A short version of this paper will appear in the proceedings of USENIX Security 2026.

# Secure Protocol Composition under Dynamic Corruption: Scaling Up Symbolic Analysis for Real-World Security Properties

Cas Cremers
CISPA Helmholtz Center for Information Security
cremers@cispa.de

Erik Pallas*
CISPA Helmholtz Center for Information Security
erik.pallas@cispa.de

Aleksi Peltonen
CISPA Helmholtz Center for Information Security

Version 1.0†
May 7, 2026

**Abstract**

Although automated symbolic protocol verification has proven valuable and effective, current approaches begin to reach their limits: While small protocols can be analyzed automatically, the most complex case studies often require substantial expert time and resources. There have been many attempts to solve this problem by compositional verification, but they rely on unrealistic protocol assumptions and do not support real-world security properties like Forward Secrecy.

In this work, we enable compositional symbolic analysis for real-world security protocols with respect to modern security properties. We develop a composition result in the Applied $\pi$-Calculus that holds even in the presence of attackers capable of dynamic corruption if the protocols satisfy a disjointness requirement.

We demonstrate the applicability and effectiveness of our result on the composition of a data exchange protocol with a Diffie-Hellman key exchange and a compositional analysis of Forward Secrecy in TLS 1.3 within the scope of RFC 8446 and the ECH extension. While monolithic analyses of TLS 1.3 with ECH fail to deliver a result in 12.5% of cases, all compositional analyses succeed. Additionally, runtime decreases by 71% and memory usage by 86% on average.

## 1. Introduction

Automated verification of security protocols has proven to be a very valuable and effective approach to security analysis over the last decades. From the seminal discovery of the person-in-the-middle attack on the Needham-Schroeder protocol [51, 52], the extensive usage in the standardization of TLS 1.3 [13, 36, 37] to the analysis of large and complex protocols like WPA2 [38] and SPDM [35], automated tools have led to various significant contributions.

However, the analyses of the latter case studies also show that current approaches reach their limits as protocols become ever larger and more complex. In practice, this is typically reflected in the need for increasing amounts of expert manual guidance for the tools (e.g., specifying invariants/reusable lemmas in Tamarin, recently also implemented in ProVerif).

One way to remedy this is *compositional analysis*: Instead of verifying the properties of a protocol at large, it is divided into smaller subprotocols. These subprotocols are then analyzed separately, and the combined results provide guarantees for the entire protocol. This not only allows for simpler proofs but also reusing protocol models and their analyses when investigating other protocols. Unfortunately,

*Corresponding author
†We provide a version history in Appendix C.

the composition of protocols is not secure in general. Even if two protocols are secure in isolation, an adversary can use the behavior of or knowledge gained from one protocol to attack the other.

In the context of symbolic protocol analysis, several results exist that state under which conditions secure composition is possible, covering a multitude of composition types (e.g., parallel and sequential [32] or vertical [45]), protocol classes (e.g., password-based protocols [41] or public-key infrastructures [30]), and security properties (e.g., trace properties [54] or privacy properties [4, 5]). One would therefore expect that modern protocol analyses, notably those specified in the Applied $\pi$-Calculus, such as [14, 18, 28], would use such composition results. It turns out that these composition results are in fact not used in practice. There are two main reasons for this. First, the composition results only apply to restricted protocol classes, which prevents them from being used for real-world protocols. Second, they only provide guarantees for static corruption threat models, whereas modern protocol analyses aim to prove stronger guarantees.

Consider, for example, TLS 1.3. One of its main achievements compared to its predecessors is that it ensures Forward Secrecy in all situations, i.e., even if a long-term key gets leaked, all previously exchanged data remains secret. The enforcement of Forward Secrecy in the standard has been strongly argued for (see, e.g., [6] for a related discussion on the TLS 1.3 mailing list), and it is is now best-practice that key-exchange protocols guarantee Forward Secrecy.

To analyze protocols for Forward Secrecy and similar properties we need to consider threat models with *dynamic corruption* of values, which is out of scope for the previously mentioned results. Dynamic corruption refers to an adversary's ability to obtain secrets at some point in the lifetime of the system, as opposed to static corruption, which only allows a secret to be never compromised, or always. Dynamic corruption allows us to specify stronger security guarantees, such as Forward Secrecy, that limit the temporal impact of a compromise.

Our core contribution now is a general composition result in the Applied $\pi$-Calculus that allows us to retain security, even with respect to dynamic corruption, when composing protocols. The protocols may use any cryptographic primitives that can be modeled using equational theories. Our result can be used with tools such as ProVerif [15] or SAPIC+ [31].

Additionally, whe show that unlike previous results, our result applies to real-world protocols. We illustrate our result by composing a Diffie-Hellman key exchange with a data exchange protocol and with a compositional analysis of Forward Secrecy in TLS 1.3 ECH as defined in RFC 8446 and the ECH extension. Compared to its latest state-of-the-art analysis in [14], where the authors could not prove 63% of properties, we are able to prove all properties of interest with our result. Additionally, we see improvements in time and memory consumption of up to two orders of magnitude.

*En route*, we also generalize previous results: We prove that two protocols that have no common signature, equational theory or variables can be securely composed, regardless of the cryptographic primitives used. Earlier results only allowed for concatenation, symmetric and asymmetric encryption, signatures and hash functions [34, 46].

## 1.1. Overview

Below, we first expand some of the details and restrictions of our composition result. In Section 2 we discuss related work. We give background on corruption, disjointness, and the Applied $\pi$-Calculus in Section 3. In Section 4 we define our corruption model and provide our formal definitions for composed processes and shared data, which allows us to formulate our composition result in Section 5. We show how our result can be used in practice in Section 6 and conclude in Section 7. In the appendix, we collect results from the literature we need for our proofs.

We provide the ProVerif models of our case studies and the benchmarking script in [39].

## 1.2. Composition Result Properties

We provide a composition result that allows us to compose protocols so that they retain trace properties in the presence of an attacker able to dynamically corrupt arbitrary values. Previous results were restricted to corrupting non-shared data, i.e., information that is used in only one of the protocols.

Protocols may use any cryptographic primitives representable by an equational theory even if it is not confluent, as long as the protocols do not use the same primitives. The protocols are specified in the Applied $\pi$-Calculus and unrestricted in their control flow: They may communicate over channels, feature non-trivial else-branches and incorporate replication. Other results often lack at least one of these features. The two protocols may also pass data unidirectionally: One protocol may pass data to the other, the

second protocol may not. Many previous results do not allow this form of communication or, if they do, come with other restrictions.

We achieve this in a framework that allows us to specify if and when information has been leaked to the adversary, making it possible to formalize notions like Forward Secrecy. Previous work either lacks this or at least one of the aforementioned features.

In summary, our result is the first that enables the secure composition of protocols that share and pass data in the presence of an attacker capable of dynamic corruption while having no restrictions on the control flow and the power to reference data leakage on the trace. With this, we are able to compose large classes of security protocols, e.g., key-exchange protocols with protocols using the resulting keys; these protocols can now be analyzed separately for properties like Forward Secrecy.

However, this result also comes with a few restrictions. First of all, it requires the composed protocols to have disjoint signatures, i.e., do not use the same cryptographic primitives. While this holds for many practically relevant scenarios, it just as often does not. In most cases, we can enforce disjoint signatures through the use of appropriate tagging.

Second, the theorem is only applicable to trace properties; it does not cover equivalence properties, which are often used to model privacy-type guarantees.

Third, data may only be transferred in one direction, i.e., only one of the protocols may pass information to the other. While this is sufficient in many scenarios, it would be helpful to provide input to the other protocol, e.g., pass information to a key-derivation protocol which then returns (session) keys.

Finally, on a more technical side, channel names cannot be passed between protocols. This restricts communication between processes but can often be circumvented by simply using the variables themselves as storage for shared data.

## 2. Related Work

Secure protocol composition has been investigated in both major models of security protocols: the *symbolic model*, in which cryptographic primitives are defined over abstract terms and considered to be *perfect*, and the *computational model*, in which they are defined over bitstrings and polynomial-time Turing machines.

On the symbolic side, a line of work in the *strand spaces* formalism proves that protocols can be securely composed as long as they do not share data and use different primitives; however, these primitives are limited to encryption and concatenation [46, 47]. Expanding on this, in [34] the same result is proven for an extended list of primitives, namely encryption, signatures and hash functions. The authors develop their own formalism for this.

In the framework of [3], sequential composition retains protocol-centric properties (which, e.g., include authentication but not secrecy) and a set of fixed properties like secrecy, if they are sufficiently independent from one another. This can be achieved by tagging messages with the protocols they originate from.

[32] shows that trace properties are preserved by arbitrary interleavings of protocols as long as they do not share primitives and ensure that shared keys are fresh and remain secret. The results are achieved in a less powerful version of the Applied $\pi$-Calculus, which lacks non-trivial else branches and channels; properties relying on dynamic corruption cannot be specified, as the traces used in this work only track inputs to the protocols.

In [4], a full version of the Applied $\pi$-Calculus is used to prove that under the same conditions and as long as protocols do not exchange any data, parallel composition preserves trace and equivalence properties. The result is extended to key-exchange protocols in [5]; there, protocols may communicate in a limited manner. Neither of the two works provides a model of dynamic corruption, although the results hold if non-shared variables are dynamically compromised.

The composition of public key-infrastructures and -protocols is analyzed in [30]. Several disjointness criteria for the distributed keys and modeling requirements are identified, which ensure secure composition. However, the framework used does not allow for reasoning about dynamic corruption.

The line of work [43, 45, 57, 58] investigates the composition of secure channels. The authors do not give a direct criterion for when composing a protocol using a secure channel with a protocol implementing this channel is secure, but they develop several ways to transform this problem into one or more problems of secure parallel composition. The frameworks used do not allow equational theories but instead feature a fixed set of cryptographic primitives each.

Another approach to the same problem is [29]. Here, sequential composition retains trace properties as long as messages are properly tagged.

An alternative way to define security properties are *ideal functionalities*. They are basic protocols whose behavior is considered to be correct by construction; if another protocol is a *simulation* of the ideal

functionality, i.e., it exhibits the same behavior, it is considered to possess the same properties. In this *simulation-based security* setting, it is shown in [40] that key-exchange protocols can be securely composed with key-using protocols as long as messages are tagged with a session id; as the ideal functionality does not leak any values, this result does not subsume dynamic corruption.

In [21] it is additionally shown that two processes can be composed in parallel if they do not communicate over the same channels, and that a process can be composed with itself if in each replication, all messages are tagged with a replication-unique identifier.

The *Oracle Simulation* approach of [33] enables secure sequential and parallel composition if the involved protocols are proven secure on their own against an adversary who can simulate the respective other protocol.

In the computational setting, the *Universal Composition* framework, or UC for short, also follows the simulation-based security approach [24]. There are several variations and extensions of it, e.g. [22, 25, 26, 44]. *Reactive Simulatability* [8, 9, 10], *GNUC* [49], *iUTC* [23] and *IITM* [50] are frameworks that follow a similar approach. Many computational analyses are performed in the UC framework; its composition theorem is rarely used, however, since the requirements of the theorem are difficult to satisfy.

Owl allows the specification of protocols from disjoint subprotocols and the analysis of their properties using a type-flow system [42]. However, it currently only features a fixed set of primitives and properties, only covers a very limited class of protocols, and does not allow for dynamic corruption.

DY$^\star$ is another dependent type-based verification framework built on top of the F$^\star$ programming language [11]. [12] describes a composability result for DY$^\star$ that allows the analysis of trace properties for every layer in a protocol stack. The layers have to fulfill disjointness criteria and are restricted in their usage of shared data. Every analyzed property and every layer imposes additional proof obligations, and parallel composition is not allowed. DY$^\star$ is very expressive but currently provides little to no automation.

Finally, there are several formal and automated analyses of TLS 1.3, its extensions and their respective drafts [14, 27, 36, 37]. None of them are compositional in that they rely on composability theorems to achieve their results – with one notable exception.

In their analysis of Draft 18 of the TLS 1.3 standard, the authors of [13] split their computational model of the protocol in three sub-models and derive guarantees for the full protocol via compositional arguments fully fleshed out in [19]. However, the results have been developed specifically for use with the protocol verifier CryptoVerif [16] and its framework.

# 3. Background

We first provide background on three key concepts we use: dynamic corruption, which is the threat model we assume; the type of protocol disjointness we require; and the Applied $\pi$-Calculus, the formalism in which we achieve our results.

## 3.1. Dynamic Corruption

Security properties are always relative to a threat model. Historically, it is often assumed that an adversary initially knows none of the encryption keys or other secrets owned by the parties executing a protocol. All information an adversary learns must be derived from the information learned by observing the messages exchanged by the parties.

However, this is a fairly strong assumption. In practice, an adversary might learn secrets through successful attacks on other protocols or systems, court orders, or simply user carelessness. Even if that happens, a protocol should ideally provide guarantees on past and future communication.

The literature contains multiple security properties that capture such guarantees, most notably *Forward Secrecy*, sometimes also called *Perfect Forward Secrecy*. First proposed in [48], it states that even if a long-term encryption key is leaked, all communication encrypted with a key derived from that long-term key remains secret. Forward Secrecy is a highly desired property, provided by the likes of TLS 1.3 [59] and the Signal protocol through its usage of X3DH [53].

Consider the following equivalent definition from [55].

**Definition 3.1** (Forward Secrecy)**.** *A protocol is said to have* perfect forward secrecy *if compromise of long-term keys does not compromise past session keys.*

To formally analyze protocols for Forward Secrecy we need to model that keys can be compromised during the execution of a protocol. It requires an adversary to *dynamically corrupt* any given secret; we informally capture this as follows.

**Definition 3.2** (Dynamic Corruption)**.** *Consider a protocol $P$ that uses some secret data item $d$. An adversary is capable of* dynamic corruption *of $d$ if they can gain knowledge of $d$ at any point in time during the execution of $P$.*

Dynamic corruption of $d$ is usually modeled by adding a step to a model of $P$ that sends $d$ directly to the attacker. We also follow this approach and formalize a generic transformation of $P$ into a *dynamically corrupted process* in Section 4.1.

## 3.2. Protocol Disjointness

As we have seen in Section 2, composition results typically require some kind of *disjointness*: The protocols that are composed have to work differently enough so that they cannot critically confuse any of their own messages with those of the other protocol. Often, this is achieved through either *tagging* or *disjoint primitives*.

Tagging requires that messages sent over public channels are annotated in some way. This annotation often includes at least one of the sending party, the intended recipient, or the protocol or protocol mode that generated the message. On the other hand, two protocols use disjoint primitives if no cryptographic primitive is used in both protocols.

In this work, we assume the latter. Not only does it allow for more elegant protocol models, but it is also not a stronger restriction than tagging, as tagging can be used to achieve disjoint primitives: Simply tag every message with the protocol it has been sent from.

However, using disjoint primitives does not imply that protocols have to use strictly different cryptographic primitives – it merely requires that the usage of the primitives produces sufficiently different results. More precisely, we can consider two instantiations of the same cryptographic primitive disjoint if, given the same input, they return different outputs.

The technical reason for this lies in how we model cryptographic functions. Roughly, we abstract their output as a term containing their function name and all provided input. If two functions now have different output for the same input, we change the function name to differentiate the resulting terms.

We formalize our notion of disjoint primitives in the next subsection. As an example, consider a protocol that uses two hash functions, one of which produces a 256-bit value, while the other returns a 512-bit hash. For any input $i$ they produce different results, so we can model the functions as $H_{256}$ and $H_{512}$ and their values as $H_{256}(i)$ and $H_{512}(i)$. Similarly, if one protocol step always computes *HKDF* with constant tag $t$, and another protocol step computes *HKDF* with constant tag $t'$ ($t \neq t'$), we consider the *HKDF*s disjoint and model them as two different functions $HKDF_t$ and $HKDF_{t'}$.

## 3.3. The Applied $\pi$-Calculus

In the following, we introduce the key aspects of the Applied $\pi$-Calculus [2], using the *extended processes* notion of [1].

### 3.3.1. Variables and Terms

The basic building block of our protocol models are *function symbols*, which represent cryptographic primitives. Every function symbol $f$ has an arity $\mathrm{ar}(f)$; we write $f\langle \mathrm{ar}(f) \rangle$. All function symbols used in a protocol $P$ form its *signature* $\Sigma(P)$.

We assume that $\Sigma(P)$ contains a countably infinite set of function symbols of arity 0, which we call *names* and denote $\mathrm{names}(P)$; they represent data created during an execution of a protocol. We additionally assume a countably infinite set of variables $\mathcal{V}(P)$, for which $\Sigma(P) \cap \mathcal{V}(P) = \emptyset$. The set $\mathcal{T}(\Sigma(P), \mathcal{V}(P))$ contains all terms over signature $\Sigma(P)$ and variables $\mathcal{V}(P)$; $t \in \mathcal{T}(\Sigma(P), \mathcal{V}(P))$ if it is generated by the following grammar:

| $t :=$ | TERMS |
|---|---|
| $\mid x$ | $x \in \mathcal{V}(X)$ |
| $\mid f(t_1, \ldots, t_n)$ | $f \in \Sigma(P),\ \mathrm{ar}(f) = n$ |

The root symbol of a term $t$ is denoted $\mathrm{root}(t)$; $t$ is called a *ground term* if it does not contain any variable. If $t = f(t_1, \ldots, t_n)$, $t_1, \ldots, t_n$ are *subterms* of $t$; recursively, subterms of $t_1, \ldots, t_n$ are also subterms of $t$. We will use terms to model messages sent over the network during a protocol execution.

A *substitution* $\sigma := \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ describes a replacement of variables with terms; the domain $\mathrm{dom}(\sigma) := \{x_1, \ldots, x_n\}$ contains all variables for which $\sigma$ contains a mapping. For a term $t$, $t\sigma$ describes

the term where all variables in $\mathrm{dom}(\sigma)$ have been replaced in $t$ with the corresponding terms. $\sigma$ is called a *ground substitution* if it maps every variable in $\mathrm{dom}(\sigma)$ to a ground term. To rename variables $x_1, \ldots, x_n$ in a term $t$, we perform an *$\alpha$-renaming*, written $t\{^{x_1}/_{x_1'}, \ldots, ^{x_n}/_{x_n'}\}$.

An *equational theory* $\mathrm{E}(P)$ models the properties of a cryptographic primitive from $\Sigma(P)$. It is a finite set of equations $s = t$ of terms $s, t \in \mathcal{T}(\Sigma(P), \mathcal{V}(P))$. We denote with $=_{\mathrm{E}(P)}$ the closure of $\mathrm{E}(P)$ under equivalence, substitution of variables with terms and application of function symbols according to $\mathrm{E}(P)$. If we are concerned with the union $\mathrm{E}(P) \cup \mathrm{E}(Q)$ of two equational theories, we often write $=_{\mathrm{E}}$ instead of $=_{\mathrm{E}(P)\cup\mathrm{E}(Q)}$. The equivalence class of all terms equal to $t$ according to an equational theory $E$ is denoted $[t]_{=_{\mathrm{E}}}$. Note that in this work, we will only consider non-trivial equational theories, i.e., those that do not equate all terms. If for some substitution $\sigma$ and some term $t \in \mathcal{T}(\Sigma(P), \mathcal{V}(P))$ there is another term $r \in \mathcal{T}(\Sigma(P) \setminus \mathrm{names}(P), \mathrm{dom}(\sigma))$ such that $r\sigma =_{\mathrm{E}} t$, we say that $t$ can be *deduced* from $\sigma$ and write $\sigma \vdash t$.

*Example* 3.3. Consider the modular exponentiation used in a Diffie-Hellman key exchange. We need one function of arity 2, so we have $\Sigma(DH) := \{mexp\langle 2\rangle\}$. If we assume that $g$ is a primitive root modulo some prime number $p$, we can express that the modular exponentiation is commutative with $\mathrm{E}(DH) := \{\mathtt{mexp}(\mathtt{mexp}(g, a), b) = \mathtt{mexp}(\mathtt{mexp}(g, b), a)\}$.

### 3.3.2. Syntax

The process syntax is defined by the following grammar, where $x \in \mathcal{V}(P)$ is a variable, $c$ is a name, and $t$, $M$ and $N$ are terms built over $\Sigma(P)$.

| | |
|---|---|
| $P, Q :\equiv$ | Processes |
| $\mid 0$ | Nil process |
| $\mid \nu x.P$ | Name restriction |
| $\mid \mathbf{in}(c, x).P$ | Input |
| $\mid \mathbf{out}(c, t).P$ | Output |
| $\mid$ **if** $M = N$ **then** $P$ **else** $Q$ | Case distinction |
| $\mid P \parallel Q$ | Parallel composition |
| $\mid\ !P$ | Replication |

The meaning of the respective operators is the usual; we will formalize it in Section 3.3.3. Informally speaking, the nil process does nothing; the name restriction $\nu x.P$ binds $x$ to a name in $P$. An input $\mathbf{in}(c, x).P$ reads a value from channel $c$ and stores it in variable $x$; an output $\mathbf{out}(c, t).P$ on the other hand sends the term $t$ on channel $c$. Both then continue with $P$. After a case distinction **if** $M = N$ **then** $P$ **else** $Q$ the process continues with $P$ if $M$ equals $N$ in a given equational theory, otherwise with $Q$. A parallel composition $P \parallel Q$ interleaves the execution of $P$ and $Q$ in an arbitrary order, unless they can synchronize on an input and output on the same channel. Finally, a replication $!P$ is an parallel composition of $P$ with itself, repeated arbitrarily often, possibly infinitely.

Operator precedence in ascending order is replication !, parallel composition $\parallel$, and finally sequential composition .; if in doubt, this will be made clear by the appropriate usage of round brackets $(\ldots)$. On a structural level, we consider processes equal modulo $\alpha$-renaming and the following equivalences for processes $P$, $Q$ and $R$:

- $0.P \equiv P.0$
- $P.0 \equiv P$
- $!0 \equiv 0$
- $P \parallel 0 \equiv P$
- $P \parallel Q \equiv Q \parallel P$
- $(P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R)$

Note that the above definition of a process is minimal. Many convenient features like assignments or complex case distinctions are not explicitly included but can be replicated with the given operations. In the following, $x$ and $x_i$ are variables, $c$ and $c_i$ are fresh names, $t$, $M$, $M_i$, $N$ and $N_i$ are terms, and $P$, $P_i$, $Q$ and $R$ are processes.

- Explicit assignment: $x := t :\equiv (\mathbf{in}(c, x) \parallel \mathbf{out}(c, t))$
- Inequality check: **if** $M \neq N$ **then** $P$ **else** $Q :\equiv$ **if** $M = N$ **then** $Q$ **else** $P$
- Conjunction check: **if** $\bigwedge_{i=1}^{n} M_i = N_i$ **then** $P$ **else** $Q :\equiv$ **if** $M_1 = M_2$ **then** (**if** $\bigwedge_{i=2}^{n} M_i = N_i$ **then** $P$ **else** $Q$) **else** $Q$
- Case distinction merge: (**if** $M = N$ **then** $P$ **else** $Q$).$R :\equiv$ **if** $M = N$ **then** $P.R$ **else** $Q.R$
- Parallel composition merge: $(P_1 \| \cdots \| P_n).Q :\equiv (P_1.\mathbf{out}(c_1, t) \| \cdots \| P_n.\mathbf{out}(c_n, t) \| \mathbf{in}(c_1, x_1). \ldots . \mathbf{in}(c_n, x_n).Q)$

We will use these abbreviations for modeling, but will assume for all proofs that processes consist only of basic operations.

Formalizing the disjointness requirement from Section 3.2, we say that two protocols $P$ and $Q$ built over $\Sigma(P)$ and $\Sigma(Q)$, respectively, use *disjoint primitives* if $\Sigma(P) \cap \Sigma(Q) = \emptyset$.

For a process $P$, $\mathrm{vars}(P)$ denotes the set of all variables occurring in $P$, $\mathrm{fv}(P)$ and $\mathrm{bv}(P)$ are the sets of all free and bound variables in $P$, respectively, and $\mathrm{names}(P)$ is the set of all names occurring in $P$. We also extend $\alpha$-renaming to a process $P$ by applying it to all terms occurring in $P$.

*Example* 3.4. We model a Diffie-Hellman key exchange as a process. One party binds a fresh name to a variable $a$, which represents the generation of a random number. It then sends out the modular exponentiation of some constant term $g$ and $a$ over a channel $c$, while waiting for the other party to send its half-key $B$ on channel $d$. Once $B$ has arrived, it stores the result of a modular exponentiation of $B$ and $a$ in a variable $k_A$.

$$DH_A :\equiv \nu a.(\mathbf{out}(c, \mathtt{mexp}(g, a)) \parallel \mathbf{in}(d, B)).k_A := \mathtt{mexp}(B, a)$$

Analogously, the other party generates some $b$ to use as the exponent, sends its half-key on $d$ and listens on $c$.

$$DH_B :\equiv \nu b.(\mathbf{in}(d, A) \parallel \mathbf{out}(d, \mathtt{mexp}(g, b))).k_B := \mathtt{mexp}(A, b)$$

Composing $DH_A$ and $DH_B$ in parallel yields the full model.

$$DH :\equiv DH_A \parallel DH_B$$

### 3.3.3. Semantics

We are not only interested in the process execution itself but also in whether an adversary might be able to deduce information from the communication and whether they are able to interfere with the process in unwanted ways. Therefore, we *extend* the processes with additional structures that keep track of variable assignments and adversary knowledge.

An *extended process* consists of a set $\mathcal{E}$ of *restricted names* that are unknown to the adversary, the process $P$ to execute, a substitution $\varphi$ that represents the *adversary knowledge*, and a substitution $\sigma$ that represents the *variable assignments*.

**Definition 3.5** (Extended Processes)**.** *Let $\mathcal{E}$ be a set of names, $P$ be a process such that* $\mathrm{fv}(P) = \emptyset$*, and let $\varphi$ and $\sigma$ be ground substitutions. Then an* extended process *is a 4-tuple $(\mathcal{E}, P, \varphi, \sigma)$. Instead of $(\emptyset, P, \emptyset, \emptyset)$ we will simply write $P$.*

The semantics of an extended process are given as a labeled transition relation $\xrightarrow{l}$, where $l$ can be either an *external action*, i.e., an output to or an input from a channel, or an *internal action* $\tau$ if there is no interaction with the environment.

**Definition 3.6** (Semantics of Extended Processes)**.** *Let $\mathcal{P}$ and $\mathcal{P}'$ be two extended processes. $\mathcal{P}'$ is* reachable from $\mathcal{P}$ *if $(\mathcal{P}, \mathcal{P}') \in \xrightarrow{tr}$, where $\xrightarrow{tr}$ is the reflexive transitive closure of the relation described by the following rules:*

$$\frac{M\sigma =_{\mathrm{E}} N\sigma}{(\mathcal{E}, \mathbf{if}\ M = N\ \mathbf{then}\ P\ \mathbf{else}\ Q \parallel R, \varphi, \sigma) \xrightarrow{\tau} (\mathcal{E}, P \parallel R, \varphi, \sigma)}\ \text{IF}$$

$$\frac{M\sigma \neq_{\mathrm{E}} N\sigma}{(\mathcal{E}, \mathbf{if}\ M = N\ \mathbf{then}\ P\ \mathbf{else}\ Q \parallel R, \varphi, \sigma) \xrightarrow{\tau} (\mathcal{E}, Q \parallel R, \varphi, \sigma)}\ \text{ELSE}$$

$$\frac{c \notin \mathcal{E} \quad x \notin \mathrm{dom}(\sigma) \quad M\varphi = t \quad \mathrm{vars}(M) \subseteq \mathrm{dom}(\varphi) \quad \mathrm{names}(M) \cap \mathcal{E} = \emptyset}{(\mathcal{E}, \mathbf{in}(c,x).P \parallel R, \varphi, \sigma) \xrightarrow{\mathbf{in}(c,M)} (\mathcal{E}, P \parallel R, \varphi, \sigma \cup \{x \mapsto t\})} \text{ IN}$$

$$\frac{c \notin \mathcal{E} \quad w_n \textit{ variable} \quad n = |\varphi| + 1}{(\mathcal{E}, \mathbf{out}(c,t).P \parallel R, \varphi, \sigma) \xrightarrow{\nu w_n.\mathbf{out}(c,w_n)} (\mathcal{E}, P \parallel R, \varphi \cup \{w_n \mapsto t\sigma\}, \sigma)} \text{ OUT-T}$$

$$\frac{c, h \notin \mathcal{E}}{(\mathcal{E}, \mathbf{out}(c,h).P \parallel R, \varphi, \sigma) \xrightarrow{\mathbf{out}(c,h)} (\mathcal{E}, P \parallel R, \varphi, \sigma)} \text{ OUT-CH}$$

$$\frac{c \notin \mathcal{E} \quad h \in \mathcal{E} \quad a \textit{ fresh name}}{(\mathcal{E}, \mathbf{out}(c,h).P \parallel R, \varphi, \sigma) \xrightarrow{\nu a.\mathbf{out}(c,a)} (\mathcal{E}, (P \parallel R)\{^h/_a\}, \varphi, \sigma\{^h/_a\})} \text{ OPEN-CH}$$

$$\frac{x \notin \mathrm{dom}(\sigma)}{(\mathcal{E}, \mathbf{out}(c,t).P \parallel \mathbf{in}(c,x).Q \parallel R, \varphi, \sigma) \xrightarrow{\tau} (\mathcal{E}, P \parallel Q \parallel R, \varphi, \sigma \cup \{x \mapsto t\sigma\})} \text{ SYNC}$$

$$\frac{x \notin \mathrm{dom}(\sigma) \quad n \textit{ fresh name}}{(\mathcal{E}, \nu x.P \parallel R, \varphi, \sigma) \xrightarrow{\tau} (\mathcal{E} \cup \{n\}, P \parallel R, \varphi, \sigma \cup \{x \mapsto n\})} \text{ NEW}$$

$$\frac{x' \textit{ fresh variable}}{(\mathcal{E}, !P \parallel R, \varphi, \sigma) \xrightarrow{\tau} (\mathcal{E}, P\{^x/_{x'}\}_{x \in \mathrm{bv}(P)} \parallel (!P) \parallel R, \varphi, \sigma \cup \{x' \mapsto x\sigma\}_{x \in \mathrm{bv}(P)})} \text{ REP}$$

The rules describe the following behavior:

- By THEN and ELSE, a case distinction is resolved depending on whether two given terms are equal in the equational theory of the process after applying the current state of the variable assignment.
- If two processes that reduce to an input and an output on the same channel are composed in parallel, the output value is stored in an unbound variable through application of SYNC. As this is supposed to be an internal action, it is labeled as such.
- When a process expects an input on an unrestricted channel, the adversary provides some term, which is evaluated under the current state of their knowledge and stored in an unbound variable. The input is noted through a label on the transition by IN.
- If a term is sent over an unrestricted channel, it is added to the adversary knowledge. OUT-T marks this with a label.
- If an unrestricted channel name is sent over an unrestricted channel, OUT-CH labels this accordingly.
- If a restricted channel name is sent over an unrestricted channel, the former is no longer hidden from the adversary; to encode this, OPEN-CH replaces all its subsequent occurrences with a fresh name that is not restricted. This fresh channel name is also used to label the transition.
- If an unbound variable is subject of a name restriction, it is instantiated with a fresh name by NEW. This fresh name is restricted.
- Through REP, a process under replication is instantiated with copies of all its already bound variables.

Note that in the OUT-T rule, the term $t$ is evaluated under $\sigma$ *before* being added to the adversary knowledge so that the latter does not depend on the state of $\sigma$. The adversary only learns about the state of a term at the point in time it is send out. No further evaluations of $t$ will take place later on.

Also note the SYNC, IN and NEW rules: Since the semantics assume a global scope of variables, these rules prohibit attempts of binding variables more than once by requiring that the respective variable does not already have an assignment in $\sigma$.

We write $\mathcal{P} \xrightarrow{tr} \mathcal{P}'$ instead of $(\mathcal{P}, \mathcal{P}') \in \xrightarrow{tr}$, and we write $\mathcal{P} \Rightarrow \mathcal{P}'$ if $\mathcal{P}'$ can be reached from $\mathcal{P}$ via some unspecified series of actions. The *length* $|tr|$ of a transition is the number of actions needed to perform this transition.

*Example* 3.7. The transition rules can be applied to $\mathcal{DH}$ in a plethora of different ways. For instance, a single transition suffices to generate either of the random values $a$ or $b$. For $a$, it would be the following transition.

$$\begin{aligned}\mathcal{DH} \xrightarrow{\tau} (&\{c_A, c_B, n_a\},\\ &(\mathbf{out}(c, \mathtt{mexp}(g, a)p) \parallel \mathbf{in}(d, B)).k_A := \mathtt{mexp}(B, a)p \parallel DH_B,\\ &\emptyset,\\ &\{a \mapsto n_a\})\end{aligned}$$

Going through all possible transitions could lead to various results, modeling different interferences of the adversary. If they decide to not intervene at all, every input and output is able to synchronise as intended. This would be represented by a series of transitions and an extended process as follows.

$$\begin{aligned}\mathcal{DH} \xrightarrow{l_1} \cdots \xrightarrow{l_n} (&\{c_A, c_B, n_a, n_b\},\\ &0,\\ &\emptyset,\\ &\{a \mapsto n_a, b \mapsto n_b,\\ &\ A \mapsto \mathtt{mexp}(g, a)p, B \mapsto \mathtt{mexp}(g, b)p,\\ &\ k_a \mapsto \mathtt{mexp}(\mathtt{mexp}(g, b)p, a)p,\\ &\ k_b \mapsto \mathtt{mexp}(\mathtt{mexp}(g, a)p, b)p\})\end{aligned}$$

Finally, an adversary could also block the execution of $\mathcal{DH}$ by intercepting all output and not providing any input; in doing so, they learn both the half-keys.

$$\begin{aligned}\mathcal{DH} \xrightarrow{l_1} \cdots \xrightarrow{l_m} (&\{c_A, c_B, n_a, n_b\},\\ &\mathbf{in}(d, B).k_A := \mathtt{mexp}(B, a)p \parallel \mathbf{in}(c, A).k_B := \mathtt{mexp}(A, b)p,\\ &\{w_1 \mapsto \mathtt{mexp}(g, a)p, w_2 \mapsto \mathtt{mexp}(g, b)p\},\\ &\{a \mapsto n_a, b \mapsto n_b\})\end{aligned}$$

### 3.3.4. Traces and Trace Properties

The properties of an extended process are defined relative to the labels it produces and the knowledge an adversary acquires along the way. To capture the latter, we use *frames*. A frame combines the adversary knowledge and the restricted names at a given point of an execution – essentially, all the information which the adversary has and they may not have.

**Definition 3.8** (Frame)**.** *Let $\varphi$ be a substitution, and let $\mathcal{E}$ be a finite set of names. Then a* frame *is defined as $\phi := \nu\mathcal{E}.\varphi$.*

Two frames might contain the same information modulo an equational theory, even if they are not structurally equal. For example, if an adversary learns $\mathtt{mexp}(\mathtt{mexp}(g, a), b)$ from one execution of a process and $\mathtt{mexp}(\mathtt{mexp}(g, b), a)$ from another, they have actually learnt the same thing. Frames like these are called *statically equivalent*.

**Definition 3.9** (Static Equivalence of Frames)**.** *Let $\phi_1 = \nu\mathcal{E}.\varphi_1$ and $\phi_2 = \nu\mathcal{E}.\varphi_2$ be two frames such that* $\mathrm{dom}(\varphi_1) = \mathrm{dom}(\varphi_2)$*. If for every pair of terms $M, N$ such that* $(\mathrm{names}(M) \cup \mathrm{names}(N)) \cap \mathcal{E} = \emptyset$ *it holds that $M\varphi_1 =_{\mathrm{E}} N\varphi_1 \Leftrightarrow M\varphi_2 =_{\mathrm{E}} N\varphi_2$ modulo $\alpha$-renaming, then $\phi_1$ and $\phi_2$ are* statically equivalent*, or $\phi_1 \sim \phi_2$.*

When an extended process is reduced to any form, the semantics induce a series of labels. Paired with the frame resulting from the execution, this constitutes a *trace*.

**Definition 3.10** (Trace)**.** *Let $\mathcal{P} = (\mathcal{E}, P, \varphi, \sigma)$ be an extended process. The following set contains its* traces*:*

$$\mathrm{tr}(\mathcal{P}) := \{(tr, \nu\mathcal{E}'.\varphi') \mid \exists(\mathcal{E}', P', \varphi', \sigma') \colon \mathcal{P} \xrightarrow{tr} (\mathcal{E}', P', \varphi', \sigma')\}$$

If we want to indicate that a trace $\lambda = (tr, \nu\mathcal{E}'.\varphi')$ is the result of a transition from the extended process $\mathcal{P}$ to $\mathcal{P}'$, instead of $\mathcal{P} \xrightarrow{tr} \mathcal{P}'$, we will write $\mathcal{P} \xrightarrow{\lambda} \mathcal{P}'$. We denote with $\mathrm{tr}(\mathcal{P} \Rightarrow \mathcal{P}')$ and $\mathrm{tr}(\mathcal{P} \xrightarrow{tr} \mathcal{P}')$ the

trace resulting from a transition from $\mathcal{P}$ to $\mathcal{P}'$; we will also apply this to multiple consecutive transitions, e.g., $\mathrm{tr}(\mathcal{P} \Rightarrow \mathcal{P}' \xrightarrow{tr'} \mathcal{P}'')$. Finally, we say that $tr$ is a *trace of $P$ over* $\Sigma(Q)$, or $\lambda \in \mathrm{tr}(P, \Sigma(Q))$, if $\lambda$ is a trace of $P$ when $P$ would be defined over $\Sigma(Q)$, where $\Sigma(P) \subseteq \Sigma(Q)$.

To state the properties of a trace, we extend the notion of static equivalence to traces: Two traces are statically equivalent if they have the same sequence of observable actions and if their frames are statically equivalent.

**Definition 3.11** (Static Equivalence of Traces)**.** *Let $\lambda_1 = \{tr_1, \phi_1\}$ and $\lambda_2 = \{tr_2, \phi_2\}$ be two traces. Then, $\lambda_1$ and $\lambda_2$ are* statically equivalent *if $tr_1 = tr_2$ modulo $\alpha$-renaming and $\phi_1 \sim \phi_2$.*

A *trace property* is a set of traces closed under static equivalence.

**Definition 3.12** (Trace Property)**.** *Let $P$ be a process, and let $\alpha \subseteq \mathrm{tr}(P)$ be a set of traces of $P$. If $\alpha = \{\lambda \mid \lambda \in \alpha \vee \exists \lambda' \in \alpha \colon \lambda \sim \lambda'\}$, then $\alpha$ is called a* trace property *defined over $P$.*

*A trace $\lambda \in \mathrm{tr}(P)$* satisfies *a trace property $\alpha$ defined over $P$, or $\lambda \vdash \alpha$, if and only if $\lambda \in \alpha$:*

$$\lambda \vdash \alpha :\Leftrightarrow \lambda \in \alpha$$

*An extended process $\mathcal{P}$* satisfies *a trace property $\alpha$, also denoted $\mathcal{P} \vDash \alpha$, if and only if all traces of $\mathcal{P}$ satisfy $\alpha$:*

$$\mathcal{P} \vDash \alpha :\Leftrightarrow \forall \lambda \in \mathrm{tr}(\mathcal{P}) \colon \lambda \vdash \alpha$$

*Example* 3.13. We can define a trace property over all the components of a trace. One of the key properties of $\mathcal{DH}$ is that the adversary does not learn the generated shared key. A trace property could capture this as follows:

$$\mathrm{KeySecrecy}(\mathcal{DH}) := \{(tr, \nu\mathcal{E}.\varphi) \in \mathrm{tr}(\mathcal{DH}) \mid \neg(\varphi \vdash \mathtt{mexp}(\mathtt{mexp}(g, a), b))\}$$

# 4. Corruption, Composition and Shared Data

In order to formulate our composition result, we now introduce our model of dynamic corruption. Afterwards, we describe how processes can be composed and how they can share data.

## 4.1. Corruption Model

Since the semantics do not include a way for the adversary to retrieve variable values at will, we have to encode this into the process we want to analyze. This encoding needs to achieve three things: It should be able to leak any given variable, it should be able to do so at any point in time during the execution of the process, and it should not alter the behavior of the process any further than enabling leaks.

To achieve this, we fix a set $\omega$ of *corruptible variables*. Then, whenever we bind a corruptible variable $x$ with a $\nu x$ or an $\mathbf{in}(c, x)$, we compose an $\mathbf{out}(lc, x)$ in parallel to the next step in the process. $lc$ has to be a channel that the adversary can listen to (i.e., not a restricted name); for simplicity, we will require that it does not occur in the base process at all. Since parallel composition is a nondeterministic operator, the $\mathbf{out}(lc, x)$ may be activated before any step in the process execution following the binding of $x$. Activating the $\mathbf{out}(lc, x)$ leads to the corruption of $x$: The adversary learns its value. Overall, we formalize this with the following transformation.

**Definition 4.1** (Dynamically Corrupted Process)**.** *Let $P$ be a process, $\omega \subseteq \mathrm{vars}(P)$ be a subset of variables of $P$, and let $lc \notin \mathrm{names}(P)$ be a name that is not used in $P$. Then the* dynamically corrupted process *$\varsigma(P, \omega)$* leaking on channel *$lc$ is given as follows.*

$$\varsigma(P,\omega) :\equiv \begin{cases} 0 & \text{if } P \equiv 0 \\ \nu x.(\varsigma(Q,\omega) \parallel \mathbf{out}(lc,x)) & \text{if } P \equiv \nu x.Q \wedge x \in \omega \\ \nu x.\varsigma(Q,\omega) & \text{if } P \equiv \nu x.Q \wedge x \notin \omega \\ \mathbf{in}(d,x).(\varsigma(Q,\omega) \parallel \mathbf{out}(lc,x)) & \text{if } P \equiv \mathbf{in}(d,x).Q \wedge x \in \omega \\ \mathbf{in}(d,x).\varsigma(Q,\omega) & \text{if } P \equiv \mathbf{in}(d,x).Q \wedge x \notin \omega \\ \mathbf{out}(d,t).\varsigma(Q,\omega) & \text{if } P \equiv \mathbf{out}(d,t).Q \\ \mathbf{if}\ M = N\ \mathbf{then}\ \varsigma(Q_1,\omega)\ \mathbf{else}\ \varsigma(Q_2,\omega) & \text{if } P \equiv \mathbf{if}\ M = N\ \mathbf{then}\ Q_1\ \mathbf{else}\ Q_2 \\ \varsigma(Q_1,\omega) \parallel \varsigma(Q_2,\omega) & \text{if } P \equiv Q_1 \parallel Q_2 \\ !\varsigma(Q,\omega) & \text{if } P \equiv\ !Q \end{cases}$$

Note that unbound variables are not leaked under $\varsigma$. However, unbound variables do not carry any information either way, so they can safely be ignored.

*Example* 4.2. Recall $DH$, the model of a Diffie-Hellman key exchange we developed in Section 3.3. If we want to analyze the properties it has when the resulting keys are leaked, we can turn $DH$ into a corrupted process and leak $k_A$ and $k_B$:

$$\begin{aligned}\varsigma(DH, \{k_A, k_B\}) :\equiv (\nu a.(\mathbf{out}(c, \mathtt{mexp}(g, a)) \parallel \mathbf{in}(d, B)).k_A := \mathtt{mexp}(B, a).\mathbf{out}(lc, k_A)) \\ \parallel (\nu b.(\mathbf{out}(d, \mathtt{mexp}(g, b)) \parallel \mathbf{in}(c, A)).k_B := \mathtt{mexp}(A, b).\mathbf{out}(lc, k_B))\end{aligned}$$

Note that the leaking operation is not composed in parallel, but sequentially to the derivations of $k_A$ and $k_B$, respectively. Strictly following the definition of $\varsigma(DH, \{k_A, k_B\})$, the **out** leaking $k_A$ (or $k_B$, respectively) has to be put in parallel composition with the subprocess following $k_A := \mathtt{mexp}(B, a)$ – which is 0. The resulting $k_A := \mathtt{mexp}(B, a).(0 \parallel \mathbf{out}(lc, k_A))$ then is structurally equivalent to $k_A := \mathtt{mexp}(B, a).\mathbf{out}(lc, k_A)$.

Of course, this is especially interesting if $DH$ and its keys are actually used by another protocol. We will see in the next section how $DH$ can be composed with other protocols.

## 4.2. Composed Processes and Shared Data

Next, we describe how two protocols can be composed and how they share data. We are interested in two kinds of shared data: *pre-shared data*, which is assumed to be distributed amongst all participants before protocol execution, and *transferred data*, which is sent from one protocol to another.

### 4.2.1. Pre-shared Data and Composition Contexts

If some value is already fixed before the start of a protocol execution, we call it *pre-shared data*. To model it, we can leave the corresponding variables unbound and instantiate them in a *composition context* instead. As in [4, 5], a composition context is essentially a process restricted to $\nu x$ bindings and replication !, with exactly one hole that the composed process will fill. As such, it also describes the way in which processes are composed, e.g., by putting a process under replication if it is supposed to repeatedly generate a session key.

**Definition 4.3** (Composition Context)**.** *A* composition context $C$ *is defined by the following grammar for a variable* $x$*:*

$$C :\equiv \ _ \mid \nu x.C \mid \ !C$$

*For a process* $P$*,* $C[P]$ *is the* filled composition context*.*

The shorthands defined for processes in Section 3.3.2 carry over to composition contexts by ignoring the hole; e.g., $\mathrm{vars}(\nu x._) = \{x\}$. To make explicit that $\mathrm{vars}(C)$ represents pre-shared data, we will also write $\mathrm{psh}(C)$ and define $\mathrm{psh}(C[P]) := \mathrm{psh}(C)$ for a process $P$.

*Example* 4.4. In $DH$, the exact value of the generator $g$ is of no great interest. We can thus model $g$ as an unbound variable that is instantiated by a simple composition context $C :\equiv \nu g._$. This context can then be filled by a $DH$ key exchange.

$$C[DH] \equiv \nu g.(DH_A \parallel DH_B)$$

### 4.2.2. Transferred Data and Executables

Often, not all data shared by two protocols is fixed from the very beginning, but it is generated by one protocol and transferred to the other. This *transferred data* can also be modeled by unbound variables. If we add a placeholder to the process, we can later replace it with another process that binds the transferred data.

**Definition 4.5** (Partial Process)**.** *Let* $x \in \mathcal{V}()$ *be a variable,* $c$ *a name, and* $t$*,* $M$ *and* $N$ *terms built over*

$\Sigma(P)$. *Then a* partial process *is defined by the following grammar.*

| | |
|---|---|
| $P[], Q[] :\equiv$ | PARTIAL PROCESS |
| $\mid 0$ | Nil process |
| $\mid _.P[]$ | Hole |
| $\mid \nu x.P[]$ | Name restriction |
| $\mid \mathbf{in}(c, x).P[]$ | Input |
| $\mid \mathbf{out}(c, t).P[]$ | Output |
| $\mid$ **if** $M = N$ **then** $P[]$ **else** $Q[]$ | Case distinction |
| $\mid P[] \parallel Q[]$ | Parallel composition |
| $\mid\ !P[]$ | Replication |

At its core, a partial process is a process with holes that can be filled by other processes. For simplicity, we will assume that every partial process has exactly one hole. The unbound variables of a partial process $P[]$ and a composition context $C$ represent transferred data and will be denoted by $\mathrm{trf}(C[P[]]) := \mathrm{fv}(C[P[]])$. When we combine $C$ and $P[]$ (and $Q$), we call the union of pre-shared and transferred data *shared data* and write $\mathrm{sh}(C[P[Q]]) := \mathrm{psh}(C) \cup \mathrm{trf}(C[P[]])$.

*Example* 4.6. We model a basic data exchange protocol. Consider the equational theory $\mathrm{E}(XP) := \{\mathtt{sdec}(\mathtt{senc}(x, k), k) = x\}$ and the following processes.

$$XP_A :\equiv \nu data.\mathbf{out}(xc, \mathtt{senc}(data, k_A^{\mathrm{sess}}))$$
$$XP_B :\equiv \mathbf{in}(xc, t).x := \mathtt{sdec}(t, k_B^{\mathrm{sess}})$$

$A$ generates a data item, encrypts it with a session key, and sends it on channel $xc$. $B$ listens to this channel, retrieves the encrypted message, and stores its decryption in variable $x$.

The session key can be provided by any key exchange protocol. We thus combine $XP_A$ and $XP_B$ to a partial process that could be completed by $DH$.

$$XP :\equiv\ !_.(XP_A \parallel XP_B)$$

Many protocols require a certain output from other protocols they use; e.g., for $XP$ to work as intended, it needs symmetric encryption keys. $DH$ thus has to guarantee that $k_A =_{\mathrm{E}} k_B$. We call this type of guarantee on the relations of variables provided by a protocol a *set-up guarantee*.

**Definition 4.7** (Set-up Guarantee)**.** *Let $\varsigma(Q, \omega)$ be a corrupted process, and let $x_i, y_j \in \mathrm{vars}(Q)$ be variables. A* set-up guarantee *$\mu$ is the transitive closure of a set $\{x_i =_{\mathrm{E}(Q)} y_j \mid 1 \le i, j \le n\}$ of $n$ equations for $n \in \mathbb{N}$. Its* domain $\mathrm{dom}(\mu) := \{x \mid \exists y \in \mathrm{vars}(Q) \colon x = y \in \mu\}$ *is the set of all variables occurring in $\mu$. We define that $\varsigma(Q, \omega)$ satisfies $\mu$, or $\varsigma(Q, \omega) \Vdash \mu$, as follows:*

$$\begin{aligned}
\varsigma(Q, \omega) \Vdash \mu :\Leftrightarrow \forall \lambda \in \mathrm{tr}(\varsigma(Q, \omega)) \colon (\varsigma(Q, \omega) \xrightarrow{\lambda} (\mathcal{E}, Q', \varphi, \sigma) \\
\Rightarrow ((\forall w \in \omega \colon \varphi \nvdash w\sigma) \\
\Rightarrow (((x \in \mathrm{dom}(\sigma) \wedge y \in \mathrm{dom}(\sigma) \wedge x =_{\mathrm{E}(Q)} y \in \mu) \\
\Rightarrow x\sigma =_{\mathrm{E}(Q)} y\sigma))) \\
\wedge \exists \lambda' \colon \varsigma(Q, \omega) \xrightarrow{\lambda} \xrightarrow{\lambda'} (\mathcal{E}', Q'', \varphi', \sigma') \\
\wedge \mathrm{dom}(\mu) \subseteq \mathrm{dom}(\sigma'))
\end{aligned}$$

This statement requires that for every trace $\lambda$ of a process $Q$, two things hold: First, if two variables have been assigned a value at the end of $\lambda$, and these variables are required to be equal by the set-up guarantee $\mu$, then the two values have to have a value equal in the equational theory $\mathrm{E}(Q)$ of $Q$. Second, the extended process to which $\lambda$ leads has to enable that, if executed further, all variables appearing in $\mu$ have been assigned a value. However, this only has to hold if none of the corruptible variables has been leaked yet – in this case, we cannot get a guarantee on the variable values anyways.

If a protocol is modeled as a partial process, it will often still be useful to analyze it without having to specify a concrete protocol to complete it; a generic process that just sets up variables to satisfy a set-up guarantee suffices for that. Such a process only needs the ability to bind variables with a name restriction and to assign terms to variables.

**Definition 4.8** (Executable)**.** *Let $\mu$ be a set-up guarantee, let $x$ and $y$ be variables and let $t$ be a term. Let $\langle\rangle$ be a process generated by the following grammar:*

$$\begin{aligned}\langle\rangle &:\equiv\ 0 \mid \nu x.\langle\rangle \mid y := t \parallel P\\ P &:\equiv\ 0 \mid y := t \parallel P\end{aligned}$$

*Then, if $\langle\rangle \vDash \mu$, $\langle\rangle$ is called an* executable*; we write $\langle\mu\rangle\texttt{.exe}$.*

Note that although we use the assignment operator in this definition, we will assume in all following proofs that it has been resolved to synchronizing inputs and outputs as described in Section 3.3.2. Similarly, if an executable is placed in a partial process where it is succeeded by a sequentially composed operation, e.g., as it would in $P[] := \nu x._.\mathbf{out}(c, \texttt{senc}(x, k))$, we assume that a parallel composition merge is unrolled at the end of the executable.

*Example* 4.9. *XP* needs an executable that guarantees $k_A =_\mathrm{E} k_B$ to work properly. This is easily implementable as $\langle k_A =_\mathrm{E} k_B\rangle\texttt{.exe} :\equiv \nu k.k_A := k.k_B := k$. We can then use this executable to investigate $XP[\langle k_A =_\mathrm{E} k_B\rangle\texttt{.exe}]$, which allows us to analyze the data exchange protocol for its desired properties.

### 4.2.3. Composed Trace Properties

So far, we have defined how two processes can be composed to a single process. Their trace properties, however, are defined specifically for the respective individual protocols; they can not simply be applied to a composed protocol. In this section, we describe how a trace property of an individual protocol is projected to a composed protocol.

First of all, this requires some additional notions about terms; specifically, we need to be able to handle the union of two equational theories. For this, we take the following notions from [7], adapting notation to our needs.

A *context* is a term $C \in \mathcal{T}(\Sigma(P) \cup \{_\}, \mathcal{V}())$; the _ represents a *hole* that can be filled with other terms. If a context $C$ contains $n$ blanks, then $C[t_1, \ldots, t_n]$ is the term in which the $i^{\text{th}}$ _ in $C$ is replaced with $t_i$.

A term $t \in \mathcal{T}(\Sigma(P) \cup \Sigma(Q), \mathcal{V}(P) \cup \mathcal{V}(Q))$ is called *pure* if $t \in \mathcal{T}(\Sigma(P), \mathcal{V}(P))$ or $t \in \mathcal{T}(\Sigma(Q), \mathcal{V}(Q))$; analogously, a context $C \in \mathcal{T}(\Sigma(P) \cup \Sigma(Q) \cup \{_\}, \mathcal{V}(P) \cup \mathcal{V}(Q))$ is called *pure* if $C \in \mathcal{T}(\Sigma(P) \cup \{_\}, \mathcal{V}(P))$ or $C \in \mathcal{T}(\Sigma(Q) \cup \{_\}, \mathcal{V}(Q))$. To make precise over which signature a pure term or context is built, we will call them $\Sigma(P)$- or $\Sigma(Q)$-pure if appropriate.

If $R \in \{P, Q\}$, $K \in \{P, Q\} \setminus \{R\}$, $t$ is a term, $C$ a pure $\Sigma(R)$-context, and $t = C[s_1, \ldots, s_n]$ with $\mathrm{root}(s_j) \in \Sigma(\overline{R})$ for all $1 \leq j \leq n$, then $s_1, \ldots, s_n$ are the *alien subterms* of $t$, and we write $t = C[[s_1, \ldots, s_n]]$. Note that by Lemma A.1, every term is uniquely determined by a pure context and a set of alien subterms.

A key notion for our proofs are *collapsed terms*. Collapsing a term replaces each of its alien subterms by a variable of that term; subterms that are equal in the equational theory are replaced by the same variable to retain the equivalence. This transforms an arbitrary term into a pure term while retaining equivalences, which makes reasoning about them easier. Collapsed terms serve as a kind of normal form.

**Definition 4.10** (Collapsed Terms)**.** *Let $t$ be a term such that for two contexts $C$ and $D$, $t = C[[t_1, \ldots, t_n]]$ and $C[\downarrow t_1, \ldots, \downarrow t_n] = D[[s_1, \ldots, s_k]]$. Then $\downarrow t$ is defined recursively as follows.*

$$\downarrow t := \begin{cases} s_j & \exists j\colon D[x_{[s_1]_{=_\mathrm{E}}}, \ldots, x_{[s_k]_{=_\mathrm{E}}}] =_\mathrm{E} x_{[s_j]_{=_\mathrm{E}}} \\ C[\downarrow t_1, \ldots, \downarrow t_n] & \textit{otherwise} \end{cases}$$

*In the first case, if there are several values $j$ which satisfy the condition, we choose the minimal such $j$.*

*$t$ is* collapsed *if $t = \downarrow t$.*

Note that collapsing yields a pure term, but only in the signature of the root symbol of the collapsed term. If we need to transform a term into a pure term in a specific signature, we need to *purify* it; for this, we replace the entire term with a variable if its root symbol is not in the desired signature.

**Definition 4.11** (Purification)**.** *Let $\Sigma(R)$ and $\Sigma(S)$ be signatures. Let $t, s_1, \ldots, s_n$ be terms and $C$ be a context built over $\Sigma(R) \cup \Sigma(S)$ such that $\downarrow t = C[s_1, \ldots, s_n]$. Then* purifying *$t$ is defined as follows.*

$$\downarrow\downarrow_{\Sigma(R)}(t) := \begin{cases} C[x_{[s_1]_{=_\mathrm{E}}}, \ldots, x_{[s_n]_{=_\mathrm{E}}}] & C \textit{ is a pure } \Sigma(R)\textit{-context} \\ x_{[C[s_1,\ldots,s_n]]_{=_\mathrm{E}}} & \textit{else} \end{cases}$$

Note that by Theorem A.3, if two terms $s$ and $t$ are equal in some equational theory $E$, their purified versions are also equal in $E$.

We extend the definition of purification to substitutions by purifying all involved terms.

**Definition 4.12** (Purified Substitution)**.** *Let $\Sigma(P)$ be a signature, and let $\sigma$ be a variable substitution. Then a* purified substitution *is defined as $\downarrow\downarrow_{\Sigma(P)}(\sigma) := \{x \mapsto \downarrow\downarrow_{\Sigma(P)}(t) \mid x\sigma = t\}$.*

Next we define the combination of individual traces and extend the above definitions from terms to trace. We start with the notion of an *interleaving* of two traces $\lambda_1$ and $\lambda_2$. Every action in the interleaving is either from $\lambda_1$ or $\lambda_2$, and any two actions from the same trace that appear in any order on the interleaving also appear in that same order on the original trace.

**Definition 4.13** (Interleaving)**.** *Let $\lambda_1 = (tr_1, \nu\mathcal{E}_1.\varphi_1)$ and $\lambda_2 = (tr_2, \nu\mathcal{E}_2.\varphi_2)$ be traces, with $tr_1 = a_1, \ldots, a_n$ and $tr_2 = b_1, \ldots, b_m$. Let $c_1, \ldots, c_{n+m}$ be a permutation of $a_1, \ldots, a_n, b_1, \ldots, b_m$. Then $\lambda_1 \parallel \lambda_2 := (tr_1 \parallel tr_2, \nu(\mathcal{E}_1 \cup \mathcal{E}_2).(\varphi_1 \cup \varphi_2))$ is their* interleaving*, where $tr_1 \parallel tr_2 \in \{c_1, \ldots, c_{n+m} \mid \forall i, j \in \{1, \ldots, n+m\} : ((c_i = a_k \wedge c_j = a_l) \vee (c_i = b_k \wedge c_j = b_l)) \wedge k < l \Rightarrow i < j\}$.*

We will use $\lambda_1 \parallel \lambda_2$ to denote both a specific interleaving of $\lambda_1$ and $\lambda_2$ and the set of all interleavings of $\lambda_1$ and $\lambda_2$. The respective meaning will be clear from context.

Next, choosing labels from a trace while maintaining their order forms a *subtrace*.

**Definition 4.14** (Subtrace)**.** *Let $\lambda := (tr, \nu\mathcal{E}.\varphi)$ be a trace. Then $\lambda'$ is a* subtrace *of $\lambda$, or $\lambda' \preceq \lambda$, if there is a $\lambda''$ such that $\lambda' \parallel \lambda'' = \lambda$.*

Note that we do not require the labels in a subtrace to be contiguous in the main trace. Analogously, we will write $tr' \preceq tr$ if there is a $tr''$ such that $tr' \parallel tr'' = tr$.

Finally, we need to extend the notion of purification to traces.

**Definition 4.15** (Purified Trace)**.** *Let $\Sigma(P)$ be a signature, and let $\lambda = (tr, \nu\mathcal{E}.\varphi)$ be a trace. Then a* purified trace *is defined as $\downarrow\downarrow_{\Sigma(P)}(\lambda) := (\downarrow\downarrow_{\Sigma(P)}(tr), \nu\mathcal{E}.\downarrow\downarrow_{\Sigma(P)}(\varphi))$, which is recursively defined as $\downarrow\downarrow_{\Sigma(P)}(\mathbf{in}(c, M).tr') := \mathbf{in}(c, \downarrow\downarrow_{\Sigma(P)}(M).\downarrow\downarrow_{\Sigma(P)}(tr'))$ and $\downarrow\downarrow_{\Sigma(P)}(a.tr') := a.\downarrow\downarrow_{\Sigma(P)}(tr')$ if $a \neq \mathbf{in}(c, M)$.*

With all of the above in place, we can move on to define a *composed trace property*: If the traces of a composed process contain only subtraces of one of its base protocols that satisfy a given trace property, then this property also holds for the composed process.

**Definition 4.16** (Composed Trace Property)**.** *Let $P$ be a process, and let $\beta$ be a trace property defined over a process $Q$ such that $\Sigma(Q) \subseteq \Sigma(P)$. A trace $\lambda \in \mathrm{tr}(P)$ satisfies $\beta$ if all subtraces of $\lambda$ that are traces of $Q$ over $\Sigma(P)$ satisfy $\beta$ when rid of all symbols from $\Sigma(Q)$:*

$$\lambda \vdash \beta :\Leftrightarrow \forall \lambda_Q \preceq \lambda\colon \lambda_Q \in \mathrm{tr}(Q, \Sigma(P)) \Rightarrow \downarrow\downarrow_{\Sigma(Q)}(\lambda_Q) \vdash \beta$$

*$\lambda$ vacuously satisfies a trace property $\gamma$ defined over a process $R$ such that $\Sigma(R) \cap \Sigma(P) = \emptyset$.*

# 5. Composition Results

We are now able to state and prove our main composition result. Before doing so, we will however prove another, weaker result: If two protocols share neither primitives nor data, then they can securely be run in parallel: All guarantees they provide on their own also hold for the composed protocol. We call this *disjoint composition*.

This is a generalisation of [34], where the same result is shown for a limited set of primitives. There, protocols may only use concatenation, symmetric and asymmetric encryption, signatures and hash functions – none of which can be used by both protocols.

Afterwards, we will tackle our main result: All security guarantees encodable as trace properties that a protocol provides against an attacker capable of dynamic corruption still hold if it is composed with another protocol, as long as they do not share primitives, and some side conditions hold. The main idea of the proof is to transform the problem into a disjoint composition, on which we then can apply the disjoint composition result.

## 5.1. Disjoint Composition

In this section, we prove that for two protocols that have no common equational theory and share neither names nor variables, every execution of their parallel composition consists of two executions of the individual protocols. This theorem is a generalization of [34]; we will need it for two crucial arguments in the proof of our main result: It implies that if two protocols that have disjoint signatures, variables and names, both trace properties and set-up guarantees can be securely composed.

Technically, the theorem states that whenever two such processes are composed in parallel, any execution can be divided into an execution of each of the individual protocols. When doing so, we can cleanly separate all the elements of an extended process – we merely have to remove references to the equational theory of the respective other process by purifying terms on the trace, in the adversary knowledge and the variable assignments.

**Lemma 5.1.** *Let $P$ and $Q$ be two processes built over $\Sigma(P)$ and $\Sigma(Q)$ with equational theories $\mathrm{E}(P)$ and $\mathrm{E}(Q)$, respectively, such that $\Sigma(P) \cap \Sigma(Q) = \emptyset$, $\mathrm{vars}(P) \cap \mathrm{vars}(Q) = \emptyset$ and $\mathrm{names}(P) \cap \mathrm{names}(Q) = \emptyset$. Then:*

$$
\begin{aligned}
&\exists \lambda, \mathcal{E}, R, \varphi, \sigma \colon P \parallel Q \xrightarrow{\lambda} (\mathcal{E}, R, \varphi, \sigma) \\
\Longrightarrow\ &\exists \lambda_P, \lambda_Q, \mathcal{E}_P, \mathcal{E}_Q, P', Q', \varphi_P, \varphi_Q, \sigma_P, \sigma_Q \colon \\
&P \xrightarrow{\Downarrow_{\Sigma(P)}(\lambda_P)} (\mathcal{E}_P, P', \Downarrow_{\Sigma(P)}(\varphi_P), \Downarrow_{\Sigma(Q)}(\sigma_P)) \\
&\wedge Q \xrightarrow{\Downarrow_{\Sigma(Q)}(\lambda_Q)} (\mathcal{E}_Q, Q', \Downarrow_{\Sigma(Q)}(\varphi_Q), \Downarrow_{\Sigma(Q)}(\sigma_Q)) \\
&\wedge \mathcal{E} = \mathcal{E}_P \cup \mathcal{E}_Q \wedge R = P' \parallel Q' \wedge \varphi = \varphi_P \cup \varphi_Q \wedge \sigma = \sigma_P \cup \sigma_Q \wedge \lambda \in \lambda_P \parallel \lambda_Q
\end{aligned}
$$

*Proof.* Let $\lambda$, $\mathcal{E}$, $R$, $\varphi$ and $\sigma$ be such that $P \parallel Q \xrightarrow{\lambda} (\mathcal{E}, R, \varphi, \sigma)$ for $\lambda = (tr, \nu\mathcal{E}.\varphi)$. From these, we can construct both $\lambda_P$, $\mathcal{E}_P$, $P'$, $\varphi_P$ and $\sigma_P$ such that $P \xrightarrow{\Downarrow_{\Sigma(P)}(\lambda_P)} (\mathcal{E}_P, P', \Downarrow_{\Sigma(P)}(\varphi_P), \Downarrow_{\Sigma(P)}(\sigma_P))$, and $\lambda_Q$, $\mathcal{E}_Q$, $Q'$, $\varphi_Q$ and $\sigma_Q$ such that $Q \xrightarrow{\Downarrow_{\Sigma(Q)}(\lambda_Q)} (\mathcal{E}_Q, Q', \Downarrow_{\Sigma(Q)}(\varphi_Q), \Downarrow_{\Sigma(Q)}(\sigma_Q))$.

- Since $\mathrm{names}(P) \cap \mathrm{names}(Q) = \emptyset$, there are no names in $\mathcal{E}$ that occur in both $P$ and $Q$. Thus, $\mathcal{E}_P := \mathcal{E} \cap \mathrm{names}(P)$ and $\mathcal{E}_Q := \mathcal{E} \cap \mathrm{names}(Q)$.
- Similarly, since $\mathrm{vars}(P) \cap \mathrm{vars}(Q) = \emptyset$, $\sigma$ does not contain a mapping for a variable that appears in $P$ and $Q$; so $\sigma_P := \{x \mapsto t \mid x \in \mathrm{vars}(P) \wedge x\sigma = t\}$ and $\sigma_Q := \{x \mapsto t \mid x \in \mathrm{vars}(Q) \wedge x\sigma = t\}$.
- $P$ and $Q$ cannot synchronize on an input in one process and an output in the other process, as $\mathrm{names}(P) \cap \mathrm{names}(Q) = \emptyset$. Every transition thus reduces either $P$ or $Q$, and $R :\equiv P' \parallel Q'$ must be a parallel composition at the top level, with a process on each side that $P$ and $Q$, respectively, can be reduced to.
- The adversary knowledge can be cleanly divided into messages sent by an **out** statement in $P$ and an **out** statement in $Q$. However, since in $P \parallel Q$ all messages are built over $\Sigma(P) \cup \Sigma(Q)$, the terms in the adversary knowledge have to be purified before being used in $P$ and $Q$ alone. Formally, let $W_P \subseteq \mathrm{dom}(\varphi)$ such that $w_i \in W$ if and only if $P' \parallel Q' \xrightarrow{\nu w_i.\mathbf{out}(c,w_i)} P'' \parallel Q'$ for some reduction $P' \parallel Q'$ of $P \parallel Q$; indices $i$ are updated such that they are continuous from 1 to $|W_P|$, preserving order. Then $\varphi_P := \{w \mapsto \Downarrow_{\Sigma(P)}(w\varphi) \mid w \in W_P\}$ and analogously for $\varphi_Q$.
- Finally, since all transitions in $\lambda$ reduce either $P$ or $Q$, we can separate $\lambda$ into two traces: All transitions that reduce $P$ form $\lambda_P$, and all transitions that reduce $Q$ form $\lambda_Q$. To formally define this *filtering* of $\lambda$ for $P$ (and, analogously, for $Q$), let $\lambda = (tr, \nu\mathcal{E}.\varphi)$.

  Then, if $\lambda = (a.tr^*, \nu\mathcal{E}.\varphi) \wedge P \parallel Q \xrightarrow{a} P^* \parallel Q$, we have $\mathrm{filter}(P \parallel Q \xrightarrow{\lambda} P' \parallel Q', P) := a.\mathrm{filter}(P^* \parallel Q \xrightarrow{\lambda^*} P' \parallel Q', P)$; if $\lambda = (a.tr^*, \nu\mathcal{E}.\varphi) \wedge P \parallel Q \xrightarrow{a} P \parallel Q^*$, we have $\mathrm{filter}(P \parallel Q \xrightarrow{\lambda} P' \parallel Q', P) := \mathrm{filter}(P \parallel Q^* \xrightarrow{\lambda^*} P' \parallel Q', P)$, and if $\lambda = (\emptyset, \nu\mathcal{E}.\varphi)$, we have $\mathrm{filter}(P \parallel Q \xrightarrow{\lambda} P' \parallel Q', P) := \emptyset$.

  With this and $\mathcal{E}_P$, $\mathcal{E}_Q$, $\varphi_P$ and $\varphi_Q$ as defined above, we have $\lambda_P := (\mathrm{filter}(P \parallel Q \xrightarrow{\lambda} P' \parallel Q', P), \nu\mathcal{E}_P.\varphi_P)$ and $\lambda_Q := (\mathrm{filter}(P \parallel Q \xrightarrow{\lambda} P' \parallel Q', Q), \nu\mathcal{E}_Q.\varphi_Q)$.

We prove via induction on the length of $\lambda$ that with these constructions $P \xrightarrow{\Downarrow_{\Sigma(P)}(\lambda_P)} (\mathcal{E}_P, P', \Downarrow_{\Sigma(P)}(\varphi_P), \Downarrow_{\Sigma(P)}(\sigma_P))$ and $Q \xrightarrow{\Downarrow_{\Sigma(Q)}(\lambda_Q)} (\mathcal{E}_Q, Q', \Downarrow_{\Sigma(Q)}(\varphi_Q), \Downarrow_{\Sigma(Q)}(\sigma_Q))$.

In the base case $|\lambda| = 0$, this holds trivially.

For the inductive step, assume that $P \parallel Q \xrightarrow{\lambda_P \parallel \lambda_Q} (\mathcal{E}_P \cup \mathcal{E}_Q, P' \parallel Q', \varphi_P \cup \varphi_Q, \sigma_P \cup \sigma_Q)$, and both $P \xrightarrow{\downarrow\!\downarrow_{\Sigma(P)}(\lambda_P)} (\mathcal{E}_P, P', \downarrow\!\downarrow_{\Sigma(P)}(\varphi_P), \downarrow\!\downarrow_{\Sigma(P)}(\sigma_P))$ and $Q \xrightarrow{\downarrow\!\downarrow_{\Sigma(Q)}(\lambda_Q)} (\mathcal{E}_Q, Q', \downarrow\!\downarrow_{\Sigma(Q)}(\varphi_Q), \downarrow\!\downarrow_{\Sigma(Q)}(\sigma_Q))$.

Next, assume that there is a transition $a$ such that $(\mathcal{E}_P \cup \mathcal{E}_Q, P' \parallel Q', \varphi_P \cup \varphi_Q, \sigma_P \cup \sigma_Q) \xrightarrow{a} (\mathcal{E}_a, R_a, \varphi_a, \sigma_a)$ and $\mathcal{E}_a = \mathcal{E}'_P \cup \mathcal{E}_Q$, $\varphi_a = \varphi'_P \cup \varphi_Q$ and $\sigma_a = \sigma'_P \cup \sigma_Q$. Since $P'$ and $Q'$ have disjoint names, they cannot synchronize and thus both be reduced; therefore, assume without loss of generality that $a$ reduces $P'$, so $R_a \equiv P'' \parallel Q'$. We now have to show that $\mathcal{E}'_P$, $\varphi'_P$ and $\sigma'_P$ can be derived from $\mathcal{E}_a$, $\varphi_a$ and $\sigma_a$ as described above such that $(\mathcal{E}_P, P', \varphi_P, \sigma_P) \xrightarrow{\downarrow\!\downarrow_{\Sigma(P)}(a)} (\mathcal{E}'_P, P'', \varphi'_P, \sigma'_P)$. For this, we go through all possible transitions from which $a$ may have resulted. We use term, name and variable identifiers as they are used in the definition of the transition rules in Definition 3.6.

THEN/ELSE First, we have to show that $M\sigma_P =_\mathrm{E} N\sigma_P$ (or $M\sigma_P \neq_\mathrm{E} N\sigma_P$; in the following, we will omit the inequality case, as it works analogous to the equality case). We know that $M(\sigma_P \cup \sigma_Q) =_\mathrm{E} N(\sigma_P \cup \sigma_Q)$. $M$ and $N$ are built over $P$ and thus do not have any variables from $\mathrm{dom}(\sigma_Q)$, $M(\sigma_P \cup \sigma_Q) = M\sigma_P$ and $N(\sigma_P \cup \sigma_Q) = N\sigma_P$, so $M\sigma_P = N\sigma_P$.

As neither the restricted names, the adversary knowledge nor the variable assignments change, i.e., $\mathcal{E}_a = \mathcal{E}_P \cup \mathcal{E}_Q$, $\varphi_a = \varphi_P \cup \varphi_Q$ and $\sigma_a = \sigma_P \cup \sigma_Q$, we have that $\mathcal{E}'_P = \mathcal{E}_P$, $\varphi'_P = \varphi_P$ and $\sigma'_P = \sigma_P$.

SYNC First, the requirement of the transition rule is satisfied: We have $x \notin \mathrm{dom}(\sigma_P \cup \sigma_Q)$, so especially $x \notin \mathrm{dom}(\downarrow\!\downarrow_{\Sigma(P)}(\sigma_P)) = \mathrm{dom}(\sigma_P)$.

Next, we have to show that the update to the variable assignment works, i.e., $\downarrow\!\downarrow_{\Sigma(P)}(\sigma'_P) = \downarrow\!\downarrow_{\Sigma(P)}(\sigma_P \cup \{x \mapsto t\sigma_P\})$. By definition, $\downarrow\!\downarrow_{\Sigma(P)}(\sigma_P \cup \{x \mapsto t\sigma_P\}) = \downarrow\!\downarrow_{\Sigma(P)}(\sigma_P) \cup \{x \mapsto \downarrow\!\downarrow_{\Sigma(P)}(t\sigma_P)\}$. Yet, $\downarrow\!\downarrow_{\Sigma(P)}(t\sigma_P) = \downarrow\!\downarrow_{\Sigma(P)}(t)\downarrow\!\downarrow_{\Sigma(P)}(\sigma_P)$, and since $t \in \mathcal{T}(\Sigma(P), \mathrm{vars}(P))$, we have $\downarrow\!\downarrow_{\Sigma(P)}(t) = t$. Thus, $\downarrow\!\downarrow_{\Sigma(P)}(\sigma'_P) = \downarrow\!\downarrow_{\Sigma(P)}(\sigma_P \cup \{x \mapsto t\sigma_P\}) = \downarrow\!\downarrow_{\Sigma(P)}(\sigma_P) \cup \{x \mapsto t\downarrow\!\downarrow_{\Sigma(P)}(\sigma_P)\}$ as required.

IN We assume a transition $P' \parallel Q' \xrightarrow{\mathbf{in}(c,M)} P'' \parallel Q'$, so we have to show that a transition $P' \xrightarrow{\mathbf{in}(c, \downarrow\!\downarrow_{\Sigma(P)}(M))} P''$ is possible. For this, we go through the requirements of the transition rule: Since $c \notin \mathcal{E}_P \cup \mathcal{E}_Q$ and $x \notin \mathrm{dom}(\sigma_P \cup \sigma_Q)$, especially $c \notin \mathcal{E}_P$ and $x \notin \mathrm{dom}(\downarrow\!\downarrow_{\Sigma(P)}(\sigma_P)) = \mathrm{dom}(\sigma_P)$. By definition, $\mathrm{vars}(\downarrow\!\downarrow_{\Sigma(P)}(M)) \subseteq \mathrm{dom}(\varphi_P)$, and since $\mathrm{names}(M) \cap (\mathcal{E}_P \cup \mathcal{E}_Q) = \emptyset$, this especially means that $\mathrm{names}(\downarrow\!\downarrow_{\Sigma(P)}(M)) \cap \mathcal{E}_P = \emptyset$. Finally, since $M(\varphi_P \cup \varphi_Q) = t$, by Theorem A.3 $\downarrow\!\downarrow_{\Sigma(P)}(t) =_\mathrm{E} \downarrow\!\downarrow_{\Sigma(P)}(M(\varphi_P \cup \varphi_Q)) = \downarrow\!\downarrow_{\Sigma(P)}(M)\downarrow\!\downarrow_{\Sigma(P)}(\varphi_P)$.

As in SYNC, we have to show that the variable assignments for $P''$ are properly updated, i.e. $\downarrow\!\downarrow_{\Sigma(P)}(\sigma'_P) = \downarrow\!\downarrow_{\Sigma(P)}(\sigma_P \cup \{x \mapsto t\})$. But by definition of purification, this is equal to $\downarrow\!\downarrow_{\Sigma(P)}(\sigma_P) \cup \{x \mapsto \downarrow\!\downarrow_{\Sigma(P)}(t)\}$ as required.

OUT-T As before, we will first go through the prerequisites of the transition rule. Since $c \notin \mathcal{E}_P \cup \mathcal{E}_Q$, especially $c \notin \mathcal{E}_P$. $w_{n'}$ is a variable, and by inductive assumption, $n' = |\sigma_P| + 1$. $t$ is a term of base type built over $\Sigma(P)$ by assumption. Note that this means that $t$ is a pure $\Sigma(P)$-term.

It remains to show that the adversary knowledge hs been updated correctly in $P'$, i.e., $\downarrow\!\downarrow_{\Sigma(P)}(\varphi'_P) = \downarrow\!\downarrow_{\Sigma(P)}(\varphi_P) \cup \{w_{n'} \mapsto t\downarrow\!\downarrow_{\Sigma(P)}(\varphi_P)\}$. We have $\downarrow\!\downarrow_{\Sigma(P)}(\varphi'_P) = \downarrow\!\downarrow_{\Sigma(P)}(\varphi_P \cup \{w_n \mapsto t\sigma_P\}) = \downarrow\!\downarrow_{\Sigma(P)}(\varphi_P) \cup \{w_n \mapsto \downarrow\!\downarrow_{\Sigma(P)}(t\sigma_P)\}$. Yet, since $t$ is a pure $\Sigma(P)$-term, this results in $\downarrow\!\downarrow_{\Sigma(P)}(\varphi_P) \cup \{w_n \mapsto t\downarrow\!\downarrow_{\Sigma(R)}(\sigma_P)\}$; updating the indices now yields the desired $\downarrow\!\downarrow_{\Sigma(P)}(\varphi_P) \cup \{w_{n'} \mapsto t\downarrow\!\downarrow_{\Sigma(P)}(\sigma_P)\}$.

OUT-CH Once again, since $c, h \notin \mathcal{E}_P \cup \mathcal{E}_Q$, especially $c, h \notin \mathcal{E}_P$. A transition $\mathbf{out}(c, h)$ is thus possible in $P'$; the rest of the extended process remains unchanged.

OPEN-CH In the usual order, we start with the rule prerequisites: $c \notin \mathcal{E}_P$ with the same reasoning as above; $h \in \mathcal{E}_P \cup \mathcal{E}_Q$, but since $h$ is from $P'$ and thus $h \in \mathrm{names}(P)$, $h \in \mathcal{E}_P$; if $a$ is a fresh name in $P' \parallel Q'$, then it is also a fresh name in $P'$. Thus, a transition $P' \xrightarrow{\nu a.\mathbf{out}(c,a)} P'\{^h/_a\}$ is possible.

To conclude the case, consider the variable assignments: Since $h, a \in \mathrm{names}(P)$, it holds that $\downarrow\!\downarrow_{\Sigma(P)}(\sigma_P \cup \{h \mapsto a\}) = \downarrow\!\downarrow_{\Sigma(P)}(\sigma_P) \cup \{h \mapsto \downarrow\!\downarrow_{\Sigma(P)}(a)\}$.

NEW We can use the same arguments as above to show $x \notin \mathrm{dom}(\downarrow\!\downarrow_{\Sigma(P)}(\sigma_P))$; if $n$ is a fresh name in $P' \parallel Q'$, then it is also a fresh name in $P'$ by inductive assumption.

We can thus apply the NEW rule to $P'$; it remains to show that the variable assignments are updated correctly. Starting with the assignments as mandated by NEW for $P'$, we have $\downarrow\!\downarrow_{\Sigma(P)}(\sigma_P) \cup \{x \mapsto n\} = \downarrow\!\downarrow_{\Sigma(P)}(\sigma_P) \cup \{x \mapsto \downarrow\!\downarrow_{\Sigma(P)}(n)\} = \downarrow\!\downarrow_{\Sigma(P)}(\sigma_P \cup \{x \mapsto n\})$ as required.

REP For the final transition rule we need a fresh variable $x'$ for every bound variable $x \in \mathrm{bv}(P')$. However, this holds by inductive assumption, since there is a fresh $x'$ for every $x \in \mathrm{bv}(P' \parallel Q')$.

To prove that the variable assignments are updated correctly, we can use the same chain of transformation as above to conclude that $\downarrow\!\downarrow_{\Sigma(P)}(\sigma_P) \cup \{x' \mapsto x\downarrow\!\downarrow_{\Sigma(P)}(\sigma_P)\} = \downarrow\!\downarrow_{\Sigma(P)}(\sigma_P \cup \{x' \mapsto x\sigma_P\})$.

Finally, we need to show that $(tr.a, \nu\mathcal{E}_a.\varphi_a) \in \mathrm{filter}(P \parallel Q \xrightarrow{\lambda} P' \parallel Q' \xrightarrow{a} P'' \parallel Q', P) \parallel \mathrm{filter}(P \parallel Q \xrightarrow{\lambda} P' \parallel Q' \xrightarrow{a} P'' \parallel Q', Q)$, or, in a more readable form, abusing notation: $\lambda.a \in \lambda_P.a \parallel \lambda_Q$. By inductive

assumption, $\text{filter}(P \parallel Q \xrightarrow{\lambda} P' \parallel Q', P) = \lambda_P$, $\text{filter}(P \parallel Q \xrightarrow{\lambda} P' \parallel Q', Q) = \lambda_Q$ and $\lambda \in \lambda_P \parallel \lambda_Q$. Since $P' \parallel Q' \xrightarrow{a} P'' \parallel Q'$, we have $\text{filter}(P \parallel Q \xrightarrow{\lambda} P' \parallel Q' \xrightarrow{a} P'' \parallel Q', Q) = \lambda_Q$ and $\text{filter}(P \parallel Q \xrightarrow{\lambda} P' \parallel Q' \xrightarrow{a} P'' \parallel Q', P) = (tr.a, \nu\mathcal{E}'_P.\varphi'_P)$. As already seen, $\mathcal{E}_a = \mathcal{E}'_P \cup \mathcal{E}_Q$ and $\varphi_a = \varphi'_P \cup \varphi_Q$; since $tr \in tr_P \parallel tr_Q$, we have $tr.a \in tr_P.a \parallel tr_Q$. Thus, $\lambda_a = (tr.a, \nu\mathcal{E}_a.\varphi_a) \in (tr_P.a, \nu\mathcal{E}'_P.\varphi'_P) \parallel (tr_Q, \nu\mathcal{E}_Q.\varphi_Q) = \lambda_P.a \parallel \lambda_Q$. □

This lemma allows us to prove *disjoint composition*: If two protocols are entirely disjoint, they can be securely composed in parallel – regardless of the cryptographic primitives in use.

**Theorem 5.2.** *Let $P$ and $Q$ be two processes built over $\Sigma(P)$ and $\Sigma(Q)$, respectively, such that $\Sigma(P) \cap \Sigma(Q) = \emptyset$ and $\text{vars}(P) \cap \text{vars}(Q) = \emptyset$. Let $\alpha$ be a trace property defined over $P$ or $Q$. Then:*

$$P \vDash \alpha \land Q \vDash \alpha \implies P \parallel Q \vDash \alpha$$

*Proof.* Without loss of generality, we first assume that $\alpha$ is defined over $P$. Proving via contraposition, we second assume there is some trace $\lambda \in \text{tr}(P \parallel Q)$ with $P \parallel Q \xrightarrow{\lambda} P' \parallel Q'$ such that $\lambda \nvDash \alpha$. Third, we assume that $\lambda$ is the shortest such trace – if it is not, we repeatedly remove the last step from $\lambda$ until the removal would result in a trace that satisfies $\alpha$.

By Definition 3.12, $\lambda \nvDash \alpha$ implies that there is some subtrace $\lambda_{\text{attack}} \in \text{tr}(P, \Sigma(P) \cup \Sigma(Q))$ with both $\Downarrow_{\Sigma(P)}(\lambda_{\text{attack}}) \nvDash \alpha$ and $\lambda_{\text{attack}} \preceq \lambda$. Since $\lambda$ is the shortest trace with $\lambda \nvDash \alpha$, $\lambda_{\text{attack}}$ contains all steps resulting from a transition in $P$.

By Lemma 5.1, we can split $\lambda$ into two traces $\lambda_P$ and $\lambda_Q$ such that $P \xrightarrow{\Downarrow_{\Sigma(P)}(\lambda)} P'$ and $\lambda \in \lambda_P \parallel \lambda_Q$. Thus, $\lambda_P$ has to contain all steps resulting from a transition in $P$ as well.

Now, $\lambda_{\text{attack}} = \lambda_P$: They consist of the same series of transitions, and since they are both a trace of $P$ over $\Sigma(P) \cup \Sigma(Q)$, they take the same transitions when reducing $P$, thus acquiring the same restricted names and adversary knowledge.

Since $\lambda_{\text{attack}} \nvDash \alpha$, it also holds that $\lambda_P \nvDash \alpha$, and therefore $P \nvDash \alpha$. □

Finally, we can prove the analog of Theorem 5.2 for set-up guarantees. We will need this result to prove our main composition theorem.

**Corollary 5.3.** *Let $P$ and $Q$ be two processes built over $\Sigma(P)$ and $\Sigma(Q)$, respectively, such that $\Sigma(P) \cap \Sigma(Q) = \emptyset$ and $\text{vars}(P) \cap \text{vars}(Q) = \emptyset$. Let $\mu$ be a set-up guarantee. Then:*

$$P \vDash \alpha \land Q \vDash \mu \implies P \parallel Q \vDash \mu$$

*Proof.* Since a set-up guarantee is a trace property, this is a direct consequence of Theorem 5.2. □

## 5.2. Composition under Dynamic Corruption

We now move on to our main result. After providing a high-level overview of what the theorem achieves and what it requires, we will state it formally and give a proof sketch.

Consider two protocols $P$ and $Q$. Intuitively, our composition theorem states that a property of $P$ or $Q$ is also a property of the composed protocol $P[Q]$, if some conditions hold.

Formally, we consider $P$ to be a partial process by a process $Q$ and put in some composition context $C$, where $Q$ satisfies some set-up guarantee $\mu$ on the variables transferred from $Q$ to $P$. We assume two sets of corruptible variables $\omega_P \subseteq \text{vars}(C[P])$ and $\omega_Q \subseteq \text{vars}(C[Q])$ and some trace property $\alpha$ that is defined over one of the dynamically corrupted processes $\varsigma(C[P[\langle\mu\rangle\texttt{.exe}]], \omega_P)$ or $\varsigma(C[Q], \omega_Q)$. Our composition theorem states that, given some preconditions, it suffices to verify that $\alpha$ holds for $\varsigma(C[P[\langle\mu\rangle\texttt{.exe}]], \omega_P)$ and $\varsigma(C[Q], \omega_Q)$ alone to show that $\alpha$ holds for $\varsigma(C[P[Q]], \omega_P \cup \omega_Q)$.

A typical instantiation of this would be a data exchange or messaging protocol $P$ that uses some key exchange protocol $Q$ to establish long-term keys. $\mu$ would state that the keys provided to the participating parties by $Q$ are equal; $\omega_P$ and $\omega_Q$ could contain the long-term keys, and $\alpha$ could encode Forward Secrecy in case of leaked long-term keys. We investigate this case closer in 6.2.

We will now go through the theorem requirements, what they imply in practice, and how we can efficiently verify them.

**1. Disjoint Primitives** Most importantly, we require that $P$ and $Q$ use disjoint signatures, i.e., $\Sigma(P) \cap \Sigma(Q) = \emptyset$. This means that $P$ and $Q$ may not use the same cryptographic primitives. With this limitation, it is difficult for an adversary to trick a protocol into using messages sent by the other protocol to misuse its behavior, as neither of the two protocols has the tools – i.e., primitives – to process information sent out by the other protocol. To achieve this, practitioners should ensure that their protocols

are properly domain-separated, e.g., by tagging the primitives they use; this is a best practice in security protocol design anyways. We discussed the implications of disjoint primitives more broadly in Section 3.2 already. As of now, this requirement needs to be manually verified, but since it is a static condition, this can be easily done.

**2. Syntactic Requirements** Additionally, there are more technical preconditions: The free variables of $Q$ have to be a subset of the variables of $C$, and the free variables of $P$ that are not also variables of $C$ – i.e., the transferred variables – have to be a subset of the bound variables of $Q$. This is essentially a sanity check that $P$, $Q$ and $C$ can be properly composed, i.e., $C[P[Q]]$ does not contain any free variables anymore and the protocols only share variables that are properly initialized. This does not put any limitations on a protocol model, but requires more care when naming variables.

Also, if $P$ and $Q$ leak shared variables, both have to leak the same shared variables; otherwise the adversary could get access to new information and, with that, new attack vectors through composition. Since in practice, two protocols that are meant to be composed should assume the same threat model, and thus, the same set of corruptible variables, anyways, this is a natural assumption.

As these requirements are syntactic conditions, they are trivial to verify manually.

**3. Non-shared Channels** Next, we require that $P$ and $Q$ do not use their shared variables to exchange any channel names. If they communicated over a channel received from the respective other protocol, they could potentially synchronize in unintended ways, which could lead to unintended behavior that is not present in $P$ or $Q$ alone and could thus enable new attacks. With the following formalization, this can be checked using a protocol verifier.

**Definition 5.4.** *Let $P$ be a process built over a signature $\Sigma(P)$ with equational theory $\mathrm{E}(P)$, and let $x \in \mathrm{vars}(P)$. Then, $x$* appears in place of a name in *$P$ if there is a subprocess $\mathbf{in}(v_c, y) \preceq P$ or $\mathbf{out}(v_c, t) \preceq P$ such that $v_c =_{\mathrm{E}(P)} x$. The set of all variables appearing in place of a name in $P$ is denoted* $\mathrm{nv}(P) := \{x \in \mathrm{vars}(P) \mid \exists \mathbf{in}(v_c, y) \preceq P \colon v_c =_{\mathrm{E}(P)} x \vee \exists \mathbf{out}(v_c, t) \preceq P \colon v_c =_{\mathrm{E}(P)} x\}$.

**4. Secrecy of Shared Variables** We also require that $P$ and $Q$ do not reveal any shared variable on their own; this is defined as follows.

**Definition 5.5** (Revealing a Variable)**.** *Let $P$ be a process and let $x \in \mathrm{vars}(P)$ be a variable of $P$. Then $P$* reveals *$x$ if there is a trace $\lambda$ such that $P \xrightarrow{\lambda} (\mathcal{E}, P', \varphi, \sigma)$ and $\varphi \vdash x\sigma$.*

Why is this necessary? Intuitively, if some shared variable is revealed, even though it might not be corruptible, the adversary could use that value to construct new terms to send to the parties, changing their intended behavior. More technically, on one hand, we need this restriction to prove that all terms that are sent out from $P$ or $Q$ have their root in the respective signature; with this we can then prove that all synchronizations that take place in the composed protocol also happen in one of $P$ or $Q$ on their own. On the other hand, this restriction is necessary to make sure that shared variables can always be instantiated with fresh names if necessary. It can be checked automatically, using any protocol verifier that can reason about secrecy properties.

**5. Freshness of Shared Variables** Finally, shared variables also need to be *fresh*. A variable $x$ is fresh relative to variables $y_1, \ldots, y_n$ if none of them have the same value as $x$. This is necessary to properly keep track of shared data, which we need to make general statements about information that has been leaked to an adversary. In practice, this requires that no copies are made of shared variables. Whenever we need a shared value, we should read it from the original variable and only store it in a new one if we change it sufficiently. This can also be checked with an automated protocol verifier.

Formally, freshness means that a process has to ensure that none of the variables relative to which $x$ is supposed to be fresh is assigned a value equal to that of $x$.

**Definition 5.6** (Fresh Variables)**.** *Let $P$ be a process built over $\Sigma(P)$ with equational theory $\mathrm{E}(P)$. Let $x, y_1, \ldots y_n \in \mathrm{vars}(P)$ be variables. Then $x$ is* fresh *relative to $y_1, \ldots, y_n$ if for every $P \Rightarrow (\mathcal{E}, P', \varphi, \sigma)$ and for every $i \in \{1, \ldots, n\}$, $x\sigma \neq_{\mathrm{E}(P)} y_i\sigma$.*

In summary, requirements 1 and 2 can easily be checked by hand, while a protocol verifier can check requirements 3 through 5. However, as we will see in Sections 6.2 and 6.3, all five requirements can be verified manually with little effort even for protocols the size of TLS 1.3 ECH.

With all requirements and definitions in place, we are now able to formally state our main theorem.

**Theorem 5.7.** *Let $P$ be a partial process and $Q$ be a process built over $\Sigma(P)$ and $\Sigma(Q)$, respectively, such that $\Sigma(P) \cap \Sigma(Q) = \emptyset$ and $\mathrm{names}(P) \cap \mathrm{names}(Q) = \emptyset$, with equational theories $\mathrm{E}(P)$ and $\mathrm{E}(Q)$. Let $C$ be a composition context such that $\mathrm{fv}(P) \subseteq \mathrm{vars}(C)$, $\mathrm{fv}(Q) \subseteq \mathrm{vars}(C)$ and $\mathrm{trf}(C[P[Q]]) \subseteq \mathrm{bv}(Q)$.*

*Let* $\omega_P \subseteq \mathrm{vars}(C[P])$ *and* $\omega_Q \subseteq \mathrm{vars}(C[Q])$ *be a subset of variables of* $P$ *and* $Q$*, respectively, such that* $\omega_P \cap \mathrm{sh}(C[P[Q]]) = \omega_Q \cap \mathrm{sh}(C[P[Q]])$*. Let* $\mu$ *be a set-up guarantee such that* $\varsigma(C[Q], \omega_Q) \Vdash \mu$ *and* $\mathrm{dom}(\mu) \subseteq \mathrm{trf}(C[P[Q]])$*. Finally, let* $\alpha$ *be a trace property defined over either* $P$ *or* $Q$*. Then, if neither* $C[P[\langle\mu\rangle\,.\,\mathtt{exe}]]$ *nor* $C[Q]$ *reveal any variable in* $\mathrm{sh}(C[P[Q]])$*, every* $x \in \mathrm{sh}(C[P[Q]])$ *is fresh relative to every* $y \in \mathrm{vars}(C[P[Q]]) \setminus \mathrm{sh}(C[P[Q]])$ *in* $\varsigma(C[P[Q]], \omega_P \cup \omega_Q)$*, and* $\mathrm{nv}(C[P[Q]]) \cap \mathrm{sh}(C[P[Q]]) = \emptyset$*:*

$$\varsigma(C[P[\langle\mu\rangle\,.\,\mathtt{exe}]], \omega_P) \vDash \alpha \wedge \varsigma(C[Q], \omega_Q) \vDash \alpha \implies \varsigma(C[P[Q]], \omega_P \cup \omega_Q) \vDash \alpha$$

*Proof.* We will prove this theorem by contraposition, so assume that $S := \varsigma(C[P[Q]], \omega_P \cup \omega_Q) \nvDash \alpha$. This means that there is some *attack trace* $\lambda \in \mathrm{tr}(\varsigma(C[P[Q]], \omega_P \cup \omega_Q))$ such that $\lambda \nvdash \alpha$. Our goal is now to transform $\lambda$ into an attack trace on the *disjoint case*

$$D :\equiv \varsigma(C[P[\langle\mu\rangle\,.\,\mathtt{exe}]], \omega_P)\{^{x}/_{x_P}\}_{x \in \mathrm{vars}(C[P[Q]])} \parallel \varsigma(C[Q], \omega_Q)\{^{x}/_{x_Q}\}_{x \in \mathrm{vars}(C[P[Q]])}$$

and with that, into an attack trace on one of the composed protocols alone.

To do so, we will actually prove a stronger statement: Every trace of $S$ has a corresponding trace of $D$ that satisfies the same trace properties. Then, by the following lemma, an attack trace on $S$ has a corresponding attack trace on $D$.

**Lemma 5.8.** *Let* $\lambda$ *and* $\lambda'$ *be two traces, and let* $\alpha$ *be a trace property. Then:*

$$\lambda \vdash \alpha \wedge \lambda \sim \lambda' \implies \lambda' \vdash \alpha$$

*Proof.* This follows directly from Definition 3.12. □

Unfortunately, we are not able to simply replicate a trace of $S$ in $D$: Since we move both the composition context and the dynamic corruption below a top-level parallel composition, every shared variable can be leaked twice in the disjoint case. For every shared variable, a trace of the shared case may have at most one leak, while in the disjoint case, there can be up to two leaks. We therefore have to find a way to duplicate leaks on traces of $S$.

We do this by defining a *re-leakable process* that behaves exactly like any given process under dynamic corruption but may leak all values that can be leaked, i.e., all corruptible shared variables, twice on the same channel. Later, we will show that for every attack trace on $S$, there is a corresponding trace of the re-leakable process that violates the same properties.

**Definition 5.9** (Re-leakable Processes)**.** *Let* $R :\equiv \varsigma(C[P[Q]], \omega)$ *be a dynamically corrupted process leaking on channel* $lc$*. Then its* re-leakable process *is defined as follows.*

$$\mathrm{rl}(R) :\equiv \begin{cases} 0 & \text{if } R \equiv 0 \\ \nu x.\mathrm{rl}(Q) & \text{if } R \equiv \nu x.Q \\ \mathbf{in}(d,x).\mathrm{rl}(Q) & \text{if } R \equiv \mathbf{in}(d,x).Q \\ \mathbf{out}(lc,x) \parallel \mathbf{out}(lc,x) & \text{if } R \equiv \mathbf{out}(lc,x).Q \wedge x \in \omega \\ \mathbf{out}(lc,x) & \text{if } R \equiv \mathbf{out}(lc,x).Q \wedge x \notin \omega \\ \mathbf{out}(d,t).\mathrm{rl}(Q) & \text{if } R \equiv \mathbf{out}(d,t).Q \wedge d \neq lc \\ \mathbf{if}\ M = N\ \mathbf{then}\ \mathrm{rl}(Q_1)\ \mathbf{else}\ \mathrm{rl}(Q_2) & \text{if } R \equiv \mathbf{if}\ M = N\ \mathbf{then}\ Q_1\ \mathbf{else}\ Q_2 \\ \mathrm{rl}(Q_1) \parallel \mathrm{rl}(Q_2) & \text{if } R \equiv Q_1 \parallel Q_2 \\ !\mathrm{rl}(Q) & \text{if } R \equiv\ !Q \end{cases}$$

We will require that both leaks of the same shared variable on attack traces on $S$ happen right after another. This is captured by the following definition of *re-leaking traces*, which are those traces of a re-leakable process on which every leak is instantly repeated: For every leak on the trace, another leak happens either right before or right after it.

**Definition 5.10.** *Let* $\mathrm{rl}(\varsigma(C[P[Q]], \omega))$ *be a re-leakable process leaking on channel* $lc$*. Let* $\lambda = (tr, \nu\mathcal{E}.\varphi) \in \mathrm{tr}(\mathrm{rl}(\varsigma(C[P[Q]], \omega)))$ *such that* $\mathrm{rl}(\varsigma(C[P[Q]], \omega)) \xrightarrow{\lambda} (\mathcal{E}, R, \varphi, \sigma)$*. Then,* $\lambda$ *is a* re-leaking trace *if the following holds.*

$$\begin{aligned} \forall w_n \in \mathrm{dom}(\varphi)\colon\ & ((\nu w_n.\mathbf{out}(lc, w_n) \preceq tr \wedge (\exists x \in \mathrm{dom}(\sigma) \cap \mathrm{sh}(C[P[Q]])\colon x\sigma = w_n\varphi)) \\ & \Rightarrow ((\nu w_n.\mathbf{out}(lc, w_n).\nu w_{n+1}.\mathbf{out}(lc, w_{n+1}) \preceq tr \wedge w_{n+1}\varphi = x\sigma) \\ & \vee (\nu w_{n-1}.\mathbf{out}(lc, w).\nu w_n.\mathbf{out}(lc, w_n) \preceq tr \wedge w_{n-1}\varphi = x\sigma))) \end{aligned}$$

$\Lambda(C[P[Q]],\omega)$ *denotes the set of all re-leaking traces of* $\varsigma(C[P[Q]],\omega)$.

$$\Lambda(C[P[Q]],\omega) := \{\lambda \in \mathrm{tr}(\mathrm{rl}(\varsigma(C[P[Q]],\omega))) \mid \lambda \textit{ is re-leaking}\}$$

Note that since a re-leakable process only allows for two leaks of the same variable, we do not have to exclude the case that there is another leak before *and* after a leak. Also note that due to the freshness requirement on shared variables, this definition is unambiguous, as there cannot be a non-shared variable that has the same value and thus the same structure as a shared one.

We now state that for every attack trace on a dynamically corruptible composed protocol, there is a re-leaking trace that violates the attacked property as well.

**Lemma 5.11.** *Let* $\varsigma(C[P[Q]],\omega)$ *be a dynamically corrupted process leaking on channel* $lc$*, and let* $\alpha$ *be a trace property. Then:*

$$\varsigma(C[P[Q]],\omega) \not\models \alpha \implies \exists \lambda \in \Lambda(C[P[Q]],\omega)\colon \lambda \not\models \alpha$$

*Proof.* Let $\lambda \in \mathrm{tr}(\varsigma(C[P[Q]],\omega))$ such that $\lambda \not\models \alpha$. We can construct a re-leaking trace $\mathrm{rl}(\lambda)$ as follows.

First, let $a.(tr, \nu\mathcal{E}.\varphi) := (a.tr, \nu\mathcal{E}.\varphi)$. We assume that $\varsigma(C[P[Q]],\omega) \xrightarrow{\lambda} (\mathcal{E}, R, \varphi, \sigma)$. Now, if $a = \nu w_n.\mathbf{out}(lc, w_n)$, we have that $a$ labels a leak. If $\exists x \in \mathrm{dom}(\sigma) \cap \mathrm{sh}(C[P[Q]])\colon w_n\varphi = x\sigma$, then $a$ labels the leak of a shared variable: Since $x$ is fresh relative to non-shared variables, none of them can be equal to $x$ modulo the equational theory, let alone be structurally equal. If both conditions hold, we duplicate the leak:

$$\begin{aligned}\mathrm{rl}(a.tr, \nu\mathcal{E}.\varphi) :=& a.\nu w_{n+1}.\mathbf{out}(lc, w_{n+1}).\mathrm{rl}(tr\{^{w_i}/_{w_{i+1}}\}_{n<i\leq|\varphi|},\\ & \nu\mathcal{E}.(\varphi\{^{w_i}/_{w_{i+1}}\}_{n<i\leq|\varphi|} \cup \{w_{n+1} \mapsto w_n\varphi\}))\end{aligned}$$

Otherwise, we just move on to the next label on the trace:

$$\mathrm{rl}(a.tr, \nu\mathcal{E}.\varphi) := a.\mathrm{rl}(tr, \nu\mathcal{E}.\varphi)$$

It is easy to see that $\mathrm{rl}(\lambda) \in \Lambda(C[P[Q]],\omega)$; essentially, it adds to the trace an immediate additional output of every shared variable sent out over the leaking channel $lc$. However, since we only added new steps and neither removed nor changed the order of any labels, we still have $\lambda$ as a subtrace of $\mathrm{rl}(\lambda)$. We can now strictly follow Definition 3.12: $\lambda \in \mathrm{tr}(\varsigma(C[P[Q]],\omega), \Sigma(P) \cup \Sigma(Q))$, since a re-leakable process uses the same signature as its original process; it remains to show that $\Downarrow_{\Sigma(P)\cup\Sigma(Q)}(\lambda) \not\models \alpha$, but $\Downarrow_{\Sigma(P)\cup\Sigma(Q)}(\lambda) = \lambda \not\models \alpha$, so $\mathrm{rl}(\lambda) \not\models \alpha$. □

This leads us to the main lemma of our proof: Every re-leaking trace is in static equivalence with a trace from the disjoint case. With Lemma 5.11 and Lemma 5.8 we will then be able to conclude that we have an attack trace on the disjoint case.

**Lemma 5.12.** *Let* $P[]$ *be a partial process, let* $Q$ *be a process such that* $Q \vDash \mu$ *for some set-up guarantee* $\mu$*, let* $\omega_P \subseteq \mathrm{vars}(P)$ *and* $\omega_Q \subseteq \mathrm{vars}(Q)$ *be sets of variables, and let* $C$ *be a composition context such that the requirements of Theorem 5.7 hold. Then:*

$$\forall \lambda \in \Lambda(C[P[Q]], \omega_P \cup \omega_Q) \exists \lambda' \in \mathrm{tr}(D)\colon \lambda \sim \lambda'$$

*Proof.* To state the proof in a more concise way, we introduce a few shorthands. Recall that $S$ denotes the *shared case*, while $D$ denotes the *disjoint case*. On top of that, we will assume that both $S$ and $D$ leak on a channel $lc$. When we reference a part of an extended process, we will index it by the name of the process, e.g., $S = (\mathcal{E}_S, P_S, \varphi_S, \sigma_S)$.

In the following, we will consider two processes $S'$ and $D'$ such that $S \xrightarrow{\lambda_S} S'$ and $D \xrightarrow{\lambda_D} D'$, and assume that $\lambda_S \sim \lambda_D$.

We prove the statement by induction over the length of a transition $S \xrightarrow{tr} S'$ such that its trace is re-leaking. To make the induction work, we will in fact prove a few additional properties. Informally, we want to show that the following three properties hold for each pair of $S'$ and $D'$ such that their traces are statically equivalent:

1. $P[]$ and $Q$ are reduced to the same form respectively in both $S'$ and $D'$. This makes sure that in both $S'$ and $D'$, every transition in $S'$ can also be performed in $D'$.

2. There are always two occurrences of $\mathbf{out}(lc, x)$ actions that leak a variable $x$ composed in parallel in $S'$. If this holds, we always have the option to perform a leak of $x$ twice, enabling both the construction of another re-leaking trace, and to recreate the leaks on both sides of the composition in $D'$.

3. When there is a $\nu x$ binding for a shared variable $x$ in the composition context on one side of the composition in $D'$, there is a corresponding binding on the other side of the composition in $D'$. With this, we can replicate a $\nu x$ binding in $S'$ on both sides of the composition in $D'$, allowing us to bind both versions of $x$ in $D'$ in order to enable the first two properties in this list.

We will now formalize these properties so that we can rigorously argue that they hold. For the first property we need a representation of a process that has been reduced from a composition context and some process filling it. However, whenever there is a transition originating from a composition context, the remaining actions in the context are a composition context itself – even if it is only a hole. The analogue is true for partial process: If a partial process is reduced, the remains are still are partial process; if the process completing it has already been reduced away, then it is a partial process without a hole, i.e., a regular process. This leads to the observation that a composition context, a partial process and their process can be considered to be reduced independently.

However, since we are interested in corrupted processes, we have to make a slight adjustment to this observation: If we corrupt a process, we might add **out** statements to a composition context, turning it into a partial process. For the reasons stated above, this partial process can then still be considered to be reduced independently.

*Observation* 5.13. Let $C$ be a composition context, let $P[]$ be a partial process, let $Q$ be a process, let $\omega \subseteq \mathrm{vars}(C[P[Q]])$, and let $\varsigma(C[P[Q]], \omega) \Rightarrow R$. Then, there are partial processes $C'[]$, $P'[]$, and a process $Q'$ such that $R \equiv C'[P'[Q']]$.

Based on this, we can now formalize $P$ and $Q$ having *the same form* in $S'$ and $D'$: They need to be representable by the same (partial) processes $C'$, $P'$ and $Q'$. Ignoring leaks, the idea is to show that when $S' \equiv C'[P'[Q']]$, then we will show that $D' \equiv C'[P'[\langle \sigma_{S'} \rangle .\mathtt{exe}']] \parallel C'[Q']$, where $\langle \sigma_{S'} \rangle .\mathtt{exe}$ mimics the variable assignments of $S$.

However, we also have to make sure that the **out** statements leaking shared variables are evenly distributed: If such a variable $x$ is supposed to be leaked, $S$ contains two $\mathbf{out}(lc, x)$ statements in $C$, $P$ or $Q$, respectively, yet in $D$, there is only one $\mathbf{out}(lc, x)$ on either side of the top-level parallel composition. Thus, we index each of $C'$, $P'$ and $Q'$ with a multiset $[x, x, y, \ldots]$ of shared variables that can be leaked through the next transition; we call these variables the *leakable shared variables*. E.g., $P'_{[x,x,y]} :\equiv P'' \parallel \mathbf{out}(lc, x) \parallel \mathbf{out}(lc, x) \parallel \mathbf{out}(lc, y)$. We show that when $S' \equiv C'_{[s,s,\ldots]}[P'[Q'_{[q,q,\ldots]}]]$, then $D' \equiv C'_{[s_P,\ldots]}[P'[\langle \sigma_{S'} \rangle .\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}]$.

For the latter two properties, we introduce the notions of a *subprocess* and a *twin subprocess*. A subprocess is nothing else than a part of a larger process.

**Definition 5.14** (Subprocess)**.** *Let $P$ and $Q$ be processes. $Q$ is a* subprocess *of $P$, or $Q \preceq P$, if there is a process template $P'[]$ such that $P \equiv P'[Q]$.*

A twin subprocess now is another subprocess that is structurally equal.

**Definition 5.15** (Twin Subprocess)**.** *Let $P_1$ be a subprocess of $Q$. A subprocess $P_2$ of $Q$ is a* twin subprocess *of $P_1$ if $P_1$ and $P_2$ are structurally equal without having to apply an $\alpha$-renaming.*

Note that according to this definition, every subprocess of a given process is a twin subprocess of itself.

Using a twin subprocess, we can now express the latter properties: Both having always two actions leaking a variable in $S'$ and having a binding for the corresponding version of a shared name in $D'$ means that there has to be a twin subprocess.

For the leaks we have to show that for every $x \in (\omega_P \cup \omega_Q) \cap \mathrm{sh}(C[P[Q]])$, if there is a subprocess $L \equiv \mathbf{out}(lc, x)$ in $S'$, where $lc$ is the channel dedicated to leaks by $\varsigma$, then $L$ has exactly two twin subprocesses in $S'$: itself, and the re-leak.

The procedure for the name bindings in $D'$ is similar. For every $x \in \mathrm{psh}(C)$, if there is a subprocess $\nu x_P$ in $D'$, we have to show that there is a subprocess $\nu x_Q$ in $D'$, and vice versa.

Finally, we need a way to construct the extended process $D'$ from $S'$. Especially for $\sigma_{D'}$ and $\varphi_{D'}$, we make use of the construction of the disjoint case from [32] with some notational changes.

We start with the (non-extended) process itself. It has to enable the replication of all transitions in $S'$, and it needs to properly set up the copies of the shared variables. So if we have $P_{S'} = C'[P'[Q']]$, then $P_{D'} = C'[P'[\langle\sigma'_{S'}\rangle\,.\,\texttt{exe}]] \parallel C'[Q']$, where $\langle\sigma_{S'}\rangle\,.\,\texttt{exe} \Rightarrow \langle\sigma'_{S'}\rangle\,.\,\texttt{exe} \equiv 0$.

Note that we assume that $Q$ is not under replication in $P[Q]$; if it was, we would have $P_{D'} = C'[P'[\langle\sigma'_{S'}\rangle\,.\,\texttt{exe}]] \parallel C'[!Q']$. All following constructions and the remainder of the proof also work in this case safe for the additional notational overhead of tracking all replications of $Q$ and how far they have been reduced. For brevity and notational clarity, we omit this here.

Now, the main idea of $\langle\sigma_{S'}\rangle\,.\,\texttt{exe}$ is that it initializes the copies of the transferred variables for $P$ in the same way they were originally set up in $S'$. Now, if $Q$ has set up its copy of a shared variable, we run $\langle\sigma_{S'}\rangle\,.\,\texttt{exe}$ until it has set up the copy of the shared variable for $P$. Being an executable, $\langle\sigma_{S'}\rangle\,.\,\texttt{exe}$ leaves only internal actions on the trace, retaining static equivalence of the resulting trace.

So how to construct $\langle\sigma_{S'}\rangle\,.\,\texttt{exe}$? On the one hand, we take the copies of all instantiated transferred variables from $\sigma_{S'}$ and assign to them the name representing its equivalent variables; so if $x \in \mathrm{trf}(C[P[]]) \cap \mathrm{dom}(\sigma_{S'})$, then we have $x_P := n_{[x_Q \sigma^{\leftarrow}_{D'}]_{=_\mathrm{E}}}$ in the executable. On the other hand, the remaining transferred variables are set up to adhere to $\mu$: If $y \notin \mathrm{dom}(\sigma_{S'})$, but $z =_\mathrm{E} y \in \mu$, then we have an assignment $y := n_{[z_Q \sigma^{\leftarrow}_{D'}]_{=_\mathrm{E}}}$ if $z \in \mathrm{dom}(\sigma_{S'})$, or $y := n$ for some name $n$ if $z \notin \mathrm{dom}(\sigma_{S'})$.

Formally, we need the equivalence classes of variables equal module $\mu$, $[x]_{=_\mu}$, and a name $n_{[x]_{=_\mu}}$ for each of these classes. They are necessary to properly set up uninstantiated transferred variables. The already instantiated variables will be initialized with the names used in the variable assignments for the disjoint case.

$$
\begin{aligned}
&\langle\sigma_{S'}\rangle\,.\,\texttt{exe} :\equiv \\
&\quad \nu\{v_{[x_P \sigma^{\leftarrow}_{S'}]_{=_\mathrm{E}}}\}_{x \in \mathrm{trf}(C[P[]]) \cap \mathrm{dom}(\sigma^{\leftarrow}_{S'})} \cdot \\
&\quad \nu\{v_{[x]_{=_\mu}}\}_{x \in \mathrm{trf}(C[P[]]) \setminus \mathrm{dom}(\sigma^{\leftarrow}_{S'})\colon \neg(\exists y \in \mathrm{dom}(\sigma^{\leftarrow}_{S'}) \wedge x =_\mathrm{E} y \in \mu)} \cdot \\
&\quad (\Big\|_{x \in \mathrm{trf}(C[P[]]) \cap \mathrm{dom}(\sigma^{\leftarrow}_{S'})} x_P := v_{[x_P \sigma^{\leftarrow}_{S'}]_{=_\mathrm{E}}} \| \\
&\quad \Big\|_{x \in \mathrm{trf}(C[P[]]) \setminus \mathrm{dom}(\sigma^{\leftarrow}_{S'})\colon \exists y \in \mathrm{dom}(\sigma^{\leftarrow}_{S'}) \wedge x =_\mathrm{E} y \in \mu} x_P := v_{[y_P \sigma^{\leftarrow}_{S'}]_{=_\mathrm{E}}} \| \\
&\quad \Big\|_{x \in \mathrm{trf}(C[P[]]) \setminus \mathrm{dom}(\sigma^{\leftarrow}_{S'})\colon \neg(\exists y \in \mathrm{dom}(\sigma^{\leftarrow}_{S'}) \wedge x =_\mathrm{E} y \in \mu)} x_P := v_{[x]_{=_\mu}})
\end{aligned}
$$

Note that we do not have to explicitly cover the case of a variable being leaked before or during execution of the executable since there are no requirements on the values of the shared variables then. It does not violate $\mu$ if they still receive their intended values.

To define the restricted names $\mathcal{E}_{D'}$, adversary knowledge $\varphi_{D'}$ and variable assignments $\sigma_{D'}$, we need a set of names. We do not have to exactly replicate the names used in $S'$, we merely have to ensure that equalities of terms carry over to $D'$. Thus, consider the variables shared by $P$ and $Q$, $x_1, \ldots, x_k \in \mathrm{sh}(C[P[Q]])$. In $D$, after renaming, we have the variables $x_{1_P}, \ldots, x_{k_P}$ and $x_{1_Q}, \ldots, x_{1_Q}$. We use a fresh name for each of them, except when they are evaluated to the same term. Thus, for $1 \leq i \leq p$, we consider the fresh names $n_{[x_{i_P} \sigma_{S'}]_{=_\mathrm{E}}}$ and $n_{[x_{i_Q} \sigma_{S'}]_{=_\mathrm{E}}}$.

In some cases, we will be interested in the extended process immediately preceding $S'$ on the trace. To indicate this process, we write $\overleftarrow{S'}$. More formally, $\overleftarrow{S'}$ is an extended process such that $S \xrightarrow{tr} \overleftarrow{S'}$, and $\overleftarrow{S'} \xrightarrow{a} S$ for some transition with label $a$.

Finally, by Lemma A.2 we assume without loss of generality that $\sigma_{S'}$ is collapsed, i.e., $x\sigma_{S'} = {\downarrow}x\sigma_{S'}$ for all $x \in \mathrm{dom}(\sigma_{S'})$: By this, two terms are equal if and only if their collapsed versions are, so collapsing $\sigma_{S'}$ will preserve equivalences. Since $\sigma^{\leftarrow}_{S'}$ does not contain any substitutions that are not in $\sigma_{S'}$, this also means that $\sigma^{\leftarrow}_{S'}$ is collapsed.

With all these ingredients in place, we can now describe a transformation on terms that replaces all shared secrets, i.e., the names used to instantiate shared variables, with fresh names such that term equalities are preserved.

**Definition 5.16** (Name Separation)**.** *Let $R \in \{P, Q\}$, let $K \in \{P, Q\} \setminus \{R\}$, and let $t = f(t_1, \ldots, t_k)$ be a term built over $\Sigma(P) \cup \Sigma(Q)$. For $1 \leq i \leq p$ and $1 \leq j \leq q$, let $n_{[t]_{=_\mathrm{E}}}$ be a fresh name for every $t$ such that there is a shared variable $x$ with $x\sigma^{\leftarrow}_{S'}$. Then, the* name separation *$\delta_R(t)$ is defined as follows.*

$$
\delta_R(t) := \begin{cases} n_{[t]_{=_\mathrm{E}}} & \exists x \in \mathrm{sh}(C[P[Q]])\colon t =_\mathrm{E} x\sigma^{\leftarrow}_{S'} \wedge \mathrm{root}(t) \notin \Sigma(R) \\ f(\delta_K(t_1), \ldots, \delta_K(t_k)) & \textit{otherwise for } f \in \Sigma(K) \end{cases}
$$

As stated in Lemma A.8, name separation actually retains equivalences between terms. It enables construction of the remaining components of $D'$.

Recall that in the disjoint case, variables are indexed by the side of the parallel composition they are used in; e.g., variables $x \in \mathrm{vars}(\varsigma(C[Q], \omega_Q))$ are indexed $x_Q$. $\sigma_{D'}$ now maps all variables to their name separation, and assigns the names necessary for name separation to the corresponding variables as used in the executable.

Non-shared variables receive the same assignments as in $S'$, although name separated, and the shared variables are mapped to names representing their equivalent terms if they have been instantiated already. The helper variables used in the executable are mapped to their corresponding names; since we will only bind the helper variables in the executable after initializing the first transferred variable, these mappings are only added if at least one transferred variable has already been initialized.

**Definition 5.17** (Variable Assignments: Disjoint Case)**.**

$$\begin{aligned}\sigma_{D'} :=& \{x_P \mapsto \delta_P(x\sigma_{S'}) \mid x \in (\mathrm{vars}(\varsigma(C[P[]], \omega_P)) \cap \mathrm{dom}(\sigma_{S'}))\}\\ &\cup \{x_Q \mapsto \delta_Q(x\sigma_{S'}) \mid x \in (\mathrm{vars}(\varsigma(C[Q], \omega_Q)) \cap \mathrm{dom}(\sigma_{S'}))\}\\ &\cup \{v_{[x_P\sigma_{D'}]_{=_\mathrm{E}}} \mapsto n_{[x_P\sigma_{D'}]_{=_\mathrm{E}}} \mid x \in \mathrm{trf}(C[P[]]) \wedge \exists y \in \mathrm{trf}(C[P[]]) \cap \mathrm{dom}(\sigma_{\overleftarrow{S'}})\}\\ &\cup \{v_{[x]_{=_\mu}} \mapsto n_{[x]_{=_\mu}} \mid x \in \mathrm{trf}(C[P[]]) \setminus \mathrm{dom}(\sigma_{\overleftarrow{S'}}):\\ &\quad \neg(\exists y \in \mathrm{dom}(\sigma_{\overleftarrow{S'}}) \wedge x =_\mathrm{E} y \in \mu) \wedge \exists z \in \mathrm{trf}(C[P[]]) \cap \mathrm{dom}(\sigma_{\overleftarrow{S'}})\}\end{aligned}$$

The adversary knowledge $\varphi_{D'}$ is also based on $S'$. Variables used in terms are indexed by the subprocess they originate from, and these terms are subjected to $\sigma_{D'}$ instead of $\sigma_{S'}$. For this, we say that a term $t$ has been *emitted from* $P$ if there are partial processes $C'$, $P'$, $P''$ and $Q''$ such that $S \Rightarrow C'[P'[Q']] \xrightarrow{\nu w.\mathbf{out}(w,t)} C'[P''[Q']] \Rightarrow S'$ (analogously for $Q$).

Special attention has to be paid to the leaks of shared variables. Since we only consider re-leaking traces, all leaks of shared variables are instantly repeated, so we can consider them in pairs. For one of the leaks, the variables of the leaked term will be indexed by $P$, the other by $Q$. This covers the leaks from $P$ and $Q$ in the disjoint case – which also always happen subsequently, as we will prove. For simplicity, we will map the first leak of each pair to $P$ and the second to $Q$; we could also switch this up for all or some pairs, which would make the proof more tedious, however.

**Definition 5.18** (Adversary Knowledge: Disjoint Case)**.** *Let* $\varphi_{S'} := \{w_1 \mapsto t_1\sigma_{S'}, \ldots, w_l \mapsto t_l\sigma_{S'}\}$*. We define* $t'_{i+j}$ *as follows.*

$$t'_i := \begin{cases} t_i\{^x/_{x_P}\}_{x\in\mathrm{vars}(t)} & t_i \text{ emitted from } P\\ t_i\{^x/_{x_Q}\}_{x\in\mathrm{vars}(t)} & t_i \text{ emitted from } Q\\ x_P & t_i = t_{i+1} = x\sigma_{S'} \text{ for } x \in \mathrm{sh}(C[P[Q]])\\ x_Q & t_i = t_{i-1} = x\sigma_{S'} \text{ for } x \in \mathrm{sh}(C[P[Q]])\end{cases}$$

*Then:*

$$\varphi_{D'} := \{w_1 \mapsto t'_1\sigma_{D'}, \ldots, w_l \mapsto t'_l\sigma_{D'}\}$$

Note that this definition is exhaustive since all variables that could be emitted from the composition context $C$ are shared and thus covered by the last two cases.

For the restricted names, we have to distinguish between the shared and non-shared variables, i.e., those which occur only in $P$ or $Q$. If the name assigned to a non-shared variable is restricted in $S'$, the same name – subject to name restriction – is restricted in $D'$. In the following definition, this is handled by the first two sets, depending on whether the variable is from $P$ or $Q$. Recall that we required these two processes to be variable distinct, so each variable is captured only once.

If a restricted name belongs to a shared variable however, it has to be duplicated, as shared variables are bound twice in the disjoint case: Once in each composition context. We thus have to add the name twice, once for each side of the top-level composition. To make sure that the restricted name is for the correct side, we check whether the variable has an assignment already. This is done in the third and fourth set. The remaining sets contain the names used in the executable; as with the variable assignments, these sets only contain names if at least one transferred variable has already been initialized.

**Definition 5.19** (Restricted Names: Disjoint Case)**.**

$$\begin{aligned}\mathcal{E}_{D'} := &\{\delta_P(x_P)\sigma_{S'} \mid x\sigma_{S'} \in \mathcal{E}_{S'} \wedge x \in (\mathrm{vars}(P[]) \setminus \mathrm{sh}(C[P[Q]]) \cap \mathrm{dom}(\sigma_{S'}))\}\\ \cup&\{\delta_Q(x_Q)\sigma_{S'} \mid x\sigma_{S'} \in \mathcal{E}_{S'} \wedge x \in (\mathrm{vars}(Q) \setminus \mathrm{sh}(C[P[Q]]) \cap \mathrm{dom}(\sigma_{S'}))\}\\ \cup&\{\delta_P(x_P\sigma_{S'}) \mid x\sigma_{S'} \in \mathcal{E}_{S'} \wedge x \in \mathrm{sh}(C[P[Q]]) \cap \mathcal{E}_{S'} \wedge x_P \in \mathrm{dom}(\sigma_{S'})\}\\ \cup&\{\delta_Q(x_Q\sigma_{S'}) \mid x\sigma_{S'} \in \mathcal{E}_{S'} \wedge x \in \mathrm{sh}(C[P[Q]]) \cap \mathcal{E}_{S'} \wedge x_Q \in \mathrm{dom}(\sigma_{S'})\}\\ \cup&\{n_{[x_P\sigma_{D'}]_{=_\mathrm{E}}} \mid x \in \mathrm{trf}(C[P[]]) \wedge \exists y \in \mathrm{trf}(C[P[]]) \cap \mathrm{dom}(\sigma_{\overleftarrow{S'}})\}\\ \cup&\{n_{[x]_{=_\mu}} \mid x \in \mathrm{trf}(C[P[]]) \setminus \mathrm{dom}(\sigma_{\overleftarrow{S'}}):\\ &\quad \neg(\exists y \in \mathrm{dom}(\sigma_{\overleftarrow{S'}}) \wedge x =_\mathrm{E} y \in \mu) \wedge \exists z \in \mathrm{trf}(C[P[]]) \cap \mathrm{dom}(\sigma_{\overleftarrow{S'}})\}\end{aligned}$$

In summary, we will prove that the following holds for any trace $\lambda_S \in \Lambda(C[P[Q]], \omega_P \cup \omega_Q)$ such that $S \xrightarrow{\lambda_S} S'$:

1. $\exists \lambda_D \in \mathrm{tr}(D)\colon \lambda_S \sim \lambda_D$;
2. for $D'$ such that $D \xrightarrow{\lambda_D} D'$:
   a) $S' \equiv C'_{[s,s,\ldots]}[P'[Q'_{[q,q,\ldots]}]]$, then $D' \equiv C'_{[s_P,\ldots]}[P'[\langle\sigma_{S'}\rangle\,\texttt{.exe}'_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}]$, such that $\langle\sigma_{S'}\rangle\,\texttt{.exe} \Rightarrow \langle\sigma_{S'}\rangle\,\texttt{.exe}'$;
   b) for every $x \in (\omega_P \cup \omega_Q) \cap \mathrm{sh}(C[P[Q]])$, if there is a subprocess $\mathbf{out}(lc, x)$ in $S'$, it has exactly two twin subprocesses;
   c) for every $x \in \mathrm{psh}(C)$, if there is a subprocess $\nu x_P$ in $D'$, then there is a subprocess $\nu x_Q$ in $D'$, and vice versa.

We will call the first property *static equivalence* as usual, and the latter three properties *form symmetry*, *leak symmetry* and *restriction symmetry* respectively. Note that for static equivalence, for each inductive step we will only show that $\lambda_S$ and $\lambda_D$ produce the same series of labels modulo $\alpha$-renaming; we will prove that the two resulting frames are statically equivalent in every case after concluding the inductive argument.

In the base case, we have an execution of length 0. Its trace $\lambda_S = (\emptyset, \emptyset)$ is the empty trace. For $\lambda_D = (\emptyset, \emptyset) \in \mathrm{tr}(D)$, it trivially holds that $\lambda_S \sim \lambda_D$. The remaining statements follow from the construction of $S$ and $D$.

For the inductive step, assume some transition of length $n$ with trace $\lambda_S \in \Lambda(C[P[Q]], \omega_P \cup \omega_Q)$. We thus have $S \to S'$ with $S \xrightarrow{\lambda_S} S'$, and by inductive assumption there is a $\lambda_D$ such that $D \xrightarrow{\lambda_D} D'$ and $\lambda_S \sim \lambda_D$. The next transition from $S'$ can now be achieved by applying one of the transition rules, or, in the case of a leak, applying the OUT-T rule twice; we will show case by case that the above claims hold for each possible transition. Through all of them, we assume that the transition originates from $P$ – if it is from $Q$, a symmetric argument applies – or in the case of NEW from $C$.

We first handle the transitions which result in an internal action and are thus not visible on the trace. For all of these cases, we have to show that $\mathrm{tr}(S \xrightarrow{\lambda_S} S' \xrightarrow{\tau} S'') \in \Lambda(C[P[Q]], \omega_P \cup \omega_Q)$. However, when $\lambda_S = \mathrm{tr}(S \xrightarrow{\lambda_S} S') = (tr, \nu\mathcal{E}_{S'}.\varphi_{S'})$, then $\mathrm{tr}(S \xrightarrow{\lambda_S} S' \xrightarrow{\tau} S'') = (tr, \nu\mathcal{E}_{S''}.\varphi_{S''})$, since the internal action is not visible on the trace. By inductive assumption, $\lambda_S \in \Lambda(C[P[Q]], \omega_P \cup \omega_Q)$, and since the same $tr$ occurs in both traces, $\mathrm{tr}(S \xrightarrow{\lambda_S} S' \xrightarrow{\tau} S'') \in \Lambda(C[P[Q]], \omega_P \cup \omega_Q)$ as well.

THEN We assume that we applied THEN on $S'$, so we have $P_{S'} \equiv C'[(\mathbf{if}\ M = N\ \mathbf{then}\ P'_1\ \mathbf{else}\ P'_2 \parallel P'_3)[Q']]$. By inductive assumption, we have $P_{D'} \equiv C'[(\mathbf{if}\ M' = N'\ \mathbf{then}\ P'_1\ \mathbf{else}\ P'_2 \parallel P'_3)[\langle\mu\rangle\,\texttt{.exe}']] \parallel C'[Q']$ where $M' = M\{^x/_{x_P}\}_{x \in \mathrm{vars}(C[P[Q]])}$ and $N' = N\{^x/_{x_P}\}_{x \in \mathrm{vars}(C[P[Q]])}$; we need to show that $P_{D'} \xrightarrow{\tau} C'[(P'_1 \parallel P'_3)[\langle\mu\rangle\,\texttt{.exe}']] \parallel C'[Q']$. This is the case if we can apply the THEN rule, so we need to prove that $M'\sigma_{D'} =_\mathrm{E} N'\sigma_{D'}$.

By Lemma A.1, there are $\Sigma(P)$-contexts $C$ and $D$ and terms $s_1, \ldots, s_m, t_1, \ldots, t_n$ built over $\Sigma(Q)$ such that $M\sigma_{S'} = C[[s_1, \ldots, s_m]]$ and $N\sigma_{S'} = D[[t_1, \ldots, t_n]]$. Since $C$ and $D$ pure $\Sigma(P)$-contexts, and $M$ and $N$ are built over $\Sigma(P)$, $C$ and $D$ must entirely subsume $M$ and $N$ – they essentially are $M$ and $N$ but with holes instead of variables. Thus, all $s_i$ and $t_j$ have been assigned by $\sigma_{S'}$, which is assumed to be in collapsed form. With Lemma A.4, $s_i$ and $t_j$ are collapsed.

Since $M\sigma_{S'} =_\mathrm{E} N\sigma_{S'}$, their representations as contexts are also equal, meaning that $C[[s_1, \ldots, s_m]] =_\mathrm{E} D[[t_1, \ldots, t_n]]$. We have seen that their subterms are collapsed; thus, by Lemma A.6, there are names $n_{[s_1]_{=_\mathrm{E}}}, \ldots, n_{[s_m]_{=_\mathrm{E}}}, n_{[t_1]_{=_\mathrm{E}}}, \ldots, n_{[t_n]_{=_\mathrm{E}}}$ such that $C[n_{[s_1]_{=_\mathrm{E}}}, \ldots, n_{[s_m]_{=_\mathrm{E}}}] =_\mathrm{E} D[n_{[t_1]_{=_\mathrm{E}}}, \ldots, n_{[t_n]_{=_\mathrm{E}}}]$.

We know from Lemma A.8 that $\delta_P(s_a) =_\mathrm{E} \delta_P(s_b)$ if $s_a =_\mathrm{E} s_b$ for all $a, b \in \{1, \dots, m\}$, and analogous for all $t_j$. Thus, if we apply the substitution $\varrho := \{n_{[s_1]_{=_\mathrm{E}}} \mapsto \delta_P(s_1), \dots, n_{[s_m]_{=_\mathrm{E}}} \mapsto \delta_P(s_m), n_{[t_1]_{=_\mathrm{E}}} \mapsto \delta_P(t_1), \dots, n_{[t_n]_{=_\mathrm{E}}} \mapsto \delta_P(t_n)\}$, we map for each $s_i$ (and $t_j$, respectively) the name by which it was replaced to its name separation, yielding $C[\delta_P(s_1), \dots, \delta_P(s_m)] =_\mathrm{E} D[\delta_P(t_1), \dots, \delta_P(t_n)]$.

By definition of $\delta$, this is the same as $\delta_P(M\sigma_{S'}) =_\mathrm{E} \delta_P(N\sigma_{S'})$, which is how $M'$, $N'$ and $\sigma_{D'}$ is defined. With this, we conclude that $M'\sigma_{D'} =_\mathrm{E} N'\sigma_{D'}$.

Since neither restricted names nor adversary knowledge change through the transition, $\mathrm{tr}(D \xrightarrow{\lambda_D} D' \xrightarrow{\tau} D'') = \mathrm{tr}(D \xrightarrow{\lambda_D} D') = \lambda_D$, and $\lambda_S \sim \lambda_D$ by inductive assumption. Form symmetry follows from the fact that the case distinction may be reduced to the same two cases in the shared and disjoint case by inductive assumption, and is in fact reduced to the same subprocess as we have seen that both $M = N$ and $M' = N'$ hold; since no variables are bound, the set of leakable shared variables also remains the same. Leak and restriction symmetry are not affected.

ELSE Here, the analogue argument to THEN applies, where all equalities $M'\sigma_{D'} =_\mathrm{E} N'\sigma_{D'}$ are replaced by inequalities $M'\sigma_{D'} \neq_\mathrm{E} N'\sigma_{D'}$.

SYNC In this case, we have to prove $x_P \notin \mathrm{dom}(\sigma_{S'})$ (or $x_Q \notin \mathrm{dom}(\sigma_{S'})$, respectively); this, however, is by inductive assumption a direct consequence of Definition 5.17. Additionally, we have to prove the three symmetries, the static equivalence and that the correct substitution is added to the variable assignment.

Although one $\mathbf{out}(c, t)$ is reduced away, leak symmetry is maintained: $D$ leaks on channel $lc$, but by definition of $\varsigma$, $lc$ is not used anywhere else in $D$. Thus, $c \neq lc$, and $\mathbf{out}(c, t)$ is not required to have a twin subprocess.

For the remaining properties, we have to distinguish whether the transition assigns a value to a transferred or a non-shared variable (preshared variables are only set up via $\nu$ and are thus of no concern here). If it is a non-shared variable, restriction symmetry holds by inductive assumption and is not affected by this transition. The transition does not change the trace, so static equivalence holds by the same argument as used in the THEN case. Finally, for the variable assignments, assume that this transition results in $D' \xrightarrow{\tau} D''$. We have to prove $x_P\sigma_{D''} =_\mathrm{E} t\{^y/_{y_p}\}_{y \in \mathrm{vars}(C[P[]])}\sigma_{D''}$, for which we can use the same reasoning as in the THEN case.

If the transition initializes a transferred variable, we need to set up both its copies in $D'$. Since the transition then must have been from $Q$, we can apply the same reasoning as for a non-shared variable for $x_Q$; for $x_P$ however, we have to handle its setup in $\langle\sigma_{S'}\rangle\texttt{.exe}$.

Here, we have to do yet another case distinction: If $x_P$ is the first transferred variable to be set up in the executable, we have to first go through all the name restrictions in $\langle\sigma_{S'}\rangle\texttt{.exe}$; these transitions are possible since every $v_{[x_P\sigma_{D'}]_{=_\mathrm{E}}}$ maps to a $n_{[x_P\sigma_{D'}]_{=_\mathrm{E}}}$, which is fresh. Else, and afterwards, we have to show that $\mathrm{root}(x\sigma_{S'}) \in \Sigma(Q)$, since we then have $x_P\sigma_{D''} = \delta_P(x\sigma_{S''}) = n_{[x\sigma_{S''}]_{=_\mathrm{E}}}$ by the definition of name separation. We introduce a lemma for this.

**Lemma 5.20.** *Let $P$ and $Q$ be processes built over signatures $\Sigma(P)$ and $\Sigma(Q)$, respectively, such that $\Sigma(P) \cap \Sigma(Q) = \emptyset$ and $\mathrm{names}(P) \cap \mathrm{names}(Q) = \emptyset$. Let $x \in \mathrm{vars}(P)$ be a variable that is not revealed by $P$. Let $\omega$ be a set of variables. Then, if $\varsigma(P \parallel Q, \omega) \xrightarrow{\lambda} (\mathcal{E}, R, \varphi, \sigma)$ and $x \in \mathrm{dom}(\sigma)$, $\mathrm{root}(x\sigma) \in \Sigma(P)$.*

*Proof.* The value of $x\sigma$ can originate from one of three operations: Either a name restriction $\nu x$, an input $\mathbf{in}(c, x)$ over a non-restricted channel $c$ or a synchronization $\mathbf{out}(d, t).R_1 \parallel \mathbf{in}(d, x).R_2$ on a restricted channel $d$. All of these are operations in $P$; for the synchronization this holds for both sides of the parallel composition as we require $\mathrm{names}(P) \cap \mathrm{names}(Q) = \emptyset$. We will show that $x$ cannot receive its value over an input but only via a name restriction or a synchronization in $P$. Since all terms in $P$ must be constructed over $\Sigma(P)$, this means that $\mathrm{root}(x) \in \Sigma(P)$ regardless of the signatures the adversary has access to.

Thus, consider a transition $P \xrightarrow{\lambda_P} (\mathcal{E}_P, P', \varphi_P, \sigma_P) \xrightarrow{a} (\mathcal{E}'_P, P'', \varphi'_P, \sigma'_P)$.

First, assume that $P' \equiv \mathbf{in}(c, x).P'_1 \parallel P'_2$ and $a = \mathbf{in}(c, M)$. Then, $x\sigma'_P = t = M\varphi_P$, but since $\varphi_P = \varphi'_P \vdash M\varphi_P$, we have $\varphi'_P \vdash x\sigma'_P$. This contradicts the assumption that $x$ is not revealed by $P$, so $x$ cannot be initialized through some network input.

Next, assume that $P' \equiv \nu x.P'_1 \parallel P'_2$ and $a = \tau$. Then, $x\sigma'_P = n \in \mathrm{names}(P) \subseteq \Sigma(P)$, so $\mathrm{root}(x) \in \Sigma(P)$. Since $n \in \mathrm{names}(P) \cup \mathrm{names}(Q)$, we also have $\mathrm{root}(x\sigma) = \mathrm{root}(n) \in \Sigma(P)$.

Finally, assume that $P' \equiv \mathbf{out}(d, t) \parallel \mathbf{in}(d, x).P'_2 \parallel P'_3$ and $a = \tau$. Then, either $t = y$ for some variable $y \in \mathrm{vars}(P)$ or $t = f(\dots, x, \dots)$ for some function symbol $f \in \Sigma(P)$. In the latter case, $\mathrm{root}(x\sigma'_P) = \mathrm{root}(t\sigma_P) = \mathrm{root}(f(\dots, x, \dots)\sigma_P) = f \in \Sigma(P)$. In the former case, $y$ also must not be revealed by $P$ – otherwise $P$ would reveal $x$. $y$ must thus have been initialized by a name restriction or a synchronization, in which case we can inductively apply this part of the proof until we reach a

variable that has been instantiated either by a name restriction or a synchronization. In either case, $\mathrm{root}(t) = \mathrm{root}(y) \in \Sigma(P)$.

However, $t$ is a fixed term and does not change when $P$ is composed in parallel with any other process, and since $\mathrm{root}(x\sigma'_P) = \mathrm{root}(t) \in \Sigma(P)$, we thus also have $\mathrm{root}(x\sigma) = \mathrm{root}(x\sigma) \in \Sigma(P)$. □

Even though there are several name restrictions performed, none of them concern a preshared variable; restriction symmetry is thus preserved.

Through this transition, in the case of a non-shared variable (the synchronization happens in $P$) we have:

$$S' \equiv C'_{[s,s,\ldots]}[P'[Q'_{[q,q,\ldots]}]] \xrightarrow{\tau} C'_{[s,s,\ldots]}[P[''[Q'_{[q,q,\ldots]}]]]$$

We therefore have for the disjoint case:

$$\begin{aligned} D' &\equiv C'_{[s_P,\ldots]}[P'[\langle\mu\rangle.\,\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}] \\ &\xrightarrow{\tau} C'_{[s_P,\ldots]}[P''[\langle\mu\rangle.\,\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}] \end{aligned}$$

In the case of a shared variable not in $\omega_P \cup \omega_Q$,(the synchronization takes place in $Q$, and we reduce $\langle\mu\rangle.\,\mathtt{exe}'$ until the copy of the variable is instantiated) we have:

$$\begin{aligned} D' &\equiv C'_{[s_P,\ldots]}[P'[\langle\mu\rangle.\,\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}] \\ &\xrightarrow{\tau.\ldots.\tau} C'_{[s_P,\ldots]}[P'[\langle\mu\rangle.\,\mathtt{exe}''_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q''_{[q_Q,\ldots]}] \end{aligned}$$

If this synchronization binds $x \in \mathrm{sh}(C[P[Q]]) \cap (\omega_P \cup \omega_Q)$, we finally have:

$$S' \equiv C'_{[s,s,\ldots]}[P'[Q'_{[q,q,\ldots]}]] \xrightarrow{\tau} C'_{[s,s,\ldots]}[P[''[Q'_{[q,q,x,x,\ldots]}]]]$$

This leads to:

$$\begin{aligned} D' &\equiv C'_{[s_P,\ldots]}[P'[\langle\mu\rangle.\,\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}] \\ &\xrightarrow{\tau.\ldots.\tau} C'_{[s_P,\ldots]}[P'[\langle\mu\rangle.\,\mathtt{exe}''_{[q_P,x_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q''_{[q_Q,x_Q,\ldots]}] \end{aligned}$$

In all three cases, form symmetry holds.

Since only internal actions occur, the traces remain equivalent.

NEW Regardless of whether the name restriction $\nu x$ in question binds a shared or a non-shared variable, we have $x \notin \mathrm{dom}(\sigma_{S'})$ and thus by construction $x \notin \mathrm{dom}(\sigma_{D'})$, or $x_P \notin \mathrm{dom}(\sigma_{D'})$ and $x_Q \notin \mathrm{dom}(\sigma_{D'})$, respectively. In both cases we have to show that $x$ or $x_P$ and $x_Q$ are assigned a fresh name, i.e., $x\sigma_{D''}$ or $x_P\sigma_{D''}$ and $x_Q\sigma_{D''}$ have to be fresh.

If $x$ is not shared, then $x\sigma_{D''} = \delta_P(x\sigma_{S''}) = x\sigma_{S''}$ is fresh by inductive assumption. No shared variables are instantiated, nothing sent out and we have $S' \equiv C'_{[s,s,\ldots]}[P'[Q'_{[q,q,\ldots]}]] \xrightarrow{\tau} C'_{[s,s,\ldots]}[P''[Q'_{[q,q,\ldots]}]]$ and thus:

$$\begin{aligned} D' &\equiv C'_{[s_P,\ldots]}[P'[\langle\mu\rangle.\,\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}] \\ &\xrightarrow{\tau} C'_{[s_P,\ldots]}[P''[\langle\mu\rangle.\,\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}] \end{aligned}$$

Therefore, symmetries hold and labels are equivalent.

If $x$ is shared, then it is not revealed and thus $x_P\sigma_{D''} = \delta_P(x\sigma_{S''})$ is fresh by definition of name separation. For the remaining properties we have to distinguish whether $x$ is preshared or transferred.

If $x$ is preshared, we have to bind both $x_P$ and $x_Q$ in $D'$. If $x \notin \omega_P \cup \omega_Q$, for $S' = C'_{[s,s,\ldots]}[P'[Q'_{[q,q,\ldots]}]] \xrightarrow{\tau} C''_{[s,s,\ldots]}[P'[Q'_{[q,q,\ldots]}]]$ we thus have:

$$\begin{aligned} D' &\equiv C'_{[s_P,\ldots]}[P'[\langle\mu\rangle.\,\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}] \\ &\xrightarrow{\tau}\xrightarrow{\tau} C''_{[s_P,\ldots]}[P'[\langle\mu\rangle.\,\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C''_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}] \end{aligned}$$

If $x$ is corruptible, so $x \in \omega_P \cup \omega_Q$, then $S' = C'_{[s,s,\ldots]}[P'[Q'_{[q,q,\ldots]}]] \xrightarrow{\tau} C''_{[s,s,x,x,\ldots]}[P'[Q'_{[q,q,\ldots]}]]$, so in the disjoint case we have:

$$\begin{aligned} D' &\equiv C'_{[s_P,\ldots]}[P'[\langle\mu\rangle.\,\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}] \\ &\xrightarrow{\tau}\xrightarrow{\tau} C''_{[s_P,x_P,\ldots]}[P'[\langle\mu\rangle.\,\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C''_{[s_Q,x_Q,\ldots]}[Q'_{[q_Q,\ldots]}] \end{aligned}$$

Thus, form symmetry, restriction symmetry and equivalence of labels modulo internal actions hold, while leak symmetry is not affected by this transition.

If $x$ is transferred, we have to instantiate both copies of $x$ in $D'$; for the process in the executable, we can use the argument from SYNC analogously. If $x \notin \omega_P \cup \omega_Q$, in the shared case we then have $S' = C'_{[s,s,\dots]}[P'[Q'_{[q,q,\dots]}]] \xrightarrow{\tau} C'_{[s,s,\dots]}[P'[Q''_{[q,q,\dots]}]]$ and in the disjoint case we have:

$$\begin{aligned} D' &\equiv C'_{[s_P,\dots]}[P'[\langle\mu\rangle\,.\,\mathtt{exe}'_{[q_P,\dots]}]] \parallel C'_{[s_Q,\dots]}[Q'_{[q_Q,\dots]}] \\ &\xrightarrow{\tau}\xrightarrow{\tau.\dots.\tau} C'_{[s_P,\dots]}[P'[\langle\mu\rangle\,.\,\mathtt{exe}''_{[q_P,\dots]}]] \parallel C'_{[s_Q,\dots]}[Q''_{[q_Q,\dots]}] \end{aligned}$$

Else, $S' = C'_{[s,s,\dots]}[P'[Q'_{[q,q,\dots]}]] \xrightarrow{\tau} C'_{[s,s,\dots]}[P'[Q''_{[q,q,x,x,\dots]}]]$, and in the disjoint case:

$$\begin{aligned} D' &\equiv C'_{[s_P,\dots]}[P'[\langle\mu\rangle\,.\,\mathtt{exe}'_{[q_P,\dots]}]] \parallel C'_{[s_Q,\dots]}[Q'_{[q_Q,\dots]}] \\ &\xrightarrow{\tau}\xrightarrow{\tau.\dots.\tau} C'_{[s_P,\dots]}[P'[\langle\mu\rangle\,.\,\mathtt{exe}''_{[q_P,x_P,\dots]}]] \parallel C'_{[s_Q,\dots]}[Q''_{[q_Q,x_Q,\dots]}] \end{aligned}$$

The symmetries hold, and since only internal actions are put on the trace, both traces still have an equivalent series of labels.

REP Similarly to SYNC, we do not need to prove any requirements, but we have to prove the symmetries, static equivalence and the correct construction of the variable assignments. We also need to distinguish two cases: The replication can be from $P$, or from the composition context $C$.

In the former case, no name restrictions on preshared variables are affected, so restriction symmetry holds. Form symmetry also holds as we can by inductive assumption perform the replication in $D'$. Additionally, if there are any leaks $\mathbf{out}(lc, x)$ of a variable $x \in \omega_P \cup \omega_Q$ under replication, we know by inductive assumption that this $\mathbf{out}(lc, x)$ has exactly two twin subprocesses; by construction of a releaking process, these two $\mathbf{out}(lc, x)$ both have to be under the same replication. Thus, after replication, $\mathbf{out}(lc, x')$ has exactly two twin subprocesses and form symmetry holds.

Static equivalence holds once again by the same argument as used in the THEN case since the trace does not change through $D' \xrightarrow{\tau} D''$, and for the variable assignments, $x\sigma_{D''} =_{\mathrm{E}} x'\sigma_{D''}$ for all $x' \in \mathrm{bv}(P)$ by construction.

In the latter case, where the replication is in $C$, we have to emulate this replication on both sides of the parallel composition in $D'$: If $S' \equiv C'_{[s,s,\dots]}[P'[Q'_{[q,q,\dots]}]] \xrightarrow{\tau} \mathrm{rl}(C''_{[s,s,\dots]}[P'[Q'_{[q,q,\dots]}]])$, then if $C' \equiv (C'_1 \parallel \mathbf{out}(lc, s') \parallel \mathbf{out}(lc, s') \parallel \dots) \parallel C'_2$:

$$\begin{aligned} D' &\equiv C'_{[s_P,\dots]}[P'[\langle\mu\rangle\,.\,\mathtt{exe}'_{[q_P,\dots]}]] \parallel C'_{[s_Q,\dots]}[Q'_{[q_Q,\dots]}] \\ &\xrightarrow{\tau} C''_{[s_P,s'_P,\dots]}[P'[\langle\mu\rangle\,.\,\mathtt{exe}'_{[q_P,\dots]}]] \parallel C'_{[s_P,\dots]}[Q'_{[q_Q,\dots]}] \\ &\xrightarrow{\tau} C''_{[s_P,s'_P,\dots]}[P'[\langle\mu\rangle\,.\,\mathtt{exe}'_{[q_P,\dots]}]] \parallel C''_{[s_Q,s'_Q,\dots]}[Q'_{[q_Q,\dots]}] \end{aligned}$$

Otherwise, we have:

$$\begin{aligned} D' &\equiv C'_{[s_P,\dots]}[P'[\langle\mu\rangle\,.\,\mathtt{exe}'_{[q_P,\dots]}]] \parallel C'_{[s_Q,\dots]}[Q'_{[q_Q,\dots]}] \\ &\xrightarrow{\tau} C''_{[s_P,\dots]}[P'[\langle\mu\rangle\,.\,\mathtt{exe}'_{[q_P,\dots]}]] \parallel C'_{[s_P,\dots]}[Q'_{[q_Q]}] \\ &\xrightarrow{\tau} C''_{[s_P,\dots]}[P'[\langle\mu\rangle\,.\,\mathtt{exe}'_{[q_P,\dots]}]] \parallel C''_{[s_Q,\dots]}[Q'_{[q_Q,\dots]}] \end{aligned}$$

This ensures form symmetry in both cases. Also, if there are any name restrictions of preshared variables under replication in $S'$, both copies are replicated in $D'$, maintaining restriction symmetry.

Analogously to the former case, any leaks $\mathbf{out}(lc, x)$ under replication have exactly two twin subprocesses by inductive assumption, so the replicated leaks $\mathbf{out}(lc, x')$ also have exactly two twin subprocesses.

Finally, static equivalence holds by the analogue argument as in the first case, since $x\sigma_{D''} =_{\mathrm{E}} x'\sigma_{D''}$ for all $x' \in \mathrm{bv}(P) \cup \mathrm{bv}(Q)$ by construction.

We proceed with the transitions which are visible on the trace.

IN For an operation $\mathbf{in}(c, x)$ with input $M\varphi_{S'}$ we first have to show that the requirements for an analogous transition in $D'$ hold, and we need to show $x_P\sigma_{D'} =_{\mathrm{E}} M\{^{x}/_{x_P}\}_{x\in\mathrm{sh}(C[P[Q]])}\varphi_{D'}$ to prove the construction of the adversary knowledge correct (we assume without loss of generality that $x$ is a shared variable).

First, since $c \notin \mathcal{E}_{S'}$, we have by construction $\delta_P(c) \notin \mathcal{E}_{D'}$. Since $x \notin \mathrm{dom}(\sigma_{S'})$, $\mathrm{vars}(M) \subseteq \mathrm{dom}(\varphi_{S'})$ and $\mathrm{names}(M) \cap \mathcal{E}_{S'} = \emptyset$, by construction of $\sigma_{S'}$, $\varphi_{D'}$ and $\mathcal{E}_{D'}$, it holds that $x_P \notin \sigma_{D'}$, that $\mathrm{vars}(M\{^{x}/_{x_P}\}_{x\in\mathrm{sh}(C[P[Q]])}) \subseteq \mathrm{dom}(\varphi_{D'})$ and it also holds that $\mathrm{names}(M\{^{x}/_{x_P}\}_{x\in\mathrm{sh}(C[P[Q]])}) \cap \mathcal{E}_{D'} = \emptyset$.

Next we will show that $M\{^x/_{x_P}\}_{x\in\mathrm{sh}(C[P[Q]])}\varphi_{D'} =_\mathrm{E} x_P\sigma_{D''}$. We know that $M\varphi_{S'} =_\mathrm{E} x\sigma_{S''}$; with Lemma A.2 we can assume that $M\varphi_{S'}$ is collapsed, so by Lemma A.8 we have $\delta_P(M\varphi_{S'}) =_\mathrm{E} \delta_P(x\sigma_{S''})$.

From the definition of $\varphi_{D'}$ it follows that $\delta_P(M\varphi_{S'}) =_\mathrm{E} M\{^x/_{x_P}\}_{x\in\mathrm{sh}(C[P[Q]])}\varphi_{D'}$ and by definition of $\sigma_{D''}$ that $\delta_P(x\sigma_{S''}) =_\mathrm{E} x_P\sigma_{D''}$. Thus, we have $M\{^x/_{x_P}\}_{x\in\mathrm{sh}(C[P[Q]])}\varphi_{D'} =_\mathrm{E} x_P\sigma_{D''}$.

Now we move on to the symmetries; we have to distinguish whether the input sets up a shared variable. If $x \notin \mathrm{sh}(C[P[Q]])$, then we have a transition$S' \equiv C'_{[s,s,\ldots]}[P'[Q'_{[q,q,\ldots]}]] \xrightarrow{\mathbf{in}(c,M)} C'_{[s,s,\ldots]}[P''[Q'_{[q,q,\ldots]}]]$,yet:

$$D' \equiv C'_{[s_P,\ldots]}[P'[\langle\sigma_{S'}\rangle.\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}]$$
$$\xrightarrow{\mathbf{in}(c,M\{^x/_{x_P}\}_{x\in\mathrm{sh}(C[P[Q]])})} C'_{[s_P,\ldots]}[P''[\langle\sigma_{S'}\rangle.\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}]$$

No name restrictions or outputs are affected, so all three symmetries hold by inductive assumption, and the produced traces have equivalent labels.

If the variable is shared, it has to be a transferred since no inputs can occur in a composition context; it therefore must be from $Q$. We thus have to initialize its copies not only through the input itself, but also through the executable; this works analogously to the SYNC case. We then have $S' \equiv C'_{[s,s,\ldots]}[P'[Q'_{[q,q,\ldots]}]] \xrightarrow{\mathbf{in}(c,M)} C'_{[s,s,\ldots]}[P''[Q'_{[q,q,\ldots]}]]$ and, if $x \notin \omega_P \cup \omega_Q$, maintaining form symmetry and equivalent labels:

$$D' \equiv C'_{[s_P,\ldots]}[P'[\langle\sigma_{S'}\rangle.\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}]$$
$$\xrightarrow{\tau.\ldots.\tau.\mathbf{in}(c,M\{^x/_{x_P}\}_{x\in\mathrm{vars}(C[P[Q]])})} C'_{[s_P,\ldots]}[P''[\langle\sigma_{S'}\rangle.\mathtt{exe}''_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q''_{[q_Q,\ldots]}]$$

No preshared variables are initialized, so restriction symmetry holds, and since no output occurs, leak symmetry holds. This, however, is also true if $x \in \omega_P \cup \omega_Q$, where $S' \equiv C'_{[s,s,\ldots]}[P'[Q'_{[q,q,\ldots]}]] \xrightarrow{\mathbf{in}(c,M)} C'_{[s,s,\ldots]}[P''[Q'_{[q,q,x,x,\ldots]}]]$ and for the disjoint case:

$$D' \equiv C'_{[s_P,\ldots]}[P'[\langle\sigma_{S'}\rangle.\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}]$$
$$\xrightarrow{\tau.\ldots.\tau.\mathbf{in}(c,M\{^x/_{x_P}\}_{x\in\mathrm{vars}(C[P[Q]])})} C'_{[s_P,\ldots]}[P''[\langle\sigma_{S'}\rangle.\mathtt{exe}''_{[q_P,x_P,\ldots]}]] \parallel C'_{[s_Q,\ldots]}[Q''_{[q_Q,x_Q,\ldots]}]$$

OUT-T If $c \notin \mathcal{E}_{S'}$, then by construction of $\mathcal{E}_{D'}$, $\delta_P(c) \notin \mathcal{E}_{D'}$. Now since $w_n\varphi_{S''} = t\sigma_{S'}$, then by construction of $\varphi_{D''}$ we have $w_n\varphi_{D''} = t\sigma_{D'}$.

If $t$ is not the value of a leakable shared variable, i.e., $\neg(\exists x \in \mathrm{vars}(C[P[Q]]) \cap (\omega_P \cup \omega_Q)\colon x\sigma_{S'} =_\mathrm{E} t)$, we omit the sets of leakable shared variables as they do not change and have $S' \equiv C'[P'[Q']] \xrightarrow{\nu w_n.\mathbf{out}(c,w_n)} C'[P''[\langle\sigma_{S'}\rangle.\mathtt{exe}']] \parallel C'[Q']$, symmetries hold, and labels are equivalent.

If $t$ is the value of a corruptible shared variable, or formally, $\exists x \in \mathrm{sh}(C[P[Q]]) \cap (\omega_P \cup \omega_Q)\colon x\sigma_{S'} =_\mathrm{E} t$, we have to leak it twice in $S'$ in order to be able to have both copies leaked in $D'$. Thus, when:

$$S' \equiv C'_{[s,s,x,x,\ldots]}[P'[Q'_{[q,q,\ldots]}]]$$
$$\xrightarrow{\nu w_n.\mathbf{out}(lc,w_n)}\xrightarrow{\nu w_{n+1}.\mathbf{out}(lc,w_{n+1})} C''_{[s,s,\ldots]}[P'[Q'_{[q,q,\ldots]}]]$$

Then:

$$D' \equiv C'_{[s_P,x_P,\ldots]}[P'[\langle\sigma_{S'}\rangle.\mathtt{exe}'_{[q_P]}]] \parallel C'_{[s_Q,x_Q,\ldots]}[Q'_{[q_Q,\ldots]}]$$
$$\xrightarrow{\nu w_n.\mathbf{out}(lc,w_n)}\xrightarrow{\nu w_{n+1}.\mathbf{out}(lc,w_{n+1})} C''_{[s_P,\ldots]}[P'[\langle\sigma_{S'}\rangle.\mathtt{exe}'_{[q_P,\ldots]}]] \parallel C''_{[s_Q,\ldots]}[Q'_{[q_Q,\ldots]}]$$

Symmetries hold, and labels are equivalent (analogously for $P$ and $Q$ if the shared variable is transferred).

OUT-CH For an $\mathbf{out}(c,h)$ we merely have to show $c, h \notin \mathcal{E}_{S'}$ implies $\delta_P(c), \delta_P(h) \notin \mathcal{E}_{D'}$ – which is immediate from the definition of $D'$.

The symmetries hold: This transition does not affect a name restriction, send out a name and not a variable, so – omitting the respective sets of leakable shared variables as they do not change – for $S' = C'[P'[Q']] \xrightarrow{\mathbf{out}(c,h)} C'[P''[Q']]$ we have $D' = C'[P'[\langle\mu\rangle.\mathtt{exe}']] \parallel C'[Q'] \xrightarrow{\mathbf{out}(c,h)} C'[P''[\langle\mu\rangle.\mathtt{exe}']] \parallel C'[Q']$. Due to the latter, the labels on the traces are equivalent as well.

Aside from that we have nothing to show since none of the components of the extended process aside from the (non-extended) process are changed.

OPEN-CH In this case we have an output $\mathbf{out}(c,h)$ such that $c \notin \mathcal{E}_{S'}$, $h \in \mathcal{E}_{S'}$ and $a$ is a fresh name. By inductive hypothesis and construction of $\mathcal{E}_{D'}$ we then have $\delta_P(c) \notin \mathcal{E}_{D'}$, $\delta_P(h) \in \mathcal{E}_{D'}$ and by construction of $\sigma_{D'}$, $\delta_P(a)$ is fresh.

No name restrictions are involved and no variables are sent out, so restriction and leak symmetry are maintained.

We again omit the leakable shared variables as they are unaffected and have $S' = C'[P'[Q']] \xrightarrow{\nu a.\mathbf{out}(c,a)} C'[P''[Q']]\{^h/_a\}$, so $D' = C'[P'[\langle\mu\rangle.\texttt{exe}']] \parallel C'[Q'] \xrightarrow{\nu a.\mathbf{out}(c,a)} C'[P''[\langle\mu\rangle.\texttt{exe}']] \parallel C'[Q']\{^h/_a\}$. On the one hand, we thus have equivalent transition labels on the trace. On the other hand, since $\mathrm{names}(P) \cap \mathrm{names}(Q) = \emptyset$ and $a \in \mathrm{names}(P)$, we have $C'[P''[Q']]\{^h/_a\} \equiv C'[(P''\{^h/_a\})[Q']]$, while also $C'[P''[\langle\mu\rangle.\texttt{exe}']] \parallel C'[Q']\{^h/_a\} \equiv C'[(P''\{^h/_a\})[\langle\mu\rangle.\texttt{exe}']] \parallel C'[Q']$, maintaining form symmetry.

This concludes the case distinction on the applied transition rule.

Recall that we currently aim to prove $\lambda_S \sim \lambda_D$; we have seen that the visible actions on the trace are the same, so we now need to show the following.

$$\begin{aligned} & M\varphi_{S'} =_{\mathrm{E}} N\varphi_{S'} \\ \Leftrightarrow\ & M\{^x/_{x_P}\}_{x\in\mathrm{sh}(C[P[Q]])}\varphi_{D'} =_{\mathrm{E}} N\{^x/_{x_P}\}_{x\in\mathrm{sh}(C[P[Q]])}\varphi_{D'} \end{aligned}$$

The case of $M\{^x/_{x_Q}\}_{x\in\mathrm{sh}(C[P[Q]])}$ is analogous.

By Lemma A.2 we assume without loss of generality that $M\varphi_{S'}$ and $N\varphi_{S'}$ are collapsed. It thus follows with Lemma A.8 that $\delta_P(M\varphi_{S'}) =_{\mathrm{E}} \delta_P(N\varphi_{S'})$. Since by definition of $\varphi_{D'}$, we have that $\delta_P(M\varphi_{S'}) =_{\mathrm{E}} M\{^x/_{x_P}\}_{x\in\mathrm{sh}(C[P[Q]])}\varphi_{D'}$ and $\delta_P(N\varphi_{S'}) =_{\mathrm{E}} N\{^x/_{x_P}\}_{x\in\mathrm{sh}(C[P[Q]])}\varphi_{D'}$ we conclude $M\{^x/_{x_P}\}_{x\in\mathrm{sh}(C[P[Q]])}\varphi_{D'} =_{\mathrm{E}} N\{^x/_{x_P}\}_{x\in\mathrm{sh}(C[P[Q]])}\varphi_{D'}$.

So far, we haven proven that $\lambda_S \sim \lambda_D$ for $S \xrightarrow{\lambda_S} S'$ and $D \xrightarrow{\lambda_D} D'$ with $D = \varsigma(C[P[\langle\sigma_{S'}\rangle.\texttt{exe}]], \omega_P)\{^x/_{x_P}\}_{x\in\mathrm{vars}(C[P[\langle\mu\rangle.\texttt{exe}]])} \parallel \varsigma(C[Q], \omega_Q)\{^x/_{x_Q}\}_{x\in\mathrm{vars}(C[Q])}$. It remains to show that $\langle\sigma_{S'}\rangle.\texttt{exe}$ is a $\mu$-executable.

For brevity, we define the following two abbreviations.

$$\begin{aligned} \mu_P &:= \mu\{^x/_{x_P}\}_{x\in\mathrm{vars}(\varsigma(C[P[Q]],\omega_P\cup\omega_Q))} \\ \mu_Q &:= \mu\{^x/_{x_Q}\}_{x\in\mathrm{vars}(\varsigma(C[P[Q]],\omega_P\cup\omega_Q))} \end{aligned}$$

In a first step, we will show that all variables $x_P \in \mathrm{dom}(\sigma_{D'})$ adhere to $\mu_P$: If both variables in an equation $x_P =_{\mathrm{E}} y_P \in \mu_P$ have been initialized, then $x_P\sigma_{D'} =_{\mathrm{E}} y_P\sigma_{D'}$. In a second step, we will then show that if $\langle\sigma_{S'}\rangle.\texttt{exe}'$ is executed further within $D'$, then all $x_P \in \mathrm{dom}(\mu_P)$ are set up to adhere to $\mu_P$.

So first, consider an equation $x_P =_{\mathrm{E}} y_P \in \mu_P$ such that $x_P, y_P \in \mathrm{dom}(\sigma_{D'})$. By definition of $\sigma_{D'}$, this means that $x_Q, y_Q \in \mathrm{dom}(\sigma_{D'})$. We assume that $\varsigma(C[Q], \omega_Q) \Vdash \mu$, which implies that $\varsigma(C[Q], \omega_Q)\{^x/_{x_Q}\}_{x\in\mathrm{vars}(\varsigma(C[P[Q]],\omega_P\cup\omega_Q))} \Vdash \mu_Q$. By Corollary 5.3 we therefore have $\varsigma(C[P[\langle\sigma_{S'}\rangle.\texttt{exe}]], \omega_P)\{^x/_{x_P}\}_{x\in\mathrm{vars}(C[P[\langle\mu\rangle.\texttt{exe}]])} \parallel \varsigma(C[Q], \omega_Q)\{^x/_{x_Q}\}_{x\in\mathrm{vars}(C[Q])} \Vdash \mu_Q$. This means that for every $x_Q, y_Q \in \mathrm{dom}(\mu_Q) \cap \mathrm{dom}(\sigma_{D'})$, if $x_Q =_{\mathrm{E}} y_Q \in \mu_Q$, then $x_Q\sigma_{D'} =_{\mathrm{E}} y_Q\sigma_{D'}$. By definition of $\sigma_{D'}$ we thus have $\delta_Q(x\sigma_{S'}) =_{\mathrm{E}} \delta_Q(y\sigma_{S'})$.

We will now prove that removing the name separation retains the equality.

**Lemma 5.21.** *Let $s$ and $t$ be collapsed terms. Let $R \in \{P, Q\}$, and let $K \in \{P, Q\} \setminus \{R\}$. If $\delta_R(s) =_{\mathrm{E}} \delta_R(t)$, then $s =_{\mathrm{E}} t$.*

*Proof.* We prove this via induction over the size of $s$ and $t$. By definition of name separation, there are four options, of which the first is the base case of the induction, and the last two work analogously and are thus conflated into one case.

1. $\delta_R(s) = n_{[s]_{=_{\mathrm{E}}}} =_{\mathrm{E}} n_{[t]_{=_{\mathrm{E}}}} = \delta_R(t)$. Then, $s \in [t]_{=_{\mathrm{E}}}$, which means that $s =_{\mathrm{E}} t$.

2. $\delta_R(s) = C[\delta_K(s_1), \ldots, \delta_K(s_n)] =_{\mathrm{E}} D[\delta_K(t_1), \ldots, \delta_K(t_m)] = \delta_R(t)$. Then we have $s = C[[s_1, \ldots, s_n]]$ and $t = D[[t_1, \ldots, t_m]]$ for some pure $\Sigma(K)$-contexts $C$ and $D$. By Lemma A.4 $s_1, \ldots, s_n, t_1, \ldots, t_m$ are collapsed, so we can apply the induction hypothesis; since $\delta_K$ thus preserves equivalences between the subterms, we can apply Lemma A.6 where $\delta_K(s_i) = n_{[s_i]_{=_{\mathrm{E}}}}$ and $\delta_K(t_j) = n_{[t_j]_{=_{\mathrm{E}}}}$ are the names on the right side of the equivalence. This yields $C[[s_1, \ldots, s_n]] =_{\mathrm{E}} D[[t_1, \ldots, t_M]]$ and thus $s =_{\mathrm{E}} t$.

3. $\delta_R(s) = n_{[s]_{=_{\mathrm{E}}}} =_{\mathrm{E}} D[\delta_K(t_1), \ldots, \delta_K(t_m)]$. Since $n_{[s]_{=_{\mathrm{E}}}} = C[n_{[s]_{=_{\mathrm{E}}}}]$ for some pure, but empty context $C := _$ this works analogously to the previous case by applying the induction hypothesis and Lemma A.6. □

Since $\sigma_{S'}$ is collapsed, so are $x\sigma_{S'}$ and $y\sigma_{S'}$; thus, with this lemma, we know that $x\sigma_{S'} =_{\mathrm{E}} y\sigma_{S'}$. Applying Lemma A.8 now yields $\delta_P(x\sigma_{S'}) =_{\mathrm{E}} \delta_P(y\sigma_{S'})$, which by definition of $\sigma_{D'}$ implies $x_P\sigma_{D'} =_{\mathrm{E}} y_P\sigma_{D'}$.

Second, consider an equation $x_P =_{\mathrm{E}} y_P \in \mu_P$ such that $x_P \notin \mathrm{dom}(\sigma_{D'})$. If $y_P \in \mathrm{dom}(\sigma_{D'})$, then $y \in \mathrm{dom}(\sigma_{S'})$ and $\langle \sigma_{S'} \rangle$.exe contains the assignments $x_P := v_{[y_P \sigma_{\overleftarrow{S'}}]_{=_{\mathrm{E}}}}$ and $y_P := v_{[y_P \sigma_{\overleftarrow{S'}}]_{=_{\mathrm{E}}}}$, so there is a $D''$ such that $D' \Rightarrow D''$ where $\langle \sigma_{S'} \rangle$.exe is executed further and $x_P \sigma_{D''} = y_P \sigma_{D''}$.

If $y_P \notin \mathrm{dom}(\sigma_{D'})$, then $y \notin \mathrm{dom}(\sigma_{S'})$ and $\langle \sigma_{S'} \rangle$.exe contains the assignments $x_P := v_{[x]_{=_\mu}}$ and $y_P := v_{[y]_{=_\mu}}$, but since $x_P =_{\mathrm{E}} y_P \in \mu_P$, $v_{[x]_{=_\mu}} = v_{[y]_{=_\mu}}$ there is a $D''$ such that $D' \Rightarrow D''$ where $\langle \sigma_{S'} \rangle$.exe is executed further and $x_P \sigma_{D''} = y_P \sigma_{D''}$ – which concludes the proof. □

With Lemma 5.8 we now have $\lambda' \nvdash \alpha$, and thus $D \not\models \alpha$. Since $D$ is the parallel composition of two entirely disjoint protocols, we can follow with Theorem 5.2 that $\varsigma(C[P[\langle \mu \rangle$.exe$]], \omega_P) \not\models \alpha \vee \varsigma(C[Q], \omega_Q) \not\models \alpha$, which proves the contraposition and concludes the proof. □

# 6. Applications

When analyzing security protocols, automated protocol verifiers may run out of memory or fail to produce a result within a reasonable timeframe. This is not a problem that can be universally solved by increasing the available resources, since security properties are generally undecidable [56].

However, by reducing the analysis of large protocols to smaller subprotocols, compositional analysis can often circumvent this problem. We will now demonstrate how our result enables compositional analysis, and, with that, the analysis of larger and more complex protocols.

First, we discuss the compatibility of our result with existing protocol verifiers, before illustrating the usage of our composition theorem with a Diffie-Hellman key exchange and a data exchange protocol. We then perform a compositional analysis of Forward Secrecy in TLS 1.3 within the scope of RFC 8446 and the ECH extension. Finally, we provide an example of how our result can be used to enable compositional reasoning in other scenarios.

## 6.1. Tool Compatibility

Although there are to the best of our knowledge no dedicated tools for compositional reasoning, traditional protocol verifiers can still be used with a composition theorem: Prove properties of individual protocols with tool support, and derive properties of the composed protocol by applying the theorem. Our result is directly applicable to two state-of-the-art verification tools: ProVerif [15, 20] and SAPIC+ [31]. Both model protocols in a dialect of the Applied $\pi$-Calculus. Even though the ProVerif dialect is richer than the core Applied $\pi$-Calculus we use here, they are actually equivalent. In [17] a translation from the former to the latter is introduced and proven to produce equivalent traces (cf. Chapter 4).

Similarly, it is shown in [31] that a SAPIC+ dialect process and its translation to ProVerif dialect produce the same traces. Since the latter is equivalent to core Applied $\pi$-Calculus, by transitivity, core Applied $\pi$-Calculus is equivalent to the SAPIC+ dialect.

This leaves the properties: Are all properties expressible in the tools covered by our result? We did not fix any specification language but consider trace properties to be sets of traces closed under static equivalence. Trace sets are encoded by *queries* in ProVerif and by *trace formulas* in SAPIC+; if a query or trace formula is defined such that it is closed under static equivalence, it is therefore covered by Theorem 5.7.

In conclusion, our result can be applied to protocols and trace properties analyzed with both ProVerif and SAPIC+; however, since Theorem 5.7 only covers trace properties, we cannot use it for a compositional analysis of equivalence properties. We make use of this observation in our case studies we present in the next sections.

## 6.2. Diffie-Hellman and a Data Exchange

Recall the data exchange protocol $XP$ we modeled over the course of Section 4.2. It models the repeated transfer of a data item from one party to another.

$$XP :\equiv !_.(\nu data.\mathbf{out}(xc, \mathtt{senc}(data, k_A^{\mathrm{sess}})) \parallel \mathbf{in}(xc, t).x := \mathtt{sdec}(t, k_B^{\mathrm{sess}}))$$

As you can see, $XP$ contains a hole; this is since we do not care how the two session keys $k_A^{\mathrm{sess}}$ and $k_B^{\mathrm{sess}}$ are created. The only requirement is that they are equal, i.e., they are covered by the set-up guarantee $k_A^{\mathrm{sess}} =_{\mathrm{E}} k_B^{\mathrm{sess}}$.

We also modeled a Diffie-Hellman key exchange $DH$ that satisfies this set-up guarantee, which we slightly extend to an authenticated version: The two parties sign their respective half keys, and only process the other party's half key if its signature is valid. For this, we use the symbols $\mathtt{sign}\langle 2\rangle$ for signing a message with a private key, $\mathtt{check}\langle 2\rangle$ for verifying the signature of a message with a public key, and $\mathtt{get}\langle 1\rangle$ for retrieving the message from a signature. The corresponding equational theory consists of the equations $\mathtt{check}(\mathtt{sign}(m,k),\mathtt{pk}(k)) = m$ and $\mathtt{get}(\mathtt{sign}(m,k)) = m$.

$$\begin{aligned} DH :\equiv\ & (\nu a.(\mathbf{out}(c, \mathtt{sign}(\mathtt{mexp}(g,a), sk_A)) \\ & \quad \| \mathbf{in}(d,B).\, \mathbf{if}\ \mathtt{check}(B,\mathtt{pk}(sk_B)) = \mathtt{get}(B)\ \mathbf{then}\ k_A^{\mathrm{sess}} := \mathtt{mexp}(\mathtt{get}(B),a)\ \mathbf{else}\ 0)) \\ & \| (\nu b.(\mathbf{out}(d, \mathtt{sign}(\mathtt{mexp}(g,b), sk_B)) \\ & \quad \| \mathbf{in}(c,A).\, \mathbf{if}\ \mathtt{check}(A,\mathtt{pk}(sk_A)) = \mathtt{get}(A)\ \mathbf{then}\ k_B^{\mathrm{sess}} := \mathtt{mexp}(\mathtt{get}(A),b)\ \mathbf{else}\ 0)) \end{aligned}$$

For $DH$ to work properly, we need the two private keys $sk_A$ and $sk_B$, from which we can derive the respective public keys, and a generator $g$. We put these values into the composition context as they are pre-shared values agreed on beforehand.

$$C :\equiv \nu sk_A.\nu sk_B.\nu g._{_}$$

Finally, recall the equational theories for $XP$ and $DH$, which specify how symmetric encryption and modular exponentiation work. To the latter, we add the two equations modeling signatures. The key derivation function in our model does not need an equational theory; we assume it to essentially work like a hash function.

$$\begin{aligned} \mathrm{E}(XP) &:= \{\mathtt{sdec}(\mathtt{senc}(x,k),k) = x\} \\ \mathrm{E}(DH) &:= \{\mathtt{mexp}(\mathtt{mexp}(g,a),b) = \mathtt{mexp}(\mathtt{mexp}(g,b),a), \\ &\qquad \mathtt{check}(\mathtt{sign}(m,k),\mathtt{pk}(k)) = m, \mathtt{get}(\mathtt{sign}(m,k)) = m\} \end{aligned}$$

In combination, we now have $C[XP[DH]]$ and the associated equational theories as a monolithic model of the data exchange protocol. We want to know if it provides Forward Secrecy. Theorem 5.7 allows us to ignore the specific key exchange, so instead of analyzing $C[XP[DH]]$, we can define Forward Secrecy over the corrupted version of $C[DH]$.

$$\begin{aligned} FS :=& \{(tr, \nu\mathcal{E}.\varphi) \in \mathrm{tr}(\varsigma(C[DH], \{sk_A, sk_B\})) \mid \\ & \exists i\colon w_i \in \mathrm{dom}(\varphi) \wedge w_i\varphi = \mathtt{sign}(\mathtt{mexp}(g,b), sk_B) \wedge \varphi \vdash k_A^{\mathrm{sess}} \\ & \Rightarrow \exists j\colon w_j \in \mathrm{dom}(\varphi) \wedge (w_j\varphi = sk_A \vee w_j\varphi = sk_B) \wedge j < i\} \end{aligned}$$

This states Forward Secrecy of $k_A^{\mathrm{sess}}$: If the message containing $B$'s half key has been sent from $B$ to $A$ (i.e., $A$ should be able to calculate $k_A^{\mathrm{sess}}$), and if the adversary is able to deduce $k_A^{\mathrm{sess}}$, then at least one of the secret long-term keys $sk_A$ and $sk_B$ must have been leaked at an earlier point in time. Forward Secrecy of $k_B^{\mathrm{sess}}$ is defined analogously.

Before we apply Theorem 5.7 to $XP$ and $DH$, we have to ensure that it is applicable. First, $FS$ is closed under static equivalence: Informally, every leak is a visible action on the trace and reflected in the adversary knowledge – which both have to be equal for two traces to be statically equivalent. Next, we argue that the requirements of the theorem hold.

**1. Disjoint Primitives** The signatures of $XP$, $\Sigma(XP) = \{\mathtt{senc}\langle 2\rangle, \mathtt{sdec}\langle 2\rangle, \mathtt{kdf}\langle 1\rangle\}$, and $DH$, $\Sigma(DH) = \{\mathtt{mexp}\langle 2\rangle, \mathtt{sign}\langle 2\rangle, \mathtt{check}\langle 2\rangle, \mathtt{get}\langle 1\rangle\}$, are clearly disjoint.

**2. Syntactic Requirements** We have to check whether $\mathrm{fv}(DH) = \{sk_A, sk_B, g\} \subseteq \mathrm{vars}(C)$ and $\mathrm{fv}(XP) \setminus \mathrm{vars}(C) = \{k_A^{\mathrm{sess}}, k_B^{\mathrm{sess}}\} \subseteq \{A, B, k_A^{\mathrm{sess}}, k_B^{\mathrm{sess}}\} = \mathrm{bv}(DH)$, both of which holds. We also analyze both $DH$ and $XP$ under the assumption that $sk_A$ and $sk_B$ can be leaked.

**3. Non-shared Channels** Neither $XP$ nor $DH$ have variables in place of a channel name.

**4. Secrecy of Shared Variables** Since $sk_A$, $sk_B$ and $g$ only occur in positions where they are not retrievable through the equational theory, they are not revealed in $C[XP[\langle k_A^{\mathrm{sess}} =_{\mathrm{E}} k_B^{\mathrm{sess}}\rangle.\,\mathtt{exe}]]$ and $C[DH]$. $k_A^{\mathrm{sess}}$ and $k_B^{\mathrm{sess}}$ remaining unknown to the adversary is an essential property of $DH$, and in $XP$ they, too, are not retrievable through the equational theory. None of the shared values are thus revealed.

**5. Freshness of Shared Variables** The shared variables are $sk_A$, $sk_B$, $g$, $k_A^{\mathrm{sess}}$ and $k_B^{\mathrm{sess}}$. The former three are used only in $DH$, and only to create new terms unequal to the respective variables. The latter two are used in the same way in $XP$, while the values they get assigned in $DH$ are by construction different to those of the non-shared variables $A$ and $B$.

Thus, all requirements of Theorem 5.7 are satisfied, and we can use it to conclude that $\varsigma(C[XP[DH]], \{sk_A, sk_B\}) \vDash FS$ – if we can show that $\varsigma(C[DH], \{sk_A, sk_B\}) \Vdash \{k_A^{\text{sess}} =_{\text{E}} k_B^{\text{sess}}\}$ and $\varsigma(C[DH], \{sk_A, sk_B\}) \vDash FS$ hold. We do so in a ProVerif model, which we provide in [39][1]. As a sanity check, we also perform a monolithic analysis of $\varsigma(C[XP[DH]], \{sk_A, sk_B\})$ and show that it does in fact provide $FS$.

## 6.3. A Compositional Analysis of Forward Secrecy in TLS 1.3 ECH

TLS is one of the most widely used protocols and crucial for protecting internet communication. It is also one of the most thoroughly examined protocols, with automated verification playing a major rule in the standardization of TLS 1.3. It solved several privacy issues as compared to TLS 1.2, but important metadata, including the intended communication partner of the client, is still visible in plaintext. To mitigate these issues, the Encrypted Client Hello (ECH) extension is currently under standardization [60].

We will now do a compositional analysis of Forward Secrecy of the Master Secret in TLS 1.3 ECH to illustrate how Theorem 5.7 can be used to improve automated analysis. For this, we split TLS in two subprotocols and a composition context such that our result is applicable.

A TLS 1.3 ECH handshake can be roughly divided into three phases: The *Key Exchange*, the *Authentication Phase* and the *Data Exchange*. The Key Exchange performs a Diffie-Hellman key exchange and derives several keys, including the *master secret* as the basis for all upcoming session keys. During the Authentication Phase, the server authenticates itself to the client and vice versa. The Data Exchange finally transfers application data, encrypted with session keys derived from the keys established earlier. Additionally, both client and server have some pre-established keys.

Put in the terminology of Theorem 5.7, we can now model the Key Exchange up to the derivation of the master secret as a process $KEx$, while the Authentication Phase and Data Exchange form a partial process $AD :\equiv _.Auth.Data$. Pre-established keys are set up in a composition context $C$.

Then, TLS is modeled by $C[AD[KEx]]$, and we can analyze whether Forward Secrecy holds for the master secret on $KEx$ alone if the private ECH key $dk_F$ of the server is corruptible. We write $FS(ms)$.

While we could split TLS a few steps into $AD$, we would gain less from a compositional analysis since we would have to analyze a larger part of the protocol.

We can now use Theorem 5.7 to perform this compositional analysis: As already argued in Section 6.2, Forward Secrecy is closed under static equivalence; we also observe that $KEx$ can be abstracted by an executable satisfying the set-up guarantee $\mu := \{ms = k_{h,c}, ms = k_{h,s}, ms = k_{m,c}, ms = k_{m,s}\}$, which equates all the keys derived at the end of $KEx$. Finally, we will go through the theorem requirements.

**1. Disjoint Primitives** $KEx$ uses HPKE, modular exponentiation, and three different KDFs $\texttt{kdf}_0$, $\texttt{kdf}_{hs}$ and $\texttt{kdf}_{ms}$; $AD$ uses symmetric encryption, signatures, MACs, and other KDFs $\texttt{kdf}_k$ and $\texttt{kdf}_{psk}$. Note that all of these KDFs are configured to produce different keys even for the same input, so they can safely be modeled as different functions.

**2. Syntactic Requirements** The free variables of $KEx$, i.e., the public-private key pairs, are bound by $C$; the remaining free variables of $AD$ (all derived keys, e.g., $ms$) are bound by $KEx$. Finally, $dk_F$ is corruptible in both parts of TLS.

**3. Non-shared Channels** No shared variable is used in the place of a channel name, and since shared variables are fresh with respect to non-shared variables, none of their values is used as a channel name.

**4. Secrecy of Shared Variables** Pre-shared variables, namely $sk_C$, $pk_C$, $psk_{C,S}$, $sk_S$, $pk_S$, $sk'_S$, $pk'_S$, $dk_F$, and $ek_F$, are only used to encrypt or decrypt terms and thus not deducible; this is also true for transferred variables in $AD$, i.e., the keys $ms$, $k_{h,c}$, $k_{h,s}$, $k_{m,c}$, and $k_{m,s}$. In $KEx$, transferred variables cannot be deduced since all keys have the same value as $ms$ and Forward Secrecy holds for $ms$.

**5. Freshness of Shared Variables** All pre-shared variables are freshly generated through a name restriction, and the transferred variables are constructed such that they do not share a value with a non-shared variable. Additionally, no value of a shared variable is copied into another variable. Shared variables are thus fresh.

An extensive analysis of TLS 1.3 ECH using ProVerif has been performed in [14]. The authors analyze 14 security properties in 480 different scenarios of different protocol functionalities and behaviors, agents involved, and key compromise scenarios. However, they can only prove 63% of the properties due to ProVerif running out of time or memory.

We base our analysis on the most recent version of the ProVerif model used for this analysis[2]; it covers the scope of RFC 8446 and the ECH extension, i.e., the entirety of TLS itself, but not attacks on the

[1] `/dh/dh_auth.pv` in the `secure_composition_under_dynamic_corruption_case_studies.zip` archive

[2] Last updated September 1st, 2022: https://gitlab.cs.ox.ac.uk/vinval/ech-tls/-/tree/9fd6b2a0523087d41ba480c6180b9da92833ca50

wider TLS ecosystem or different versions of TLS. We extend the model with a forward secrecy query on the master secret and implement a composition mode, which allows us to measure the improvements of a compositional verification approach. The composition mode corresponds to an analysis of $\varsigma(C[KEx], \{dk_F\})$; the standard mode analyzes the full protocol monolithically, corresponding to $\varsigma(C[AD[KEx]], \{dk_F\})$.

In this updated model, we evaluate both queries in a representative subset of 32 relevant scenarios and compare the runtime of a compositional and monolithic analysis with the same timeout of 48 hours as in [14]. The scenarios covered include all functionalities provided in the original model; we exclude scenarios that are not relevant for the comparision, as they, e.g., do not allow ECH clients in the execution. Our server features an Intel® Xeon® CPU E5-4650L with 2.60 GHz and 1 TB of RAM. The results of our benchmark are shown in Table 1.

We observe substantial speed-ups in all but four cases and a reduced memory usage of at least 57 % in all cases, sometimes going as far as 99 %. We especially see high improvements in scenarios that allow for complex interactions after the initial handshake is finished. This is since our composition result allows us to cut out post-handshake interactions in a compositional analysis.

We also observe that several scenarios need similar or equal resources for a compositional analysis. The differences in the respective scenarios are only relevant after the initial handshake is concluded, and they therefore do not affect the compositional analysis. Consider for example the scenarios 2222 and 3222, which need exactly the same amount of time and memory for a compositional analysis. This is since the only difference between the **F** scenarios 2 and 3 is that 3 allows sending post-handshake data, while 2 does not. Post-handshake data, however, is not considered by the compositional analysis and therefore does not affect its performance.

The same mechanism explains the four scenarios 7222, 7224, 8222, and 8224. Here, compositional analysis is barely faster, exactly as fast or even slightly slower than a monolithic analysis. Since the scenarios consider only a very limited amount of post-handshake interaction, a compositional analysis has to consider very similar parts of TLS 1.3 as a monolithic analysis and therefore also needs a very similar amount of resources.

In summary, compositional analysis saves us 71 % of time and 86 % of memory on average, with most numbers consistently ranging from 60 % to 95 %. We therefore expect these results to scale to more extensive investigations of TLS 1.3 ECH as well as other complex protocols.

## 6.4. Iterative Composition

Theorem 5.7 can not only be used for compositional analysis, but it can also serve as the basis for other composition results. In this section, we give an example for such a corollary and prove that we can securely compose protocols with a chain-like structure arbitrarily often with themselves, given that they are properly *tagged*.

We consider a process tagged if every instantiation of it uses primitives that add a unique *tag* to its outputs. Then the messages of a specific process can only be used by that specific process. In its most basic form, tagging means that instead of using a function $f$, we use a function $f_i$. As discussed in Section 3.2, the decisive factor here is that the functions produce different outputs from the same input.

Our corollary now assumes a protocol that works by repeatedly calling upon itself and that is tagged differently for every iteration. Then it suffices to prove that a trace property holds for one iteration to conclude that the property holds for an arbitrary number of iterations.

How do we prove this? The central observation is that a protocol running $n$ iterations is nothing more than a protocol running $n-1$ iterations composed with a protocol running a single iteration. Since the protocol is tagged, every iteration satisfies the disjointness criteria to inductively apply Theorem 5.7. Then we only need to verify a single iteration to prove a property for an arbitrary number of runs.

This approach could be used in the verification of many practically relevant protocols. One-Time Password protocols often rely on a hash chain, and the Double Ratchet used in the Signal protocol essentially calls itself repeatedly. The approach can also be applied to TLS 1.3: The session keys are derived from, amongst others, transcripts that strictly increase in size for every iteration. Given the collision-resistance of the hash and key derivation functions, we can consider the transcript length as a tag for the top-level key derivation functions and can thus consider them as disjoint KDF functions.

In order to formally and concisely state our corollary, we first introduce some additional notation.

Consider a partial process $P$ built over $\Sigma(P)$ and an executable $Q$. $P$ is the protocol we want to iterate, and $Q$ provides the initial setup for the first iteration of $P$. We call $P$ *iterable* if for every free, i.e., shared,

Table 1: Performance Comparison

**F** refers to different optional functionalities (e.g., early data, HRR); **S** determines the compromised keys, cipher suits, and groups; **A** specifies the allowed agent behaviors (e.g., PSK, client certificates); **B** specifies the instantiated agents types (ECH or non-ECH). A detailed description of all scenarios is available in `/tls_ech/README.md` of [39]. If an analysis exceeds a runtime of 48 h, we note this by *timeout* and give the memory used at this point in time.

| **F** | **S** | **B** | **A** | **monolithic** | | **compositional** | | **improvement** | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | time (hh:mm) | memory (GB) | time (hh:mm) | memory (GB) | time | memory |
| 2 | 2 | 2 | 2 | 00:11 | 02.64 | 0:06 | 0.66 | 45 % | 75 % |
| 2 | 2 | 2 | 4 | 00:16 | 03.49 | 0:07 | 0.76 | 56 % | 78 % |
| 2 | 2 | 3 | 2 | 00:48 | 08.08 | 0:06 | 0.79 | 88 % | 90 % |
| 2 | 2 | 3 | 4 | 01:22 | 12.25 | 0:08 | 0.79 | 90 % | 94 % |
| 3 | 2 | 2 | 2 | 00:12 | 02.64 | 0:06 | 0.66 | 50 % | 75 % |
| 3 | 2 | 2 | 4 | 00:17 | 03.49 | 0:07 | 0.76 | 59 % | 78 % |
| 3 | 2 | 3 | 2 | 00:49 | 08.08 | 0:06 | 0.76 | 88 % | 91 % |
| 3 | 2 | 3 | 4 | 01:25 | 12.25 | 0:07 | 0.87 | 92 % | 93 % |
| 4 | 2 | 2 | 2 | 01:18 | 12.25 | 0:06 | 0.66 | 92 % | 95 % |
| 4 | 2 | 2 | 4 | 01:41 | 18.66 | 0:07 | 0.76 | 95 % | 96 % |
| 4 | 2 | 3 | 2 | 03:43 | 37.51 | 0:06 | 0.66 | 97 % | 98 % |
| 4 | 2 | 3 | 4 | 05:03 | 65.67 | 0:08 | 0.76 | 97 % | 99 % |
| 5 | 2 | 2 | 2 | 01:19 | 12.25 | 0:06 | 0.66 | 92 % | 95 % |
| 5 | 2 | 2 | 4 | 01:41 | 18.66 | 0:07 | 0.76 | 93 % | 96 % |
| 5 | 2 | 3 | 2 | 03:45 | 37.51 | 0:06 | 0.66 | 97 % | 98 % |
| 5 | 2 | 3 | 4 | 05:00 | 65.67 | 0:08 | 0.76 | 97 % | 99 % |
| 7 | 2 | 2 | 2 | 02:55 | 10.66 | 2:47 | 4.62 | 05 % | 57 % |
| 7 | 2 | 2 | 4 | 03:22 | 12.25 | 3:22 | 4.62 | 00 % | 62 % |
| 7 | 2 | 3 | 2 | 14:10 | 32.65 | 2:50 | 4.62 | 80 % | 86 % |
| 7 | 2 | 3 | 4 | 16:29 | 43.10 | 3:32 | 5.31 | 79 % | 88 % |
| 8 | 2 | 2 | 2 | 02:51 | 10.66 | 2:50 | 4.62 | 01 % | 57 % |
| 8 | 2 | 2 | 4 | 03:22 | 12.25 | 3:30 | 4.62 | −04 % | 62 % |
| 8 | 2 | 3 | 2 | 14:15 | 32.65 | 2:54 | 4.62 | 76 % | 86 % |
| 8 | 2 | 3 | 4 | 16:59 | 43.10 | 3:33 | 5.31 | 79 % | 88 % |
| 9 | 2 | 2 | 2 | 25:01 | 49.53 | 2:51 | 4.62 | 89 % | 91 % |
| 9 | 2 | 2 | 4 | 28:51 | 65.67 | 3:30 | 4.62 | 88 % | 93 % |
| 9 | 2 | 3 | 2 | timeout | 132.06 | 2:54 | 4.62 | N/A | N/A |
| 9 | 2 | 3 | 4 | timeout | 151.72 | 3:41 | 5.31 | N/A | N/A |
| 10 | 2 | 2 | 2 | 25:53 | 56.92 | 2:52 | 4.62 | 89 % | 92 % |
| 10 | 2 | 2 | 4 | 29:43 | 65.67 | 3:31 | 4.62 | 88 % | 93 % |
| 10 | 2 | 3 | 2 | timeout | 132.06 | 3:02 | 4.62 | N/A | N/A |
| 10 | 2 | 3 | 4 | timeout | 151.71 | 3:39 | 5.31 | N/A | N/A |
| | | | | | | | ∅ | 71 % | 86 % |

variable $x$, there is a bound variable $iter_x$ that contains the values passed on to the next iteration of $P$.

$$P \text{ iterable} :\Leftrightarrow \forall x \in \mathrm{fv}(P) \exists iter_x \in \mathrm{bv}(P)$$

Next, we tag $P$ by indexing all its function symbols and variables with the number of its iteration: For the $i^{\text{th}}$ iteration of $P$, we define $P_i$ as follows.

$$P_i :\equiv P\{^f/_{f_i}\}_{f\in\Sigma(P)}\{^{iter_x}/_{x_i}\}_{iter_x\in\mathrm{bv}(P)}\{^x/_{x_i}\}_{x\in\mathrm{bv}(P)}\{^x/_{x_{i-1}}\}_{x\in\mathrm{fv}(P)}$$

Note that we index the free variables with the number of the previous iteration, which allows us to pass data from one iteration of $P$ to the next through shared variables.

Analogously, we define $\Sigma(P_i) := \{f_i \mid f \in \Sigma(P)\}$ as the signature of $P_i$, and $\mathrm{E}(P_i) := \{s\{^f/_{f_i}\}_{f\in\Sigma(P)} = t\{^f/_{f_i}\}_{f\in\Sigma(P)} \mid s = t \in \mathrm{E}(P)\}$ as its equational theory. Then, for every $i, j$ with $i \neq j$, we have that $P_i \cap P_j = \emptyset$.

For the first iteration of $P$, we need to adapt $Q$ such that the shared variables of $P$ and $Q$ can properly synchronize. We do this by indexing them in $Q$ and define $Q_i :\equiv Q\{^x/_{x_0}\}_{x\in\mathrm{sh}(P[Q])}$. Note that here and in the remainder of this section, we assume a composition context $C :\equiv _$ and define $\mathrm{sh}(P[Q]) := \mathrm{sh}(C[P[Q]])$.

With this we can define *$n$ iterations of $P$*:

$$P^n[Q] :\equiv P_n[P_{n-1}[\ldots P_1[Q_0]]]$$

Both for dynamically corrupting $P^n[Q]$ and for specifying one of its trace properties we need to apply the tagging to the corruptible variables and the property. Thus, let $\omega$ be a set of corruptible variables of $P[Q]$. Then $\omega^n := \omega\{^x/_{x_1}\}_{x\in\mathrm{vars}(P)} \cup \cdots \cup \omega\{^x/_{x_n}\}_{x\in\mathrm{vars}(P)}$ contains all tagged corruptible variables in all iterations. Next, let $\alpha$ be a trace property defined over $P[Q]$. Then the following states that $\alpha$ holds after $n$ iterations of $P$.

$$\alpha^n := \alpha\{^f/_{f_n}\}_{f\in\Sigma(P)}\{^{iter_x}/_{x_n}\}_{iter_x\in\mathrm{bv}(P)}\{^x/_{x_n}\}_{x\in\mathrm{bv}(P)}\{^x/_{x_{n-1}}\}_{x\in\mathrm{fv}(P)}$$

Finally, we need to adapt the set-up guarantee $\mu$. If $\mu$ describes the setup of the $i^{\text{th}}$ iteration of $P$, we have $\mu_i := \mu\{^{iter_x}/_{x_{i-1}}\}_{x\in\mathrm{dom}(\mu)}$.

With the above definitions in place we can now state and prove *iterative composition*.

**Corollary 6.1.** *Let $P$ be an iterable partial process built over $\Sigma(P)$ with an equational theory $\mathrm{E}(P)$. Let $Q$ be an executable. Let $\omega \subseteq \mathrm{vars}(P[Q])$ be a subset of variables of $P[Q]$. Let $\mu$ be a set-up guarantee such that $\varsigma(Q,\omega) \Vdash \mu$, $\varsigma(P[Q],\omega) \Vdash \mu$ and $\mathrm{dom}(\mu) \subseteq \mathrm{trf}(P[Q])$. Finally, let $\alpha$ be a trace property defined over $P[Q]$. Then, if $P[Q]$ does not reveal any variable in $\mathrm{sh}(P[Q])$, every $x \in \mathrm{sh}(P[Q])$ is fresh relative to every $y \in \mathrm{vars}(P[Q]) \setminus \mathrm{sh}(P[Q])$ in $\varsigma(P[Q], \omega_P \cup \omega_Q)$, and $\mathrm{nv}(P[Q]) \cap \mathrm{sh}(P[Q]) = \emptyset$:*

$$\varsigma(P[Q],\omega) \vDash \alpha \Rightarrow \varsigma(P^n[Q_0],\omega^n) \vDash \alpha^n$$

*Proof.* Let $C :\equiv _$ be a composition context. Then $\varsigma(P^n[Q_0],\omega^n) \equiv \varsigma(C[P^n[Q_0]],\omega^n)$.

We proceed by induction over $n$; we use the induction to additionally prove that $\varsigma(P^n[Q_0],\omega^n) \Vdash \mu_{n+1}$.

The base case $\varsigma(P^1[Q_0],\omega^1) \vDash \alpha^1$ and $\varsigma(P^1[Q_0],\omega^1)\mu_2$ is a direct consequence of the required $\varsigma(P[Q],\omega) \vDash \alpha$ and $\varsigma(P[Q],\omega) \Vdash \mu$ save for renaming.

For the inductive step, assume that $\varsigma(P^n[Q_0],\omega^n) \Vdash \mu^{n+1}$. By definition, we vacuously have $\varsigma(P^n[Q_0],\omega^n) \vDash \alpha_{n+1}$ since $\alpha_{n+1}$ is defined over $\Sigma(P_{n+1})$, which is disjoint from $\Sigma(P_n)$. It remains to show that $\varsigma(P_{n+1}[Q_n], \omega_{n+1} \setminus \omega_n) \vDash \alpha_{n+1}$.

We know that $\varsigma(P[Q],\omega) \vDash \alpha$. However, with a few renamings, we can transform this into the required statement:

$$\begin{aligned}
&\varsigma(P[Q],\omega) \vDash \alpha \\
\Leftrightarrow\ &\varsigma(P[Q],\omega)\{^f/_{f_{n+1}}\}_{f\in\Sigma(P)}\{^{iter_x}/_{x_{n+1}}\}_{iter_x\in\mathrm{bv}(P)}\{^x/_{x_{n+1}}\}_{x\in\mathrm{bv}(P)}\{^x/_{x_n}\}_{x\in\mathrm{fv}(P)} \\
\vDash\ &\alpha\{^f/_{f_{n+1}}\}_{f\in\Sigma(P)}\{^{iter_x}/_{x_{n+1}}\}_{iter_x\in\mathrm{bv}(P)}\{^x/_{x_{n+1}}\}_{x\in\mathrm{bv}(P)}\{^x/_{x_n}\}_{x\in\mathrm{fv}(P)} \\
\Leftrightarrow\ &\varsigma(P_{n+1}[Q_n], \omega_{n+1} \setminus \omega_n) \vDash \alpha_{n+1}
\end{aligned}$$

With the same renamings we get from $\varsigma(Q,\omega) \Vdash \mu$ that $\varsigma(Q_n,\omega) \Vdash \mu_{n+1}$, and we have $\varsigma(P^n[Q_0],\omega^n) \Vdash \mu_{n+1}$ by inductive assumption.

Now, since all remaining requirements are satisfied by the requirements of this corollary, we can use Theorem 5.7 to conclude that $\varsigma(P_{n+1}[P^n[Q_0]], (\omega_{n+1} \setminus \omega_n) \cup \omega_n) \vDash \alpha_n$. However, this already proves the inductive step since $\varsigma(P_{n+1}[P^n[Q_0]], (\omega_{n+1} \setminus \omega_n) \cup \omega_n) = \varsigma(P^{n+1}[Q_0], \omega_{n+1})$.

To conclude the proof, we now need to show that $\varsigma(P^{n+1}[Q_0], \omega_{n+1}) \Vdash \mu_{n+2}$. Since this is also simply a trace property, it holds with the same reasoning as above. □

## 7. Conclusion and Future Work

We have proven that protocols can be securely composed in the presence of an adversary capable of dynamic corruption and retain their trace properties, as long as they use a disjoint set of cryptographic primitives. This result enabled us to perform a compositional analysis of Forward Secrecy in TLS 1.3 ECH. Notably, we were able to do so with a generic composition theorem that has not been tailored to any specific protocol but fits the framework of two major protocol verifiers, ProVerif and SAPIC+. In the future, we will adapt our results to Multiset Rewriting to also fit the framework of Tamarin.

Additionally, we intend to advance our work in three directions. First, we plan to further demonstrate the usefulness of our result through a large-scale case study. An obvious choice is to analyze TLS more extensively, but other large-scale protocols like SPDM are also viable candidates.

Second, we believe that our composition result can be useful in protocol analysis beyond merely speeding up automated verification. One example would be to use the contraposition of our main theorem for locating vulnerabilities in insecure protocols: Given an attack trace on a protocol and a separation of the protocol into two subprotocols that satisfy the requirements of our theorem, we can conclude that the cause for the vulnerability is in one of the sub-protocols alone. We plan to investigate this and identify other non-standard applications of composability.

Finally, while we have seen that our core composition result is already applicable to real-world protocols, its requirement of disjoint signatures will not be met by some deployed protocols. We plan to investigate alternative notions of disjointness to further increase the applicability of our theorem.

## References


[1] Martín Abadi, Bruno Blanchet, and Cédric Fournet. 2016. The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. doi:10.48550/ARXIV.1609.03003

[2] Martín Abadi and Cédric Fournet. 2001. Mobile Values, New Names, and Secure Communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL01)*. ACM, New York, NY, USA, 104–115. doi:10.1145/360204.360213

[3] Suzana Andova, Cas Cremers, Kristian Gjøsteen, Sjouke Mauw, Stig F. Mjølsnes, and Saša Radomirović. 2008. A framework for compositional verification of security protocols. *Information and Computation* 206, 2–4 (Feb. 2008), 425–459. doi:10.1016/j.ic.2007.07.002

[4] Myrto Arapinis, Vincent Cheval, and Stephanie Delaune. 2012. Verifying Privacy-Type Properties in a Modular Way. In *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE, Cambridge, MA, USA, 95–109. doi:10.1109/csf.2012.16

[5] Myrto Arapinis, Vincent Cheval, and Stéphanie Delaune. 2015. *Composing Security Protocols: From Confidentiality to Privacy*. Springer Berlin Heidelberg, Berlin, Heidelberg, 324–343. doi:10.1007/978-3-662-46666-7_17

[6] Various Authors. 2016. Thread on Banking Industry Concerns about TLS 1.3. IETF TLS Mailing List (tls@ietf.org). https://mailarchive.ietf.org/arch/browse/tls/?gbt=1&index=CzjJB1g0uFypY8UDdr6P9SCQBqA.

[7] Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK. doi:10.1017/cbo9781139172752

[8] Michael Backes, Markus Dürmuth, Dennis Hofheinz, and Ralf Küsters. 2006. *Conditional Reactive Simulatability*. Springer Berlin Heidelberg, Berlin, Heidelberg, 424–443. doi:10.1007/11863908_26

[9] Michael Backes, Birgit Pfitzmann, and Michael Waidner. 2004. *A General Composition Theorem for Secure Reactive Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 336–354. doi:10.1007/978-3-540-24638-1_19

[10] Michael Backes, Birgit Pfitzmann, and Michael Waidner. 2007. The Reactive Simulatability (RSIM) Framework for Asynchronous Systems. *Information and Computation* 205, 12 (Dec. 2007), 1685–1720. doi:10.1016/j.ic.2007.05.002

[11] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021. DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Vienna, Austria, 523–542. doi:10.1109/eurosp51992.2021.00042

[12] Karthikeyan Bhargavan, Abhishek Bichhawat, Pedram Hosseyni, Ralf Küsters, Klaas Pruiksma, Guido Schmitz, Clara Waldmann, and Tim Würtele. 2024. *Layered Symbolic Security Analysis in DY**. Springer Nature Switzerland, Cham, 3–21. doi:10.1007/978-3-031-51479-1_1

[13] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, USA, 483–502. doi:10.1109/sp.2017.26

[14] Karthikeyan Bhargavan, Vincent Cheval, and Christopher Wood. 2022. A Symbolic Analysis of Privacy for TLS 1.3 with Encrypted Client Hello. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. ACM, New York, NY, USA, 365–379. doi:10.1145/3548606.3559360

[15] 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.

[16] Bruno Blanchet. 2005. A Computationally Sound Mechanized Prover for Security Protocols. Cryptology ePrint Archive, Paper 2005/401.

[17] 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. doi:10.1561/3300000004

[18] Bruno Blanchet. 2017. Symbolic and Computational Mechanized Verification of the ARINC823 Avionic Protocols. In *30th IEEE Computer Security Foundations Symposium (CSF'17)*. IEEE, Santa Barbara, CA, USA, 68–82. doi:10.1109/csf.2017.7

[19] Bruno Blanchet. 2018. Composition Theorems for CryptoVerif and Application to TLS 1.3. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, Oxford, UK, 16–30. doi:10.1109/csf.2018.00009

[20] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. 2023. ProVerif 2.05: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial. https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf

[21] Florian Böhl and Dominique Unruh. 2014. Symbolic universal composability. *Journal of Computer Security* 24, 1 (Dec. 2014), 1–38. doi:10.3233/jcs-140523

[22] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. 2016. *Universal Composition with Responsive Environments*. Springer Berlin Heidelberg, Berlin, Heidelberg, 807–840. doi:10.1007/978-3-662-53890-6_27

[23] Jan Camenisch, Stephan Krenn, Ralf Küsters, and Daniel Rausch. 2019. *iUC: Flexible Universal Composability Made Simple*. Springer International Publishing, Cham, 191–221. doi:10.1007/978-3-030-34618-8_7

[24] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, IEEE, Newport Beach, CA, USA, 136–145. doi:10.1109/SFCS.2001.959888

[25] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. 2007. *Universally Composable Security with Global Setup*. Springer Berlin Heidelberg, Berlin, Heidelberg, 61–85. doi:10.1007/978-3-540-70936-7_4

[26] Ran Canetti and Tal Rabin. 2003. *Universal Composition with Joint State*. Springer Berlin Heidelberg, Berlin, Heidelberg, 265–281. doi:10.1007/978-3-540-45146-4_16

[27] Sofía Celi, Jonathan Hoyland, Douglas Stebila, and Thom Wiggers. 2022. *A Tale of Two Models: Formal Verification of KEMTLS via Tamarin.* Springer Nature Switzerland, Cham, 63–83. doi:10.1007/978-3-031-17143-7_4

[28] Vincent Cheval, Véronique Cortier, and Alexandre Debant. 2023. Election Verifiability with ProVerif. In *Proceedings of the 36th IEEE Computer Security Foundations Symposium (CSF'23)*. IEEE, Dubrovnik, Croatia, 43–58. doi:10.1109/csf57540.2023.00032

[29] Vincent Cheval, Véronique Cortier, and Eric le Morvan. 2015. Secure Refinements of Communication Channels. In *35th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 575–589. doi:10.4230/LIPICS.FSTTCS.2015.575

[30] Vincent Cheval, Veronique Cortier, and Bogdan Warinschi. 2017. Secure Composition of PKIs with Public Key Protocols. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, Santa Barbara, CA, USA, 144–158. doi:10.1109/csf.2017.28

[31] Vincent Cheval, Charlie Jacomme, Steve Kremer, and Robert Künnemann. 2022. SAPIC+: protocol verifiers of the world, unite!. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3935–3952. https://www.usenix.org/conference/usenixsecurity22/presentation/cheval

[32] Stefan Ciobâca and Véronique Cortier. 2010. Protocol Composition for Arbitrary Primitives. In *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE, Edinburgh, UK, 322–336. doi:10.1109/csf.2010.29

[33] Hubert Comon, Charlie Jacomme, and Guillaume Scerri. 2020. Oracle Simulation: A Technique for Protocol Composition with Long Term Shared Secrets. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. ACM, New York, NY, USA, 1427–1444. doi:10.1145/3372297.3417229

[34] Véronique Cortier and Stéphanie Delaune. 2008. Safely composing security protocols. *Formal Methods in System Design* 34, 1 (Oct. 2008), 1–36. doi:10.1007/s10703-008-0059-4

[35] Cas Cremers, Alexander Dax, and Aurora Naska. 2023. Formal Analysis of SPDM: Security Protocol and Data Model version 1.2. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, Anaheim, CA, USA, 6611–6628. https://www.usenix.org/conference/usenixsecurity23/presentation/cremers-spdm

[36] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. A Comprehensive Symbolic Analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1773–1788. doi:10.1145/3133956.3134063

[37] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. 2016. Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, USA, 470–485. doi:10.1109/SP.2016.35

[38] Cas Cremers, Benjamin Kiesl, and Niklas Medinger. 2020. A Formal Analysis of IEEE 802.11's WPA2: Countering the Kracks Caused by Cracking the Counters. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Berkeley, CA, United States, 1–17. https://www.usenix.org/conference/usenixsecurity20/presentation/cremers

[39] Cas Cremers, Erik Pallas, and Aleksi Peltonen. 2025. Secure Protocol Composition under Dynamic Corruption: Models and Proofs. Figshare repository. doi:10.6084/m9.figshare.29958290 ProVerif Models for the DHKE and TLS 1.3 ECH Case Studies, and Full Version of the Paper with all Proofs.

[40] Stéphanie Delaune, Steve Kremer, and Olivier Pereira. 2009. Simulation Based Security in the Applied Pi Calculus. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 169–180. doi:10.4230/LIPICS.FSTTCS.2009.2316

[41] Stéphanie Delaune, Steve Kremer, and Mark Ryan. 2008. Composition of Password-Based Protocols. In *2008 21st IEEE Computer Security Foundations Symposium*. IEEE, Pittsburgh, PA, USA, 239–251. doi:10.1109/csf.2008.6

[42] Joshua Gancher, Sydney Gibson, Pratap Singh, Samvid Dharanikota, and Bryan Parno. 2023. OWL: Compositional Verification of Security Protocols via an Information-Flow Type System. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1130–1147. doi:10.1109/sp46215.2023.10179477

[43] Sébastien Gondron and Sebastian Mödersheim. 2021. Vertical Composition and Sound Payload Abstraction for Stateful Protocols. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, Dubrovnik, Croatia, 1–16. doi:10.1109/csf51468.2021.00038

[44] Mike Graf, Ralf Küsters, and Daniel Rausch. 2023. AUC: Accountable Universal Composability. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1148–1167. doi:10.1109/sp46215.2023.10179384

[45] Thomas Groß and Sebastian Mödersheim. 2011. Vertical Protocol Composition. In *2011 IEEE 24th Computer Security Foundations Symposium*. IEEE, Cernay-la-Ville, France, 235–250. doi:10.1109/csf.2011.23

[46] J.D. Guttman and F.J.F. Thayer. 2000. Protocol independence through disjoint encryption. In *Proceedings 13th IEEE Computer Security Foundations Workshop. CSFW-13 (CSFW-00)*. IEEE Comput. Soc, Cambridge, UK, 24–34. doi:10.1109/csfw.2000.856923

[47] Joshua D. Guttman. 2009. *Cryptographic Protocol Composition via the Authentication Tests*. Springer Berlin Heidelberg, Berlin, Heidelberg, 303–317. doi:10.1007/978-3-642-00596-1_22

[48] Christoph G. Günther. 1989. *An Identity-Based Key-Exchange Protocol*. Springer Berlin Heidelberg, Berlin, Heidelberg, 29–37. doi:10.1007/3-540-46885-4_5

[49] Dennis Hofheinz and Victor Shoup. 2013. GNUC: A New Universal Composability Framework. *Journal of Cryptology* 28, 3 (Oct. 2013), 423–508. doi:10.1007/s00145-013-9160-y

[50] Ralf Küsters, Max Tuengerthal, and Daniel Rausch. 2020. The IITM Model: A Simple and Expressive Model for Universal Composability. *Journal of Cryptology* 33, 4 (June 2020), 1461–1584. doi:10.1007/s00145-020-09352-1

[51] Gavin Lowe. 1995. An attack on the Needham-Schroeder public-key authentication protocol. *Inform. Process. Lett.* 56, 3 (Nov. 1995), 131–133. doi:10.1016/0020-0190(95)00144-2

[52] Gavin Lowe. 1996. *Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR*. Springer Berlin Heidelberg, Berlin, Heidelberg, 147–166. doi:10.1007/3-540-61042-1_43

[53] Moxie Marlinspike and Trevor Perrin. 2016. *The X3DH Key Agreement Protocol*. Technical Report. Signal Technology Foundation. https://signal.org/docs/specifications/x3dh/x3dh.pdf Last accessed March 28th, 2025.

[54] J. McLean. 1994. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of 1994 IEEE Computer Society Symposium on Research in Security and Privacy (RISP-94)*. IEEE Comput. Soc. Press, Oakland, CA, USA, 79–93. doi:10.1109/risp.1994.296590

[55] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. 1996. *Handbook of Applied Cryptography*. CRC Press, Boca Raton. doi:10.1201/9781439821916

[56] J. Mitchell, A. Scedrov, N.A. Durgin, and P.D. Lincoln. 1999. Undecidability of bounded security protocols. In *Workshop on formal methods and security protocols*. Citeseer. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=5efcab64d347abbcad0fd3527c9c8d43dfca449c

[57] Sebastian Mödersheim and Luca Viganò. 2009. *Secure Pseudonymous Channels*. Springer Berlin Heidelberg, Berlin, Heidelberg, 337–354. doi:10.1007/978-3-642-04444-1_21

[58] Sebastian Mödersheim and Luca Viganò. 2014. Sufficient conditions for vertical composition of security protocols. In *Proceedings of the 9th ACM symposium on Information, computer and communications security (ASIA CCS '14)*. ACM, New York, NY, USA, 435–446. doi:10.1145/2590296.2590330

[59] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. doi:10.17487/RFC8446

[60] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. 2025. *TLS Encrypted Client Hello*. Internet-Draft draft-ietf-tls-esni-25. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-tls-esni/25/ Work in Progress.

# A. Terms and All That

In this section, we collect notions and results about term rewriting from the literature that we need in our proofs. We will adapt the notation to fit the notation used in this work, and we will omit proofs, as they can be found easily in the respective sources. All signatures are assumed to be nontrivial.

First, some basic results as collected in [7].

**Lemma A.1.** *Let $t$ be a term. Then there is a uniquely determined non-empty context $C$ and there are uniquely determined alien subterms $s_1, \ldots, s_n$ of $t$ such that $t = C[[s_1, \ldots, s_n]]$.*

**Lemma A.2.** *Let $t$ be a term built over some signature $\Sigma(P)$ with equational theory $\mathrm{E}(P)$. Then $\downarrow s =_{\mathrm{E}(P)} s$.*

**Theorem A.3.** *Let $s$ and $t$ be terms built over the same signature $\Sigma(P) \cup \Sigma(Q)$ with equational theory $\mathrm{E}(P) \cup \mathrm{E}(Q)$. Then, for $R \in \{P, Q\}$:*

$$s =_{\mathrm{E}} t \Leftrightarrow \Downarrow_R(s) =_{\mathrm{E}} \Downarrow_R(t)$$

The following results are from [32].

**Lemma A.4.** *All alien subterms of a collapsed term are collapsed.*

**Lemma A.5.** *Let $t_1 = C[[s_1, \ldots, s_k]]$ and $t_2 = D[[s'_1, \ldots, s'_l]]$ be two collapsed terms. Let $K \in \{P, Q\}$ be such that $\mathrm{root}(C) \in \Sigma(K)$. If $\mathrm{root}(D) \notin \Sigma(K)$, then $t_1 \neq_{\mathrm{E}} t_2$. If $\mathrm{root}(D) \in \Sigma(K)$, then:*

$$t_1 =_{\mathrm{E}} t_2 \Leftrightarrow t_1\{^{s_j}/_{n_{[s_j]_{=_{\mathrm{E}}}}}\} =_{\mathrm{E}} t_2\{^{s_j}/_{n_{[s_j]_{=_{\mathrm{E}}}}}\}$$

*where $n_{[s_j]_{=_{\mathrm{E}}}}$ are names not appearing in $t_1$ or $t_2$.*

**Lemma A.6.** *Let $R \in \{P, Q\}$, and let $K \in \{P, Q\} \setminus \{R\}$.*

*If $s_1, \ldots, s_k, s'_1, \ldots, s'_l$ are collapsed terms with the root symbols from $\Sigma(R)$, and $C$, $D$ are both pure $\Sigma(K)$-contexts, we have that:*

$$C[s_1, \ldots, s_k] =_{\mathrm{E}} D[s'_1, \ldots, s'_l] \Leftrightarrow C[n_{[s_1]_{=_{\mathrm{E}}}}, \ldots, n_{[s_k]_{=_{\mathrm{E}}}}] =_{\mathrm{E}} D[n_{[s'_1]_{=_{\mathrm{E}}}}, \ldots, n_{[s'_l]_{=_{\mathrm{E}}}}]$$

**Lemma A.7.** *If $t_1$ and $t_2$ are collapsed terms such that $t_1 =_{\mathrm{E}} t_2$, then:*

$$\mathrm{root}(t_1) \in \Sigma(P) \Leftrightarrow \mathrm{root}(t_2) \in \Sigma(P)$$

**Lemma A.8.** *Let $\Sigma(P)$ and $\Sigma(Q)$ be signatures with equational theories $\mathrm{E}(P)$ and $\mathrm{E}(Q)$, and let $R \in \{P, Q\}$. Let $t_1$ and $t_2$ be collapsed terms built over $\Sigma(P) \cup \Sigma(Q)$. If $t_1 =_{\mathrm{E}} t_2$, then $\delta_R(t_1) =_{\mathrm{E}} \delta_R(t_2)$.*

# B. Differences to USENIX Security 2026 Version

- More detailed survey of related work
- Additional and expanded examples, explanations and intuition in Section 3, Section 4, and Section 5
- Proofs moved from the appendix to paper body
- Results and discussion of TLS 1.3 ECH case study moved from the appendix to paper body

# C. Changelog

v1.0 Consolidated version of the USENIX Security 2026 Short and Full Versions