Cryptographically Sound Implementations for Communicating Processes (Extended Abstract)

Pedro Adão[1]* and Cédric Fournet[2]

[1] Center for Logic and Computation, IST, Lisboa, Portugal
[2] Microsoft Research

Abstract. We design a core language of principals running distributed programs over a public network. Our language is a variant of the pi calculus, with secure communications, mobile names, and high-level certificates, but without any explicit cryptography. Within this language, security properties can be conveniently studied using trace properties and observational equivalences, even in the presence of an arbitrary (abstract) adversary.
With some care, these security properties can be achieved in a concrete setting, relying instead on standard cryptographic primitives and computational assumptions, even in the presence of an adversary modeled as an arbitrary probabilistic polynomial-time algorithm. To this end, we develop a cryptographic implementation that preserves all properties for all safe programs. We give a series of soundness and completeness results that precisely relate the language to its implementation. We also illustrate our approach using a series of protocols and properties expressible in our language, and motivate some unusual design choices.

1 Secure Implementations of Communications Abstractions

When designing and verifying security protocols, a certain level of idealization is needed to provide manageable mathematical treatment. Accordingly, two views of cryptography have been developed over the years. In the first view, cryptographic protocols are expressed algebraically, within simple languages. This formal view is suitable for automated computer tools, but is also arguably too abstract. In the second view, cryptographic primitives are probabilistic algorithms that operate on bitstrings. This view involves probabilities and limits in computing power; it is harder to handle formally, especially when dealing with large protocols. Getting the best of both views is appealing, and is the subject of active research that aims at building security abstractions with formal semantics and sound computational implementations.

In this work, we develop a first sound and complete implementation of a distributed process calculus. Our calculus is a variant of the pi calculus; it provides name mobility, reliable messaging and authentication primitives, but neither explicit cryptography nor probabilistic behaviors. Taking advantage of concurrency theory, it supports simple reasoning, based on labeled transitions and observational equivalence. We precisely define its concrete implementation in a computational setting. We establish general soundness and completeness results in the presence of active adversaries, for both trace properties and observational equivalences, essentially showing that high level reasoning accounts

* Partially supported by FCT grant SFRH/BD/8148/2002. Additional support from FEDER/FCT project QuantLog POCI/MAT/55796/2004.

for all low-level adversaries. We illustrate our approach by coding security protocols and establishing their computational correctness by simple formal reasoning.

We implement high-level functionalities using cryptography, not high-level views of cryptographic primitives. In the spirit of recent related works, we could instead have proceeded in two steps, by first compiling high-level communications to an intermediate calculus with ideal, explicit cryptography (in the spirit of [3,2]), then establishing the computational soundness of this calculus with regards to computational cryptography. However, this second step is considerably more delicate than our present goal, inasmuch as one must provide a sound implementation for an arbitrary usage of ideal cryptography. In contrast, for instance, our language keeps all keys implicit, so no high-level program may ever leak a key or create an encryption cycle. (We considered targeting existing idealized cryptographic frameworks with soundness theorems, but their reuse turned out to be more complex than a direct implementation.)

Our concrete implementation relies on standard cryptographic primitives, computational security definitions, and networking assumptions. It also combines typical distributed implementation mechanisms (abstract machines, marshaling and unmarshaling, multiplexing, and basic communications protocol.) This puts interesting design constraints on our high-level semantics, as we need to faithfully reflect their properties and, at the same time, be as abstract as possible. In particular, our high-level environments should be given precisely the same capabilities as low-level probabilistic polynomial-time (PPT) adversaries. For example, our language supports abstract reliable messaging: message senders and receivers are authenticated, message content is protected, and messages are delivered at most once. On the other hand, under the conservative assumption that the adversary controls the network, we cannot guarantee message delivery, nor implement private channels (such that some communications may be undetected). Hence, the simple rule $\overline{c}\langle M\rangle.P \mid c(x).Q \rightarrow P \mid Q\{M/x\}$, which models silent communication "in the ether" for the pi calculus, is too abstract for our purposes. (For instance, if P and Q are implemented on different machines connected by a public network, and even if c is a restricted channel, the adversary can simply block all communications.) Instead, we design high-level rules for communications between explicit principals, mediated by an adversary, with abstract labels that enable the environment to perform traffic analysis but not forge messages or observe their payload. Similarly, process calculi feature non-deterministic infinite computations, and we will need to curb these features to meet our low-level complexity requirements.

Contents This paper is organized as follows. Section 2 defines our low-level target model. Section 3 presents our high-level language and semantics. Section 4 defines and illustrates high-level equivalences. Section 5 outlines our concrete implementation. Section 6 states our soundness and correctness theorems. Section 7 concludes. Appendix A recalls cryptographic definitions. Appendix B develops applications of our approach. Appendix C details our low-level implementation. A companion paper [8] provides the proofs for our main results, with an interesting combination of techniques from process calculi and cryptography.

Related Work Within formal cryptography, process calculi and process algebras are widely used to model security protocols. For example, the spi calculus of Abadi and

Gordon [4] neatly models secret keys and fresh nonces using names and their dynamic scopes. Representing active attackers as pi calculus contexts, one can state (and prove) trace properties and observational equivalences that precisely capture the security goals for these protocols. Automated provers (e.g. [14]) also help verify these goals.

Abadi, Fournet, and Gonthier developed distributed implementations for variants of the join calculus, with high-level security but no cryptography, roughly comparable to our high-level language. Their implementation is coded within a lower-level calculus with formal cryptography. They established full abstraction for observational equivalence [3,2]. Our general approach is similar, but our implementation is considerably more concrete. Also, due to the larger distance between high-level processes and low-level computational machines, soundness results are more demanding. Abadi and Fournet also recently proposed a labeled semantics for traffic analysis, in the context of a pi calculus model of a fixed protocol for private authentication [1].

The computational soundness of formal cryptography is an active area of research, with many recent results for languages including selected cryptographic primitives. Abadi and Rogaway initially considered formal encryption against passive attackers [6] and established the soundness of indistinguishability; [5,7] extend their results. Backes, Pfitzmann and Waidner [11] achieved a first soundness result with active attackers, initially for public-key encryption and digital signatures. They extended their result to symmetric authentication [12] and encryption [10]. Micciancio and Warinschi [19] also established soundness in the presence of active attacks, under different simpler assumptions. These models enable cryptographic proofs for complex protocols [9,16].

Other works also develop computationally-sound implementations of more abstract security functions on top of cryptography. For example, Canetti and Krawczyk build computational abstractions of secure channels in the context of key-exchanges protocols, with modular implementations and establish sufficient conditions to realize these channels [15]. Targeting the idealized cryptographic model of [11], Laud [17] implements a deterministic process calculus and establishes the computational soundness of a type system for secrecy.

Another interesting approach is to supplement process calculi with concrete probabilistic or polynomial-time semantics. Unavoidably, reasoning on processes become more difficult. For example, Lincoln, Mitchell, Mitchell, and Scedrov [18] introduce a probabilistic process algebra for analyzing security protocols, such that parallel contexts coincide with probabilistic polynomial-time adversaries. In this framework, further extended by Mitchell, Ramanathan, Scedrov, and Teague [20], they develop an equational theory and bisimulated-based proof techniques.

2 Low-Level Target Model

Before presenting our language design and implementation, we specify the target systems. (We recall the underlying notions of cryptography in Appendix A.)

We consider systems that consist of a finite number of communicating principals $a, b, c, \dots \in \mathsf{Prin}$. Each principal is meant to run its own program, written in our high-level language and implemented as a PPT machine defined in Appendix C. Each machine M_a has two wires, $?\mathbf{input}_a$ and $!\mathbf{output}_a$, representing a basic network interface. When activated, the machine reads a bitstring from $?\mathbf{input}_a$, performs some local computation, then writes a bitstring on $!\mathbf{output}_a$ and yields. The machine embeds encryption, signing and random number generator algorithms—thus defining random variables. The machine is also parameterized by a security parameter $\eta \in \mathbb{N}$—intuitively, the length for all keys—thus defining an ensemble of probability.

Some of these machines may be corrupted, under the control of the attacker; their implementation is then unspecified and treated as part of the attacker. We let $a, b \in \mathcal{H}$ with $\mathcal{H} \subset \mathsf{Prin}$ range over principals that comply with our implementation, and let $\mathsf{M} = (\mathsf{M}_a)_{a \in \mathcal{H}}$ describe our whole system—of course, the implementation of M_a does not rely on any knowledge of $\mathcal{H}$.

The adversary, A, is a PPT algorithm that controls the network, the global scheduler, and some compromised principals. At each moment, only one machine is active: whenever an adversary delivers a message to a principal, this principal is activated, runs until completion, and yields an output to the adversary.

Definition 1 (Run). *A run of* A *and* M *with security parameter* $\eta \in \mathbb{N}$ *goes as follows:*

1. *key materials are generated for every principal* $a \in \mathsf{Prin}$*;*
2. *every* M_a *is activated with* 1^η*,* a*'s keys, and the public keys for all* $b \in \mathsf{Prin}$*;*
3. A *is activated with* 1^η*, all keys for* $e \in \mathsf{Prin} \setminus \mathcal{H}$*, and the public keys for* $a \in \mathcal{H}$*;*
4. A *performs a series of low-level exchanges, as follows:*
   - A *writes a bitstring on any wire* $?\mathbf{input}_a$ *and activates* M_a*;*
   - *upon completion,* A *reads a bitstring on* $!\mathbf{output}_a$*;*
5. A *returns a result* s*, written* $s \longleftarrow \mathsf{A}[\mathsf{M}]$.

To study their security properties, we compare systems that consist of machines running on behalf of the same principals $\mathcal{H} \subseteq \mathsf{Prin}$, but with different internal programs and states. Intuitively, two systems are equivalent when no adversary, starting with the information normally given to other principals, can distinguish between their two behaviors, except with negligible probability (written $\mathrm{neg}(\eta)$). Our goal is to develop a simpler, higher-level semantics that entail such equivalences.

Definition 2 (Low-Level Equivalence). *Two PPT systems* M^0 *and* M^1 *are equivalent, written* $\mathsf{M}^0 \approx_\eta \mathsf{M}^1$*, when for every PPT adversary* A*, we have* $|\Pr[1 \longleftarrow \mathsf{A}[\mathsf{M}^0]_\eta] - \Pr[1 \longleftarrow \mathsf{A}[\mathsf{M}^1]_\eta]| \leq \mathrm{neg}(\eta)$.

3 A Distributed Calculus with Principals and Authentication

We now present our high-level language. We successively define terms, patterns, processes, configurations, and systems. We then give their operational semantics. Although some aspects of the design are unusual, the resulting calculus is still reasonably abstract and convenient for distributed programming.

Let Prin be a finite set of *principal identities* and Name be a countable set of *names* disjoint from Prin. Let `f` range over a finite number of function symbols, each with a fixed arity $k \geq 0$. Terms and patterns are defined by the following grammar:

$V, W ::=$	terms
x, y	variable
$m, n \in \mathsf{Name}$	name
$a, b \in \mathsf{Prin}$	principal identity
$\mathtt{f}(V_1, \ldots, V_k)$	constructed term (when $\mathtt{f}$ has arity k)
$T, U ::=$	patterns
$?x$	variable (binds x)
$T\ \mathit{as}\ ?x$	alias (binds x to the term that matches T)
V	constant pattern
$\mathtt{f}(T_1, \ldots, T_k)$	constructed pattern (when $\mathtt{f}$ has arity k)

Names and principal identities are atoms, or "pure names", which may be compared with one another but otherwise do not have any structure. Constructed terms represent structured data, much like algebraic data types in ML or discriminated unions in C. They can represent constants and tags (when $k = 0$), tuples, and formatted messages. As usual, we write `tag` and (V_1, V_2) instead of $\mathtt{tag}()$ and $\mathtt{pair}(V_1, V_2)$. Patterns are used for analyzing terms and binding selected subterms to variables. For instance, the pattern $(\mathtt{tag}, ?x)$ matches any pair whose first component is `tag` and binds x to its second component. We write _ for a variable pattern binding a fresh variable.

Local processes represent the active state of principals, with the following grammar:

$P, Q, R ::=$	local processes
V	asynchronous output
$(T).Q$	input (binds $bv(T)$ in Q)
$*(T).Q$	replicated input (binds $bv(T)$ in Q)
$\mathtt{match}\ V\ \mathtt{with}\ T\ \mathtt{in}\ Q\ \mathtt{else}\ Q'$	match (binds $bv(T)$ in Q)
$\nu n.P$	name restriction ("new", binds n in P)
$P \mid P'$	parallel composition
$\mathbf{0}$	inert process

The asynchronous output V is just a pending message; its data structure is explained below. The input $(T).Q$ waits for a message that matches T then runs Q. The replicated input $*(T).Q$ behaves similarly but it can consume any number of outputs that match T, and fork a copy of Q for each of them. The match process runs Q if V matches T, and runs Q' otherwise. The name restriction creates a fresh name n then runs P. Parallel composition represents processes that run in parallel, with the inert process **0** as unit.

A local context is a process with a hole instead of a subprocess; it is an evaluation context (resp. a guarded context) when the hole replaces a subprocess P or P' (resp. Q or Q') in the grammar above. Free and bound names and variables for terms, patterns, and processes are defined as usual: x is bound in T if $?x$ occurs in T; n is bound in $\nu n.P$. x is free in T if it occurs in T and is not bound in T. An expression is closed when it has no free variables; it may have free names.

Our language features two forms of authentication, represented as two constructors `auth` and `cert` of arity 3 plus well-formed conditions on their usage in processes.

– Authenticated messages between principals are represented as terms of the form $\mathtt{auth}(V_1, V_2, V_3)$, written $V_1{:}V_2\langle V_3\rangle$, where V_1 is the sender, V_2 the receiver, and

V_3 the content. We let M and N range over messages. The message M is *from* a (respectively *to* a) if a is the sender (respectively the receiver) of M.

– Certificates issued by principals are represented as terms of the form $\mathtt{cert}(V_1, V_2, V_3)$, written $V_1\{V_2\}_{V_3}$, where V_1 is the issuer, V_2 the content, and V_3 the label. Labels in certificates reflect cryptographic signature values in their implementation. They are often unimportant (and omitted), since our processes use a constant label 0 in their certificates and ignore labels (using _) in their certificate patterns. Nonetheless, they are necessary to account for the cryptographic possibility of generating different signature values for certificates with identical issuer and content.

Although both authenticated messages and certificate provide some form of authentication, authenticated messages are delivered at most once, to their designated receiver, whereas certificates can be freely copied and forwarded. Hence, our certificates conveniently represent transferable credentials and capabilities. They may be used, for instance, to code decentralized access-control mechanisms.

As an example, $a{:}b\langle\mathtt{Hello}\rangle$ is an (authentic) message from a to b with content Hello, a constructor with arity 0, whereas $a\{b, \mathtt{Hello}\}$ is a certificate with the same subterms that can be sent, received, and verified by any principal.

Let $\phi(V)$ be the set of certificates included in V and let $\phi(V)_X \subseteq \phi(V)$ be those certificates issued by $x \in X$. The process P is *well-formed for* $a \in \mathsf{Prin}$ when

1. any certificate in P that includes a variable or a bound name is of the form $a\{V_2\}_{\mathtt{0}}$;
2. no pattern in P binds any certificate label; and
3. no pattern used for input in P matches any authenticated message not sent to a.

Condition 1 states that the process may produce certificates only with issuer a; in addition, the process may contain previously-received certificates issued by other principals. (We do not restrict certificate patterns—a pattern that tests a certificate not available to a will never be matched.) Condition 2 restricts access to labels, which only affect certificates comparisons. Condition 3 prevents that authenticated messages sent by a be read back by some local input, thereby ensuring that routing is deterministic.

Finally, systems and configurations represent assemblies of communicating principals, with the following grammar:

$C ::=$	configurations
$\quad a[P]$	principal a with local state P
$\quad M/i$	intercepted message i with content M
$\quad C \mid C'$	distributed parallel composition
$\quad \nu n.C$	name restriction ("new", binds n in C)
$S ::=$	systems
$\quad \Phi \vdash C$	configuration C exporting certificates Φ

A configuration is an assembly of running principals, each with its own local state, plus an abstract record of the messages intercepted by the environment and not forwarded yet to their intended recipients. A system is a top-level configuration plus an abstract record of the environment's knowledge, as a set of certificates previously issued and sent to the environment by the principals in C.

We rely on well-formed conditions: in configurations, intercepted messages have distinct identifiers i, and principals have distinct identities a and well-formed local processes P_a for a. In systems, let $\mathcal{H}$ be the set of identities for all defined principals, called *compliant principals*. All intercepted messages are from a to b for some $a, b \in \mathcal{H}$ with $a \neq b$, and Φ is a set of certificates with label `0` such that $\phi(\Phi)_{\mathcal{H}} = \Phi$.

Operational Semantics—Local Reductions Our high-level semantics is defined in two stages: local reductions between processes, then global labeled transitions between systems and their (adverse) environment. Processes, configurations, and systems are considered up to renaming of bound names and variables.

Structural equivalence, written $P \equiv P'$, represents structural rearrangements for local processes. As usual in the pi calculus, it is defined as the smallest equivalence such that $P \equiv P \mid 0$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$, $(\nu n.P) \mid Q \equiv \nu n.(P \mid Q)$ when $n \notin fn(Q)$, $\nu m.\nu n.P \equiv \nu n.\nu m.P$, and $\nu n.\mathbf{0} \equiv \mathbf{0}$. Intuitively, structural rearrangements are not observable (although this is quite hard to implement).

Local reduction step, written $P \rightarrow_a P'$, represents internal computation between well-formed local processes running on behalf of principal a. It is defined as the smallest relation such that

(LCOMM) $\quad (T).Q \mid T\sigma \rightarrow_a Q\sigma$

(LREPL) $\quad *(T).Q \mid T\sigma \rightarrow_a Q\sigma \mid *(T).Q$

(LMATCH) $\quad \texttt{match}\ T\sigma\ \texttt{with}\ T\ \texttt{in}\ P\ \texttt{else}\ Q \rightarrow_a P\sigma$

(LNOMATCH) $\quad \texttt{match}\ V\ \texttt{with}\ T\ \texttt{in}\ P\ \texttt{else}\ Q \rightarrow_a Q$ when $V \neq T\sigma$ for any σ

(LPARCTX)
$$\frac{P \rightarrow_a Q}{P \mid R \rightarrow_a Q \mid R}$$

(LNEWCTX)
$$\frac{P \rightarrow_a Q}{\nu n.P \rightarrow_a \nu n.Q}$$

(LSTRUCT)
$$\frac{P \equiv P' \quad P' \rightarrow_a Q' \quad Q' \equiv Q}{P \rightarrow_a Q}$$

where σ ranges over substitutions of closed terms for the variables bound in T. Let P be a local process for a. P is *stable* when it has no local reduction step, written $P \not\rightarrow_a$. We write $P \twoheadrightarrow_a Q$ when $P \rightarrow_a^* \equiv Q$ and $Q \not\rightarrow_a$.

Operational Semantics—System Transitions We define a labeled transition semantics for configurations, then for systems. We rely on an auxiliary relation $\equiv$ that represent structural equivalence for configurations, with the same rules as for processes plus

$\nu n.a[P] \equiv a[\nu n.P]$.

$$(\textsc{CfgOut})\frac{c \neq a}{a[a{:}c\langle V\rangle \mid Q] \xrightarrow{a:c\langle V\rangle} a[Q]} \qquad (\textsc{CfgIn})\frac{c{:}a\langle V\rangle \mid P \twoheadrightarrow_a Q \quad c \neq a}{a[P] \xrightarrow{(c:a\langle V\rangle)} a[Q]}$$

$$(\textsc{CfgBlock})\frac{C \xrightarrow{c:a\langle V\rangle} C' \quad i \text{ not in } C}{C \mid a[P] \xrightarrow{\nu i.c:a} C' \mid c{:}a\langle V\rangle/i \mid a[P]} \qquad (\textsc{CfgFwd})\frac{C \xrightarrow{(M)} C'}{C \mid M/i \xrightarrow{(i)} C'}$$

$$(\textsc{CfgPrinCtx})\frac{C \xrightarrow{\alpha} C' \quad \alpha \text{ not from/to } a}{C \mid a[P] \xrightarrow{\alpha} C' \mid a[P]} \qquad (\textsc{CfgNewCtx})\frac{C \xrightarrow{\alpha} C' \quad n \text{ not in } \alpha}{\nu n.C \xrightarrow{\alpha} \nu n.C'}$$

$$(\textsc{CfgMsgCtx})\frac{C \xrightarrow{\alpha} C' \quad i \text{ not in } \alpha}{C \mid M/i \xrightarrow{\alpha} C' \mid M/i} \qquad (\textsc{CfgOpen})\frac{C \xrightarrow{\alpha} C' \quad n \text{ free in } \alpha}{\nu n.C \xrightarrow{\nu n.\alpha} C'}$$

$$(\textsc{CfgStr})\frac{C \equiv D \quad D \xrightarrow{\alpha} D' \quad D' \equiv C'}{C \xrightarrow{\alpha} C'}$$

The first pair of rules represents 'intended' interactions with the environment; they enable local processes to send messages to other principals, and to receive their message. The transition label states the complete message contents.

The second pair of rules reflects the actions of an active attacker that intercepts, then selectively forwards, messages exchanged between compliant principals. In Rule (CFGBLOCK), the label reflects the interception of a message by the environment and includes only the information on the message that can be observed "on the wire": the environment gains partial knowledge $c : a$ of the message content. In addition, the complete message content is recorded within the configuration, using a fresh index i that can be used to forward the original message later on when the environment perform an input with label (i), using Rule (CFGFWD). The rule (CFGFWD) consumes the message record, so that intercepted messages are delivered at most once.

The local-reduction hypothesis in Rules (CFGIN) makes local computations "atomic", as they must complete immediately upon receiving a message and lead to some updated stable state Q. Intuitively, this enforces a transactional semantics for local steps, and prevent any observation of their transient internal state. (Otherwise, the environment may for instance observe the order of appearance of outgoing messages.) On the other hand, any outgoing messages are kept within Q; the environment can obtain all of them via rules (CFGOUT) and (CFGBLOCK) at any time, as those outputs commute with any subsequent transitions.

The rest of the rules for configurations are standard closure rules with regards to contexts and structural rearrangements: Rule (CFGOPEN) is the standard "scope extrusion" rule of the pi calculus for opening the scope of a restricted name included in a message sent to the environment. In contrast with intercepted messages, messages sent to a principal not defined in the configuration are transmitted unchanged to the environment, after applying the context rules. In Rule (CFGPRINCTX), α not from a excludes inputs from the environment that forge a message from a, whereas α not to a excludes outputs that may be transformed by Rule (CFGBLOCK).

Finally, we have a pair of top-level rules that deal with the attacker knowledge:

$$(\textsc{SysOut})\frac{C \xrightarrow{\alpha} C' \quad \alpha \text{ output}}{\Phi \vdash C \xrightarrow{\alpha} \Phi \cup \phi(\alpha)_{\mathcal{H}} \vdash C} \quad (\textsc{SysIn})\frac{C \xrightarrow{\alpha} C' \quad \alpha \text{ input} \quad \phi(\alpha)_{\mathcal{H}} \subseteq \mathcal{M}(\Phi)}{\Phi \vdash C \xrightarrow{\alpha} \Phi \vdash C}$$

where $\mathcal{H}$ is the set of principals defined in C and $\mathcal{M}(\Phi) = \{a\{V\}_\ell : a\{V\}_0 \in \Phi\}$ is the set of certificates the attacker might produce from Φ (cf. Appendix A).

By design, our semantics is compositional, as its rules are inductively defined on the structure of configurations. For instance, we obtain that interactions with a principal that is implicitly controlled by the environment are *at least* as expressive as those with any principal explicited within the system.

Anticipating our implementation of low-level interactions, we define auxiliary notions of transitions. We say that S is *stable* when all local processes are stable and S has no output transition. (Informally, S is waiting for any input from the environment.) We say that a series of transitions is *normal* when every input is followed by a series of outputs leading to a stable system.

4 High-Level Equivalences and Safety

Since our labeled transitions reflect our specific implementation constraints, we can apply standard definitions and proof techniques from concurrency theory to reason about systems. Our computational soundness results are useful (and non-trivial) inasmuch as transitions are simpler and more abstract than low-level adversaries. In addition to trace properties (where, for instance, some correspondence between transitions may be used to capture authentication properties), we consider equivalences between systems.

Intuitively, two systems are equivalent when their environment observes the same transitions. Looking at immediate observations, we say that two systems S_1 and S_2 *have the same labels* when, if $S_1 \xrightarrow{\alpha} S_1'$ for some S_1' (and the name extruded by α are not free in S_2), then $S_2 \xrightarrow{\alpha} S_2'$ for some S_2', and vice-versa. More generally, bisimilarity demands that this remain the case after any transitions:

Definition 3 (Bisimilarity). *The relation $\mathcal{R}$ on systems is a labeled simulation when, for all $S_1 \ \mathcal{R}\ S_2$, if $S_1 \xrightarrow{\alpha} S_1'$ (and the names extruded by α are not free in S_2) then $S_2 \xrightarrow{\alpha} S_2'$ and $S_1' \ \mathcal{R}\ S_2'$. Labeled bisimilarity, written $\approx$, is the largest symmetric labeled simulation.*

In particular, if $\Phi \vdash C \approx \Phi' \vdash C'$ then C and C' define the same principals, intercepted-message identifiers, and certificates ($\Phi = \Phi'$).

We also easily verify some congruence properties: our equivalence is preserved by addition of principals, deletion of intercepted messages, and deletion of certificates.

Lemma 1. *1. If $\Phi \vdash C_1 \approx \Phi \vdash C_2$, then $\Phi \cup \Phi_a \vdash C_1 \mid a[P] \approx \Phi \cup \Phi_a \vdash C_2 \mid a[P]$ for any certificates Φ_a issued by a such that the systems are well-formed.*
2. If $\Phi \vdash C_1[M_1/i] \approx \Phi \vdash C_2[M_2/i]$, then $\Phi \vdash C_1[\mathbf{0}] \approx \Phi \vdash C_2[\mathbf{0}]$.
3. If $\Phi \cup \{V\} \vdash C_1 \approx \Phi \cup \{V\} \vdash C_2$ and $V \notin \phi(\Phi)$, then $\Phi \vdash C_1 \approx \Phi \vdash C_2$.

Bounding processes As we quantify over all local processes, we must at least bound their computational power. Indeed, our language is expressive enough to code Turing machines and, for instance, one can easily write a local process that receives a high-level encoding of the security parameter η (e.g. as a series of η messages) then delays a message output by 2^{η} reduction steps, or even implements an 'oracle' that performs some brute-force attacks on top of (high-level implementations of) cryptographic algorithms.

Similarly, we must restrict non-determinism behaviors. Process calculi often feature non-determinism as a convenience when writing specifications, to express uncertainty as regards the environment. Sources on non determinism include local scheduling, hidden in the associative-commutative laws for parallel composition, and internal choices. Accordingly, abstract properties and equivalences typically only consider the existence of transitions—not their probability. Observable non-determinism is problematic in a computational cryptographic setting, as for instance a non-deterministic linear process may be used as an oracle to guess a key, one bit at a time.

These restrictions are serious, but they are also easily established when writing simple programs and protocols. (Still, it would be interesting to relax them, maybe using a probabilistic process calculus.) Hence, our language design prevents trivial sources of non-determinism (e.g. with pattern matching on values, and replicated inputs instead of full-fledge replication); further, most internal choices can be coded as external choices driven by the inputs of our abstract environment.

We arrive at the following definitions. We compute the size of values, processes, labels, systems, and transitions by structural induction, with for instance $|S \xrightarrow{\alpha} S'| = |S| + |\alpha| + |S'| + 1$. We let φ range over series of transition labels, and let $input(\varphi)$ be the series of input labels in φ.

Definition 4 (Safe Systems). *A system S is* polynomial *when there exists a polynom p such that, for any φ, if $S \xrightarrow{\varphi} S'$ then $|S \xrightarrow{\varphi} S'| \leq p(|input(\varphi)|)$.*

A system S is safe *when it is polynomial and, for any φ, if $S \xrightarrow{\varphi} S_1$ and $S \xrightarrow{\varphi} S_2$ then S_1 and S_2 have the same labels.*

Hence, starting from a safe process, a series of labels fully determine any further observations. Note that safety is preserved by all transitions, and also uniformly bounds (for example) the number of local processes, new names, and certificates. This excludes in particular any diverging local computation.

We can adapt usual bisimulation proof techniques to establish both equivalences and safety: instead of examining all series of labels φ, it suffices to examine all labels for the systems in the candidate relation.

Lemma 2 (Bisimulation Proof). *Let $\mathcal{R}$ be a symmetric labeled simulation such that, for every system S related by $\mathcal{R}$ and every label α, if $S \xrightarrow{\alpha} S_1$ and $S \xrightarrow{\alpha} S_2$, then $S_1 \mathcal{R} S_2$. Polynomial systems related by $\mathcal{R}$ are safe and bisimilar.*

We illustrate our definitions using basic examples of secrecy and authentication stated as equivalences between a protocol and its specification (adapted from [2]). Consider a principal a that sends a single message. In isolation, we have $a[a{:}b\langle V\rangle] \approx a[a{:}b\langle V'\rangle]$ if and only if $V = V'$, since the environment directly observes M on the label of the transition $a[a{:}b\langle M\rangle] \xrightarrow{a:b\langle M\rangle} a[\mathbf{0}]$.

Consider now a definition for principal b that receives the message and, assuming the message is a pair, runs P with the first element of the pair substituted for x. We have

$$a[a{:}b\langle V, W\rangle] \mid b[(a{:}\langle x, \_\rangle).P] \approx a[a{:}b\langle V, W'\rangle] \mid b[(a{:}\langle x, \_\rangle).P]$$

for any terms W and W'. This equivalence states the strong secrecy of W, as its value has no effect on the environment. The system now has the two transitions $\xrightarrow{\nu i.a:b}\xrightarrow{i}$ $a[\mathbf{0}] \mid b[P\{V/x\}]$ interleaved with inputs from any $e \notin \{a, b\}$. Further, the equivalence

$$a[a{:}b\langle V, W\rangle] \mid b[(a{:}\langle x, \_\rangle).P] \approx a[a{:}b\langle\rangle] \mid b[(a{:}\langle \_\rangle).P\{V/x\}]$$

captures both the authentication of V and the absence of information leaks on V, W in the communicated message, since the protocol (on the right) behaves just like another protocol that sends a dummy message instead of V, W.

5 A Concrete Implementation (Outline)

We systematically map high-level systems S to the machines of Section 2, mapping each principal $a[P_a]$ of S to a PPT machine M_a that executes P_a. Due to space constraints, we only give an outline of our implementation, defined in Appendix C. The implementation mechanisms are simple, but they need to be carefully specified and composed. (For instance, when a machine outputs several messages, possibly to the same principals, we must sort the messages after encryption so that their ordering on the wire leak no information on the computation that produced them.)

We use two concrete representations for terms: a wire format for (encrypted, signed) messages between principals, and an internal representation for local terms. Various bitstrings represent constructors, principal identities, names, and certificates. Marshaling and unmarshaling functions convert between internal and wire representations. Signatures are verified as part of unmarshaling. Signatures for self-issued certificates are generated on-demand, as part of marshaling, and cached, so that the same signature value is used for any certificate with identical content.

Local processes are represented in normal form for structural equivalence, using internal terms and multisets of local inputs, local outputs, and outgoing messages. We implement local reductions using an abstract machine parameterized by a scheduler that matches inputs and outputs. This scheduler is an arbitrary deterministic polynomial-time algorithm. When activating a process with a new name restriction ($\nu n.P$), we draw a bitstring s of length η uniformly at random and substitute it for n in P.

To keep track of the runtime state for our machines, we supplement high-level systems S with *shadow states* D that record sufficient information so that each machine be a function $\mathsf{M}_a(S, \mathsf{D})$. For instance, D records maps from principals to their keys, from names, certificates, and intercepted messages to bitstrings, and also the current content of the anti-replay cache for each machine. The shadow D also determines the information initially available to the attacker, coded as a bitstring $public(\mathsf{D})$.The structure of $public(\mathsf{D})$ sets the interface between attackers and low-level systems, called the *shape* of D. For instance, this shape fixes the number of free names of S, and $public(\mathsf{D})$ provides their binary representations.

Instead of explicitly coding a low-level initialization of our machines leading to $\mathsf{M}(S, \mathsf{D})$ and $\mathit{public}(\mathsf{D})$, we *define* it as the implementation of a high-level initialization protocol $S^\circ \xrightarrow{\varphi^\circ} S$ that let the principals exchanges names and certificates with the environment (see Appendix B). Applying a variant of Theorem 1, there is a PPT algorithm $\mathsf{B}_{\varphi^\circ}$ that controls initialization and produces $\mathit{public}(\mathsf{D})$, so that a run of $\mathsf{M}(S, \mathsf{D})$ with adversary B, written $\mathsf{B}[\mathsf{M}(S, \mathsf{D})]$, can be generally defined as a run of $(\mathsf{B}_{\varphi^\circ}; \mathsf{B})[\mathsf{M}(S^\circ, \mathsf{D}^\circ)]$ where $\mathsf{B}_{\varphi^\circ}; \mathsf{B}$ first runs $\mathsf{B}_{\varphi^\circ}$ then starts B with input $\mathit{public}(\mathsf{D})$.

6 Soundness and Completeness Results

In this section we show that properties that hold with the high-level semantics can be carried over to the low-level implementation, and the other way around. Due to space constraints, most auxiliary results and all proofs appear in a companion paper [8].

Our first theorem expresses the soundness of the high-level operational semantics: every high-level trace can be characterized by a low-level attacker. Said otherwise, our high-level semantics does not give too much power to the environment.

Theorem 1. *For any shape of* D *and normal transition labels* φ*, there is a PPT algorithm* B_φ *such that, for any safe stable* S *with valid shadow* D*:*

- $\Pr[0 \longleftarrow \mathsf{B}_\varphi[\mathsf{M}(S, \mathsf{D})]] = 1 - \mathrm{neg}\,(\eta)$*, and* $S \overset{\varphi}{\not\rightarrow}$*; or*
- $\Pr[1 \longleftarrow \mathsf{B}_\varphi[\mathsf{M}(S, \mathsf{D})]] = 1 - \mathrm{neg}\,(\eta)$*, and there exists* S' *such that* $S \xrightarrow{\varphi} S'$*.*

Since we can characterize any trace using an adversary, we also obtain completeness for trace equivalence: low-level equivalence implies high-level trace equivalence.

Theorem 2. *Let* S *and* S' *be safe stable systems with valid shadow* D*.*
If $\mathsf{M}(S, \mathsf{D}) \approx_\eta \mathsf{M}(S', \mathsf{D})$*, then* S *and* S' *have the same normal transition labels.*

Our next theorem expresses the completeness of our high-level transitions: every low-level attack can be described in terms of high-level transitions. More precisely, the probability that an interaction with a PPT adversary yields a final machine state unexplained by any high-level transitions is negligible.

Theorem 3. *Let* S *be a stable system with valid shadow* D*. For any PPT algorithm* B*, the probability that* $\mathsf{B}[\mathsf{M}(S, \mathsf{D})]$ *completes leaving the system in a state* M' *with* $\mathsf{M}' \neq \mathsf{M}(S', \mathsf{D}')$ *for any normal trace* $S \xrightarrow{\varphi} S'$ *with state* D' *is negligible.*

Finally, our main result expresses soundness for high-level equivalence: to show that two stable systems are low-level equivalent, it suffices to show that they are safe and bisimilar.

Theorem 4. *Let* S *and* S' *be safe stable systems with valid shadow* D*.*
If $S \approx S'$*, then* $\mathsf{M}(S, \mathsf{D}) \approx_\eta \mathsf{M}(S', \mathsf{D})$*.*

7 Conclusions and Future Work

We designed a simple, abstract language for distributed communications with two forms of authentication (but no explicit cryptography). Our language provides uniform protection for all messages; it is expressive enough to program a large class of protocols; it also enables simple reasoning about security properties in the presence of active attackers, using labeled traces and equivalences. We implemented this calculus as a collection of concrete PPT machines embedding standard cryptographic algorithms, and established that low-level PPT adversaries that control their scheduling and the network have essentially the same power as (much simpler) high-level environments. To the best of our knowledge, these are the first cryptographic soundness and completeness results for a distributed process calculus.

We also identified and discussed difficulties that stem from the discrepancy between the two models. Our proofs involve a novel combination of techniques from process calculi and cryptography, but they are less modular than we expected. It would be interesting (and hard) to extend the expressiveness of our calculus, for instance with secrecy and probabilistic choices.

Acknowledgments This paper benefited from discussions with Martín Abadi, Karthik Bhargavan, and Tuomas Aura.

References

1. M. Abadi and C. Fournet. Private authentication. *Theoretical Computer Science*, 322(3):427–476, 2004. Special issue on Foundations of Wide Area Network Computing.
2. M. Abadi, C. Fournet, and G. Gonthier. Authentication primitives and their compilation. In *POPL 2000*, pages 302–315. ACM, 2000.
3. M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83, 2002.
4. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, 1999.
5. M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In *TACS 2001*, LNCS 2215, pages 82–94. Springer, 2001.
6. M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.
7. P. Adão, G. Bana, and A. Scedrov. Computational and information-theoretic soundness and completeness of formal encryption. In *CSFW-18*, pages 170–184. IEEE, 2005.
8. P. Adão and C. Fournet. Cryptographically sound implementations for communicating processes. Available at `http://research.microsoft/com/~fournet/crypto-sound-processes-draft.pdf`, 2006.
9. M. Backes and M. Dürmuth. A cryptographically sound Dolev-Yao style security proof of an electronic payment system. In *CSFW-18*, pages 78–93. IEEE, 2005.
10. M. Backes and B. Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *CSFW-17*, pages 204–218. IEEE, 2004.
11. M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *CCS 2003*, pages 220–230. ACM Press, 2003.
12. M. Backes, B. Pfitzmann, and M. Waidner. Symmetric authentication within a simulatable cryptographic library. *International Journal of Information Security*, 4(3):135–154, 2005.

13. M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *CRYPTO'98*, LNCS 1462, pages 26–45. 1998.
14. B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *LICS 2005*, pages 331–340. IEEE, 2005.
15. R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *Eucrocrypt 2001*, LNCS 2045. Springer, 2001.
16. V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. In *ESOP 2005*, LNCS 3444, pages 157–171. Springer, 2005.
17. P. Laud. Secrecy types for a simulatable cryptographic library. In *CCS 2005*, pages 26–35. ACM Press, 2005.
18. P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *CCS 1998*, pages 112–121, 1998.
19. D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *TCC 2004*, LNCS 2951, pages 133–151. Springer, 2004.
20. J. C. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague. A probabilistic polynomial-time calculus for the analysis of cryptographic protocols. *Theoretical Computer Science*.
21. C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *CRYPTO'91*, LNCS 576, pages 433–444. Springer, 1991.

A Cryptographic Definitions

Cryptographic Primitives An encryption scheme is a triple of algorithms $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ with key generation $\mathcal{K}$, encryption $\mathcal{E}$ and decryption $\mathcal{D}$. Let $\mathbf{plaintexts}$, $\mathbf{ciphertexts}$, $\mathbf{publickey}$ and $\mathbf{secretkey}$ be nonempty subsets of $\mathbf{strings}$. The set $\mathbf{coins}$ is some probability field that stands for coin-tossing, *i.e.*, randomness.

Definition 5 (Encryption Scheme). *An* asymmetric encryption scheme *is a triple* $\mathit{\Pi} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ *where:*

– $\mathcal{K} : \mathbb{N} \times \mathbf{coins} \to \mathbf{publickey} \times \mathbf{secretkey}$ *is a key-generation algorithm with security parameter* η*,*
– $\mathcal{E} : \mathbf{publickey} \times \mathbf{plaintexts} \times \mathbf{coins} \to \mathbf{ciphertexts}$ *is an encryption function,*
– $\mathcal{D} : \mathbf{secretkey} \times \mathbf{strings} \to \mathbf{plaintexts}$ *is such that for all* $(e, d) \in \mathbf{publickey} \times \mathbf{secretkey}$ *and* $\omega \in \mathbf{coins}$*,* $\mathcal{D}(d, \mathcal{E}(e, m, \omega)) = m$ *for all* $m \in \mathbf{plaintexts}$.

All these algorithms must be computable in polynomial-time in the size of the input. We insist that $|\mathcal{E}(e, m, w)| = |\mathcal{E}(e, m, w')|$ for all $e \in \mathbf{publickey}, m \in \mathbf{plaintexts}$ and $w, w' \in \mathbf{coins}$, where $|x|$ stands for the binary length of x.

There are several different notions of security for an encryption scheme. The one that we adopt here, introduced by [21], has been shown to be strictly stronger than almost all other definitions, including semantic security [13].

Definition 6 (IND-CCA2—Adaptive Chosen Ciphertext Security). *A computational public-key encryption scheme* $\mathit{\Pi} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ *provides* indistinguishability under the adaptive chosen-ciphertext attack *if for all PPT adversaries* A *and for all sufficiently large security parameters* η*:*

$$\begin{array}{ll}
\Pr[\ (e,d) \longleftarrow \mathcal{K}(1^\eta); & \\
\quad m_0, m_1 \longleftarrow \mathsf{A}^{\mathcal{D}_1(\cdot)}(1^\eta, e); & \\
\quad b \longleftarrow \{0,1\}\,; & \\
\quad c \longleftarrow \mathcal{E}(e, m_b); & \\
\quad g \longleftarrow \mathsf{A}^{\mathcal{D}_2(\cdot)}(1^\eta, e, c) : & \\
\quad b = g &] \leq \frac{1}{2} + \mathrm{neg}\,(\eta)
\end{array}$$

The oracle $\mathcal{D}_1(x)$ returns $\mathcal{D}(d,x)$, and $\mathcal{D}_2(x)$ returns $\mathcal{D}(d,x)$ if $x \neq c$ and returns $\perp$ otherwise. The adversary is assumed to keep state between the two invocations. It is required that m_0 and m_1 be of the same length.

That is, an adversary should not be able to learn from a ciphertext whether it is the encryption of the plaintext m_0 or the plaintext m_1, even if the adversary knows the public key used to encrypt, the adversary can choose the messages m_0 and m_1 itself, so long as the messages have the same length, and the adversary can request and receive the decryption of any *other* ciphertext.

Definition 7 (Signature Scheme). *A* signature scheme *is a triple $\Sigma = (\mathcal{G}, \mathcal{S}, \mathcal{V})$ where:*

- $\mathcal{G} : \mathbb{N} \times \mathbf{coins} \to \mathbf{sigkey} \times \mathbf{verifykey}$ *is a key-generation algorithm with security parameter η,*
- $\mathcal{S} : \mathbf{sigkey} \times \mathbf{plaintexts} \times \mathbf{coins} \to \mathbf{signedtexts}$ *is the signing algorithm, and*
- $\mathcal{V} : \mathbf{verifykey} \times \mathbf{strings} \times \mathbf{signedtexts} \to \{0,1\}$ *is such that for every pair $(s,v) \longleftarrow \mathcal{G}(1^\eta)$ and $\omega \in \mathbf{coins}$, $\mathcal{V}(v, m, \mathcal{S}(s, m, \omega)) = 1$ for all $m \in \mathbf{plaintexts}$.*

All these algorithms must be computable in polynomial-time in the size of the input. We call (s,v) a pair of signing/verification keys, and the string $\mathcal{S}(s,m,\omega)$ a signature of the plaintext m with the key s.

Similarly to encryption, there are several different notions of security for signature schemes. We adopt the notion of *unforgeable signature under chosen message attack.*

Definition 8 (Adaptive Chosen Message Security). *A signature scheme $\Sigma = (\mathcal{G}, \mathcal{S}, \mathcal{V})$ is secure against* forgery under adaptive chosen-message attack *if for all PPT adversaries* A *and for all sufficiently large security parameters η:*

$$\begin{array}{ll}
\Pr[\ (s,v) \longleftarrow \mathcal{G}(1^\eta); & \\
\quad (m, \mathit{sig}) \longleftarrow \mathsf{A}^{\mathcal{S}_1(\cdot)}(1^\eta, v) : & \\
\quad \mathcal{V}(v, m, \mathit{sig}) = 1 & \mid \quad m \notin \mathit{Queries}\ ] \leq \mathrm{neg}\,(\eta)
\end{array}$$

The oracle $\mathcal{S}_1(x)$ returns $\mathcal{S}(s,x)$ and adds x to the set $\mathit{Queries}$.

This game intuitively says that, after requesting as many signatures as he wants from the signing oracle $\mathcal{S}_1$, an adversary cannot produce a pair (m, sig) such that sig is the signature of the message m. Of course this game is only fair if the produced pair is not one of the pairs obtained by querying the oracle. Note that the adversary can also access the verification algorithm since he knows the verification key v.

Definition 8 does not state whether the adversary (or even the signer) is able to generate other values sig' such that $\mathcal{V}(v, m, sig') = 1$ given v, m, and sig (and even s for the signer). In practice, this ability may come from the underlying cryptographic algorithms, or simply from the lack of normalization for signature values. Conservatively, our high-level semantics assumes this is always possible, as specified in Rule (SYSIN), whereas our low-level implementation does not rely on this ability. Still, for establishing some of our results (e.g. the existence of some adversary in completeness proofs), we need to be more specific. To this end, we then use a signature scheme $\Sigma' = (\mathcal{G}, \mathcal{S}, \mathcal{V}, \mathcal{M})$ such that $(\mathcal{G}, \mathcal{S}, \mathcal{V})$ meets Definition 8 and the fourth algorithm $\mathcal{M}$ is such that, if $\mathcal{V}(v, m, sig) = 1$, then the values $sig_n = \mathcal{M}(v, m, sig, 1^n)$ for $n < \eta$ are pairwise distinct and such that $\mathcal{V}(v, m, sig_n) = 1$.

It is straightforward (if not very useful) to build such a signature scheme from any given Σ, as follows: for signing, we concatenate η zeros to the signature value sig; for signature verification, we ignore the last η bits of the signature value; for producing other signature value, we increment the last bits of the signature value.

As for authentication, we use a symmetric signing scheme. A symmetric authentication scheme $\Lambda = (\mathcal{G}_\Lambda, \mathcal{A}, \mathcal{C})$ is defined in the same way as Definition 7 except that $\mathcal{G}_\Lambda$ generates a single key, used both for signing and encryption, that verifies the equation $\mathcal{C}(k, m, \mathcal{A}(k, m, \omega)) = 1$ for all $m \in \mathbf{plaintexts}$. The notion of security is the same as in Definition 8, except that the adversary is not given the verification key v (which is also the signing key).

B Applications

We present three coding examples within our language, dealing with anonymous forwarders, electronic contracts, and system initialization. In addition, we coded a translation from asynchronous pi calculus processes into local processes, using terms $\texttt{chan}(n)$ to represent channels. (The scope of name n represents the scope of the channel, and channel-based communications is implemented by pattern matching on channel terms.) We also coded distributed communications for the authenticated join-calculus channels of [2], using certificates $a\{\texttt{chan}(n)\}$ to represent output capabilities of channels.

Anonymous Forwarders We consider a (simplified, synchronous) anonymizing mix hosted by principal c. This principal receives a single message V from every participant $a \in A$, then forwards all those messages to some sender-designated address b. The forwarded message does not echo the sender identity—however this identity may be included as a certificate in the message V. We study a single round, and assume that, for this round, the participants trust c but do not trust one another. We use the following local processes (indexed by principal) and systems:

$$P_c = \textstyle\prod_{a \in A} (a{:}c\langle ?b, ?V\rangle).(\texttt{tick} \mid (\texttt{go}).c{:}b\langle \texttt{forward}(V)\rangle)$$

$$Q_c = (\texttt{tick}).^{\text{for each } a \in A} \textstyle\prod_{a \in A} \texttt{go}$$

$$P_a^\sigma = a{:}c\langle b_{a\sigma}, V_{a\sigma}\rangle \mid P'_a$$

$$S^\sigma = c[P_c \mid Q_c] \mid \textstyle\prod_{a \in A'} a[P_a^\sigma]$$

The process P_c receives a single message from every $a \in A$, then it emits a local `tick` message and wait for a local go message. The process Q_c runs in parallel with P_c and provides synchronization; it waits for a `tick` message for every participant, then sends go messages to trigger the forwarding of all messages.

Let $A' \subseteq A$ be a subset of participants that comply with the protocol. We set $\mathcal{H} = A' \uplus \{c\}$. Anonymity for this round may be stated as follows: no coalition of principals in $A \setminus A'$ should be able to distinguish between two systems that differ only by a permutation of the messages sent by the participants in A'. Formally, for any such permutations σ and σ', we verify the equivalence $S^{\sigma} \approx S^{\sigma'}$. Hence, even if the environment knows all the V messages, the attacker gains no information on σ. (Conversely, the equivalence fails, due to traffic analysis, if we use instead a naive mix that does not wait for all messages before forwarding, or that accepts messages from any sender.)

Electronic Payment Protocol As a benchmark, we consider the electronic payment protocol presented by Backes and Dürmuth in [9]. We refer to their work for a detailed presentation of the protocol and its properties. The authors provide a computationally sound implementation of the protocol on top of an idealized cryptographic library [11]. We obtain essentially the same security properties, but our coding of the protocol is more abstract and shorter than theirs (by a factor of 10) and yields simpler proofs, essentially because it does not have to deal with the details of signatures, marshalling, and local state—coded once and for all as part of our language implementation.

The protocol has four roles, a client c, a vendor v, an acquirer ac, and a trusted third party $\mathtt{ttp}$. For simplicity, we assume that ac and $\mathtt{ttp}$ are unique and well-known. In addition, we use a distinct, abstract principal U that sends or receives all events considered in trace properties. Initially, the client, vendor, and acquirer tentatively agree on their respective identities and a transaction descriptor t that describes the goods and their price. The protocol essentially relies on the forwarding of certificates. We let $x:\{y, V\}$ abbreviates a message with a certified content $x{:}y\langle x\{y, V\}\rangle$, and use $\mathit{as}\ \mathit{sig}$ to bind the corresponding certificate $x\{y, V\}$.

$$\begin{aligned}
\mathit{Client}_c = {} & *(U{:}c\langle \mathtt{pay}(?t, ?v)\rangle).\\
& (v:\{c, \mathtt{invoice}(t)\}\ \mathit{as}\ \mathit{sig}_v).\\
& \quad (c:\{v, \mathtt{payment}(t)\} \mid\\
& \quad (v{:}c\langle \mathtt{confirm}(\mathsf{ac}\{v, \mathtt{response}(t,c)\}\ \mathit{as}\ \mathit{sig}_{\mathsf{ac}})\rangle).\\
& \qquad (c{:}U\langle \mathtt{paid}(t,v)\rangle \mid\\
& \qquad (U{:}c\langle \mathtt{dispute}(t)\rangle).c{:}\mathtt{ttp}\langle \mathtt{client\_dispute}(\mathit{sig}_v, \mathit{sig}_{\mathsf{ac}})\rangle)))\\
\mathit{Vendor}_v = {} & *(U{:}v\langle \mathtt{receive}(?t, ?c)\rangle).\\
& (v:\{c, \mathtt{invoice}(t)\}\ \mathit{as}\ \mathit{sig}_v \mid\\
& (c:\{v, \mathtt{payment}(t)\}\ \mathit{as}\ \mathit{sig}_c).\\
& \quad (v{:}\mathsf{ac}\langle \mathtt{request}(\mathit{sig}_v, \mathit{sig}_c)\rangle \mid\\
& \quad (\mathsf{ac}:\{v, \mathtt{response}(t,c)\}\ \mathit{as}\ \mathit{sig}_{\mathsf{ac}}).\\
& \qquad (v{:}c\langle \mathtt{confirm}(\mathit{sig}_{\mathsf{ac}})\rangle \mid v{:}U\langle \mathtt{received}(t,c)\rangle \mid\\
& \qquad (U{:}v\langle \mathtt{dispute}(t)\rangle).v{:}\mathtt{ttp}\langle \mathtt{vendor\_dispute}(\mathit{sig}_c, \mathit{sig}_{\mathsf{ac}})\rangle))))
\end{aligned}$$

$$
\begin{aligned}
\textit{Acquirer}_{\mathsf{ac}} = {} & *(U{:}\mathsf{ac}\langle \mathtt{allow}(?t, ?c, ?v)\rangle). \\
& \quad (v{:}\mathsf{ac}\langle \mathtt{request}(v\{c, \mathtt{invoice}(t)\}\ \textit{as sig}_v, \\
& \qquad\qquad c\{v, \mathtt{payment}(t)\}\ \textit{as sig}_c)\rangle). \\
& \qquad (\mathsf{ac}:\{v, \mathtt{response}(t,c)\} \mid \mathsf{ac}{:}U\langle \mathtt{transfer}(t,c,v)\rangle \mid \\
& \qquad (U{:}\mathsf{ac}\langle \mathtt{dispute}(t)\rangle).\mathsf{ac}{:}\mathsf{ttp}\langle \mathtt{acquirer\_dispute}(\textit{sig}_c, \textit{sig}_v)\rangle) \\
P_{\mathsf{ttp}} = {} & *(?c{:}\mathsf{ttp}\langle \mathtt{client\_dispute}(?d)\rangle). \\
& \quad \mathtt{match}\ d\ \mathtt{with}\ ?v\{c, \mathtt{invoice}(?t)\}, \mathsf{ac}\{v, \mathtt{response}(t,c)\}\ \mathtt{in} \\
& \qquad \mathsf{ttp}{:}U\langle \mathtt{accept\_client}(t,c,v)\rangle\ \mathtt{else} \\
& \qquad \mathsf{ttp}{:}U\langle \mathtt{reject\_client}(t,c,v)\rangle
\end{aligned}
$$

A system S consists of any number of principals (potentially) running the three roles, plus a unique principal ttp running P_{ttp}. The system should not define U, which represents an arbitrary, abstract environment that controls the actions of the other principals. For a given normal trace φ, we say that the payment t, c, v, ac is *complete* when φ includes the following input labels: (i) if $c \in \mathcal{H}$, then $U{:}c\langle \mathtt{pay}(t,v)\rangle$; (ii) if $v \in \mathcal{H}$, then $U{:}v\langle \mathtt{receive}(t,c)\rangle$; and (iii) if $\mathsf{ac} \in \mathcal{H}$, then $U{:}\mathsf{ac}\langle \mathtt{allow}(t,c,v)\rangle$. We can now state the following properties:

– *Weak atomicity* is a trace property expressed as follows: if φ includes any output of the form $c{:}U\langle \mathtt{paid}(t,v)\rangle$, $v{:}U\langle \mathtt{received}(t,c)\rangle$, or $\mathsf{ac}{:}U\langle \mathtt{transfer}(t,c,v)\rangle$, then the payment t, c, v, ac is complete.
– *Correct client dispute* states that an honest client—who starts a dispute for transaction t only after completing the protocol for t, as coded in the last line of *Client*$_c$—always wins his dispute: for any trace φ, if $c \in \mathcal{H}$ and $c{:}U\langle \mathtt{paid}(t,v)\rangle$ is in φ, then $\mathsf{ttp}{:}U\langle \mathtt{reject}(t,c,v)\rangle$ is not in φ. (This property is rather weak, as the vendor and acquirer complete the protocol before the client.) We omit the corresponding dispute code and properties for *vendor* and *acquirer*.
– *No framing* states that the ttp does not wrongly involve parties that have not initiated the protocol with matching parameters. It is a variant of weak atomicity: outputs of the form $\mathsf{ttp}{:}U\langle \mathtt{accept}(t,c,v)\rangle$ only occur for complete payments.

These properties are directly established by induction on the high-level transitions of S.

Initialization This technical example shows that it suffices to develop concrete implementations for *initial systems* that do not share any names, certificates, or intercepted messages with the environment. Up to structural equivalence, every system is of the form $S = \Phi \vdash \nu\widetilde{n}.(\prod_{a\in\mathcal{H}} a[P_a] \mid \prod_{i\in I} M/i)$. The sharing of names and certificates between principals and the environment can be quite complex, and is best handled using an ad hoc (but high-level) "bootstrapping" protocol, outlined below:

1. Free names of S and restricted non-local names $\widetilde{n}$ are partitioned between honest principals; let $(n_{a,1}, \ldots, n_{a,k_a})_{a\in\mathcal{H}}$ be those names.
2. Free names and non-self-issued certificates that occur in the local processes P_a are exchanged using a series of initialization messages of the form $M_{ab,r} = a{:}b$ $\langle \mathtt{init}_{ab,r}(n_{a,1}, \ldots, n_{a,k_a}, a\{V_{ab,1}\}, \ldots, a\{V_{ab,m_r}\})\rangle$, carrying names and certificates issued by a that occur in P_b. Similarly, initialization messages sent to a fixed

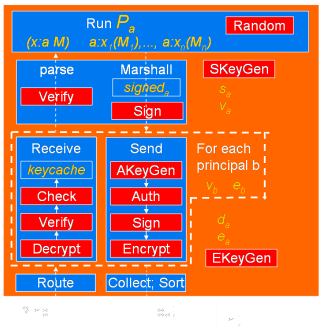

Fig. 1. Local machine for principal a connected to the adversary machine

principal $e \notin \mathcal{H}$ export the free names of S and the certificates of Φ, whereas initialization messages from e import certificates issued by principals not in $\mathcal{H}$. Each principal $a \in \mathcal{H}$ sequentially receives and checks all initialization messages addressed to him, using input patterns of the form $(T_{ba,r})$ where $T_{ba,r}$ is $M_{ba,r}$ with binding variables $?n_1, \ldots, ?n_k$ instead of the names and aliases $b\{V_{ba,r}\}\ as\ ?x$ for checking and binding certificates.

3. Finally, each principal a sends a message M for every intercepted message M/i from a defined in S, then starts P_a.

For instance, in case $\mathcal{H} = \{a, b\}$ with neither nested certificates nor intercepted messages, the local initialization process for a is $P^\circ_a = \nu n_1, \ldots, n_{k_a}.(M_{ab,1} \mid M_{ae,1} \mid (T_{ba,1}).(T_{ea,1}).P_a)$. In the general case, several rounds of initialization messages may be needed to exchange certificates whose contents include names and certificates, and to emit messages with the same shape one at a time.

C A Concrete Implementation

We are now ready to define the machines outlined in Section 2, relying on translations from high-level terms and processes to keep track of their runtime state. The rules of Section 3 declare that all communications be authentic and confidential. In order to meet these requirements, our implementation relies on concrete bitstrings and cryptographic protocols. The runtime state of machine M_a consists of the following data:

- id_a, d_a, and s_a are bitstrings that represent the low-level identifier for principal a and its private keys for decryption and signing.
- $peers = \{(id_x, e_x, v_x) \mid x \in \mathsf{Prin}\}$ binds, for every principal, a low-level identifier to public keys for encryption and signature verification.
- p_a is a low-level representation of a local process running at a (defined below).
- $keycache_a$ is a set of authentication keys for all received messages.
- $signed_a$ is a partial function from certificates issued by a to signature values.

The main machine components are depicted in Figure 1. Before detailing their definitions, we describe a complete run of the machine. Recall that M_a is connected to the environment by two wires, $?\mathbf{input}_a$ and $!\mathbf{output}_a$. The wire format for messages is the concatenated bitstring $id_x\, id_y\, msg$ where x and y are the (apparent) sender and receivers and msg is some encrypted, authenticated, marshaled message. When it receives such a message (with $id_y = id_a$), M_a uses id_x to dispatch msg to the `receive` protocol (Definition 13) for remote principal x— there is an instance of the `receive` protocol for each peer principal x. The protocol verifies the freshness, integrity, and authenticity of the message, updates $keycache_a$, then returns a decrypted bitstring s. If a verification step fails, the message is discarded.

At this stage, msg is a genuine message from x to a, but its content is not necessarily well-formed. For instance x may have included a certificate apparently issued by b but with an invalid signature. Content validation occurs as s is unmarshaled (Definition 11) from its wire format into some internal (trusted) representation $parse_a(s)$ of a high-level term V. In particular, this representation embeds a valid signature for every certificate of V. After successful reception and unmarshaling, a representation m of the incoming message $x{:}a\langle V\rangle$ may react with an input within p_a and trigger local computations. To this end, a local interpreter (Definition 9) derived from an abstract machine for the local reductions runs on $p_a \mid m$. If the interpreter terminates, it yields a new stable internal process in normal form p'_a plus a set of outgoing messages X to be sent to the network.

Each message $a{:}x_i\langle V_i\rangle$ represented in X is then marshaled (Definition 10) and passed to the instance of the `send` protocol (Definition 12) associated with the intended recipient x_i. The resulting bitstrings, all in wire format, of the form $id_a\, id_{x_i}\, msg_i$, are eventually sorted (by receiving principal, then encrypted value msg_i)—to ensure that their ordering leaks no information on their payload or their internal production process—and written on $!\mathbf{output}_a$. A final `done` bitstring is issued and the machine terminates. (Hence, for instance, if p does not react with m, the machine simply writes done on $!\mathbf{output}_a$ and terminates.)

Next, we describe in turn each of the components of the local machine.

Low-level Processes The internal representation of terms uses the same grammar as in the high-level language except for atomic subterms: principals x are boxed, fixed-sized bitstrings $\mathtt{prin}(id_x)$; free names are boxed, bitstrings $\mathtt{name}(s)$ (of size η); and certificate labels are linear-sized bitstrings s such that either s is a valid signature for the certificate or $s = \mathtt{0}$ and the certificate is self-issued. Bound variables and restricted names may still occur in terms under input guards.

Definition 9 (Internal Reductions). *The local reduction algorithm refines the abstract machine of Section 3 as follows:*

1. *it represents the multisets X, M, and G using internal terms;*
2. *it uses a deterministic, polynomial-time, complete scheduler;*
3. *instead of lifting new name restriction $\nu n.Q$, it generates a bitstring s uniformly at random and substitutes $\mathtt{name}(s)$ for all bound instances of the n in Q.*

Marshaling and Unmarshaling We use a fixed, injective function from all constructors plus `name` and `prin` to bitstrings of a given fixed size; we still write `f`, `name`, `prin` for the corresponding bitstrings. We write $s\,s'$ for the bitstring obtained by concatenating s and s'.

Definition 10 (Marshaling). *The function $[\![\cdot]\!]_a$ maps principal a's internal representations of closed terms to bitstrings, as follows:*

$$\begin{array}{rll}
[\![\mathtt{name}(s)]\!]_a &= \mathtt{name}\,s & \\
[\![\mathtt{prin}(s)]\!]_a &= \mathtt{prin}\,s & \\
[\![\mathtt{f}(v_1,\ldots,v_n)]\!]_a &= \mathtt{f}\;[\![v_1]\!]_a\;\ldots\;[\![v_n]\!]_a & \textit{when } \mathtt{f} \notin \{\mathtt{name},\mathtt{prin},\mathtt{cert}\} \\
[\![v_1\{v_2\}_s]\!]_a &= \mathtt{cert}\;[\![v_1]\!]_a\;[\![v_2]\!]_a\;s & \textit{when } s \neq \mathtt{0} \\
[\![v_1\{v_2\}_{\mathtt{0}}]\!]_a &= \mathtt{cert}\;[\![v_1]\!]_a\;[\![v_2]\!]_a\;\mathit{signed}_a(v_1\{v_2\}_{\mathtt{0}}) & \textit{when } v_1 = \mathtt{prin}(id_a) \\
& \multicolumn{2}{l}{\textit{adding } \mathit{signed}_a(v_1\{v_2\}_{\mathtt{0}}) = \mathcal{S}(s_a,[\![v_2]\!]_a) \textit{ when undefined}}
\end{array}$$

We assume that after marshalling all our messages are padded to a fixed length (that is polynomial in the security parameter η). We could have assumed that this was not the case and if so, we needed to consider this difference of length in our high-level semantics. We could have done it using sorts and sizes for input and output messages.

Definition 11 (Unmarshaling). *The partial function $\mathit{parse}_a(\cdot)$ maps bitstrings to a's internal representations of closed terms, as follows, and fails in all other cases.*

$$\begin{array}{rll}
\mathit{parse}_a(\mathtt{name}\,s) &= \mathtt{name}(s) & \textit{when } |s| = \ell_{\mathtt{name}} \\
\mathit{parse}_a(\mathtt{prin}\,s) &= \mathtt{prin}(s) & \textit{when } |s| = \ell_{\mathtt{prin}} \textit{ and } (s,e_s,v_s) \in \mathit{peers} \\
\mathit{parse}_a(\mathtt{f}\;s_1\ldots s_n) &= \mathtt{f}(v_1,\ldots,v_n) & \textit{when } \mathtt{f} \notin \{\mathtt{name},\mathtt{prin},\mathtt{cert}\} \textit{ has arity } n \\
& & \mathit{parse}_a(s_i) = v_i \textit{ for } i = 1..n \\
\mathit{parse}_a(\mathtt{cert}\,s_1\,s_2\,s_3) &= v_1\{v_2\}_s & \textit{when, for some } (id_x,e_x,v_x) \in \mathit{peers}, \\
& & \mathit{parse}_a(s_1) = \mathtt{prin}(id_x), \mathit{parse}_a(s_2) = v_2 \\
& & \mathcal{V}(v_x,s_2,s_3) = 1 \\
& & s = \textit{if } \mathit{signed}_a(v_1\{v_2\}_{\mathtt{0}}) = s_3 \textit{ then } \mathtt{0} \textit{ else } s_3
\end{array}$$

Unmarshaling includes signature verification for any received certificate, and is otherwise standard; it is specified here as a partial function from strings to internal representations, and can easily be implemented as a parser. Our treatment of self-issued certificates with label `0` reflects our choice of internal representations: `0` stands for the (unique) signature generated by the local machine for this certificate content, on demand, the first time this certificate is marshaled. (In addition, the adversary may be able to derive a variant of this certificate with a different signature, unmarshaled with a non-zero label; such certificates are then treated using the default case for marshaling.)

Although we give a concrete definition of $[\![\cdot]\!]_a$, $\mathit{parse}_a(\cdot)$, and message formats, our results only depend on their generic properties. We only require that, for a given local machine, every string be unmarshaled to at most one internal term, whose marshaling yields back the original string, that is, $\mathit{parse}_a([\![\ulcorner V \urcorner^a]\!]_a) = \ulcorner V \urcorner^a$. ($\ulcorner V \urcorner^a$ denotes the internal representation for V.) For simplicity, we have that the length of the string be a function of the structure of the internal term and of the security parameter.

Sending and Receiving Protocols The sending protocol takes a bitstring s (containing a marshaled message from a to x), protects it, and returns it in wire format. Conversely, the receiving protocol takes a message in wire format presumably from x, verifies it, and returns its payload. These protocols are intended as a simple example; other choices are possible. We may for instance consider long term shared keys between principals, in order to reduce the overhead of public-key cryptography.

Definition 12 (Sending to x). *Given a bitstring s, the protocol*

1. *generates a fresh authentication key $k \longleftarrow \mathcal{G}_A(1^\eta)$;*
2. *computes $msg = \mathcal{E}(e_x, s\,id_a\,k\,\mathcal{S}(s_a, k\,id_x)\,\mathcal{A}(k, s))$; and*
3. *returns the bitstring $id_a\,id_x\,msg$.*

Definition 13 (Receiving from x). *Given a bitstring $id_x\,id_a\,msg$, the protocol*

1. *computes $s\,id_x\,k\,s_{sig}\,s_{auth} = \mathcal{D}(d_a, msg)$;*
2. *checks that there is an entry $(id_x, e_x, v_x) \in peers$ with $\mathcal{V}(v_x, k\,id_a, s_{sig}) = 1$;*
3. *checks that $\mathcal{C}(k, s, s_{auth}) = 1$;*
4. *checks that k is not in $keycache$, and adds it to $keycache$;*
5. *returns s.*

The entire message is discarded if any step of the protocol fails.

Mapping High-Level Systems to Low-Level Machines In order to systematically relate the runtime state of low-level machines to the abstract state of high-level systems, we define an associated *shadow state*. This structure provides a consistent interpretation of terms across machines. In combination, a system and its shadow state determine their implementation, obtained as a compositional translation of terms, local processes, and configurations. (This state is shadow as it need not be maintained at runtime in the low-level implementation; it is used solely as an abstraction to reason about the correctness of our implementations.) We further partition this state into public parts, intended to be part of the attacker's knowledge, and private parts.

Definition 14 (Shadow State). *Let $S = \Phi \vdash \nu\widetilde{n}.C$ be a system such that the configuration $C = \prod_{a\in\mathcal{H}} a[P_a] \mid \prod_{i\in I} M/i$ is in normal form. A* shadow state *for S, written D, consists of the following data structure:*

- *$prin \in \mathsf{Prin} \rightarrow (\{0,1\}^\eta)^5$ is a function from $a \in \mathsf{Prin}$ to bitstrings id_a, e_a, v_a, d_a, s_a such that $a \rightarrow id_a$ is injective, $(e_a, d_a) \longleftarrow \mathcal{K}(1^\eta)$, and $(v_a, s_a) \longleftarrow \mathcal{G}(1^\eta)$. The bitstrings id_a, e_a, v_a are public; d_x and s_x are public if $x \notin \mathcal{H}$.*
- *$name \in \mathsf{Name} \rightharpoonup \{0,1\}^\eta$ is a partial function defined at least on every name that occurs free in C or D. The bitstring $name(m)$ is public for every name $m \notin \widetilde{n}$.*
- *$certval$ is a partial function from certificates $b\{V\}_\ell$ to $s \in \{0,1\}^\eta$ such that $\mathcal{V}(v_b, [\![\ulcorner V \urcorner^{\mathsf{D},b}]\!]_b, s) = 1$, defined at least on the certificates of D, Φ, and $b\{V\}_\ell$ of P_a with $a \neq b$. The bitstring $certval(V)$ is public when $V \in \Phi$ or V issued by $x \notin \mathcal{H}$.*

- *wire is a partial function from identifiers i to (M, k, s, del) defined at least on I, where $M = a{:}b\langle V\rangle$ with $a, b \in \mathcal{H}$, the bitstrings s and k are the output and the authentication key produced by $\mathtt{send}_b$ on input $[\![\ulcorner V\urcorner^{\mathsf{D},a}]\!]_a$, and $\mathit{del} = 0$ if $i \in I$ and $\mathit{del} = 1$ otherwise. The bitstrings s and del are public.*
- *$\mathit{keycache}$ is a function from $a \in \mathcal{H}$ to sets of bitstrings such that, if $\mathit{wire}(i) = (M, k, s, \mathit{del})$ with M to a, then $k \in \mathit{keycache}(a)$ if and only if $i \notin I$.*
- *$\mathit{signedkeys}$ is a function from $a \in \mathcal{H}$ to sets of bitstrings $(k, \mathit{id}_x, \mathit{sig})$ such that $\mathcal{V}(v_a, (k, \mathit{id}_x), \mathit{sig}) = 1$. These tuples are public when $x \notin \mathcal{H}$.*

Intuitively, wire records all messages sent between honest principals; $\mathit{keycache}(a)$ records all messages received by a so far. When D is clear from the context, we write $\mathit{prin}(a)$ instead of $\mathsf{D}.\mathit{prin}(a)$, and similarly for the other components of D.

Definition 15 (Concrete Terms and Processes). *A shadow state D defines a partial map from high-level terms V to internal terms of a, as follows:*

- $\ulcorner n\urcorner^{\mathsf{D},a} = \mathtt{name}(\mathit{name}(n))$, *for any name n;*
- $\ulcorner x\urcorner^{\mathsf{D},a} = \mathtt{prin}(\pi_1(\mathit{prin}(x)))$ *for any principal $x \in \mathsf{Prin}$;*
- $\ulcorner a\{V\}_{\mathsf{0}}\urcorner^{\mathsf{D},a} = \ulcorner a\urcorner^{\mathsf{D},a}\{\ulcorner V\urcorner^{\mathsf{D},a}\}_{\mathsf{0}}$;
- $\ulcorner x\{V\}_\ell\urcorner^{\mathsf{D},a} = \ulcorner x\urcorner^{\mathsf{D},a}\{\ulcorner V\urcorner^{\mathsf{D},a}\}_s$ *where $s = \mathit{certval}(x\{V\}_\ell)$;*
- $\ulcorner \mathtt{f}(V_1, \ldots, V_n)\urcorner^{\mathsf{D},a} = \mathtt{f}(\ulcorner V_1\urcorner^{\mathsf{D},a}, \ldots, \ulcorner V_n\urcorner^{\mathsf{D},a})$ *for any $f \neq \mathtt{cert}$ with arity n.*

We extend this map to translate local processes to low-level processes, as follows: high-level terms within local processes are translated as above, except for variables and locally-bound names (left unchanged); high-level patterns are translated by applying the translation to all high-level terms in the pattern and leaving the rest unchanged; local processes P running on behalf of principal a are translated to internal processes $\ulcorner P\urcorner^{\mathsf{D},a}$ by translating their high-level terms to internal terms.

In particular, if D is a shadow state for S, and $a \in \mathsf{Prin}$ then $\ulcorner\cdot\urcorner^{\mathsf{D},a}$ is defined for every subterm and subprocess of S and D. We often write $\ulcorner V\urcorner$ instead of $\ulcorner V\urcorner^{\mathsf{D},a}$ when D and a are clear from the context. Our intent is that, with overwhelming probability, we have $V = V'$ iff $\ulcorner V\urcorner^{\mathsf{D},a} = \ulcorner V'\urcorner^{\mathsf{D},a}$ whenever D defines these representations.

Definition 16. *Let S be a system with shadow state D. The implementation of S and D is the collection of machines $\mathsf{M}(S, \mathsf{D}) = (\mathsf{M}_a(S, \mathsf{D}))_{a\in\mathcal{H}}$ where each machine $\mathsf{M}_a(S, \mathsf{D})$ has the following state: $\mathit{id}_a, d_a, s_a, \mathit{peers}_a$ are read from prin; $p_a = \ulcorner P_a\urcorner^{\mathsf{D},a}$; $\mathit{keycache}_a = \mathit{keycache}(a)$; $\mathit{signed}_a(a\{V\}_{\mathsf{0}}) = \mathit{certval}(a\{V\}_{\mathsf{0}})$ when defined.*

Low-Level Initialization Our soundness results show that $\mathsf{M}(S, \mathsf{D})$ captures (with overwhelming probability) any reachable state of our implementation in the presence of a PPT adversary. Still, they do not indicate whether an adversary may actually lead our implementation to state $\mathsf{M}(S, \mathsf{D})$ using an interactive run (Definition 1). We say that D is a *valid shadow* for S when such a run exists. Relying on the high-level initialization protocols of Appendix B, for every safe stable system S, we can build a valid shadow D reachable from an initial system S° (with no shared names or certificates and no intercepted messages) using labeled transitions $S^\circ \xrightarrow{\varphi^\circ} S$, as follows:

- S° has a simple initial shadow D° obtained from D by erasing everything except $prin$; to initialize $\mathsf{M}(S^\circ, \mathsf{D}^\circ)$, we simply generate all keys, pass them to every machine so that they reach the initial local state $\mathsf{M}_a(S^\circ, \mathsf{D}^\circ)$, and start the adversary with input $public(\mathsf{D}^\circ)$ (as prescribed in step 3 of Definition 1).
- As a corollary of soundness for traces, if D is a valid shadow for S there exists a PPT algorithm $\mathsf{B}_{\varphi^\circ}$ that takes as input $public(\mathsf{D}^\circ)$ and, interacting with $\mathsf{M}(S^\circ, \mathsf{D}^\circ)$, outputs $public(\mathsf{D})$ and leaves the machines in state $\mathsf{M}(S, \mathsf{D})$ with overwhelming probability.

Accordingly, we *define* a low level run starting from S with shadow D against B, written $\mathsf{B}[\mathsf{M}(S, \mathsf{D})]$, as $(\mathsf{B}_{\varphi^\circ}; \mathsf{B})[\mathsf{M}(S^\circ, \mathsf{D}^\circ)]$ where $\mathsf{B}_{\varphi^\circ}; \mathsf{B}$ represents an adversary that first runs $\mathsf{B}_{\varphi^\circ}$ and then runs B with input $public(\mathsf{D})$.