A Type System for Privacy Properties

Véronique Cortier
CNRS, LORIA
Nancy, France
veronique.cortier@loria.fr

Niklas Grimm
TU Wien
Vienna, Austria
niklas.grimm@tuwien.ac.at

Joseph Lallemand
Inria, LORIA
Nancy, France
joseph.lallemand@loria.fr

Matteo Maffei
TU Wien
Vienna, Austria
matteo.maffei@tuwien.ac.at

ABSTRACT

Mature push button tools have emerged for checking trace properties (e.g. secrecy or authentication) of security protocols. The case of indistinguishability-based privacy properties (e.g. ballot privacy or anonymity) is more complex and constitutes an active research topic with several recent propositions of techniques and tools.

We explore a novel approach based on type systems and provide a (sound) type system for proving equivalence of protocols, for a bounded or an unbounded number of sessions. The resulting prototype implementation has been tested on various protocols of the literature. It provides a significant speed-up (by orders of magnitude) compared to tools for a bounded number of sessions and complements in terms of expressiveness other state-of-the-art tools, such as ProVerif and Tamarin: e.g., we show that our analysis technique is the first one to handle a faithful encoding of the Helios e-voting protocol in the context of an untrusted ballot box.

CCS CONCEPTS

• **Security and privacy → Formal security models**; **Logic and verification**;

KEYWORDS

Protocols; privacy; symbolic models; type systems

1 INTRODUCTION

Formal methods proved to be indispensable tools for the analysis of advanced cryptographic protocols such as those for key distribution [55], mobile payments [37], e-voting [11, 36, 44], and e-health [51]. In the last years, 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 [10] and Scyther [42]), Horn clause resolution (e.g., ProVerif [19]), term rewriting (e.g., Scyther [42] and Tamarin [52]), and type systems [12, 17, 23–28, 46, 47].

A current and very active topic is the adaptation of these techniques to the more involved case of *trace equivalence* properties. These are the natural symbolic counterpart of cryptographic indistinguishability properties, and they are at the heart of privacy properties such as ballot privacy [44], untraceability [7], differential privacy [45], or anonymity [3, 8]. They are also used to express stronger forms of confidentiality, such as strong secrecy [40] or game-based like properties [34].

Related Work. Numerous model checking-based tools have recently been proposed for the case of a bounded number of sessions, i.e., when protocols are executed a bounded number of times. These tools encompass SPEC [43], APTE [13, 31], Akiss [30], or SAT-Equiv [35]. These tools vary in the class of cryptographic primitives and the class of protocols they can consider. However, due to the complexity of the problem, they all suffer from the state explosion problem and most of them can typically analyse no more than 3-4 sessions of (relatively small) protocols, with the exception of SAT-Equiv which can more easily reach about 10 sessions. The only tools that can verify equivalence properties for an unbounded number of sessions are ProVerif [21]. Maude-NPA [54], and Tamarin [16]. ProVerif checks a property that is stronger than trace equivalence, namely diff equivalence, which works well in practice provided that protocols have a similar structure. However, as for trace properties, the internal design of ProVerif renders the tool unable to distinguish between exactly one session and infinitely many: this over-approximation often yields false attacks, in particular when the security of a protocol relies on the fact that some action is only performed once. Maude-NPA also checks diff-equivalence but often does not terminate. Tamarin can handle an unbounded number of sessions and is very flexible in terms of supported protocol classes but it often requires human interactions. Finally, some recent work has started to leverage type systems to enforce relational properties for programs, exploring this approach also in the context of cryptographic protocol implementations [14]: like ProVerif, the resulting tool is unable to distinguish between exactly one session and infinitely many, and furthermore it is only semi-automated, in that it often requires non-trivial lemmas to guide the tool and a specific programming discipline.

Many recent results have been obtained in the area of relational verification of programs [6, 15, 18, 29, 48, 50, 56, 57]. While these

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3133998

results do not target cryptographic protocols and, in particular, do not handle the semantics of cryptographic primitives or an active adversary interference with the program execution, exploring the suitability of the underlying ideas in the context of cryptographic protocols is an interesting subject of future work.

Our contribution. In this paper, we consider a novel type checking-based approach. Intuitively, a type system over-approximates protocol behavior. Due to this over-approximation, it is no longer possible to *decide* security properties but the types typically convey sufficient information to *prove* security. Extending this approach to equivalence properties is a delicate task. Indeed, two protocols P and Q are in equivalence if (roughly) any trace of P has an equivalent trace in Q (and conversely). Over-approximating behavior may not preserve equivalence.

Instead, we develop a somewhat hybrid approach: we design a type system to over-approximate the set of possible traces and we collect the set of sent messages into *constraints*. We then propose a procedure for proving (static) equivalence of the constraints. These do not only contain sent messages but also reflect internal checks made by the protocols, which is crucial to guarantee that whenever a message is accepted by P, it is also accepted by Q (and conversely).

As a result, we provide a sound type system for proving equivalence of protocols for both a bounded and an unbounded number of sessions, or a mix of both. This is particularly convenient to analyse systems where some actions are limited (e.g., no revote, or limited access to some resource). More specifically, we show that whenever two protocols P and Q are type-checked to be equivalent, then they are in trace equivalence, for the standard notion of trace equivalence [22], against a full Dolev-Yao attacker. In particular, one advantage of our approach is that it proves security directly in a security model that is similar to the ones used by the other popular tools, in contrast to many other security proofs based on type systems. Our result holds for protocols with all standard primitives (symmetric and asymetric encryption, signatures, pairs, hash), with atomic long-term keys (no fresh keys) and no private channels. Similarly to ProVerif, we need the two protocols P and Q to have a rather similar structure.

We provide a prototype implementation of our type system, that we evaluate on several protocols of the literature. In the case of a bounded number of sessions, our tool provides a significant speed-up (less than one second to analyse a dozen of sessions while other tools typically do not answer within 12 hours, with a few exceptions). To be fair, let us emphasize that these tools can *decide* equivalence while our tool checks sufficient conditions by the means of our type system. In the case of an unbounded number of sessions, the performance of our prototype tool is comparable to ProVerif. In contrast to ProVerif, our tool can consider a mix of bounded and unbounded number of sessions. As an application, we can prove for the first time ballot privacy of the well-known Helios e-voting protocol [5], without assuming a reliable channel between honest voters and the ballot box. ProVerif fails in this case as ballot privacy only holds under the assumption that honest voters vote at most once, otherwise the protocol is subject to a copy attack [53]. For similar reasons, also Tamarin fails to verify this protocol.

In most of our example, only a few straightforward type annotations were needed, such as indicated which keys are supposed to be secret or public. The case of the helios protocol is more involved and requires to describe the form of encrypted ballots that can be sent by a voter.

Our prototype, the protocol models, as well as a technical report are available online [38, 39].

2 OVERVIEW OF OUR APPROACH

In this section, we introduce the key ideas underlying our approach on a simplified version of the Helios voting protocol. Helios [5] is a verifiable voting protocol that has been used in various elections, including the election of the rector of the University of Louvain-la-Neuve. Its behavior is depicted below:

$$\begin{aligned} S &\to V_i : \quad r_i \\ V_i &\to S : \quad [\{v_i\}^{r_i, r_i'}_{\mathsf{pk}(k_s)}]_{k_i} \\ S &\to V_1, \ldots, V_n : \; v_1, \ldots, v_n \end{aligned}$$

where $\{m\}^r_{\mathsf{pk}(k)}$ denotes the asymmetric encryption of message m with the key $\mathsf{pk}(k)$ randomized with the nonce r, and $[m]_k$ denotes the signature of m with key k. v_i is a value in the set $\{0, 1\}$, which represents the candidate V_i votes for. In the first step, the voter casts her vote, encrypted with the election's public key $\mathsf{pk}(k_s)$ and then signed. Since generating a good random number is difficult for the voter's client (typically a JavaScript run in a browser), a typical trick is to input some randomness (r_i) from the server and to add it to its own randomness (r_i'). In the second step the server outputs the tally (i.e., a randomized permutation of the valid votes received in the voting phase). Note that the original Helios protocol does not assume signed ballots. Instead, voters authenticate themselves through a login mechanism. For simplicity, we abstract this authenticated channel by a signature.

A voting protocol provides vote privacy [44] if an attacker is not able to know which voter voted for which candidate. Intuitively, this can be modeled as the following trace equivalence property, which requires the attacker not to be able to distinguish A voting 0 and B voting 1 from A voting 1 and B voting 0. Notice that the attacker may control an unbounded number of voters:

$$\begin{aligned} &Voter(k_a, 0) \mid Voter(k_b, 1) \mid CompromisedVoters \mid S \\ \approx_t\ &Voter(k_a, 1) \mid Voter(k_b, 0) \mid CompromisedVoters \mid S \end{aligned}$$

Despite its simplicity, this protocol has a few interesting features that make its analysis particularly challenging. First of all, the server is supposed to discard ciphertext duplicates, otherwise a malicious eligible voter E could intercept A's ciphertext, sign it, and send it to the server [41], as exemplified below:

$$\begin{aligned} A &\to S : \quad [\{v_a\}^{r_a, r_a'}_{\mathsf{pk}(k_s)}]_{k_a} \\ E &\to S : \quad [\{v_a\}^{r_a, r_a'}_{\mathsf{pk}(k_s)}]_{k_e} \\ B &\to S : \quad [\{v_b\}^{r_b, r_b'}_{\mathsf{pk}(k_s)}]_{k_b} \\ S &\to A, B : \; v_a, v_b, v_a \end{aligned}$$

This would make the two tallied results distinguishable, thereby breaking trace equivalence since $v_a, v_b, v_a \not\approx_t v_b, v_a, v_b$

Even more interestingly, each voter is supposed to be able to vote *only once*, otherwise the same attack would apply [53] even if

the server discards ciphertext duplicates (as the randomness used by the voter in the two ballots would be different). This makes the analysis particularly challenging, and in particular out of scope of existing cryptographic protocol analyzers like ProVerif, which abstract away from the number of protocol sessions.

With our type system, we can successfully verify the aforementioned privacy property using the following types:

$$r_a : \tau_{r_a}^{\mathsf{LL},1},\ r_b : \tau_{r_b}^{\mathsf{LL},1}, r'_a : \tau_{r'_a}^{\mathsf{HH},1},\ r'_b : \tau_{r'_b}^{\mathsf{HH},1}$$

$$k_a : \mathrm{key}^{\mathsf{HH}}(\{[\![\tau_0^{\mathsf{LL},1}\,;\,\tau_1^{\mathsf{LL},1}]\!] * \mathsf{HL} * \tau_{r'_a}^{\mathsf{HH},1}\}_{k_s})$$

$$k_b : \mathrm{key}^{\mathsf{HH}}(\{[\![\tau_1^{\mathsf{LL},1}\,;\,\tau_0^{\mathsf{LL},1}]\!] * \mathsf{HL} * \tau_{r'_b}^{\mathsf{HH},1}\}_{k_s})$$

$$k_s : \mathrm{key}^{\mathsf{HH}}\left(\begin{array}{l}([\![\tau_0^{\mathsf{LL},1}\,;\,\tau_1^{\mathsf{LL},1}]\!] * \mathsf{HL} * \tau_{r'_a}^{\mathsf{HH},1}) \ \vee \\ ([\![\tau_1^{\mathsf{LL},1}\,;\,\tau_0^{\mathsf{LL},1}]\!] * \mathsf{HL} * \tau_{r'_b}^{\mathsf{HH},1})\end{array}\right)$$

We assume standard security labels: HH stands for high confidentiality and high integrity, HL for high confidentiality and low integrity, and LL for low confidentiality and low integrity (for simplicity, we omit the low confidentiality and high integrity type, since we do not need it in our examples). The type $\tau_i^{l,1}$ describes randomness of security label l produced by the randomness generator at position i in the program, which can be invoked at most once. $\tau_i^{l,\infty}$ is similar, with the difference that the randomness generator can be invoked an unbounded number of times. These types induce a partition on random values, in which each set contains at most one element or an unbounded number of elements, respectively. This turns out to be useful, as explained below, to type-check protocols, like Helios, in which the number of times messages of a certain shape are produced is relevant for the security of the protocol.

The type of k_a (resp. k_b) says that this key is supposed to encrypt 0 and 1 (resp. 1 and 0) on the left- and right-hand side of the equivalence relation, further describing the type of the randomness. The type of k_s inherits the two payload types, which are combined in disjunctive form. In fact, public key types implicitly convey an additional payload type, the one characterizing messages encrypted by the attacker: these are of low confidentiality and turn out to be the same on the left- and right-hand side. Key types are crucial to type-check the server code: we verify the signatures produced by A and B and can then use the ciphertext type derived from the type of k_a and k_b to infer after decryption the vote cast by A and B, respectively. While processing the other ballots, the server discards the ciphertexts produced with randomness matching the one used by A or B: given that these random values are used only once, we know that the remaining ciphertexts must come from the attacker and thus convey the same vote on the left- and on the right-hand side. This suffices to type-check the final output, since the two tallied results on the left- and right-hand side are the same, and thus fulfill trace equivalence.

The type system generates a set of constraints, which, if "consistent", suffice to prove that the protocol is trace equivalent. Intuitively, these constraints characterize the indistinguishability of the messages output by the process. The constraints generated for this simplified version of Helios are reported below:

$$\begin{aligned} C = \{(\{&\mathtt{sign}(\mathsf{aenc}(\langle 0, \langle x, r'_a\rangle\rangle, \mathsf{pk}(k_S)), k_a) \sim \\ &\qquad \mathtt{sign}(\mathsf{aenc}(\langle 1, \langle x, r'_a\rangle\rangle, \mathsf{pk}(k_S)), k_a), \\ &\mathsf{aenc}(\langle 0, \langle x, r'_a\rangle\rangle, \mathsf{pk}(k_S)) \sim \mathsf{aenc}(\langle 1, \langle x, r'_a\rangle\rangle, \mathsf{pk}(k_S)), \\ &\mathtt{sign}(\mathsf{aenc}(\langle 1, \langle y, r'_b\rangle\rangle, \mathsf{pk}(k_S)), k_b) \sim \\ &\qquad \mathtt{sign}(\mathsf{aenc}(\langle 0, \langle y, r'_b\rangle\rangle, \mathsf{pk}(k_S)), k_b), \\ &\mathsf{aenc}(\langle 1, \langle y, r'_b\rangle\rangle, \mathsf{pk}(k_S)) \sim \mathsf{aenc}(\langle 0, \langle y, r'_b\rangle\rangle, \mathsf{pk}(k_S))\}, \\ &[x : \mathsf{LL}, y : \mathsf{LL}])\} \end{aligned}$$

These constraints are consistent if the set of left messages of the constraints is in (static) equivalence with the set of the right messages of the constraints. This is clearly the case here, since encryption hides the content of the plaintext. Just to give an example of non-consistent constraints, consider the following ones:

$$C' = \{\{\mathsf{h}(n_1) \sim \mathsf{h}(n_2),\ \ \mathsf{h}(n_1) \sim \mathsf{h}(n_1)\}\}$$

where n_1, n_2 are two confidential nonces. While the first constraint alone is consistent, since n_1 and n_2 are of high confidentiality and the attacker cannot thus distinguish between $\mathsf{h}(n_1)$ and $\mathsf{h}(n_2)$, the two constraints all together are not consistent, since the attacker can clearly notice if the two terms output by the process are the same or not. We developed a dedicated procedure to check the consistency of such constraints.

3 FRAMEWORK

In symbolic models, security protocols are typically modeled as processes of a process algebra, such as the applied pi-calculus [2]. We present here a calculus close to [32] inspired from the calculus underlying the ProVerif tool [20].

3.1 Terms

Messages are modeled 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, ranged over by k, and an infinite set of variables $\mathcal{V}$. Cryptographic primitives are modeled through a *signature* $\mathcal{F}$, that is a set of function symbols, given with their arity (that is, the number of arguments). Here, we will consider the following signature:

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

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:

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

We also consider a set $\mathcal{C}$ of (public) constants (used as agents 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 names(t) (resp. vars(t)) the set of names (resp. variables) occurring in t. A term is *ground* if it does not contain variables.

Here, we will 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 from

$\mathcal{T}(\mathcal{F}_c \cup \mathcal{C}, \mathcal{V}, \mathcal{N} \cup \mathcal{K})$ with atomic keys, that is, a term $t \in \mathcal{T}(\mathcal{F}_c \cup \mathcal{C}, \mathcal{V}, \mathcal{N} \cup \mathcal{K})$ is a message if any subterm of t of the form $\mathsf{pk}(t')$, $\mathsf{vk}(t')$, $\mathsf{enc}(t_1, t')$, $\mathsf{aenc}(t_1, t_2)$, or $\mathsf{sign}(t_1, t')$ is such that $t' \in \mathcal{K}$ and $t_2 = \mathsf{pk}(t_2')$ with $t_2' \in \mathcal{K}$. 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 $\mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_d \cup \mathcal{C}, \mathcal{AX}, \mathcal{FN})$.

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 application of the cryptographic primitives. For example, the decryption succeeds only if the right decryption key is used. Formally, $t\downarrow$ is recursively defined as follows.

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

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

Note that the evaluation of term t succeeds only if the underlying keys are atomic and always returns a message or $\bot$. We write $t =_\downarrow t'$ if $t\downarrow = t'\downarrow$.

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 Figure 1. We identify processes up to α-renaming, i.e., capture avoiding substitution of bound names and variables, which are defined as usual. Furthermore, we assume that all bound names and variables in the process are distinct.

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

- $\mathcal{P}$ is a multiset of processes that represents the current active processes;
- $\mathcal{E}$ is a set of names, which represents the private names of the 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 already output.
- σ is a ground substitution;

The semantics of processes is given through a transition relation $\xrightarrow{\alpha}$ on the quadruples provided in Figure 2 (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 $=_\tau$.

Destructors used in processes:
$d ::= \mathsf{dec}(\cdot, k) \mid \mathsf{adec}(\cdot, k) \mid \mathsf{checksign}(\cdot, \mathsf{vk}(k)) \mid \pi_1(\cdot) \mid \pi_2(\cdot)$

Processes:

P, Q	$::=$	
	$\emptyset$	
$\mid$	$\mathsf{new}\ n.P$	for $n \in \mathcal{BN}(n \text{ bound in } P)$
$\mid$	$\mathsf{out}(M).P$	
$\mid$	$\mathsf{in}(x).P$	for $x \in \mathcal{X}(x \text{ bound in } P)$
$\mid$	$P \mid Q$	
$\mid$	$\mathsf{let}\ x = d(y)\ \mathsf{in}\ P\ \mathsf{else}\ Q$	for $x, y \in \mathcal{X}(x \text{ bound in } P)$
$\mid$	$\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$	
$\mid$	$!P$	

where M, N are messages.

Figure 1: Syntax for processes.

Intuitively, process $\mathsf{new}\ n.P$ creates a fresh nonce, stored in $\mathcal{E}$, and behaves like P. Process $\mathsf{out}(M).P$ emits M and behaves like P. Process $\mathsf{in}(x).P$ inputs any term computed by the attacker provided it evaluates as a message and then behaves like P. Process $P \mid Q$ corresponds to the parallel composition of P and Q. Process $\mathsf{let}\ x = d(y)\ \mathsf{in}\ P\ \mathsf{else}\ Q$ behaves like P in which x is replaced by $d(y)$ if $d(y)$ can be successfully evaluated and behaves like Q otherwise. Process $\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$ 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.

$$\mathrm{trace}(P) = \{(w, \mathsf{new}\ \mathcal{E}.\phi, \sigma) \mid (\emptyset; \{P\}; \emptyset; \emptyset) \xrightarrow{w}_* (\mathcal{E}; \mathcal{P}; \phi; \sigma)\}$$

Example 3.1. Consider the Helios protocol presented in Section 2. For simplicity, we describe here a simplified version with only two (honest) voters A and B and a voting server S. This (simplified) protocol can be modeled by the process:

$$\mathsf{new}\ r_a.Voter(k_a, v_a, r_a) \mid \mathsf{new}\ r_b.Voter(k_b, v_b, r_b) \mid P_S$$

where $Voter(k, v, r)$ represents voter k willing to vote for v using randomness r while P_S represents the voting server. $Voter(k, v, r)$ simply outputs a signed encrypted vote.

$$Voter(k, v, r) = \mathsf{out}(\mathsf{sign}(\mathsf{aenc}(\langle v, r\rangle, \mathsf{pk}(k_S)), k))$$

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

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

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

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

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

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

$$(\mathcal{E}; \{\mathsf{let}\ x = d(M)\ \mathsf{in}\ P\ \mathsf{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\mathcal{E}; \{Q\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{Let-Else}$$
if $M\sigma$ is ground and $d(M\sigma)\downarrow = \bot$, *i.e.* d cannot be applied to $M\sigma$

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

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

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

Figure 2: Semantics

The voting server receives ballots from A and B and then outputs the decrypted ballots, after some mixing.

$$\begin{aligned} P_S = {} & \mathsf{in}(x_1).\mathsf{in}(x_2). \\ & \mathsf{let}\ y_1 = \mathsf{checksign}(x_1, \mathsf{vk}(k_a))\ \mathsf{in} \\ & \mathsf{let}\ y_2 = \mathsf{checksign}(x_2, \mathsf{vk}(k_b))\ \mathsf{in} \\ & \mathsf{let}\ z_1 = \mathsf{adec}(y_1, k_s)\ \mathsf{in} \quad \mathsf{let}\ z_1' = \pi_1(z_1)\ \mathsf{in} \\ & \mathsf{let}\ z_2 = \mathsf{adec}(y_2, k_s)\ \mathsf{in} \quad \mathsf{let}\ z_2' = \pi_1(z_2)\ \mathsf{in} \\ & (\mathsf{out}(z_1') \mid \mathsf{out}(z_2')) \end{aligned}$$

3.3 Equivalence

When processes evolve, sent messages are stored in a substitution ϕ while private names are stored in $\mathcal{E}$. A *frame* is simply an expression of the form $\mathsf{new}\ \mathcal{E}.\phi$ where $\mathrm{dom}(\phi) \subseteq \mathcal{AX}$. We define $\mathrm{dom}(\mathsf{new}\ \mathcal{E}.\phi)$ as $\mathrm{dom}(\phi)$. Intuitively, a frame represents the knowledge of an attacker.

Intuitively, two sequences of messages are indistinguishable to an attacker if he cannot perform any test that could distinguish them. This is typically modeled 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 [32].

Definition 3.2 (Static Equivalence). Two ground frames $\mathsf{new}\ \mathcal{E}.\phi$ and $\mathsf{new}\ \mathcal{E}'.\phi'$ 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.

$$\begin{aligned} l ::= {} & \mathtt{LL} \mid \mathtt{HL} \mid \mathtt{HH} \\ T ::= {} & l \mid T * T \mid \mathrm{key}^l(T) \mid (T)_k \mid \{T\}_k \\ \mid {} & [\![\tau_n^{l,a}\ ;\ \tau_m^{l',a}]\!] \text{ with } a \in \{1, \infty\} \mid T \vee T \end{aligned}$$

Figure 3: Types for terms (selected)

Definition 3.3 (Trace Equivalence). Let P, Q be two processes. We write $P \sqsubseteq_t Q$ if for all $(s, \psi, \sigma) \in \mathrm{trace}(P)$, there exists $(s', \psi', \sigma') \in \mathrm{trace}(Q)$ such that $s =_\tau s'$ and $\psi\sigma$ and $\psi'\sigma'$ 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 behavior, since processes may input any message forged by the attacker.

Example 3.4. As explained in Section 2, ballot privacy is typically modeled as an equivalence property [44] that requires that an attacker cannot distinguish when Alice is voting 0 and Bob is voting 1 from the scenario where the two votes are swapped.

Continuing Example 3.1, ballot privacy of Helios can be expressed as follows:

$$\begin{aligned} & \mathsf{new}\ r_a.Voter(k_a, 0, r_a) \mid \mathsf{new}\ r_b.Voter(k_b, 1, r_b) \mid P_S \\ \approx_t\ & \mathsf{new}\ r_a.Voter(k_a, 1, r_a) \mid \mathsf{new}\ r_b.Voter(k_b, 0, r_b) \mid P_S \end{aligned}$$

4 TYPING

We now introduce a type system to statically check trace equivalence between processes. Our typing judgements thus capture properties of pairs of terms or processes, which we will refer to as *left* and *right* term or process, respectively.

4.1 Types

A selection of the types for messages are defined in Figure 3 and explained below. We assume three security labels (namely, HH, HL, LL), ranged over by l, whose first (resp. second) component denotes the confidentiality (resp. integrity) level. Intuitively, messages of high confidentiality cannot be learned by the attacker, while messages of high integrity cannot originate from the attacker. Pair types describe the type of their components, as usual. Type $\text{key}^l(T)$ describes keys of security level l used to encrypt (or sign) messages of type T. The type $(T)_k$ (resp. $\{T\}_k$) describes symmetric (resp. asymmetric) encryptions with key k of a message of type T. The type $\tau_i^{l,a}$ 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 i for each constant and restriction in the process. The type $\tau_i^{l,1}$ is populated by a single name, (i.e., i describes a constant or a non-replicated nonce) and $\tau_i^{l,\infty}$ is a special type, that is instantiated to $\tau_{i_j}^{l,1}$ 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 values which can be taken by 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 simply write $\tau_n^{l,a}$. Messages of type $T \vee T'$ are messages that can have type T or type T'.

4.2 Constraints

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

4.3 Typing Messages

Typing judgments are parametrized over a typing environment Γ, which is a list of mappings from names and variables to types. The typing judgement for messages is of the form the form $\Gamma \vdash M \sim N : T \rightarrow c$ 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 holds true. We present an excerpt of the typing rules for messages in Figure 4 and comment on them in the following.

Confidential nonces (i.e. nonces with label $l = \text{HH}$ or $l = \text{HL}$) are typed with their label from the typing environment. As the attacker may not observer them, they may be different in the left and the right message and we do not add any constraints (TNonce). Public terms are given type LL if they are the same in the left and the right message (TNonceL, TCstFN, TPubKey, TVKey). We require keys and variables to be the same in the two processes, deriving their type from the environment (TKey and TVar). The rule for pairs operates recursively component-wise (TPair).

For symmetric key encryptions (TEnc), we have to make sure that the payload type matches the key type (which is achieved by rule TEncH). We add the generated ciphertext to the set of constraints, because even though the attacker cannot read the plaintext, he can perform an equality check on the ciphertext that he observed. If we type an encryption with a key that is of low confidentiality (i.e., the attacker has access to it), then we need to make sure the payload is of type LL, because the attacker can simply decrypt the message and recover the plaintext (TEncL). The rules for asymmetric encryption are the same, with the only difference that we can always chose to ignore the key type and use type LL to check the payload. This allows us to type messages produced by the attacker, which has access to the public key but does not need to respect its type. Signatures are also handled similarly, the difference here is that we need to type the payload with LL even if an honest key is used, as the signature does not hide the content. The first typing rule for hashes (THash) gives them type LL and adds the term to the constraints, without looking at the arguments of the hash function: intuitively this is justified, because the hash function makes it impossible to recover the argument. The second rule (THashL) gives type LL only if we can also give type LL to the argument of the hash function, but does not add any constraints on its own, it is just passing on the constraints created for the arguments. This means we are typing the message as if the hash function would not have been applied and use the message without the hash, which is a strictly stronger result. Both rules have their applications: while the former has to be used whenever we hash a secret, the latter may be useful to avoid the creation of unnecessary constraints when hashing terms like constants or public nonces. Rule THigh states that we can give type HL to every message, which intuitively means that we can treat every message as if it were confidential. Rule TSub allows us to type messages according to the subtyping relation, which is standard and defined in Figure 5. Rule TOr allows us to give a union type to messages, if they are typable with at least one of the two types. TLR^1 and TLR^∞ are the introduction rules for refinement types, while TLR' and TLRL' are the corresponding elimination rules. Finally, TLRVar allows to derive a new refinement type for two variables for which we have singleton refinement types, by taking the left refinement of the left variable and the right refinement of the right variable. We will see application of this rule in the e-voting protocol, where we use it to combine A's vote (0 on the left, 1 on the right) and B's vote (1 on the left, 0 on the right), into a message that is the same on both sides.

4.4 Typing Processes

The typing judgement for processes is of the form $\Gamma \vdash P \sim Q \rightarrow C$ 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 assume in this section that P and Q do not contain replication and that variables and names are renamed to avoid any capture. We also assume processes to be given with type annotations for nonces.

When typing processes, the typing environment Γ is passed down and extended from the root towards the leafs of the syntax tree of the process, i.e., following the execution semantics. The generated constraints C however, are passed up from the leafs towards the root, so that at the root we get all generated constraints, modeling the attacker's global view on the process execution.

More precisely, each possible execution path of the process - there may be multiple paths because of conditionals - creates its own set of constraints c together with the typing environment

$$\frac{\Gamma(n) = \tau_n^{l,a} \qquad \Gamma(m) = \tau_m^{l,a} \qquad l \in \{\mathsf{HH}, \mathsf{HL}\}}{\Gamma \vdash n \sim m : l \to \emptyset}\ (\textsc{TNonce}) \qquad \frac{\Gamma(n) = \tau_n^{\mathsf{LL},a}}{\Gamma \vdash n \sim n : \mathsf{LL} \to \emptyset}\ (\textsc{TNonceL}) \qquad \frac{a \in C \cup \mathcal{FN}}{\Gamma \vdash a \sim a : \mathsf{LL} \to \emptyset}\ (\textsc{TCstFN})$$

$$\frac{k \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathsf{pk}(k) \sim \mathsf{pk}(k) : \mathsf{LL} \to \emptyset}\ (\textsc{TPubKey}) \qquad \frac{k \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathsf{vk}(k) \sim \mathsf{vk}(k) : \mathsf{LL} \to \emptyset}\ (\textsc{TVKey}) \qquad \frac{\Gamma(k) = T}{\Gamma \vdash k \sim k : T \to \emptyset}\ (\textsc{TKey})$$

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

$$\frac{\Gamma \vdash M \sim N : T \to c}{\Gamma \vdash \mathsf{enc}(M,k) \sim \mathsf{enc}(N,k) : (T)_k \to c}\ (\textsc{TEnc}) \qquad \frac{\Gamma \vdash M \sim N : (T)_k \to c \qquad \Gamma(k) = \mathrm{key}^{\mathsf{HH}}(T)}{\Gamma \vdash M \sim N : \mathsf{LL} \to c \cup \{M \sim N\}}\ (\textsc{TEncH})$$

$$\frac{\Gamma \vdash M \sim N : (\mathsf{LL})_k \to c \qquad \Gamma(k) = \mathrm{key}^{\mathsf{LL}}(T)}{\Gamma \vdash M \sim N : \mathsf{LL} \to c}\ (\textsc{TEncL}) \qquad \frac{\Gamma \vdash M \sim N : T \to c}{\Gamma \vdash \mathsf{aenc}(M, \mathsf{pk}(k)) \sim \mathsf{aenc}(N, \mathsf{pk}(k)) : \{T\}_k \to c}\ (\textsc{TAenc})$$

$$\frac{\Gamma \vdash M \sim N : \{T\}_k \to c \qquad \Gamma(k) = \mathrm{key}^{\mathsf{HH}}(T)}{\Gamma \vdash M \sim N : \mathsf{LL} \to c \cup \{M \sim N\}}\ (\textsc{TAencH}) \qquad \frac{\Gamma \vdash M \sim N : \{\mathsf{LL}\}_k \to c \qquad k \in \mathrm{dom}(\Gamma)}{\Gamma \vdash M \sim N : \mathsf{LL} \to c}\ (\textsc{TAencL})$$

$$\frac{\Gamma \vdash M \sim N : T \to c \qquad \Gamma \vdash M \sim N : \mathsf{LL} \to c' \qquad \Gamma(k) = \mathrm{key}^{\mathsf{HH}}(T)}{\Gamma \vdash \mathsf{sign}(M,k) \sim \mathsf{sign}(N,k) : \mathsf{LL} \to c \cup c' \cup \{\mathsf{sign}(M,k) \sim \mathsf{sign}(N,k)\}}\ (\textsc{TSignH})$$

$$\frac{\Gamma \vdash M \sim N : \mathsf{LL} \to c \qquad \Gamma(k) = \mathrm{key}^{\mathsf{LL}}(T)}{\Gamma \vdash \mathsf{sign}(M,k) \sim \mathsf{sign}(N,k) : \mathsf{LL} \to c}\ (\textsc{TSignL}) \qquad \frac{\mathrm{names}(M) \cup \mathrm{names}(N) \cup \mathrm{vars}(M) \cup \mathrm{vars}(N) \subseteq \mathrm{dom}(\Gamma) \cup \mathcal{FN}}{\Gamma \vdash \mathsf{h}(M) \sim \mathsf{h}(N) : \mathsf{LL} \to \{\mathsf{h}(M) \sim \mathsf{h}(N)\}}\ (\textsc{THash})$$

$$\frac{\Gamma \vdash M \sim N : \mathsf{LL} \to c}{\Gamma \vdash \mathsf{h}(M) \sim \mathsf{h}(N) : \mathsf{LL} \to c}\ (\textsc{THashL}) \qquad \frac{\mathrm{names}(M) \cup \mathrm{names}(N) \cup \mathrm{vars}(M) \cup \mathrm{vars}(N) \subseteq \mathrm{dom}(\Gamma) \cup \mathcal{FN}}{\Gamma \vdash M \sim N : \mathsf{HL} \to \emptyset}\ (\textsc{THigh})$$

$$\frac{\Gamma \vdash M \sim N : T' \to c \qquad T' <: T}{\Gamma \vdash M \sim N : T \to c}\ (\textsc{TSub}) \qquad \frac{\Gamma \vdash M \sim N : T \to c}{\Gamma \vdash M \sim N : T \vee T' \to c}\ (\textsc{TOr})$$

$$\frac{\begin{array}{l}\Gamma(m) = \tau_m^{l,1} \quad \text{or} \quad m \in \mathcal{FN} \cup C \ \wedge\ l = \mathsf{LL} \\ \Gamma(n) = \tau_n^{l',1} \quad \text{or} \quad n \in \mathcal{FN} \cup C \ \wedge\ l' = \mathsf{LL}\end{array}}{\Gamma \vdash m \sim n : [\![\tau_m^{l,1}; \tau_n^{l',1}]\!] \to \emptyset}\ (\mathrm{TLR}^1) \qquad \frac{\Gamma(m) = \tau_m^{l,\infty} \qquad \Gamma(n) = \tau_n^{l',\infty}}{\Gamma \vdash m \sim n : [\![\tau_m^{l,\infty}; \tau_n^{l',\infty}]\!] \to \emptyset}\ (\mathrm{TLR}^\infty)$$

$$\frac{\Gamma \vdash M \sim N : [\![\tau_m^{l,a}; \tau_n^{l,a}]\!] \to c \qquad l \in \{\mathsf{HL}, \mathsf{HH}\}}{\Gamma \vdash M \sim N : l \to c}\ (\textsc{TLR'}) \qquad \frac{\Gamma \vdash M \sim N : [\![\tau_n^{\mathsf{LL},a}; \tau_n^{\mathsf{LL},a}]\!] \to c}{\Gamma \vdash M \sim N : \mathsf{LL} \to c}\ (\textsc{TLRL'})$$

$$\frac{\Gamma \vdash x \sim x : [\![\tau_m^{l,1}; \tau_n^{l',1}]\!] \to \emptyset \qquad \Gamma \vdash y \sim y : [\![\tau_{m'}^{l'',1}; \tau_{n'}^{l''',1}]\!] \to \emptyset}{\Gamma \vdash x \sim y : [\![\tau_m^{l,1}; \tau_{n'}^{l''',1}]\!] \to \emptyset}\ (\textsc{TLRVar})$$

Figure 4: Rules for Messages

Γ that contains types for all names and variables appearing in c. Hence a *constraint set* C is a set elements of the form (c, Γ) for a set of constraints c. The typing environments are required in the constraint checking procedure, as they helps us to be more precise when checking the consistency of constraints.

An excerpt of our typing rules for processes is presented in Figure 6 and explained in the following. Rule PZero copies the current typing environment in the constraints and checks the well-formedness of the environment ($\Gamma \vdash \diamond$), which is defined as expected. Messages output on the network are possibly learned by the attacker, so they have to be of type LL (POut). The generated constraints are added to each element of the constraint set for the continuation process, using the operator $\cup_\forall$ defined as

$$C \cup_\forall c' := \left\{(c \cup c', \Gamma) \mid (c, \Gamma) \in C\right\}.$$

Conversely, messages input from the network are given type LL (PIn). Rule PNew introduces a new nonce, which may be used in the continuation processes. While typing parallel composition (PPar), we type the individual subprocesses and take the product union of the generated constraint sets as the new constraint set. The *product union* of constraint sets is defined as

$$C \cup_\times C' := \{(c \cup c', \Gamma \cup \Gamma') \mid (c, \Gamma) \in C \ \wedge\ (c', \Gamma') \in C' \ \wedge\ \Gamma, \Gamma' \text{ are compatible}\}$$

where *compatible* environments are those that agree on the type of all arguments of the shared domain. This operation models the fact that a process $P \mid P'$ can have every trace that is a combination of any trace of P with any trace of P'. The branches that are discarded due to incompatible environments correspond to impossible executions

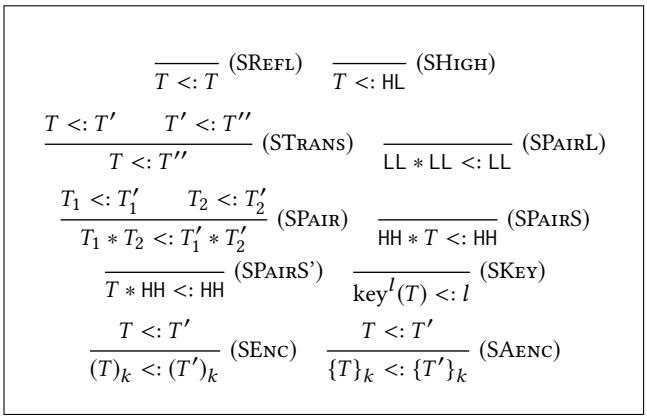

Figure 5: Subtyping Rules

(e.g., taking the left branch in P and the right branch in P' in two conditionals with the same guard). POr is the elimination rule for union types, which requires the continuation process to be well-typed with both types.

To ensure that the destructor application fails or succeeds equally in the two processes, we allow only the same destructor to be applied to the same variable in both processes (PLet). As usual, we then type-check the then as well as the else branch and then take the union of the corresponding constraints. The typing rules for destructors are presented in Figure 7. These are mostly standard: for instance, after decryption, the type of the payload is determined by the one of the decryption key, as long as this is of high integrity (DDecH). We can as well exploit strong types for ciphertexts, typically introduced by verifying a surrounding signature (see, e.g., the types for Helios) to derive the type of the payload (DDecT). In the case of public key encryption, we have to be careful, since the public encryption key is accessible to the attacker: we thus give the payload type $T \vee \mathsf{LL}$ (rule DADecH). For operations involving corrupted keys (label LL) we know that the payload is public and hence give the derived message type LL.

In the special case in which we know that the concrete value of the argument of the destructor application is a nonce or constant due to a refinement type, and we know statically that any destructor application will fail, we only need to type-check the else branch (PLetLR). As for destructor applications, the difficulty while typing conditionals is to make sure that the same branch is taken in both processes (PIfL). To ensure this we use a trick: We type both the left and the right operands of the conditional with type LL and add both generated sets of constraints to the constraint set. Intuitively, this means that the attacker could perform the equality test himself, since the guard is of type LL, which means that the conditional must take the same branch on the left and on the right. In the special case in which we can statically determine the concrete value of the terms in the conditional (because the corresponding type is populated by a singleton), we have to typecheck only the single combination of branches that will be executed (PIfLR). Another special case is if the messages on the right are of type HH and the ones on the left of type LL. As a secret of high integrity can never be equal to a public value of low integrity, we know that both processes will take the else branch (PIfS). This rule is crucial, since it may allow us to prune the low typing branch of asymmetric decryption. The last special case for conditionals is when we have a refinement type with replication for both operands of the equality check (PIfLR*). Although we know that the nonces on both sides are of the same type and hence both are elements of the same set, we cannot assume that they are equal, as the sets are infinite, unlike in rule PIfLR. Yet, concrete instantiations of nonces will have the same index for the left and the right process. This is because we check for a variant of diff-equivalence. This ensures that the equality check always yields the same result in the two processes. All these special cases highlight how a careful treatment of names in terms of equivalence classes (statically captured by types) is a powerful device to enhance the expressiveness of the analysis.

Finally, notice that we do not have any typing rule for replication: this is in line with our general idea of typing a bounded number of sessions and then extending this result to the unbounded case in the constraint checking phase, as detailed in Section 6.

5 CONSISTENCY OF CONSTRAINTS

Our type system guarantees trace equivalence of two processes only if the generated constraints are *consistent*. In this section we give a slightly simplified definition of consistency of constraints and explain how it captures the attacker's capability to distinguish processes based on their outputs.

To define consistency, we need 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).
- $\phi^\Gamma_{\mathsf{LL}}$ 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.
- Let $\mathcal{E}_\Gamma$ be the set of all nonces occurring in Γ.
- Two ground substitutions σ, σ' are well-formed in Γ if they preserve the types for variables in Γ (i.e., $\Gamma \vdash \sigma(x) \sim \sigma'(x) : \Gamma(x) \to c_x$).

Definition 5.1 (Consistency). A set of constraints c is *consistent* in an environment Γ if for all substitutions σ,σ' well-typed in Γ the frames $\mathsf{new}\ \mathcal{E}_\Gamma.(\phi^\Gamma_{\mathsf{LL}} \cup \phi_\ell(c)\sigma)$ and $\mathsf{new}\ \mathcal{E}_\Gamma.(\phi^\Gamma_{\mathsf{LL}} \cup \phi_r(c)\sigma')$ 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.

We define consistency of constraints in terms of static equivalence, as this notion exactly captures all capabilities of our attacker: to distinguish two processes, he can arbitrarily apply constructors and destructors on observed messages to create new terms, on which he can then perform equality tests or check the applicability of destructors. We require that this property holds for any well-typed substitutions, to soundly cover that fact that we do not know the content of variables statically, except for the information we get by typing. In Section 6.3 we introduce an algorithm to check consistency of constraints.

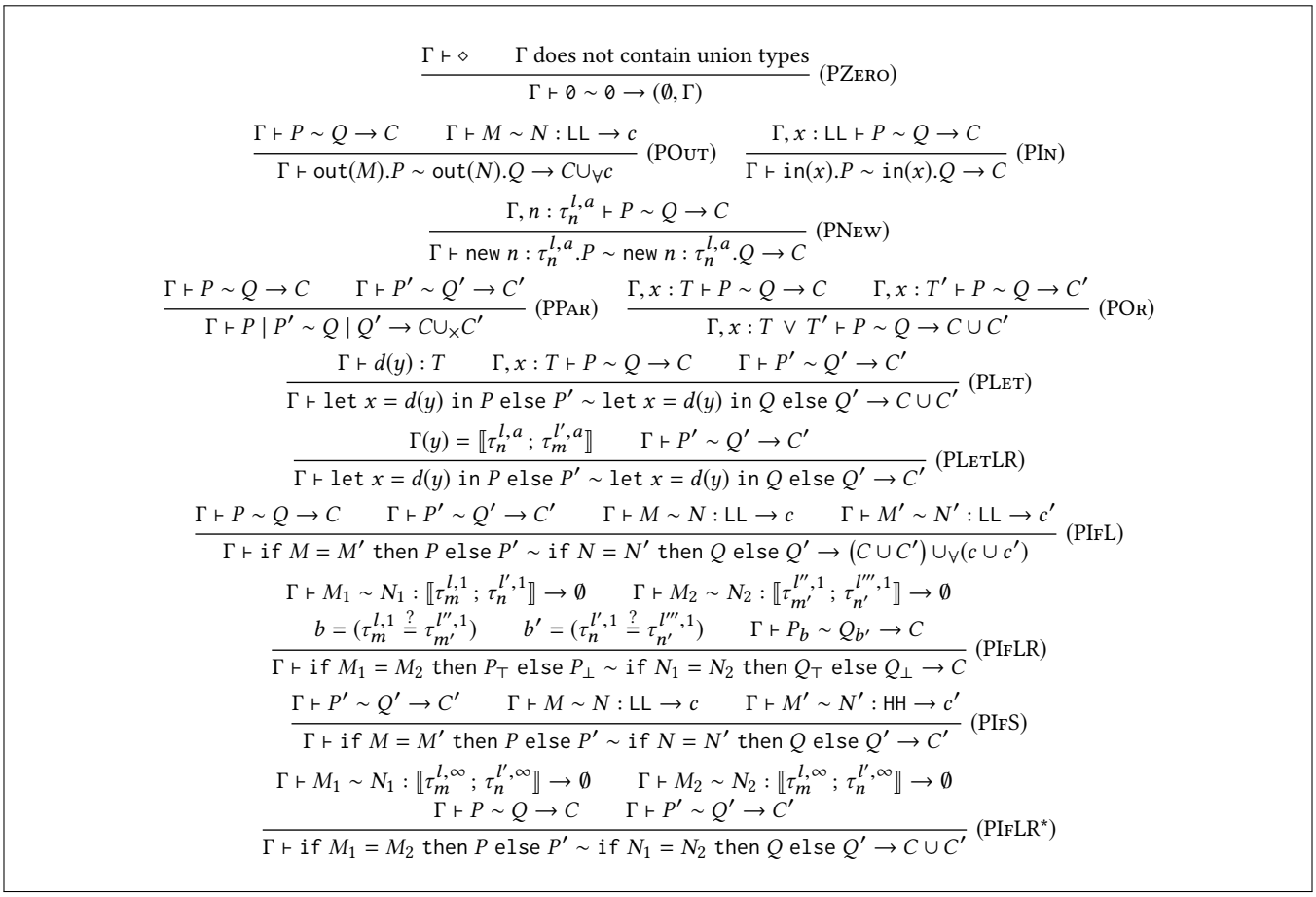

Figure 6: Rules for processes

6 MAIN RESULTS

In this section, we state our two main soundness theorems, entailing trace equivalence by typing for the bounded and unbounded case, and we explain how to automatically check consistency.

6.1 Soundness of the type system

Our type system soundly enforces trace equivalence: if we can typecheck P and Q then P and Q are equivalent, provided that the corresponding constraint set is consistent.

Theorem 6.1 (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 $P \approx_t Q$.*

To prove this theorem, we first show that typing is preserved by reduction, and guarantees that the same actions can be observed on both sides. More precisely, we show that if $\mathcal{P}$ and $\mathcal{Q}$ are multisets of processes that are pairwise typably equivalent (with consistent constraints), and if a reduction step with action α can be performed to reduce $\mathcal{P}$ into $\mathcal{P}'$, then $\mathcal{Q}$ can be reduced in one or several steps, with the same action α, to some multiset $\mathcal{Q}'$ such that the processes in $\mathcal{P}'$ and $\mathcal{Q}'$ are still typably equivalent (with consistent constraints). This is done by carefully examining all the possible typing rules used to type the processes in $\mathcal{P}$ and $\mathcal{Q}$. In addition we show that the frames of messages output when reducing $\mathcal{P}$ and $\mathcal{Q}$ are typably equivalent with consistent constraints; and that this entails their static equivalence.

This implies that if P and Q are typable with a consistent constraint, then for each trace of P, by induction on the length of the trace, there exists a trace of Q with the same sequence of actions, and with a statically equivalent frame. That is to say $P \sqsubseteq_t Q$. Similarly we show $Q \sqsubseteq_t P$, and we thus have $P \approx_t Q$.

Since we do not have typing rules for replication, Theorem 6.1 only allows us to prove equivalence of protocols for a *finite* number of sessions. An arguably surprising result, however, is that, thanks to our infinite nonce types, we can prove equivalence for an *unbounded* number of sessions, as detailed in the next section.

6.2 Typing replicated processes

For more clarity, in this section, without loss of generality we consider that for each infinite nonce type $\tau_m^{l,\infty}$ appearing in the processes, the set of names $\mathcal{BN}$ contains an infinite number of fresh names $\{m_i \mid i \in \mathbb{N}\}$ which do not appear in the processes

$$\frac{\Gamma(k) = \mathrm{key}^{\mathsf{HH}}(T) \qquad \Gamma(x) = \mathsf{LL}}{\Gamma \vdash \mathsf{dec}(x,k) : T}\ (\textsc{DDecH})$$

$$\frac{\Gamma(k) = \mathrm{key}^{\mathsf{LL}}(T) \qquad \Gamma(x) = \mathsf{LL}}{\Gamma \vdash \mathsf{dec}(x,k) : \mathsf{LL}}\ (\textsc{DDecL})$$

$$\frac{\Gamma(x) = (T)_k}{\Gamma \vdash \mathsf{dec}(x,k) : T}\ (\textsc{DDecT})$$

$$\frac{\Gamma(k) = \mathrm{key}^{\mathsf{HH}}(T) \qquad \Gamma(x) = \mathsf{LL}}{\Gamma \vdash \mathsf{adec}(x,k) : T \vee \mathsf{LL}}\ (\textsc{DAdecH})$$

$$\frac{\Gamma(k) = \mathrm{key}^{\mathsf{LL}}(T) \qquad \Gamma(x) = \mathsf{LL}}{\Gamma \vdash \mathsf{adec}(x,k) : \mathsf{LL}}\ (\textsc{DAdecL})$$

$$\frac{\Gamma(x) = \{T\}_k}{\Gamma \vdash \mathsf{adec}(x,k) : T}\ (\textsc{DAdecT})$$

$$\frac{\Gamma(k) = \mathrm{key}^{\mathsf{HH}}(T) \qquad \Gamma(x) = \mathsf{LL}}{\Gamma \vdash \mathsf{checksign}(x, \mathsf{vk}(k)) : T}\ (\textsc{DCheckH})$$

$$\frac{\Gamma(k) = \mathrm{key}^{\mathsf{LL}}(T) \qquad \Gamma(x) = \mathsf{LL}}{\Gamma \vdash \mathsf{checksign}(x, \mathsf{vk}(k)) : \mathsf{LL}}\ (\textsc{DCheckL})$$

$$\frac{\Gamma(x) = T * T'}{\Gamma \vdash \pi_1(x) : T}\ (\textsc{DFst}) \qquad \frac{\Gamma(x) = T * T'}{\Gamma \vdash \pi_2(x) : T'}\ (\textsc{DSnd})$$

$$\frac{\Gamma(x) = \mathsf{LL}}{\Gamma \vdash \pi_1(x) : \mathsf{LL}}\ (\textsc{DFstL}) \qquad \frac{\Gamma(x) = \mathsf{LL}}{\Gamma \vdash \pi_2(x) : \mathsf{LL}}\ (\textsc{DSndL})$$

Figure 7: Destructor Rules

or environments. We similarly assume that for all the variables x appearing in the processes, the set $\mathcal{X}$ of all variables also contains fresh variables $\{x_i \mid i \in \mathbb{N}\}$ which do not appear in the processes or environments.

Intuitively, whenever we can typecheck a process of the form $\mathsf{new}\ n : \tau_n^{l,1}.\ \mathsf{new}\ m : \tau_m^{l,\infty}.\ P$, we can actually typecheck

$$\mathsf{new}\ n : \tau_n^{l,1}.\ (\mathsf{new}\ m_1 : \tau_{m_1}^{l,1}.P_1 \mid \ldots \mid \mathsf{new}\ m_k : \tau_{m_k}^{l,1}.P_k)$$

where in P_i, the nonce m has been replaced by m_i and variables x have been renamed to x_i.

Formally, we denote by $[\ t\ ]_i^\Gamma$, the term t in which names n such that $\Gamma(n) = \tau_n^{l,\infty}$ for some l are replaced by n_i, and variables x are replaced by x_i.

Similarly, when a term is of type $[\![\tau_m^{l,\infty}\ ;\ \tau_p^{l',\infty}]\!]$, it can be of type $[\![\tau_{m_i}^{l,1}\ ;\ \tau_{p_i}^{l',1}]\!]$ for any i. The nonce type $\tau_m^{l,\infty}$ represents infinitely many nonces (one for each session). That is, for n sessions, the type $[\![\tau_m^{l,\infty}\ ;\ \tau_p^{l',\infty}]\!]$ represents all $[\![\tau_{m_i}^{l,1}\ ;\ \tau_{p_i}^{l',1}]\!]$. Formally, given a type T, we define its expansion to n sessions, denoted $[\ T\ ]^n$, as follows.

$$\begin{aligned}
[\ l\ ]^n &= l \\
[\ T * T'\ ]^n &= [\ T\ ]^n * [\ T'\ ]^n \\
[\ T + T'\ ]^n &= [\ T\ ]^n + [\ T'\ ]^n \\
\left[\ \mathrm{key}^l(T)\ \right]^n &= \mathrm{key}^l([\ T\ ]^n) \\
\left[\ (T)_k\ \right]^n &= ([\ T\ ]^n)_k \\
\left[\ \{T\}_k\ \right]^n &= \{[\ T\ ]^n\}_k \\
[\ T \vee T'\ ]^n &= [\ T\ ]^n \vee [\ T'\ ]^n \\
\left[\ [\![\tau_m^{l,1}\ ;\ \tau_p^{l',1}]\!]\ \right]^n &= [\![\tau_m^{l,1}\ ;\ \tau_p^{l',1}]\!] \\
\left[\ [\![\tau_m^{l,\infty}\ ;\ \tau_p^{l',\infty}]\!]\ \right]^n &= \bigvee_{j=1}^n [\![\tau_{m_j}^{l,1}\ ;\ \tau_{p_j}^{l',1}]\!]
\end{aligned}$$

where $l, l' \in \{\mathsf{LL}, \mathsf{HH}, \mathsf{HL}\}$, $k \in \mathcal{K}$. Note that the size of the expanded type $[\ T\ ]^n$ depends on n.

We need to adapt typing environments accordingly. For any typing environment Γ, we define its renaming for session i as:

$$\begin{aligned}
[\ \Gamma\ ]_i = \{x_i : T \mid \Gamma(x) = T\} &\cup \{k : T \mid \Gamma(k) = T\} \\
&\cup\ \{m : \tau_m^{l,1} \mid \Gamma(m) = \tau_m^{l,1}\} \\
&\cup\ \{m_i : \tau_{m_i}^{l,1} \mid \Gamma(m) = \tau_m^{l,\infty}\}.
\end{aligned}$$

and then its expansion to n sessions as

$$\begin{aligned}
[\ \Gamma\ ]_i^n = \{x_i : [\ T\ ]^n \mid [\ \Gamma\ ]_i(x_i) = T\} &\cup \{k : [\ T\ ]^n \mid [\ \Gamma\ ]_i(k) = T\} \\
&\cup\ \{m : \tau_m^{l,1} \mid [\ \Gamma\ ]_i(m) = \tau_m^{l,1}\}.
\end{aligned}$$

Note that in $[\ \Gamma\ ]_i^n$, due to the expansion, the size of the types depends on n.

By construction, the environments contained in the constraints generated by typing do not contain union types. However, refinement types with infinite nonce types introduce union types when expanded. In order to recover environments without union types after expanding, which, as we will explain in the next subsection, is needed for our consistency checking procedure, we define branches($[\ \Gamma\ ]_i^n$) as the set of all Γ', with the same domain as $[\ \Gamma\ ]_i^n$, such that for all x, $\Gamma'(x)$ is not a union type, and either

- $[\ \Gamma\ ]_i^n(x) = \Gamma'(x)$;
- or there exist types $T_1,\ldots,T_k,T_1',\ldots,T_{k'}'$ such that

$$[\ \Gamma\ ]_i^n(x) = T_1 \vee \ldots \vee T_k \vee \Gamma'(x) \vee T_1' \vee \ldots \vee T_{k'}'$$

Finally, when typechecking two processes containing nonces with infinite nonce types, we collect constraints that represent families of constraints.

Given a set of constraints c, and an environment Γ, we define the renaming of c for session i in Γ as $[\ c\ ]_i^\Gamma = \{[\ u\ ]_i^\Gamma \sim [\ v\ ]_i^\Gamma \mid u \sim v \in c\}$. This is propagated to constraint sets as follows: the renaming of C for session i is $[\ C\ ]_i = \{([\ c\ ]_i^\Gamma, [\ \Gamma\ ]_i) \mid (c,\Gamma) \in C\}$ and its expansion to n sessions is $[\ C\ ]_i^n = \{([\ c\ ]_i^\Gamma, \Gamma') \mid \exists \Gamma.\ (c,\Gamma) \in C \wedge \Gamma' \in \mathrm{branches}([\ \Gamma\ ]_i^n)\}$.

Again, note that the size of $[\ C\ ]_i$ does not depend on the number of sessions considered, while the size of the types present in $[\ C\ ]_i^n$ does. For example, for $C = \{(\{\mathsf{h}(x) \sim \mathsf{h}(x)\}, [x : [\![\tau_m^{\mathsf{HH},\infty}\ ;\ \tau_p^{\mathsf{HH},\infty}]\!]])\}$, we have $[\ C\ ]_i = \{(\{\mathsf{h}(x_i) \sim \mathsf{h}(x_i)\}, [x_i : [\![\tau_m^{\mathsf{HH},\infty}\ ;\ \tau_p^{\mathsf{HH},\infty}]\!]])\}$ and $[\ C\ ]_i^n = \{(\{\mathsf{h}(x_i) \sim \mathsf{h}(x_i)\}, [x_i : \bigvee_{j=1}^n [\![\tau_{m_j}^{\mathsf{HH},1}\ ;\ \tau_{p_j}^{\mathsf{HH},1}]\!]])\}$.

Our type system is sound for replicated processes provided that the collected constraint sets are consistent, when instantiated with all possible instantiations of the nonces and keys.

Theorem 6.2. *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 types of the form $\tau_n^{l,1}$.*

Assume that P and Q only bind nonces with infinite nonce types, i.e. *using* $\mathsf{new}\ m : \tau_m^{l,\infty}$ *for some label l; while P' and Q' only bind nonces with finite types,* i.e. *using* $\mathsf{new}\ m : \tau_m^{l,1}$.

Let us abbreviate by $\mathsf{new}\ \overline{n}$ *the sequence of declarations of each nonce $m \in \mathrm{dom}(\Gamma)$. If*

- $\Gamma \vdash P \sim Q \to C$,
- $\Gamma \vdash P' \sim Q' \to C'$,
- $C' \cup_\times (\cup_{\times 1 \le i \le n} [\, C \,]_i^n)$ *is consistent for all n,*

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

Theorem 6.1 requires to check consistency of one constraint set. Theorem 6.2 now requires to check consistency of an infinite family of contraint sets. Instead of *deciding* consistency, we provide a procedure that checks a slightly stronger condition.

6.3 Procedure for consistency

Checking consistency of a set of constraints amounts to checking static equivalence of the corresponding frames. Our procedure follows the spirit of [4] for checking computational indistinguishability: we first open encryption, signatures and pairs as much as possible. Note that the type of a key indicates whether it is public or secret. The two resulting frames should have the same shape. Then, for unopened components, we simply need to check that they satisfy the same equalities.

From now on, we only consider constraint sets that can actually be generated when typing processes, as these are the only ones for which we need to check consistency.

Formally, the procedure `check_const` is described in Figure 8. It consists of four steps. First, we replace variables with refinements of finite nonce types by their left and right values. In particular a variable with a union type is not associated with a single value and thus cannot be replaced. This is why the branching operation needs to be performed when expanding environments containing refinements with types of the form $\tau_n^{l,\infty}$. Second, we recursively open the constraints as much as possible. Third, we check that the resulting constraints have the same shape. Finally, as soon as two constraints $M \sim M'$ and $N \sim N'$ are such that M, N are unifiable, we must have $M' = N'$, and conversely. The condition is slightly more involved, especially when the constraints contain variables of refined types with infinite nonce types.

Example 6.3. Continuing Example 3.1, when typechecked with appropriate key types, the simplified model of Helios yields constraint sets containing notably the following two constraints.

$$\{\ \mathsf{aenc}(\langle 0, r_a\rangle, \mathsf{pk}(k_s)) \sim \mathsf{aenc}(\langle 1, r_a\rangle, \mathsf{pk}(k_s)),\\ \mathsf{aenc}(\langle 1, r_b\rangle, \mathsf{pk}(k_s)) \sim \mathsf{aenc}(\langle 0, r_b\rangle, \mathsf{pk}(k_s))\ \}$$

For simplicity, consider the set c containing only these two constraints, together with a typing environment Γ where r_a and r_b

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

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

$\mathtt{step2}_\Gamma(c)$ is recursively defined by, for all M, N, M', N':

- $\mathtt{step2}_\Gamma(\{\langle M, N\rangle \sim \langle M', N'\rangle\} \cup c') :=$ $\mathtt{step2}_\Gamma(\{M \sim M', N \sim N'\} \cup c')$
- For all $k \in \mathcal{K}$, if $\exists T.\Gamma(k) = \mathrm{key}^{\mathsf{LL}}(T)$:
  - $\mathtt{step2}_\Gamma(\{\mathsf{enc}(M, k) \sim \mathsf{enc}(M', k)\} \cup c, c') :=$ $\mathtt{step2}_\Gamma(\{M \sim M'\} \cup c')$
  - $\mathtt{step2}_\Gamma(\{\mathsf{aenc}(M, \mathsf{pk}(k)) \sim \mathsf{aenc}(M', \mathsf{pk}(k))\} \cup c, c') :=$ $\mathtt{step2}_\Gamma(\{M \sim M'\} \cup c')$
  - $\mathtt{step2}_\Gamma(\{\mathsf{sign}(M, k) \sim \mathsf{sign}(M', k)\} \cup c') :=$ $\mathtt{step2}_\Gamma(\{M \sim M'\} \cup c')$
- For all $k \in \mathcal{K}$, if $\exists T.\Gamma(k) = \mathrm{key}^{\mathsf{HH}}(T)$:

  $\mathtt{step2}_\Gamma(\{\mathsf{sign}(M, k) \sim \mathsf{sign}(M', k)\} \cup c') :=$ $\{\mathsf{sign}(M, k) \sim \mathsf{sign}(M', k)\} \cup \mathtt{step2}_\Gamma(\{M \sim M'\} \cup c')$
- For all other terms M, N:

  $\mathtt{step2}_\Gamma(\{M \sim N\} \cup c') := \{M \sim N\} \cup \mathtt{step2}_\Gamma(c')$.

$\mathtt{step3}_\Gamma(c)$:= check that for all $M \sim N \in c$, M and N are both

- a key $k \in \mathcal{K}$ such that $\exists T.\Gamma(k) = \mathrm{key}^{\mathsf{LL}}(T)$;
- nonces $m, n \in \mathcal{N}$ such that $\exists a \in \{1, \infty\}.\ \Gamma(n) = \tau_n^{\mathsf{LL},a}\ \wedge\ \Gamma(m) = \tau_n^{\mathsf{LL},a}$,
- or public keys, verification keys, or constants;
- or $\mathsf{enc}(M', k)$, $\mathsf{enc}(N', k)$ such that $\exists T.\Gamma(k) = \mathrm{key}^{\mathsf{HH}}(T)$;
- or either $\mathsf{h}(M')$, $\mathsf{h}(N')$ or $\mathsf{aenc}(M', \mathsf{pk}(k))$, $\mathsf{aenc}(N', \mathsf{pk}(k))$, where $\exists T.\Gamma(k) = \mathrm{key}^{\mathsf{HH}}(T)$; such that M' and N' contain directly under pairs some n with $\Gamma(n) = \mathsf{HH}$ or k such that $\exists T.\Gamma(k) = \mathrm{key}^{\mathsf{HH}}(T)$;
- or $\mathsf{sign}(M', k)$, $\mathsf{sign}(N', k)$ such that $\exists T.\Gamma(k) = \mathrm{key}^{\mathsf{HH}}(T)$.

$\mathtt{step4}_\Gamma(c)$:= 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 $\{x \in \mathrm{dom}(\mu) \mid \Gamma(x) = \mathsf{LL}\ \wedge\ \mu(x) \in \mathcal{N}\}$; and if the symmetric condition for the case where M', N' are unifiable holds as well, then return `true`.

`check_const`(C) := for all $(c, \Gamma) \in C$, let $c_1 := \mathtt{step2}_\Gamma(\mathtt{step1}_\Gamma(c))$ and check that $\mathtt{step3}_\Gamma(c_1) = \mathtt{true}$ and $\mathtt{step4}_\Gamma(c_1) = \mathtt{true}$.

Figure 8: Procedure for checking consistency.

are respectively given types $\tau_{r_a}^{\mathsf{HH},1}$ and $\tau_{r_b}^{\mathsf{HH},1}$, and k_s is given type $\mathsf{key}^{\mathsf{HH}}(T)$ for some T.

The procedure $\mathtt{check\_const}(\{(c,\Gamma)\})$ can detect that the constraint c is consistent and returns $\mathtt{true}$. Indeed, as c does not contain variables, $\mathtt{step1}_\Gamma(c)$ simply returns (c,Γ). Since c only contains messages encrypted with secret keys, $\mathtt{step2}_\Gamma(c)$ also leaves c unmodified. $\mathtt{step3}_\Gamma(c)$ then returns $\mathtt{true}$, since the messages appearing in c are messages asymmetrically encrypted with secret keys, which contain a secret nonce (r_a or r_b) directly under pairs. Finally $\mathtt{step4}_\Gamma(c)$ trivially returns $\mathtt{true}$, as the messages $\mathsf{aenc}(\langle 0, r_a\rangle, \mathsf{pk}(k_s))$ and $\mathsf{aenc}(\langle 1, r_b\rangle, \mathsf{pk}(k_s))$ cannot be unified, as well as the messages $\mathsf{aenc}(\langle 1, r_a\rangle, \mathsf{pk}(k_s))$ and $\mathsf{aenc}(\langle 0, r_b\rangle, \mathsf{pk}(k_s))$.

Consider now the following set c', where encryption has not been randomised:

$$c' = \{\ \mathsf{aenc}(0, \mathsf{pk}(k_s)) \sim \mathsf{aenc}(1, \mathsf{pk}(k_s)),\\ \mathsf{aenc}(1, \mathsf{pk}(k_s)) \sim \mathsf{aenc}(0, \mathsf{pk}(k_s))\ \}$$

The procedure $\mathtt{check\_const}(\{(c',\Gamma)\})$ returns $\mathtt{false}$. Indeed, contrary to the case of c, $\mathtt{step3}_\Gamma(c')$ fails, as the encrypted message do not contain a secret nonce. Actually, the corresponding frames are indeed not statically equivalent since the adversary can reconstruct the encryption of 0 and 1 with the key $\mathsf{pk}(k_s)$ (in his initial knowledge), and check for equality.

For constraint sets without infinite nonce types, $\mathtt{check\_const}$ entails consistency.

Theorem 6.4. *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 $\mathtt{check\_const}(C) = \mathtt{true}$*, then C is consistent.*

We prove this theorem by showing that, for each of the first two steps of the procedure, if $\mathtt{step}i_\Gamma(c)$ is consistent in Γ, then c is consistent in Γ. It then suffices to check the consistency of the constraint $\mathtt{step2}_\Gamma(\mathtt{step1}_\Gamma(c))$ in Γ. Provided that $\mathtt{step3}_\Gamma$ holds, we show that this constraint is saturated in the sense that any message obtained by the attacker by decomposing terms in the constraint already occurs in the constraint; and the constraint only contains messages which cannot be reconstructed by the attacker from the rest of the constraint. Using this property, we finally prove that the simple unification tests performed in $\mathtt{step4}$ are sufficient to ensure static equivalence of each side of the constraint for any well-typed instantiation of the variables.

As a direct consequence of Theorems 6.1 and 6.4, we now have a procedure to prove trace equivalence of processes without replication.

For proving trace equivalence of processes with replication, we need to check consistency of an infinite family of constraint sets, as prescribed by Theorem 6.2. As mentioned earlier, not only the number of constraints is unbounded, but the size of the type of some (replicated) variables is also unbounded (*i.e.* of the form $\bigvee_{j=1}^{n} [\![\tau_{m_j}^{l,1} ; \tau_{p_j}^{l',1}]\!]$). We use here two ingredients: we first show that it is sufficient to apply our procedure to two constraints only. Second, we show that our procedure applied to variables with replicated types, *i.e.* nonce types of the form $\tau_n^{l,\infty}$ implies consistency of the corresponding constraints with types of unbounded size.

6.4 Two constraints suffice

Consistency of a constraint set C does not guarantee consistency of $\cup_{\times 1\leq i\leq n}[\ C\ ]_i^n$. For example, consider

$$C = \{(\{\mathsf{h}(m) \sim \mathsf{h}(p)\}, [m : \tau_m^{\mathsf{HH},\infty}, p : \tau_p^{\mathsf{HH},1}])\}$$

which can be obtained when typing

$$\mathsf{new}\ m : \tau_m^{\mathsf{HH},\infty}.\ \mathsf{new}\ p : \tau_p^{\mathsf{HH},1}.\ \mathsf{out}(\mathsf{h}(m)) \sim\\ \mathsf{new}\ m : \tau_m^{\mathsf{HH},\infty}.\ \mathsf{new}\ p : \tau_p^{\mathsf{HH},1}.\ \mathsf{out}(\mathsf{h}(p)).$$

C is consistent: since m, p are secret, the attacker cannot distinguish between their hashes. However $\cup_{\times 1\leq i\leq n}[\ C\ ]_i^n$ contains (together with some environment):

$$\{\mathsf{h}(m_1) \sim \mathsf{h}(p), \mathsf{h}(m_2) \sim \mathsf{h}(p), \ldots, \mathsf{h}(m_n) \sim \mathsf{h}(p)\}$$

which is not, since the attacker can notice that the value on the right is always the same, while the value on the left is not.

Note however that the inconsistency of $\cup_{\times 1\leq i\leq n}[\ C\ ]_i^n$ would have been discovered when checking the consistency of two copies of the constraint set only. Indeed, $[\ C\ ]_1^n \cup_\times [\ C\ ]_2^n$ contains (together with some environment):

$$\{\mathsf{h}(m_1) \sim \mathsf{h}(p), \mathsf{h}(m_2) \sim \mathsf{h}(p)\}$$

which is already inconsistent, for the same reason.

Actually, checking consistency (with our procedure) of two constraints $[\ C\ ]_1^n$ and $[\ C\ ]_2^n$ entails consistency of $\cup_{\times 1\leq i\leq n}[\ C\ ]_i^n$. Note that this does not mean that consistency of $[\ C\ ]_1^n$ and $[\ C\ ]_2^n$ implies consistency of $\cup_{\times 1\leq i\leq n}[\ C\ ]_i^n$. Instead, our procedure ensures a stronger property, for which two constraints suffice.

Theorem 6.5. *Let C and C' be two constraint sets, which do not contain any common variables. For all $n \in \mathbb{N}$,*

$$\mathtt{check\_const}([\ C\ ]_1^n \cup_\times [\ C\ ]_2^n \cup_\times [\ C'\ ]_1^n) = \mathtt{true} \Rightarrow\\ \mathtt{check\_const}((\cup_{\times 1\leq i\leq n}[\ C\ ]_i^n) \cup_\times [\ C'\ ]_1^n) = \mathtt{true}.$$

To prove Theorem 6.5, we first (easily) show that if

$$\mathtt{check\_const}([\ C\ ]_1^n \cup_\times [\ C\ ]_2^n \cup_\times \big[\ C'\ \big]_1^n) = \mathtt{true},$$

then the first three steps of the procedure $\mathtt{check\_const}$ can be successfully applied to each element of $(\cup_{\times 1\leq i\leq n}[\ C\ ]_i^n) \cup_\times [\ C'\ ]_1^n$. However the case of the fourth step is more intricate. When applying the procedure $\mathtt{check\_const}$ to an element of the constraint set $(\cup_{\times 1\leq i\leq n}[\ C\ ]_i^n) \cup_\times [\ C'\ ]_1^n$, if $\mathtt{step4}$ fails, then the constraint contains an inconsistency, *i.e.* elements $M \sim M'$ and $N \sim N'$ for which the unification condition from $\mathtt{step4}$ does not hold. Then we show that we can find a similar inconsistency when considering only the first two constraint sets, *i.e.* in $[\ C\ ]_1^n \cup_\times [\ C\ ]_2^n \cup_\times [\ C'\ ]_1^n$. This is done by reindexing the nonces and variables. The proof actually requires a careful examination of the structure of the constraint set $(\cup_{\times 1\leq i\leq n}[\ C\ ]_i^n) \cup_\times [\ C'\ ]_1^n$, to establish this reindexing.

6.5 Reducing the size of types

The procedure $\mathtt{check\_const}$ applied to replicated types implies consistency of corresponding constraints with unbounded types.

Theorem 6.6. *Let C be a constraint set. Then for all i,*

$$\mathtt{check\_const}([\ C\ ]_i) = \mathtt{true} \Rightarrow\\ \forall n \geq 1.\mathtt{check\_const}([\ C\ ]_i^n) = \mathtt{true}$$

Again here, it is rather easy to show that if check_const($[\ C\ ]_i$) = true then the first three steps of the procedure check_const can successfully be applied to each element of $[\ C\ ]_i^n$. The case of step4 is more involved. The property holds thanks to the condition on the most general unifier expressed in step4. Intuitively, this condition is written in such a way that if, when applying step4 to an element of $[\ C\ ]_i^n$, two messages can be unified, then the corresponding messages (with replicated types) in $[\ C\ ]_i$ can be unified with a most general unifier μ satisfying the condition. The proof uses this idea to show that if step4 succeeds on all elements of $[\ C\ ]_i$, then it also succeeds on the elements of $[\ C\ ]_i^n$.

6.6 Checking the consistency of the infinite constraint

Theorems 6.2, 6.5, and 6.6 provide a sound procedure for checking trace equivalence of processes with and without replication.

Theorem 6.7. *Let C, and C' be two constraint sets without any common variable.*

$$\texttt{check\_const}([\ C\ ]_1 \cup_\times [\ C\ ]_2 \cup_\times [\ C'\ ]_1) = \texttt{true} \Rightarrow$$
$$\forall n.\ [\ C'\ ]_1^n \cup_\times (\cup_{\times 1 \le i \le n} [\ C\ ]_i^n) \text{ is consistent.}$$

All detailed proofs are available online [38].

7 EXPERIMENTAL RESULTS

We have implemented a prototype type-checker TypeEq and applied it on various examples briefly described below.

Symmetric key protocols. For the sake of comparison, we consider 5 symmetric key protocols taken from the benchmark of [35], and described in [33]: Denning-Sacco, Wide Mouth Frog, Needham-Schroeder, Yahalom-Lowe, and Otway-Rees. All these protocols aim at exchanging a key k. We prove strong secrecy of the key, as defined in [1], i.e., $P(k_1) \approx_t P(k_2)$ where k_1 and k_2 are public names. Intuitively, an attacker should not be able to tell which key is used even if he knows the two possible values in advance. For some of the protocols, we truncated the last step, when it consists in using the exchanged key for encryption, since our framework currently covers only encryption with long-term (fixed) keys.

Asymmetric key protocols. In addition to the symmetric key protocols, we consider the well-known Needham-Schroeder-Lowe (NSL) protocol [49] and we again prove strong secrecy of the nonce sent by the receiver (Bob).

Helios. We model the Helios protocol for two honest voters and infinitely many dishonest ones, as informally described in Section 2. The corresponding process includes a non trivial else branch, used to express the weeding phase [41], where dishonest ballots equal to some honest one are discarded. As emphasised in Section 2, Helios is secure only if honest voters vote at most once. Therefore the protocol includes non replicated processes (for voters) as well as a replicated process (to handle dishonest ballots).

All our experiments have been run on a single Intel Xeon E5-2687Wv3 3.10GHz core, with 378GB of RAM (shared with the 19 other cores). All corresponding files can be found online at [39].

7.1 Bounded number of sessions

We first compare our tool with tools designed for a bounded number of sessions: SPEC [43], APTE (and its APTE-POR variant) [13, 31], Akiss [30], or SAT-Equiv [35]. The protocol models may slightly differ due to the subtleties of each tool. For example, several of these tools require *simple* processes where each sub-process emits on a distinct channel. We do not need such an assumption. In addition, SAT-Equiv only covers symmetric encryption and therefore could not be applied to Helios or NSL. SAT-Equiv further assumes protocols to be well-typed, which sometimes requires to tag protocols. Since we consider only untagged versions (following the original description of each protocol), SAT-Equiv failed to prove the Otway-Rees protocol. Moreover, Helios involves non-trivial else branches, which are only supported by APTE.

The number of sessions we consider denotes the number of processes in parallel in each scenario. For symmetric key protocols, we start with a simple scenario with only two honest participants A, B and a honest server S (3 sessions). We consider increasingly more complex scenarios (6, 7, 10, 12, and 14 sessions) featuring a dishonest agent C. In the complete scenario (14 sessions) each agent among A, B (and C) runs the protocol once as the initiator, and once as the responder with each other agent (A, B, C). In the case of NSL, we similarly consider a scenario with two honest agents A, B running the protocol once (2 sessions), and two scenarios with an additional dishonest agent C, up to the complete scenario (8 sessions) where each agent runs NSL once as initiator, once as responder, with each agent. For Helios, we consider 2 honest voters, and one dishonest voter only, as well as a ballot box. The corresponding results are reported in Figure 9. We write TO for Time Out (12 hours), MO for Memory Out (more than 64 GB of RAM), SO for Stack Overflow, BUG in the case of APTE, when the proof failed due to bugs in the tool, and x when the tool could not handle the protocol for the reasons discussed previously. In all cases, our tool is almost instantaneous and outperforms by orders of magnitude the competitors.

7.2 Unbounded numbers of sessions

We then compare our type-checker with ProVerif [21], for an unbounded number of sessions, on three examples: Helios, Denning-Sacco, and NSL. As expected, ProVerif cannot prove Helios secure since it cannot express that voters vote only once. This may sound surprising, since proofs of Helios in ProVerif already exist (e.g. [9, 41]). Interestingly, these models actually implicitly assume a reliable channel between honest voters and the voting server: whenever a voter votes, she first sends her vote to the voting server on a secure channel, before letting the attacker see it. This model prevents an attacker from reading and blocking a message, while this can be easily done in practice (by breaking the connection). We also failed to prove (automatically) Helios in Tamarin [16]. The reason is that the weeding procedure makes Tamarin enter a loop where it cannot detect that, as soon as a ballot is not weed, it has been forged by the adversary.

For the sake of comparison, we run both tools (ProVerif and TypeEq) on a symmetric protocol (Denning-Sacco) and an asymmetric protocol (Needham-Schroeder-Lowe). The execution times are very similar. The corresponding results are reported in Figure 10.

Protocols (# sessions)		Akiss	APTE	APTE-POR	Spec	Sat-Eq	TypeEq
Denning - Sacco	3	0.08s	0.32s	0.02s	9s	0.09s	0.002s
	6	3.9s	TO	1.6s	191m	0.3s	0.003s
	7	29s		3.6s	TO	0.8s	0.004s
	10	SO		12m		1.8s	0.004s
	12			TO		3.4s	0.005s
	14					5s	0.006s
Wide Mouth Frog	3	0.03s	0.05s	0.009s	8s	0.06s	0.002s
	6	0.4s	28m	0.4s	52m	0.2s	0.003s
	7	1.4s	TO	1.9s	MO	2.3s	0.003s
	10	46s		5m31s		5s	0.004s
	12	71m		TO		1m	0.005s
	14	TO				4m20s	0.006s
Needham - Schroeder	3	0.1s	0.4s	0.02s	52s	0.5s	0.003s
	6	20s	TO	4s	MO	4s	0.003s
	7	2m		8m		36s	0.003s
	10	SO		TO		1m50s	0.005s
	12					4m47s	0.005s
	14					11m	0.007s
Yahalom - Lowe	3	0.16s	3.6s	0.03s	6s	1.4s	0.003s
	6	33s	TO	44s	132m	1m	0.004s
	7	11m		36m	MO	17m	0.004s
	10	SO		TO		63m	0.009s
	12					TO	0.04s
	14						0.05s
Otway-Rees	3	2m12s	BUG	1.7s	27m	x	0.004s
	6	TO		SO	MO		0.011s
	7						0.012s
	10						0.02s
	12						0.03s
	14						0.1s
Needham-Schroeder-Lowe	2	0.1s	4s	0.06s	31s	x	0.003s
	4	2m	BUG	BUG	MO		0.003s
	8	TO					0.007s
Helios	3	x	TO	BUG	x	x	0.002s

Figure 9: Experimental results for the bounded case

Protocols	ProVerif	TypeEq
Helios	x	0.003s
Denning-Sacco	0.05s	0.05s
Needham-Schroeder-Lowe	0.08s	0.09s

Figure 10: Experimental results for unbounded numbers of sessions

8 CONCLUSION

We presented a novel type system for verifying trace equivalence in security protocols. It can be applied to various protocols, with support for else branches, standard cryptographic primitives, as well as a bounded and an unbounded number of sessions. We believe that our prototype implementation demonstrates that this approach is promising and opens the way to the development of an efficient technique for proving equivalence properties in even larger classes of protocols.

Several interesting problems remain to be studied. For example, a limitation of ProVerif is that it cannot properly handle global states. We plan to explore this case by enriching our types to express the fact that an event is "consumed". Also, for the moment, our type system only applies to protocols P, Q that have the same structure. One advantage of a type system is its modularity: it is relatively easy to add a few rules without redoing the whole proof. We plan to add rules to cover protocols with different structures (e.g. when branches are swapped). Another direction is the treatment of primitives with algebraic properties (e.g. Exclusive Or, or homomorphic encryption). It seems possible to extend the type system and discharge the difficulty to the consistency of the constraints, which seems easier to handle (since this captures the static case). Finally, our type system is sound w.r.t. equivalence in a symbolic model. An interesting question is whether it also entails computational indistinguishability. Again, we expect that an advantage of our type system is the possibility to discharge most of the difficulty to the constraints.

Acknowledgments

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

REFERENCES

[1] Martín Abadi. 2000. Security Protocols and their Properties. In *Foundations of Secure Computation*, F Bauer and R Steinbriiggen (Eds.). NATO Science Series, Vol. for the 20th International Summer School on Foundations of Secure Computation held in Marktoberdorf Germany. IOS Press, 39–60.

[2] Martín Abadi and Cédric Fournet. 2001. Mobile Values, New Names, and Secure Communication. In *28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*. ACM, 104–115.

[3] Martín Abadi and Cédric Fournet. 2004. Private Authentication. *Theoretical Computer Science* 322, 3 (2004), 427 – 476.

[4] Martín Abadi and Phillip Rogaway. 2000. Reconciling Two Views of Cryptography. In *International Conference on Theoretical Computer Science (IFIP TCS2000)*. Springer, 3–22.

[5] Ben Adida. 2008. Helios: Web-based Open-Audit Voting. In *17th USENIX Security symposium (SS'08)*. USENIX Association, 335–348.

[6] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition Instead of Self-Composition for Proving the Absence of Timing Channels. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*. ACM, 362–375.

[7] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. 2009. Untraceability in the Applied Pi Calculus. In *1st International Workshop on RFID Security and Cryptography*. IEEE, 1–6.

[8] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. 2010. Analysing Unlinkability and Anonymity Using the Applied Pi Calculus. In *2nd IEEE Computer Security Foundations Symposium (CSF'10)*. IEEE Computer Society Press, 107–121.

[9] Myrto Arapinis, Véronique Cortier, and Steve Kremer. 2016. When Are Three Voters Enough for Privacy Properties?. In *21st European Symposium on Research in Computer Security (ESORICS'16) (Lecture Notes in Computer Science)*. Springer, Heraklion, Crete, 241–260.

[10] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuellar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. 2005. The AVISPA Tool for the automated validation of internet security protocols and applications. In *17th International Conference on Computer Aided Verification, CAV'2005 (Lecture Notes in Computer Science)*, Vol. 3576. Springer, Edinburgh, Scotland, 281–285.

[11] Michael Backes, Cătălin Hriţcu, and Matteo Maffei. 2008. Automated Verification of Remote Electronic Voting Protocols in the Applied Pi-Calculus. In *21st IEEE Computer Security Foundations Symposium (CSF '08)*. IEEE Computer Society, Washington, DC, USA, 195–209.

[12] Michael Backes, Cătălin Hriţcu, and Matteo Maffei. 2014. Union, Intersection and Refinement Types and Reasoning About Type Disjointness for Secure Protocol Implementations. *Journal of Computer Security* 22, 2 (March 2014), 301–353.

[13] David Baelde, Stéphanie Delaune, and Lucca Hirschi. 2015. Partial Order Reduction for Security Protocols. In *26th International Conference on Concurrency Theory (CONCUR'15) (LIPIcs)*, Vol. 42. Leibniz-Zentrum für Informatik, 497–510.

[14] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella Béguelin. 2014. Probabilistic Relational Verification for Cryptographic Implementations. In *41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, 193–206.

[15] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009. ACM, 90–101.

[16] David Basin, Jannik Dreier, and Ralf Sasse. 2015. Automated Symbolic Proofs of Observational Equivalence. In *22nd ACM SIGSAC Conference on Computer and Communications Security (ACM CCS 2015)*. ACM, ACM, 1144–1155.

[17] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Transactions on Programming Languages and Systems* 33, 2 (2011), 8:1–8:45.

[18] Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. ACM, New York, NY, USA, 14–25.

[19] Bruno Blanchet. 2001. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*. IEEE Computer Society, Cape Breton, Nova Scotia, Canada, 82–96.

[20] Bruno Blanchet. 2016. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends in Privacy and Security* 1, 1–2 (2016), 1–135.

[21] Bruno Blanchet, Martín Abadi, and Cédric Fournet. 2008. Automated Verification of Selected Equivalences for Security Protocols. *Journal of Logic and Algebraic Programming* 75, 1 (Feb.–March 2008), 3–51.

[22] Michele Boreale, Rocco de Nicola, and Rosario Pugliese. 2002. Proof Techniques for Cryptographic Processes. *SIAM J. Comput.* 31, 3 (2002), 947–986.

[23] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. 2011. Resource-Aware Authorization Policies for Statically Typed Cryptographic Protocols. In *24th IEEE Computer Security Foundations Symposium (CSF '11)*. IEEE Computer Society, Washington, DC, USA, 83–98.

[24] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. 2013. Logical Foundations of Secure Resource Management in Protocol Implementations. In *2nd International Conference on Principles of Security and Trust (POST 2013)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 105–125.

[25] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. 2015. Affine Refinement Types for Secure Distributed Programming. *ACM Transactions on Programming Languages and Systems* 37, 4, Article 11 (Aug. 2015), 66 pages.

[26] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. 2004. Authenticity by Tagging and Typing. In *2004 ACM Workshop on Formal Methods in Security Engineering (FMSE '04)*. ACM, New York, NY, USA, 1–12.

[27] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. 2005. Analysis of Typed Analyses of Authentication Protocols. In *18th IEEE Workshop on Computer Security Foundations (CSFW '05)*. IEEE Computer Society, Washington, DC, USA, 112–125.

[28] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. 2007. Dynamic Types for Authentication. *Journal of Computer Security* 15, 6 (Dec. 2007), 563–617.

[29] Ştefan Ciobâcă, Dorel Lucanu, Vlad Rusu, and Grigore Rosu. 2016. A language-independent proof system for full program equivalence. *Formal Asp. Comput.* 28, 3 (2016), 469–497.

[30] Rohit Chadha, Stefan Ciobâcă, and Steve Kremer. 2012. Automated Verification of Equivalence Properties of Cryptographic Protocols. In *Programming Languages and Systems —21th European Symposium on Programming (ESOP'12) (Lecture Notes in Computer Science)*, Vol. 7211. Springer, Tallinn, Estonia, 108–127.

[31] Vincent Cheval. 2014. APTE: an Algorithm for Proving Trace Equivalence. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14) (Lecture Notes in Computer Science)*, Vol. 8413. Springer, Grenoble, France, 587–592.

[32] Vincent Cheval, Véronique Cortier, and Antoine Plet. 2013. Lengths May Break Privacy – or How to Check for Equivalences With Length. In *25th International Conference on Computer Aided Verification (CAV'13) (Lecture Notes in Computer Science)*, Vol. 8043. Springer, St Petersburg, Russia, 708–723.

[33] John Clark and Jeremy Jacob. 1997. A Survey of Authentication Protocol Literature: Version 1.0. (1997).

[34] Hubert Comon-Lundh and Véronique Cortier. 2008. Computational Soundness of Observational Equivalence. In *15th ACM Conference on Computer and Communications Security (CCS'08)*. ACM Press, Alexandria, Virginia, USA, 109–118.

[35] Véronique Cortier, Stéphanie Delaune, and Antoine Dallon. 2017. SAT-Equiv: an efficient tool for equivalence properties. In *30th IEEE Computer Security Foundations Symposium (CSF'17)*. IEEE Computer Society Press.

[36] Véronique Cortier, Fabienne Eigner, Steve Kremer, Matteo Maffei, and Cyrille Wiedling. 2015. Type-Based Verification of Electronic Voting Protocols. In *4th International Conference on Principles of Security and Trust - Volume 9036*. Springer-Verlag New York, Inc., New York, NY, USA, 303–323.

[37] Veronique Cortier, Alicia Filipiak, Said Gharout, and Jacques Traore. 2017. Designing and Proving an EMV-compliant Payment Protocol for Mobile Devices. In *2nd IEEE European Symposium on Security and Privacy (EuroS&P'17)*. IEEE Computer Society, 467–480.

[38] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. 2017. A Type System for Privacy Properties (Technical Report). arXiv:1708.08340. (Aug. 2017). https://arxiv.org/abs/1708.08340

[39] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. 2017. TypeEQ. Source Code. (Aug. 2017). https://secpriv.tuwien.ac.at/tools/typeeq

[40] Véronique Cortier, Michaël Rusinowitch, and Eugen Zălinescu. 2006. *Relating Two Standard Notions of Secrecy*. Springer Berlin Heidelberg, 303–318.

[41] Véronique Cortier and Ben Smyth. 2011. Attacking and Fixing Helios: An Analysis of Ballot Secrecy. In *24th IEEE Computer Security Foundations Symposium (CSF'11)*. IEEE Computer Society Press, 297–311.

[42] Cas J. F. Cremers. 2008. The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA (Lecture Notes in Computer Science)*, Vol. 5123/2008. Springer, 414–418.

[43] Jeremy Dawson and Alwen Tiu. 2010. Automating Open Bisimulation Checking for the Spi Calculus. In *23rd IEEE Computer Security Foundations Symposium (CSF 2010)*. IEEE Computer Society, 307–321.

[44] Stéphanie Delaune, Steve Kremer, and Mark D. Ryan. 2009. Verifying Privacy-type Properties of Electronic Voting Protocols. *Journal of Computer Security* 17, 4 (2009), 435–487.

[45] Fabienne Eigner and Matteo Maffei. 2013. Differential Privacy by Typing in Security Protocols. In *26th IEEE Computer Security Foundations Symposium (CSF '13)*. IEEE Computer Society, Washington, DC, USA, 272–286.

[46] Riccardo Focardi and Matteo Maffei. 2011. Types for Security Protocols. In *Formal Models and Techniques for Analyzing Security Protocols*. Cryptology and Information Security Series, Vol. 5. IOS Press, Chapter 7, 143–181.

[47] Andrew D. Gordon and Alan Jeffrey. 2003. Authenticity by Typing for Security Protocols. *Journal of Computer Security* 11, 4 (July 2003), 451–519.

[48] Niklas Grimm, Kenji Maillard, Cédric Fournet, Cătălin Hriţcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, and Santiago Zanella-Béguelin. 2017. A Monadic Framework for Relational Verification: Applied to Information Security, Program Equivalence, and Optimizations. arXiv:1703.00055. (July 2017). https://arxiv.org/abs/1703.00055

[49] Gavin Lowe. 1996. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96) (Lecture Notes in Computer Science)*, Vol. 1055. Springer-Verlag, 147–166.

[50] Dorel Lucanu and Vlad Rusu. 2015. Program equivalence by circular reasoning. *Formal Asp. Comput.* 27, 4 (2015), 701–726.

[51] Matteo Maffei, Kim Pecina, and Manuel Reinert. 2013. Security and Privacy by Declarative Design. In *26th IEEE Computer Security Foundations Symposium (CSF '13)*. IEEE Computer Society, Washington, DC, USA, 81–96.

[52] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification, 25th International Conference, CAV 2013, Princeton, USA (Lecture Notes in Computer Science)*, Vol. 8044. Springer, 696–701.

[53] Peter Roenne. 2016. Private communication. (2016).

[54] Sonia Santiago, Santiago Escobar, Catherine A. Meadows, and José Meseguer. 2014. A Formal Definition of Protocol Indistinguishability and Its Verification Using Maude-NPA. In *STM 2014 (Lecture Notes in Computer Science)*. IEEE Computer Society, 162–177.

[55] Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David A. Basin. 2012. Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties. In *24th IEEE Computer Security Foundations Symposium (CSF'12)*. IEEE Computer Society, 78–94.

[56] Marcelo Sousa and Isil Dillig. 2016. Cartesian Hoare Logic for Verifying k-Safety Properties. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*. ACM, 57–69.

[57] Hongseok Yang. 2007. Relational Separation Logic. *Theorertical Compututer Science* 375, 1-3 (2007), 308–334.