Equivalence Properties by Typing in Cryptographic Branching Protocols

Véronique Cortier[1], Niklas Grimm[2], Joseph Lallemand[1(✉)], and Matteo Maffei[2]

[1] Université de Lorraine, CNRS, Inria, LORIA, Vandœuvre-lès-Nancy, France
joseph.lallemand@loria.fr
[2] TU Wien, Vienna, Austria

Abstract. Recently, many tools have been proposed for automatically analysing, in symbolic models, equivalence of security protocols. Equivalence is a property needed to state privacy properties or game-based properties like strong secrecy. Tools for a bounded number of sessions can decide equivalence but typically suffer from efficiency issues. Tools for an unbounded number of sessions like Tamarin or ProVerif prove a stronger notion of equivalence (diff-equivalence) that does not properly handle protocols with else branches.

Building upon a recent approach, we propose a type system for reasoning about branching protocols and dynamic keys. We prove our type system to entail equivalence, for all the standard primitives. Our type system has been implemented and shows a significant speedup compared to the tools for a bounded number of sessions, and compares similarly to ProVerif for an unbounded number of sessions. Moreover, we can also prove security of protocols that require a mix of bounded and unbounded number of sessions, which ProVerif cannot properly handle.

1 Introduction

Formal methods provide a rigorous and convenient framework for analysing security protocols. In particular, mature push-button analysis tools have emerged and have been successfully applied to many protocols from the literature in the context of *trace properties* such as authentication or confidentiality. These tools employ a variety of analysis techniques, such as model checking (e.g., Avispa [6] and Scyther [31]), Horn clause resolution (e.g., ProVerif [13]), term rewriting (e.g., Scyther [31] and Tamarin [38]), and type systems [7,12,16–21,34,36,37].

In the recent years, attention has been given also to equivalence properties, which are crucial to model privacy properties such as vote privacy [8,33], unlikability [5], or anonymity [9]. For example, consider an authentication protocol P_{pass} embedded in a biometric passport. P_{pass} preserves anonymity of passport holders if an attacker cannot distinguish an execution with Alice from an execution with Bob. This can be expressed by the equivalence $P_{pass}(Alice) \approx_t P_{pass}(Bob)$. Equivalence is also used to express properties closer to cryptographic games like strong secrecy.

L. Bauer and R. Küsters (Eds.): POST 2018, LNCS 10804, pp. 160–187, 2018.
https://doi.org/10.1007/978-3-319-89722-6_7

Two main classes of tools have been developed for equivalence. First, in the case of an unbounded number of sessions (when the protocol is executed arbitrarily many times), equivalence is undecidable. Instead, the tools ProVerif [13,15] and Tamarin [11,38] try to prove a stronger property, namely diff-equivalence, that may be too strong e.g. in the context of voting. Tamarin covers a larger class of protocols but may require some guidance from the user. Maude-NPA [35,40] also proves diff-equivalence but may have non-termination issues. Another class of tools aim at deciding equivalence, for bounded number of sessions. This is the case in particular of SPEC [32], APTE [23], Akiss [22], and SatEquiv [26]. SPEC, APTE, and Akiss suffer from efficiency issues and can typically not handle more than 3–4 sessions. SatEquiv is much more efficient but is limited to symmetric encryption and requires protocols to be well-typed, which often assumes some additional tagging of the protocol.

Our Contribution. Following the approach of [28], we propose a novel technique for proving equivalence properties for a bounded number of sessions as well as an unbounded number of sessions (or a mix of both), based on typing. [28] proposes a first type system that entails trace equivalence $P \approx_t Q$, provided protocols use fixed (long-term) keys, identical in P and Q. In this paper, we target a larger class of protocols, that includes in particular key-exchange protocols and protocols whose security relies on branching on the secret. This is the case e.g. of the private authentication protocol [3], where agent B returns a true answer to A, encrypted with A's public key if A is one of his friends, and sends a decoy message (encrypted with a dummy key) otherwise.

We devise a new type system for reasoning about keys. In particular, we introduce bikeys to cover behaviours where keys in P differ from the keys in Q. We design new typing rules to reason about protocols that may branch differently (in P and Q), depending on the input. Following the approach of [28], our type system collects sent messages into constraints that are required to be consistent. Intuitively, the type system guarantees that any execution of P can be matched by an execution of Q, while consistency imposes that the resulting sequences of messages are indistinguishable for an attacker. We had to entirely revisit the approach of [28] and prove a finer invariant in order to cope with the case where keys are used as variables. Specifically, most of the rules for encryption, signature, and decryption had to be adapted to accommodate the flexible usage of keys. For messages, we had to modify the rules for keys and encryption, in order to encrypt messages with keys of different type (bi-key type), instead of only fixed keys. We show that our type system entails equivalence for the standard notion of trace equivalence [24] and we devise a procedure for proving consistency. This yields an efficient approach for *proving* equivalence of protocols for a bounded and an unbounded number of sessions (or a combination of both).

We implemented a prototype of our type-checker that we evaluate on a set of examples, that includes private authentication, the BAC protocol (of the biometric passport), as well as Helios together with the setup phase. Our tool requires a light type annotation that specifies which keys and names are likely to be secret or public and the form of the messages encrypted by a given key. This can be

easily inferred from the structure of the protocol. Our type-checker outperforms even the most efficient existing tools for a bounded number of sessions by two (for examples with few processes) to three (for examples with more processes) orders of magnitude. Note however that these tools *decide* equivalence while our type system is incomplete. In the case of an unbounded number of sessions, on our examples, the performance is comparable to ProVerif, one of the most popular tools. We consider in particular vote privacy in the Helios protocol, in the case of a dishonest ballot board, with no revote (as the protocol is insecure otherwise). ProVerif fails to handle this case as it cannot (faithfully) consider a mix of bounded and unbounded number of sessions. Compared to [28], our analysis includes the setup phase (where voters receive the election key), which could not be considered before.

The technical details and proofs omitted due to space constraints are available in the companion technical report [29].

2 High-Level Description

2.1 Background

Trace equivalence of two processes is a property that guarantees that an attacker observing the execution of either of the two processes cannot decide which one it is. Previous work [28] has shown how trace equivalence can be proved statically using a type system combined with a constraint checking procedure. The type system consists of typing rules of the form $\Gamma \vdash P \sim Q \rightarrow C$, meaning that in an environment Γ two processes P and Q are equivalent if the produced set of constraints C, encoding the attacker observables, is consistent.

The typing environment Γ is a mapping from nonces, keys, and variables to types. Nonces are assigned security labels with a confidentiality and an integrity component, e.g. `HL` for high confidentiality and low integrity. Key types are of the form $\mathrm{key}^l(T)$ where l is the security label of the key and T is the type of the payload. Key types are crucial to convey typing information from one process to another one. Normally, we cannot make any assumptions about values received from the network – they might possibly originate from the attacker. If we however successfully decrypt a message using a secret symmetric key, we know that the result is of the key's payload type. This is enforced on the sender side, whenever outputting an encryption.

A core assumption of virtually any efficient static analysis for equivalence is uniform execution, meaning that the two processes of interest always take the same branch in a branching statement. For instance, this means that all decryptions must always succeed or fail equally in the two processes. For this reason, previous work introduced a restriction to allow only encryption and decryption with keys whose equality could be statically proved.

2.2 Limitation

There are however protocols that require non-uniform execution for a proof of trace equivalence, e.g., the private authentication protocol [3]. The protocol aims

$$
\begin{array}{rll}
\Gamma(k_b, k_b) = \text{key}^{\texttt{HH}}(\texttt{HL} * \texttt{LL}) & \text{initial message uses same key on both sides} \\
\Gamma(k_a, k) = \text{key}^{\texttt{HH}}(\texttt{HL}) & \text{authentication succeeded on the left, failed on the right} \\
\Gamma(k, k_c) = \text{key}^{\texttt{HH}}(\texttt{HL}) & \text{authentication succeeded on the right, failed on the left} \\
\Gamma(k_a, k_c) = \text{key}^{\texttt{HH}}(\texttt{HL}) & \text{authentication succeeded on both sides} \\
\Gamma(k, k) = \text{key}^{\texttt{HH}}(\texttt{HL}) & \text{authentication failed on both sides}
\end{array}
$$

Fig. 1. Key types for the private authentication protocol

at authenticating B to A, anonymously w.r.t. other agents. More specifically, agent B may refuse to communicate with agent A but a third agent D should not learn whether B declines communication with A or not. The protocol can be informally described as follows, where $\texttt{pk}(k)$ denotes the public key associated to key k, and $\texttt{aenc}(M, \texttt{pk}(k))$ denotes the asymmetric encryption of message M with this public key.

$$
\begin{array}{ll}
A \to B: & \texttt{aenc}(\langle N_a, \texttt{pk}(k_a)\rangle, \texttt{pk}(k_b)) \\
B \to A: & \begin{cases} \texttt{aenc}(\langle N_a, \langle N_b, \texttt{pk}(k_b)\rangle\rangle, \texttt{pk}(k_a)) & \text{if } B \text{ accepts } A\text{'s request} \\ \texttt{aenc}(N_b, \texttt{pk}(k)) & \text{if } B \text{ declines } A\text{'s request} \end{cases}
\end{array}
$$

If B declines to communicate with A, he sends a decoy message $\texttt{aenc}(N_b, \texttt{pk}(k))$ where $\texttt{pk}(k)$ is a decoy key (no one knows the private key k).

2.3 Encrypting with Different Keys

Let $P_a(k_a, \texttt{pk}(k_b))$ model agent A willing to talk with B, and $P_b(k_b, \texttt{pk}(k_a))$ model agent B willing to talk with A (and declining requests from other agents). We model the protocol as:

$$
\begin{array}{l}
P_a(k_a, pk_b) = \texttt{new}\ N_a.\texttt{out}(\texttt{aenc}(\langle N_a, \texttt{pk}(k_a)\rangle, pk_b)).\ \texttt{in}(z) \\
P_b(k_b, pk_a) = \texttt{new}\ N_b.\ \texttt{in}(x). \\
\qquad \texttt{let}\ y = \texttt{adec}(x, k_b)\ \texttt{in}\ \ \texttt{let}\ y_1 = \pi_1(y)\ \texttt{in}\ \ \texttt{let}\ y_2 = \pi_2(y)\ \texttt{in} \\
\qquad\quad \texttt{if}\ y_2 = pk_a\ \texttt{then} \\
\qquad\qquad \texttt{out}(\texttt{aenc}(\langle y_1, \langle N_b, \texttt{pk}(k_b)\rangle\rangle, pk_a)) \\
\qquad\quad \texttt{else}\ \texttt{out}(\texttt{aenc}(N_b, \texttt{pk}(k)))
\end{array}
$$

where $\texttt{adec}(M, k)$ denotes asymmetric decryption of message M with private key k. We model anonymity as the following equivalence, intuitively stating that an attacker should not be able to tell whether B accepts requests from the agent A or C:

$$
P_a(k_a, \texttt{pk}(k_b)) \ \mid\ P_b(k_b, \texttt{pk}(k_a)) \ \approx_t\ P_a(k_a, \texttt{pk}(k_b)) \ \mid\ P_b(k_b, \texttt{pk}(k_c))
$$

We now show how we can type the protocol in order to show trace equivalence. The initiator P_a is trivially executing uniformly, since it does not contain any branching operations. We hence focus on typing the responder P_b.

The beginning of the responder protocol can be typed using standard techniques. Then however, we perform the test $y_2 = \texttt{pk}(k_a)$ on the left side and

$y_2 = \mathtt{pk}(k_c)$ on the right side. Since we cannot statically determine the result of the two equality checks – and thus guarantee uniform execution – we have to typecheck the four possible combinations of `then` and `else` branches. This means we have to typecheck outputs of encryptions that use different keys on the left and the right side.

To deal with this we do not assign types to single keys, but rather to pairs of keys (k, k') – which we call *bikeys* – where k is the key used in the left process and k' is the key used in the right process. The key types used for typing are presented in Fig. 1.

As an example, we consider the combination of the `then` branch on the left with the `else` branch on the right. This combination occurs when A is successfully authenticated on the left side, while being rejected on the right side. We then have to typecheck B's positive answer together with the decoy message: $\Gamma \vdash \mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k)) : \mathtt{LL}$. For this we need the type for the bikey (k_a, k).

2.4 Decrypting Non-uniformly

When decrypting a ciphertext that was potentially generated using two different keys on the left and the right side, we have to take all possibilities into account. Consider the following extension of the process P_a where agent A decrypts B's message.

$$\begin{aligned} P_a(k_a, pk_b) = \mathtt{new}\ N_a.\mathtt{out}(\mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, pk_b)).\ \mathtt{in}(z).\\ \mathtt{let}\ z' = \mathtt{adec}(z, k_a)\ \mathtt{in}\ \mathtt{out}(1)\\ \mathtt{else}\ \mathtt{out}(0) \end{aligned}$$

In the decryption, there are the following possible cases:

- The message is a valid encryption supplied by the attacker (using the public key $\mathtt{pk}(k_a)$), so we check the `then` branch on both sides with $\Gamma(z') = \mathtt{LL}$.
- The message is not a valid encryption supplied by the attacker so we check the `else` branch on both sides.
- The message is a valid response from B. The keys used on the left and the right are then one of the four possible combinations $(k_a, k), (k_a, k_c), (k, k_c)$ and (k, k).
  - In the first two cases the decryption will succeed on the left and fail on the right. We hence check the `then` branch on the left with $\Gamma(z') = \mathtt{HL}$ with the `else` branch on the right. If the type $\Gamma(k_a, k)$ were different from $\Gamma(k_a, k_c)$, we would check this combination twice, using the two different payload types.
  - In the remaining two cases the decryption will fail on both sides. We hence would have to check the two `else` branches (which however we already did).

While checking the `then` branch together with the `else` branch, we have to check $\Gamma \vdash 1 \sim 0 : \mathtt{LL}$, which rightly fails, as the protocol does not guarantee trace equivalence.

3 Model

In symbolic models, security protocols are typically modelled as processes of a process algebra, such as the applied pi-calculus [2]. We present here a calculus used in [28] and inspired from the calculus underlying the ProVerif tool [14]. This section is mostly an excerpt of [28], recalled here for the sake of completeness, and illustrated with the private authentication protocol.

3.1 Terms

Messages are modelled as terms. We assume an infinite set of names $\mathcal{N}$ for nonces, further partitioned into the set $\mathcal{FN}$ of free nonces (created by the attacker) and the set $\mathcal{BN}$ of bound nonces (created by the protocol parties), an infinite set of names $\mathcal{K}$ for keys similarly split into $\mathcal{FK}$ and $\mathcal{BK}$, and an infinite set of variables $\mathcal{V}$. Cryptographic primitives are modelled through a *signature* $\mathcal{F}$, that is, a set of function symbols, given with their arity (*i.e.* the number of arguments). Here, we consider the following signature:

$$\mathcal{F}_c = \{\mathtt{pk}, \mathtt{vk}, \mathtt{enc}, \mathtt{aenc}, \mathtt{sign}, \langle\cdot,\cdot\rangle, \mathtt{h}\}$$

that models respectively public and verification key, symmetric and asymmetric encryption, concatenation and hash. The companion primitives (symmetric and asymmetric decryption, signature check, and projections) are represented by the following signature:

$$\mathcal{F}_d = \{\mathtt{dec}, \mathtt{adec}, \mathtt{checksign}, \pi_1, \pi_2\}$$

We also consider a set $\mathcal{C}$ of (public) constants (used as agent names for instance). Given a signature $\mathcal{F}$, a set of names $\mathcal{N}$, and a set of variables $\mathcal{V}$, the set of *terms* $\mathcal{T}(\mathcal{F}, \mathcal{V}, \mathcal{N})$ is the set inductively defined by applying functions to variables in $\mathcal{V}$ and names in $\mathcal{N}$. We denote by $\mathrm{names}(t)$ (resp. $\mathrm{vars}(t)$) the set of names (resp. variables) occurring in t. A term is *ground* if it does not contain variables.

We consider the set $\mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_d \cup \mathcal{C}, \mathcal{V}, \mathcal{N} \cup \mathcal{K})$ of *cryptographic terms*, simply called *terms*. *Messages* are terms with constructors from $\mathcal{T}(\mathcal{F}_c \cup \mathcal{C}, \mathcal{V}, \mathcal{N} \cup \mathcal{K})$. We assume the set of variables to be split into two subsets $\mathcal{V} = \mathcal{X} \uplus \mathcal{AX}$ where $\mathcal{X}$ are variables used in processes while $\mathcal{AX}$ are variables used to store messages. An *attacker term* is a term from $\mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_d \cup \mathcal{C}, \mathcal{AX}, \mathcal{FN} \cup \mathcal{FK})$. In particular, an attacker term cannot use nonces and keys created by the protocol's parties.

A *substitution* $\sigma = \{M_1/x_1, \ldots, M_k/x_k\}$ is a mapping from variables $x_1, \ldots, x_k \in \mathcal{V}$ to messages $M_1, \ldots, M_k$. We let $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_k\}$. We say that σ is ground if all messages $M_1, \ldots, M_k$ are ground. We let $\mathrm{names}(\sigma) = \bigcup_{1 \le i \le k} \mathrm{names}(M_i)$. The application of a substitution σ to a term t is denoted $t\sigma$ and is defined as usual.

The *evaluation* of a term t, denoted $t\downarrow$, corresponds to the bottom-up application of the cryptographic primitives and is recursively defined as follows.

$$\begin{array}{rll}
u\downarrow = u & & \text{if } u \in \mathcal{N} \cup \mathcal{V} \cup \mathcal{K} \cup \mathcal{C} \\
\mathtt{pk}(t)\downarrow = \mathtt{pk}(t\downarrow) & & \text{if } t\downarrow \in \mathcal{K} \\
\mathtt{vk}(t)\downarrow = \mathtt{vk}(t\downarrow) & & \text{if } t\downarrow \in \mathcal{K} \\
\mathtt{h}(t)\downarrow = \mathtt{h}(t\downarrow) & & \text{if } t\downarrow \neq \bot \\
\langle t_1, t_2\rangle\downarrow = \langle t_1\downarrow, t_2\downarrow\rangle & & \text{if } t_1\downarrow \neq \bot \text{ and } t_2\downarrow \neq \bot \\
\mathtt{enc}(t_1, t_2)\downarrow = \mathtt{enc}(t_1\downarrow, t_2\downarrow) & & \text{if } t_1\downarrow \neq \bot \text{ and } t_2\downarrow \in \mathcal{K} \\
\mathtt{sign}(t_1, t_2)\downarrow = \mathtt{sign}(t_1\downarrow, t_2\downarrow) & & \text{if } t_1\downarrow \neq \bot \text{ and } t_2\downarrow \in \mathcal{K} \\
\mathtt{aenc}(t_1, t_2)\downarrow = \mathtt{aenc}(t_1\downarrow, t_2\downarrow) & & \text{if } t_1\downarrow \neq \bot \text{ and } t_2\downarrow = \mathtt{pk}(k) \\
\multicolumn{3}{c}{\text{for some } k \in \mathcal{K}}
\end{array}$$

$$\begin{array}{rl}
\pi_1(t)\downarrow = t_1 & \text{if } t\downarrow = \langle t_1, t_2\rangle \\
\pi_2(t)\downarrow = t_2 & \text{if } t\downarrow = \langle t_1, t_2\rangle \\
\mathtt{dec}(t_1, t_2)\downarrow = t_3 & \text{if } t_1\downarrow = \mathtt{enc}(t_3, t_4) \text{ and } t_4 = t_2\downarrow \\
\mathtt{adec}(t_1, t_2)\downarrow = t_3 & \text{if } t_1\downarrow = \mathtt{aenc}(t_3, \mathtt{pk}(t_4)) \text{ and } t_4 = t_2\downarrow \\
\mathtt{checksign}(t_1, t_2)\downarrow = t_3 & \text{if } t_1\downarrow = \mathtt{sign}(t_3, t_4) \text{ and } t_2\downarrow = \mathtt{vk}(t_4) \\
t\downarrow = \bot & \text{otherwise}
\end{array}$$

Note that the evaluation of term t succeeds only if the underlying keys are atomic and always returns a message or $\bot$. For example we have $\pi_1(\langle a, b\rangle)\downarrow = a$, while $\mathtt{dec}(\mathtt{enc}(a, \langle b, b\rangle), \langle b, b\rangle)\downarrow = \bot$, because the key is non atomic. We write $t =_\downarrow t'$ if $t\downarrow = t'\downarrow$.

Destructors used in processes:

$$d ::= \mathtt{dec}(x, t) \mid \mathtt{adec}(x, t) \mid \mathtt{checksign}(x, t') \mid \pi_1(x) \mid \pi_2(x)$$

where $x \in \mathcal{X}$, $t \in \mathcal{K} \cup \mathcal{X}$, $t' \in \{\mathtt{vk}(k) | k \in \mathcal{K}\} \cup \mathcal{X}$.

Processes:

$$\begin{array}{rl}
P, Q ::= & \mathtt{0} \mid \mathtt{new}\ n.P \mid \mathtt{out}(M).P \mid \mathtt{in}(x).P \mid (P \mid Q) \mid !P \\
& \mid \mathtt{let}\ x = d\ \mathtt{in}\ P\ \mathtt{else}\ Q \mid \mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q
\end{array}$$

where $n \in \mathcal{BN} \cup \mathcal{BK}$, $x \in \mathcal{X}$, and M, N are messages.

Fig. 2. Syntax for processes.

3.2 Processes

Security protocols describe how messages should be exchanged between participants. We model them through a process algebra, whose syntax is displayed in Fig. 2. We identify processes up to α-renaming, *i.e.*, avoiding substitution of bound names and variables, which are defined as usual. Furthermore, we assume that all bound names, keys, and variables in the process are distinct.

A *configuration* of the system is a tuple $(\mathcal{P}; \phi; \sigma)$ where:

- $\mathcal{P}$ is a multiset of processes that represents the current active processes;
- ϕ is a substitution with $\mathrm{dom}(\phi) \subseteq \mathcal{AX}$ and for any $x \in \mathrm{dom}(\phi)$, $\phi(x)$ (also denoted $x\phi$) is a message that only contains variables in $\mathrm{dom}(\sigma)$. ϕ represents the terms that have been sent;
- σ is a ground substitution.

The semantics of processes is given through a transition relation $\xrightarrow{\alpha}$, defined in Fig. 3 (τ denotes a silent action). The relation $\xrightarrow{w}_*$ is defined as the reflexive transitive closure of $\xrightarrow{\alpha}$, where w is the concatenation of all actions. We also write equality up to silent actions $=_\tau$.

Intuitively, process $\mathtt{new}\ n.P$ creates a fresh nonce or key, and behaves like P. Process $\mathtt{out}(M).P$ emits M and behaves like P, provided that the evaluation of M is successful. The corresponding message is stored in the frame ϕ, corresponding to the attacker knowledge. A process may input any message that an attacker can forge (rule In) from her knowledge ϕ, using a recipe R to compute a new message from ϕ. Note that all names are initially assumed to be secret. Process $P \mid Q$ corresponds to the parallel composition of P and Q. Process $\mathtt{let}\ x = d\ \mathtt{in}\ P\ \mathtt{else}\ Q$ behaves like P in which x is replaced by d if d can be successfully evaluated and behaves like Q otherwise. Process $\mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q$ behaves like P if M and N correspond to two equal messages and behaves like Q otherwise. The replicated process $!P$ behaves as an unbounded number of copies of P.

A *trace* of a process P is any possible sequence of transitions in the presence of an attacker that may read, forge, and send messages. Formally, the set of traces $\mathrm{trace}(P)$ is defined as follows.

$$\mathrm{trace}(P) = \{(w, \phi, \sigma) | (\{P\}; \emptyset; \emptyset) \xrightarrow{w}_* (\mathcal{P}; \phi; \sigma)\}$$

Example 1. Consider the private authentication protocol (PA) presented in Sect. 2. The process $P_b(k_b, \mathtt{pk}(k_a))$ corresponding to responder B answering a request from A has already been defined in Sect. 2.3. The process $P_a(k_a, \mathtt{pk}(k_b))$ corresponding A willing to talk to B is:

$$P_a(k_a, pk_b) = \mathtt{new}\ N_a.\mathtt{out}(\mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, pk_b)).\ \mathtt{in}(z)$$

Altogether, a session between A and B is represented by the process:

$$P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_a))$$

where $k_a, k_b \in \mathcal{BK}$, which models that the attacker initially does not know k_a, k_b.

$$(\{P_1 \mid P_2\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P_1, P_2\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{Par}$$

$$(\{\mathtt{0}\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\mathcal{P}; \phi; \sigma) \quad \textsc{Zero}$$

$$(\{\mathtt{new}\ n.P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{New}$$

$$(\{\mathtt{new}\ k.P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{NewKey}$$

$$(\{\mathtt{out}(t).P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\mathtt{new}\ ax_n.\mathtt{out}(ax_n)} (\{P\} \cup \mathcal{P}; \phi \cup \{t/ax_n\}; \sigma) \quad \textsc{Out}$$
if $t\sigma$ is a ground term, $(t\sigma) \downarrow \neq \bot$, $ax_n \in \mathcal{AX}$ and $n = |\phi| + 1$

$$(\{\mathtt{in}(x).P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\mathtt{in}(R)} (\{P\} \cup \mathcal{P}; \phi; \sigma \cup \{(R\phi\sigma) \downarrow /x\}) \quad \textsc{In}$$
if R is an attacker term such that $\mathrm{vars}(R) \subseteq \mathrm{dom}(\phi)$, and $(R\phi\sigma) \downarrow \neq \bot$

$$(\{\mathtt{let}\ x = d\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P\} \cup \mathcal{P}; \phi; \sigma \cup \{(d\sigma) \downarrow /x\}) \quad \textsc{Let-In}$$
if $d\sigma$ is ground and $(d\sigma) \downarrow \neq \bot$

$$(\{\mathtt{let}\ x = d\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{Q\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{Let-Else}$$
if $d\sigma$ is ground and $(d\sigma) \downarrow = \bot$, *i.e.* d fails

$$(\{\mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{If-Then}$$
if M, N are messages such that $M\sigma$, $N\sigma$ are ground, $(M\sigma) \downarrow \neq \bot$, $(N\sigma) \downarrow \neq \bot$, and $M\sigma = N\sigma$

$$(\{\mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{Q\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{If-Else}$$
if M, N are messages such that $M\sigma$, $N\sigma$ are ground and $(M\sigma) \downarrow = \bot$ or $(N\sigma) \downarrow = \bot$ or $M\sigma \neq N\sigma$

$$(\{!P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P, !P\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{Repl}$$

Fig. 3. Semantics

An example of a trace describing an "honest" execution, where the attacker does not interfere with the intended run of the protocol, can be written as (tr, ϕ) where

$$tr =_\tau \mathtt{new}\ x_1.\mathtt{out}(x_1).\mathtt{in}(x_1).\mathtt{new}\ x_2.\mathtt{out}(x_2).\mathtt{in}(x_2)$$

and

$$\phi = \{x_1 \mapsto \mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, \mathtt{pk}(k_b)), x_2 \mapsto \mathtt{aenc}(\langle N_a, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a))\}.$$

The trace tr describes A outputting the first message of the protocol, which is stored in $\phi(x_1)$. The attacker then simply forwards $\phi(x_1)$ to B. B then performs several silent actions (decrypting the message, comparing its content to $\mathtt{pk}(k_a)$), and outputs a response, which is stored in $\phi(x_2)$ and forwarded to A by the attacker.

$$\begin{array}{lcl} l & ::= & \mathtt{LL} \mid \mathtt{HL} \mid \mathtt{HH} \\ KT & ::= & \mathrm{key}^l(T) \mid \mathrm{eqkey}^l(T) \mid \mathrm{seskey}^{l,a}(T) \text{ with } a \in \{1, \infty\} \\ T & ::= & l \mid T * T \mid T \vee T \mid [\![\tau_n^{l,a}\ ;\ \tau_m^{l',a}]\!] \text{ with } a \in \{1, \infty\} \\ & & \mid KT \mid \mathrm{pkey}(KT) \mid \mathrm{vkey}(KT) \mid (T)_T \mid \{T\}_T \end{array}$$

Fig. 4. Types for terms

3.3 Equivalence

When processes evolve, sent messages are stored in a substitution ϕ while the values of variables are stored in σ. A *frame* is simply a substitution ψ where $\mathrm{dom}(\psi) \subseteq \mathcal{AX}$. It represents the knowledge of an attacker. In what follows, we will typically consider $\phi\sigma$.

Intuitively, two sequences of messages are indistinguishable to an attacker if he cannot perform any test that could distinguish them. This is typically modelled as static equivalence [2]. Here, we consider of variant of [2] where the attacker is also given the ability to observe when the evaluation of a term fails, as defined for example in [25].

Definition 1 (Static Equivalence). *Two ground frames ϕ and ϕ' are statically equivalent if and only if they have the same domain, and for all attacker terms R, S with variables in $\mathrm{dom}(\phi) = \mathrm{dom}(\phi')$, we have*

$$(R\phi =_{\downarrow} S\phi) \iff (R\phi' =_{\downarrow} S\phi')$$

Then two processes P and Q are in equivalence if no matter how the adversary interacts with P, a similar interaction may happen with Q, with equivalent resulting frames.

Definition 2 (Trace Equivalence). *Let P, Q be two processes. We write $P \sqsubseteq_t Q$ if for all $(s, \phi, \sigma) \in \mathrm{trace}(P)$, there exists $(s', \phi', \sigma') \in \mathrm{trace}(Q)$ such that $s =_\tau s'$ and $\phi\sigma$ and $\phi'\sigma'$ are statically equivalent. We say that P and Q are trace equivalent, and we write $P \approx_t Q$, if $P \sqsubseteq_t Q$ and $Q \sqsubseteq_t P$.*

Note that this definition already includes the attacker's behaviour, since processes may input any message forged by the attacker.

Example 2. As explained in Sect. 2, anonymity is modelled as an equivalence property. Intuitively, an attacker should not be able to know which agents are executing the protocol. In the case of protocol PA, presented in Example 1, the anonymity property can be modelled by the following equivalence:

$$P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_a)) \approx_t P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_c))$$

4 A Type System for Dynamic Keys

Types. In our type system we give types to pairs of messages – one from the left process and one from the right one. We store the types of nonces, variables, and keys in a typing environment Γ. While we store a type for a single nonce or variable occurring in both processes, we assign a potentially different type to every different combination of keys (k, k') used in the left and right process – so called *bikeys*. This is an important non-standard feature that enables us to type protocols using different encryption and decryption keys.

The types for messages are defined in Fig. 4 and explained below. Selected subtyping rules are given in Fig. 5. We assume three security labels $\mathtt{HH}$, $\mathtt{HL}$ and $\mathtt{LL}$,

$$\frac{}{\text{eqkey}^l(T) <: \text{key}^l(T)}\ (\textsc{SEqKey}) \qquad \frac{}{\text{seskey}^{l,a}(T) <: \text{eqkey}^l(T)}\ (\textsc{SSesKey})$$

$$\frac{}{\text{key}^l(T) <: l}\ (\textsc{SKey}) \qquad \frac{T <: \text{eqkey}^l(T')}{\text{pkey}(T) <: \mathtt{LL}}\ (\textsc{SPubKey}) \qquad \frac{T <: \text{eqkey}^l(T')}{\text{vkey}(T) <: \mathtt{LL}}\ (\textsc{SVKey})$$

$$\frac{T <: T'}{(T)_{T''} <: (T')_{T''}}\ (\textsc{SEnc}) \qquad \frac{T <: T'}{\{T\}_{T''} <: \{T'\}_{T''}}\ (\textsc{SAenc})$$

Fig. 5. Selected subtyping rules

ranged over by l, whose first (resp. second) component denotes the confidentiality (resp. integrity) level. Intuitively, values of high confidentiality may never be output to the network in plain, and values of high integrity are guaranteed not to originate from the attacker. Pair types $T * T'$ describe the type of their components and the type $T \vee T'$ is given to messages that can have type T or type T'.

The type $\tau_n^{l,a}$ describes nonces and constants of security level l: the label a ranges over $\{\infty, 1\}$, denoting whether the nonce is bound within a replication or not (constants are always typed with $a = 1$). We assume a different identifier n for each constant and restriction in the process. The type $\tau_n^{l,1}$ is populated by a single name, (i.e., n describes a constant or a non-replicated nonce) and $\tau_n^{l,\infty}$ is a special type, that is instantiated to $\tau_{n_j}^{l,1}$ in the jth replication of the process. Type $[\![\tau_n^{l,a}\,;\,\tau_m^{l',a}]\!]$ is a refinement type that restricts the set of possible values of a message to values of type $\tau_n^{l,a}$ on the left and type $\tau_m^{l',a}$ on the right. For a refinement type $[\![\tau_n^{l,a}\,;\,\tau_n^{l,a}]\!]$ with equal types on both sides we write $\tau_n^{l,a}$.

Keys can have three different types ranged over by KT, ordered by a subtyping relation (SEqKey, SSesKey): $\text{seskey}^{l,a}(T) <: \text{eqkey}^l(T) <: \text{key}^l(T)$. For all three types, l denotes the security label (SKey) of the key and T is the type of the payload that can be encrypted or signed with these keys. This allows us to transfer typing information from one process to another one: e.g. when encrypting, we check that the payload type is respected, so that we can be sure to get a value of the payload type upon decryption. The three different types encode different relations between the left and the right component of a bikey (k, k'). While type $\text{key}^l(T)$ can be given to bikeys with different components $k \neq k'$, type $\text{eqkey}^l(T)$ ensures that the keys are equal on both sides in the specific typed instruction. Type $\text{seskey}^{l,a}(T)$ additionally guarantees that the key is always the same on the left and the right throughout the whole process. We allow for dynamic generation of keys of type $\text{seskey}^{l,a}(T)$ and use a label a to denote whether the key is generated under replication or not – just like for nonce types.

For a key of type T, we use types $\text{pkey}(T)$ and $\text{vkey}(T)$ for the corresponding public key and verification key, and types $(T')_T$ and $\{T'\}_T$ for symmetric and asymmetric encryptions of messages of type T' with this key. Public keys and verification keys can be treated as LL if the corresponding keys are equal (SPubkey, SVkey) and subtyping on encryptions is directly induced by subtyping of the payload types (SEnc, SAenc) (Fig. 6).

$$\frac{\Gamma(n) = \tau_n^{l,a} \quad \Gamma(m) = \tau_m^{l,a} \quad l \in \{\mathtt{HH}, \mathtt{HL}\}}{\Gamma \vdash n \sim m : l \to \emptyset}\ (\textsc{TNonce}) \qquad \frac{\Gamma(n) = \tau_n^{\mathtt{LL},a}}{\Gamma \vdash n \sim n : \mathtt{LL} \to \emptyset}\ (\textsc{TNonceL})$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x \sim x : T \to \emptyset}\ (\textsc{TVar}) \qquad \frac{\Gamma \vdash M \sim N : T' \to c \qquad T' <: T}{\Gamma \vdash M \sim N : T \to c}\ (\textsc{TSub})$$

$$\frac{\Gamma \vdash M \sim N : T \to c \qquad \Gamma \vdash M' \sim N' : T' \to c'}{\Gamma \vdash \langle M, M' \rangle \sim \langle N, N' \rangle : T * T' \to c \cup c'}\ (\textsc{TPair})$$

$$\frac{M, N \text{ well formed}}{\Gamma \vdash M \sim N : \mathtt{HL} \to \emptyset}\ (\textsc{THigh})$$

$$\frac{\Gamma(k, k') = T}{\Gamma \vdash k \sim k' : T \to \emptyset}\ (\textsc{TKey}) \qquad \frac{k \in \mathrm{keys}(\Gamma) \cup \mathcal{FK}}{\Gamma \vdash \mathtt{pk}(k) \sim \mathtt{pk}(k) : \mathtt{LL} \to \emptyset}\ (\textsc{TPubKeyL})$$

$$\frac{\Gamma \vdash M \sim N : T \to \emptyset \qquad \exists T', l.T <: \mathrm{key}^l(T')}{\Gamma \vdash \mathtt{pk}(M) \sim \mathtt{pk}(N) : \mathrm{pkey}(T) \to \emptyset}\ (\textsc{TPubKey})$$

$$\frac{\begin{array}{c}\Gamma \vdash M \sim N : T \to c \qquad \Gamma \vdash M' \sim N' : T' \to c' \\ T' = \mathtt{LL} \ \vee\ (\exists T'', T''', l.T' = \mathrm{pkey}(T'') \wedge T'' <: \mathrm{key}^l(T'''))\end{array}}{\Gamma \vdash \mathtt{aenc}(M, M') \sim \mathtt{aenc}(N, N') : \{T\}_{T'} \to c \cup c'}\ (\textsc{TAenc})$$

$$\frac{\Gamma \vdash M \sim N : \{T\}_{\mathrm{pkey}(T')} \to c \qquad T' <: \mathrm{key}^{\mathtt{HH}}(T)}{\Gamma \vdash M \sim N : \mathtt{LL} \to c \cup \{M \sim N\}}\ (\textsc{TAencH})$$

$$\frac{\Gamma \vdash M \sim N : \{\mathtt{LL}\}_T \to c \qquad (T = \mathrm{pkey}(T') \wedge T' <: \mathrm{eqkey}^l(T'')) \text{ or } T = \mathtt{LL}}{\Gamma \vdash M \sim N : \mathtt{LL} \to c}\ (\textsc{TAencL})$$

Fig. 6. Selected rules for messages

Constraints. When typing messages, we generate constraints of the form $(M \sim N)$, meaning that the attacker may see M and N in the left and right process, respectively, and these two messages are thus required to be indistinguishable.

Due to space reasons we only present a few selected rules that are characteristic of the typing of branching protocols. The omitted rules are similar in spirit to the presented ones or are standard rules for equivalence typing [28].

4.1 Typing Messages

The typing judgement for messages is of the form $\Gamma \vdash M \sim N : T \to c$ which reads as follows: under the environment Γ, M and N are of type T and either this is a high confidentiality type (i.e., M and N are not disclosed to the attacker) or M and N are indistinguishable for the attacker assuming the set of constraints c is consistent.

Confidential nonces can be given their label from the typing environment in rule TNonce. Since their label prevents them from being released in clear, the attacker cannot observe them and we do not need to add constraints for

them. They can however be output in encrypted form and will then appear in the constraints of the encryption. Public nonces (labeled as LL) can be typed if they are equal on both sides (rule TNonceL). These are standard rules, as well as the rules TVar, TSub, TPair and THigh [28].

A non-standard rule that is crucial for the typing of branching protocols is rule TKey. As the typing environment contains types for bikeys (k, k') this rule allows us to type two potentially different keys with their type from the environment. With the standard rule TPubKeyL we can only type a public key of the same keys on both sides, while rule TPubKey allows us to type different public keys $\mathtt{pk}(M), \mathtt{pk}(N)$, provided we can show that there exists a valid key type for the terms M and N. This highlights another important technical contribution of this work, as compared to existing type systems for equivalence: we do not only support a fixed set of keys, but also allow for the usage of keys in variables, that have been received from the network.

To show that a message is of type $\{T\}_{T'}$ – a message of type T encrypted asymmetrically with a key of type T', we have to show that the corresponding terms have exactly these types in rule TAenc. The generated constraints are simply propagated. In addition we need to show that T' is a valid type for a public key, or LL, which models untrusted keys received from the network. Note, that this rule allows us to encrypt messages with different keys in the two processes. For encryptions with honest keys (label HH) we can use rule TAenc to give type LL to the messages, if we can show that the payload type is respected. In this case we add the entire encryptions to the constraints, since the attacker can check different encryptions for equality, even if he cannot obtain the plaintext. Rule TAencL allows us to give type LL to encryptions even if we do not respect the payload type, or if the key is corrupted. However, we then have to type the plaintexts with type LL since we cannot guarantee their confidentiality. Additionally, we have to ensure that the same key is used in both processes, because the attacker might possess the corresponding private keys and test which decryption succeeds. Since we already add constraints for giving type LL to the plaintext, we do not need to add any additional constraints.

4.2 Typing Processes

From now on, we assume that processes assign a type to freshly generated nonces and keys. That is, $\mathtt{new}\ n.P$ is now of the form $\mathtt{new}\ n : T.\ P$. This requires a (very light) type annotation from the user. The typing judgement for processes is of the form $\Gamma \vdash P \sim Q \to C$ and can be interpreted as follows: If two processes P and Q can be typed in Γ and if the generated constraint set C is consistent, then P and Q are trace equivalent. We present selected rules in Fig. 7.

Rule POut states that we can output messages to the network if we can type them with type LL, i.e., they are indistinguishable to the attacker, provided that the generated set c of constraints is consistent. The constraints of c are then added to all constraints in the constraint set C. We define $C \cup_\forall c' := \{(c \cup c', \Gamma) \mid (c, \Gamma) \in C\}$. This rule, as well as the rules PZero, PIn, PNew, PPar, and PLet, are standard rules [28].

$$\frac{\Gamma \vdash P \sim Q \to C \qquad \Gamma \vdash M \sim N : \mathtt{LL} \to c}{\Gamma \vdash \mathtt{out}(M).P \sim \mathtt{out}(N).Q \to C \cup_\forall c} \text{(POUT)}$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \text{ does not contain union types}}{\Gamma \vdash \mathtt{0} \sim \mathtt{0} \to (\emptyset, \Gamma)} \text{(PZERO)} \qquad \frac{\Gamma, x : \mathtt{LL} \vdash P \sim Q \to C}{\Gamma \vdash \mathtt{in}(x).P \sim \mathtt{in}(x).Q \to C} \text{(PIN)}$$

$$\frac{\Gamma, n : \tau_n^{l,a} \vdash P \sim Q \to C}{\Gamma \vdash \mathtt{new}\ n : \tau_n^{l,a}.P \sim \mathtt{new}\ n : \tau_n^{l,a}.Q \to C} \text{(PNEW)}$$

$$\frac{\Gamma, (k,k) : \mathrm{seskey}^{l,a}(T) \vdash P \sim Q \to C}{\Gamma \vdash \mathtt{new}\ k : \mathrm{seskey}^{l,a}(T).P \sim \mathtt{new}\ k : \mathrm{seskey}^{l,a}(T).Q \to C} \text{(PNEWKEY)}$$

$$\frac{\Gamma \vdash P \sim Q \to C \qquad \Gamma \vdash P' \sim Q' \to C'}{\Gamma \vdash P \mid P' \sim Q \mid Q' \to C \cup_\times C'} \text{(PPAR)}$$

$$\frac{\Gamma \vdash_d t \sim t' : T \qquad \Gamma, x : T \vdash P \sim Q \to C \qquad \Gamma \vdash P' \sim Q' \to C'}{\Gamma \vdash \mathtt{let}\ x = t\ \mathtt{in}\ P\ \mathtt{else}\ P' \sim \mathtt{let}\ x = t'\ \mathtt{in}\ Q\ \mathtt{else}\ Q' \to C \cup C'} \text{(PLET)}$$

(PLETADECSAME)

$$\frac{\begin{array}{c}\Gamma(y) = \mathtt{LL} \qquad \Gamma(k,k) <: \mathrm{key}^{\mathtt{HH}}(T) \\ \Gamma, x : T \vdash P \sim Q \to C \qquad \Gamma, x : \mathtt{LL} \vdash P \sim Q \to C' \qquad \Gamma \vdash P' \sim Q' \to C'' \\ (\forall T'.\forall k' \neq k.\ \Gamma(k,k') <: \mathrm{key}^{\mathtt{HH}}(T') \Rightarrow \Gamma, x : T' \vdash P \sim Q' \to C_{k'}) \\ (\forall T'.\forall k' \neq k.\ \Gamma(k',k) <: \mathrm{key}^{\mathtt{HH}}(T') \Rightarrow \Gamma, x : T' \vdash P' \sim Q \to C'_{k'})\end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{let}\ x = \mathtt{adec}(y,k)\ \mathtt{in}\ P\ \mathtt{else}\ P' \sim \mathtt{let}\ x = \mathtt{adec}(y,k)\ \mathtt{in}\ Q\ \mathtt{else}\ Q' \\ \to C \cup C' \cup C'' \cup (\bigcup_{k'} C_{k'}) \cup (\bigcup_{k'} C'_{k'})\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash P \sim Q \to C_1 \\ \Gamma \vdash P \sim Q' \to C_2 \qquad \Gamma \vdash P' \sim Q \to C_3 \qquad \Gamma \vdash P' \sim Q' \to C_4\end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{if}\ M = M'\ \mathtt{then}\ P\ \mathtt{else}\ P' \sim \mathtt{if}\ N = N'\ \mathtt{then}\ Q\ \mathtt{else}\ Q' \\ \to C_1 \cup C_2 \cup C_3 \cup C_4\end{array}} \text{(PIFALL)}$$

Fig. 7. Selected rules for processes

Rule PNEWKEY allows us to generate new session keys at runtime, which models security protocols more faithfully. It also allows us to generate infinitely many keys, by introducing new keys under replication.

Rule PLETADECSAME treats asymmetric decryptions where we use the same fixed honest key (label HH) for decryptions in both processes. Standard type systems for equivalence have a simplifying (and restrictive) invariant that guarantees that encryptions are always performed using the same keys in both processes and hence guarantee that both processes always take the same branch in decryption (compare rule PLET). In our system however, we allow encryptions with potentially different keys, which requires cross-case validation in order to retain soundness. Still, the number of possible combinations of encryption keys is limited by the assignments in the typing environment Γ. To cover all the possibilities, we type the following combinations of continuation processes:

$$\frac{\Gamma(k,k) <: \text{key}^{\texttt{LL}}(T) \qquad \Gamma(x) = \texttt{LL}}{\Gamma \vdash_d \texttt{adec}(x,k) \sim \texttt{adec}(x,k) : \texttt{LL}} \text{ (DADECL)}$$

$$\frac{\Gamma(y) = \text{seskey}^{\texttt{HH},a}(T) \qquad \Gamma(x) = \texttt{LL}}{\Gamma \vdash_d \texttt{adec}(x,y) \sim \texttt{adec}(x,y) : T \vee \texttt{LL}} \text{ (DADECH')}$$

$$\frac{(\Gamma(y) = \text{seskey}^{\texttt{LL},a}(T) \vee \Gamma(y) = \texttt{LL}) \qquad \Gamma(x) = \texttt{LL}}{\Gamma \vdash_d \texttt{adec}(x,y) \sim \texttt{adec}(x,y) : \texttt{LL}} \text{ (DADECL')}$$

$$\frac{\Gamma(k,k) = \text{seskey}^{l,a}(T') \qquad \Gamma(x) = \{T\}_{\text{pkey}(\text{seskey}^{l,a}(T'))}}{\Gamma \vdash_d \texttt{adec}(x,k) \sim \texttt{adec}(x,k) : T} \text{ (DADECT)}$$

$$\frac{\Gamma(y) = \text{seskey}^{l,a}(T') \qquad \Gamma(x) = \{T\}_{\text{pkey}(\text{seskey}^{l,a}(T'))}}{\Gamma \vdash_d \texttt{adec}(x,y) \sim \texttt{adec}(x,y) : T} \text{ (DADECT')}$$

Fig. 8. Selected destructor rules

- Both `then` branches: In this case we know that key k was used for encryption on both sides. Because of $\Gamma(k,k) = \text{key}^{\texttt{HH}}(T)$, we know that in this case the payload type is T and we type the continuation with $\Gamma, x : T$.
  Because the message may also originate from the attacker (who also has access to the public key), we have to type the two `then` branches also with $\Gamma, x : \texttt{LL}$.
- Both `else` branches: If decryption fails on both sides, we type the two `else` branches without introducing any new variables.
- Left `then`, right `else`: The encryption may have been created with key k on the left side and another key k' on the right side. Hence, for each $k' \neq k$, such that $\Gamma(k,k')$ maps to a key type with label `HH` and payload type T', we have to typecheck the left `then` branch and the right `else` branch with $\Gamma, x : T'$.
- Left `else`, right `then`: This case is analogous to the previous one.

The generated set of constraints is simply the union of all generated constraints for the subprocesses. Rule PIFALL lets us typecheck any conditional by simply checking the four possible branch combinations. In contrast to the other rules for conditionals that we present in a companion technical report, this rule does not require any other preconditions or checks on the terms M, M', N, N'.

Destructor Rules. The rule PLET requires that a destructor application succeeds or fails equally in the two processes. To ensure this property, it relies on additional rules for destructors. We present selected rules in Fig. 8. Rule DADECL is a standard rule that states that a decryption of a variable of type `LL` with an untrusted key (label `LL`) yields a result of type `LL`. Decryption with a trusted (label `HH`) session key gives us a value of the key's payload type or type `LL` in case the encryption was created by the attacker using the public key. Here it is important that the key is of type $\text{seskey}^{\texttt{HH},a}(T)$, since this guarantees that the key is never used in combination with a different key and hence decryption will always equally succeed or fail in both processes. Rule DADECL' is similar to

$$\dfrac{\dfrac{}{\Gamma \vdash \langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle \sim N_b : \mathtt{HL} \to \emptyset}\ \textsc{THigh} \quad * \quad \dfrac{\dfrac{\Gamma(k_a, k) = \mathrm{key}^{\mathtt{HH}}(\mathtt{HL})}{\Gamma \vdash k_a \sim k : \mathrm{key}^{\mathtt{HH}}(\mathtt{HL}) \to \emptyset}\ \textsc{TKey}}{\Gamma \vdash \mathtt{pk}(k_a) \sim \mathtt{pk}(k) : \mathrm{pkey}(\mathrm{key}^{\mathtt{HH}}(\mathtt{HL})) \to \emptyset}\ \textsc{TPubKey}}{\Gamma \vdash \mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k)) : \{\mathtt{HL}\}_{\mathrm{pkey}(\mathrm{key}^{\mathtt{HH}}(\mathtt{HL}))} \to \emptyset}\ \textsc{TAenc}$$

$$\dfrac{\Gamma \vdash \mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k)) : \mathtt{LL} \to C}{}\ \textsc{TAencH}$$

where $C = \{\mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k))\}$.

Fig. 9. Type derivation for the response to A and the decoy message

rule DAdecL except it uses a variable for decryption instead of a fixed key. Rule DAdecT treats the case in which we know that the variable x is an asymmetric encryption of a specific type. If the type of the key used for decryption matches the key type used for encryption, we know the exact type of the result of a successful decryption. DAdecT' is similar to DAdecT, with a variable as key. In a companion technical report we present similar rules for symmetric decryption and verification of signatures.

4.3 Typing the Private Authentication Protocol

We now show how our type system can be applied to type the Private Authentication protocol presented in Sect. 2.3, by showing the most interesting parts of the derivation. We type the protocol using the initial environment Γ presented in Fig. 1.

We focus on the responder process P_b and start with the asymmetric decryption. As we use the same key k_b in both processes, we apply rule PLetAdecSame. We have $\Gamma(x) = \mathtt{LL}$ by rule PIn and $\Gamma(k_b, k_b) = \mathrm{key}^{\mathtt{HH}}(\mathtt{HH}, \mathtt{LL})$. We do not have any other entry using key k_b in Γ. We hence typecheck the two `then` branches once with $\Gamma, y : (\mathtt{HH} * \mathtt{LL})$ and once with $\Gamma, y : \mathtt{LL}$, as well as the two `else` branches (which are just `0` in this case).

Typing the let expressions is straightforward using rule PLet. In the conditional we check $y_2 = \mathtt{pk}(k_a)$ in the left process and $y_2 = \mathtt{pk}(k_c)$ in the right process. Since we cannot guarantee which branches are taken or even if the same branch is taken in the two processes, we use rule PIfAll to typecheck all four possible combinations of branches. We now focus on the case where A is successfully authenticated in the left process and is rejected in the right process. We then have to typecheck B's positive answer together with the decoy message: $\Gamma \vdash \mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_c, \mathtt{pk}(k)) : \mathtt{LL}$.

Figure 9 presents the type derivation for this example. We apply rule TAenc to give type LL to the two terms, adding the two encryptions to the constraint set. Using rule TAencH we can show that the encryptions are well-typed with type $\{\mathtt{HL}\}_{\mathrm{pkey}(\mathrm{key}^{\mathtt{HH}}(\mathtt{HL}))}$. The type of the payload is trivially shown with rule THigh.

To type the public key, we use rule TPubKey followed by rule TKey, which looks up the type for the bikey (k_a, k) in the typing environment Γ.

5 Consistency

Our type system collects constraints that intuitively correspond to (symbolic) messages that the attacker may see (or deduce). Therefore, two processes are in trace equivalence only if the collected constraints are in static equivalence for any plausible instantiation.

However, checking static equivalence of symbolic frames for any instantiation corresponding to a real execution may be as hard as checking trace equivalence [24]. Conversely, checking static equivalence for *any* instantiation may be too strong and may prevent proving equivalence of processes. Instead, we use again the typing information gathered by our type system and we consider only instantiations that comply with the type. Actually, we even restrict our attention to instantiations where variables of type LL are only replaced by deducible terms. This last part is a key ingredient for considering processes with dynamic keys. Hence, we define a constraint to be *consistent* if the corresponding two frames are in static equivalence for any instantiation that can be typed and produces constraints that are included in the original constraint.

Formally, we first introduce the following ingredients:

- $\phi_\ell(c)$ and $\phi_r(c)$ denote the frames that are composed of the left and the right terms of the constraints respectively (in the same order).
- $\phi^\Gamma_{\mathtt{LL}}$ denotes the frame that is composed of all low confidentiality nonces and keys in Γ, as well as all public encryption keys and verification keys in Γ. This intuitively corresponds to the initial knowledge of the attacker.
- Two ground substitutions σ, σ' are well-typed in Γ with constraint c_σ if they preserve the types for variables in Γ, *i.e.*, for all x, $\Gamma \vdash \sigma(x) \sim \sigma'(x) : \Gamma(x) \to c_x$, and $c_\sigma = \bigcup_{x \in \mathrm{dom}(\Gamma)} c_x$.

The instantiation of a constraint is defined as expected. If c is a set of constraints, and σ, σ' are two substitutions, let $[\![c]\!]_{\sigma,\sigma'}$ be the instantiation of c by σ on the left and σ' on the right, that is, $[\![c]\!]_{\sigma,\sigma'} = \{M\sigma \sim N\sigma' \mid M \sim N \in c\}$.

Definition 3 (Consistency). *A set of constraints c is* consistent *in an environment Γ if for all substitutions σ, σ' well-typed in Γ with a constraint c_σ such that $c_\sigma \subseteq [\![c]\!]_{\sigma,\sigma'}$, the frames $\phi^\Gamma_{\mathtt{LL}} \cup \phi_\ell(c)\sigma$ and $\phi^\Gamma_{\mathtt{LL}} \cup \phi_r(c)\sigma'$ are statically equivalent. We say that (c, Γ) is consistent if c is consistent in Γ and that a constraint set C is consistent in Γ if each element $(c, \Gamma) \in C$ is consistent.*

Compared to [28], we now require $c_\sigma \subseteq [\![c]\!]_{\sigma,\sigma'}$. This means that instead of considering any (well typed) instantiations, we only consider instantiations that use fragments of the constraints. For example, this now imposes that low variables are instantiated by terms deducible from the constraint. This refinement of consistency provides a tighter definition and is needed for non fixed keys, as explained in the next section.

6 Soundness

In this section, we provide our main results. First, soundness of our type system: whenever two processes can be typed with consistent constraints, then they are in trace equivalence. Then we show how to automatically prove consistency. Finally, we explain how to lift these two first results from finite processes to processes with replication. But first, we discuss why we cannot directly apply the results from [28] developed for processes with long term keys.

6.1 Example

Consider the following example, typical for a key-exchange protocol: Alice receives some key and uses it to encrypt, e.g. a nonce. Here, we consider a semi-honest session, where an honest agent A is receiving a key from a dishonest agent D. Such sessions are typically considered in combination with honest sessions.

$$C \to A : \mathtt{aenc}(\langle k, C\rangle, \mathtt{pk}(A))$$
$$A \to C : \mathtt{aenc}(n, k)$$

The process modelling the role of Alice is as follows.

$$P_A = \mathtt{in}(x).\ \mathtt{let}\ x' = \mathtt{adec}(x, k_A)\ \mathtt{in}\ \ \mathtt{let}\ y = \pi_1(x')\ \mathtt{in}\ \ \mathtt{let}\ z = \pi_2(x')\ \mathtt{in}$$
$$\mathtt{if}\ z = C\ \mathtt{then}\ \ \mathtt{new}\ n.\ \mathtt{out}(\mathtt{enc}(n, y))$$

When type-checking $P_A \sim P_A$ (as part as a more general process with honest sessions), we would collect the constraint $\mathtt{enc}(n, y) \sim \mathtt{enc}(n, y)$ where y comes from the adversary and is therefore a low variable (that is, of type $\mathtt{LL}$). The approach of [28] consisted in opening messages as much as possible. In this example, this would yield the constraint $y \sim y$ which typically renders the constraint inconsistent, as exemplified below.

When typechecking the private authentication protocol, we obtain constraints containing $\mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k))$ (as seen in Fig. 9), where y_1 has type $\mathtt{HL}$. Assume now that the constraint also contains $y \sim y$ for some variable y of type $\mathtt{LL}$ and consider the following instantiations of y and y_1: $\sigma(y_1) = \sigma'(y_1) = a$ for some constant a and $\sigma(y) = \sigma'(y) = \mathtt{aenc}(N_b, \mathtt{pk}(k))$. Note that such an instantiation complies with the type since $\Gamma \vdash \sigma(y) \sim \sigma'(y) : \mathtt{LL} \to c$ for some constraint c. The instantiated constraint would then contain

$$\{\mathtt{aenc}(\langle a, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k)),$$
$$\mathtt{aenc}(N_b, \mathtt{pk}(k)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k))\}$$

and the corresponding frames are not statically equivalent, which makes the constraint inconsistent for the consistency definition of [28].

Therefore, our first idea consists in proving that we only collect constraints that are saturated w.r.t. deduction: any deducible subterm can already be constructed from the terms of the constraint. Second, we show that for any execution, low variables are instantiated by terms deducible from the constraints.

This guarantees that our new notion of consistency is sound. The two results are reflected in the next section.

6.2 Soundness

Our type system, together with consistency, implies trace equivalence.

Theorem 1 (Typing implies trace equivalence). *For all P, Q, and C, for all Γ containing only keys, if $\Gamma \vdash P \sim Q \to C$ and C is consistent, then $P \approx_t Q$.*

Example 3. We can typecheck PA, that is

$$\Gamma \vdash P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_a)) \sim P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_c)) \to C_{PA}$$

where Γ has been defined in Fig. 1 and assuming that nonce N_a of process P_a has been annotated with type $\tau_{N_a}^{\mathtt{HH},1}$ and nonce N_b of P_b has been annotated with type $\tau_{N_b}^{\mathtt{HH},1}$. The constraint set C_{PA} can be proved to be consistent using the procedure presented in the next section. Therefore, we can conclude that

$$P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_a)) \approx_t P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_c))$$

which shows anonymity of the private authentication protocol.

The first key ingredient in the proof of Theorem 1 is the fact that any well-typed low term is deducible from the constraint generated when typing it.

Lemma 1 (Low terms are recipes on their constraints). *For all ground messages M, N, for all Γ, c, if $\Gamma \vdash M \sim N : \mathtt{LL} \to c$ then there exists an attacker recipe R without destructors such that $M = R(\phi_\ell(c) \cup \phi_{\mathtt{LL}}^{\Gamma})$ and $N = R(\phi_r(c) \cup \phi_{\mathtt{LL}}^{\Gamma})$.*

The second key ingredient is a finer invariant on protocol executions: for any typable pair of processes P, Q, any execution of P can be mimicked by an execution of Q such that low variables are instantiated by well-typed terms constructible from the constraint.

Lemma 2. *For all processes P, Q, for all ϕ, σ, for all multisets of processes $\mathcal{P}$, constraint sets C, sequences s of actions, for all Γ containing only keys, if $\Gamma \vdash P \sim Q \to C$, C is consistent, and $(\{P\}, \emptyset, \emptyset) \xrightarrow{s}_* (\mathcal{P}, \phi, \sigma)$, then there exist a sequence s' of actions, a multiset $\mathcal{Q}$, a frame ϕ', a substitution σ', an environment Γ', a constraint c such that:*

– $(\{Q\}, \emptyset, \emptyset) \xrightarrow{s'}_* (\mathcal{Q}, \phi', \sigma')$, with $s =_\tau s'$

– $\Gamma' \vdash \phi\sigma \sim \phi'\sigma' : \mathtt{LL} \to c$, and for all $x \in \mathrm{dom}(\sigma) \cap \mathrm{dom}(\sigma')$, there exists c_x such that $\Gamma' \vdash \sigma(x) \sim \sigma(x) : \Gamma'(x) \to c_x$ and $c_x \subseteq c$.

Note that this finer invariant guarantees that we can restrict our attention to the instantiations considered for defining consistency.

As a by-product, we obtain a finer type system for equivalence, even for processes with long term keys (as in [28]). For example, we can now prove equivalence of processes where some agent signs a low message that comes from the adversary. In such a case, we collect $\mathtt{sign}(x, k) \sim \mathtt{sign}(x, k)$ in the constraint, where x has type $\mathtt{LL}$, which we can now prove to be consistent (depending on how x is used in the rest of the constraint).

6.3 Procedure for Consistency

We devise a procedure $\texttt{check\_const}(C)$ for checking consistency of a constraint C, depicted in Fig. 10. Compared to [28], the procedure is actually simplified. Thanks to Lemmas 1 and 2, there is no need to open constraints anymore. The rest is very similar and works as follows:

- First, variables of refined type $[\![\tau_m^{l,1}\,;\,\tau_n^{l',1}]\!]$ are replaced by m on the left-hand-side of the constraint and n on the right-hand-side.
- Second, we check that terms have the same shape (encryption, signature, hash) on the left and on the right and that asymmetric encryption and hashes cannot be reconstructed by the adversary (that is, they contain some fresh nonce).
- The most important step consists in checking that the terms on the left satisfy the same equalities than the ones on the right. Whenever two left terms M and N are unifiable, their corresponding right terms M' and N' should be equal after applying a similar instantiation.

For constraint sets without infinite nonce types, `check_const` entails consistency.

Theorem 2. *Let C be a set of constraints such that*

$$\forall (c, \Gamma) \in C.\ \forall l, l', m, p.\ \Gamma(x) \neq [\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!].$$

If $\texttt{check\_const}(C) = \texttt{true}$*, then C is consistent.*

Example 4. Continuing Example 3, typechecking the PA protocol yields the set C_{PA} of constraint sets. C_{PA} contains in particular the set

$$\{\texttt{aenc}(\langle N_a, \texttt{pk}(k_a)\rangle, \texttt{pk}(k_b)) \sim \texttt{aenc}(\langle N_a, \texttt{pk}(k_a)\rangle, \texttt{pk}(k_b)),$$
$$\texttt{aenc}(\langle y_1, \langle N_b, \texttt{pk}(k_b)\rangle\rangle, \texttt{pk}(k_a)) \sim \texttt{aenc}(N_b, \texttt{pk}(k))\}$$

where variable y_1 has type `HL` (we also have the same constraint but where y_1 has type `LL`). The other constraint sets of C_{PA} are similar and correspond to the various cases (`else` branch of P_a with `then` branch of P_b, etc.). The procedure `check_const` returns true since no two terms can be unified, which proves consistency. Similarly, the other constraints generated for PA can be proved to be consistent applying `check_const`.

6.4 From Finite to Replicated Processes

The previous results apply to processes without replication only. In the spirit of [28], we lift our results to replicated processes. We proceed in two steps.

1. Whenever $\Gamma \vdash P \sim Q \to C$, we show that:
$[\,\Gamma\,]_1 \cup \cdots \cup [\,\Gamma\,]_n \vdash [\,P\,]_1 | \ldots | [\,P\,]_n \sim [\,Q\,]_1 | \ldots | [\,Q\,]_n \to [\,C\,]_1 \cup_\times \cdots \cup_\times [\,C\,]_n$,
where $[\,\Gamma\,]_i$ is intuitively a copy of Γ, where variables x have been replaced by x_i, and nonces or keys n of infinite type $\tau_n^{l,\infty}$ (or $\text{seskey}^{l,\infty}(T)$) have been replaced by n_i. The copies $[\,P\,]_i$, $[\,Q\,]_i$, and $[\,C\,]_i$ are defined similarly.

$\texttt{step1}_\Gamma(c) := ([\![c]\!]_{\sigma_F,\sigma'_F}, \Gamma')$, with

$$F := \{x \in \mathrm{dom}(\Gamma) \mid \exists m, n, l, l'.\ \Gamma(x) = [\![\tau_m^{l,1}\ ;\ \tau_n^{l',1}]\!]\}$$

and σ_F, σ'_F defined by

$$\begin{cases} \bullet\ \mathrm{dom}(\sigma_F) = \mathrm{dom}(\sigma'_F) = F \\ \bullet\ \forall x \in F.\ \forall m, n, l, l'.\Gamma(x) = [\![\tau_m^{l,1}\ ;\ \tau_n^{l',1}]\!] \Rightarrow \sigma_F(x) = m\ \wedge\ \sigma'_F(x) = n \end{cases}$$

and Γ' is $\Gamma|_{\mathrm{dom}(\Gamma)\setminus F}$ extended with $\Gamma'(n) = \tau_n^{l,1}$ for all nonce n such that $\tau_n^{l,1}$ occurs in Γ.

$\texttt{step2}_\Gamma(c) :=$ check that for all $M \sim N \in c$, M and N are both

- $\texttt{enc}(M', M'')$, $\texttt{enc}(N', N'')$ where M'', N'' are either
  - keys k, k' where $\exists T.\Gamma(k, k') <: \mathrm{key}^{\mathtt{HH}}(T)$;
  - or a variable x such that $\exists T.\Gamma(x) <: \mathrm{key}^{\mathtt{HH}}(T)$;
- or encryptions $\texttt{aenc}(M', M'')$, $\texttt{aenc}(N', N'')$ where
  - M' and N' contain directly under pairs a nonce n such that $\Gamma(n) = \tau_n^{\mathtt{HH},a}$ or a secret key k such that $\exists T, k'.\Gamma(k, k') <: \mathrm{key}^{\mathtt{HH}}(T)$ or $\Gamma(k', k) <: \mathrm{key}^{\mathtt{HH}}(T)$, or a variable x such that $\exists m, n, a.\Gamma(x) = [\![\tau_m^{\mathtt{HH},a}\ ;\ \tau_n^{\mathtt{HH},a}]\!]$, or a variable x such that $\exists T.\Gamma(x) <: \mathrm{key}^{\mathtt{HH}}(T)$;
  - M'' and N'' are either
    - public keys $\texttt{pk}(k)$, $\texttt{pk}(k')$ where $\exists T.\Gamma(k, k') <: \mathrm{key}^{\mathtt{HH}}(T)$;
    - or public keys $\texttt{pk}(x)$, $\texttt{pk}(x)$ where $\exists T.\Gamma(x) <: \mathrm{key}^{\mathtt{HH}}(T)$;
    - or a variable x such that $\exists T, T'.\Gamma(x) = \mathrm{pkey}(T)$ and $T <: \mathrm{key}^{\mathtt{HH}}(T')$;
- or hashes $\texttt{h}(M')$, $\texttt{h}(N')$, where M', N' similarly contain a secret value under pairs;
- or signatures $\texttt{sign}(M', M'')$, $\texttt{sign}(N', M'')$ where M'', N'' are either
  - keys k, k' where $\exists T.\Gamma(k, k') <: \mathrm{key}^{\mathtt{HH}}(T)$;
  - or a variable x such that $\exists T.\Gamma(x) <: \mathrm{key}^{\mathtt{HH}}(T)$;

$\texttt{step3}_\Gamma(c) :=$ If for all $M \sim M'$ and $N \sim N' \in c$ such that M, N are unifiable with a most general unifier μ, and such that

$$\forall x \in \mathrm{dom}(\mu).\exists l, l', m, p.\ (\Gamma(x) = [\![\tau_m^{l,\infty}\ ;\ \tau_p^{l',\infty}]\!]) \Rightarrow (x\mu \in \mathcal{X}\ \vee\ \exists i.\ x\mu = m_i)$$

we have

$$M'\alpha\theta = N'\alpha\theta$$

where

$$\forall x \in \mathrm{dom}(\mu).\forall l, l', m, p, i.(\Gamma(x) = [\![\tau_m^{l,\infty}\ ;\ \tau_p^{l',\infty}]\!]\ \wedge\ \mu(x) = m_i) \Rightarrow \theta(x) = p_i$$

and α is the restriction of μ to $\{x \in \mathrm{dom}(\mu) \mid \Gamma(x) = \mathtt{LL}\ \wedge\ \mu(x) \in \mathcal{N}\}$;
and if the symmetric condition for the case where M', N' are unifiable holds as well, then return `true`.

$\texttt{check\_const}(C) :=$ for all $(c, \Gamma) \in C$, let $(c_1, \Gamma_1) := \texttt{step1}_\Gamma(c)$ and check that $\texttt{step2}_{\Gamma_1}(c_1) = \texttt{true}$ and $\texttt{step3}_{\Gamma_1}(c_1) = \texttt{true}$.

Fig. 10. Procedure for checking consistency.

2. We cannot directly check consistency of infinitely many constraints of the form $[\,C\,]_1 \cup_\times \cdots \cup_\times [\,C\,]_n$. Instead, we show that it is sufficient to check consistency of two copies $[\,C\,]_1 \cup_\times [\,C\,]_2$ only. The reason why we need two copies (and not just one) is to detect when messages from different sessions may become equal.

Formally, we can prove trace equivalence of replicated processes.

Theorem 3. *Consider P, Q, P', Q', C, C', such that P, Q and P', Q' do not share any variable. Consider Γ, containing only keys and nonces with finite types.*

Assume that P and Q only bind nonces and keys with infinite nonce types, i.e. *using* `new` $m : \tau_m^{l,\infty}$ *and* `new` $k : \mathrm{seskey}^{l,\infty}(T)$ *for some label l and type T; while P' and Q' only bind nonces and keys with finite types,* i.e. *using* `new` $m : \tau_m^{l,1}$ *and* `new` $k : \mathrm{seskey}^{l,1}(T)$.

Let us abbreviate by `new` $\overline{n}$ *the sequence of declarations of each nonce $m \in \mathrm{dom}(\Gamma)$ and session key k such that $\Gamma(k,k) = \mathrm{seskey}^{l,1}(T)$ for some l, T. If*

- $\Gamma \vdash P \sim Q \to C$,
- $\Gamma \vdash P' \sim Q' \to C'$,
- `check_const`$([\,C\,]_1 \cup_\times [\,C\,]_2 \cup_\times [\,C'\,]_1) =$ `true`,

then $\quad$ `new` $\overline{n}.\ ((!P) \mid P') \approx_t$ `new` $\overline{n}.\ ((!Q) \mid Q')$.

Interestingly, Theorem 3 allows to consider a mix of finite and replicated processes.

7 Experimental Results

We implemented our typechecker as well as our procedure for consistency in a prototype tool TypeEq. We adapted the original prototype of [28] to implement additional cases corresponding to the new typing rules. This also required to design new heuristics w.r.t. the order in which typing rules should be applied. Of course, we also had to support for the new bikey types, and for arbitrary terms as keys. This represented a change of about 40% of the code of the software. We ran our experiments on a single Intel Xeon E5-2687Wv3 3.10 GHz core, with 378 GB of RAM (shared with the 19 other cores). Actually, our own prototype does not require a large amount of RAM. However, some of the other tools we consider use more than 64 GB of RAM on some examples (at which point we stopped the experiment). More precise figures about our tool are provided in the table of Fig. 11. The corresponding files can be found at [27].

We tested TypeEq on two symmetric key protocols that include a handshake on the key (Yahalom-Lowe and Needham-Schroeder symmetric key protocols). In both cases, we prove key usability of the exchanged key. Intuitively, we show that an attacker cannot distinguish between two encryptions of public constants: $P.\mathtt{out}(\mathtt{enc}(a,k)) \approx_t P.\mathtt{out}(\mathtt{enc}(b,k))$. We also consider one standard asymmetric key protocol (Needham-Schroeder-Lowe protocol), showing strong secrecy of the exchanged nonce.

Helios [4] is a well known voting protocol. We show ballot privacy, in the presence of a dishonest board, assuming that voters do not revote (otherwise the protocol is subject to a copy attack [39], a variant of [30]). We consider a more precise model than the previous Helios models which assume that voters initially know the election public key. Here, we model the fact that voters actually receive the (signed) freshly generated election public key from the network. The BAC protocol is one of the protocols embedded in the biometric passport [1]. We show anonymity of the passport holder $P(A) \approx_t P(B)$. Actually, the only data that distinguish $P(A)$ from $P(B)$ are the private keys. Therefore we consider an additional step where the passport sends the identity of the agent to the reader, encrypted with the exchanged key. Finally, we consider the private authentication protocol, as described in this paper.

7.1 Bounded Number of Sessions

We first compare TypeEq with the tools for a bounded number of sessions. Namely, we consider Akiss [22], APTE [23] as well as its optimised variant with partial order reduction APTE-POR [10], SPEC [32], and SatEquiv [26]. We step by step increase the number of sessions until we reach a "complete" scenario where each role is instantiated by A talking to B, A talking to C, B talking to A, and B talking to C, where A, B are honest while C is dishonest.

Protocols (# sessions)		Akiss	APTE	APTE-POR	Spec	Sat-Eq	TypeEq	
							Time	Memory
Needham - Schroeder (symmetric)	3	4.2s	0.39s	0.086s	59.3s	0.14s	0.006s	4.0 MB
	6	TO	TO	9m22s	TO	0.53s	0.009s	4.7 MB
	10			SO		3.7s	0.012s	5.0 MB
	14					18s	0.015s	6.9 MB
Yahalom - Lowe	3	1.0s	2.9s	0.095s	10s	0.063s	0.006s	3.8 MB
	6	MO	TO	11m20s	MO	0.26s	0.017s	4.9 MB
	10			SO		3.0s	0.015s	4.9 MB
	14					18s	0.019s	5.0 MB
Needham-Schroeder-Lowe	2	0.10s	3.8s	0.06s	28s	x	0.004s	3.1 MB
	4	1m8s	BUG	BUG	TO		0.004s	3.4 MB
	8	TO					0.007s	4.7 MB
Private Authentication	2	0.19s	1.2s	0.034s	x	x	0.004s	3.2 MB
	4	99m	TO	24.6s			0.013s	4.9 MB
	8	MO		TO			1s	37 MB
Helios	3	MO	BUG	BUG	x	x	0.005s	3.5 MB
BAC	2	4.0s	0.20s	0.032s	x	x	0.004s	2.9 MB
	3	SO	185m	2.6s			0.004s	3.1 MB
	5		TO	107m			0.005s	3.4 MB
	7			TO			0.005s	3.8 MB

TO: Time Out (>12h) MO: Memory Overflow (>64GB) SO: Stack Overflow

Fig. 11. Experimental results for the bounded case

This yields 14 sessions for symmetric-key protocols with two agents and one server, and 8 sessions for a protocol with two agents. In some cases, we further increase the number of sessions (replicating identical scenarios) to better compare tools performance. The results of our experiments are reported in Fig. 11. Note that SatEquiv fails to cover several cases because it does not handle asymmetric encryption nor else branches.

7.2 Unbounded Number of Sessions

We then compare TypeEq with Proverif. As shown in Fig. 12, the performances are similar except that ProVerif cannot prove Helios. The reason lies in the fact that Helios is actually subject to a copy attack if voters revote and ProVerif cannot properly handle processes that are executed only once. Similarly, Tamarin cannot properly handle the else branch of Helios (which models that the ballot box rejects duplicated ballots). Tamarin fails to prove that the underlying check either succeeds or fails on both sides.

Protocols	ProVerif	TypeEq
Helios	x	0.005s
Needham-Schroeder (sym)	0.23s	0.016s
Needham-Schroeder-Lowe	0.08s	0.008s
Yahalom-Lowe	0.48s	0.020s
Private Authentication	0.034s	0.008s
BAC	0.038s	0.005s

Fig. 12. Experimental results for an unbounded number of sessions

8 Conclusion and Discussion

We devise a new type system to reason about keys in the context of equivalence properties. Our new type system significantly enhances the preliminary work of [28], covering a larger class of protocols that includes key-exchange protocols, protocols with setup phases, as well as protocols that branch differently depending on the decryption key.

Our type system requires a light type annotation that can be directly inferred from the structure of the messages. As future work, we plan to develop an automatic type inference system. In our case study, the only intricate case is the Helios protocol where the user has to write a refined type that corresponds to an over-approximation of any encrypted message. We plan to explore whether such types could be inferred automatically.

We also plan to study how to add phases to our framework, in order to cover more properties (such as unlinkability). This would require to generalize our type system to account for the fact that the type of a key may depend on the phase in which it is used.

Another limitation of our type system is that it does not address processes with too dissimilar structure. While our type system goes beyond diff-equivalence, e.g. allowing else branches to be matched with then branches, we cannot prove equivalence of processes where traces of P are dynamically mapped to traces of Q, depending on the attacker's behaviour. Such cases occur for example when proving unlinkability of the biometric passport. We plan to explore how to enrich our type system with additional rules that could cover such cases, taking advantage of the modularity of the type system.

Conversely, the fact that our type system discards processes that are in equivalence shows that our type system proves something stronger than trace equivalence. Indeed, processes P and Q have to follow some form of uniformity. We could exploit this to prove stronger properties like oblivious execution, probably further restricting our typing rules, in order to prove e.g. the absence of side-channels of a certain form.

Acknowledgments. This work has been partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreements No. 645865-SPOOC and No. 771527-BROWSEC).

References

1. Machine readable travel document. Technical report 9303. International Civil Aviation Organization (2008)
2. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2001), pp. 104–115. ACM (2001)
3. Abadi, M., Fournet, C.: Private authentication. Theoret. Comput. Sci. **322**(3), 427–476 (2004)
4. Adida, B.: Helios: web-based open-audit voting. In: 17th Conference on Security Symposium, SS 2008, pp. 335–348 (2008)
5. Arapinis, M., Chothia, T., Ritter, E., Ryan, M.: Analysing unlinkability and anonymity using the applied pi calculus. In: 2nd IEEE Computer Security Foundations Symposium (CSF 2010). IEEE Computer Society Press (2010)
6. Armando, A., et al.: The AVISPA Tool for the automated validation of internet security protocols and applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_27
7. Backes, M., Catalin, H., Maffei, M.: Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. J. Comput. Secur. **22**(2), 301–353 (2014)
8. Backes, M., Hritcu, C., Maffei, M.: Automated verification of remote electronic voting protocols in the applied pi-calculus. In: 21st IEEE Computer Security Foundations Symposium, CSF 2008, pp. 195–209. IEEE Computer Society (2008)
9. Backes, M., Maffei, M., Unruh, D.: Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In: IEEE Symposium on Security and Privacy, SP 2008, pp. 202–215. IEEE Computer Society (2008)

10. Baelde, D., Delaune, S., Hirschi, L.: Partial order reduction for security protocols. In: Proceedings of the 26th International Conference on Concurrency Theory (CONCUR 2015). LIPIcs, vol. 42, pp. 497–510. Leibniz-Zentrum für Informatik (2015)
11. Basin, D., Dreier, J., Sasse, R.: Automated symbolic proofs of observational equivalence. In: 22nd ACM SIGSAC Conference on Computer and Communications Security (ACM CCS 2015), pp. 1144–1155. ACM, October 2015
12. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. ACM Trans. Program. Lang. Syst. **33**(2), 8:1–8:45 (2011)
13. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: 14th IEEE Computer Security Foundations Workshop (CSFW 2014), pp. 82–96. IEEE Computer Society, June 2001
14. Blanchet, B.: Modeling and verifying security protocols with the applied pi calculus and ProVerif. Found. Trends Priv. Secur. **1**(1–2), 1–135 (2016)
15. Blanchet, B., Abadi, M., Fournet, C.: Automated verification of selected equivalences for security protocols. J. Logic Algebraic Program. **75**(1), 3–51 (2008)
16. Bugliesi, M., Calzavara, S., Eigner, F., Maffei, M.: Resource-aware authorization policies for statically typed cryptographic protocols. In: 24th IEEE Computer Security Foundations Symposium, CSF 2011, pp. 83–98. IEEE Computer Society (2011)
17. Bugliesi, M., Calzavara, S., Eigner, F., Maffei, M.: Logical foundations of secure resource management in protocol implementations. In: Basin, D., Mitchell, J.C. (eds.) POST 2013. LNCS, vol. 7796, pp. 105–125. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36830-1_6
18. Bugliesi, M., Calzavara, S., Eigner, F., Maffei, M.: Affine refinement types for secure distributed programming. ACM Trans. Program. Lang. Syst. **37**(4), 11:1–11:66 (2015)
19. Bugliesi, M., Focardi, R., Maffei, M.: Authenticity by tagging and typing. In: 2004 ACM Workshop on Formal Methods in Security Engineering, FMSE 2004, pp. 1–12. ACM (2004)
20. Bugliesi, M., Focardi, R., Maffei, M.: Analysis of typed analyses of authentication protocols. In: 18th IEEE Workshop on Computer Security Foundations, CSFW 2005, pp. 112–125. IEEE Computer Society (2005)
21. Bugliesi, M., Focardi, R., Maffei, M.: Dynamic types for authentication. J. Comput. Secur. **15**(6), 563–617 (2007)
22. Chadha, R., Ciobâcă, Ş., Kremer, S.: Automated verification of equivalence properties of cryptographic protocols. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 108–127. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_6
23. Cheval, V.: APTE: an algorithm for proving trace equivalence. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 587–592. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_50
24. Cheval, V., Cortier, V., Delaune, S.: Deciding equivalence-based properties using constraint solving. Theoret. Comput. Sci. **492**, 1–39 (2013)
25. Cheval, V., Cortier, V., Plet, A.: Lengths may break privacy – or how to check for equivalences with length. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 708–723. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_50
26. Cortier, V., Delaune, S., Dallon, A.: SAT-Equiv: an efficient tool for equivalence properties. In: Proceedings of the 30th IEEE Computer Security Foundations Symposium (CSF 2017). IEEE Computer Society Press, August 2017

27. Cortier, V., Grimm, N., Lallemand, J., Maffei, M.: TypeEq. https://members.loria.fr/JLallemand/files/typing
28. Cortier, V., Grimm, N., Lallemand, J., Maffei, M.: A type system for privacy properties. In: 24th ACM Conference on Computer and Communications Security (CCS 2017), pp. 409–423. ACM (2017)
29. Cortier, V., Grimm, N., Lallemand, J., Maffei, M.: Equivalence properties by typing in cryptographic branching protocols. Research report, Université de Lorraine, CNRS, Inria, LORIA; TU Wien, February 2018. https://hal.archives-ouvertes.fr/hal-01715957
30. Cortier, V., Smyth, B.: Attacking and fixing Helios: an analysis of ballot secrecy. J. Comput. Secur. **21**(1), 89–148 (2013)
31. Cremers, C.J.F.: The Scyther tool: verification, falsification, and analysis of security protocols. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 414–418. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_38
32. Dawson, J., Tiu, A.: Automating open bisimulation checking for the spi-calculus. In: IEEE Computer Security Foundations Symposium (CSF 2010) (2010)
33. Delaune, S., Kremer, S., Ryan, M.D.: Verifying privacy-type properties of electronic voting protocols. J. Comput. Secur. **17**(4), 435–487 (2009)
34. Eigner, F., Maffei, M.: Differential privacy by typing in security protocols. In: 26th IEEE Computer Security Foundations Symposium, CSF 2013, pp. 272–286. IEEE Computer Society (2013)
35. Escobar, S., Meadows, C., Meseguer, J.: A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. Theoret. Comput. Sci. **367**(1–2), 162–202 (2006)
36. Focardi, R., Maffei, M.: Types for security protocols. In: Formal Models and Techniques for Analyzing Security Protocols, Cryptology and Information Security Series, chap. 7, vol. 5, pp. 143–181. IOS Press (2011)
37. Gordon, A.D., Jeffrey, A.: Authenticity by typing for security protocols. J. Comput. Secur. **11**(4), 451–519 (2003)
38. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN prover for the symbolic analysis of security protocols. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 696–701. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_48
39. Roenne, P.: Private communication (2016)
40. Santiago, S., Escobar, S., Meadows, C., Meseguer, J.: A formal definition of protocol indistinguishability and its verification using Maude-NPA. In: Mauw, S., Jensen, C.D. (eds.) STM 2014. LNCS, vol. 8743, pp. 162–177. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11851-2_11