A Formal Implementation of Value Commitment

Cédric Fournet[2,1], Nataliya Guts[1], and Francesco Zappa Nardelli[3,1]

[1] MSR-INRIA Joint Centre
[2] Microsoft Research
[3] INRIA

Abstract. In an optimistic approach to security, one can often simplify protocol design by relying on audit logs, which can be analyzed a posteriori. Such auditing is widely used in practice, but no formal studies guarantee that the log information suffices to reconstruct past runs of the protocol, in order to reliably detect (and provide evidence of) any cheating. We formalize audit logs for a sample optimistic scheme, the value commitment. It is specified in a pi calculus extended with committable locations, and compiled using standard cryptography to implement secure logs. We show that our distributed implementation either respects the abstract semantics of commitments or, using information stored in the logs, detects cheating by a hostile environment.

1 A cautiously optimistic approach to security

Mutual distrust in distributed computing makes enforcing system-wide security assurances particularly challenging. Common protocols perform an important number of mandatory runtime checks and allow only legal computations to progress: in session-establishment protocols, for instance, a strong security invariant is usually enforced at every step of the run of the protocol. These runtime checks have a cost, in terms of cryptographic and networking operations; they may also conflict with other goals of the protocol, such as confidentiality.

A different approach, which we call optimistic, presumes instead that all involved principals are honest and well-behaved, and thus omits some runtime checks. Traces of protocol runs are stored in a secure log and can be used a posteriori to verify the compliance of each principal to its role: principals who attempt non-compliant actions will be blamed using the logged evidence. The security invariant is weaker than those achieved by more conservative protocols, but adequate for many non-critical applications.

Some protocols inherently rely on logs to establish their security properties. These protocols are often based on a *commitment* scheme. A principal commits to a value kept hidden; other principals of a system cannot read this value, but have a procedure to detect any change to the value after the commitment. Distant coin flipping is a simple protocol that illustrates commitments: suppose that A and B are not physically at the same place and want to toss a coin. Both A and B flip their own coin, exchange commitments on their results, then reveal and compare these results; A wins the toss if the two results are the same. For fairness, A's commitment should neither reveal any information to B, nor enable A to change her committed result after receiving B's.

Commitment is a building block for many protocols such as mental poker [3], sealed bid auctions, e-voting [6, 5], and online games [12]. For instance, mental poker relies on commitment to build a fair shuffling of the deck, then gradually reveal cards as the game proceeds. At the end of the game, the deck permutations used by each player can be revealed for auditing purposes.

Secure logging is not only an essential component of optimistic schemes, but is also widely used in standard practice. Much research effort has been devoted to techniques for implementing logs so as to guarantee properties such as *correctness*, *forward integrity*, and *forward secrecy* [15, 18, 17]. Still, which data should be logged? and why? Between general recommendations such as "an audit trail should include sufficient information to establish what events occurred and who (or what) caused them" [14, 11] and efficient implementation techniques, we are not aware of any formal studies that characterize and verify the security properties achieved by protocols relying of logs.

In this paper, we give a formal answer to this question for the commitment scheme. We extend a simple distributed language, the applied pi calculus [1], with commitment datatypes and primitives, and we illustrate this extension by programming an online game. To abstract away from the possible misbehaviors of the environment, we propose a trustful and strong operational semantics for our commitment primitives. We show that our language can be compiled to the applied pi calculus, using standard cryptographic primitives, with adequate protection against an arbitrary, possibly hostile environment. We obtain an important security property stating that, for any source systems, our distributed implementation either respects the semantics of commitments or, using information stored in the logs, detects (and proves) cheating by a hostile environment.

Related work Value commitments appear in formal models of protocols (e.g. [13]) and implementations of language abstractions (e.g. [19]). More closely related to our work, Etalle et al. [10, 4] advocate the usage of logs for optimistic security enforcement. They formalize audit-based discretionary access control in collaborative work environments, and develop a logical framework for user accountability; they also design cryptographic support for communication evidence in a decentralized setting [8].

Contents Section 2 presents our source language with value commitment. Section 3 illustrates the use of commitment for programming online games. Section 4 describes the language implementation, as a cryptographic translation to the applied pi calculus. Section 5 develops a labeled semantics and an extended translation to keep track of source-program invariants. Section 6 states our main results. Section 7 reports on our prototype implementation. Section 8 discusses future work.

Additional details appear online, at http://www.msr-inria.inria.fr/projects/sec/logs, including complete definitions for the source and target semantics and all proofs.

2 A language with value commitment

The applied pi calculus is a process language parametrized by an equational theory on terms, which provides flexible support for modeling symbolic cryptographic primitives and data structures. We refer to [1] for a general presentation of its semantics.

To express the value commitment scheme, we extend an instance of applied pi with *committable cells*. The grammar for terms (M, V), processes (P), and systems (A) is given below. Our extensions to the standard syntax appear in grey boxes.

$$\begin{array}{lll}
M,\ V & ::= & \\
 & \mid & u \\
 & \mid & \mathit{func}\,(\widetilde{M}) \\
 & \mid & u.\mathbf{Idu} \\
 & \mid & u.\mathbf{Idc}(p) \\
 & \mid & u.\mathbf{Rd}(p\ M)
\end{array}
\qquad
\begin{array}{lll}
P & ::= & \\
 & \mid & 0 \\
 & \mid & P_1 \mid P_2 \\
 & \mid & \nu\, c\,.\,P \\
 & \mid & u?(x).P \\
 & \mid & u!\langle M\rangle.P \\
 & \mid & \mathsf{if}\ M = M'\ \mathsf{then}\ P\ \mathsf{else}\ P' \\
 & \mid & \mathsf{repl}\ P \\
 & \mid & \mathsf{newloc}\,(x, y).P \\
 & \mid & \mathsf{commit}\ M\ u\,(x).P
\end{array}
\qquad
\begin{array}{lll}
A & ::= & \\
 & \mid & 0 \\
 & \mid & A_1 \mid A_2 \\
 & \mid & \nu\, u\,.\,A \\
 & \mid & \{\, M \,/\, x\,\} \\
 & \mid & p[P] \\
 & \mid & u.(p) \\
 & \mid & u.(p\ M)
\end{array}$$

Terms are built from variables (denoted $x, y, \ldots$), names (denoted $c, l, s, \ldots$), function applications, and capabilities (described below). We assume that functions include at least a pairing function, denoted $+$, with associated projections $+_1, +_2$ and equations $+_i(x_1 + x_2) = x_i$ for $i = 1, 2$. (Our results extend to arbitrary data structures; we use integer constants in examples.) The metavariable u ranges over names and variables. Among names, we distinguish the set of *principals*, denoted p, a, e, and the set of *location names*, ranged over by l. Contrarily to standard applied pi, each process P runs under the control of a principal p, denoted $p[P]$.

Committable cells and capabilities A cell is a memory location owned by a principal who can, once, commit its content to a value of its choice. In addition, the owner can pass capabilities to other principals, thereby granting these principals partial read access to the cell.

Our language features three kinds of capabilities. The *read capability* $l\,.\,\mathbf{Rd}\,(p\ \ M)$ is created by the owner p of the location l when it commits to a value M. Any principal can use a read capability to read the content of the location associated to the capability. The *identity capabilities* instead partially disclose the state of a cell without actually revealing the value possibly committed. So the *committed id capability* $l\,.\,\mathbf{Idc}\,(p\ )$ proves that the location l is committed and reveals the owner p of the location. The *uncommitted id capability* $l\,.\,\mathbf{Idu}$ just asserts the identity l of the location.

The language of terms is sorted: we distinguish *marshallable values*, that include all the terms except location and channel names, and *committable values*, that include all marshallable values except those that mention committed id and read capabilities.

The state of each committable cell is represented by a process: $l.(p\ )$ denotes an uncommitted cell named l owned by p; $l.(p\ M)$ denotes the same cell once it has been committed to the committable value M. Two new kinds of processes manipulate cells. The newloc process creates a fresh, uncommitted location and binds both its unique identifier l (from $\mathcal{L}$) and its uncommitted capability in its continuation:

$$a[\mathsf{newloc}\,(x, y).P] \ \longrightarrow\ \nu\, l\,.\,(l.(a\ ) \ \mid\ a[P\{^{l}/_{x}\}\{^{l\,.\,\mathbf{Idu}}/_{y}\}])$$

where l is fresh for P. The unique identifier l can then be used to commit an uncommitted cell to some committable value M:

$$l.(a\ ) \ \mid\ a[\mathsf{commit}\,M\ l\,(x).P] \ \longrightarrow\ l.(a\ M) \ \mid\ a[P\{^{l\,.\,\mathbf{Rd}\,(a\ \ M)}/_{x}\}]$$

The commit process yields a read capability for the newly-committed cell. The sort system does not allow to communicate or store in another location the cell name l: hence, only the principal that created the cell can commit a value into it. The abbreviation newcommit creates a new committed location (where x', x'' are fresh for P):

$$p[\mathsf{newcommit}\ M\ (x).P] \stackrel{\text{def}}{=} p[\mathsf{newloc}\ (x', x'').\mathsf{commit}\ M\ x'\ (x).P]$$

Capabilities can be communicated over channels; they can also be manipulated using special functions, according to the equational theory below.

$$\mathsf{read}(x\,.\,\mathbf{Rd}\,(p\ \ v)) = v \qquad \mathsf{get\_idc}(x\,.\,\mathbf{Rd}\,(p\ \ v)) = x\,.\,\mathbf{Idc}\,(p\ )$$
$$\mathsf{get\_idu}(x\,.\,\mathbf{Idc}\,(p\ )) = x\,.\,\mathbf{Idu} \qquad \mathsf{get\_prin}(x\,.\,\mathbf{Idc}\,(p\ )) = p$$
$$\mathsf{is\_idu}(x\,.\,\mathbf{Idu}) = \mathsf{ok} \qquad \mathsf{is\_idc}(x\,.\,\mathbf{Idc}\,(p\ )) = \mathsf{ok} \qquad \mathsf{is\_rd}(x\,.\,\mathbf{Rd}\,(p\ \ v)) = \mathsf{ok}$$

The read function yields the value from read capabilities. Since the read capability is generated when committing the cell, the semantics of the source language guarantees that all reads for a given cell always return the same value. The get_prin function yields the principal that owns the cell from committed capabilities. (We could also provide get_prin from uncommitted capabilities, at some additional cost in the cryptographic implementation.) The get_idu and get_idc functions downgrade capabilities, yielding a more restrictive capability for the same cell. Hence, get_idu yields an uncommitted capability, which can be used only to identify the cell, whereas get_idc takes a read capability and hides its committed value. The language finally has functions that support dynamic typechecking of capabilities. In particular, $\mathsf{is\_idc}(x) = \mathsf{ok}$ or $\mathsf{is\_rd}(x) = \mathsf{ok}$ implies that the cell associated with x is committed.

3 Example: an online game

Our example describes a game run by a server a_0, between n players $a_1, \ldots, a_n$. The game is played in one turn, with all players revealing their moves simultaneously. (A simple instance of the game with $n = 2$ is Rock, Paper, Scissors.) The players and the server are willing to cooperate, but with minimal trust assumptions between them; however, it is deemed sufficient to detect any dishonest principal at the end of the game. Similar examples include multiparty protocols for online auctions, voting, or partial-information games [16, 3, 6].

The protocol has three exchange rounds between the server and each player, using channels c_i for $i = 1..n$: (1) the server sets up the game, distributes the details to the players, and collects their sealed moves; (2) the server distributes all the players' sealed moves and collects their actual moves; (3) the server distributes the result of the game.

We begin with the server code, given below. For simplicity, the code does not provide any error handling—execution stops when a test fails.

$$\begin{aligned}A_0 = a_0[&\mathsf{newloc}\,(l, \mathit{result}_{id}).\mathsf{newcommit}\ \mathit{result}_{id} + \mathit{details}\ (\mathit{challenge}).\\ &\big(c_i!\langle \mathit{challenge}\rangle.c_i?(\mathit{promise}_i).\mathsf{if}\ \mathsf{get\_prin}(\mathit{promise}_i) = a_i\ \mathsf{then}\ \big)_{i=1..n}\\ &\mathsf{newcommit}\ \mathit{challenge} + \widetilde{\mathit{promise}}\ (\mathit{game}).\\ &\big(c_i!\langle \mathit{game}\rangle.c_i?(\mathit{move}_i).\mathsf{if}\ \mathsf{get\_idc}(\mathit{move}_i) = \mathit{promise}_i\ \mathsf{then}\ \big)_{i=1..n}\\ &\mathsf{commit}\ \mathsf{winner}(\widetilde{\mathit{move}}, \mathit{challenge})\ l\ (\mathit{result}).\big(c_i!\langle \mathit{result}\rangle.0\big)_{i=1..n}]\end{aligned}$$

In round (1), the server creates an uncommitted cell l for storing the outcome of the game, and a readable cell $challenge$ that provides the identifier for l and the (unspecified) details of the game. Upon receiving each player's response, the server authenticates it as a committed capability from that player. In round (2), the server creates a second committed cell that binds the challenge to the received commitments from all players. Upon receiving each player's second response, the server correlates it as the read capability associated with their first response. In round (3), the server has all the players' information: it resolves the game and finally commits the cell l to the published result of the game (which may include, for instance, selected information from the players' moves). We omit the code for the function winner that computes this result.

The code for the players performs symmetric operations:

$$\begin{aligned} A_i = a_i[\, & c_i?(challenge).\mathsf{if}\ \mathsf{get\_prin}(\mathsf{get\_idc}(challenge)) = a_0\ \mathsf{then} \\ & \mathsf{newcommit}\ z_i\ (move_i).c_i!\langle \mathsf{get\_idc}(move_i)\rangle. \\ & c_i?(game).\mathsf{if}\ \mathsf{valid\_game}\ (\ game\ ,\ challenge\ ,\ move_i\ )\ \mathsf{then} \\ & \quad c_i!\langle move_i\rangle.c_i?(result_i).\mathsf{if}\ \mathsf{no\_cheat}\ (\ result_i\ ,\ \mathsf{read}(game)\ )\ \mathsf{then}\ P_i] \end{aligned}$$

In round (1), after receiving the challenge, each player confirms its validity, for instance by checking that it is a genuine readable capability from a_0, then it selects a move and sends back its commitment. In round (2), after receiving all commitments, the player correlates them to the challenge and verifies that its own commitment is recorded (using for instance valid_game) then it releases its move in clear. In round (3), the player checks the outcome of the game and verifies a posteriori that the server followed the rules (using for instance no_cheat). The tests are defined as follows:

$$\begin{aligned} \mathsf{valid\_game}\ (\ x_1\ ,\ x_2\ ,\ x_3\ ) &\overset{\text{def}}{=} +_1(\mathsf{read}(x_1)) = x_2\ \mathsf{and}\ \mathsf{get\_idc}(x_3) \in +_2(\mathsf{read}(x_1)) \\ \mathsf{no\_cheat}\ (\ x\ ,\ y\ ) &\overset{\text{def}}{=} \mathsf{get\_idu}(\mathsf{get\_idc}(x)) = +_1(y)\ \mathsf{and}\ \mathsf{get\_idc}(x) \in +_2(y) \end{aligned}$$

Guarantees offered to the players We distinguish *language level* guarantees, enforced by the abstract semantics of locations, and *application level* guarantees, relying on high-level, application-specific checks on top of the language semantics. For each kind of guarantees, we also distinguish between immediate (conservative) and deferred (optimistic) enforcement. For instance, enforcement may be deferred until the content of a cell becomes readable.

As an illustration of immediate language-level checks, committed values offer basic authentication guarantees to the participants. For instance, each player has the privilege to choose its moves, and the move is securely attributed to the player even if the communication channels c_i are unprotected; participants can also check this attribution later.

To protect application integrity, the code must perform sufficient checks before proceeding with the game. Systematic testing of the owner identities for the received capabilities avoids unauthorized, possibly non-accountable, participants. Some checks are immediate, e.g. testing if two capabilities are associated to the same location; other checks that depend on the commitment semantics are delayed. In the example, players are guaranteed that they all get the same result (if any) for any given game, since they must get the same location read capability, but it is up to the application code to correlate the received read capability to the initial uncommitted capability.

At the same time, the applicative logic of our protocol guarantees that, even if the server is willing to leak information to the other players, those players cannot get that information before committing to their own moves.

4 Distributed cryptography implementation

The target language is an instance of applied pi, with standard (symbolic) cryptographic primitives and data structures but without ad-hoc rules or constructs for locations.

We rely on a cryptographic hash function, denoted h, and a public-key signature mechanism satisfying the equation $\mathsf{verify}(v, \mathsf{sign}(v, \mathsf{sk}(m)), \mathsf{pk}(m)) = \mathsf{ok}$. The functions $\mathsf{sk}(m)$ and $\mathsf{pk}(m)$ generate a pair of secret/public keys from a nonce m. All other data constructors admit a projection function $\mathit{func}_i(\mathit{func}(x_1, \ldots, x_n)) = x_i$.

To every principal p, we associate a keypair and export its public key tagged with constructor prin using an active substitution of the form $\{ \mathsf{prin}(\mathsf{pk}(m_p)) / p \}$.

Cryptographic implementation of capabilities We compile the capabilities associated to a location $l.(p\ V)$ as follows:

$l\,.\,\mathbf{Rd}\,(p\ \ V)$	$\mathsf{rd}(p, s, [\![V]\!], w)$
$l\,.\,\mathbf{Idc}\,(p\,)$	$\mathsf{idc}(p, \mathsf{h}(s) + \mathsf{h}(s + [\![V]\!]), w)$
$l\,.\,\mathbf{Idu}$	$\mathsf{idu}(\mathsf{h}(p + \mathsf{h}(s)))$

where $p = \mathsf{prin}(\mathsf{pk}(m_p))$ is the owner's public key, s is a fresh value used as a seed, and $w = \mathsf{sign}(\mathsf{h}(s) + \mathsf{h}(s + [\![V]\!]), \mathsf{sk}(m_p))$ signs the committed value $[\![V]\!]$.

A read capability is a tagged tuple that includes these elements. A committed id capability is a tagged tuple that provides p and verifiable evidence of the commitment without actually revealing $[\![V]\!]$. To this end, it includes both a hash of the committed value, first concatenated with the seed s, to protect against brute force attacks, yielding $\mathsf{h}(s + [\![V]\!])$, and the hash $\mathsf{h}(s)$, to enable the receiver to correlate the owner and signature with a previously-received uncommitted id capability by recomputing the identifier $\mathsf{h}(p + \mathsf{h}(s))$. An uncommitted id capability just includes this unique location identifier, which may be compared to other capabilities and, later, associated with p and s. The receiver can compute committed capabilities from read capabilities, and uncommitted capabilities from committed capabilities, but not the converse.

The signature w authenticates read and committed id capabilities, binding their content to the owner's key $\mathsf{sk}(m_p)$. Their receiver can extract p and $\mathsf{h}(s) + \mathsf{h}(s + [\![V]\!])$ from these tagged tuples and use them to verify w. When the signature is valid, the public key identifies the owner of the location associated to the capability.

Detection of multiple commitments In a typical run, an honest principal receives a commitment to some value from the principal p, say $\mathsf{idc}(p, v_1 + v_2, w)$, and later the value itself, say $\mathsf{rd}(p, s, z, w')$. The receiver can easily check that the two capabilities refer to the same location, by testing $\mathsf{h}(s) = v_1$, and verify the two signatures $w = \mathsf{sign}(v_1 + v_2, \mathsf{sk}(m_p))$ and $w' = \mathsf{sign}(\mathsf{h}(s) + \mathsf{h}(s + z), \mathsf{sk}(m_p))$. If these tests succeed, then the receiver can check whether $v_2 = \mathsf{h}(s + M)$: if the test fails, the principal p can be convicted of multiply committing the location identified by $\mathsf{h}(p + \mathsf{h}(s))$.

In preparation for the translation, we introduce functions that operate on tuples representing capabilities in the target language. For instance, the function read implements source-language reads as a projection, and check_idc verifies the seal of committed ids.

$$\begin{aligned}
\mathsf{read}(x) &\overset{\text{def}}{=} \mathsf{rd}_3(x)\\
\mathsf{get\_idc}(x) &\overset{\text{def}}{=} \mathsf{idc}(\mathsf{rd}_1(x)\,,\ \mathsf{h}(\mathsf{rd}_2(x)) + \mathsf{h}(\mathsf{rd}_2(x) + \mathsf{rd}_3(x))\,,\ \mathsf{rd}_4(x))\\
\mathsf{check\_idc}(x) &\overset{\text{def}}{=} \mathsf{verify}(\mathsf{idc}_2(x)\,,\ \mathsf{idc}_3(x)\,,\ \mathsf{prin}_1(\mathsf{idc}_1(x))) \ = \ \mathsf{ok}\\
\mathsf{get\_idu}(x) &\overset{\text{def}}{=} \mathsf{idu}(\mathsf{h}(\mathsf{idc}_1(x) + (+_1\ \mathsf{idc}_2(x))))
\end{aligned}$$

In general, inconsistent capabilities may be scattered in the whole system. To detect such inconsistencies and reliably blame cheating principals, a compiled system logs all the committed capabilities generated or received by honest principals by sending them over the channel *log* to the following resolution process R:

$$\begin{aligned}
R = \mathsf{repl}\ & \mathit{log}?(y_1).\mathit{log}?(y_2).\\
& \mathsf{if}\ \mathsf{check\_idc}(y_1)\ \mathsf{and}\ \mathsf{check\_idc}(y_2)\ \mathsf{then}\\
& \mathsf{if}\ \mathsf{get\_idu}(y_1) \ = \ \mathsf{get\_idu}(y_2)\ \mathsf{and}\ \mathsf{idc}_2(y_1) \neq \mathsf{idc}_2(y_2)\ \mathsf{then}\ \mathit{bad}!\langle \mathsf{get\_prin}(y_1)\rangle
\end{aligned}$$

This resolution process repeatedly reads pairs of Idc capabilities over the *log* channel and tests them for inconsistencies, as described above. If cheating is detected, the principal is blamed on channel *bad*. The resolution process acts as an external judge auditing the compiled system, and the data sent over the channel *log* as a secure audit trail. Since all messages on *log* are replicated, log entries cannot be erased or modified by a malicious principal, and every principal may run its own copy of process R. At the same time, a malicious principal cannot forge capabilities that would accuse an honest principal, as it cannot produce a valid seal associated with the honest principal.

Translation of initial configurations Protocol descriptions can be expressed as initial configurations of a source system that do not contain, or refer to, locations and capabilities; these are created later, during the run of the protocol. We describe the translation of such configurations; a full treatment of capabilities and locations is deferred to Section 5. Our translation is a homomorphism over terms and over most systems.

$$[\![x]\!] = x \qquad [\![c]\!] = c \qquad [\![\mathit{func}(M_1\,,\ \ldots\,,\ M_n)]\!] = \mathit{func}([\![M_1]\!]\,,\ \ldots\,,\ [\![M_n]\!])$$

$$[\![\![A]\!]\!] = [\![A]\!] \mid R \mid E \qquad [\![a[P]]\!] = \nu\, m_a\,.\,([\![P]\!]_a \mid \{\, \mathsf{prin}(\mathsf{pk}(m_a)) \,/\, a \,\})$$

$$[\![A_1 \ \mid\ A_2]\!] = [\![A_1]\!] \ \mid\ [\![A_2]\!] \qquad [\![\nu\, u\,.\,A]\!] = \nu\, u\,.\,[\![A]\!] \qquad [\![\{\, M \,/\, x \,\}]\!] = \{\, [\![M]\!] \,/\, x \,\}$$

Let $\mathcal{A}$ the set of principals running a process in the system and $\mathcal{E}$ the set of other (possibly dishonest) principals whose names occur in the system ($\mathcal{E} = \mathcal{P} \cap \mathrm{fn}(A) \setminus \mathcal{A}$).

For each principal $a \in \mathcal{A}$, the translation creates a secret seed m_a used to generate the pair of secret/public keys of the principal. The public key is published using an active substitution, while the process run by the principal is compiled within the scope of the private seed m_a used for signing. Similarly, the translation includes active substitutions $E = \prod_{e\in\mathcal{E}}(\{\, \mathsf{prin}(\mathsf{pk}(m_e)) \,/\, e \,\} \mid \{\, H_e \,/\, m_e \,\})$ that records, for each principal $e \in \mathcal{E}$, a public key $\mathsf{pk}(m_e)$ and an associated secret H_e. The translation also spawns a replicated resolution server R.

The translation of processes is given next. (We omit the homomorphic clauses for 0, $P_1 \mid P_2$, $\mathsf{repl}\ P$, and $\nu\, c\,.\,P$).

$$\llbracket \mathsf{newloc}\,(x,y).P\rrbracket_a = \nu\, {s'}_l\,.\,\nu\, c_l\,.\,(c_l!\langle \mathsf{None}\rangle \mid \llbracket P\rrbracket_a \,\{{}^{c_l}/_{c_x}\}\,\{{}^{{s'}_l}/_{s_x}\}\,\{{}^{\mathsf{idu}(\mathsf{h}(a+\mathsf{h}({s'}_l)))}/_{y}\})$$

$$\llbracket \mathsf{commit}\ V\ x\,(x').P\rrbracket_a = c_x?(y).(\llbracket P\rrbracket_a \mid \mathsf{repl}\ \mathit{log}!\langle \mathsf{idc}(a\,,\ v_x\,,\ w_x)\rangle)$$
$$\{{}^{\mathsf{h}(s_x)+\mathsf{h}(s_x+\llbracket V\rrbracket)}/_{v_x}\}\,\{{}^{\mathsf{sign}(v_x\,,\,\mathsf{sk}(m_a))}/_{w_x}\}\,\{{}^{\mathsf{rd}(a\,,\,s_x\,,\,\llbracket V\rrbracket\,,\,w_x)}/_{x'}\}$$

```
parse $x$ $P$ =
  if is_rd($x$) = ok then
    if check_idc(get_idc($x$)) then parse read($x$) ($P$ | repl $log!\langle$get_idc($x$)$\rangle$) else $r!\langle$None$\rangle$
  else if is_idc($x$) = ok then if check_idc($x$) then $P$ | repl $log!\langle x\rangle$ else $r!\langle$None$\rangle$
    else if is_prin($x$) = ok or is_idu($x$) = ok then $P$
      else if is_pair($x$) = ok then parse $(+_1\ x)$ (parse $(+_2\ x)$ $P$) else $r!\langle$None$\rangle$
```

$$\llbracket c!\langle M\rangle.P\rrbracket_a = c!\langle\llbracket M\rrbracket\rangle.\llbracket P\rrbracket_a$$

$$\llbracket c?(x).P\rrbracket_a = \nu\, r\,.\,(c?(x).\mathsf{parse}\ x\ \llbracket P\rrbracket_a \mid \mathsf{repl}\ (r?(\_).c?(x).\mathsf{parse}\ x\ \llbracket P\rrbracket_a))$$

$$\llbracket \mathsf{if}\ M\ =\ M'\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\rrbracket_a = \mathsf{if}\ \llbracket M\rrbracket\ =\ \llbracket M'\rrbracket\ \mathsf{then}\ \llbracket P_1\rrbracket_a\ \mathsf{else}\ \llbracket P_2\rrbracket_a$$

The translation of newloc creates a fresh location seed s'_l and a local channel c_l (with a message None, recording that the location is uncommitted), and substitutes c_l for c_x, s'_l for s_x and the idu capability for y in the continuation.

The translation of commit can proceed only if the location has not been previously committed (the message on c_x provides mutual exclusion); it then substitutes the rd capability for x' in the continuation code. It also generates the corresponding idc capability for the location and logs it by sending it to the resolution protocol.

The parse function filters any received value received over channels. If the value is tagged with rd or idc, then it might (or not) be a valid capability, depending on the validity of its embedded signature: valid capabilities are passed to the continuation, while the associated idc is sent to the resolution protocol. If the value is tagged as a principal or an uncommitted capability, it is always passed to the continuation. For compound data, here pairs, each element is separately parsed. Other values, as well as non-valid committed capabilities, are silently discarded. In the translation of an input, we assume that the channel r is fresh for $\llbracket P\rrbracket_a$, and use this channel to loop after discarding such values.

5 Model and translation of environment interactions

We define a labeled source semantics that explicitly captures all possible interactions between a system composed of honest principals and an abstract environment composed of potentially hostile principals. To maintain the committable-cell invariants, this semantics keeps track of the capabilities exported to the environment and of the partial knowledge acquired when receiving capabilities from the environment. We then extend our translation from initial configurations to any such reachable configuration.

Extended location states and capabilities We use overlapping syntaxes for capabilities appearing in values, in transition labels, and in the processes representing the state of the cells. Their general form is $l\,.\,Cap\,(\,[\,p\,]\,[\,H\,]\,[\,V\,]\,)$, where l is the location identifier; $Cap \in \{0, \mathbf{Idu}, \mathbf{Idc}, \mathbf{Rd}\}$ is a capability tag; p is a principal name; H

ranges over terms of the target language; and V is a value of the source language. (This syntaxes extend those given in Section 2 for capabilities and location states, with $l.(a\ M) = l\,.\,0\,(a\ \ M)$). The fields p, H, and V are optional. The presence of a value V indicates that the location is committed to this value. The term H plays no role in the source language, but is technically convenient in its translation: it enables us to represent any reachable state of our implementation as the translation of a source system.

The interpretation of Cap depends on the principal p that owns the location. If a location is owned by $a \in \mathcal{A}$, then Cap represents the most permissive capability *sent to* the environment (and H is omitted), with $Cap = 0$ when no capabilities have been exported so far. If a location is owned by $e \notin \mathcal{A}$, then Cap represents the most permissive capability *received from* the environment (and H records some opaque cryptographic value in its received representation).

Ordering capabilities We formalize the notion of "more permissive capability" by defining a preorder $\preceq$ on capabilities. Intuitively, $C \preceq C'$ holds if C and C' have compatible contents and C can be derived from C' using the equational theory. We also introduce a special capability $\bot$ that represents the absence of knowledge on a location. The order is defined by the axioms below:

$$\bot \preceq 0\,ct \qquad 0\,ct \preceq \textbf{Idu}\,ct \qquad \textbf{Idu}\,\mathsf{f}_u\,(ct) \preceq \textbf{Idc}\,ct \qquad \textbf{Idc}\,\mathsf{f}_c\,(ct) \preceq \textbf{Rd}\,ct$$
$$Cap\,(p\ H) \preceq Cap\,(p\ H\ V)$$

where ct is any fixed contents and f_u and f_c are fixed functions that rewrite H in ct. We write $C \curlyvee C'$ for the sup of C and C' with respect to $\preceq$, when it exists.

Normal form We say that a system is in *normal form* when it is of the form

$$S \ = \ \nu\mathcal{N}\,(\textstyle\prod_{l\in\mathcal{L}} l\,.\,C_l \mid \prod_{a\in\mathcal{A}} a[P_a] \mid \phi)$$

for some finite sets of names $\mathcal{N}$, $\mathcal{L}$, and $\mathcal{A}$ and active substitutions ϕ. Every initial configuration can be written in normal form (with $\mathcal{L} = \emptyset$) using structural equivalence. A system S is *well-formed* when it is structurally equivalent to a normal form such that if l is a location name within S then $l \in \mathcal{L}$ and l occurs only

1. in terms $l.C$ such that: (a) if $\mathsf{get\_prin}(l\,.\,C_l) \in \mathcal{A}$, then C and C_l are owned by the same principal and if C has a value, then C_l has the same value; and (b) if $\mathsf{get\_prin}(l\,.\,C_l) \notin \mathcal{A}$, then $C \preceq C_l$ (informally, for a cell owned by the environment, the system cannot have capabilities more permissive than those received);
2. in subprocesses $\mathsf{commit}\,M\ l\,(x).P$ of P_a when $a = \mathsf{get\_prin}(l\,.\,C_l)$;
3. in $\mathcal{N}$ when $\mathsf{get\_prin}(l\,.\,C_l) \in \mathcal{A}$ and $C_l = 0\,ct$.

In the labeled semantics below, we require that the initial and final systems and the label be well-formed. We define labeled transitions $A \xrightarrow{\alpha} A'$ between source systems on top of an auxiliary relation $C \xrightarrow{\gamma} C'$ between capabilities.

Labeled transitions on capabilities Input/output actions with the environment can affect the state of memory cells. To model these updates compositionally we define a labeled transition semantics between capabilities.

$$\frac{}{C \xrightarrow{!\,C'} C \curlyvee C'} \qquad \frac{C' \preceq C \quad \mathsf{prin}(C') \in \mathcal{A}}{C \xrightarrow{?\,C'} C} \qquad \frac{\mathsf{prin}(C') \notin \mathcal{A}}{C \xrightarrow{?\,C'} C \curlyvee C'}$$

The label $!\,C'$ records that the capability C' is exported to the environment: the outcome of the transition $C \curlyvee C'$ is an updated record of the most permissive exported capability. The label $?\,C'$ records that the capability C' is imported from the environment. There are two import rules, depending on the owner of C'. If the owner is in $\mathcal{A}$, then the capability refers to a location which is part of the system, so the environment can send back at most capabilities that can be derived from those exported by the system, hence the $C' \preceq C$ condition. On the contrary, if the owner is not in $\mathcal{A}$, the environment can send any capability, provided that the capability is compatible with the partial knowledge that the system already has, i.e. that $C \curlyvee C'$ exists.

Labeled transitions on systems The labeled semantics for systems is adapted from the one for the applied pi calculus. We point out the novelties, and refer to the companion paper for the full semantics.

The labeled semantics has silent steps for all system reductions, including the location-specific reductions described in Section 2. The axioms for input and output are recalled below. (We refer to [1] for a discussion of admissible output values when the equational theory includes cryptographic primitives.)

$$a[c!\langle M\rangle.P] \xrightarrow{c\,!\,M} a[P] \qquad a[c?(x).P] \xrightarrow{c?M} a[P\{^{M^\sharp}/_x\}]$$

When a capability is received, the rule substitutes in a capability value $M^\sharp$ obtained from the capability label M by erasing information used only to update the cell state.

The context rules below ensure that the communication of capabilities is reflected in the state of the cells of the system; the condition $l.C$ in M checks whether the cell $l.C$ occurs in the transmitted capability (possibly within another capability).

$$\frac{A \xrightarrow{c\,!\,M} A' \quad C_0 \xrightarrow{!\,C} C_1 \quad l.C \;\mathsf{in}\; M}{l\,.\,C_0 \mid A \xrightarrow{c\,!\,M} l\,.\,C_1 \mid A'} \qquad \frac{A \xrightarrow{c?M} A' \quad C_0 \xrightarrow{?\,C} C_1 \quad l.C \;\mathsf{in}\; M}{l\,.\,C_0 \mid A \xrightarrow{c?M} l\,.\,C_1 \mid A'}$$

$$\frac{A \xrightarrow{\alpha} A' \quad l.C \;\mathsf{not\;in}\; \alpha}{l\,.\,C_0 \mid A \xrightarrow{\alpha} l\,.\,C_0 \mid A'}$$

We equate $l\,.\,\perp \mid A$ to A, so that the input rule covers the case of an input carrying fresh, unknown locations from the environment. (The resulting configuration must be well-formed, which excludes the introduction of a fresh location state for l if one already exists in the system.) We impose the following well-formedness conditions on labels: (1) in every label, a location name occurs at most in a single, well-formed capability, plus possibly in the label restriction—this excludes e.g. pairs of simultaneous, incompatible commitments; and (2) the target term H, the principal in uncommitted capabilities, and the value in committed capabilities, appear iff the transition is an input and the capability is owned by $e \notin \mathcal{A}$.

Example of transitions in the source language Consider the third round of the game of Section 3, with two honest players a_1 and a_2 and an external, untrusted principal $e_0 \notin \mathcal{A}$ running the server. A simplified configuration of this system can be written

$$A' = l \,.\, \mathbf{Idu}\,(e_0\, H) \;\mid\; a_1[c_1?(x_1).P_1] \;\mid\; a_2[c_2?(x_2).P_2]$$

where l is the uncommitted cell pre-allocated by e_0 to store the winning move. (Here $H = \mathsf{h}(e_0 + \mathsf{h}(s))$ for some secret s created by e_0.) We have possible input transitions on channels c_1 and c_2, to notify the winning move to each of the players. The first transition may be:

$$A' \xrightarrow{c_1?l\,.\,\mathbf{Rd}\,(e_0\; s\; \mathbf{11})} l\,.\,\mathbf{Rd}\,(e_0\; s\; \mathbf{11}) \;\mid\; a_1[P_1\{^{l\,.\,\mathbf{Rd}\,(e_0\;\; \mathbf{11})}/_{x_1}\}] \;\mid\; a_2[c_2?(x_2).P_2]$$

which triggers the final process P_1 with a read capability for l substituted for x_1, carrying the game result (here 11). At the same time, the state for l is updated by the third capability-transition rule, since $\mathbf{Idu}\,(e_0\; H) \curlyvee \mathbf{Rd}\,(e_0\; s\; \mathbf{11}) = \mathbf{Rd}\,(e_0\; s\; \mathbf{11})$. Conversely, for instance, transitions with a label that attributes l to a_1 instead of e_0 are disabled. At this stage, the configuration records the commitment on l, so the only subsequent input transition $A'' \xrightarrow{c_2?l\,.\,C'} A'''$ carrying a read capability C' for l must be such that $\mathbf{Rd}\,(e_0\; s\; \mathbf{11}) \preceq C'$ (by the third capability-transition rule), that is, $C' = \mathbf{Rd}\,(e_0\; s\; \mathbf{11})$. This guarantees that the second player gets exactly the same result as the first one.

Relating the reduction-based and labeled semantics for the source language The labeled semantics precisely characterizes the interactions between a system and an arbitrary environment. Given two systems A and E consisting of principals in $\mathcal{A}$ and $\mathcal{E}$, respectively, if $E \mid A \longrightarrow^* S$ then there exist two such systems A' and E' and transitions $A \xrightarrow{\phi} A'$ such that $S \equiv \nu\mathcal{N}.(E' \mid A')$, where $\mathcal{N}$ is the set of names exported in the labels of ϕ. Conversely, for all systems A and transitions $A \xrightarrow{\phi} A'$, there exists a system E' and reductions $E \mid A \longrightarrow^* \nu\mathcal{N}.(E' \mid A')$.

Translation of extended location states and capabilities We extend the translation of Section 4 to cover all configurations reachable by transitions from initial configurations. This extended translation is inductively defined for all well-formed configurations in normal form, using the clauses of Section 4 plus the rules below for location states and capabilities.

We extensively rely on active substitutions [1] with the following naming conventions: for a location l, c_l denotes the local channel that contains the state of the location, s_l the secret seed, v_l the hidden value, and w_l the seal. We define two extended processes that compute and log identifiers, commitment values, and seals for a location owned by a given principal p using active substitutions.

$$\varphi(M_1, M_2)_p = \{\, \mathsf{h}(p + M_1) \,/\, l \,\} \mid \varsigma(M_1, M_2)_p$$
$$\varsigma(M_1, M_2)_p = \{\, M_1 + M_2 \,/\, v_l \,\} \mid \{\, \mathsf{sign}(v_l\,,\, \mathsf{sk}(m_p)) \,/\, w_l \,\} \mid \mathsf{repl}\; \mathit{log}!\langle \mathsf{idc}(p\,,\, v_l\,,\, w_l)\rangle$$

We first translate locations owned by honest principals $a \in \mathcal{A}$. The translation implements these locations by sending the location state on the local channel c_l, activating the relevant substitutions, creating a fresh secret and, for committed locations only, running a replicated log entry:

$$[\![l\,.\,0\,(a\,)]\!] = [\![l\,.\,\mathbf{Idu}\,(a\,)]\!] = c_l!\langle \mathsf{None}\rangle \mid \{\, \mathsf{h}(a + \mathsf{h}(s_l)) \,/\, l \,\} \mid \nu\, s\,.\,\{\, s \,/\, s_l \,\}$$
$$[\![l\,.\,0\,(a\;\; V)]\!] = [\![l\,.\,\mathbf{Idc}\,(a\;\; V)]\!] = [\![l\,.\,\mathbf{Rd}\,(a\;\; V)]\!] = \varphi(\mathsf{h}(s_l), \mathsf{h}(s_l + [\![V]\!]))_a \mid \nu\, s\,.\,\{\, s \,/\, s_l \,\}$$

We also translate locations owned by principals $e \notin \mathcal{A}$ whose capabilities have been previously received by some principals in $\mathcal{A}$. The translation records partial knowledge of these locations, in the form of active substitutions plus, for committed locations only, a replicated log entry. The form of the terms in these substitutions reflect the test that processes in $\mathcal{A}$ have successfully performed before accepting these values, e.g. that the seal is well-formed signature from e.

$$\begin{aligned}
&[\![l \,.\, \mathbf{Idu}\,(e\, H)]\!] = \{\, H \,/\, l \,\} \\
&[\![l \,.\, \mathbf{Idc}\,(e\,(\, M' \,+\, M'' \,)\; V)]\!] = \varphi(M', M'')_e \\
&[\![l \,.\, \mathbf{Rd}\,(e\, M\;\; V)]\!] = \{\, M \,/\, s_l \,\} \mid \varphi(\mathsf{h}(M), \mathsf{h}(M + [\![V]\!]))_e
\end{aligned}$$

In a well-formed system, there is a location state for every capability that occurs in the system. Accordingly, the translation of capabilities relies on the active substitutions introduced by the translation of location states, as follows:

$$[\![l \,.\, \mathbf{Idu}]\!] = \mathsf{idu}(l) \quad [\![l \,.\, \mathbf{Idc}\,(p\,)]\!] = \mathsf{idc}(p\,,\; v_l\,,\; w_l) \quad [\![l \,.\, \mathbf{Rd}\,(p\;\; V)]\!] = \mathsf{rd}(p\,,\; s_l\,,\; [\![V]\!]\,,\; w_l)$$

The compilation of each location state $l\,.\,C$ introduces name c_l and variables s_l, v_l, w_l, l whose visibility from the environment depend on the exported capability recorded in C. Thus, our translation finally introduces the following top-level restrictions: for every location, if no capability have been exported, all these names and variables are restricted; if C has tag **Idu**, the identifier l is unrestricted. for locations owned by principals in $\mathcal{A}$; if C has tag **Idc**, the variables w_l and v_l are also unrestricted; if C has tag **Rd**, only the channel c_l is restricted.

Example of transitions in the target language Let us consider how our translation operates on the following transition, which represents player a_1 receiving the result of the game from server e_0 (with $H = h(e_0 + h(s))$).

$$l \,.\, \mathbf{Idu}\,(e_0\; H) \;\mid\; a_1[c_1?(x).P_1] \xrightarrow{c_1 ? l \,.\, \mathbf{Rd}\,(e_0\; s\; \mathbf{11})} l \,.\, \mathbf{Rd}\,(e_0\; s\; \mathbf{11}) \;\mid\; a_1[P_1\{^{l \,.\, \mathbf{Rd}\,(e_0\;\; \mathbf{11})}/_{x}\}]$$

The translated system $\{\, H \,/\, l \,\} \mid [\![a_1[c_1?(x).P_1]]\!]$ simulates the source transition by an input with label $c_1\,?\,(\,\mathsf{rd}(e_0\,,\; s\,,\; \mathbf{11}\,,\; \mathsf{sign}(\mathsf{h}(s) + \mathsf{h}(s + \mathbf{11})\,,\; \mathsf{sk}(m_{e_0})))\,)$, followed by a series of reductions through the code of parse, including dynamic checks on $\mathsf{is\_rd}$ and $\mathsf{check\_idc}$. In 6 silent steps (including 3 steps for recursive processing of value 11), this yields the process

$$\begin{aligned}
&\{\, H \,/\, l \,\} \mid [\![a_1[P_1]]\!]\{^{\mathsf{rd}(e_0\,,\; s\,,\; \mathbf{11}\,,\; \mathsf{sign}(\mathsf{h}(s)+\mathsf{h}(s+\mathbf{11})\,,\; \mathsf{sk}(m_{e_0})))}/_{x}\} \\
&\quad \mid \mathsf{repl}\; log!\langle \mathsf{get\_idc}(x) \rangle \mid \nu\, r\,.\,(\mathsf{repl}\; r?(\_).c_1?(x).\mathsf{parse}\; x\; [\![P]\!]_a) \mid R \mid E.
\end{aligned}$$

After applying structural equivalence with active substitutions and eliminating the dead loop on channel r, we obtain a system

$$\nu\, s_l\,.\,\nu\, v_l\,.\,\nu\, w_l\,.\,(\{\, s \,/\, s_l \,\} \mid \varphi(\mathsf{h}(s_l), \mathsf{h}(s_l + \mathbf{11}))_{e_0} \mid [\![a_1[P_1]]\!]\{^{\mathsf{rd}(e_0\,,\; s_l\,,\; \mathbf{11}\,,\; w_l)}/_{x}\}) \mid R \mid E$$

that matches the translation of the resulting source system above.

6 Correctness results

The first proposition states that the behavior of every source system can be simulated by its translation. That is, for any labeled trace of all source systems, there is a labeled trace of the process resulting from its translation. This shows the correctness (or functional adequacy) of our translation. We let $\xrightarrow{\phi}$ (resp. $\xrightarrow{\psi}$) range over series of transitions in the labeled semantics of the source (resp. target) language.

Theorem 1 (Functional adequacy). *Let A be a well-formed source system.*
For all series of transitions $A \xrightarrow{\phi}^{*} A'$, there exist transitions $[\![A]\!] \xrightarrow{\psi}^{*} [\![A']\!]$.

The proof of the theorem is by induction on a series of source transitions between systems in normal forms. For each source transition, we exhibit target transitions that commute with the translation.

The "upwards" direction is more challenging: the trace produced by the translation of a source process A can be related to a trace produced by A *unless* its translation emits the name of a cheating principal on the special channel bad. This property uniformly guarantees the security of the translation of all systems with respect to the source semantics, provided that a proof that a principal cheated is a reasonable exceptional outcome for the other principals.

We let $S \longrightarrow_D^* S'$ denote that a target system S goes to S' with a (possibly empty) series of silent deterministic transitions, and let $S \Downarrow M$ abbreviate $S \longrightarrow_D^* \xrightarrow{bad!M} S'$ for some S'; we then say that M is blamed.

Theorem 2 (Security). *For all transitions $[\![A]\!] \xrightarrow{\psi}^{*} S$ starting from a well-formed source system A, we have*

1. *either there are source transitions $A \xrightarrow{\phi}^{*} A'$ leading to a well-formed source system A' such that $S \longrightarrow_D^* [\![A']\!]$; or $S \Downarrow e$ for some $e \notin \mathcal{A}$;*
2. *if $S \Downarrow M$, then $M \notin \mathcal{A}$.*

The proof is by induction on the series of transitions in the target language that do not trigger a blame. The first part of the theorem states that either the source semantics is respected, or the implementation at least provides the honest participants with the name of one dishonest principal to blame. Said otherwise, its statement excludes the possibility of cheating without eventual detection. The second part of the theorem expresses that honest participants are never blamed (even in the case some dishonest participants cheat), a necessary property for any optimistic implementation.

The form of our theorem differs from security properties for other programming abstractions (e.g. [7, 2]), where any run or labeled trace of the cryptographic implementation of a source program is related to a run or labeled trace of the program on the source level. Reflecting a more flexible approach to security, it enables bad runs as long as malicious principals are reliably detected and blamed.

We illustrate how the Resolution protocol and the verifications made by the translation of receive suffice to detect write-after-commit attacks. Consider the online game example and suppose that $a_1, a_2 \in \mathcal{A}$ and $e_0 \notin \mathcal{A}$, that is, the server implementation is malicious. In particular, the server implementation may commit location l twice, to convince a_1 that he is the winner with his bid 11 and a_2 that he is the winner with his bid 8. The system composed by the translation of the two clients $[\![A_1 \mid A_2]\!]$ generates a trace

$$[\![A_1 \mid A_2]\!] \to \cdots \to [\![A']\!] \xrightarrow{c_1\,?\,(\,\mathsf{rd}(e_0\,,\,s\,,\,\mathbf{11}\,,\,w)\,)} \xrightarrow{c_2\,?\,(\,\mathsf{rd}(e_0\,,\,s\,,\,8\,,\,w')\,)} S$$

where the seals w and w' sign commitments of l to 11 and 8, respectively.

For the first input transition, there exists a matching source transition, with a resulting source system A'' that includes the location state $l\,.\,\mathbf{Rd}\,(e_0\;s\;\mathbf{11})$. Moreover, the translation of A'' emits the corresponding idc on log.

For the second input transition, however, there is no matching source transition. This would require a capability transition from $\mathbf{Rd}\,(e_0\;s\;\mathbf{11})$ to $\mathbf{Rd}\,(e_0\;s\;8)$, which is excluded by our definition of the $\preceq$ preorder. Instead, the resulting system sends a second $\mathbf{Idc}$ on log. As soon as the Resolution process reads both commitments, it detects that they are inconsistent, and blames e_0 on bad.

7 Prototype implementation

We have implemented committable cells as a library for OCaml [9]. We have also coded a series of examples, including simple online games and sealed-bid auctions.

The library provides abstract datatypes and access functions that closely follow those of our source language. Its implementation relies on standard cryptographic libraries and on a public-key (X.509) infrastructure for processing capabilities; it uses pseudo-random number generation for creating fresh secret seeds. Programs that use our library may communicate with one another using OCaml marshalling and network socket interfaces—cryptographic validation of received capabilities then occurs during unmarshalling.

The main difference between the implementation and its formal semantics is the handling of resolution. We refine the idealized resolution mechanism of Section 4 as follows: instead of relying on a central resolution process, our implementation keeps track of all principals and cells involved in a run of the system, and eventually implements the exchange and local resolution for all shared commitments.

8 Conclusions and future work

We presented a simple language for specifying systems based on optimistic commitments, and we compiled this language into a realistic concurrent framework modeled in the applied pi calculus. We established two security properties relating the labeled traces of a source semantics with commitment primitives to those of their implementation, with a target semantics that uses only ordinary communications and cryptographic functions. We only consider authenticity for now, but we believe it would also be possible to guarantee some properties of formal secrecy.

Although committable cells provide a reasonably useful (and formally challenging) block for building protocols, we focused on one particular usage of secure logs, rather than proposing a comprehensive language design for optimistic protocols. Our formal approach could be extended to other, more involved datatypes—as long as we can represent their live cycles using a preorder on exported capabilities, as detailed in Section 5. It would be interesting, for instance, to design compilers for such datatypes with incremental commitment properties.

More generally, audit logs constitute an important tool for designing protocols and applications. Although their efficient implementation has been thoroughly studied, we believe ours is the first work to address their reliable, principled usage from a programming-language viewpoint.

References

1. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, 2001.
2. M. Abadi, C. Fournet, and G. Gonthier. Secure Implementation of Channel Abstractions. *Information and Computation*, 174(1):37–83, 2002.
3. J. Castellà-Roca, J. Domingo-Ferrer, A. Riera, and J. Borrell. Practical mental poker without a ttp based on homomorphic encryption. In *Progress in Cryptology-Indocrypt*, LNCS, 2003.
4. J.G. Cederquist, R. Corin, M.A.C. Dekker, S. Etalle, J.I. den Hartog, and G. Lenzini. Audit-based compliance control. *Int'l Journal of Information Security*, 6(2):133–151, 2007.
5. D. Chaum. Secret-ballot receipts : True voter-verifiable elections. *IEEE Security and Privacy*, 2(1):38–47, 2004.
6. D. Chaum, P.Y.A. Ryan, and S. Schneider. A practical, voter-verifiable election scheme. Technical Report CS-TR-880, 2004.
7. R. Corin, P.-M. Denielou, C. Fournet, K. Bhargavan, and J. Leifer. Secure implementations for typed session abstractions. In *IEEE Computer Security Foundations Symposium*, 2007.
8. R. Corin, D. Galindo, and J.H. Hoepman. Securing data accountability in decentralized systems. In *1st Int'l Workshop on Information Security (IS'06)*, LNCS, 2006.
9. X. Leroy et al. Objective caml. http://caml.inria.fr.
10. S. Etalle and W.H. Winsborough. A posteriori compliance control. In *12th ACM Symposium on Access Control Models and Technologies*, 2007.
11. ISO/IEC. Common criteria for information technology security evaluation. http://www.commoncriteriaportal.org/public/expert/index.php?menu=3, 2004.
12. S. Jha, S. Katzenbeisser, C. Schallhart, H. Veith, and S. Chenney. Enforcing semantic integrity on untrusted clients in networked virtual environments. In *IEEE Symposium on Security and Privacy*, 2007.
13. S. Kremer and M. D. Ryan. Analysing the vulnerability of protocols to produce known-pair and chosen-text attacks. In *2nd Int'l Workshop on Security Issues in Coordination Models, Languages and Systems (SecCo'04)*, ENTCS, 2005.
14. NIST Special Publications. Generally accepted principles and practices for securing information technology systems, 1996.
15. B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 2(2):159–176, 1999.
16. A. Shamir, R.Rivest, and L. Adleman. Mental poker. *Mathematical Gardener*, 1981.
17. B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters. Building an encrypted and searchable audit log. In *Network and Distributed System Security Symposium (NDSS)*, 2004.
18. W. Xu, D. Chadwick, and S. Otenko. A PKI Based Secure Audit Web Server. In *IASTED Communications, Network and Information and CNIS*, 2005.
19. L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *IEEE Symposium on Security and Privacy*, 2003.