be compromised, that is, come under the control of the adversary. We explore such a threat model within a pi-calculus. A new process construct records the ordering between security levels, including the possibility of compromise. Another expresses the expectation of conditional secrecy of a message—that a particular message is unknown to the adversary unless particular levels are compromised. Our main technical contribution is the first system of secrecy types for a process calculus to support multiple, dynamically-generated security levels, together with the controlled compromise or downgrading of security levels. A series of examples illustrates the effectiveness of the type system in proving secrecy of messages, including dynamically-generated messages. It also demonstrates the improvement over prior work obtained by including a security ordering in the type system. Perhaps surprisingly, the soundness proof for our type system for symbolic cryptography is via a simple translation into a core typed pi-calculus, with no need to take symbolic cryptography as primitive.

1 Introduction

Ever since the Internet entered popular culture it has had associations of insecurity. The Morris worm of 1989 broke the news by attacking vulnerable computers on the network and exploiting them to attack others. At least since then, compromised hosts and untrustworthy users have been a perpetual presence on the Internet, and, perhaps worse, inside many institutional intranets. Hence, like all effective risk management, good computer security does not focus simply on prevention, but also on management and containment.

There is by now a substantial literature on language-based techniques to prevent disclosure of secrets [21]. This paper contributes new language constructs to help manage and contain the impact of partial compromise on a system: we generalize the attacker model from a completely untrusted outsider to include attacks mounted by compromised insiders. We use the pi-calculus [17], a theory of concurrency that already supports reasoning about multiple, dynamically generated identities, and security based on abstract channels or symbolic cryptography [1,4]. We formalize the new idea of *conditional secrecy*, that a message is secret unless particular principals are compromised.

⋆ This material is based upon work supported by the National Science Foundation under Grant No. 0208549.

We describe a type system that checks conditional secrecy, and hence may help assess the containment of compromise within a system.

Specifying Compromise and Conditional Secrecy We model systems as collections of processes, that interact by exchanging messages on named channels. Most of the examples in the paper rely on channel abstractions for security, but our methods also handle protocols that rely on cryptography. The opponent is an implicit process that runs alongside the processes making up the system under attack, and may interact with it using channels (or keys) in its possession. We say a message is *public* if it may come into the possession of the opponent (possibly after a series of interactions).

We base our model of partially compromised systems on a *security ordering* between abstract *security levels*. Secrecy levels model individual (or groups of) principals, hosts, sessions, and other identifiers. For instance, the level of the opponent is the distinguished lowest security level $\bot$.

The process construct $L_1 \leq L_2$, called a *statement*, declares that level L_1 is less than level L_2. Hence, any process defines a security ordering between levels; it is given by the set of active statements occurring in the process, closed under a set of inference rules including reflexivity and transitivity. (Statements are akin to the use of process constructs to describe the occurrence of events [6,14] or to populate a database of facts [10].) We say a level L is *compromised* when $L \leq \bot$. Compromise may arise indirectly: if $L_1 \leq L_2$ and subsequently L_2 is compromised, then so too is L_1, by transitivity. So $L_1 \leq L_2$ can be read "L_1 is at risk from L_2" as well as "L_1 is less secure than L_2."

Compromise may be contained or non-catastrophic in the sense that despite the compromise of one part of a system, another part may reliably keep messages secret. For example, key establishment protocols often have the property that A and B can keep their session key secret even though a session key established between B and a compromised party C has become public. However, as soon as either A or B is compromised, their session key may become public.

The process construct **secret** M **amongst** L, called an *expectation of conditional secrecy*, declares the invariant "M is secret unless L is compromised". For example, the process **secret** S **amongst** (A,B) asserts that S is secret unless the composite security level (A,B) has been compromised, which occurs if either A or B has been compromised. This definition of conditional secrecy via a syntactic process construct is new and may be of interest independently of our type system. By embedding falsifiable expectations within processes, we can express the conditional secrecy of freshly generated messages, unlike previous definitions [2]. Our trace-based notion of secrecy concerns direct flows to an active attacker; we do not address indirect flows or noninterference.

Checking Conditional Secrecy by Typing Our main technical contribution is the first system of secrecy types for a process calculus that supports multiple, dynamically-generated security levels, together with compromise or downgrading of security levels. Abadi's original system [1] of secrecy types for cryptographic protocols, and its descendants, are limited to two security levels, and therefore cannot conveniently model the dynamic creation and compromise of security levels, or the possibility of attack from compromised insiders. Our treatment of asymmetric communication channels builds on our recent work on types for authentication properties [13].

Our main technical result, Theorem 2, is that no expectation of conditional secrecy is ever falsified when a well-typed process interacts with any opponent process.

We anticipate applications of this work both in the design of security-typed languages and in the verification of cryptographic protocols. Security types with multiple security levels are common in the literature on information flow in programming languages, but ours is apparently the first use in the analysis of cryptographic protocols.

Section 2 describes our core pi-calculus. Section 3 exhibits a series of example protocols that make use of secure channels. Theorem 2 can be applied to show these protocols preserve the secrecy of dynamically generated data. Previous type systems yield unconditional secrecy guarantees, and therefore cannot handle the dynamic declassification of data in these protocols. Section 4 presents our type system formally. Section 5 outlines the extension of our results to a pi-calculus with symbolic cryptography. Section 6 discusses related work, and Section 7 concludes.

A companion technical report [15] includes further explanations and examples, an extension of the core calculus and type system to cover symbolic cryptography, and proofs. Notably, the soundness of the extended type system follows via a straightforward translation into our core pi-calculus. We represent ciphertexts as processes, much like the encoding [17] of other data structures in the pi-calculus. Although such a representation of ciphertexts is well known to admit false attacks in general, it is adequate in our typed setting.

2 A Pi Calculus with Expectations of Conditional Secrecy

Our core calculus is an asynchronous form of Odersky's polarized pi-calculus [19] extended with secrecy expectations and security levels.

Computation is based on communication of messages between processes on named channels. The calculus is polarized in the sense that there are separate capabilities to send and receive on each channel. The send capability $k!$ confers the right to send (but not receive) on a channel k. Conversely, the receive capability $k?$ confers the right to receive (but not send) on k. The asymmetry of these capabilities is analogous to the asymmetry between public encryption and private decryption keys, and allows us to write programs with the flavour of cryptographic protocols in a small calculus.

Messages are values communicated over channels between processes. As well as send and receive capabilities, messages include names, pairs, tagged messages, and the distinguished security levels $\top$ and $\bot$.

Processes include the standard pi-calculus constructs plus operations to access pairs and tagged unions. To track direct flows of messages, each output is tagged with its security level; for instance, an output by the opponent may be tagged $\bot$. The only new process constructs are statements $M \leq N$ and expectations **secret** M **amongst** L.

Names, Messages, Processes:

$a, \ldots, n, v, \ldots, z$	names and variables
$L, M, N ::=$	message, security level
x	name, variable
$M?$	capability to input on M

$M!$	capability to output on M
(M,N)	message pair
inl M	left injection
inr M	right injection
$\top$	highest security
$\bot$	lowest security
$C ::= M \leq N$	clause: level M less secure than level N
$\vec{M},\vec{N} ::= M_1,\ldots,M_m$	sequence of messages ($m \geq 0$)
T,U	type: defined in Section 4
$P,Q,R ::=$	process
out $M\ N :: L$	asynchronous output at level L
inp $M(x{:}T);P$	input (scope of x is P)
new $x{:}T;P$	name generation (scope of x is P)
repeat P	replication
$P \mid Q$	parallel composition
stop	inactivity
split M **is** $(x \leq y{:}T, z{:}U);P$	pair splitting (scope of x,y is U, P, of z just P)
match M **is** $(x \leq N{:}T, z{:}U);P$	pair matching (scope of x is U, P, of z just P)
case M **is inl** $(x{:}T)\ P$ **is inr** $(y{:}U)\ Q$	union case (scope of x is P, of y is Q)
C	statement of clause C
secret M **amongst** L	expectation of conditional secrecy

We write $P \rightarrow Q$ to mean P may reduce to Q, and $P \equiv Q$ to mean P and Q are structurally equivalent. The mostly standard definitions of these relations are in [15]. The only nonstandard reductions are for **split** and the first-component-matching operation **match**, which bind an extra variable. (We motivate the use of this variable in Section 3).

$$\textbf{split}\ (M,N)\ \textbf{is}\ (x \leq y{:}T, z{:}U);P \rightarrow P\{x{\leftarrow}M\}\{y{\leftarrow}M\}\{z{\leftarrow}N\}$$
$$\textbf{match}\ (M,N)\ \textbf{is}\ (x \leq M, z{:}U);P \rightarrow P\{x{\leftarrow}M\}\{z{\leftarrow}N\}$$

Any message M can be seen as a *security level*. Levels are ordered, with bottom element $\bot$, top element $\top$, and meet given by (M,N). We write S for a set of clauses of the form $M \leq N$, and write $S \vdash M \leq N$ when $M \leq N$ is derivable from hypotheses S.

Set of Clauses:

$S ::= \{C_1,\ldots,C_n\}$	set of clauses
$\{C_1,\ldots,C_n\} \stackrel{\triangle}{=} C_1 \mid \cdots \mid C_n \mid \textbf{stop}$	when considered as a process

Preorder on Security Levels: $S \vdash M \leq N$

$C \in S \Rightarrow S \vdash C$	(Order Id)
$S \vdash M \leq M$	(Order Refl)
$S \vdash L \leq M \wedge S \vdash M \leq N \Rightarrow S \vdash L \leq N$	(Order Trans)
$S \vdash \bot \leq M$	(Order Bot-L)
$S \vdash M \leq \top$	(Order Top-R)
$S \vdash (M,N) \leq M$	(Order Meet-L-1)
$S \vdash (M,N) \leq N$	(Order Meet-L-2)
$S \vdash L \leq M \wedge S \vdash L \leq N \Rightarrow S \vdash L \leq (M,N)$	(Order Meet-R)

Since processes contain ordering statements, we can derive $P \vdash M \leq N$ whenever P contains statements S, and $S \vdash M \leq N$.

Security Order Induced by a Process: $P \vdash M \leq N$

Let $P \vdash M \leq N$ if and only if $P \equiv \textbf{new}\ \vec{x}{:}\vec{T}; (S \mid Q)$ and $S \vdash M \leq N$ and $\mathsf{fn}(M,N) \cap \{\vec{x}\} = \varnothing$.

An expectation **secret** M **amongst** N in a process is justified if every output of M is at a level L such that $N \leq L$. That is, the secret M may flow up, not down. We say P is *safe for conditional secrecy* to mean no unjustified expectation exists in any process reachable from P. The "robust" extension of this definition means the process is safe when composed with any opponent process, much as in earlier work [12].

Safety:

A process P is *safe for conditional secrecy* if and only if
whenever $P \rightarrow^* \textbf{new}\ \vec{x}{:}\vec{T}; (\textbf{secret}\ M\ \textbf{amongst}\ N \mid \textbf{out}\ !x\ M :: L \mid Q)$, we have $Q \vdash N \leq L$.

Opponent Processes and Robust Safety:

A process O is **Un**-*typed* if and only if every type occurring in O is **Un**.
Write $\mathsf{erase}(P)$ for the **Un**-typed process given by replacing all types in P by **Un**.
A process O is **secret**-*free* if any only if there are no **secret** expectations in O.
A process $O\{\vec{x}\}$ with $\mathsf{fn}(O\{\vec{x}\}) = \{\vec{x}\}$ is an *opponent* if and only if it is **Un**-typed and **secret**-free.
A process P is *robustly safe for conditional secrecy despite* $\vec{L}$ if and only if
$P \mid O\{\vec{L}\}$ is safe for secrecy for all opponents $O\{\vec{x}\}$.

3 Examples of Secrecy Despite Compromise

The examples in this section illustrate some protocols and their secrecy properties, and also informally introduce some aspects of our type system. We use mostly standard abbreviations for common message and process idioms, such as arbitrary-length tuples. These are much the same as in previous work [13], and are given in [15].

A Basic Example Consider a world with just the two security levels $\top$ and $\bot$. The following processes, at level $\top$, communicate along a shared channel k. (We use the keyword **process** to declare non-recursive process abbreviations.)

process Sender(k:**Ch**(**Secret**{ $\top$ })) =
  new s:**Secret**{ $\top$ }; **out** k!(s) :: $\top$ | **secret** s **amongst** $\top$.
process Receiver(k:**Ch**(**Secret**{ $\top$ })) =
  inp k?(s:**Secret**{ $\top$ }); **secret** s **amongst** $\top$.

The parallel composition Sender(k) | Receiver(k) is robustly safe despite $\varnothing$ but not, for example, despite either {k!} (because the attacker can send public data to falsify the receiver's expectation) or {k?} (because the attacker can obtain the secret s to falsify the sender's expectation).

Our type system can verify the robust safety property of this system based on its type annotations. Messages of type **Secret**$\{L\}$ are secrets at level L. Messages of type **Ch** T are channels for exchanging messages of type T. Later on, we use types ?**Ch** T and !**Ch** T for the input and output capabilities on channels of T messages.

An Example of Secrecy Despite Host Compromise To establish secrecy properties (for example, that A and B share a secret) in the presence of a compromised insider (for example C, who also shares a secret with B) requires more security levels than just $\top$ and $\bot$. For example, consider the following variant on an example of Abadi and Blanchet [3] (rewritten to include the identities of the principals).

```
process Sender(a:Un, b:Un, cA:Type2(a,b), cB:Type1(b)) =
  new k:Secret{a,b}; secret k amongst (a,b);
  new s:Secret{a,b}; secret s amongst (a,b);
  out cB (a, k, cA!) :: a |
  inp cA? (match k, cAB:!Type3(a,b)); // pattern-matching syntax
  out cAB (s) :: a.

process Receiver(b:Un, cB:Type1(b)) =
  inp cB? (a≤a′:Un, k:Secret{a,b}, cA:!Type2(a,b)); // pattern-matching syntax
  new cAB:Type3(a,b);
  out cA (k, cAB!) :: b |
  inp cAB? (z:Secret{a,b}); stop.
```

Here, sender A sends to receiver B a tuple (A,k,cA!), along a trusted output channel cB, whose matching input channel is known only to B. She then waits to receive a message of the form (k,cAB), whose first component matches the freshly generated name k, along the channel cA?, which must have come from B, as only A and B know k. Hence, A knows that cAB is a trusted channel to B, and so it is safe to send s along cAB.

Receiver B runs the matching half of the protocol, but gets much weaker guarantees, as the output channel cB is public, and so anyone (including an attacker) can send messages along it. When B receives (A,k,cA?), he knows that it claims to be from A, and binds a' to A's security level. However, he does not know who the message really came from: it could be A, or it could be an attacker masquerading as A. All B knows is that there is some security level $a \leq a'$ indicating who really sent the message.

When a process such as Receiver receives an input such as (A,k,cA!), it binds two variables $a \leq a'$ reflecting the actual and claimed security level of the message. This is reflected in the dependent type $(\pi x \leq y : T, U)$, which binds two variables in U. The variable x is bound to the actual security level, and the variable y is bound to the claimed security level. At run-time, the binding for x is unknown, so it is restricted to only being used in types, not in messages. In examples, we often elide x when it is unused.

Processes have two ways of accessing a pair: they may use the **split** construct to extract the components of the pair, or they may use the **match** construct to match the first element of the pair against a constant. For example, the Sender process above contains the input **inp** ca?(**match** k, cAB:!Type3(a,b)), which requires the first component to match the known name k, or else fails, and (implicitly) uses **split** to bind the second component to cAB. (We are using pattern-matching abbreviations to avoid introducing

large numbers of temporary variables, as discussed in [15].) These two forms of access to tuples are not new, and have formed the basis of our previous work on typechecking cryptographic protocols [12,13]. What is new is that these forms of access are reflected in the types. We tag fields with a marker π, which is either **split** or **match**, to indicate how they are used.

The types for this example are:

type Type3(a,b) = **Ch** (**Secret**{a,b})
type Type2(a,b) = **Ch** (**match** k:**Secret**{a,b}, **split** cAB:!Type3(a,b))
type Type1(b) = **Ch** (**split** $a \leq a'$:**Un**, **split** k:**Secret**{a,b}, **split** cA:!Type2(a,b))

Given the environment:

A:**Un**, CA:Type2(A,B), B:**Un**, CB:Type1(B), C:**Un**, CC:Type2(C,B)

we can typecheck:

repeat Sender(A,B,CA,CB!) | **repeat** Receiver(B,CB) |
repeat Sender(C,B,CC,CB!) | $C \leq \bot$

Hence, soundness of the type system (Theorem 2) implies the system is robustly safe for secrecy despite {A,B,C,CA!,CB!,CC}. The statement $C \leq \bot$ represents the compromise of C. Thus, A and B are guaranteed to preserve their secrecy, even though compromised C shares a secret CC with B.

An Example of Secrecy Despite Session Compromise Finally, we consider an adaption of the previous protocol to allow for declassification of secrets. Declassification may be deliberate, or it may model the consequences of an exploitable software defect. We regard the session identifier k as a new security level, that may be compromised independently of A and B. We modify the example by allowing the sender to declassify the secret after receiving a message on channel d.

process Sender(a:**Un**, b:**Un**, cA:Type2(a,b), cB:Type1(b), d:**Un**) =
  new k:**Secret**{a,b}; **secret** k **amongst** (a,b);
  new s:**Secret**{a,b,k}; **secret** s **amongst** (a,b,k);
  out cB (a, k, cA!) :: a |
  inp cA? (**match** k, cAB:!Type3(a,b));
  out cAB (s) :: a |
  inp d?(); $k \leq \bot$; **out** d!(s) :: a

Here, the sender declassifies s by the statement $k \leq \bot$. Since k is mentioned in the security level of s, this statement allows s to be published on public channel d. The rest of the system remains unchanged and the types are now:

type Type3(a,b,k) = **Ch** (**Secret**{a,b,k})
type Type2(a,b) = **Ch** (**match** k:**Secret**{a,b}, **split** cAB:!Type3(a,b,k))
type Type1(b) = **Ch** (**split** $a \leq a'$:**Un**, **split** k:**Secret**{a,b}, **split** cA:!Type2(a,b))

Theorem 2 now gives us not only that A and B can maintain secrecy despite compromise of C, but also that it is possible to compromise one session k, and hence declassify the matching secret s, without violating secrecy of the other sessions.

4 A Type System for Checking Conditional Secrecy

A basic idea in our type system is to identify classes of public and tainted types [13]. Intuitively, messages of public type can flow to the opponent, while messages of tainted type may flow from the opponent. More formally, if **Un** is the type of all messages known to the opponent and $<:$ is the subtype relation, a type T is *public* just when $T <:$ **Un**, and a type T is *tainted* just when **Un** $<: T$.

Both classes depend on the security ordering. Just as the attacker encroaches on the compromised parts of a system over time, types may become public or tainted over time. We reflect this dependency syntactically by decorating types with symbolic kinds. A *kind* K is a pair $\{?M, !N\}$ of security levels. A message of a type decorated $\{?M, !N\}$ can be assumed to flow from a source at level M (or higher), and is allowed to flow to a target at level N (or higher). If $M \leq \bot$ the type is tainted; if $N \leq \bot$ the type is public. We often write shorthand such as $\{\mathrm{A}, ?\mathrm{B}, !\mathrm{C}\}$ for the kind $\{?(\mathrm{A},\mathrm{B}), !(\mathrm{A},\mathrm{C})\}$.

Kinds:

$K ::= \{?M, !N\}$	tainted if $M \leq \bot$, public if $N \leq \bot$

Write $\{L_1, \ldots, L_l, ?M_1, \ldots, ?M_m, !N_1, \ldots, !N_n\}$
for $\{?(L_1, \ldots, L_l, M_1, \ldots, M_m), !(L_1, \ldots, L_l, N_1, \ldots, N_n)\}$.

Our language of types consists of standard constructs for channels with optional read-only and write-only attributes, sum types, and **Ok** types [11]. The only non-standard types are the dependent pairs $(\pi x \leq y : T, U)$, discussed previously in Section 3.

Types:

$\nu ::= ? \mid !$	input-only (?) or output-only (!) attribute
$\pi ::=$ **split** $\mid$ **match**	split-only or match-only attribute
$T, U ::=$	type
Ch $K\ T$	channel for T messages
ν**Ch** $K\ T$	input or output capability on channel for T messages
$(\pi x \leq y{:}T, U)$	split-only or match-only dependent pair (scope of x, y is U)
$T + U$	tagged sum type
Ok S	proof of security ordering

Our judgments are defined with respect to an *environment*, a list of all names in scope, paired with their types. A generative type is one that can be freshly generated.

Environments:

$E ::= \varnothing \mid E, x{:}T$	environment: list of name typings
$\mathsf{dom}(\varnothing) = \varnothing \quad \mathsf{dom}(E, x{:}T) = \mathsf{dom}(E) \cup \{x\}$	
$\mathsf{clauses}(\varnothing) = \varnothing \quad \mathsf{clauses}(E, x{:}T) = \mathsf{clauses}(E) \cup \{C_1, \ldots, C_n \mid T \text{ is } \mathbf{Ok}\{C_1, \ldots, C_n\}\}$	

Generative Types and Environments:

Let a type be *generative* if and only if it is a channel type **Ch** $K\ T$.
Let an environment E be *generative* if and only if $E(x)$ is generative for each $x \in \mathsf{dom}(E)$.

Judgments of the Type System:

$E \vdash \diamond$	good environment
$E \vdash \mathit{Public}(T)$	public type: T data may flow to the opponent
$E \vdash \mathit{Tainted}(T)$	tainted type: T data may flow from the opponent
$E \vdash T <: T'$	subtype
$E \vdash M : T$	good message of type T
$E \vdash P$	good process

Next, we present the rules defining these judgments. We rely on several abbreviations.

Abbreviations:

Write E, S for the environment $E, x : \mathbf{Ok}\ S$ where x is fresh.
Write $E \vdash M$ for $E \vdash \diamond$ and $\mathsf{fn}(M) \subseteq \mathsf{dom}(E)$.
Write $E \vdash M \leq N$ for $E \vdash (M, N)$ and $\mathsf{clauses}(E) \vdash M \leq N$.
Write $E \vdash S$ for $E \vdash M \leq N$ for every $(M \leq N) \in S$.
Write $E \vdash M \leftrightarrow N$ for $E \vdash M \leq N$ and $E \vdash N \leq M$.
Write $E \vdash T <:> U$ for $E \vdash T <: U$ and $E \vdash U <: T$.

The following standard rules state that in a good environment, each declared name must be fresh, and each name occurring in a type must be declared previously.

Good Environment:

(Env $\varnothing$)
$$\frac{}{\varnothing \vdash \diamond}$$

(Env x)
$$\frac{E \vdash \diamond \quad x \notin \mathsf{dom}(E) \quad \mathsf{fn}(T) \subseteq \mathsf{dom}(E)}{E, x{:}T \vdash \diamond}$$

The judgments $E \vdash \mathit{Public}(T)$ and $E \vdash \mathit{Tainted}(T)$ formalize the classes of public and tainted types. The rules follow the pattern of previous work [13]. The most interesting rules are those for determining when a dependent pair $(\pi x \leq y{:}T, U)$ is tainted. If data of this type has been received from the opponent, then we know that the real security level of the term is $\bot$, and so when we check U for taintedness, we first replace x by $\bot$. In the case when π is **match**, we can be even more liberal, and add into the environment extra clauses generated by tainting the type T: for example $(\mathbf{match}\ x \leq y{:}\mathbf{Secret}\,\{a\}, \mathbf{Secret}\,\{a\})$ is tainted, because we add the clause $a \leq \bot$ to the environment before checking taintedness of the type $\mathbf{Secret}\,\{a\}$.

Extracting a Set of Clauses from a Tainted Type:

$\mathsf{taint}(\mathbf{Ch}\{?M, !N\}\ T) \triangleq \{M \leq \bot, N \leq \bot\}$
$\mathsf{taint}(\nu\mathbf{Ch}\ \{?M, !N\}\ T) \triangleq \{M \leq \bot\}$
$\mathsf{taint}(\pi x \leq y{:}T, U) \triangleq \mathsf{taint}(T + U) \triangleq \mathsf{taint}(\mathbf{Ok}\ S) \triangleq \varnothing$

Public and Tainted Types:

(Public I/O)
$$\frac{E \vdash M \leftrightarrow \bot \quad E \vdash N \leftrightarrow \bot \quad E \vdash \mathit{Public}(T) \quad E \vdash \mathit{Tainted}(T)}{E \vdash \mathit{Public}(\mathbf{Ch}\ \{?M, !N\}\ T)}$$

(Tainted I/O)
$$\frac{E \vdash M \leftrightarrow \bot \quad E \vdash N \leftrightarrow \bot \quad E \vdash \mathit{Public}(T) \quad E \vdash \mathit{Tainted}(T)}{E \vdash \mathit{Tainted}(\mathbf{Ch}\ \{?M, !N\}\ T)}$$

(Public I)
$$\frac{E \vdash M \quad E \vdash N \leftrightarrow \bot \quad E \vdash \mathit{Public}(T)}{E \vdash \mathit{Public}(?\mathbf{Ch}\ \{?M, !N\}\ T)}$$

(Tainted I)
$$\frac{E \vdash M \leftrightarrow \bot \quad E \vdash N \quad E \vdash \mathit{Tainted}(T)}{E \vdash \mathit{Tainted}(?\mathbf{Ch}\ \{?M, !N\}\ T)}$$

(Public O)
$$\frac{E \vdash M \quad E \vdash N \leftrightarrow \bot \quad E \vdash \mathit{Tainted}(T)}{E \vdash \mathit{Public}(!\mathbf{Ch}\ \{?M, !N\}\ T)}$$

(Tainted O)
$$\frac{E \vdash M \leftrightarrow \bot \quad E \vdash N \quad E \vdash \mathit{Public}(T)}{E \vdash \mathit{Tainted}(!\mathbf{Ch}\ \{?M, !N\}\ T)}$$

(Public Split)
$$\frac{E \vdash \mathit{Public}(T) \quad E, x{:}T, y{:}T, x \le y \vdash \mathit{Public}(U)}{E \vdash \mathit{Public}((\mathbf{split}\ x \le y{:}T, U))}$$

(Tainted Split)
$$\frac{E \vdash \mathit{Tainted}(T) \quad E, y{:}T \vdash \mathit{Tainted}(U\{x \leftarrow \bot\})}{E \vdash \mathit{Tainted}((\mathbf{split}\ x \le y{:}T, U))}$$

(Public Match)
$$\frac{E \vdash \mathit{Public}(T) \quad E, x{:}T, y{:}T, x \le y \vdash \mathit{Public}(U)}{E \vdash \mathit{Public}((\mathbf{match}\ x \le y{:}T, U))}$$

(Tainted Match)
$$\frac{E, \mathsf{taint}(T) \vdash \mathit{Tainted}(T) \quad E, y{:}T, \mathsf{taint}(T) \vdash \mathit{Tainted}(U\{x \leftarrow \bot\})}{E \vdash \mathit{Tainted}((\mathbf{match}\ x \le y{:}T, U))}$$

(Tainted Sum)
$$\frac{E \vdash \mathit{Tainted}(T) \quad E \vdash \mathit{Tainted}(U)}{E \vdash \mathit{Tainted}(T + U)}$$

(Public Sum)
$$\frac{E \vdash \mathit{Public}(T) \quad E \vdash \mathit{Public}(U)}{E \vdash \mathit{Public}(T + U)}$$

(Public Order)
$$\frac{E \vdash \diamond \quad \mathsf{fn}(S) \subseteq \mathsf{dom}(E)}{E \vdash \mathit{Public}(\mathbf{Ok}\ S)}$$

(Tainted Order)
$$\frac{E \vdash \diamond \quad E \vdash M_i \leftrightarrow \bot \quad E \vdash N_i \quad \forall i \in 1..n}{E \vdash \mathit{Tainted}(\mathbf{Ok}\ \{M_1 \le N_1, \ldots, M_n \le N_n\})}$$

The rules for subtyping are mostly taken from [13]. The main exception is the rule for $(\mathbf{match} x \le y{:}T, U)$, which requires an extra condition to ensure that subtyping preserves the taint function used in the definition of $E \vdash \mathit{Tainted}(T)$.

Subtyping:

(Sub Public/Tainted)
$$\frac{E \vdash \mathit{Public}(T) \quad E \vdash \mathit{Tainted}(U)}{E \vdash T <: U}$$

(Sub I/O)
$$\frac{E \vdash M \leftrightarrow M' \quad E \vdash N \leftrightarrow N' \quad E \vdash T <:> T'}{E \vdash \mathbf{Ch}\ \{?M, !N\}\ T <: \mathbf{Ch}\ \{?M', !N'\}\ T'}$$

(Sub I)
$$\frac{E \vdash M' \le M \quad E \vdash N \le N' \quad E \vdash T <: T'}{E \vdash ?\mathbf{Ch}\ \{?M, !N\}\ T <: ?\mathbf{Ch}\ \{?M', !N'\}\ T'}$$

(Sub O)
$$\frac{E \vdash M' \le M \quad E \vdash N \le N' \quad E \vdash T' <: T}{E \vdash !\mathbf{Ch}\ \{?M, !N\}\ T <: !\mathbf{Ch}\ \{?M', !N'\}\ T'}$$

(Sub Split)
$$\frac{\begin{array}{c} E \vdash T <: T' \\ E, x{:}T, y{:}T, x \le y \vdash U <: U' \end{array}}{E \vdash (\mathbf{split}\ x \le y{:}T, U) <: (\mathbf{split}\ x \le y{:}T', U')}$$

(Sub Match)
$$\frac{\begin{array}{c} E \vdash T <: T' \quad E, \mathsf{taint}(T') \vdash \mathsf{taint}(T) \\ E, x{:}T, y{:}T, x \le y \vdash U <: U' \end{array}}{E \vdash (\mathbf{match}\ x \le y{:}T, U) <: (\mathbf{match}\ x \le y{:}T', U')}$$

(Sub Sum)
$$\frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash T + U <: T' + U'}$$

(Sub Hierarchy)
$$\frac{E \vdash \diamond \quad E, S \vdash S'}{E \vdash \mathbf{Ok}\ S <: \mathbf{Ok}\ S'}$$

To illustrate the judgments defined so far, we derive the types **Un** and **Secret** K, used already in examples. (Our examples rely also on standard abbreviations, such as tuple types encoded using pair types. Full details are in [15].)

Abbreviations for Un and Secret K:

$\mathbf{Un} \triangleq \mathbf{Ch}\ \{\bot\}\ (\mathbf{Ok}\ \{\})$	generative type of messages known to opponent
$\mathbf{Secret}\,K \triangleq ?\mathbf{Ch}\ K\ \mathbf{Un}$	type of secrets at kind K

Given these derived types, the four types $\mathbf{Secret}\,\{?M, !N\}$ where $M, N \in \{\top, \bot\}$ have the following properties, assuming that $\bot < \top$. Moreover, the subtype ordering induces a diamond lattice, with Any at the top, and Empty at the bottom. The Empty type is uninhabited, and the remaining inhabited types are exactly those of Abadi [1].

The Four Types $\mathbf{Secret}\,\{?M, !N\}$ with $M, N \in \{\top, \bot\}$:

$\mathrm{Any} \triangleq \mathbf{Secret}\,\{?\bot, !\top\}$	tainted, not public
$\mathrm{Pub} \triangleq \mathbf{Secret}\,\{?\bot, !\bot\}$	tainted, public
$\mathrm{Sec} \triangleq \mathbf{Secret}\,\{?\top, !\top\}$	not tainted, not public
$\mathrm{Empty} \triangleq \mathbf{Secret}\,\{?\top, !\bot\}$	not tainted, public

Next, here are the type assignment rules for messages.

Good Message:

(Msg Subsum)
$$\frac{E \vdash M : T \quad E \vdash T <: T'}{E \vdash M : T'}$$

(Msg x)
$$\frac{E \vdash \diamond \quad (x{:}T) \in E}{E \vdash x : T}$$

(Msg I)
$$\frac{E \vdash L : \mathbf{Ch}\ \{?M, !N\}\ T}{E \vdash L? : ?\mathbf{Ch}\ \{M\}\ T}$$

(Msg O)
$$\frac{E \vdash L : \mathbf{Ch}\ \{?M, !N\}\ T}{E \vdash L! : !\mathbf{Ch}\ \{N\}\ T}$$

(Msg Pair)
$$\frac{E \vdash M : T \quad E \vdash N : U\{x{\leftarrow}M\}\{y{\leftarrow}M\}}{E \vdash (M, N) : (\pi x \le y{:}T, U)}$$

(Msg $\bot$)
$$\frac{E \vdash \diamond}{E \vdash \bot : \mathbf{Un}}$$

(Msg Inl)
$$\frac{E \vdash M : T \quad \mathsf{fn}(U) \subseteq \mathsf{dom}(E)}{E \vdash \mathbf{inl}\ M : T + U}$$

(Msg Inr)
$$\frac{E \vdash N : U \quad \mathsf{fn}(T) \subseteq \mathsf{dom}(E)}{E \vdash \mathbf{inr}\ N : T + U}$$

(Msg Ok)
$$\frac{E \vdash \diamond \quad E \vdash S}{E \vdash \top : \mathbf{Ok}\ S}$$

The type rules for processes are standard, with two exceptions. The rule for output performs an extra check on the security level of the output, to ensure that the data can be published at that level: the assumption $E, L \le \bot \vdash \mathit{Public}(T)$ can be read "if the level L were compromised, the type T would be public". The rule for composition typechecks each component in an environment extended with any top-level statement $M \le N$ occurring in the other component.

Extracting Environments from Processes:

$\mathsf{env}(P \mid Q) = \mathsf{env}(P), \mathsf{env}(Q)$
$\mathsf{env}(\mathbf{repeat}\ P) = \mathsf{env}(P)$
$\mathsf{env}(M \le N) = x{:}\mathbf{Ok}\ \{M \le N\}$ for fresh x

$\mathsf{env}(\mathbf{new}\ x{:}T;P) = y{:}T, \mathsf{env}(P\{x{\leftarrow}y\})$ for fresh y
$\mathsf{env}(P) = \varnothing$ otherwise

Good Process:

(Proc Output)
$$\frac{E, L \leq \bot \vdash \mathit{Public}(T) \quad E \vdash M : !\mathbf{Ch}\ K\ T \quad E \vdash N : T}{E \vdash \mathbf{out}\ M\ N :: L}$$

(Proc Input)
$$\frac{E \vdash M : ?\mathbf{Ch}\ K\ T \qquad E, x{:}T \vdash P}{E \vdash \mathbf{inp}\ M(x{:}T); P}$$

(Proc Res)
$$\frac{E, x{:}T \vdash P \quad T \text{ generative}}{E \vdash \mathbf{new}\ x{:}T; P}$$

(Proc Repl)
$$\frac{E \vdash P}{E \vdash \mathbf{repeat}\ P}$$

(Proc Par Mutual)
$$\frac{E, \mathsf{env}(Q) \vdash P \quad E, \mathsf{env}(P) \vdash Q}{E \vdash P \mid Q}$$

(Proc Stop)
$$\frac{E \vdash \diamond}{E \vdash \mathbf{stop}}$$

(Proc Split)
$$\frac{\begin{array}{l} x \notin \mathsf{fn}(\mathsf{erase}(P)) \\ E \vdash M : (\mathbf{split}\ x \leq y{:}T, U) \\ E, x{:}T, y{:}T, x \leq y, z{:}U \vdash P \end{array}}{E \vdash \mathbf{split}\ M\ \mathbf{is}\ (x \leq y{:}T, z{:}U); P}$$

(Proc Match)
$$\frac{\begin{array}{l} x \notin \mathsf{fn}(\mathsf{erase}(P)) \\ E \vdash M : (\mathbf{match}\ x \leq y{:}T, U) \quad E \vdash N : T \\ E, x{:}T, x \leq N, z{:}U\{x{\leftarrow}N\} \vdash P \end{array}}{E \vdash \mathbf{match}\ M\ \mathbf{is}\ (x \leq N, z{:}U\{y{\leftarrow}N\}); P}$$

(Proc Case)
$$\frac{E \vdash M : T + U \quad E, x{:}T \vdash P \quad E, y{:}U \vdash Q}{E \vdash \mathbf{case}\ M\ \mathbf{is\ inl}\ (x{:}T)\ P\ \mathbf{is\ inr}\ (y{:}U)\ Q}$$

(Proc Clause)
$$\frac{E \vdash M \quad E \vdash N}{E \vdash M \leq N}$$

(Proc Secret Cap)
$$\frac{E \vdash M : \nu\mathbf{Ch}\ \{?L\}\ T}{E \vdash \mathbf{secret}\ M\ \mathbf{amongst}\ L}$$

We can now state the main result of the paper, that the type system is sound with respect to robust safety. (Proofs are in [15].)

Theorem 1 (Safety). *If $E \vdash P$ and E is generative then P is safe for conditional secrecy.*

Theorem 2 (Robust Safety). *If $E \vdash P$, E is generative, and $E \vdash \vec{M} : \mathbf{Un}$ then P is robustly safe for conditional secrecy despite $\vec{M}$.*

5 An Extended Calculus with Symbolic Cryptography (Outline)

To express cryptographic protocols, we can add symbolic encryption and decryption operations to our core calculus to obtain a form of the spi-calculus [5]. We can easily extend our type system to accommodate these operations, much as in previous work [13]; for example, encryption and decryption keys are treated analogously to the output and input capabilities in our core calculus. Somewhat surprisingly, we can prove soundness of the extended type system by a straightforward translation into the core calculus. Keys are translated to channels, encryption keys to output channels, decryption keys to input channels, and ciphertexts to the constant $\bot$. The translation is not fully abstract, but preserves typings and reflects safety, which suffices to establish that well-typed spi-calculus processes are robustly safe. (The companion report [15] has full details of the extended calculus, type system, and the translation.) As an example of using the extended calculus, consider Lowe's variant of the Needham–Schroeder public key protocol:

Message 1. A $\rightarrow$B: $\{|$msg1(A,sA)$|\}$kB
Message 2. B $\rightarrow$A: $\{|$msg2(B,sA,sB)$|\}$kA
Message 3. A $\rightarrow$B: $\{|$msg3(sB)$|\}$kB

In [15] we show that this protocol robustly preserves conditional secrecy of sA and sB amongst $\{A,B\}$, in the presence of compromised insiders. The proof is based on the type for a key for use by principal p:

type NS(p) = **Key**(msg1(**split** a$\leq$a′:**Un**, **split** sa:**Secret**{a,p})
| msg2(**match** b:**Un**, **match** sa:**Secret**{p,b}, **split** sb:**Secret**{p,b})
| msg3(**match** sb:**Secret**{?$\perp$,!p})).

Abadi and Blanchet [3] consider the same protocol, under similar assumptions of compromise, but rely on two separate typing derivations to prove the secrecy of sA and sB.

6 Related Work

Abadi [1] proposes the use of security types for establishing secrecy properties in cryptographic protocols expressed in the spi-calculus [5]. Abadi takes a fixed, binary view of security, where the world is divided into system and attacker, and a secret is something the attacker does not have. We are the first to generalize his work to multiple security levels and to allow the boundary between system and attacker to shift as levels are created and compromised. Another generalization of Abadi's work is the type system of Bugliesi, Focardi, and Maffei [8], which checks security properties in the presence of a fixed set of compromised hosts, but assumes this set is known during typechecking.

Abadi's type system establishes an equationally-defined secrecy property of Abadi and Gordon [5], that prevents some indirect flows as well as direct flows. Our expectations of conditional secrecy generalize the notion of explicit flow introduced by Abadi [2], and since used in several papers on process calculi [6,9].

The decentralized label model (DLM) of Myers and Liskov [18] is the basis of the Jif language in which security types track ownership and possible compromise of data. DLM policies govern which principals can downgrade data—the system of the present paper does not address this question. A "declassify" expression converts the level of a whole expression, but it does not alter the security ordering. Since they convert high data into low data, programs using declassification typically falsify noninterference properties; there have been several proposals of modified noninterference properties to handle declassification [22].

Pottier and Simonet's Flow Caml [23], has global, static declarations of flows, but no local or dynamic declarations.

Two recent papers consider dynamic additions to the security ordering. Boudol and Matos [7] introduce block-structured declarations of orderings, in which edges may temporarily be added to the security ordering. They present a type and effect system that establishes a form of noninterference. They do not consider dynamic creation of security levels and they do not associate levels with code. Tse and Zdancewic [24] consider dynamic creation and communication of principal identities, and propose a delegation operation that allows temporary modification of the lattice of security levels.

We mention a couple of the many studies of security orderings within process calculi. Hennessy and Riely [20] study mobile agents migrating between locations, that may or may not be compromised. By a combination of static and dynamic checks they prevent type violations at uncompromised sites. Hoshina, Sumii, and Yonezawa [16] introduce a security order between protection domains in a process calculus. They use a type system with dependent types to prevent access violations. To the best of our knowledge, the present paper is the first to consider runtime compromise of security levels in the setting of a process calculus.

Finally, many of the techniques for the Dolev-Yao model other than type systems deal with host compromise and insider attacks; type systems such as ours do require some human effort to construct type annotations, but given these annotations admit automatic, efficient protocol checking.

7 Conclusion

This paper introduces a mutable security ordering into a process calculus, in order to model a dynamically growing population of principals, some of which may become compromised. We advocate the placement of conditional secrecy annotations in processes to express containment of compromise; that particular messages are kept secret, unless particular principals are compromised. We describe a type system for checking that no opponent can interact with the system to falsify these annotations. As well as proving a soundness theorem for the type system, we assess our proposal by exhibiting a series of typed examples, showing an improvement over prior work. Our system verifies versions of all the examples considered by Abadi and Blanchet [3] (modified to include multiple principals, and multiple simultaneous runs of the protocols).

We end by discussing three criticisms. First, our present system tracks only secrecy properties. We expect it is possible to combine our system with prior constructs expressing authentication and authorization properties [10,14]. Second, our type system allows any process to augment any part of the security ordering. This is acceptable in short programs modelling cryptographic protocols, but for larger programs there should be an enforceable policy governing additions to the security ordering. Prior work on policies for declassification may be applicable. Third, our type-based verification method requires the programmer to supply type annotations. A type inference algorithm would lessen this burden, although the lack of principal types would make such an algorithm non-trivial. A complementary approach may be to adapt logic programming interpretations of the pi-calculus [4] to obtain a logic-based method for checking conditional secrecy. We leave these directions for future work.

Acknowledgements We thank Gérard Boudol, Ana Matos, Andrei Sabelfeld, and Dave Sands for sending us previews of their CSFW'05 papers [7,22]. Thanks also to Tony Hoare and the anonymous reviewers for useful comments.

References

1. M. Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, Sept. 1999.
2. M. Abadi. Security protocols and their properties. In *Foundations of Secure Computation*, pages 39–60. IOS Press, Amsterdam, 2000.
3. M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theoretical Comput. Sci.*, 298(3):387–415, 2003.
4. M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. *Journal of the ACM*, 52(1):102–146, 2005.
5. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999.
6. B. Blanchet. From secrecy to authenticity in security protocols. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *LNCS*, pages 242–259. Springer, 2002.
7. G. Boudol and A. Matos. On declassification and the non-disclosure policy. In *18th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2005. To appear.
8. M. Bugliesi, R. Focardi, and M. Maffei. Authenticity by tagging and typing. In *Formal Methods in Security Engineering (FMSE'04)*, pages 1–12, 2004.
9. L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. *Information and Computation*, 196(2):127–155, 2005.
10. C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. In *European Symposium on Programming (ESOP'05)*, LNCS, pages 141–156. Springer, 2005.
11. A. D. Gordon and A. Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In *Software Security—Theories and Systems*, volume 2609 of *LNCS*, pages 270–282. Springer, 2002.
12. A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–521, 2003.
13. A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3/4):435–484, 2003.
14. A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300:379–409, 2003.
15. A. D. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. Technical Report MSR–TR–2005–76, Microsoft Research, 2005.
16. D. Hoshina, E. Sumii, and A. Yonezawa. A typed process calculus for fine-grained resource access control in distributed computation. In *4th International Symposium on Theoretical Aspects of Computer Software (TACS 2001)*, volume 2215 of *LNCS*, pages 64–81. Springer, 2001.
17. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. CUP, 1999.
18. A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.
19. M. Odersky. Polarized name passing. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 324–335. Springer, 1995.
20. J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *26th ACM Symposium on Principles of Programming Languages*, pages 93–104, 1999.
21. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
22. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2005. To appear.
23. V. Simonet. The Flow Caml system: documentation and user's manual. Technical Report 0282, INRIA, 2003.
24. S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 2004.