Bridging the Gap from Trace Properties to Uniformity

Michael Backes[1,2] Esfandiar Mohammadi[1] Tim Ruffing[3]

[1] CISPA, Saarland University, Germany
[2] Max Planck Institute for Software Systems (MPI-SWS), Germany
[3] MMCI, Saarland University, Germany

backes@cs.uni-saarland.de
mohammadi@cs.uni-saarland.de
tim.ruffing@mmci.uni-saarland.de

November 27, 2014

Abstract

Dolev-Yao models of cryptographic operations constitute the foundation of many successful verification tools for security protocols, such as the protocol verifier ProVerif. Research over the past decade has shown that many of these symbolic abstractions are computationally sound, i.e., the absence of attacks against the abstraction entails the security of suitable cryptographic realizations. Most of these computational soundness (CS) results, however, are restricted to trace properties such as authentication. The few promising results that strive for CS for the more comprehensive class of equivalence properties, such as strong secrecy or anonymity, either only consider a limited class of protocols, or are not amenable to fully automated verification, or rely on abstractions for which it is not clear how to formalize any equivalence property beyond (strong) secrecy of payloads.

In this work, we identify a general condition under which CS for trace properties implies CS for uniformity of bi-processes, i.e., the class of equivalence properties that ProVerif is able to verify for the applied π-calculus. As a case study, we show that this general condition holds for a Dolev-Yao model that contains signatures, public-key encryption, and corresponding length functions. We prove this result in the CoSP framework (a general framework for establishing CS results). To this end, we extend the CoSP framework to equivalence properties, and we show that an existing embedding of the applied π-calculus to CoSP can be re-used for uniform bi-processes. On the verification side, as analyses in ProVerif with symbolic length functions often do not terminate, we show how to combine the recent protocol verifier APTE with ProVerif. As a result, we establish a computationally sound automated verification chain for uniformity of bi-processes in the applied π-calculus that use public-key encryption, signatures, and length functions.

Contents

1 Introduction **3**

2 Equivalence properties in the CoSP framework **4**
2.1 The CoSP framework . . . 4
2.2 Symbolic indistinguishability . . . 5
2.3 Computational indistinguishability . . . 7
2.4 Computational soundness . . . 10

3 Self-monitoring **10**
3.1 CS for trace properties . . . 10
3.2 Bridging the gap from trace properties to uniformity . . . 11

4 The applied π-calculus **13**
4.1 Embedding into CoSP . . . 14

5 Case study: encryption and signatures with lengths **15**
5.1 The symbolic model . . . 16
5.2 Implementation conditions . . . 18
5.3 CS for trace properties with length functions . . . 20
5.4 Distinguishing self-monitors for the symbolic model **M** . . . 21
5.5 The branching monitor . . . 21
5.5.1 Construction of the branching monitor . . . 22
5.5.2 Extended symbolic model . . . 25
5.5.3 Extended symbolic execution . . . 25
5.5.4 Computational self-monitoring of the branching monitor . . . 28
5.6 The knowledge monitor . . . 31
5.6.1 Construction of the knowledge monitor . . . 32
5.6.2 Symbolic self-monitoring of the knowledge monitor . . . 32
5.6.3 The faking simulator Sim_f . . . 32
5.6.4 CS for trace properties with length functions . . . 36
5.6.5 Decision variant of a protocol . . . 36
5.6.6 Uniqueness of a symbolic operation . . . 38
5.6.7 Unrolled variants . . . 39
5.6.8 Computational self-monitoring of the knowledge monitor . . . 40
5.7 **M** allows for self-monitoring . . . 47
5.8 CS for uniform bi-processes in the applied π-calculus . . . 47

6 Conclusion **47**

A Equivalence notions **48**

1 Introduction

Manual security analyses of protocols that rely on cryptographic operations are complex and error-prone. As a consequence, research has strived for the automation of such proofs soon after the first protocols were developed. To eliminate the inherent complexity of cryptographic operations that verification tools are struggling to deal with, cryptographic operations have been abstracted as symbolic terms that obey simple cancelation rules, so-called Dolev-Yao models [30, 31]. A variety of automated verification tools have been developed based on this abstraction, and they have been successfully used for reasoning about various security protocols [1, 4, 8, 14, 27, 28, 29, 34]. In particular, a wide range of these tools is capable of reasoning about the more comprehensive class of *equivalence properties*, such as strong secrecy and anonymity, which arguably is the most important class of security properties for privacy-preserving protocols.

Research over the past decade has shown that many of these Dolev-Yao models are computationally sound, i.e., the absence of attacks against the symbolic abstraction entails the security of suitable cryptographic realizations. Most of these computational soundness (CS) results against active attacks, however, have been specific to the class of trace properties [2, 3, 6, 12, 15, 24, 25, 26, 32, 33, 35], which is only sufficient as long as strong notions of privacy are not considered, e.g., in particular for establishing various authentication properties. Only few CS results are known for the class of equivalence properties against active attackers, which are restricted in of the following three ways: either they are restricted to a small class of simple processes, e.g., processes that do not contain private channels and abort if a conditional fails [16, 21, 22], or they rely on non-standard abstractions for which it is not clear how to formalize any equivalence property beyond the secrecy of payloads [5, 10, 11], such as anonymity properties in protocols that encrypt different signatures, or existing automated tool support is not applicable [23, 36]. We are thus facing a situation where CS results, despite tremendous progress in the last decade, still fall short in comprehensively addressing the class of equivalence properties and protocols that state-of-the-art verification tools are capable to deal with. Moreover, it is unknown to which extent existing results on CS for trace properties can be extended to achieve more comprehensive CS results for equivalence properties.

Our contribution. We close this gap by providing the first result that allows to leverage existing CS results for trace properties to CS results for an expressive class of equivalence properties: the uniformity of bi-processes in the applied π-calculus. Bi-processes are pairs of processes that differ only in the messages they operate on but not in their structure; a bi-process is uniform if for all surrounding contexts, i.e., all interacting attackers, both processes take the same branches. Blanchet, Abadi, and Fournet [14] have shown that uniformity already implies observational equivalence. Moreover, uniformity of bi-processes corresponds precisely to the class of properties that the state-of-the-art verification tool ProVerif [14] is capable to analyze, based on a Dolev-Yao model in the applied π-calculus. In contrast to previous work dealing with equivalence properties, we consider bi-protocols that use the fully fledged applied π-calculus, in particular including private channels and non-determinate processes.

To establish this main result of our paper, we first identify the following general condition for Dolev-Yao models: "whenever a computational attacker can distinguish a bi-process, there is a test in the Dolev-Yao model that allows to successfully distinguish the bi-process." We say that Dolev-Yao models with this property *allow for self-monitoring.* We show that if a specific Dolev-Yao model fulfills this property, then there is for every bi-process a so-called *self-monitor*, i.e., a process that performs all relevant tests that the attacker could perform on the two processes of the bi-process, and that raises an exception if of these tests in the symbolic model distinguishes the bi-process. We finally show that whenever a Dolev-Yao model allows for self-monitoring, CS for uniformity of bi-processes automatically holds whenever CS for trace properties has already been established. This result in particular allows for leveraging existing CS results for trace properties to more comprehensive CS results for uniformity of bi-processes, provided that the Dolev-Yao model can be proven to allow for self-monitoring.

We exemplarily show how to construct a self-monitor for a symbolic model that has been recently introduced and proven to be computationally sound for trace properties by Backes, Malik, and Unruh [9]. This symbolic model contains signatures and public-key encryption and allows to freely send and receive decryption keys. To establish that the model allows for self-monitoring, we first extend it using the common concept of a length function (without a length function, CS for uniformity of bi-processes and hence the existence of self-monitors trivially cannot hold, since encryptions of different lengths are distinguishable in general), and we show that this extension preserves the existing proof of CS for trace properties. Our main result in this paper then immediately implies that this extended model satisfies CS for uniformity of bi-processes.

We moreover investigate how computationally sound automated analyses can still be achieved in

those frequent situations in which ProVerif does not manage to terminate whenever the Dolev-Yao model supports a length function. We proceed in two steps: first, we feed a stripped-down version of the protocol without length functions in ProVerif; ProVerif then yields a result concerning the uniformity of bi-processes, but only for this stripped-down protocol. Second, we analyze the original protocol using the APTE tool by Cheval, Cortier, and Plet [20], which is specifically tailored to length functions. This yields a result for the original protocol but only concerning trace equivalences. We show that both results can be combined to achieve uniformity of bi-processes for the original protocol, and thus a corresponding CS result for uniformity of bi-processes.

We present the first general framework for CS for equivalence properties, by extending the CoSP framework: a general framework for symbolic analysis and CS results for trace properties [3]. CoSP decouples the CS of Dolev-Yao models from the calculi, such as the applied π-calculus or RCF: proving x cryptographic Dolev-Yao models sound for y calculi only requires $x + y$ proofs (instead of $x \cdot y$). We consider this extension to be of independent interest. Moreover, we prove the existence of an embedding from the applied π-calculus to the extended CoSP framework that preserves the uniformity of bi-processes, using a slight variation of the already existing embedding for trace properties.

2 Equivalence properties in the CoSP framework

2.1 The CoSP framework

The results in this work are formulated within CoSP [3], a framework for conceptually modular CS proofs that decouples the treatment of the cryptographic primitives from the treatment of the calculi. Several calculi such as the applied π-calculus [3] and RCF [7] (a core calculus of F#) can be embedded into CoSP and combined with CS results for cryptographic primitives.

The original CoSP framework is only capable of handling CS with respect to trace properties, i.e., properties that can be formulated in terms of a single trace. Typical examples include the non-reachability of a certain "bad" protocol state, in that the attacker is assumed to have succeeded (e.g., the protocol never reveals a secret), or correspondence properties such as authentication (e.g., a user can access a resource only after proving a credential). However, many interesting protocol properties cannot be expressed in terms of a single trace. For instance, strong secrecy or anonymity are properties that are, in the computational setting, usually formulated by means of a game in which the attacker has to distinguish between several scenarios.

To be able to handle the class of equivalence properties, we extend the CoSP framework to support equivalence properties. First, we recall the basic definitions of the original framework. Dolev-Yao models are formalized as follows in CoSP.

Definition 1 (Symbolic model) A *symbolic model* $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ consists of a set of constructors $\mathbf{C}$, a set of nonces $\mathbf{N}$, a message type $\mathbf{T}$ over $\mathbf{C}$ and $\mathbf{N}$ (with $\mathbf{N} \subseteq \mathbf{T}$), a set of destructors $\mathbf{D}$ over $\mathbf{T}$. We require that $\mathbf{N} = \mathbf{N_E} \uplus \mathbf{N_P}$ for countable infinite sets $\mathbf{N_P}$ of protocol nonces and attacker nonces $\mathbf{N_E}$. $\diamond$

We write $\underline{t}$ for a list $t_1, \ldots, t_n$ if n is clear from the context. A *constructor* C/n is a symbol with (possibly zero) arity. A *nonce* N is a symbol with zero arity. A *message type* $\mathbf{T}$ *over* $\mathbf{C}$ *and* $\mathbf{N}$ is a set of terms over constructors $\mathbf{C}$ and nonces $\mathbf{N}$. A *destructor* D/n of arity n, over a message type $\mathbf{T}$ is a partial map $\mathbf{T}^n \to \mathbf{T}$. If D is undefined on $\underline{t}$, we write $D(\underline{t}) = \bot$.

To unify notation, we define for every constructor or destructor $F/n \in \mathbf{D} \cup \mathbf{C}$ and every nonce $F \in \mathbf{N}$ the partial function $eval_F : \mathbf{T}^n \to \mathbf{T}$, where $n = 0$ for a nonce, as follows.

Definition 2 (Evaluation of terms) If F is a constructor, $eval_F(\underline{t}) := F(\underline{t})$ if $F(\underline{t}) \in \mathbf{T}$ and $eval_F(\underline{t}) := \bot$ otherwise. If F is a nonce, $eval_F() := F$. If F is a destructor, $eval_F(\underline{t}) := F(\underline{t})$ if $F(\underline{t}) \neq \bot$ and $eval_F(\underline{t}) := \bot$ otherwise. $\diamond$

Protocols. In CoSP, a protocol is represented as a tree. Each node in this tree corresponds to an action in the protocol: *computation nodes* are used for drawing fresh nonces, applying constructors, and applying destructors; *input* and *output nodes* are used to send and receive messages; *control nodes* are used for allowing the attacker to schedule the protocol.

Definition 3 (CoSP protocol) A *CoSP protocol* Π is a tree of infinite depth with a distinguished root and labels on both edges and nodes. Each node has a unique identifier ν and one of the following

types:[1]

- *Computation nodes* are annotated with a constructor, nonce or destructor F/n together with the identifiers of n (not necessarily distinct) nodes; we call these annotations *references*, and we call the referenced nodes *arguments*. Computation nodes have exactly two successors; the corresponding edges are labeled with `yes` and `no`, respectively.
- *Input nodes* have no annotations. They have exactly one successor.
- *Output nodes* have a reference to exactly one node in their annotations. They have exactly one successor.
- *Control nodes* are annotated with a bitstring l. They have at least one and up to countably many successors; the corresponding edges are labeled with distinct bitstrings l'. (We call l the *out-metadata* and l' the *in-metadata*.)

We assume that the annotations are part of the node identifier. A node ν can only reference other nodes ν' on the path from the root to ν; in this case ν' must be a computation node or input node. If ν' is a computation node, the path from ν' to ν has additionally to go through the outgoing edge of ν' with label `yes`. ◇

Bi-protocols. To compare two variants of a protocol, we consider bi-protocols, which rely on the same idea as bi-processes in the applied π-calculus [13]. Bi-protocols are pairs of protocols that only differ in the messages they operate on.

Definition 4 (CoSP bi-protocol) A *CoSP bi-protocol* Π is defined like a protocol but uses bi-references instead of references. A *bi-reference* is a pair $(\nu_{\mathsf{left}}, \nu_{\mathsf{right}})$ of node identifiers of two (not necessarily distinct) nodes in the protocol tree. In the *left protocol* $\mathsf{left}(\Pi)$ the bi-references are replaced by their left components; the *right protocol* $\mathsf{right}(\Pi)$ is defined analogously. ◇

2.2 Symbolic indistinguishability

In this section, we define a symbolic notion of indistinguishability. First, we model the capabilities of the symbolic attacker. Operations that the symbolic attacker can perform on terms are defined as follows, including the destruction of already known terms and the creation of new terms.[2]

Definition 5 (Symbolic operation) Let $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ be a symbolic model. A *symbolic operation O/n (of arity n) on* $\mathbf{M}$ is a finite tree whose nodes are labeled with constructors from $\mathbf{C}$, destructors from $\mathbf{D}$, nonces from $\mathbf{N}$, and formal parameters x_i with $i \in \{1, \dots, n\}$. For constructors and destructors, the children of a node represent its arguments (if any). Formal parameters x_i and nonces do not have children.

We extend the evaluation function to symbolic operations. Given a list of terms $\underline{t} \in \mathbf{T}^n$, the evaluation function $eval_O : \mathbf{T}^n \to \mathbf{T}$ recursively evaluates the tree O starting at the root as follows: The formal parameter x_i evaluates to t_i. A node with $F \in \mathbf{C} \cup \mathbf{N}_E \cup \mathbf{D}$ evaluates according to $eval_F$. If there is a node that evaluates to $\perp$, the whole tree evaluates to $\perp$. ◇

Note that the identity function is included. It is the tree that contains only x_1 as node.

A symbolic execution of a protocol is basically a valid path through the protocol tree. It induces a *view*, which contains the communication with the attacker.

Definition 6 (Symbolic execution) Let a symbolic model $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ and a CoSP protocol Π be given. A *symbolic view* of the protocol Π is a (finite) list of triples (V_i, ν_i, f_i) with the following conditions:
For the first triple, we have $V_1 = \varepsilon$, ν_1 is the root of Π, and f_1 is an empty partial function, mapping node identifiers to terms. For every two consecutive tuples (V, ν, f) and (V', ν', f') in the list, let $\underline{\tilde{\nu}}$ be the nodes referenced by ν and define $\underline{\tilde{t}}$ through $\tilde{t}_j := f(\tilde{\nu}_j)$. We conduct a case distinction on ν.

[1] In contrast to the definition in the original CoSP framework [3], we do not allow the type "non-deterministic node". This modification is not severe, because all embeddings of symbolic calculi that have been established in the original framework so far are compatible with our definition.

[2] We deviate from the definition in the original CoSP framework [3], where a deduction relation describes which terms the attacker can deduce from the already seen terms. This modification is not essential; all results for trace properties that have been established in the original framework so far are compatible with our definition.

- ν **is a computation node with constructor, destructor or nonce** F. Let $V' = V$. If $m := eval_F(\underline{\tilde{t}}) \neq \bot$, ν' is the yes-successor of ν in Π, and $f' = f(\nu := m)$. If $m = \bot$, then ν' is the no-successor of ν, and $f' = f$.

- ν **is an input node.** If there exists a term $t \in \mathbf{T}$ and a symbolic operation O on $\mathbf{M}$ with $eval_O(V_{Out}) = t$, let ν' be the successor of ν in Π, $V' = V :: (\texttt{in}, (t, O))$, and $f' = f(\nu := t)$.

- ν **is an output node.** Let $V' = V :: (\texttt{out}, \tilde{t}_1)$, ν' is the successor of ν in Π, and $f' = f$.

- ν **is a control node with out-metadata** l. Let ν' be the successor of ν with the in-metadata l' (or the lexicographically smallest edge if there is no edge with label l'), $f' = f$, and $V' = V :: (\texttt{control}, (l, l'))$.

Here, V_{Out} denotes the list of terms in V that have been sent at output nodes, i.e., the terms t contained in entries of the form $(\texttt{out}, t)$ in V. Analogously, $V_{Out\text{-}Meta}$ denotes the list of out-metadata in V that has been sent at control nodes.

The set of all symbolic views of Π is denoted by $\mathsf{SViews}(\Pi)$. Furthermore, V_{In} denotes the partial list of V that contains only entries of the form $(\texttt{in}, (*, O))$ or $(\texttt{control}, (*, l'))$ for some symbolic operation O and some in-metadata l', where the input term and the out-metadata have been masked with the symbol $*$. The list V_{In} is called *attacker strategy.* We write $[V_{In}]_{\mathsf{SViews}(\Pi)}$ to denote the class of all views $U \in \mathsf{SViews}(\Pi)$ with $U_{In} = V_{In}$. ◇

The knowledge of the attacker comprises the results of all the symbolic tests the attacker can perform on the messages output by the protocol. To define the attacker knowledge formally, we have to pay attention to two important details. First, we concentrate on whether a symbolic operation fails or not, i.e., if it evaluates to $\bot$ or not; we are not interested in the resulting term in case the operation succeeds. The following example illustrates why: suppose the left protocol of a bi-protocol does nothing more than sending a ciphertext c to the attacker, whereas the right protocol sends a different ciphertext c' (with the same plaintext length) to the attacker. Assume that the decryption key is kept secret. This bi-protocol should be symbolically indistinguishable. More precisely, the attacker knowledge in the left protocol should be statically indistinguishable from the attacker knowledge in the right protocol. Recall that $O = x_1$ is the symbolic operation that just returns the first message received by the attacker. If the result of O were part of the attacker knowledge, the knowledge in the left protocol (containing c) would differ from the knowledge in the right protocol (containing c'), which is not what we would like to express. On the other hand, our definition, which only cares about the failure or success of a operation, requires that the symbolic model contains an operation *equals* to be reasonable. This operation *equals* allows the attacker to test equality between terms: consider the case where the right protocol sends a publicly known term t instead of c', but still of the same length as c. In that case the attacker can distinguish the bi-protocol with the help of the symbolic operation $equals(t, x_1)$.

The second observation is that the definition should cover the fact that the attacker knows which symbolic operation leads to which result. This is essential to reason about indistinguishability: consider a bi-protocol such that the left protocol sends the pair (n, t), but the right protocol sends the pair (t, n), where t is again a publicly known term and n is a fresh protocol nonce. The two protocols do not differ in the terms that the attacker can deduce after their execution; the deducible terms are all publicly known terms as well as n. Still, the protocols are trivially distinguishable by the symbolic operation $equals(O_t, snd(x_1))$ because $equals(O_t, snd((n, t))) \neq \bot$ but $equals(t, snd((t, n))) = \bot$, where snd returns the second component of a pair and O_t is a symbolic operation that constructs t.

Definition 7 (Symbolic knowledge) Let $\mathbf{M}$ be a symbolic model. Given a view V with $|V_{Out}| = n$, a *symbolic knowledge function* $f_V : \mathsf{SO}(\mathbf{M})_n \to \{\top, \bot\}$ is a partial function from symbolic operations (see Definition 5) of arity n to $\{\top, \bot\}$. The *full symbolic knowledge function* is a total symbolic knowledge function is defined by

$$K_V(O) := \begin{cases} \bot & \text{if } eval_O(V_{Out}) = \bot \\ \top & \text{otherwise.} \end{cases}$$

◇

Intuitively, we would like to consider two views *equivalent* if they look the same for a symbolic attacker. Despite the requirement that they have the same order of output, input and control nodes, this is the case if they agree on the out-metadata (the control data sent by the protocol) as well as the symbolic knowledge that can be gained out of the terms sent by the protocol.

Definition 8 (Equivalent views) Let two views V, V' of the same length be given. We denote their ith entry by V_i and V_i', respectively. V and V' are *equivalent* ($V \sim V'$), if the following three conditions hold:

1. (Same structure) V_i is of the form $(s, \cdot)$ if and only if V_i' is of the form $(s, \cdot)$ for some $s \in \{\mathtt{out}, \mathtt{in}, \mathtt{control}\}$.
2. (Same out-metadata) $V_{\mathit{Out\text{-}Meta}} = V'_{\mathit{Out\text{-}Meta}}$.
3. (Same symbolic knowledge) $K_V = K_{V'}$. ◇

Finally, we define two protocols (e.g., the left and right variant of a bi-protocol) to be symbolically indistinguishable if its two variants lead to equivalent views when faced with the same attacker strategy.[3] Thus, a definition of "symbolic indistinguishability" should compare the symbolic knowledge of two protocol runs only if the attacker behaves identically in both runs.

Definition 9 (Symbolic indistinguishability) Let $\mathbf{M}$ be a symbolic model and P be a class of protocols on $\mathbf{M}$. Given an attacker strategy V_{In} (in the sense of Definition 28), two protocols $\Pi_1, \Pi_2 \in \mathrm{P}$ are *symbolically indistinguishable under* V_{In} if for all views $V_1 \in [V_{In}]_{\mathsf{SViews}(\Pi_1)}$ of Π_1 under V_{In}, there is a view $V_2 \in [V_{In}]_{\mathsf{SViews}(\Pi_2)}$ of Π_2 under V_{In} such that $V_1 \sim V_2$, and vice versa.

Two protocols $\Pi_1, \Pi_2 \in \mathrm{P}$ are *symbolically indistinguishable* ($\Pi_1 \approx_s \Pi_2$), if Π_1 and Π_2 are indistinguishable under all attacker strategies. For a bi-protocol Π, we say that Π is symbolically indistinguishable if $\mathsf{left}(\Pi) \approx_s \mathsf{right}(\Pi)$. ◇

2.3 Computational indistinguishability

On the computational side, the constructors and destructors in a symbolic model are realized with cryptographic algorithms, which we call computational implementations.

Definition 10 (Computational implementation) Let $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ be a symbolic model. A *computational implementation of* $\mathbf{M}$ is a family of functions $\mathbf{A} = (A_x)_{x \in \mathbf{C} \cup \mathbf{D} \cup \mathbf{N}}$ such that A_F for $F/n \in \mathbf{C} \cup \mathbf{D}$ is a partial deterministic function $\mathbb{N} \times (\{0,1\}^*)^n \to \{0,1\}^*$, and A_N for $N \in \mathbf{N}$ is a total probabilistic function with domain $\mathbb{N}$ and range $\{0,1\}^*$. The first argument of A_F and A_N represents the security parameter.

All functions A_F have to be computable in deterministic polynomial time, and all A_N have to be computable in probabilistic polynomial time (ppt). ◇

The computational execution of a protocol is a randomized interactive machine that runs against a ppt attacker $\mathcal{A}$. The transcript of the execution contains essentially the computational counterparts of a symbolic view.

Definition 11 (Computational challenger) Let $\mathbf{A}$ be a computational implementation of the symbolic model $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ and Π be a CoSP protocol. Let $\mathcal{A}$ be a ppt machine. For a security parameter k, the *computational challenger* $\mathsf{Exec}_{\mathbf{M},\mathit{Imp},\Pi}(k)$ is the following interactive machine:

Initially, let ν be the root of Π. Let f and n be empty partial functions from node identifiers to bitstrings and from $\mathbf{N}$ to bitstrings, respectively. Enter a loop and proceed depending on the type of ν:

- **ν is a computation node with nonce $N \in \mathbf{N}$.** If $n(N) \neq \bot$, let $m' := n(N)$; otherwise sample m' according to $A_N(k)$. Let ν' be the yes-successor of ν. Let $f := f(\nu := m')$, $n := n(N := m')$, and $\nu_i := \nu'$.
- **ν is a computation node with constructor or destructor F.** Let $\underline{\tilde{\nu}}$ be the nodes referenced by ν and $\tilde{m}_j := f(\tilde{\nu}_j)$. Then, $m' := A_F(k, \underline{\tilde{m}})$. If $m' \neq \bot$, then ν' is the $\mathtt{yes}$-successor of ν, if $m' = \bot$, then ν' is the $\mathtt{no}$-successor of ν. Let $f := f(\nu := m')$ and $\nu_i := \nu'$.
- **ν is an input node.** Ask the adversary $\mathcal{A}$ for a bitstring m. Let ν' be the successor of ν. Let $f := f(\nu := m)$ and $\nu_i := \nu'$.
- **ν is an output node.** Send $\tilde{m}_1$ to $\mathcal{A}$. Let ν' be the successor of ν, and let $\nu_i := \nu'$.

[3] For the sake of convenience, we define CS for bi-protocols. However, our definition can be easily generalized to arbitrary pairs of protocols.

- **ν is a control node with out-metadata l.** Send l to $\mathcal{A}$. Upon receiving in-metadata l', let ν' be the successor of ν along the edge labeled l' (or the edge with the lexicographically smallest label if there is no edge with label l'). Let $\nu_i := \nu'$. $\diamond$

Definition 12 (Computational execution) The interaction between the challenger $\mathsf{Exec}_{\mathbf{M},\mathbf{A},\Pi}(k)$ and the adversary $\mathcal{A}(k)$ is called the *computational execution*, denoted by $\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\Pi}(k) \mid \mathcal{A}(k)\rangle$. It stops whenever one of the two machines stops, and the output of $\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\Pi}(k) \mid \mathcal{A}(k)\rangle$ is the output of $\mathcal{A}(k)$. $\diamond$

Definition 13 (Efficient protocol) We call a CoSP (bi-)protocol *efficient* if:

- There is a polynomial p such that for any node N, the length of the identifier of N is bounded by $p(m)$ where m is the length (including the total length of the edge-labels) of the path from the root to N.
- There is a deterministic polynomial-time algorithm that, given the identifiers of all nodes and the edge labels on the path to a node N, computes the identifier of N.
- There is a deterministic polynomial-time algorithm that, given the identifier of a control node N, the identifiers of all nodes and all edge labels on the path to N, computes the lexicographically smallest label of an edge (i.e., the in-metadata) of all edges that lead from N to one of its successors. $\diamond$

A note on the polynomial-time computability of the computational execution. For efficient protocols, a CoSP challenger can be computed by a polynomially time-bounded machine. The deterministic polynomial-time algorithm from Definition 13 is a compact description of the infinite CoSP protocol. Since the identifier is poly-time computable from the path to the root and the data sent by the attacker, the out-metadata is part of the identifier, all algorithms are poly-time, and the lexicographically smallest edge can be found in poly-time, all messages that the CoSP challenger sends to the attacker can be computed in poly-time.

We rely on the notion of termination-insensitive computational indistinguishability (tic-indistinguishability) [38] to capture that two protocols are indistinguishable in the computational world. In comparison to standard computational indistinguishability, tic-indistinguishability on one hand does not require the protocols (formally, the interactive machines) to be polynomial-time, but on the other hand it solely considers decisions that were made after a polynomially-bounded prefix of the interaction (where, both, the adversary's and the protocol's steps are counted). If after an activation, (say) the second protocol does not output anything within a polynomial numbers of steps, then the pair of protocols will be considered indistinguishable no matter how the first protocol will behave in this case.

To understand why tic-indistinguishability is a useful tool, consider a pair of protocols that is symbolically equivalent and constitutes a bi-protocol from a uniformity enforcing class of bi-protocols. Intuitively, a meaningful CS result should state that the bi-protocol is computationally indistinguishable. However, it might be the case that after receiving input, the first protocol Π_1 produces an output immediately but the second protocol Π_2 enters a loop that would take a superpolynomial number of steps. (Indeed, such a difference does not contradict the symbolic equivalence because the symbolic world abstracts away from issues such as running time.) However, a polynomial-time distinguisher can obverse that the second protocol Π_2 halts prematurely. That is, the protocols are computationally distinguishable, even if we abstract from polynomial differences in the running time by allowing the protocols to run with different running-time polynomials (the polynomials that determine the bound at which the computational challenger (Definition 11) halts the corresponding protocol). Thus, we will not be able to prove normal computational indistinguishability. Tic-indistinguishability however holds and allows us to establish a CS result.

A second problem arises, because many real protocols (e.g. reactive systems such as file) actually have no a priori-bound for their running time. Unruh [38] gives the following example (we adopt it slightly to our setting): Consider a bi-protocol Π that describes a file server. The file server allows clients to store values to files, read (parts of) files, store the concatenation of two files as a new file, and store the encryption of a files as a new file. Assume that the file server uses an IND-CCA secure public encryption scheme and has only access to the public key; the secret key is never used. Further assume that in $\mathsf{left}(\Pi)$, the encryption uses the actual contents of the file of length n, and in $\mathsf{right}(\Pi)$, the encryption instead uses 0^n. Now consider the following polynomial-time distinguisher $\mathcal{A}$: First, $\mathcal{A}$ creates a file f_0 with content 1. Second, $\mathcal{A}$ creates k files $f_1, \dots, f_k$ by concatenation operations such that $f_i = 1^{2^i}$, where k is the

security parameter. Third, $\mathcal{A}$ creates a file f by letting the file server encrypt the file f_k. Finally, $\mathcal{A}$ reads the first bit of f. Since IND-CCA security does not consider input messages of superpolynomial size (e.g., 1^{2^k}), nothing is guaranteed about the resulting ciphertext f. Indeed, the first bit of f could the first bit of f_k, in which case $\mathcal{A}$ can easily distinguish $\mathsf{left}(\Pi)$ from $\mathsf{right}(\Pi)$.

The essence of this problem is that protocols that could (potentially) run in superpolynomial time prevent us of from using computational assumptions. However, the point of using computational assumptions in cryptography is actually to abstract away from problems that occur after a superpolynomial running time. While we cannot use normal computational indistinguishability, tic-indistinguishability allows us to establish a meaningful CS result, because it guarantees indistinguishability as long as the protocol does not long actually run in a superpolynomial time, while it ignores the situation after a superpolynomial number of steps.

Note that both these problems could be solved also by restricting the protocol class to bi-protocols that always run in polynomial time. In that case, we could use normal computational indistinguishability. However, this way comes with drastic restrictions: First, it excludes many plausible protocols like the protocol described above. Second, to enable mechanized verification, it is highly desirable to define the class of bi-protocols captured by the CS result using statically-verifiable conditions, e.g., syntactic conditions such as disallowing loops. (This way is taken by Comon-Lundh and Cortier [21] as well as Comon-Lundh, Cortier and Scerri [22].) This however comes with even more restrictions, because such syntactic restrictions are typically only a rough over-approximation of polynomial-time protocols. Thus, even more protocols are excluded.

Definition 14 (Tic-indistinguishability) Given two machines M, M' and a polynomial p, we write $\Pr[\langle M \mid M' \rangle \Downarrow_{p(k)} x]$ for the probability that the interaction between M and M' terminates within $p(k)$ steps and M' outputs x. We call two machines A and B *termination-insensitively computationally indistinguishable for a machine $\mathcal{A}$ ($A \approx_{tic}^{\mathcal{A}} B$)* if for for all polynomials p, there is a negligible function μ such that for all $z, a, b \in \{0,1\}^*$ with $a \neq b$,

$$\Pr[\langle A(k) \mid \mathcal{A}(k,z) \rangle \Downarrow_{p(k)} a] + \Pr[\langle B(k) \mid \mathcal{A}(k,z) \rangle \Downarrow_{p(k)} b] \leq 1 + \mu(k).$$

Here, z represents an auxiliary string. Additionally, we call A and B *termination-insensitively computationally indistinguishable ($A \approx_{tic} B$)* if we have $A \approx_{tic}^{\mathcal{A}} B$ for all polynomial-time machines $\mathcal{A}$. ⋄

Termination insensitive computational indistinguishability is not transitive, i.e., $A \approx_{tic} B$ and $B \approx_{tic} C$ does not imply $A \approx_{tic} C$ in general (see [38] for more details). Due to this limitation, we additionally use the standard notion of computational indistinguishability, denoted as $A \approx_{comp} B$. Moreover, we also use a notion of perfect indistinguishability with the additional requirement that both computations have a running time that at most polynomially differs. This notion is denoted as $A \approx_{time} B$ (see [38] for more details).

We refer the reader to Unruh [38] for a further discussion of tic-indistinguishability and a precise technical definition of the underlying machine model.

Notation: interfaces for interactions between machines. We use the notion of *interfaces* to connect a network of machines. An interface is like a channel through which two machines communicate. We assume that all machines in this work have a network interface, denoted as net. As a notational convention, we assume that for three machines A, B, C where B has a left and a right network interface (often call the execution network interface and the network interface, respectively) $\langle A \mid \langle B \mid C \rangle \rangle$ denotes that the network interface of A is connected to the left network interface (i.e., the execution network interface) of B and the right network interface (i.e., the network interface) of B is connected to the network interface of C. Moreover, we assume that the final output of a machine is sent over an output interface. Typically, $\langle A \mid \langle B \mid C \rangle \rangle$ denotes that B has a sub-output interface that is connected to the output interface of C, and for the interaction $\langle A \mid \langle B \mid C \rangle \rangle$ the output is the message that is sent over the output interface (of B). Similarly, in the case where we only have two machines A, B the output of $\langle A \mid B \rangle$ is typically the message sent over the output interface of B.

Given this definition, computational indistinguishability for bi-protocols is naturally defined. A bi-protocol is indistinguishable if its challengers are computationally indistinguishable for every ppt attacker $\mathcal{A}$.

Definition 15 (Computational indistinguishability) Let Π be an efficient CoSP bi-protocol and let $\mathbf{A}$ be a computational implementation of $\mathbf{M}$. Π is *(termination-insensitively) computationally indistinguishable* if for all ppt attackers $\mathcal{A}$ and for all polynomials p,

$$\mathsf{Exec}_{\mathbf{A},\mathbf{M},\mathsf{left}(\Pi)} \approx_{tic} \mathsf{Exec}_{\mathbf{A},\mathbf{M},\mathsf{right}(\Pi)}. \qquad \diamond$$

2.4 Computational soundness

Having defined symbolic and computational indistinguishability, we are finally able to relate them. The previous definitions culminate in the definition of CS for equivalence properties. It states that the symbolic indistinguishability of a bi-protocol implies its computational indistinguishability. In other words, it suffices to check the security of the symbolic bi-protocol, e.g., using mechanized protocol verifiers such as ProVerif.

Definition 16 (Computational soundness for equivalence properties) Let a symbolic model $\mathbf{M}$ and a class P of efficient bi-protocols be given. An implementation $\mathbf{A}$ of $\mathbf{M}$ is *computationally sound* for $\mathbf{M}$ if for every $\Pi \in$ P, we have that Π is computationally indistinguishable whenever Π is symbolically indistinguishable. $\diamond$

Definition 17 (Symbolic execution of a parameterized CoSP protocol) Let Π be a parameterized CoSP protocol and f_{par} be a total function $par(\Pi^{\mathsf{s}}) \to \mathbf{N}$. A *full trace of* Π^{s} *with parameters* f_{par} is defined like a full trace of Π^{s} (according to the standard symbolic CoSP execution) with the modification that parameter identifiers are treated like node identifiers and f_1 in the first list entry is f_{par} instead of the empty function. $\diamond$

Definition 18 (Computational execution of a parameterized CoSP protocol) Let Π be a parameterized CoSP protocol and f_{par} be a function $par(\Pi^{\mathsf{p}}) \to \{0,1\}^*$. The computational execution of Π^{p} with parameters f_{par} is the standard computational CoSP execution with the modification that parameter identifiers are treated like node identifiers and in the initial state f is set to f_{par} instead of the empty function. $\diamond$

3 Self-monitoring

In this section, we identify a sufficient condition for symbolic models under which CS for trace properties implies CS for equivalence properties for a class of *uniformity-enforcing protocols*, which correspond to uniform bi-processes in the applied π-calculus. We say that a symbolic model that satisfies this condition *allows for self-monitoring*. The main idea behind self-monitoring is that a symbolic model is sufficiently expressive (and its implementation is sufficiently strong) such that the following holds: whenever a computational attacker can distinguish a bi-process, there is a test in the symbolic model that allows to successfully distinguish the bi-process.

3.1 CS for trace properties

A trace property that captures that a certain bad state (a certain class of bad node identifiers) is not reachable, would be formalized as the set of all traces that do not contain bad nodes.

Formally, a trace property is a prefix-closed set of node-traces. We say that a protocol Π *symbolically satisfies* a trace property $\mathcal{P}$ if for all traces t resulting from a symbolic execution, $\mathcal{P}$ holds for t, i.e., $t \in \mathcal{P}$. Correspondingly, we state that a protocol Π *computationally satisfies* $\mathcal{P}$ if for any ppt attacker $\mathcal{A}$ and polynomial p the probability is overwhelming that $\mathcal{P}$ holds for t, i.e., $t \in \mathcal{P}$, for any trace t resulting from a computational execution with $\mathcal{A}$ and p.

We first review the relevant definitions from the original CoSP framework. Instead of a view that contains the communication with the attacker, we are interested in a trace of the execution of a protocol.

Definition 19 (Symbolic and computational traces) Let (V_i, ν_i, f_i) be a (finite) list of triples produced internally by the symbolic execution (Definition 28) of a CoSP protocol Π. Then a list ν_i is a *symbolic trace* of Π.

Analogously, given a polynomial p, the probability distribution on the list ν_i computed by the computational execution (Definition 12) of Π with polynomial p is called *computational trace* of Π. $\diamond$

CS for trace properties states that all attacks (against trace properties) that can be excluded for the symbolic abstraction can be excluded for the computational implementation as well. Hence, if all the symbolic traces satisfy a certain trace property, then all computational traces satisfy this property as well.

Definition 20 (Computational soundness for trace properties [3]) A symbolic model $(\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ is *computationally sound for trace properties* with respect to an implementation $\mathbf{A}$ for a class

P of efficient protocols if the following holds: for each protocol $I \in \mathrm{P}$ and each trace property $\mathcal{P}$, I computationally satisfies $\mathcal{P}$ whenever I symbolically satisfies $\mathcal{P}$. ◇

Uniformity-enforcing. For the connection to trace properties, we consider only *uniform* bi-protocols. A bi-protocol is uniform if for each symbolic attacker strategy, both its variants reach the same nodes in the CoSP tree, i.e., they never branch differently.[4] Formally, we require that the bi-protocols are *uniformity-enforcing*, i.e., when the left and the right protocol of the bi-protocol Π take different branches, the attacker is informed. Since taking different branches is only visible after a control node is reached, we additionally require that computation nodes are immediately followed by control nodes.

Definition 21 (Uniformity-enforcing) A class P of CoSP bi-protocols is *uniformity-enforcing* if for all bi-protocols $\Pi \in \mathrm{P}$:

1. Every control node in Π has unique out-metadata.
2. For every computation node ν in Π and for every path rooted at ν, a control node is reached before an output node. ◇

All embeddings of calculi the CoSP framework described so far, namely those of the applied π-calculus [3] and RCF [7], are formalized such that protocols written in these calculi fulfill the second property: these embeddings give the attacker a scheduling decision, using a control node, basically after every execution step. We stress that it is straightforward to extend them to fulfill the first property by tagging the out-metadata with the address of the node in the protocol tree.

3.2 Bridging the gap from trace properties to uniformity

The key observation for the connection to trace properties is that, given a bi-protocol Π, some computationally sound symbolic models allow to construct a self-monitor protocol $\mathsf{Mon}(\Pi)$ (not a bi-protocol!) that has essentially the same interface to the attacker as the bi-protocol Π and checks at run-time whether Π would behave uniformly. In other words, non-uniformity of bi-protocols can be formulated as a trace property `bad`, which can be detected by the protocol $\mathsf{Mon}(\Pi)$.

The self-monitor $\mathsf{Mon}(\Pi)$ of a bi-protocol Π behaves like one of the two variants of the bi-protocol Π, while additionally simulating the opposite variant such that $\mathsf{Mon}(\Pi)$ itself is able to detect whether Π would be distinguishable. (For instance, one approach to detect whether Π is distinguishable could consist of reconstructing the symbolic view of the attacker in the variant of Π that is not executed by $\mathsf{Mon}(\Pi)$.) At the beginning of the execution of the self-monitor, the attacker chooses if $\mathsf{Mon}(\Pi)$ should basically behave like $\mathsf{left}(\Pi)$ or like $\mathsf{right}(\Pi)$. We denote the chosen variant as $b \in \{\mathsf{left}, \mathsf{right}\}$ and the opposite variant as $\bar{b}$. After this decision, $\mathsf{Mon}(\Pi)$ executes the the b-variant $b(\Pi)$ of the bi-protocol Π, however, enriched with the computation nodes and the corresponding output nodes of the opposite variant $\bar{b}(\Pi)$.[5]

The goal of the self-monitor $\mathsf{Mon}(\Pi)$ is to detect whether the execution of $b(\Pi)$ would be distinguishable from $\bar{b}(\Pi)$ at the current state. If this is the case, $\mathsf{Mon}(\Pi)$ raises the event `bad`, which is the disjunction of two events `bad-branch` and `bad-knowledge`.

The event `bad-branch` corresponds to the case that the left and the right protocol of the bi-protocol Π take different branches. Since uniformity-enforcing protocols have a control node immediately after every computation node (see Definition 21), the attacker can always check whether $b(\Pi)$ and $\bar{b}(\Pi)$ take the same branch. We require (in Definition 23) the existence of a so-called *distinguishing self-monitor* $f_{\texttt{bad-branch},\Pi}$ that checks whether each destructor application in $b(\Pi)$ succeeds if and only if it succeeds in $\bar{b}(\Pi)$; if not, the distinguishing self-monitor $f_{\texttt{bad-branch},\Pi}$ raises `bad-branch`.

The event `bad-knowledge` captures that the messages sent by $b(\Pi)$ and $\bar{b}(\Pi)$ (via output nodes, i.e., not the out-metadata) are distinguishable. This distinguishability is only detectable by a protocol if the constructors and destructors, which are available to both the protocol and the symbolic attacker, capture all possible tests. We require (in Definition 23) the existence of a distinguishing self-monitor

[4] We show in Lemma 2 that uniformity of bi-protocols in CoSP corresponds to uniformity of bi-processes in the applied π-calculus.

[5] This leads to the fact that whenever there is an output node in Π, there are two corresponding output nodes in $\mathsf{Mon}(\Pi)$, which contradicts the goal that the interface of Π and $\mathsf{Mon}(\Pi)$ should be the same towards the attacker. However, this technicality can be dealt with easily when applying our method. For example, in the computational proof for our case study, we use the self-monitor in an interaction with a filter machine that hides the results of the output nodes of $\bar{b}(\Pi)$ to create a good simulation towards the computational attacker, whose goal is to distinguish Π. The filter machine is then used as a computational attacker against $\mathsf{Mon}(\Pi)$.

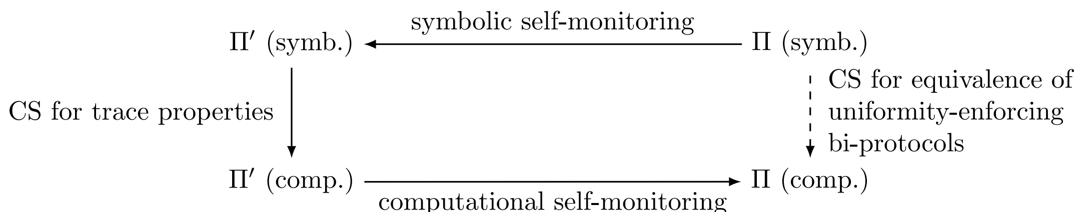

Figure 1: Symbolic and computational self-monitoring.

$f_{\texttt{bad-knowledge},\Pi}$ that raises `bad-knowledge` in $\mathsf{Mon}(\Pi)$ whenever a message, sent in Π, would allow the attacker to distinguish $b(\Pi)$ and $\bar{b}(\Pi)$.

Parameterized CoSP protocols. For a bi-protocol Π, we formalize the distinguishing self-monitors $f_{\texttt{bad-knowledge},\Pi}$ and $f_{\texttt{bad-branch},\Pi}$ with the help of *parameterized CoSP protocols*, which have the following properties: Nodes in such protocols are not required to have successors and instead of other nodes, also formal parameters can be referenced. A parameterized CoSP protocol is intended to be plugged into another protocol; in that case the parameters references must be changed to references to nodes.

Definition 22 (Self-monitor) Let Π be a CoSP bi-protocol. Let $f_{\texttt{bad-knowledge},\Pi}$ and $f_{\texttt{bad-branch},\Pi}$ be functions that map execution traces to parameterized CoSP protocols[6] whose leaves are either *ok*, in which case they have open edges, or nodes that raise the event `bad-knowledge`, or `bad-branch` respectively.

Recall that nodes ν of Π have bi-references (as defined in Definition 4) consisting of a left reference (to be used in the left protocol) and a right reference. We write $\mathsf{left}(\nu)$ for the node with only the left reference and $\mathsf{right}(\nu)$ analogously. For each node ν in Π, let tr_ν be the execution trace of Π that leads to ν, i.e., the list of node and edge identifiers on the path from the root of Π to ν, including ν. The *self-monitor* $\mathsf{Mon}(\Pi)$ protocol is defined as follows:

Insert before the root node a control node with two copies of Π, called the left branch (with $b := \mathsf{left}$) and the right branch (with $b := \mathsf{right}$). Apply the following modifications recursively for each node ν, starting at the root of Π:

1. If ν is a *computation node of* Π, replace ν with $f_{\texttt{bad-branch},\Pi}(b, tr_\nu)$. Append two copies $\mathsf{left}(\nu)$ and $\mathsf{right}(\nu)$ of the the computation node ν to each open edge of an *ok*-leaf. All left references that pointed to ν point in $\mathsf{Mon}(\Pi)$ to $\mathsf{left}(\nu)$, and all right references that pointed to ν point in $\mathsf{Mon}(\Pi)$ to $\mathsf{right}(\nu)$. The successor of $\mathsf{right}(\nu)$ is the subtree rooted at the successor of ν.

2. If ν is an *output node* of Π, replace ν with $f_{\texttt{bad-knowledge},\Pi}(b, tr_\nu)$. Append the sequence of the two output nodes $\mathsf{left}(\nu)$ (labeled with `left`) and $\mathsf{right}(\nu)$ (labeled with `right`) to each open edge of an *ok*-leaf. All left references that pointed to ν point in $\mathsf{Mon}(\Pi)$ to $\mathsf{left}(\nu)$, and all right references that pointed to ν point in $\mathsf{Mon}(\Pi)$ to $\mathsf{right}(\nu)$. The successor of $\mathsf{right}(\nu)$ is the subtree rooted at the successor of ν. $\diamond$

Our main theorem (Theorem 1) follows from two properties of the distinguishing self-monitors: *symbolic self-monitoring* and *computational self-monitoring* (see Figure 1). Symbolic self-monitoring states that whenever a bi-protocol Π is indistinguishable, the corresponding distinguishing self-monitor in $\mathsf{Mon}(\Pi)$ does not raise the event `bad`. Computational self-monitoring, in turn, states that whenever the distinguishing self-monitor in $\mathsf{Mon}(\Pi)$ does not raises the event `bad`, then Π is indistinguishable.

Shortened Protocols Π_i. Since we prove Theorem 1 by induction over the nodes in a bi-protocol, we introduce a notion of *shortened protocols* in the definition of distinguishing self-monitors. For a (bi-)protocol Π, the shortened (bi-)protocol Π_i is for the first i nodes exactly like Π but that stops after the ith node that is either a control node or an output node.[7]

Definition 23 (Distinguishing self-monitors) Let $\mathbf{M}$ be a symbolic model and $\mathbf{A}$ a computational implementation of $\mathbf{M}$. Let Π be a bi-protocol and $\mathsf{Mon}(\Pi)$ its self-monitor. Let $e \in \{\texttt{bad-knowledge}, \texttt{bad-branch}\}$ and $n_{\texttt{bad-knowledge}}$ denote the node type output node and $n_{\texttt{bad-branch}}$ denote the node type control node. Then the function $f_{e,\Pi}(b, tr)$, which takes as input $b \in \{\mathsf{left}, \mathsf{right}\}$ and the path to the root node, including all node and edge identifiers, is a *distinguishing self-monitor for e* for Π and $\mathbf{M}$ if it is computable in deterministic polynomial time, and if the following conditions hold for every $i \in \mathbb{N}$:

[6]These functions are candidates for distinguishing self-monitors for `bad-knowledge` and `bad-branch`, respectively, for the bi-protocol Π, as defined in Definition 23.

[7]Formally, the protocol only has an infinite chain of control nodes with single successors after this node.

1. *symbolic self-monitoring:* If Π_i is symbolically indistinguishable, `bad` does symbolically not occur in $\mathsf{Mon}(\Pi_{i-1})$, and the ith node in Π_i is of type n_e, then the event e does not occur symbolically in $\mathsf{Mon}(\Pi_i)$.
2. *computational self-monitoring:* If the event e in $\mathsf{Mon}(\Pi_i)$ occurs computationally with at most negligible probability, Π_{i-1} is computationally indistinguishable, and the ith node in Π_i is of type n_e, then Π_i is computationally indistinguishable.

We say that a $\mathbf{M}$ and a protocol class *allow for self-monitoring* if for every bi-protocol Π in the protocol class P, there are distinguishing self-monitors for `bad-branch` and `bad-knowledge` such that the resulting self-monitor $\mathsf{Mon}(\Pi)$ is also in P. $\diamond$

Finally, we are ready to state our main theorem.

Theorem 1 *Let $\mathbf{M}$ be a symbolic model and* P *be an efficient uniformity-enforcing class of bi-protocols. If $\mathbf{M}$ and* P *allow for self-monitoring (in the sense of Definition 23), then the following holds: If $\mathbf{A}$ is a computationally sound implementation of a symbolic model $\mathbf{M}$ with respect to trace properties then $\mathbf{A}$ is also a computationally sound implementation with respect to equivalence properties.*

Proof. Let $\mathbf{A}$ be a computationally sound implementation of $\mathbf{M}$ for trace properties. Assume that Π is symbolically indistinguishable. The goal is show that Π is also computationally indistinguishable.

First, we show that `bad` does not happen symbolically in $\mathsf{Mon}(\Pi)$. To this end, we show by induction on i that `bad` does not happen symbolically in $\mathsf{Mon}(\Pi_i)$ for all $i \in \mathbb{N}$. For the induction base $i = 0$, i.e., the empty execution, this is clear. For $i > 0$, we know by the induction hypothesis that `bad` does not happen symbolically in $\mathsf{Mon}(\Pi_{i-1})$. Furthermore, Π_i is symbolically indistinguishable, because Π is symbolically indistinguishable. We distinguish cases on the type of the ith control or output node ν_i that is reached in Π. Assume it is an control node. Then we know by Item 2 of the uniformity-enforcing property that Π_i contains in contrast to Π_{i-1} only additional input and computation nodes (and ν_i itself). Thus by construction of $\mathsf{Mon}(\Pi)$, we know that $\mathsf{Mon}(\Pi_i)$ does not contain an additional distinguishing self-monitor for `bad-knowledge` in contrast to $\mathsf{Mon}(\Pi_{i-1})$, i.e., `bad-knowledge` does not happen in $\mathsf{Mon}(\Pi_i)$. Since ν_i is a control node, Item 1 of Definition 23 implies that `bad-branch` does not happen in $\mathsf{Mon}(\Pi_i)$ either. The case that ν_i is an output node is analogous. Taken together, `bad` does not happen in $\mathsf{Mon}(\Pi_i)$. This concludes the induction proof.

Since $\mathbf{M}$ and P allow for self-monitoring, the self-monitor $\mathsf{Mon}(\Pi)$ lies in the protocol class P. As `bad` does not happen symbolically in $\mathsf{Mon}(\Pi)$, CS for trace properties thus entails that `bad` happens computationally in $\mathsf{Mon}(\Pi)$ only with negligible probability. This implies computational indistinguishability with essentially the same arguments as in the symbolic execution apply in a reverse manner, by using Item 2 of Definition 23. Note that the left and the right computational executions of Π stop with overwhelming probability at the same node due to the uniformity-enforcing property and because both protocols share the same computation nodes (they only differ in the references).[8] Thus the value of the counter *len* behaves identically in both computational executions, until they stop. This concludes the proof. $\square$

4 The applied π-calculus

In this section, we present the connection between uniform bi-processes in the applied π-calculus and our CS result in CoSP, namely that the applied π-calculus can be embedded into the extended CoSP framework. In contrast to previous work [16, 21, 22], we consider CS for bi-protocols from the full applied π-calculus. In particular, we also consider private channels and non-determinate processes.

We consider the variant of the applied π-calculus also used for the original CoSP embedding [3]. The set of terms in the applied π-calculus is generated by a countably infinite set of names, variables x, and constructors f that can be applied to terms. Destructors are partial functions that processes can apply to terms.

Plain processes are defined as follows. The null process 0 is a neutral element, i.e., it does nothing. *new* $n.P$ generates a fresh name n and then executes P. $a(x).P$ receives a message m from channel a and then executes $P\{m/x\}$. $\overline{a}\langle N\rangle.P$ outputs message n on channel a and then executes P; $P \mid Q$ runs P and Q in parallel. $!P$ behaves as an unbounded number of copies of P in parallel. *let* $x = D(\underline{t})$ *in* P *else* Q applies the destructor D to the terms $\underline{t}$; if application succeeds and produces the term t' ($D(\underline{t}) = t'$),

[8] We further assume an encoding of references that influences the length of the node identifiers equally in the left and the right protocol.

then the process behaves as $P\{t/x\}$; otherwise, i.e., if $D(\underline{t}) = \bot$, the process behaves as Q. The scope of names and variables can be bound by restrictions, inputs, and lets. A *context* $C[\bullet]$ is a process with $\bullet$ instead of a subprocess and for which an *evaluation context* $\bullet$ is not under a replication, a conditional, an input, or an output.

The operational semantics of the applied π-calculus is defined in terms of *structural equivalence* ($\equiv$) and *internal reduction* ($\rightarrow$); for a precise definition of the applied π-calculus, we refer to [13].

A uniform bi-process [14] in the applied π-calculus is the counterpart of a uniform bi-protocol in CoSP. A bi-process is a pair of processes that only differ in the terms they operate on. Formally, they contain expressions of the form $choice[a, b]$, where a is used in the left process and b is used in the right one. A bi-process Q can only reduce if both its processes can reduce in the same way.

Definition 24 (Uniform bi-process) A bi-process Q in the applied π-calculus is *uniform* if $\mathsf{left}(Q) \rightarrow R_{\mathsf{left}}$ implies that $Q \rightarrow R$ for some bi-process R with $\mathsf{left}(R) \equiv R_{\mathsf{left}}$, and symmetrically for $\mathsf{right}(Q) \rightarrow R_{\mathsf{right}}$ with $\mathsf{right}(R) \equiv R_{\mathsf{right}}$. $\diamond$

4.1 Embedding into CoSP

The execution of an process P in the applied π-calculus can be modeled as a CoSP protocol $e(P)$ as defined by Backes, Hofheinz, and Unruh [3]. The function e is called embedding. $e(P)$ sends the current state of P to the attacker, who replies with a the description of an evaluation context E that determines which part of P should be reduced next. Thus the attacker is given control over the whole scheduling of the process.

The state of P is used as a node identifier. An execution of the process performs only the following operations on terms: Applying constructors (this includes nonces) and destructors, comparing using *equals*,[9] and sending and receiving terms. However, terms are not encoded directly into the node identifier; instead, the node in which they were created (or received) is referenced instead. This is due to the fact that a CoSP protocol allows to treat terms only as black boxes. Note that the process P and the terms occurring within P will be encoded in the node identifier (encoded as bitstrings). Operations on messages can then be performed by using constructor and destructor nodes, and the input and output of terms is handled using input/output nodes. If the attacker wants to send or receive on a public channel, she is forced to produce the term that corresponds to the channel.

To ensure that the resulting protocols are uniformity-enforcing, we have to modify the embedding slightly as discussed in Section 3.1. The modification is highlighted in blue. This modification clearly does not influence the validity of the computational soundness result of the applied π-calculus that has been established in the original CoSP framework [3] for trace properties.

Definition 25 (Symbolic execution of a process in the applied π-calculus (extends [3])) Let P_0 be a closed process, and let $\mathcal{A}$ be an interactive machine called the attacker. We define the *symbolic π-execution* as an interactive machine SExec_{P_0} that interacts with $\mathcal{A}$:

- *Start*: Let $P := P_0$ (where we rename all bound variables and names such that they are pairwise distinct and distinct from all unbound ones). Let η be a totally undefined partial function mapping variables to terms, let μ be a totally undefined partial function mapping names to terms. Let $a_1, \dots, a_n$ denote the free names in P_0. For each i, choose a different $r_i \in \mathbf{N}_P$ Set $\mu := \mu(a_1 := r_1, \dots, a_n := r_n)$. Send $(r_1, \dots, r_n)$ to $\mathcal{A}$.

- *Main loop:* Send P to the attacker and include the address of the resulting control node in the protocol tree in its out-metadata. Expect an evaluation context E from the attacker and distinguish the following cases:
  - $P = E[M(x).P_1]$: Request two CoSP-terms c, m from the attacker. If $c \cong eval(M\eta\mu)$, set $\eta := \eta(x := m)$ and $P := E[P_1]$.
  - $P = E[\nu a.P_1]$: Choose $r \in \mathbf{N}_P \setminus range(\mu)$, set $P := E[P_1]$ and $\mu := \mu(a := r)$.
  - $P = E[\overline{M}_1\langle N\rangle.P_1][M_2(x).P_2]$: If $eval(M_1)\eta\mu \cong eval(M_2\eta\mu)$ then set $P := E[P_1][P_2]$ and $\eta := \eta(x := eval(N\eta\mu))$.
  - $P = E[let\ x = D\ in\ P_1\ else\ P_2]$: If $m := eval(D\eta\mu) \neq \bot$, set $\eta := \eta(x := m)$ and $P := E[P_1]$. Otherwise set $P := E[P_2]$.

[9] For instance, this is used to determine if two processes can communicate, i.e., if a channel on which a message should be sent matches a channel on which a message should be received.

– $P = E[!P_1]$: Rename all bound variables of P_1 such that they are pairwise distinct and distinct from all variables and names in P and in the domains of η and μ, yielding a process $\tilde{P}_1$. Set $P := E[\tilde{P}_1 \mid !P_1]$.
– $P = E[\overline{M}\langle N\rangle.P_1]$: Request a CoSP-term c from the attacker. If $c \cong \mathit{eval}(M\eta\mu)$, set $P := E[P_1]$ and send $\mathit{eval}(N\eta\mu)$ to the attacker.
– In all other cases, do nothing. ◇

Modifications for bi-processes. If P is the left or the right variant of a bi-process in the applied π-calculus, expressions of the form $\mathit{choice}[a, b]$ are translated to subtrees in the CoSP protocol that compute both a and b entirely. References to such expressions are translated to bi-references in the natural way. When a description of the process P is sent to the attacker, expressions of the form $\mathit{choice}[a, b]$ are sent untouched; if only a would be sent left and only b right, the attacker could trivially distinguish the bi-protocol.

Relating CoSP and the applied π-calculus. The following lemma connects uniformity in the applied π-calculus to uniformity in CoSP by demonstrating that uniform bi-processes in the applied π-calculus correspond to uniform bi-protocols in CoSP.

Lemma 2 (Uniformity in CoSP and the applied π-calculus) *Let a bi-process Q in the applied π-calculus be given. There is an embedding e from bi-processes in the applied π-calculus to CoSP bi-protocols with the following property: If $\mathsf{left}(Q)$ and $\mathsf{right}(Q)$ are uniform, then $\mathsf{left}(e(Q)) \approx_s \mathsf{right}(e(Q))$ and $\mathit{range}(e)$ is uniformity-enforcing.*

Proof. We show the contrapositive. Suppose that we have $\mathsf{left}(e(Q)) \not\approx_s \mathsf{right}(e(Q))$. Then there there is a distinguishing attacker strategy V_{In}, i.e., there is a view $V \in [V_{In}]_{\mathsf{SViews}(\mathsf{left}(e(Q)))}$ such that for all $V' \in [V_{In}]_{\mathsf{SViews}(\mathsf{right}(e(Q)))}$ we have $V \not\sim V'$ (the symmetric case is completely analogous).

Let V, V' be defined accordingly. As $V \sim V'$, there is a shortest prefix v such that for the prefix v' of V' (of the same as v), we have $v \not\sim v'$. Let v_{In} be the corresponding prefix of the attacker strategy V_{In}. Recall that v_{In} is a list that contains operations to create input terms to the protocol and in-metadata to schedule the execution of the protocol. This suffices to construct an evaluation context E_{In} from v_{In}: By construction of e, input nodes are used in $e(Q)$ to give the attacker the possibility to send or receive on a channel. If the attacker is able to produce the channel term, the corresponding action is performed; in the send case the attacker has to provide the term additionally. The operations in v_{In} are used in E_{In} to produce the required channel name as well as the term to be sent. Moreover, the in-meta data yields a unique reduction t for $\mathsf{left}(Q)$ such that $E_{In}[\mathsf{left}(Q)] \rightarrow^t P_{\mathsf{left}}$.

We distinguish three cases. First, the shape is different (1): there is an i such that $v_i = (s, x)$ and $v'_i = (s', y)$ and $s \neq s'$ ($s, s' \in \{\texttt{out}, \texttt{in}, \texttt{control}\}$). Second, the out-metadata is different (2): $v_{\mathit{Out\text{-}Meta}} \neq v'_{\mathit{Out\text{-}Meta}}$. Third, static equivalence fails (3): $K(v_{Out}) \neq K(v'_{Out})$.

In case (1), different shapes in the left and right protocol correspond to different protocol actions and thus to different cases in the main loop of the symbolic execution of the process in the applied π-calculus. By construction of this execution, we conclude that the left protocol has received other in-metadata that the right one. This contradicts the fact that V and V' are views under the same attacker strategy V_{In}.

For case (2), recall that the out-metadata at control nodes is only used to send a state of the executed process to the attacker, and to raise events. Thus, if the out-metadata differs, then the left and the right process have reduced differently. More formally, we have $\mathsf{left}(Q) \rightarrow P_{\mathsf{left}}$ but there is no bi-process P such that $Q \rightarrow P$ and $\mathsf{left}(P) = P_{\mathsf{left}}$ because otherwise there would be a view v' of $\mathsf{right}(e(Q))$ inducing a reduction $\mathsf{right}(Q) \rightarrow P_{\mathsf{right}}$ with $P_{\mathsf{right}} \equiv \mathsf{right}(P)$.[10] This shows that Q is not uniform.

In the remaining case (3) we construct an evaluation context E_u that breaks the observational equivalence (and thus the uniformity) of P. We know that there is a symbolic operation O such that $K_V(O) \neq K'_V(O)$. In other words, $O(V_{Out}) = \bot$ and $O(V'_{Out}) \neq \bot$ (or vice versa). This induces the context E_u that executes the constructors and destructors corresponding to O and branches depending on the result of the whole operation. Finally, the combined context $E_u[E_{In}[\cdot]]$ distinguishes $\mathsf{left}(Q)$ and $\mathsf{right}(Q)$. Thus Q is not uniform. □

5 Case study: encryption and signatures with lengths

We exemplify our method by proving a CS result for equivalence properties, which captures protocols that use public-key encryption and signatures. We use the CS result in [9] for trace properties, which we

[10]Note that the CoSP bi-protocol is deterministic, because it is efficient in the sense of Definition 13. Hence it additionally renames bound variables and chooses nonces $r_i \in \mathbf{N_P}$ consistently.

extend by a length function, realized as a destructor. Since encryptions of plaintexts of different length can typically be distinguished,we must reflect that fact in the symbolic model.

5.1 The symbolic model

Lengths in the symbolic model. In order to express lengths in the symbolic model, we introduce *length specifications*, which are the result of applying a special destructor *length*/1. We assume that the bitlength of every computational message m_c is of the form $|m_c| = rk$ for some natural number r, where k is the security parameter, i.e., the length of a nonce. This assumption will be made precise. With this simplification, length specifications only encode r; this can be done using Peano numbers, i.e., the constructors O (zero) and S (successor).

Even though this approach leads admittedly to rather inefficient realizations from a practical point of view,[11] the aforementioned assumption can be realized using a suitable padding. Essentially, this assumption is similar to the one introduced by Comon-Lundh and Cortier [21] for a symbolic model for symmetric encryption. The underlying problem is exactly the same: while the length of messages in the computational model, in particular the length of ciphertexts, may depend on the security parameter, there is no equivalent concept in the symbolic model. For instance, let n and m be nonces, and let ek be an encryption key. For certain security parameters in the computational model, the computational message $pair(n, m)$ may have the same length as the message $enc(ek, n)$; for other security parameters this may not be the case. Thus it is not clear if the corresponding symbolic messages should be of equal symbolic length. Comon-Lundh et al. [23] propose a different approach towards this problem, by labeling messages symbolically with an expected length and checking the correctness of these length computationally. However, it is not clear whether such a symbolic model can be handled by current automated verification tools.

Automated verification: combinding ProVerif and APTE. ProVerif is not able to handle recursive destructors such as *length*, e.g., $length(pair(t_1, t_2)) = length(t_1) + length(t_2)$. Recent work by Cheval, Cortier, and Plet [20] extends the protocol verifier APTE, which is capable of proving trace equivalence of two processes in the applied π-calculus, and supports such length functions. Since however trace equivalence is a weaker notion than uniformity, i.e., there are bi-processes that are trace equivalent but not uniform, our CS result does not carry over to APTE. Due to the lack of a tool that is able to check uniformity as well as to handle length functions properly, we elaborate and prove in Appendix A how APTE can be combined with ProVerif to make protocols on the symbolic model of our case study amenable to automated verification.

We consider the following symbolical model $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$.

Constructors and nonces. We define $\mathbf{C} := \{enc/3, ek/1, dk/1, sig/3, vk/1, sk/1, string_0/1, string_1/1, emp/0, pair/2, O/0, S/1, garbageEnc/3, garbageSig/3, garbage/2, garbageInvalidLength/1\}$ and $\mathbf{N} := \mathbf{N_E} \uplus \mathbf{N_E}$ for countably infinite sets of protocol nonces $\mathbf{N_P}$ and attacker nonces $\mathbf{N_E}$. Encryption, decryption, verification, and signing keys are represented as $ek(r), dk(r), vk(r), sk(r)$ with a nonce r (the randomness used when generating the keys). The term $enc(ek(r'), m, r)$ encrypts m using the encryption key $ek(r')$ and randomness r. $sig(sk(r'), m, r)$ is a signature of m using the signing key $sk(r')$ and randomness r. The constructors $string_0$, $string_1$, and emp are used to model arbitrary strings used as payload in a protocol, e.g., a bitstring 010 would be encoded as $string_0(string_1(string_0(emp())))$. Length specifications can be constructed using O representing zero and S representing the successor of a number. *garbage*, *garbageInvalidLength*, *garbageEnc*, and *garbageSig* are not used by the protocol; they express invalid terms the attacker may send.

Message type. We define $\mathbf{T}$ as the set of terms M according to this grammar:

$$
\begin{aligned}
M ::= {} & enc(ek(N), M, N) \mid ek(N) \mid dk(N) \mid \\
& sig(sk(N), M, N) \mid vk(N) \mid sk(N) \mid pair(M, M) \mid P \mid N \mid L \mid \\
& garbage(N, L) \mid garbageInvalidLength(N) \\
& garbageEnc(M, N, L) \mid garbageSig(M, N, L) \\
P ::= {} & emp() \mid string_0(P) \mid string_1(P) \qquad L ::= O() \mid S(L)
\end{aligned}
$$

The nonterminals P, N, and L represent payloads, nonces, and length specifications, respectively. Note that the garbage terms carry an explicit length specification to enable the attacker to send invalid terms of a certain length.

[11] Consider, e.g., a payload string that should convey n bits. This message must be encoded using at least kn bits.

Destructors. We define the set $\mathbf{D}$ of destructors of the symbolic model $\mathbf{M}$ by $\mathbf{D} := \{dec/2, isenc/1, isek/1, isdk/1, ekof/1, ekofdk/1, equals/2, verify/2, isvk/1, issk/1, issig/1, vkofsk/1, vkof/1, fst/1, snd/1, unstring_0/1, unstring_1/1, length/1, unS/1\}$. The destructors *isek*, *isdk*, *isvk*, *issk*, *isenc*, and *issig* realize predicates to test whether a term is an encryption key, decryption key, verification key, signing key, ciphertext, or signature, respectively. *ekof* extracts the encryption key from a ciphertext, *vkof* extracts the verification key from a signature. $dec(dk(r), c)$ decrypts the ciphertext c. $verify(vk(r), s)$ verifies the signature s with respect to the verification key $vk(r)$ and returns the signed message if successful. *ekofdk* and *vkofsk* compute the encryption/verification key corresponding to a decryption/signing key. The destructors *fst* and *snd* are used to destruct pairs, and the destructors $unstring_0$ and $unstring_1$ allow to parse payload-strings. The destructor *length* returns a the length of message, where the unit is the length of a nonce. The purpose of *unS* is destruct length specifications. (Destructors *ispair* and *isstring* are not necessary, they can be emulated using *fst*, $unstring_i$, and $equals(\cdot, empty)$.) The precise cancellation rules for destructors (except for *length*) are as follows; application matching none of these rules evaluates to $\bot$:

$$\begin{aligned}
dec(dk(t_1), enc(ek(t_1), m, t_2)) &= m \\
isenc(enc(ek(t_1), t_2, t_3)) &= enc(ek(t_1), t_2, t_3) \\
isenc(garbageEnc(t_1, t_2, l)) &= garbageEnc(t_1, t_2, l) \\
isek(ek(t)) &= ek(t) \\
ekof(enc(ek(t_1), m, t_2)) &= ek(t_1) \\
ekof(garbageEnc(t_1, t_2, l)) &= t_1 \\
equals(t_1, t_1) &= t_1 \\
verify(vk(t_1), sig(sk(t_1), t_2, t_3)) &= t_2 \\
isvk(vk(t_1)) &= vk(t_1) \\
issk(sk(t_1)) &= sk(t_1) \\
issig(sig(sk(t_1), t_2, t_3)) &= sig(sk(t_1), t_2, t_3) \\
issig(garbageSig(t_1, t_2, l)) &= garbageSig(t_1, t_2, l) \\
vkof(sig(sk(t_1), t_2, t_3)) &= vk(t_1) \\
vkof(garbageSig(t_1, t_2, l)) &= t_1 \\
vkofsk(sk(t_1)) &= vk(t_1) \\
fst(pair(x, y)) &= x \\
snd(pair(x, y)) &= y \\
unstring_b(string_b(s)) &= s \qquad \forall b \in \{0, 1\} \\
unS(S(t)) &= t
\end{aligned}$$

Length destructor. Our result is parametrized over the destructor *length* that must adhere to the following restrictions:

1. Each message except for *garbageInvalidLength* is assigned a length: $length(t) \neq \bot$ for all terms $t \in \mathbf{T} \setminus \{garbageInvalidLength(t') \mid t' \in \mathbf{T}\}$.

2. The length of garbage terms (constructed by the attacker) is consistent:

$$\begin{aligned}
length(garbage(t, l)) &= l, & length(garbageEnc(t_1, t_2, l)) = l, \\
length(garbageSig(t_1, t_2, l)) &= l, & length(garbageInvalidLength(t_1)) = \bot
\end{aligned}$$

3. Let $[\cdot]$ be the canonical interpretation of Peano numbers, given by $[O()] = 0$ and $[S(l)] = [l] + 1$. We require the length destructor to be linear: For each constructor $C/n \in \mathbf{C} \setminus \{garbage, garbageInvalidLength, garbageEnc, garbageSig\}$ there are $a_i \in \mathbb{N}$ (where $i = 0, \dots, n$) such that $length(t_i) = l_i$ for $i = 1, \dots, n$ and $length(C(\underline{t})) = l$ together imply $[l] = \sum_{i=1}^{n} a_i \cdot [l_i] + a_0$.

4. Given a security parameter k, a computational variant of a message $m \in \mathbf{T}$ is obtained by implementing each constructor C and nonce N in m by the corresponding algorithm A_C or A_N, respectively. (For example, for all random choices of $A_N()$, $A_{pair}(A_{string_0}(A_{emp}(), A_{ek}(A_N())))$ is a

computational variant of the message $pair(string_0(emp(), ek(N))$, where $N \in \mathbf{N}$.) We require that for each message $m \in \mathbf{T}$ and all of its computational variants m_k under security parameter k, we have that $length(m) \neq \bot$ implies $|m_k| = [length(m)] \cdot k$. Note that $|m_k|$ is well-defined, because length-regularity (implementation condition 2 in Section 5.2) ensures that $|m_k|$ does not depend on randomness.

Length specifications are ordinary messages that the protocol can send, receive and process. Thus we require length specifications to have a length itself.

5.2 Implementation conditions

A computationally sound implementation of the symbolic model $\mathbf{M}$ has to adhere to the conditions given below, which are essentially the same as in [9]. Since the message type $\mathbf{T}$ used here includes length specifications, i.e., an additional type of messages that represents natural numbers (see Section 5.1), we need basic implementation conditions such as $A_{unS}(A_S(m)) = m$ for messages m of type length specification. Furthermore, the algorithm that implements the length destructor must compute the bitlength of the argument correctly. These additional requirements are highlighted in blue. We stress that the strong requirements on the encryption scheme, which require the random oracle model, namely PROG-KDM security [37], are used only to handle protocols that send and receive decryption keys. We refer to [9] for more details. In principle, our proofs do not rely on this particular security definition. We conjecture that is possible to obtain a computational soundness result for uniformity using weaker implementation conditions (IND-CCA secure public-key encryption) but a restricted protocol class, by applying our proof technique to the computational soundness result for trace properties in [3]; we leave a formal treatment for future work however.

For lengths in the computational model, we require that the destructor *length* as well as its computational implementation Imp_{length} compute indeed the bitlength of its argument. To connect the symbolic result of the destructor *length* to bitlengths in the computational world, the destructor must be consistent with its implementation.

1. There are disjoint and efficiently recognizable sets of bitstrings representing the types nonces, ciphertexts, encryption keys, decryption keys, signatures, verification keys, signing keys, pairs, payload-strings, length specifications, and invalid-length. The set of all bitstrings of type nonce we denote Nonces_k.[12] (Here and in the following, k denotes the security parameter.)

2. The functions A_{enc}, A_{ek}, A_{dk}, A_{sig}, A_{vk}, A_{sk}, and A_{pair} are length-regular. We call an n-ary function f length regular if $|m_i| = |m'_i|$ for $i = 1, \dots, n$ implies $|f(\underline{m})| = |f(\underline{m}')|$. All $m \in \mathrm{Nonces}_k$ have the same length.

3. A_N for $N \in \mathbf{N}$ returns a uniformly random $r \in \mathrm{Nonces}_k$.

4. Every image of A_{enc} is of type ciphertext, every image of A_{ek} and A_{ekof} is of type encryption key, every image of A_{dk} is of type decryption key, every image of A_{sig} is of type signature, every image of A_{vk} and A_{vkof} is of type verification key, every image of A_{empty}, A_{string_0}, and A_{string_1} is of type payload-string, every image of A_S and A_O is of type length specification. Every $m \in \{0,1\}^*$ such that no $r \in \mathbb{N}$ with $|m| = rk$ exists, is of type invalid-length.

5. For all $m_1, m_2 \in \{0,1\}^*$ we have $A_{fst}(A_{pair}(m_1, m_2)) = m_1$ and $A_{snd}(A_{pair}(m_1, m_2)) = m_2$. Every m of type pair is in the range of A_{pair}. If m is not of type pair, $A_{fst}(m) = A_{snd}(m) = \bot$.

6. For all m of type payload-string we have that $A_{unstring_i}(A_{string_i}(m)) = m$ and $A_{unstring_i}(A_{string_j}(m)) = \bot$ for $i, j \in \{0,1\}$, $i \neq j$. For $m = A_{empty}()$ or m not of type payload-string, $A_{unstring_0}(m) = A_{unstring_1}(m) = \bot$. Every m of type payload-string is of the form $m = A_{string_0}(m')$ or $m = A_{string_1}(m')$ or $m = emp$ for some m' of type payload-string. For all m of type payload-string, we have $|A_{string_0}(m)|, |A_{string_1}(m)| > |m|$.

7. For all m of type length specification we have that $A_{unS}(A_S(m)) = m$. For $m = A_O()$ or m not of type number, $A_{unS}(m) = \bot$.

8. $A_{ekof}(A_{enc}(p, x, y)) = p$ for all p of type encryption key, $x \in \{0,1\}^*$, $y \in \mathrm{Nonces}_k$. $A_{ekof}(e) \neq \bot$ for any e of type ciphertext and $A_{ekof}(e) = \bot$ for any e that is not of type ciphertext.

[12] This would typically be the set of all bitstrings with length $k - t$, with a tag of length t denoting nonces.

9. $A_{vkof}(A_{sig}(A_{sk}(x), y, z)) = A_{vk}(x)$ for all $y \in \{0,1\}^*$, $x, z \in \mathrm{Nonces}_k$. $A_{vkof}(e) \neq \bot$ for any e of type signature and $A_{vkof}(e) = \bot$ for any e that is not of type signature.

10. $A_{enc}(p, m, y) = \bot$ if p is not of type encryption key.

11. $A_{dec}(A_{dk}(r), m) = \bot$ if $r \in \mathrm{Nonces}_k$ and $A_{ekof}(m) \neq A_{ek}(r)$. (This implies that the encryption key is uniquely determined by the decryption key.)

12. $A_{dec}(d, c) = \bot$ if $A_{ekof}(c) \neq A_{ekofdk}(d)$ or $A_{ekofdk}(d) = \bot$.

13. $A_{dec}(d, A_{enc}(A_{ekofdk}(e), m, r)) = m$ if $r \in \mathrm{Nonces}_k$ and $d := A_{ekofdk}(e) \neq \bot$.

14. $A_{ekofdk}(d) = \bot$ if d is not of type decryption key.

15. $A_{ekofdk}(A_{dk}(r)) = A_{ek}(r)$ for all $r \in \mathrm{Nonces}_k$.

16. $A_{vkofsk}(s) = \bot$ if s is not of type signing key.

17. $A_{vkofsk}(A_{sk}(r)) = A_{vk}(r)$ for all $r \in \mathrm{Nonces}_k$.

18. $A_{dec}(A_{dk}(r), A_{enc}(A_{ek}(r), m, r')) = m$ for all $r, r' \in \mathrm{Nonces}_k$.

19. $A_{verify}(A_{vk}(r), A_{sig}(A_{sk}(r), m, r')) = m$ for all $r, r' \in \mathrm{Nonces}_k$.

20. For all $p, s \in \{0,1\}^*$ we have that $A_{verify}(p, s) \neq \bot$ implies $A_{vkof}(s) = p$.

21. $A_{isek}(x) = x$ for any x of type encryption key. $A_{isek}(x) = \bot$ for any x not of type encryption key.

22. $A_{isvk}(x) = x$ for any x of type verification key. $A_{isvk}(x) = \bot$ for any x not of type verification key.

23. $A_{isenc}(x) = x$ for any x of type ciphertext. $A_{isenc}(x) = \bot$ for any x not of type ciphertext.

24. $A_{issig}(x) = x$ for any x of type signature. $A_{issig}(x) = \bot$ for any x not of type signature.

25. We define an encryption scheme $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ as follows: KeyGen picks a random $r \leftarrow \mathrm{Nonces}_k$ and returns $(A_{ek}(r), A_{dk}(r))$. $\mathsf{Enc}(p, m)$ picks a random $r \leftarrow \mathrm{Nonces}_k$ and returns $A_{enc}(p, m, r)$. $\mathsf{Dec}(k, c)$ returns $A_{dec}(k, c)$. We require that then $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is PROG-KDM secure.

26. Additionally, we require that $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is malicious-key extractable.

27. We define a signature scheme $(\mathsf{SKeyGen}, \mathsf{Sig}, \mathsf{Verify})$ as follows: $\mathsf{SKeyGen}$ picks a random $r \leftarrow \mathrm{Nonces}_k$ and returns $(A_{vk}(r), A_{sk}(r))$. $\mathsf{Sig}(p, m)$ picks a random $r \leftarrow \mathrm{Nonces}_k$ and returns $A_{sig}(p, m, r)$. $\mathsf{Verify}(p, s, m)$ returns 1 iff $A_{verify}(p, s) = m$. We require that then $(\mathsf{SKeyGen}, \mathsf{Sig}, \mathsf{Verify})$ is strongly existentially unforgeable.

28. For all e of type encryption key and all $m, m' \in \{0,1\}^*$, the probability that $A_{enc}(e, m, r) = A_{enc}(e, m', r')$ for uniformly chosen $r, r' \in \mathrm{Nonces}_k$ is negligible.

29. For all $r_s \in \mathrm{Nonces}_k$ and all $m \in \{0,1\}^*$, the probability that $A_{sig}(A_{sk}(r_s), m, r) = A_{sig}(A_{sk}(r_s), m, r')$ for uniformly chosen $r, r' \in \mathrm{Nonces}_k$ is negligible.

30. A_{ekofdk} is injective. (That is, the encryption key uniquely determines the decryption key.)

31. A_{vkofsk} is injective. (That is, the verification key uniquely determines the signing key.)

32. For all $m \in \{0,1\}^*$ that are not of type invalid-length, $A_{length}(m) = A_S^{|m|/k}(A_O())$, where k is the security parameter and A_S^n is the n-fold application of A_S. For all m of type invalid-length, $A_{length}(m) = \bot$.

5.3 CS for trace properties with length functions

The CoSP bi-protocols we consider are almost exactly those bi-protocol for which both the left and the right variant are randomness-safe as defined in [9]. The reason is that the self-monitor Π' for the case study, which uses the distinguishing self-monitors from Section 5.4, is in the protocol class of randomness-safe protocol considered in [9] if the left and the right variant of the corresponding bi-protocol Π are randomness-safe. The exact definition is as follows, changes in comparison to [9] are highlighted in blue. The new condition 6 ensures that no constructors are applied to messages with invalid lengths, i.e., terms of the form $garbageInvalidLength(n)$ for a nonce in the symbolic model. Such messages are not generated by the protocol (5). If they have been received at an input node, the protocol must check if at least one destructor does not fail before applying a constructor. This suffices as all destructors fail if one of their arguments has an invalid length.

Definition 26 (Randomness-safe bi-protocol) A CoSP **bi-**protocol Π *randomness-safe* if both its variants, i.e., $\mathsf{left}(\Pi)$ and $\mathsf{right}(\Pi)$, fulfill the following conditions:

1. The argument of every *ek*-, *dk*-, *vk*-, and *sk*-computation node and the third argument of every E- and *sig*-computation node is an N-computation node with $N \in \mathbf{N_P}$. (Here and in the following, we call the nodes referenced by a protocol node its arguments.) We call these N-computation nodes *randomness nodes*. Any two randomness nodes on the same path are annotated with different nonces.

2. Every computation node that is the argument of an *ek*-computation node or of a *dk*-computation node on some path p occurs only as argument to *ek*- and *dk*-computation nodes on that path p.

3. Every computation node that is the argument of a *vk*-computation node or of an *sk*-computation node on some path p occurs only as argument to *vk*- and *sk*-computation nodes on that path p.

4. Every computation node that is the third argument of an E-computation node or of a *sig*-computation node on some path p occurs exactly once as an argument in that path p.

5. There are no computation nodes with the constructors *garbage*, *garbageEnc*, *garbageSig*, *garbageInvalidLength*, or $N \in \mathbf{N_E}$.

6. Every computation node annotated ν with a constructor refers only to argument nodes ν' that fulfill one of the these conditions:

   (a) Either ν' does not depend on an input node, i.e., no input node is reachable by following (transitively) the references to argument nodes, or

   (b) ν is in the yes-branch of a computation node with a destructor that has ν' as one of its arguments. ◇

The following theorem follows from the CS result of Backes, Unruh, and Malik [9]. We discuss the differences to the CS proof in [9].

Theorem 3 *Let* $\mathbf{A}$ *be a computational implementation fulfilling the implementation conditions (see Appendix 5.2), i.e., in particular* $\mathbf{A}$ *is length-consistent. Then,* $\mathbf{A}$ *is a computationally sound implementation of the symbolic model* $\mathbf{M}$ *for the class of randomness-safe protocols (see Definition 26).*

Proof. The implementation conditions that we require are a superset of the conditions in the work of Backes, Malik, and Unruh (our additional conditions are marked blue). Hence, every implementation that satisfies our implementation condition also satisfies the implementation condition of their work. Moreover, we add one protocol condition for that excludes *garbageInvalidLength*-terms as arguments for constructors. This protocol condition only further restricts the class; hence, without length functions, their CS result would still hold.

We extend the simulator in the CS proof of the work of Backes, Malik, and Unruh to also parse (τ) and produce (β) length functions. The rules are straight-forward and follow the same pattern as for $string_b$ and $unstring_b$.

Length functions are constant functions that the attacker can produce on its own, just like payload strings ($string_b$). Consequently, the Dolev-Yaoness of the simulator also holds in the presence of length functions

For the translation funtions $\beta : \mathbf{T} \to \{0,1\}^*$ and $\tau : \{0,1\}^* \to \mathbf{T}$ of the simulator, $\beta(length(m)) = A_{length}(\beta(m))$, $\tau(\beta(t)) = t$, $\beta(t) \neq \bot$, and $\beta(\tau(b)) = b$ follow from the implementation condition. The indistinguishability of the simulator follows from these equation (see [9]).

As shown in the initial work on CoSP [3], a simulator that satisfies Dolev-Yaoness and Indistinguishability implies computational soundness. □

5.4 Distinguishing self-monitors for the symbolic model M

In this section, we present the distinguishing self-monitors for the symbolic model **M**. We construct a family of distinguishing self-monitors $f_{\texttt{bad-branch},\Pi}(b, tr)$ for computation nodes, which we call *branching monitors*, and a family of distinguishing self-monitors $f_{\texttt{bad-knowledge},\Pi}(b, tr)$ for output nodes, which we call *knowledge monitors*.

We construct a distinguishing self-monitor $f_{\texttt{bad-branch},\Pi}(b, tr)$ for a computation node ν that investigates each message that has been received at an input node (in the execution trace tr of $\mathsf{Mon}(\Pi)$) by parsing the message using computation nodes. The distinguishing self-monitor then reconstructs an attacker strategy by reconstructing a possible symbolic operation for every input message. In more detail, in the symbolic execution, $f_{\texttt{bad-branch},\Pi}(b, tr)$ parses the input message with all symbolic operations in the model **M** that the attacker could have performed as well, i.e., with all tests from the *shared knowledge*. This enables $f_{\texttt{bad-branch},\Pi}(b, tr)$ to simulate the symbolic execution of $\bar{b}(\Pi)$ on the constructed attacker strategy. In the computational execution of the self-monitor, the distinguishing self-monitor constructs the symbolic operations (i.e., the symbolic inputs) by parsing the input messages with the implementations of all tests in the shared knowledge (i.e., lookups on output messages and implementations of the destructors). With this reconstructed symbolic inputs (i.e., symbolic operations, from messages that were intended for $b(\Pi)$, $f_{\texttt{bad-branch},\Pi}(b, tr)$ is able to simulate the symbolic execution of $\bar{b}(\Pi)$ even in the computational execution. The branching monitor $f_{\texttt{bad-branch},\Pi}(b, tr)$ then checks whether this simulated symbolic execution of $\bar{b}(\Pi)$ takes in the same branch as $b(\Pi)$ would take, for the computation node ν in question. If this is not the case, the event `bad-branch` is raised.

Symbolic self-monitoring follows by construction because the branching monitor reconstructs a correct attacker strategy and correctly simulates a symbolic execution. Hence, $f_{\texttt{bad-branch},\Pi}(b, tr)$ found a distinguishing attacker strategy for $b(\Pi)$ and $\bar{b}(\Pi)$. We show computational self-monitoring by applying the CS result for trace properties to conclude that the symbolic simulation of $\bar{b}(\Pi)$ suffices to check whether $b(\Pi)$ computationally branches differently from $\bar{b}(\Pi)$.

The distinguishing self-monitor $f_{\texttt{bad-knowledge},\Pi}(b, tr)$ for an output node ν starts like $f_{\texttt{bad-branch},\Pi}(b, tr)$ by reconstructing a (symbolic) attacker strategy and simulating a symbolic execution of $\bar{b}(\Pi)$. However, instead of testing the branching behavior of $\bar{b}(\Pi)$, the distinguishing self-monitor $f_{\texttt{bad-knowledge},\Pi}(b, tr)$ characterizes the message m that is output in $b(\Pi)$ at the output node ν in question, and then $f_{\texttt{bad-knowledge},\Pi}(b, tr)$ compares m to the message that would be output in $\bar{b}(\Pi)$. This characterization must honor that ciphertexts generated by the protocol are indistinguishable if the corresponding decryption key has not been revealed to the attacker so far. If a difference in the output of $b(\Pi)$ and $\bar{b}(\Pi)$ is detected, the event `bad-knowledge` is raised.

Symbolic self-monitoring of the distinguishing self-monitor $f_{\texttt{bad-knowledge},\Pi}(b, tr)$ follows by the same arguments as for $f_{\texttt{bad-branch},\Pi}(b, tr)$. We show computational self-monitoring by first applying the PROG-KDM property to prove that the computational execution of $b(\Pi)$ is indistinguishable from a *faking* setting: in the faking setting, all ciphertexts generated by the protocol do not carry any information about their plaintexts (as long as the corresponding decryption key has not been leaked). The same holds analogously for $\bar{b}(\Pi)$. We then consider all remaining real messages, i.e., all messages except ciphertexts generated by the protocol with unleaked decryption keys. We conclude the proof by showing that in the faking setting, $f_{\texttt{bad-knowledge},\Pi}(b, tr)$ is able to sufficiently characterize all real messages to raise the event `bad-knowledge` whenever the bi-protocol Π is distinguishable.

Theorem 4 *Let* P *be a uniformity-enforcing class of randomness-safe bi-protocols and* **A** *a computationally sound implementation of the symbolic model* **M**. *For each bi-protocol* Π, $f_{\texttt{bad-knowledge},\Pi}$ *and* $f_{\texttt{bad-branch},\Pi}$ *as described above are distinguishing self-monitors (see Definition 23) for* **M** *and* P.

We prove Theorem 4 in Section 5.7.

5.5 The branching monitor

In this section, we define the distinguishing self-monitor $f_{\texttt{bad-branch},\Pi}(b, tr)$, which we call the branching monitor $f_{\texttt{bad-branch},\Pi}(b, tr)$. The branching monitor $f_{\texttt{bad-branch},\Pi}(b, tr)$ reconstructs an attacker strategy

by reconstructing a possible symbolic operation for every input message. In more detail, in the symbolic execution, $f_{\texttt{bad-branch},\Pi}(b, tr)$ parses the input message with all symbolic operations in the model $\mathbf{M}$ that the attacker could have performed as well, i.e., with all tests from the shared knowledge. This enables $f_{\texttt{bad-branch},\Pi}(b, tr)$ to simulate the symbolic execution of $\bar{b}(\Pi)$ on the constructed attacker strategy. In the computational execution of the self-monitor, it constructs the symbolic operations (i.e., the symbolic inputs) by parsing the input messages with the implementations of all tests in the shared knowledge (i.e., lookups on output messages and implementations of the destructors). With this reconstructed symbolic inputs (i.e., symbolic operations, from messages that were intended for $b(\Pi)$, $f_{\texttt{bad-branch},\Pi}(b, tr)$ is able to simulate the symbolic execution of $\bar{b}(\Pi)$, called the extended symbolic execution, even in the computational execution. The branching monitor $f_{\texttt{bad-branch},\Pi}(b, tr)$ then checks whether this extended symbolic execution of $\bar{b}(\Pi)$ takes the same branch as $b(\Pi)$ would take, for the computation node ν in question. If this is not the case, the event `bad-branch` is raised.

Symbolic self-monitoring follows by construction because the distinguishing self-monitor reconstructs a correct attacker strategy and correctly simulates a symbolic execution. Hence, $f_{\texttt{bad-branch},\Pi}(b, tr)$ found a distinguishing attacker strategy for $b(\Pi)$ and $\bar{b}(\Pi)$. We show computational self-monitoring by reducing a distinguishing event due to different branchings to computational soundness for trace properties. We then conclude that the symbolic simulation of $\bar{b}(\Pi)$ suffices to check whether $b(\Pi)$ computationally branches differently from $\bar{b}(\Pi)$.

5.5.1 Construction of the branching monitor

Recall that a distinguishing self-monitor for `bad-branch` shall test for a bi-protocol Π whether the computation nodes in $\mathsf{left}(\Pi)$ and $\mathsf{right}(\Pi)$ take the same branch. The self-monitor only executes one of the two protocols of Π. For testing that both protocols always take the same branch, we perform a so-called *extended symbolic execution* of the other protocol. For this extended symbolic execution, we construct shapes of the messages that the attacker sends and use these shapes of the input messages for the extended symbolic execution.

Recall that for testing whether the two protocols of a bi-protocol Π always take the same branch at each computation node ν, we execute one of the two protocols, say $b(\Pi)$, in the self-monitor $\mathsf{Mon}(\Pi)$, reconstruct for every computation node the attacker strategy and perform with this attacker strategy a symbolic execution of the other protocol $\bar{b}(\Pi)$.

We reconstruct the attacker strategy by constructing shapes, so-called *extended symbolic operations*, of all attacker-messages of the real execution. These extended symbolic operations contain addresses of the trace of an execution of $\mathsf{left}(\Pi)$ or $\mathsf{right}(\Pi)$. In some cases, it is not possible to completely reconstruct all operations that the attacker did to construct the term. As an example consider an ciphertext sent by the attacker. If the decryption key is not known to the protocol, the distinguishing self-monitor cannot reconstruct the plaintext message. Instead it creates a placeholder with the right length $\mathit{plaintextof}(t_1, \mathit{enc}(t_2, t_3, t_4))$, where t_1 corresponds to the length of the plaintext t_3. In the same manner, we create placeholders $\mathit{skofvk}(\mathit{vk})$ for signature keys of signatures for which we only know the verification key, and we create placeholders $\mathit{nonceof}(m)$ for randomness that has been used, in order to create the same extended symbolic operation for equal attacker-messages m, e.g., ciphertext or signatures with the same key, the same message and the same randomness.

The core idea is to reconstruct a symbolic attacker strategy from the transcript in a manner that works, both, in the symbolic and in the computational model. Then, we use this reconstructed attacker strategy to internally run a symbolic execution of $\bar{b}(\Pi)$ and to check whether the internal symbolic execution of $\bar{b}(\Pi)$ branches differently than the real execution of $b(\Pi)$. As described above, this internal symbolic execution is the extended symbolic execution from Definition 28.

The shape of a message. For characterizing the symbolic knowledge that the adversary has about a message, we characterize the bitstring as detailed as possible in a term-like representation, which we call a *shape*. Technically, a shape is a tree, labelled with constructor and destructor names, but while constructing a shape for a given message m the algorithm Construct-shape, applies all possible tests that the adversary could apply as well to the message and inversely construct a shape, e.g., if for a message m a decryption operation dec (or dec') succeeds with the decryption key k, and to the plaintext $\mathit{unstring}_0$ succeeds, then the shape would be $\mathit{enc}(\mathit{ekofdk}(k), \mathit{string}_0(\mathit{emp}), n)$. In order to make shapes unique and to assign the same shape to the same bitstring, we refine the nonce n as a virtual constructor $\mathit{nonceof}/1$ that gets as an argument the so-called *dual symbolic operation* that lead to the term for which the randomness is retrieved: in our example $n = \mathit{nonceof}(\mathit{dec}(k), m)$. Moreover, in order to be able to obtain a attacker strategy, we have to get rid of all bistrings in a shape. For our example this means that

$f_{\texttt{bad-branch},\Pi}(b, tr)$ **for computation node** d

```
1: V := Construct-attacker-strategy(b, tr)
2: run the extended symbolic execution of b̄(Π) with the attacker strategy V_In
3: if d has not been reached in this execution then
4:    go to a distinct abort-state
5: if the successor of d that is not at the x-edge has been reached then
6:    go to the state bad-branch
7: return ok
```

Construct-knowledge(b, tr, K, j)

```
1: for i = 1 to min(j, |tr|) do
2:    let ν_i be the ith node in b(tr)
3:    let tr' be the prefix-trace from ν_i to the root
4:    if ν_i is an input node then
5:       let m_i be message at the input node ν_i
6:       O_i := Construct-shape(m_i, x_i, K, b, tr', dec')
7:       /*The shared knowledge also increases if an attacker-message is received.*/
8:       K := K ∪ {O_i}
9:       /*saturate the knowledge after adding O_i*/
10:      K := FP-Destruct(K, tr')
11:   else if ν_i is an output node then
12:      let m_i be message at the output node ν_i
13:      O_i := Construct-shape(m_i, x_i, K, b, tr', dec')
14:      K := K ∪ {O_i}
15:      /*saturate the knowledge after adding O_i*/
16:      K := FP-Destruct(K, tr')
17: return K
```

Construct-attacker-strategy(b, tr)

```
1: K := ∅
2: for i = 1 to |tr| do
3:    K := Construct-knowledge(b, tr, K, i)
4: V := ε /*initialization with the empty list*/
5: for i = 1 to |tr| do
6:    let ν_i be the ith node in b(tr)
7:    let tr' be the prefix-trace from ν_i to the root
8:    if ν_i is an input node then
9:       let m_i be message at the input node ν_i
10:      O_i := Construct-shape(m_i, x_i, K, b, tr', dec')
11:      V := V :: (in, (*, O_i))
12:   else if ν_i is an output node then
13:      let m_i be message at the output node ν_i
14:      O_i := Construct-shape(m_i, x_i, K, b, tr', dec')
15:      V := V :: (out, (*, O_i))
16:   else if ν_i is a control node then
17:      let l' be the bitstring in the annotation of the edge between ν_i and the successor of ν_i that has been reached in tr (i.e., the in-metadata the attacker has sent)
18:      V := V :: (control, (*, l'))
19: return V
```

Figure 2: The main loop of the branching monitor.

FP-Destruct(K, tr)

```
1: repeat
2:    K' := K
3:    for all O ∈ K do
4:       K := K ∪ Destruct(D', O, tr)
5:    for all pairs (O, O') ∈ K^2 do
6:       K := K ∪ Destruct-binary(D', O, O', tr)
7:    K := Extended-Shared-Knowledge(K, b, tr')
8: until K reached a fixpoint, i.e.,
9:       ∀O ∈ K.∃O' ∈ K'.eval_O(tr) = eval'_O(tr)
10: return K
```

Destruct(D, O, tr)

```
1: K'' := ∅
2: for all unary destructors d in D do
3:    K'' := K'' ∪ {d(O)}
4: return K''
```

Destruct-binary(D, O, O', tr)

```
1: K'' := ∅
2: for all binary destructors d in D do
3:    K'' := K'' ∪ {d(O, O')}
4: return K''
```

Figure 3: The algorithms FP-Destruct, Destruct, Destruct-binary, $\mathbf{D}'$ is the extended set of destructors (see Section 5.5.2).

m and k have to be replaced in the shape. For k, we recursively apply this construction of a shape to k. As an example, the decryption key could be a message that the adversary sent to the protocol, which was received at the jth node in the protocol states trace. For the message m, we know that it is the message is about to be sent at output node i Then, Construct-shapesucceeds with an $isdk$-operation, and the corresponding shape would additionally point to the jth node in the protocol states trace: the shape is

$$enc(ekofdk(dk(nonceof(x_j))), string_0(emp), nonceof(dec(k), x_i))$$

Constructing the shared knowledge. The shared knowledge is a set of symbolic operations that is constructed by adding to the shared knowledge all messages that have been sent to the adversary and that have been received from the adversary. The algorithm Construct-knowledge(see Figure 3) slowly increases the knowledge set by iterating through the nodes of the protocol state trace, and by constructing the shapes for all previous input and output nodes and by adding these shapes to the knowledge. In this way, it happens that for the same input or output node several time a shape constructed. These different shapes for the same node are, however, consistent with each other. Since Construct-shaperecursively parses a message and the knowledge monotonically increases, each new shape for the same node only differs in that it is a refinement of the old shape, specifically $plaintextof$ sub-shapes are potentially replaced by shapes that contain more information, e.g., $pair(string_1(emp), string_0(emp))$.

Extended shared knowledge. The algorithm Extended-Shared-Knowledge (see Figure 4) is defined that extends the shared knowledge with the plaintexts that the attacker knows because it can decrypt protocol-ciphertexts that use an attacker key. From a ciphertext $enc(ek, m, r')$ with attacker-generated

Extended-Shared-Knowledge(K, b, tr')

1: **for all** $O \in K$ such that $O(b(tr'))$ is a ciphertext **do**
2: **if** $\exists$ no $O' \in K$ such that $O'(b(tr'))$ and $ekof(O(b(tr'))) = ekofdk(dk)$ **then**
3: **if** $ekof(O(b(tr')))$ is not the result of any ek-computation node in tr' **then**
4: **if** $\exists$ an enc-computation node in tr' with the second argument from the jth node **then**
5: $K := K \cup \{x_j\}$
6: **return** K

Figure 4: Extended shared knowledge.

Construct-shape$(m, O_q, K, b, tr, unaryDec)$

1: $O_r := nonceof(O_q)$
2: **switch** m **with**
3: **case** "$\exists O \in K.\ m = eval_O(b(tr))$"
4: **return** O
5: **case** "$isenc(m) \neq \bot$"
6: $O_{ek} :=$ Construct-shape("$ekof(m), ekof$"$(O_q), K, b, tr)$
7: **if** $\exists dk.dk = ekof(m) \wedge \exists O_{dk} \in K.\ dk = eval_{O_{dk}}(b(tr))$ **then**
8: /*Decrypt with the known dk.*/
9: **if** $dec(dk, m) \neq \bot$ **then**
10: $O_{m'} :=$ Construct-shape$(dec(dk, m),$ "dec"$(O, O_q), K, b, tr)$
11: **else**
12: let $l = length(m)$
13: **return** $garbageEnc(O_{ek}, O_r, l)$
14: **else if** $unaryDec(m, b(tr)) \neq \bot$ **then**
15: /*The ciphertext is protocol generated with an non-protocol key.*/
16: $O_{m'} :=$ Construct-shape$(unaryDec(m, b(tr)),$ "$unaryDec$"$(O_q), K, b, tr)$
17: **else**
18: /*The plaintext is hidden.*/
19: $l := lengthofdec(O_q)$
20: $O_{m'} := plaintextof(l, O_q)$
21: **return** $enc(O_{ek}, O_{m'}, O_r)$
22: **case** "$issig(m) \neq \bot$"
23: $O_{vk} =$ Construct-shape($vkof(m)$, "$vkof(O_q)$", $K, b, tr)$
24: **if** $verify(eval_{O_{vk}}(b(tr)), m) = m' \neq \bot$ **then**
25: $O_{m'} :=$ Construct-shape(m', "$verify$"$(O_{vk}, O_q), K, b, tr)$
26: **if** $\exists O.vkof(eval_O(b(tr))) = eval_{O_{vk}}(b(tr))$ **then**
27: /*We know the signing key.*/
28: $O_{sk} = O$
29: **else**
30: /*We know the matching vk.*/
31: $O_{sk} = skofvk(O')$
32: **return** $sig(O_{sk}, O_{m'}, O_r)$
33: **else**
34: let $l = length(m)$
35: **return** $garbageSig(O_{vk}, O_r, l)$
36: **case** "$issk(m) \neq \bot$"
37: **if** $\exists O \in K.\ v = eval_O(b(tr)) \wedge v = vkof(m)$ **then**
38: /*We know the matching vk.*/
39: $O_{sk} := skofvk(O)$
40: **else**
41: /*The key is fresh in the shared knowledge.*/
42: $O_{sk} := sk(O_r)$
43: **return** O_{sk}
44: **case** "$isvk(m) \neq \bot$"
45: **if** $\exists O \in K.\ eval_O(b(tr)) = s \wedge vkofsk(s) = m$ **then**
46: /*We know the matching sk.*/
47: $O_{vk} := vkofsk(O)$
48: **else**
49: /*The key is fresh in the shared knowledge.*/
50: $O_{vk} := vk(O_r)$
51: **return** O_{vk}
52: **case** "$isdk(m) \neq \bot$"
53: **if** $\exists O \in K.\ ek = eval_O(b(tr)) \wedge ek = ekofdk(m)$ **then**
54: /*We know the matching ek.*/
55: $O_{dk} := dkofek(O)$
56: **else**
57: /*The key is fresh in the shared knowledge.*/
58: $O_{dk} := dk(O_r)$
59: **return** O_{dk}
60: **case** "$isek(m) \neq \bot$"
61: **if** $\exists O \in K.\ k = eval_O(b(tr)) \wedge ekofdk(k) = m$ **then**
62: /*We know the matching dk.*/
63: $O_{vk} := ekofdk(O)$
64: **else**
65: /*The key is fresh in the shared knowledge.*/
66: $O_{ek} := ek(O_r)$
67: **return** O_{ek}
68: **case** "$unstring_q(m) \neq \bot$ for $q \in \{0, 1\}$"
69: $O_{m'} :=$ Construct-shape$(unstring_q(m),$ "$unstring_q$"$(O_q), K, b, tr)$
70: **return** $string_q(O_{m'})$
71: **case** "$fst(m) \neq \bot$"
72: $O_{lt} :=$ Construct-shape($fst(m)$, "fst"$(O_q), K, b, tr)$
73: $O_{rg} :=$ Construct-shape($snd(m)$, "snd"$(O_q), K, b, tr)$
74: **return** $pair(O_{lt}, O_{rg})$
75: **case** "$unS(m) \neq \bot$"
76: $O_{m'} :=$ Construct-shape($unS(m)$, "unS"$(O_q), K, b, tr)$
77: **return** $S(O_{m'})$
78: **case** "$l = length(m) \neq \bot$"
79: **return** $garbage(O_r, l)$
80: **case** "$default$"
81: **return** $garbageInvalidLength(O_r)$

$lengthofdec(O_q) := (length(O_q) - a_1 \cdot length(ekof(O_q)) - a_0)/a_2$
for $length(enc(k, m, r)) = a_1 \cdot length(k) + a_2 \cdot length(m) + a_0$

Figure 5: The algorithm Construct-shape.

ek, the attacker learns m and FP-Destruct(K, tr') for the prefix of the trace tr' from the input node's to the root.

Constructing an attacker strategy. Given a shared knowledge K, constructing an attacker strategy is canonical. Using K, we again iterate over all nodes in the trace and apply Construct-shape for each input or output node and add the result together with a `in`- or `out`-tag to the attacker strategy. Upon encountering a control node the in-metadata is stored in the attacker strategy, together with a `control`-tag.

The monitor as a parametric CoSP protocol. We stress that the branching monitor can be constructed as a parametric CoSP protocol. The expressions $eval_O(tr)$ can be evaluated with sequences of the computation nodes that refer to the respective nodes. The references to previous nodes can also be

checked using an *equals*-destructor node. Since CoSP protocols are infinite, loops only need an if-then-else command, and an if-then-else command can encoded into the deterministic polynomial-time algorithm be implemented in the that computes the next identifier (see Definition 13).

Leveraging the sub-algorithms for constructing the branching monitor. Given the algorithms for constructing an attacker strategy, we run a modified symbolic execution, called the *extended symbolic execution*, with these shapes for $\bar{b}(\Pi)$ and check whether the real execution and the extended symbolic execution always take the same branch. The extended symbolic execution is defined Section 5.5.2 and Section 5.5.3.

5.5.2 Extended symbolic model

In order to be able to define the *extended symbolic execution*, we first define the extended symbolic model that contains extended terms with the extended constructors $\mathit{skofvk}/1, \mathit{plaintextof}/2, \mathit{nonceof}/1$ and its natural extension to terms and destructors.

The extended symbolic model $\mathbf{M}'$ of the symbolic model $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ is the natural extension of $\mathbf{M}$ with the constructors $\mathit{skofvk}/1, \mathit{plaintextof}/2, \mathit{nonceof}/1$. The set of *extended constructors* is defined as $\mathbf{C}' := \mathbf{C} \cup \{\mathit{skofvk}/1, \mathit{plaintextof}/2, \mathit{nonceof}/1\}$. The set of extended nonces are defined as $\mathbf{N}' := \mathbf{N} \cup \{\mathit{nonceof}(t) \mid t \in \mathbf{T}\}$.

Extended message type $\mathbf{T}'$. The set of *extended terms* $\mathbf{T}'$ is defined as the set of terms M according to the following grammar:

$$\begin{aligned}
M ::= {} & \mathit{enc}(\mathit{ek}(N), M, N) \mid \mathit{ek}(N) \mid \mathit{dk}(N) \mid \\
& \mathit{sig}(\mathit{sk}(N), M, N) \mid \mathit{vk}(N) \mid \mathit{sk}(N) \mid \mathit{pair}(M, M) \mid P \mid N \mid L \mid \\
& \mathit{garbage}(N, L) \mid \mathit{garbageInvalidLength}(N) \\
& \mathit{garbageEnc}(M, N, L) \mid \mathit{garbageSig}(M, N, L) \\
& \mathit{skofvk}(M) \mid \mathit{plaintextof}(M, M) \mid \mathit{nonceof}(M) \\
P ::= {} & \mathit{emp}() \mid \mathit{string}_0(P) \mid \mathit{string}_1(P) \qquad L ::= O() \mid S(L)
\end{aligned}$$

The nonterminals P, N, and L represent payloads, nonces, and length specifications, respectively. Note that the garbage terms carry an explicit length specification to enable the attacker to send invalid terms of a certain length.

Additional destructor rules. The set $\mathbf{D}'$ of *extended destructors* is the set of partial functions that is defined by the destructor rules for $\mathbf{D}$, with the unary virtual destructor dec' (see Figure 6 and below) with the following additional rules for the extended constructors:

$$\begin{aligned}
\mathit{verify}(\mathit{vk}(t_1), \mathit{sig}(\mathit{skofvk}(t_1), t_2, t_3)) &= t_2 \\
\mathit{issk}(\mathit{skofvk}(t_1)) &= \mathit{skofvk}(t_1) \\
\mathit{issig}(\mathit{sig}(\mathit{skofvk}(t_1), t_2, t_3)) &= \mathit{sig}(\mathit{skofvk}(t_1), t_2, t_3) \\
\mathit{vkof}(\mathit{sig}(\mathit{skofvk}(t_1), t_2, t_3)) &= t_1 \\
\mathit{vkofsk}(\mathit{skofvk}(t_1)) &= t_1 \\
\mathit{length}(\mathit{enc}(t_1, \mathit{plaintextof}(t_2, t_3), t_4)) &= \mathit{length}(t_3)
\end{aligned}$$

In particular, decryption is undefined on extended terms that use $\mathit{plaintextof}(t, t')$ as a plaintext:

$$\mathit{dec}(\mathit{dk}(t_1), \mathit{enc}(\mathit{ek}(t_1), \mathit{plaintextof}(t_2, t_3), t_4)) = \bot$$

We define a unary virtual destructor dec' that looks up and outputs the plaintext as depicted in Figure 6.[13]

5.5.3 Extended symbolic execution

The only difference between the *extended symbolic execution* and the symbolic execution is that the extended symbolic execution operates on $\mathbf{M}'$ and, moreover, expects so-called *extended symbolic operations*, which operate on traces instead of views. Moreover, the extended symbolic model does not work on terms but on symbolic operations. In particular, input messages not immediately parsed as terms but rather only

[13]Formally, dec' is not a destructor, because its result depends on the trace that the protocol has been produced up to the invocation of dec'. However, we treat it like a destructor in the following to simplify presentation.

$dec'(c, tr)$
1: Let tr be the trace produced by the protocol so far.
2: **if** $\exists$ an enc-node ν_i in tr with the result c', with $equals(c, c') \neq \bot$, with the messages k, m, r as arguments, and $\exists$ no ek-node in tr with result k' such that $equals(k, k') \neq \bot$ **then**
3: /**the ciphertext is protocol generated with an adversarially generated key**/
4: **return** m
5: **else if** c is not the result of any enc-node in tr $\wedge$ $\exists$ a node ν_i with the result $dk = x_i(tr)$ **then**
6: /*the attacker generated the ciphertext with a protocol encryption key*/
7: $m := dec(dk, c)$
8: **return** m
9: **else**
10: **return** $\bot$

Figure 6: The shared knowledge algorithm dec' for decryption

assigned as much structure as has already been tested by the protocol via successful destructor application tests, which are executed on the real bitstrings. Another important difference is that the extended symbolic execution is run inside the CoSP protocol $\mathsf{Mon}(\Pi)$. As a consequence,s the extended symbolic operation, if $\mathsf{Mon}(\Pi)$ is run in the computational execution, uses the computational implementations of the constructors and destructors, and, when $\mathsf{Mon}(\Pi)$ is considered in the symbolic model, it uses the symbolic constructors and destructors. Since the alarms in $\mathsf{Mon}(\Pi)$ are trace properties, the assumed computational soundness result implies that $\mathsf{Mon}(\Pi)$, and thereby the extended symbolic execution, in the symbolic model coincides with $\mathsf{Mon}(\Pi)$, and thereby the extended symbolic execution, in the computational execution.

The extended symbolic operations are defined as follows.

Definition 27 (Extended symbolic operation) An *extended symbolic operation O/n (of arity n) on $\mathbf{M}$ for a protocol* Π is a symbolic operation on $\mathbf{M}'$ (as defined in Section 5.5.2) except that the projections are evaluated on traces instead of views via the function $eval_O : \mathsf{Traces}(\Pi) \to \mathbf{T}'$ (where Traces denotes the set of all finite traces in Π):

$$eval_{x_i}(tr) = \begin{cases} m & \text{, if there is an } m \text{ associated to the } i\text{th node in } tr \\ \bot & \text{, otherwise} \end{cases}$$
$$eval_{f(O_1,\dots,O_n)}(tr) = f(eval_{O_1}(tr), \dots, eval_{O_n}(tr)) \text{ for } f \in \mathbf{D}' \text{ with arity } n \qquad \diamond$$

A symbolic execution of a protocol is basically a valid path through the protocol tree. It induces a *view*, which contains the communication with the attacker.

Definition 28 (Extended symbolic execution) Let $\mathbf{M}' = (\mathbf{C}', \mathbf{N}', \mathbf{T}', \mathbf{D}')$ be the extended symbolic model from Section 5.5.2. Let a CoSP protocol I be given. An *extended symbolic view* of the protocol I is a (finite) list L of triples (V_i, ν_i, f_i) with the following conditions:
For the first triple, we have $V_1 = \varepsilon$, ν_1 is the root of I, and f_1 is an empty partial function, mapping node identifiers to terms. For every two consecutive tuples (V, ν, f) and (V', ν', f') in the list, let $\underline{\tilde{\nu}}$ be the nodes referenced by ν and define $\underline{\tilde{t}}$ through $\tilde{t}_j := f(\tilde{\nu}_j)$. We conduct a case distinction on ν.

- **ν is a computation node with constructor, destructor or nonce F.** Let $V' = V$. If $m := eval_F(\underline{\tilde{t}}) \neq \bot$, ν' is the yes-successor of ν in I, and $f' = f(\nu := m)$. If $m = \bot$, then ν' is the no-successor of ν, and $f' = f$.
- **ν is an input node.** If there exists a term $t \in \mathbf{T}$ and an extended symbolic operation O on $\mathbf{M}$ with $eval_O(L') = t$, let ν' be the successor of ν in I, $V' = V :: (\mathtt{in}, (t, O))$, and $f' = f(\nu := t)$, where L' is the prefix of the extended symbolic view L up to the tuple (V, ν, f).
- **ν is an output node.** Let $V' = V :: (\mathtt{out}, \tilde{t}_1)$, ν' is the successor of ν in I, and $f' = f$.
- **ν is a control node with out-metadata l.** Let ν' be the successor of ν with the in-metadata l' (or the lexicographically smallest edge if there is no edge with label l'), $f' = f$, and $V' = V :: (\mathtt{control}, (l, l'))$.

Here, V_{Out} denotes the list of extended terms in V that have been sent at output nodes, i.e., the extended terms t contained in entries of the form $(\mathtt{out}, t)$ in V. Analogously, $V_{Out\text{-}Meta}$ denotes the list of out-metadata in V that has been sent at control nodes.

The set of all symbolic views of I is denoted by $\mathsf{SViews}(I)$. Furthermore, V_{In} denotes the partial list of V that contains only entries of the form $(\mathtt{in}, (*, O))$ or $(\mathtt{control}, (*, l'))$ for some symbolic operation

O and some in-metadata l', where the input term and the out-metadata have been masked with the symbol $*$. The list V_{In} is called *attacker strategy*. We write $[V_{In}]_{\mathsf{SViews}(I)}$ to denote the class of all views $U \in \mathsf{SViews}(I)$ with $U_{In} = V_{In}$. ◇

Symbolic self-monitoring of the branching monitor. Finally, we are able to prove that the branching monitor satisfies symbolic and computational self-monitoring. We start with symbolic self-monitoring, which at its core, follows from the insight that all operations that the branching monitor performs are in the shared knowledge, i.e., are in the symbolic knowledge (of the symbolic adversary). As a consequence, we conclude that a run of the extended symbolic execution induces a full symbolic trace.

Before, we can prove symbolic self-monitoring, we define, as a technical vehicle, a derived view. For an output node ν of a CoSP protocol, we call the output message m *associated with* ν, i.e., m is the message to which the output node references. For an input node or a computation node ν, we call the message m produced at ν *associated with* ν.

Definition 29 (Derived view) Let Π be a bi-protocol and Π' be the corresponding self-monitor. Then, the *derived* left *view* $\mathsf{left}(tr)$ of Π' for a trace tr is iteratively constructed as follows:

$\mathsf{left}(tr) := \varepsilon$ (i.e., the empty list)
for $i = 1$ to length of tr **do**
  if node i is an input node that is associated to a message m **then**
    $\mathsf{left}(tr) := \mathsf{left}(tr) :: (\mathtt{in}, m)$
  if node i is a output node that is labeled as left and is associated to a message m **then**
    $\mathsf{left}(tr) := \mathsf{left}(tr) :: (\mathtt{out}, m)$
  if node i is an control node with input metadata label l and the edge to node $i+1$ in tr is labeled with the metadata label l' **then**
    $\mathsf{left}(tr) := \mathsf{left}(tr) :: (\mathtt{control}, (l, l'))$

The *derived* right *view* $\mathsf{right}(tr)$ is defined analogously with right instead of left for output nodes. For symbolic traces the messages in the view are terms and for computational traces the messages in the view are bitstrings. ◇

Lemma 5 (The branching monitor uses only symbolic operations) *For every extended symbolic operation O that $f_{\mathtt{bad\text{-}branch},\Pi}$ uses in the extended symbolic execution, there is a symbolic operation O' such that for all traces tr we have $eval_O(tr) = eval_{O'}(V(tr))$, where V is the derived view of tr (similar to Definition 29).*

Proof. By construction of Construct-shape, we know that there is only one case in which projections are generated that do not point to output nodes: for the plaintexts m of ciphertexts c for which the attacker knows the decryption key dk. For these cases, there is a symbolic operation $dec(dk, c)$ that outputs the plaintext m.

As a next step, we show that the extended constructors $plaintextof/2$, $skofvk/1$, $nonceof/1$ are only used whenever the attacker could replace them by derivable messages. For $plaintextof/2$, we know by construction of Construct-shape that $plaintextof/2$ is only used on ciphertexts for which either the attacker or the protocol cannot know the plaintext. Hence, the attacker could already generate the ciphertext, and thus there is a symbolic operation that contains an actual message instead of $plaintextof/2$. An analogous argumentation shows that whenever we use $skofvk/1$ or $nonceof/1$, the message was attacker-generated and the attacker new the secret key or the randomness, respectively. □

Lemma 6 (Symbolic self-monitoring of the branching monitor) *Let Π be a bi-protocol from the protocol class P, and $\mathbf{M}$ be the symbolic model from Section 5.1. The parametric CoSP protocol $f_{\mathtt{bad\text{-}branch},\Pi}$ (see Section 5.5.1) satisfies symbolic self-monitoring (see Definition 23).*

Proof. By Lemma 5, we know that for every extended symbolic operation used by $f_{\mathtt{bad\text{-}branch},\Pi}$ there is a symbolic operation. As a consequence, we know that there is a full symbolic trace for each such run of the extended symbolic execution. Moreover, this symbolic trace trace equals the resulting trace from the run of the extended symbolic execution.

Hence, whenever an alarm is raised in $\mathsf{Mon}(\Pi)$, there is an attacker strategy such that (for $b \in \{\mathsf{left}, \mathsf{right}\}$) the symbolic execution of $b(\Pi)$ (which corresponds to the real execution in $\mathsf{Mon}(\Pi)$) and the symbolic execution of $\bar{b}(\Pi)$ branch at some point differently. Thus, there is a symbolic view for which symbolic indistinguishability is violated. □

5.5.4 Computational self-monitoring of the branching monitor

For computational self-monitoring, we reduce the distinguishability of different branchings to trace properties and then apply the computational soundness result for trace properties.

Internalizing the filter into the protocol. As a technical vehicle for the proof, we define a protocol $\mathsf{Mon}(\Pi) - \mathrm{F}_b$ that combines the monitor with the filter F_b, which chooses the b-variant. Then, we show that this protocol $\mathsf{Mon}(\Pi) - \mathrm{F}_b$ is indistinguishable from $b(\Pi)$.

Definition 30 (Filter) The filter machine F_b is constructed by initially choosing the b branch in $\mathsf{Mon}(\Pi)$ and by only forwarding the messages from the output nodes that are labelled with b, thereby ignoring the messages from the output nodes labelled with $\bar{b}$. ⋄

Lemma 7 (Indistinguishability of the self-monitor) *Let* Π *be an efficient bi-protocol and* F_b *be the filter machine from Definition 30. Then, for every* $b \in \{\mathsf{left}, \mathsf{right}\}$ *and every machine* $\mathcal{A}$*, we have*

$$\begin{aligned}&\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)} \mid \mathrm{F}_{\mathsf{left}} \right\rangle \approx_{tic} \left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)} \mid \mathrm{F}_{\mathsf{right}} \right\rangle \\ &\iff \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{left}(\Pi)} \approx_{tic} \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{right}(\Pi)}\end{aligned}$$

Proof. Observe that in both relations, the messages sent to the adversary are the same for $b = \mathsf{left}$ and for $b = \mathsf{right}$. Hence, we only have to consider the running time to prove the two implications. (Indeed, if one of the tic-indistinguishabilities holds, because one of the machines needs a super-polynomial running time, then this tic-indistinguishability states nothing about the messages sent to the adversary after a super-polynomial amount of time. Thus, we could not use this tic-indistinguishability to prove the other one, because the machines machines in the desired tic-indistinguishability could potentially need only polynomial running time, which means that we have to prove a statement about all the messages sent to the adversary.) It suffices to show that the running time of the machines in the one tic-indistinguishability is related to the running time of the machines in the other tic-indistinguishability. Precisely, we show: For each adversary $\mathcal{A}$, for each output a, for all auxiliary information z, and for all polynomials p there is a polynomial q and a negligible function μ such that

$$\begin{aligned}&\Pr[\left\langle\left\langle \mathsf{Exec}_{\mathbf{M},Imp,\mathsf{Mon}(\Pi)}(k) \mid \mathrm{F}_b \right\rangle \mid \mathcal{A}(k,z)\right\rangle \Downarrow_{q(k)} a] \\ &\geq \Pr[\left\langle \mathsf{Exec}_{\mathbf{M},Imp,b(\Pi)}(k) \mid \mathcal{A}(k,z)\right\rangle \Downarrow_{p(k)} a] - \mu(k)\end{aligned} \tag{1}$$

and vice versa, i.e., for all q there is a p such that the formula holds. In other words, the probability that the amount of joint computation steps is polynomially bounded in $\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},b(\Pi)} \mid \mathcal{A} \right\rangle$ equals the probability that the joint computation steps are polynomially bounded in $\left\langle\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)} \mid \mathrm{F}_b \right\rangle \mid \mathcal{A}\right\rangle$.

The direction "$\Longrightarrow$" follows from the fact that $\mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)-\mathrm{F}_b}$ computes internally, due to the extended symbolic execution, both variants and hence has always more computation steps than $\mathsf{Exec}_{\mathbf{M},\mathbf{A},b(\Pi)}$. As a consequence, for all polynomials q, there is a $p \leq q$ such that equation 1 holds.

For the direction "$\Longleftarrow$", we have to show that the computation steps of the interaction $\left\langle\left\langle \mathsf{Exec}_{\mathbf{M},Imp,\mathsf{Mon}(\Pi)}(k) \mid \mathrm{F}_b \right\rangle \mid \mathcal{A}(k,z)\right\rangle$ are polynomially bounded as well. This statement follows from the following three other statements: (i) in $f_{\texttt{bad-knowledge}}$ the running time of the fixpoint computation (FP-Destruct) is bounded polynomially in the sum of the length of the prefix trace tr_j and the security parameter; (ii) $\mathsf{Mon}(\Pi)$ is efficient in the sense of Definition 13; (iii) the extended symbolic execution of $\bar{b}(\Pi)$ takes super-polynomially many computation steps, St against $\bar{b}(\Pi)$ only needs poly many steps.

(i) follows from the construction of FP-Destruct, since we only destruct messages. (ii) also follows from the construction $\mathsf{Mon}(\Pi)$ and from $f_{\texttt{bad-knowledge}}$. It, hence, remains to show (iii)

Since $\mathsf{Mon}(\Pi)$ internally also computes $\bar{b}(\Pi)$, we have to exclude that the extended symbolic execution of $\bar{b}(\Pi)$ takes super-polynomially many computation steps. Observe that uniformity-enforcing, symbolic indistinguishability and computational soundness for trace properties imply that the distribution of traces induced by $\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{left}(\Pi_i)} \mid \mathcal{A} \right\rangle$ is computationally indistinguishable from the distribution of node traces induced by $\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{right}(\Pi_i)} \mid \mathcal{A} \right\rangle$; otherwise, we can construct a single protocol that executes both $\mathsf{left}(\Pi_i)$ and $\mathsf{right}(\Pi_i)$, checks in the second run whether all inputs satisfy the same destructor tests as the inputs of the first run, and raises an alarm if a branching was different. This protocol breaks the computational soundness for trace properties (which holds by Theorem 11).

We know that the length of the trace is polynomially bounded. Next, we have to show that also the call of the algorithms does not cause a super-polynomial computation. By Definition 10 all implementations are polynomial-time computable, and by the definition of a length destructor and Implementation Conditions 4 and 32 the length of all bitstrings is linear in the length of the input. The length of the input, in turn,

is polynomially bounded in the sum of the length of tr_j and the security parameter. Hence, every call to an implementation algorithm only causes a polynomially bounded number of computation steps, in the sum of the security parameter and the length of the trace so far tr_j (where tr_j is the path from the computation node ν_j to the root). □

Definition 31 ($\mathsf{Mon}(\Pi) - \mathrm{F}_b$) For a protocol Π, we define the protocol $\mathsf{Mon}(\Pi) - \mathrm{F}_b$ as the unrolled variant of the protocol in which the initial node in $\mathsf{Mon}(\Pi)$ is removed and only the b-branch is taken and each the $\bar{b}$-labelled output node that points to a node ν is replaced by a so-called *virtual output* node: a computation node of the *pair* constructor that points twice to ν. In $\mathsf{Mon}(\Pi) - \mathrm{F}_b$ the monitor $\mathsf{Mon}(\Pi)$ treats virtual output nodes as it treated output nodes. ◇

Corollary 8 ($\mathsf{Mon}(\Pi) - \mathrm{F}_b$ equivalence) *For all CoSP protocols Π and all ppt adversaries $\mathcal{A}$, we have*

$$\begin{aligned} &\mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)-\mathrm{F}_{\mathsf{left}}} \approx_{tic} \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)-\mathrm{F}_{\mathsf{right}}} \\ \iff &\mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{left}(\Pi)} \approx_{tic} \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{right}(\Pi)} \end{aligned}$$

Proof. This immediately follows from the definition of $\mathsf{Mon}(\Pi) - \mathrm{F}_b$ and from Lemma 7. □

Lemma 9 (Computational self-monitoring of the branching monitor) *The parametric CoSP protocol $f_{\texttt{bad-branch},\Pi}$ (see Section 5.5.1) satisfies computational self-monitoring (see Definition 23).*

Proof. We construct an *extended branching monitor* $\mathsf{Mon}'(\Pi)$. Where the branching monitor runs the extended symbolic execution for $\bar{b}(\Pi)$, the extended branching monitor Mon' completely runs $\bar{b}(\Pi)$ once again and additionally implements input guards in the second run (see below) and checks whether the last computation node, i.e., the node before ν, branches the same in both runs if the input guards succeed executes Π_i twice and compares which branch was taken at the node ν.

1: in the first run, apply Construct-shape at each input node and store each resulting shape
2: in the second run, if a branching is different from the first run, set a flag *fail* to *true*
3: in the second run, we additionally apply Construct-shape at each input node and check whether the resulting shape coincides with the shape from same input node in the first run. If the check fails, set the flag *fail* to *true*, as well.
4: **if** the flag *fail* equals *true* **then**
5: $\quad\Pi^\nu$ halts in a state `stop`

We consider the following trace property p_ν: if `stop` is not reached (in Π^ν), then ν is reached in both runs and use the same branch in both runs.

We define a filter F_b by a machine (which typically interacts with $\mathsf{Mon}(\Pi)$) that upon the initial query of $\mathsf{Mon}(\Pi)$ chooses the branch b and then forwards all messages to the adversary or the simulator, i.e., over the network interface. The extended filter EF_b extends the filter F_b by completely hiding the second run from $\mathcal{A}$ by replaying the same messages once again, also from the distinguisher, i.e., does not output the results of the second run.

Claim 1. *Let Π be an arbitrary bi-protocol in P. If and only if Π is indistinguishable, the extended branching monitor is indistinguishable, i.e.,*

$$\begin{aligned} &\mathsf{Exec}_{\mathbf{M},\mathbf{A},b(\Pi)} \approx_{tic} \mathsf{Exec}_{\mathbf{M},\mathbf{A},\bar{b}(\Pi)} \\ \iff &\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi)} \mid \mathrm{EF}_b \right\rangle \approx_{tic} \left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi)} \mid \mathrm{EF}_{\bar{b}} \right\rangle \end{aligned}$$

Proof of Claim 1. This claim directly follows from Lemma 7 since no information is sent to the adversary $\mathcal{A}$ after the first run. ◇

As a next step, we consider a modification EF'_b of the extended filter EF_b that sends `firstRunDone` after the first run (with b) over the network interface net. Then, it waits for the string `beginSecondRun` and performs real a second run with $\bar{b}$, in contrast to EF_b not the attacker messages from the first run. Moreover, let the rewinding filter $\mathrm{RF}(\mathcal{A})$ be a machine that internally runs the adversary machine $\mathcal{A}$ and resets $\mathcal{A}$ after receiving the string `firstRunDone`. Then, it sends `beginSecondRun` over the left network interface net. We plug these machines together as follows: $\left\langle \left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi)} \mid \mathrm{EF}'_b \right\rangle \mid \mathrm{RF}(\mathcal{A}) \right\rangle$, i.e., the

left network interface of the rewinding filter is connected to the right network interface of EF'_b. Moreover, the output interface of $\mathcal{A}$ is connected to the sub-output interface of RF, and the output interface of RF is connected to the output interface of EF'_b. EF'_b outputs the guess of the adversary from the first run.

Claim 2. *The extended branching monitor $\mathsf{Mon}'(\Pi)$ is indistinguishable against the extended filter if and only if $\mathsf{Mon}'(\Pi)$ is indistinguishable against the rewinding filter, i.e., for all probabilistic polynomial-time machines $\mathcal{A}$, we have*

$$\begin{aligned}&\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi)} \mid \mathrm{EF}_b \right\rangle \approx^{\mathcal{A}}_{tic} \left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi)} \mid \mathrm{EF}_{\bar{b}} \right\rangle \\ \iff &\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi)} \mid \mathrm{EF}'_b \right\rangle \approx^{\mathrm{RF}(\mathcal{A})}_{tic} \left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi)} \mid \mathrm{EF}'_{\bar{b}} \right\rangle\end{aligned}$$

Proof of Claim 2. The statement directly follows from the construction of EF'_b against $\mathrm{RF}(\mathcal{A})$: the guess of EF'_b has exactly the same distribution as the guess of EF_b against $\mathcal{A}$. $\diamond$

Claim 3. *If Π_i is distinguishable, Π_{i-1} is indistinguishable, the last node ν in Π_i is a control node, then both extended branching monitors computationally raise an alarm, i.e.,*

$$\begin{aligned}&\mathsf{Exec}_{\mathbf{M},\mathbf{A},b(\Pi_i)} \not\approx_{tic} \mathsf{Exec}_{\mathbf{M},\mathbf{A},\bar{b}(\Pi_i)} \text{ and} \\ &\mathsf{Exec}_{\mathbf{M},\mathbf{A},b(\Pi_{i-1})} \approx_{tic} \mathsf{Exec}_{\mathbf{M},\mathbf{A},\bar{b}(\Pi_{i-1})} \text{ and } \nu \text{ is a control node} \\ &\implies \Pr\left[\left\langle\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_b\right\rangle \mid \mathrm{RF}(\mathcal{A})\right\rangle : p_\nu \text{ occurs}\right] \text{ is non-negligible}\end{aligned}$$

Moreover, we have

$$\begin{aligned}&\Pr\left[\left\langle\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_b\right\rangle \mid \mathrm{RF}(\mathcal{A})\right\rangle : p_\nu \text{ occurs}\right] \\ &= \Pr\left[\left\langle\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_{\bar{b}}\right\rangle \mid \mathrm{RF}(\mathcal{A})\right\rangle : p_\nu \text{ occurs}\right]\end{aligned}$$

Proof of Claim 3. Since Π_{i-1} is indistinguishable and by Claim 2, we know that the following holds

$$\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi_{i-1})} \mid \mathrm{EF}'_b \right\rangle \approx^{\mathrm{RF}(\mathcal{A})}_{tic} \left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi_{i-1})} \mid \mathrm{EF}'_{\bar{b}} \right\rangle$$

By Claim 1 and Claim 2, we know that the extended branching monitor for Π_i is distinguishable against the rewinding filter, i.e.,

$$\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_b \right\rangle \not\approx^{\mathrm{RF}(\mathcal{A})}_{tic} \left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_{\bar{b}} \right\rangle$$

By the definition of computational challenger (see Definition 11) and since by assumption the last node ν in Π_i is a control node we know that the only information that the distinguisher receives against Π_i that he does not receive against Π_{i-1} is the in-metadata. As a consequence, we know that the computation node ν' directly before ν branches differently in the execution of $b(\Pi_i)$ and $\bar{b}(\Pi_i)$ against $\mathcal{A}$, thus also in $\mathsf{Mon}'(\Pi_i)$ against EF'_b and $\mathrm{RF}(\mathcal{A})$. Thus, for the two runs in $\mathsf{Mon}'(\Pi_i)$ against the rewinding filter $\mathrm{RF}_b(\mathcal{A})$, the attacker produces with non-negligible probability, traces that pass all entry guards and cause the computation node ν' (just before ν) to branch differently. Since the rewinding filter resets the attacker, also the rewinding filter $\mathrm{RF}_b(\mathcal{A})$ finds such a trace with non-negligible probability, thus the statement follows:

$$\begin{aligned}&\Pr\left[\left\langle\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_b\right\rangle \mid \mathrm{RF}(\mathcal{A})\right\rangle : p_\nu \text{ occurs}\right] \\ &= \Pr\left[\left\langle\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_{\bar{b}}\right\rangle \mid \mathrm{RF}(\mathcal{A})\right\rangle : p_\nu \text{ occurs}\right] \text{ and} \\ &\Pr\left[\left\langle\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_{\bar{b}}\right\rangle \mid \mathrm{RF}(\mathcal{A})\right\rangle : p_\nu \text{ occurs}\right] \text{ is non-negligible}\end{aligned}$$

$\diamond$

Claim 4. *Let Π be an arbitrary bi-protocol. If the branching monitor $\mathsf{Mon}(\Pi)$ symbolically does not raise an alarm, then the extended branching monitor $\mathsf{Mon}'(\Pi)$ symbolically does not raise an alarm.*

Proof of Claim 4. We first show that in the extended symbolic execution the branching monitor $\mathsf{Mon}(\Pi)$ never produces an extended term *plaintextof*$(_,_)$, i.e., a term that the monitor does not know for the extended symbolic execution. By the construction of Construct-shape (see Section 5.5.1) we know that

such an extended term is only inside a ciphertext for which the protocol does not know the key and the ciphertext is attacker-generated. However, the protocol will never be able to decrypt such an encryption, i.e., such an extended term $\mathit{plaintextof}(_,_)$ is never the result of an evaluation of a computation node in the extended symbolic execution.

If extended terms $\mathit{plaintextof}(_,_)$ are never produced by any evaluation in the extended symbolic execution, the shapes always produce the same branching in the extended symbolic execution as any term for which the input guards succeed, which concludes the statement. $\diamond$

The contraposition of Claim 3 is the following statement:

$$\begin{aligned}&\mathsf{Exec}_{\mathbf{M},\mathbf{A},b(\Pi_{i-1})} \approx_{tic} \mathsf{Exec}_{\mathbf{M},\mathbf{A},\bar{b}(\Pi_{i-1})} \text{ and } \nu \text{ is a control node}\\ &\wedge \Pr\left[\langle\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_b\rangle \mid \mathrm{RF}(\mathcal{A})\rangle : p_\nu \text{ occurs}\right] \text{ is negligible}\\ &\implies \mathsf{Exec}_{\mathbf{M},\mathbf{A},b(\Pi_i)} \approx_{tic} \mathsf{Exec}_{\mathbf{M},\mathbf{A},\bar{b}(\Pi_i)}\end{aligned} \quad \square$$

Furthermore, the following statement holds by assumption:

$$\mathsf{Exec}_{\mathbf{M},\mathbf{A},b(\Pi_{i-1})} \approx_{tic} \mathsf{Exec}_{\mathbf{M},\mathbf{A},\bar{b}(\Pi_{i-1})} \text{ and } \nu \text{ is a control node}$$

By Claim 4 and by the assumption that no branching alarm is raised (with more than negligible probability), we know that

$$\Pr\left[\langle\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_b\rangle \mid \mathrm{RF}(\mathcal{A})\rangle : p_\nu \text{ occurs}\right] \text{ is negligible}$$

Hence, we conclude

$$\mathsf{Exec}_{\mathbf{M},\mathbf{A},b(\Pi_i)} \approx_{tic} \mathsf{Exec}_{\mathbf{M},\mathbf{A},\bar{b}(\Pi_i)}$$

5.6 The knowledge monitor

The knowledge monitor $f_{\texttt{bad-knowledge},\Pi}(b,tr)$ for an output node ν starts like $f_{\texttt{bad-branch},\Pi}(b,tr)$ by reconstructing a (symbolic) attacker strategy and simulating a symbolic execution of $\bar{b}(\Pi)$. However, instead of testing the branching behavior of $\bar{b}(\Pi)$, the knowledge monitor $f_{\texttt{bad-knowledge},\Pi}(b,tr)$ characterizes the message m that is output in $b(\Pi)$ at the output node ν in question, and then $f_{\texttt{bad-knowledge},\Pi}(b,tr)$ compares m to the message that would be output in $\bar{b}(\Pi)$. This characterization must honor that ciphertexts generated by the protocol are indistinguishable if the corresponding decryption key has not been revealed to the attacker so far. If a difference in the output of $b(\Pi)$ and $\bar{b}(\Pi)$ is detected, the event `bad-knowledge` is raised.

Symbolic self-monitoring for the knowledge monitor $f_{\texttt{bad-knowledge},\Pi}(b,tr)$ follows by the same arguments as for $f_{\texttt{bad-branch},\Pi}(b,tr)$. We show computational self-monitoring by first applying the PROG-KDM property to prove that the computational execution of $b(\Pi)$ is indistinguishable from a *faking* setting: in the faking setting, all ciphertexts generated by the protocol do not carry any information about their plaintexts (as long as the corresponding decryption key has not been leaked). The same holds analogously for $\bar{b}(\Pi)$. We then consider all remaining real messages, i.e., all messages except ciphertexts generated by the protocol with unleaked decryption keys. We then show that in the faking setting, $f_{\texttt{bad-knowledge},\Pi}(b,tr)$ is able to characterize all information that is information theoretically contained in a message. We conclude by showing that with this characterization, the knowledge monitor raises the event `bad-knowledge` whenever the bi-protocol Π is distinguishable.

Section 5.6.1 presents the construction of the knowledge monitor. The proof of symbolic self-monitoring goes along the lines of the proof of self-monitoring of the branching monitor (see Section 5.6.2). For the proof of computational self-monitoring, we use several technical vehicles. We prove that the construction of a so-called *faking simulator* in the computational soundness proof w.r.t. trace properties from previous work [9] can be reused. This faking simulator simulates a real computational execution but does not use sensitive information at all, i.e., for all ciphertext of which the decryption key has not been leaked yet the encryption operation does not use the actual plaintext. To this end, we first recall this faking simulator in Section 5.6.3. Then, in Section 5.6.5, we prove that the indistinguishability of the faking simulator and the real computational execution does carries over to the case with equivalence properties. Thereafter, in Section 5.6.6, we prove that for each symbolic operation there is exactly one bitstring in one run. As a final vehicle, we resolve, in Section 5.6.7, the natural double induction, over the length of the protocol and the structural size of the message to be sent by introducing so-called *unrolled variants*, which before sending a message first send all visible sub-messages this message to the adversary. Finally, in Section 5.6.8, we plug all these tools together and prove computational self-monitoring of the knowledge monitor.

$f_{\texttt{bad-knowledge},\Pi}(b, tr)$
1: $V_b :=$ Construct-attacker-strategy(b, tr)
2: $V_{\bar{b}} :=$ Construct-attacker-strategy$(\bar{b}, tr)$
3: **if** $V_b = V_{\bar{b}}$ **then**
4: **return** `ok`
5: **else**
6: go to state `bad-knowledge`

Figure 7: Construction of the knowledge monitor

5.6.1 Construction of the knowledge monitor

The knowledge monitor $f_{\texttt{bad-knowledge},\Pi}$ (see Figure 7), like the branching monitor $f_{\texttt{bad-branch},\Pi}$ (see Section 5.5.1), first constructs an attacker strategy, using Construct-attacker-strategy(see Figure 3). With the this attacker strategy, the knowledge monitor $f_{\texttt{bad-knowledge},\Pi}$ obtains for the output node for $b(\Pi)$ and for $\bar{b}(\Pi)$ a shape such that all information that is visible from the shared knowledge is contained in these shape, i.e., these shapes have maximal information. The knowledge monitor then simply checks whether these shapes are equal or not. If the shapes are not equal, an alarm is raised.

5.6.2 Symbolic self-monitoring of the knowledge monitor

In this section, we show the symbolic self-monitoring of the knowledge monitor.

Definition 32 (Shared knowledge) A *shared test* is a symbolic operation for a trace tr (for its derived view V see Definition 29) that either

- does not use any nodes labelled with nonces , or
- with attacker nonces $\mathbf{N}_E$ that were used in protocol-constructed terms in tr.

The *shared knowledge function* is the knowledge function (see Definition 7) $K_{\mathsf{sh}V} : \mathsf{SO} \to \{\top, \bot\}$ that is defined on all shared tests and undefined on all other symbolic operations.

We say that a *symbolic operation is in the shared knowledge* if it is a shared test. ◇

Lemma 10 (Symbolic self-monitoring of the knowledge monitor) *Let Π be a bi-protocol. Let $i \in \mathbb{N}$. Let Π'_i be the self-monitor for Π_i. For all $i \in \mathbb{N}$ the following holds. If there is an attacker strategy such that in Π'_i the event* `bad-knowledge` *occurs but in Π'_{i-1} the event* `bad` *does not occur and Π_{i-1} is symbolically indistinguishable, then Π_i is symbolically distinguishable because of knowledge.*

Proof. We show this statement by induction over i. For $i = 1$, either the last node of Π_1 is a control node or an output node. If the last node is a control node, the statement follows. If the last node is a output node, then observe that $f_{\texttt{bad-knowledge},\Pi}$ only performs tests that are in the shared knowledge (see Lemma 5). Hence, there is an attacker strategy that distinguishes the pair of views obtained by the symbolic execution of Π_1 and Π_1 is symbolically distinguishable. Since the last node was an output node and Π_0 is indistinguishable because it does not output anything to the attacker, Π_1 is symbolically distinguishable because of knowledge.

For $i > 1$ we know that in Π'_i for all but the last control node no alarm is raised, i.e., `bad` does not occur for any attacker strategy. Hence, in the tests of before the last control node `bad-knowledge` occurred. It remains to show that then Π_i is distinguishable because of knowledge. Observe that $f_{\texttt{bad-knowledge},\Pi}$ only performs tests that are in the shared knowledge (see Lemma 5). Hence, there is an attacker strategy that distinguishes the pair of views obtained by the symbolic execution of Π_i and Π_i is symbolically distinguishable. Since the last node was an output node and Π_{i-1} is indistinguishable by assumption, Π_1 is symbolically distinguishable because of knowledge. □

5.6.3 The faking simulator Sim_f

We show that we can reuse the simulator of the computational soundness result of Backes, Malik, and Unruh [9]. In particular, we use the hybrid execution in which all messages are faked. Technically, however, the previous computational soundness result for trace properties needs indistinguishability trace properties and computational soundness for equivalence properties needs the communication is indistinguishable for the attacker.

Hybrid execution for trace properties. A successful way of proving computational soundness (for trace properties) in the literature is the construction of a simulator that translates bitstrings to terms

and vice versa. The simulator interacts with a modified symbolic execution, called the hybrid execution, on one side and the computational attacker on the other side. Whenever the simulator receives a term from the hybrid execution, the simulator constructs a corresponding bitstring and sends it to the attacker. Whenever the simulator receives a bitstring from the attacker, the simulator parses the bitstring and assigns a term to it, which it sends to the hybrid execution. This simulator thereby assigns a symbolic attacker strategy for a run agains the computational attacker.

This simulator runs against a modified symbolic execution, called a hybrid execution challenger, that lets the simulator decide the attacker strategy. In the work of Backes, Malik, and Unruh [9], the hybrid execution challenger is a further modification in that it enables a lazy evaluation that works as follows: first, the hybrid execution challenger accepts incomplete terms with variables inside as attacker-terms for input nodes; second, whenever a term is evaluated the hybrid execution asks the simulator, so-called eval-questions, to evaluate the term, potentially using an assignment for the variables. At the end of the execution, the simulator has to send an assignment of all variables to terms that is consistent with the responses of the simulator to the evaluation queries.

Definition 33 (Hybrid challenger $\mathsf{TrH\text{-}Exec}$ (trace properties)) Let Π be a CoSP protocol. We define an interactive machine $\mathsf{TrH\text{-}Exec}_{\mathbf{M},\Pi}(k)$ run on input k. It is called the hybrid protocol machine associated with Π. It internally maintains and finally outputs a (finite) lists of tuples (S_i, ν_i, f_i), called the *full hybrid trace*, and runs a symbolic simulation of Π as follows:

Initially $S_1 := S := \varepsilon$, $\nu_1 := \nu$ is the root of Π, and $f_1 := f$ is a totally undefined partial function mapping node identifiers to $\mathbf{T}$. For $i = 2, 3, \ldots$ do the following (recall that netis the network interface):

1. Let $\underline{\tilde{\nu}}$ be the node identifiers in the label of ν. Define $\underline{\tilde{t}}$ through $\tilde{t}_j := f(\tilde{\nu}_j)$.
2. Proceed depending on the type of ν:
   - **If ν is a computation node with constructor, destructor, or nonce F**, then send the question $F(\underline{\tilde{t}})$ over net and wait for a response m. If $m = \mathtt{yes}$, let ν' be the *yes*-successor of ν and let $f' := f(\nu := F(\underline{\tilde{t}}))$. If $m = \mathtt{no}$, let ν' be the *no*-successor of ν and let $f' := f$. Set $f := f'$ and $\nu := \nu'$.
   - **If ν is an output node**, send $\tilde{t}_1$ over net. Let ν' be the unique successor of ν and let $S' := S \cup \{\tilde{t}_1\}$. Set $\nu := \nu'$ and $S := S'$.
   - **If ν is an input node**, wait on net to receive $m \in \mathbf{T}$ from *Sim*. Let $f' := f(\nu := m)$, and let ν' be the unique successor of ν. Set $f := f'$ and $\nu := \nu'$.
   - **If ν is a control node labeled with out-metadata l**, send l over net, and wait to receive a bitstring l' from net. Let ν' be the successor of ν along the edge labeled l' (or the edge with the lexicographically smallest label if there is no edge with label l'). Set $\nu := \nu'$.
3. Send (info, ν, t) over net. When receiving an answer (*proceed*) from net, continue.
4. If over net has output a final assignment from variables to symbolic operations is sent, check whether the final assignment is consistent with all the responses to the eval-questions. If the check fails, abort with `inconsistency`. Otherwise, hand the control back over net.
5. Otherwise, if over net a final assignment is not sent, let $(S_i, \nu_i, f_i) := (S, \nu, f)$. ◇

Indistinguishability and Dolev-Yaoness of the faking simulator. On one hand, in order to ensure that this translation from a transcript of bitstrings to a symbolic attacker strategy is accurate, the simulator needs to produce a symbolic attacker strategy that matches the interaction of the computational attacker with the computational execution challenger. In particular, the trace of protocol states that the hybrid execution produces has to be computationally indistinguishable from the one that the computational execution produces. In CoSP, this property is called *indistinguishability* of a simulator.

On the other hand, in spite of accurately modeling the computational execution, the produced symbolic attacker strategy has to obey the symbolic rules, i.e., w.r.t. the symbolic model it has to be a valid symbolic attacker strategy. In CoSP, this property is called *Dolev-Yaoness* of a simulator.

The Dolev-Yaoness is proven by showing that the computational execution challenger against the attacker $\mathcal{A}$ produces indistinguishable traces from the hybrid execution challenger against a faking simulator Sim_f[14] that fakes all ciphertexts and forwards all messages back and forth from the internally simulated attacker $\mathcal{A}$.

[14] In the paper by Backes, Malik, and Unruh [9], this simulator is called Sim_7.

For the proof of the computational self-monitoring property, we use this faking simulator Sim_f. We review the construction of Sim_f, but for the sake of brevity we only describe in detail how Sim_f handles ciphertexts. The full description can be found in the work of Backes, Malik, and Unruh [9].

For translating bitstrings to terms and vice versa, the simulator Sim_f has two efficiently computable stateful functions τ and β: τ assigns to each bitstring a term and β assigns to each term a bitstring. The simulator is constructed such that τ and β are partially inverse for each other, i.e., for all terms t that are sent by the hybrid execution (except for the randomness) the equation $\tau(\beta(t)) = t$ and for all bitstrings m the equation $\beta(\tau(m)) = m$ holds.

The honest simulator *Sim*. For the sake of illustration, we present the definition of the encryption-related cases of the honest simulator Sim, and subsequently briefly describe how the faking simulator Sim_f differs from Sim.

Let $\mathcal{R}$ be the set of randomness terms that has been sent by the hybrid execution before the respective invokation of β or τ. In the following definition the first case that applies is taken.

The definition of τ for the encryption-related bitstrings is as follows:

1. $\tau(r) := N$ if $r = r_N$ for some $N \in \mathcal{N} \setminus \mathcal{R}$.
2. $\tau(r) := N^r$ if r is of type nonce.
3. $\tau(c) := enc(ek(M), t, N)$ if c has earlier been output by $\beta(enc(ek(M), t, N))$ for some $M \in \mathbf{N}$, $N \in \mathcal{R}$.
4. $\tau(c) := enc(ek(N), \tau(m), N^c)$ if c is of type ciphertext and $\tau(A_{ekof}(c)) = ek(N)$ for some $N \in \mathcal{R}$ and $m := A_{dec}(A_{dk}(r_N), c) \neq \bot$.
5. $\tau(c) := garbageEnc(ek(N), N^c)$ if c is of type ciphertext and $\tau(A_{ekof}(c)) = ek(N)$ for some $N \in \mathcal{R}$ but $A_{dec}(A_{dk}(r_N), c) = \bot$.
6. $\tau(c) := x^c$ if c is of type ciphertext but $\tau(A_{ekof}(c)) \neq ek(N)$ for all $N \in \mathcal{R}$.
7. $\tau(e) := ek(N)$ if e has earlier been output by $\beta(ek(N))$ for some $N \in \mathcal{R}$.
8. $\tau(e) := ek(N^e)$ if e is of type encryption key.
9. $\tau(k) := dk(N)$ if k has earlier been output by $\beta(dk(N))$ for some $N \in \mathcal{R}$.
10. $\tau(k) := dk(N^e)$ if k is of type decryption key and $e := A_{ekofdk}(k) \neq \bot$.

The definition of β for the encryption-related terms is as follows:

1. $\beta(N) := r_N$ if $N \in \mathcal{N}$.
2. $\beta(N^m) := m$.
3. $\beta(enc(ek(t_1), t_2, M)) := Imp_{enc}(\beta(ek(t_1)), \beta(t_2), r_M)$ if $M \in \mathcal{R}$.
4. $\beta(enc(ek(t_1), t, N^m)) := m$.
5. $\beta(x^c) := c$.
6. $\beta(ek(N)) := Imp_{ek}(r_N)$ if $N \in \mathcal{R}$.
7. $\beta(ek(N^m)) := m$.
8. $\beta(dk(N)) := A_{dk}(r_N)$ if $N \in \mathcal{R}$.[15]
9. $\beta(dk(N^m)) := A^{-1}_{ekofdk}(m)$. (Note that due to Implementation Condition 30, there is at most one value $A^{-1}_{ekofdk}(m)$. And see below for a discussion of the polynomial-time computability of $A^{-1}_{ekofdk}(m)$.)

By keeping a record of all decryption keys d, the simulator can efficiently compute $A^{-1}_{ekofdk}(e)$ in Case 9.

Due to the limited compositionality of tic-indistinguishability (see Definition 15), we need to ensure that even machines that are connected to the network and execution interface cannot distinguish the honest simulator Sim from the faking simulator Sim_f. We say that a term t *is used in the randomness of the term* t' if there are terms t_1, t_2 such that $t' \in \{ek(t), dk(t), vk(t), sk(t)\} \cup \{sig(t_1, t_2, t) \mid t_1, t_2 \text{ are terms}\} \cup \{enc(t_1, t_2, t) \mid t_1, t_2 \text{ are terms}\}$. Sim checks whether each term t that is used in randomness position of a term t' is only used inside t'. If t is used somewhere else or sent in plain, Sim aborts. If Sim interacts with a hybrid execution, these checks will always succeed; in these cases Sim behaves as the simulator in the work of Backes, Malik, and Unruh [9].

Modifications for the faking simulator. In the faking simulator, the ciphertext simulator CS is used (see Implementation Conditions 25) instead of the encryption algorithm. The ciphertext simulator CS provides commands to the adversary for retrieving an encryption key of a ciphertext ($\texttt{getek}_{\texttt{ch}}$), the decryption of a ciphertext ($\texttt{getdk}_{\texttt{ch}}$), to produce a fake encryption ($\texttt{fakeenc}_{\texttt{ch}}$), and for other commands. Sim_f maintains for each $N \in \mathbf{N}_P$ an instance CS_N of the ciphertext simulator. Whenever the encryption scheme algorithm is honestly applied in Sim, Sim_f uses CS_N as follows: for $\beta(ek(N))$, Sim_f sends a

[15] Technically, before returning the value $\beta(dk(N))$ invokes $\beta(ek(N))$ and discards it return value. (This is to guarantee that $A_{ek}(N)$ can only be guessed when $\beta(ek(N))$ was invoked.) We refer to [9] for a detailed explanation.

$\mathtt{getek_{ch}}$-query to CS_N; for $\beta(enc(ek(N), t_2, M))$ with $N, M \in \mathcal{R}$, Sim_f either sends a $\mathtt{fakeenc_{ch}}$-query (if there was no $\mathtt{getdk_{ch}}$-query to CS_N yet) or a *enc*-query to CS_N (if there already was a $\mathtt{getdk_{ch}}$-query to CS_N; for $\beta(dk(N))$, Sim_f sends a $\mathtt{getdk_{ch}}$-query to CS_N.

In addition, the faking simulator internally runs an instance of the random oracle $\mathcal{O}$ and gives all ciphertext simulators CS_N access to $\mathcal{O}$ and to get the list of queries and programming capabilities for $\mathcal{O}$. To the attacker $\mathcal{A}$ the simulator Sim_f grants access to $\mathcal{O}$.

Beside these modifications, Sim_f performs book keeping for mapping honestly generated ciphertexts to their respective plaintexts, and Sim_f queries a signing oracle instead of using the signing algorithms. A detailed description of the modifications can be found in [9].

The hybrid execution for equivalence properties.

Definition 34 (Hybrid challenger H-Exec (equivalence)) Let Π be a CoSP protocol, and let *Sim* be a simulator. We define an interactive machine $\mathsf{H\text{-}Exec}_{\mathbf{M},\Pi}(k)$ run on input k. It is called the hybrid protocol machine associated with Π. It internally maintains on (finite) lists of tuples (S_i, ν_i, f_i), called the *full hybrid trace*, and runs a symbolic simulation of Π as follows:

Initially $S_1 := S := \varepsilon$, $\nu_1 := \nu$ is the root of Π, and $f_1 := f$ is a totally undefined partial function mapping node identifiers to $\mathbf{T}$. For $i = 2, 3, \ldots$ do the following (recall that netis the network interface):

1. Let $\underline{\tilde{\nu}}$ be the node identifiers in the label of ν. Define $\underline{\tilde{t}}$ through $\tilde{t}_j := f(\tilde{\nu}_j)$.

2. Proceed depending on the type of ν:

   - **ν is a computation node with constructor, destructor or nonce F.** Let $V' = V$. Send the question $F(\underline{\tilde{t}})$ over net and wait for a response m. If $m = \mathtt{yes}$, ν' is the $\mathtt{yes}$-successor of ν in I, and $f' = f(\nu := F(\underline{\tilde{t}}))$. If $m = \mathtt{no}$, then ν' is the $\mathtt{no}$-successor of ν, and $f' = f$.
   - **ν is an input node.** Wait over netfor a symbolic operation O. Let $t := xeval_O(V_{Out})$, where $xeval_O(V_{Out})$ is defined just like $eval_O(V_{Out})$ except that nodes of type variables are evaluated to the variable with the name that is annotated in the node. Let $f' := f(\nu := t)$, and let ν' be the unique successor of ν. Set $f := f'$, $V' = V :: (\mathtt{in}, (t, O))$, and $\nu := \nu'$.
   - **ν is an output node.** Send $\tilde{t}_1$ over net. Let ν' be the unique successor of ν and let $S' := S \cup \{\tilde{t}_1\}$. Set $\nu := \nu'$, $V' = V :: (\mathtt{out}, \tilde{t}_1)$, and $S := S'$.
   - **ν is a control node with out-metadata l.** Let ν' be the successor of ν with the in-metadata l' (or the edge with the lexicographically smallest label if there is no edge with label l'), $f' = f$, and $V' = V :: (\mathtt{control}, (l, l'))$.

3. Send (info, ν, t) over net. When receiving an answer $(\mathit{proceed})$ over net, continue.

4. If over net has output a final assignment from variables to symbolic operations is sent, check whether the final assignment is consistent with all the responses to the eval-questions. If the check fails, abort with `inconsistency`. Otherwise, hand the control back over net.

5. Otherwise, if over net a final assignment is not sent, let $(S_i, \nu_i, f_i) := (S, \nu, f)$.

Here, V_{Out} denotes the list of terms in V that have been sent at output nodes, i.e., the terms t contained in entries of the form $(\mathtt{out}, t)$ in V. Analogously, $V_{Out\text{-}Meta}$ denotes the list of out-metadata in V that has been sent at control nodes.

Furthermore, V_{In} denotes the partial list of V that contains only entries of the form $(\mathtt{in}, (*, O))$ or $(\mathtt{control}, (*, l'))$ for some symbolic operation O and some in-metadata l', where the input term and the out-metadata have been masked with the symbol $*$. The list V_{In} is called *attacker strategy.* We write $[V_{In}]_{\mathsf{SViews}(I)}$ to denote the class of all views $U \in \mathsf{SViews}(I)$ with $U_{In} = V_{In}$. ◇

The faking simulator Sim'_f for equivalence properties. We define a faking simulator Sim'_f for equivalence properties as the faking simulator Sim_f for trace properties except that the final output, i.e., the guess, of the attacker $\mathcal{A}$ sent over the output interface is also output by Sim'_f and all terms that are sent to the hybrid challenger from Sim_f are parsed in Sim'_f to symbolic operations, using Construct-shape.

DFC(k) : **upon out-metadata from a decision node**

```
1: respond with the in-metadata continue
```

DFC(k) : **upon another message** m

```
1: send m via the network interface
2:
3: if a message m' is received over the network interface
   then
4:     forward the message m' to execution network interface
5: else if a guess b is received over the output interface then
6:     output the guess b
```

DFNT(k) : **upon out-metadata from a decision node**

```
1: if a guess b is received vai the output interface then
2:     respond with the in-metadata decision-b
3: else
4:     respond with the in-metadata continue
```

DFNT(k) : **upon another message** m

```
1: if no guess was received over the output interface yet
   then
2:     send m over the network interface
3:     if a message m' is received over the network interface
       then
4:         send the message m' to the execution network inter-
           face
5:     else if a guess b is received over the output interface
       then
6:         store the guess b
7: else
8:     respond with dummy messages, i.e., for an input node
       send the constant 0 bitstring, for a control node that
       is not a decision node send one of the possible in-
       metadata, and for an output node give back control
       to the execution interface party
```

Figure 8: Code for the Decision Node Filter for Communication and for Node Traces

5.6.4 CS for trace properties with length functions

The following theorem follows from the CS result of Backes, Unruh, and Malik [9]. We discuss the differences to the CS proof in [9].

Theorem 11 *Let* $\mathbf{A}$ *be a computational implementation fulfilling the implementation conditions (see Appendix 5.2), i.e., in particular* $\mathbf{A}$ *is length-consistent. Then,* $\mathbf{A}$ *is a computationally sound implementation of the symbolic model* $\mathbf{M}$ *for the class of randomness-safe protocols (see Definition 26).*

Proof. The implementation conditions that we require are a superset of the conditions in the work of Backes, Malik, and Unruh (our additional conditions are marked blue). Hence, every implementation that satisfies our implementation condition also satisfies the implementation condition of their work. Moreover, we add one protocol condition that excludes *garbageInvalidLength*-terms as arguments for constructors. This protocol condition only further restricts the class; hence, without length functions, their CS result would still hold.

We extend the simulator *Sim* in the CS proof of the work of Backes, Malik, and Unruh to also parse (τ) and to produce (β) length functions. The rules are straight-forward and follow the same pattern as for $string_b$ and $unstring_b$.

Length functions are constant functions that the attacker can produce on its own, just like payload strings ($string_b$). Consequently, the Dolev-Yaoness of the simulator also holds in the presence of length functions

For the translation functions $\beta : \mathbf{T} \to \{0,1\}^*$ and $\tau : \{0,1\}^* \to \mathbf{T}$ of the simulator, $\beta(length(m)) = A_{length}(\beta(m))$, $\tau(\beta(t)) = t$, $\beta(t) \neq \bot$, and $\beta(\tau(b)) = b$ follow from the implementation condition. The indistinguishability of the simulator follows from these equation (see [9]).

As shown in the initial work on CoSP (see [3, Theorem 1]), a simulator that satisfies Dolev-Yaoness and Indistinguishability implies computational soundness. □

5.6.5 Decision variant of a protocol

In this section, we show how to reduce indistinguishability of the transcripts to indistinguishability of nodes traces. To this end, we define the decision variant of a protocol.

Definition 35 (Decision node filter for communication and node traces) The *decision node filter for communication* is the interactive machine DFC that is defined in Figure 8. The *decision node filter for node traces* is the interactive machine DFNT that is defined in Figure 8.

Both machines expect two communication partners. In our notation, there will be a left communication partner and a right communication partner. Each of these partners offer a network interface: the network interface of the left partner (typically the execution) is called the execution network interface, and the network interface of the right partner (typically the adversary or simulator) is called the network interface. The right communication partner additionally offers an output interface (over which it typically sends its distinguisher-guess). We call this interface the output interface. ◇

Definition 36 (Decision-variant) Given a bi-protocol Π, the *decision-variant* $\tilde{\Pi}$ is defined like Π but with the modification that each node ν that appears in Π is preceded by an additional control node, called *decision node*, as follows: The decision node has three successors with the edges to them labeled by `decision-`0, `decision-`1, and `continue`. The `continue`-successor is ν here (i.e., the protocol continues). Both the `decision-`0-successor and the `decision-`1-successor are roots of infinite chains of dummy control nodes with only one successor (i.e., the protocol stops).[16] $\diamond$

Lemma 12 *Let* $\mathbf{M}$ *be a symbolic model,* Imp *be an implementation,* $\mathcal{A}$ *be a ppt machine, and* P *be the class of all bi-protocols that are the left or the right variant of an efficient randomness-safe bi-protocol Definition 26. If there is a ppt machine* Sim_f *such that for all protocols* $\Pi \in \mathrm{P}$*, the node traces* $\langle \mathsf{TrExec}_{\mathbf{M},Imp,\Pi} \mid \mathcal{A}\rangle$ *and* $\langle \mathsf{TrH\text{-}Exec}_{\mathbf{M},Imp,\Pi} \mid \langle Sim_f \mid \mathcal{A}\rangle\rangle$ *are computationally indistinguishable, then for all protocols* $\Pi \in \mathrm{P}$*, the executions* $\langle \mathsf{Exec}_{\mathbf{M},Imp,\Pi} \mid \mathcal{A}\rangle$ *and* $\left\langle \mathsf{H\text{-}Exec}_{\mathbf{M},Imp,\Pi} \mid \left\langle Sim'_f \mid \mathcal{A}\right\rangle\right\rangle$ *are computationally indistinguishable.*

Proof. For convenience, we consider a computational challenger TrExec and hybrid challenger $\mathsf{TrH\text{-}Exec}$ for trace properties with the following modifications: instead of calling the distinguisher with the node trace after the interaction between the (respective) challenger and the adversary (or the simulator), the challenger assumes that the attacker consists of two machines, the normal adversary and the distinguisher, that do not share their state and it sends the node trace to the second machine. The distinguisher then ends the interaction by outputting a guess $b \in \{0,1\}$.

Accordingly, we define for an adversary $\mathcal{A}$ another machine $\mathcal{A}'$ that consists of two parts: first, a modification of $\mathcal{A}$ that instead of outputting its final guess b chooses via in-metadata the `decision-`b node and then stops; second, a distinguisher part that checks which `decision-`b was taken and outputs a guess b.

Since the simulator need to be compatible with these potentially two parts of $\mathcal{A}$ forwards, we define a variant Sim^*_f of the faking simulator Sim_f for trace properties: Sim^*_f forwards the node trace to the distinguisher part and outputs the final guess of the distinguisher part.

The following interactions are perfectly indistinguishable for the adversary $\mathcal{A}$.

$$
\begin{aligned}
& \mathsf{Exec}_{\mathbf{M},Imp,\Pi} \\
\overset{(1)}{\approx}_{time} \;& \left\langle \mathsf{Exec}_{\mathbf{M},Imp,\tilde{\Pi}} \mid \langle \mathrm{DFC} \mid \mathcal{A}\rangle \right\rangle \\
\overset{(2)}{\approx}_{time} \;& \left\langle \mathsf{Exec}_{\mathbf{M},Imp,\tilde{\Pi}} \mid \langle \mathrm{DFNT} \mid \mathcal{A}\rangle \right\rangle \\
\overset{(3)}{\approx}_{time} \;& \left\langle \mathsf{TrExec}_{\mathbf{M},Imp,\tilde{\Pi}} \mid \langle \mathrm{DFNT} \mid \mathcal{A}'\rangle \right\rangle
\end{aligned}
$$

$$
\begin{aligned}
& \left\langle \mathsf{TrH\text{-}Exec}_{\mathbf{M},Imp,\tilde{\Pi}} \mid \left\langle \left\langle Sim^*_f \mid \mathrm{DFNT}\right\rangle \mid \mathcal{A}'\right\rangle \right\rangle \\
\overset{(4)}{\approx}_{time} \;& \left\langle \mathsf{TrH\text{-}Exec}_{\mathbf{M},Imp,\tilde{\Pi}} \mid \left\langle \left\langle \mathrm{DFNT} \mid Sim^*_f\right\rangle \mid \mathcal{A}'\right\rangle \right\rangle \\
\overset{(3)}{\approx}_{time} \;& \left\langle \mathsf{H\text{-}Exec}_{\mathbf{M},Imp,\tilde{\Pi}} \mid \left\langle \left\langle \mathrm{DFNT} \mid Sim'_f\right\rangle \mid \mathcal{A}\right\rangle \right\rangle \\
\overset{(2)}{\approx}_{time} \;& \left\langle \mathsf{H\text{-}Exec}_{\mathbf{M},Imp,\tilde{\Pi}} \mid \left\langle \left\langle \mathrm{DFC} \mid Sim'_f\right\rangle \mid \mathcal{A}\right\rangle \right\rangle \\
\overset{(1)}{\approx}_{time} \;& \left\langle \mathsf{H\text{-}Exec}_{\mathbf{M},Imp,\Pi} \mid \left\langle Sim'_f \mid \mathcal{A}\right\rangle \right\rangle
\end{aligned}
$$

(1): The decision-variant $\tilde{\Pi}$ of a protocol Π sends the same messages as the original protocol Π to the adversary except for adding a special kind of control nodes: decision nodes (see Definition 36). The messages with the out-metadata from the decision nodes is filtered by the machine DFC, and except for filtering these additional messages DFC does nothing more than forwarding all messages to $\mathcal{A}$. Hence, the indistinguishability holds.

(2): Until $\mathcal{A}$ stops with outputting a bitstring b, the machine DFNT behaves just like DFC. Hence, the indistinguishability holds.

[16] Formally, $\tilde{\Pi}$ is the limes of the operation that recursively replaces each node ν by a decision node followed by ν.

(3): The two executions Exec and TrExec behave exactly the same towards the communication partner (i.e., $\langle \text{DFNT} \mid \mathcal{A}\rangle$ or $\langle \text{DFNT} \mid \langle \mathit{Sim}_f \mid \mathcal{A}\rangle\rangle$, respectively, in our case). Hence, the interactions are indistinguishable for $\mathcal{A}$. The same holds for H-Exec and TrH-Exec except that in this case we additionally replace Sim_f with Sim'_f, which, by the construction of Sim'_f (see Section 5.6.3), is not observable to $\mathcal{A}$.

(4): By the construction of Sim^*_f, we can see that the sub-machine $\mathcal{A}$ does not see a difference if first the simulator Sim^*_f is invoked, as in $\left\langle \mathit{Sim}^*_f \mid \langle \text{DFNT} \mid \mathcal{A}\rangle\right\rangle$ or first the decision nodes are filtered, as in $\left\langle \text{DFNT} \mid \left\langle \mathit{Sim}^*_f \mid \mathcal{A}\right\rangle\right\rangle$. Moreover, Sim'_f outputs the output of its sub-machine, which is in one case $\mathcal{A}$ and in the other case $\langle \text{DFNT} \mid \mathcal{A}\rangle$. Hence, the behavior towards TrH-Exec is indistinguishable, as well. □

Moreover, whenever the attacker $\mathcal{A}$ (or the simulator $\left\langle \mathit{Sim}'_f \mid \mathcal{A}\right\rangle$) outputs a guess b, DFNT eventually sends to a decision node the in-metadata `decision-`b. Hence, the indistinguishability relations (1)–(4) imply that the node traces output by

$$\left\langle \mathsf{TrExec}_{\mathbf{M},\mathit{Imp},\tilde{\Pi}} \mid \langle \text{DFNT} \mid \mathcal{A}\rangle \right\rangle \text{ and } \left\langle \mathsf{TrH\text{-}Exec}_{\mathbf{M},\mathit{Imp},\tilde{\Pi}} \mid \left\langle \mathit{Sim}^*_f \mid \langle \text{DFNT} \mid \mathcal{A}\rangle\right\rangle\right\rangle$$

are distinguishable if the adversary can distinguish

$$\langle \mathsf{Exec}_{\mathbf{M},\mathit{Imp},\Pi} \mid \mathcal{A}\rangle \text{ from } \left\langle \mathsf{H\text{-}Exec}_{\mathbf{M},\mathit{Imp},\Pi} \mid \left\langle \mathit{Sim}'_f \mid \mathit{attacker}\right\rangle\right\rangle$$

This statement is the contraposition of the claim of the lemma.

5.6.6 Uniqueness of a symbolic operation

For a CoSP bi-protocol Π and an adversary $\mathcal{A}$, we say that a symbolic operation O *occurs in a run* of $\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)-\mathrm{F}_b} \mid \mathcal{A}\right\rangle$ *for a message* m for one invokation of Construct-attacker-strategy(b, tr) if it is a sub-symbolic operation of a symbolic operation that was generated by one monitor call. More precisely, a symbolic operation O occurs in a run of $\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)-\mathrm{F}_b} \mid \mathcal{A}\right\rangle$ if in that run there is an invokation of $V :=$ Construct-attacker-strategy(b, tr) such that there is a symbolic operation context C with $V_j = (_, C[O])$ and $\mathit{eval}_{\bar{O}}(tr) = m$. Analogously, we talk about several symbolic operations $O_1, \dots, O_l$ occurring in a run of $\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)-\mathrm{F}_b} \mid \mathcal{A}\right\rangle$ for messages $m_1, \dots, m_l$ if each O_j (for $j \in \{1, \dots, l\}$) occurs in the same run of $\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)-\mathrm{F}_b} \mid \mathcal{A}\right\rangle$ for m_j. If it is clear from the context which computational execution is meant and which invokation is meant, will only say that a symbolic operation O *occurs in a run for a message* m.

Lemma 13 *Let O_{left} and O_{right} be two output shapes that are generated by the knowledge monitor. If $O_{\mathsf{left}} \neq O_{\mathsf{right}}$, then the knowledge monitor raises an alarm* `bad-knowledge`.

In contraposition: whenever two output shapes O_{left} and O_{right}, generated by an invokation of the knowledge monitor, do not cause an alarm `bad-knowledge`*, we have $O_{\mathsf{left}} = O_{\mathsf{right}}$. We call this shape the* joint symbolic operation O_b *of that invokation.*

Proof. In the construction of the monitor, in Figure 7, an alarm is only raised if the symbolic operations are not equal. □

Modifications to Construct-shape. As a technical vehicle, we run a modification of the knowledge monitor. We modify Construct-shape such that it additionally outputs all sub-shapes and all symbolic operations $((O_m, “x''_i”), (O_1, “O''_1”), \dots, (O_n, “O_n”))$ that characterize how the messages to those sub-shapes have been computed (in Figure 5 denoted in quotes: “O”). We stress that, due to the recursive implementation of Construct-shape, the sequence of sub-shapes and symbolic operations is a list in which, if reversed, the sub-shapes always occur before their parent shapes. All other parts of the knowledge monitor remain the same except that they ignore these sub-shapes and the symbolic operations and only use the (top-level) shape, as before. These sub-shapes and the symbolic operations are need to compute the visible submessages of an output message (see Figure 9).

Corollary 14 *Let Π be a CoSP bi-protocol and $\mathcal{A}$ be an adversary. If O is not a sub-symbolic operation, i.e., $C = \cdot$, then we have the following property: For each invokation of* Construct-attacker-strategy(b, tr) *in any run of $\left\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)-\mathrm{F}_b} \mid \mathcal{A}\right\rangle$, we have that for every symbolic operations O for m, if m' is the actual message for which O was generated, i.e., the invokation of $O =$* Construct-shape$(m', x_j, K, b, tr,$ *unaryDec*$)$*, then*

$$m = m'$$

Submessage(m, tr)

1: let $shapes$ = $((O_m, \text{“}x_i''\text{”}), (O_1, \text{“}O_1''\text{”}), \ldots, (O_n, \text{“}O_n\text{”}))$ be the output of the last call (for m) Construct-shape($m, x_i, K, \mathsf{left}, tr, dec'$)

2: reverse $shapes$

3: **for each** $(O_j, \text{“}O_j''\text{”})$ in $shapes$ **do**

4: let $m_j = eval_{\text{“}O_j''}(\mathsf{left}(tr))$

5: output m_j

Figure 9: The algorithm Submessage(m, tr) that is called in the unrolled variant $\tilde{\Pi}$

Proof. The dual symbolic operation output by Construct-shape′ simply points with the projection x_j to the very message m' in the computational view. □

Lemma 15 *Let* Π *be a CoSP bi-protocol and* $\mathcal{A}$ *be an adversary. For each invokation of* Construct-attacker-strategy(b, tr) *in any run of* $\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)-\mathrm{F}_b} \mid \mathcal{A} \rangle$*, we have that for every pair of sub-symbolic operations* O *and* O' *that occur in a run of* $\langle \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)-\mathrm{F}_b} \mid \mathcal{A} \rangle$ *for* m *and* m'*, respectively, the symbolic operations are unique, i.e.,*

$$O \neq O' \Leftrightarrow m \neq m'$$

Proof. Assume towards contradiction that there is a pair O, O' such that $O \neq O'$ but $m = m'$. Let m, m' be the first pair of messages in the run for which this violation occurs. W. l.o.g. assume that first m and then m' was constructed. While constructing O', the algorithm Construct-shape first checks whether there is already a symbolic operation O'' in the knowledge K such that $eval_{\bar{O}''} = m'$. The evaluation $eval$ of the dual operation $\bar{O}''$ uses exactly the same deterministic algorithms for retrieving the bitstring that were used to construct O''. Hence, m' coincides with $eval_{\bar{O}''}$, i.e., the test $eval_{\bar{O}''} = m'$ is well-defined. Thus, if the same bitstring was already parsed, the corresponding symbolic operation is found and used and no new symbolic operation is used.

Moreover, a non-constant minimal symbolic operations O' always uses as randomness $nonceof(\bar{O}'')$ with the dual operation $\bar{O}''$ of O'', which prevents two symbolic operations to be the same for two different bitstrings.

Observe that the knowledge K is already fully saturated; hence, it cannot happen that while constructing O' in Construct-shape a new symbolic operation is learned that leads to a larger K', i.e., $K \neq K'$ and $K \subset K'$. As a consequence, if O' occurs for m', $m = m'$, and O occurs for m, O is always found; hence $O' = O'$.

For showing the contradiction, we also have to consider the case where $O = O'$ but $m \neq m'$. By construction this cannot happen because $m = eval_{\bar{O}}(tr) = eval_{\bar{O}'}(tr) = m'$. □

5.6.7 Unrolled variants

In order to simplify the main proof, we define, for a given protocol Π, its unrolled variant $\tilde{\Pi}$, which before each output node of Π sends all visible sub-messages of the message from that output node.

Definition 37 (Unrolled variant) Let Submessage(m, tr) be the algorithm that is depicted in Figure 9. Given a protocol Π, we define its *unrolled variant* $\tilde{\Pi}$ as the protocol in which each output node with a message m is replaced by , where tr is the trace from the output node to the root.[17] ⋄

Corollary 16 (Unrolled variants preserve uniformity-enforcing) *For all uniformity-enforcing protocols* Π*, the unrolled variant* $\tilde{\Pi}$ *of* Π *is uniformity-enforcing.*

Proof. The inserted sub-protocol is uniformity-enforcing since before each computation node a single-successor control node is placed that has a unique out-metadata. □

Lemma 17 (Symbolic indistinguishability is preserved in unrolled variants) *For all efficient pairs of protocols* Π_1 *and* Π_2*, if* Π_1 *and* Π_2 *are symbolically indistinguishable, then the corresponding unrolled variants* $\tilde{\Pi}_1$ *and* $\tilde{\Pi}_2$ *are symbolically indistinguishable.*

Proof. First, we show the following statememt.

[17] Formally, $\tilde{\Pi}$ is the limes of the recursive replacement operation that replaces each output node with Submessage(m, tr).

Claim 1 *For each CoSP protocol* Π*, let* $\tilde{\Pi}$ *be the unrolled variant of* Π*. For all attacker-strategies* V_{In} *and for all views* $V \in [V_{In}]_{\mathsf{SViews}(\Pi)}$ *and* $V' \in [V_{In}]_{\mathsf{SViews}(\tilde{\Pi})}$ *the symbolic knowledge* K_V *of* Π *and* $K_{V'}$ *of* $\tilde{\Pi}$ *is the same, i.e.,* $K_V = K_{V'}$*.*

Proof of Claim 1. The statement follows from the fact that the only difference of an unrolled variant $\tilde{\Pi}$ and the original protocol Π is that $\tilde{\Pi}$ additionally sends the output messages computed by Submessage(m, tr). Since the additional messages sent by Submessage(m, tr) are in the symbolic knowledge after m is sent, the statement follows. $\diamond$

Assume that the unrolled variants $\tilde{\Pi}_1$ and $\tilde{\Pi}_2$ are not symbolically indistinguishable. W.l.o.g., there is an attacker strategy V_{In} and a view $\tilde{V} \in [V_{In}]_{\mathsf{SViews}(\tilde{\Pi}_1)}$ of $\tilde{\Pi}_1$ under V_{In} such that for all views $\tilde{V}' \in [V_{In}]_{\mathsf{SViews}(\tilde{\Pi}_2)}$ of $\tilde{\Pi}_2$ under V_{In} it holds that $\tilde{V} \not\sim V'$. Let $V \in [V_{In}]_{\mathsf{SViews}(\Pi_1)}$ and $V' \in [V_{In}]_{\mathsf{SViews}(\Pi_2)}$ be the corresponding views for Π_1 and Π_2.

For all $\tilde{V}' \in [V_{In}]_{\mathsf{SViews}(\tilde{\Pi}_2)}$, one of the following three statements does not hold:

1. (Same structure) $\tilde{V}_i$ is of the form $(s, \cdot)$ if and only if V'_i is of the form $(s, \cdot)$ for some $s \in \{\mathtt{out}, \mathtt{in}, \mathtt{control}\}$.
2. (Same out-metadata) $\tilde{V}_{Out\text{-}Meta} = \tilde{V}'_{Out\text{-}Meta}$.
3. (Same symbolic knowledge) $K_{\tilde{V}} = K_{\tilde{V}'}$.

If the structure is not the same (Case 1), then we first consider the following cases, which are related to invokations of Submessage: either a different different branch was taken in an invokation of Submessage or a different amount of output messages were sent by an invokation of Submessage. Since Submessage internally only branches when computing Construct-shape, all tests that Submessage performs are in the shared knowledge and hence in the symbolic knowledge of Π_i. Thus, in these cases, the symbolic knowledge is already different for Π_1 and Π_2. In all other cases, the unrolled variant $\tilde{\Pi}_i$ coincides with the original protocol Π_i; hence the same difference occurs in Π_i.

For the case that the out-metadata is different (Case 2), observe that the unrolled variant does not sends the same out-metadata as the original protocol. Hence, the statement follows.

For the case that the symbolic knowledge (Case 3), the statement follows from the claim above. $\square$

Lemma 18 (Unrolled variants preserve computational indistinguishability) *For all efficient protocols* Π_1*,* Π_2*, if the corresponding unrolled variants* $\tilde{\Pi}_1$ *and* $\tilde{\Pi}_2$ *are indistinguishable, then* Π_1 *and* Π_2 *are indistinguishable. Moreover, if the corresponding unrolled variants* $\tilde{\Pi}_1$ *and* $\tilde{\Pi}_2$ *are perfectly indistinguishable, then* Π_1 *and* Π_2 *are perfectly indistinguishable.*

Proof. We construct a reduction R from a distinguisher $\mathcal{A}$ of two original protocols Π_1 and Π_2 to a distinguisher $R_{\mathcal{A}}$ of the two unrolled variants $\tilde{\Pi}_1$ and $\tilde{\Pi}_2$. The reduction $R_{\mathcal{A}}$ internally runs $\mathcal{A}$ but drops all sub-messages and only sends the final message to the attacker. Finally, $R_{\mathcal{A}}$ outputs the same guess as the distinguisher $\mathcal{A}$. The view of $\mathcal{A}$ is perfectly indistinguishable to an interaction with Π_1 and Π_2, and the success probability of the reduction $R_{\mathcal{A}}$ to distinguish $\tilde{\Pi}_1$ from $\tilde{\Pi}_2$ equals the success probability of $\mathcal{A}$ for distinguishing Π_1 from Π_2. $\square$

5.6.8 Computational self-monitoring of the knowledge monitor

Finally, we are in a position to present the proof of the computational self-monitoring.

Symbolic distinguishability. In the proof of computational self-monitoring, we need the property that the monitor is symbolically distinguishing, i.e., that it symbolically always raises an alarm if a pair of protocols is symbolically distinguishable. Note that this essentially the opposite direction of symbolic self-monitoring (Definition 23).

Definition 38 (Symbolically distinguishing monitor) A self-monitor is *symbolically distinguishing* if for all bi-protocols Π the following holds: if Π is symbolically distinguishable, then the self-monitor raises an alarm. $\diamond$

Lemma 19 (Knowledge monitor is symbolically distinguishing) *The distinguishing self-monitor* $f_{\mathtt{bad\text{-}knowledge},\Pi}$ *is symbolically distinguishing in the sense of Definition 38.*

Proof. We have to show that if the monitor $f_{\texttt{bad-knowledge},\Pi}$ does not raise an alarm, then a bi-protocol is equivalent (see Definition 38). By Lemma 13, we know that if the monitor does not raise an alarm, then the symbolic operation that is computed for the output message for $\mathsf{left}(\Pi)$ and $\mathsf{right}(\Pi)$ is the same. We denote this symbolic operations as O_b.

We stress that for each run the symbolic operations output by Construct-shape are unique w.r.t. the messages that are sent or received for the following reason: for all symbolic operations O that use randomness, Construct-shape places $\mathit{nonceof}(\bar{O})$ as the symbolic operation for the randomness, where $\bar{O}$ is the respective dual symbolic operations of O. As a consequence, it cannot happen that Construct-shape assigns for two different messages in one run the same symbolic operation.

As a next step, we perform an induction over the length i of the protocol and show that for all cases O_b is equivalent. Assume that the monitor did not raise an alarm and Π_{i-1} is equivalent but Π_i is not equivalent.

1. **O_b is a message that is in the attacker-knowledge against Π_{i-1}.** Messages that are in the attacker-knowledge against Π_{i-1} are messages that the attacker could have sent after an interaction Π, i.e., with $\Pi_{i-1,\mathsf{left}}$ or $\Pi_{i-1,\mathsf{right}}$. For these messages the equivalence follows by induction hypothesis.

   This case, in particular, covers garbage terms, garbage encryptions, garbage signatures, and attacker-generated keys. For attacker-generated ciphertexts and signatures, since we consider unrolled variants, all visible sub-messages do not lead to distinguishing symbolic operations by induction hypothesis. Recall that, due to the recursive implementation of Construct-shape, the sequence of sub-shapes and symbolic operations is a list in which, since reversed, the sub-shapes always occur before their parent shapes. Thus, the same holds for Submessage and thus for unrolled variants. Hence, attacker-generated ciphertexts and signatures are also covered by this case. Consequently, we exclude such attacker-generated messages in the following cases.

2. **Empty string: $O_b = \mathit{emp}()$.** If $O_{\mathsf{left}} = O_{\mathsf{right}} = \mathit{emp}()$, then $t_{\mathsf{left}} = t_{\mathsf{right}} = \mathit{emp}()$. Hence, the equivalence follows.

3. **Protocol nonce: $O_b = n_P$.** This case cannot occur. In Construct-shape, a protocol nonce is always parsed as a projection.

4. **Protocol-generated encryption or verification key: $O_b \in \{\mathit{ek}(\mathit{nonceof}(O_1)), \mathit{vk}(\mathit{nonceof}(O_1))\}$.** For protocol-generated keys, equality of the symbolic operation means that the corresponding message is equal. Either the encryption key is fresh in the shared knowledge, i.e., never used before, then learning it does not help distinguishing Π_{i-1}, or the encryption key was used earlier, then it already was in the shared knowledge for Π_{i-1} and by induction hypothesis the messages are equivalent. The same argumentation holds for the case if O_b characterizes a verification key.

5. **Protocol-generated decryption key: $O_b \in \{\mathit{dk}(\mathit{nonceof}(O_1)), \mathit{dkofek}(O_1)\}$.** For protocol-generated keys, equality of the symbolic operation means that the corresponding message is equal. The decryption key can only be used to decryption ciphertexts with the corresponding encryption key. The Construct-attacker-strategy reconstructs all symbolic operations after learning the output message of node i, which O_b characterizes, and the knowledge monitor checks by whether all resulting symbolic operations are equal. Hence, whenever a decryption key is learned, all symbolic operations that characterize ciphertexts that are constructed with the matching encryption key contain the respective plaintext. As a consequence, every distinguishing test leads to different symbolic operations (see Lemma 15). By contraposition, since `bad-knowledge` was not raised and hence all symbolic operations are equal, there is no distinguishing test.

6. **Protocol-generated signing key: $O_b \in \{\mathit{sk}(\mathit{nonceof}(O_1)), \mathit{skofvk}(O_1)\}$.** By induction hypothesis, Π_{i-1} is equivalent; hence, there is no distinguishing symbolic operation for $\Pi_{i-1,\mathsf{left}}$ or $\Pi_{i-1,\mathsf{right}}$. Since $O_{\mathsf{left}} = O_{\mathsf{right}} = O_b$, learning the message that corresponds to O_b does not offer a distinguishing equality test. Since only certain plaintexts of ciphertexts (i.e., encryption terms) and randomness terms are unaccessible. Plaintexts can only be accessed via decryption keys, and in the symbolic model there is no destructor that grants access to the randomness, sk does not introduce novel distinguishing symbolic operations for Π_i.

7. **A protocol-generated, derived public-key: $O_b \in \{\mathit{vkofsk}(O_1), \mathit{ekofdk}(O_1)\}$.** Either these protocol-generated keys are already in the adversary's knowledge, or they have never been used anywhere yet. Consequently, equivalence follows from the equivalence of Π_{i-1}.

8. **The virtual term:** $O_b = \mathit{plaintextof}(O_1, O_2)$**.** This case cannot occur since $\mathit{plaintextof}(O_1, O_2)$ only occurs inside an encryption symbolic operation and it is never a visible sub-message.

9. **Protocol-generated encryption:** $O_b = \mathit{enc}(O_1, \mathit{plaintextof}(O_2, O_3), O_4)$**.** Observe that by the construction of Construct-shape this case only happens for honestly generated ciphertexts for which the decryption key has not been leaked. More precisely, if the decryption key is not in the symbolic knowledge of the adversary. Hence, it suffices to show that the length of O_2 is the same in the term that corresponds to O_b in $\mathsf{left}(\Pi_i)$ and $\mathsf{right}(\Pi_i)$. If no alarm was raised, the length of both terms are the same because the knowledge monitor implicitly performs these length tests by comparing the symbolic operations, which in turn are unique for each length of a message.

10. **Protocol-generated encryption:** $O_b = \mathit{enc}(O_1, O_2, O_3)$ **where** $O_2 \neq \mathit{plaintextof}(\ O_4, O_5)$**.** By the construction of Construct-shape, we know that this case can occur in two cases: first, if the corresponding the decryption key is in the shared knowledge and second if the encryption was produced with an attacker-key. Since we consider unrolled variants, we know that with previous output nodes the bitstrings that correspond to O_1 and O_2 have been sent to the adversary. Recall that in the unrolled variants sub-messages are always sent before their parent messages. By assumption we know that for Π_{i-1} the statement holds. By the construction of Construct-shape, we know that the term that corresponds to $\mathit{enc}(O_1, O_2, O_3)$ is an honestly generated ciphertext. As a consequence, Π_i did not leak more information that Π_{i-1} since by the protocol conditions the protocol randomness is a freshly chosen nonce. Hence, Π_i is equivalence as well.

11. **Protocol-generated signature:** $O_b \in \{\mathit{sig}(\mathit{sk}(O_1), O_2, O_3), \mathit{sig}(\mathit{skofvk}(O_1), O_2,\ O_3)\}$**.** Since we consider unrolled protocols, with the last two output nodes the protocol already sent $\mathit{vkof}(O_b)$ and O_2 to the adversary. Recall that in the unrolled variants sub-messages are always sent before their parent messages. In other words, there is no distinguishing symbolic operation against Π_{i-1}, i.e., no symbolic operation that has a different outcome in the left and the right protocol. But then there can also not be any distinguishing symbolic operation against $\mathit{sig}(\mathit{sk}(O_1), O_2, O_3)$ or $\mathit{sig}(\mathit{skofvk}(O_1), O_2, O_3)$, since the symbolic model only has for signatures the verification operation (verify), the length test (length), and the retrieval of the verification key (vkof).

12. **Pairs:** $\mathit{pair}(O_1, O_2)$**.** Since we solely consider unrolled variants of protocols, we know that both O_1 and O_2 have already been sent in Π_{i-1}. Recall that in the unrolled variants sub-messages are always sent before their parent messages. Hence, the equivalence immediately following from induction hypothesis about Π_{i-1}.

13. **Payload strings:** $O_b \in \{\mathit{string}_0(O_1), \mathit{string}_1(O_1)\}$**.** This case follows by induction hypothesis, since Lemma 18 shows that it suffices to consider unrolled variants of protocols and O_b can be computed out of O_1.

14. **Length term:** $O_b = S(O_1)$**.** This case follows by induction hypothesis, since Lemma 18 shows that it suffices to consider unrolled variants of protocols and O_b can be computed out of O_1. □

Lemma 20 *The class of randomness-safe (Definition 26) bi-protocols is closed under taking decision-variants and unrolled variants of bi-protocols:*

1. *A bi-protocol is randomness-safe if and only if its decision-variant is randomness-safe.*

2. *A bi-protocol is randomness-safe if and only if its unrolled variant is randomness-safe.*

Proof.

1. By inspection of Definition 26, randomness-safety is invariant under adding and removing control nodes (decision nodes and dummy nodes).

2. By inspection of Definition 26, randomness-safety is invariant under removing output nodes. However, adding output nodes could potentially violate protocol conditions 2 to 4 in Definition 26. Since Definition 37 adds only output nodes that do not refer to a randomness node, these conditions are not violated.

The main lemma. All the preparations in this section culminate in the following technical lemma.

Lemma 21 (Same conditional distribution for faking simulator) *Let Tr_{break} be the event that there is a trace property that holds symbolically but does not hold in the current run.*

Let $\Omega(\Pi, \mathcal{A})_b$ be defined as the following probability space:

$$guess \leftarrow \langle \mathsf{H\text{-}Exec}_{\mathbf{M},Imp,\mathsf{Mon}(\Pi)-\mathrm{F}_b} \mid \langle Sim'_f \mid \mathcal{A} \rangle \rangle$$

where $\mathsf{Mon}(\Pi) - \mathrm{F}_b$ is defined as in Definition 31, $b \in \{\mathsf{left}, \mathsf{right}\}$, and the output guess of this interaction denotes the output of $Sim_{f,\mathcal{A}}$ (and thereby of $\mathcal{A}$).

For all ppt adversaries $\mathcal{A}$, for all $i \in \mathbb{N}$, and for all uniformity-enforcing, randomness-safe efficient bi-protocols Π: If

$$\begin{aligned}&\Pr[\Omega(\Pi_{i-1}, \mathcal{A})_{\mathsf{left}} : guess = 1 \mid \neg Tr_{break} \land \neg \mathsf{bad}] \\ &= \Pr[\Omega(\Pi_{i-1}, \mathcal{A})_{\mathsf{right}} : guess = 1 \mid \neg Tr_{break} \land \neg \mathsf{bad}]\end{aligned}$$

holds, then the following holds:

$$\begin{aligned}&\Pr[\Omega(\Pi_{i}, \mathcal{A})_{\mathsf{left}} : guess = 1 \mid \neg Tr_{break} \land \neg \mathsf{bad}] \\ &= \Pr[\Omega(\Pi_{i}, \mathcal{A})_{\mathsf{right}} : guess = 1 \mid \neg Tr_{break} \land \neg \mathsf{bad}]\end{aligned}$$

Proof. Let Π_i be the shortened protocol that halts after the ith output node. By Lemma 18 and Item 2 in Lemma 20 we can assume that Π is unrolled without loss of generality. For $i = 0$, the equality holds, because $\mathsf{Mon}(\Pi_0)$ does not send a message. For $i > 0$, we know that the equality of the probabilities for $\Omega(\Pi_{i-1}, \mathcal{A})_b$ holds by induction hypothesis. Let m_{left} and m_{right} be the terms the are sent at output node i. For m_{left} and m_{right}, let O_{left} and O_{right} be their respective output shapes. Lemma 13 shows that O_{left} and O_{right} are equal; hence, we denote them as O_b.

We stress that for each run the symbolic operations output by Construct-shape are unique w.r.t. the bitstrings that are sent or received for the following reason: for all symbolic operations O that use randomness, Construct-shape places $nonceof(\bar{O})$ as the symbolic operation for the randomness, where $\bar{O}$ is the respective dual symbolic operations of O. As a consequence, it cannot happen that Construct-shape assigns for two different bitstrings in one run the same symbolic operation (see Lemma 15).

We conduct a case distinction over O_b.

1. **Projections: $O_b = x_j$.** The projection x_j corresponds to the jth message that has been sent by the protocol; hence, this message refers to a former message. For all attacker strategies $Strat_{i,O_b}$ that lead to this case for O_b for the ith output node, we construct an reduction R against the induction hypothesis, i.e., that the probabilities for $\Omega(\Pi_{i-1}, \mathcal{A})_{\mathsf{left}}$ and $\Omega(\Pi_{i-1}, \mathcal{A})_{\mathsf{right}}$ are the same.

   R forwards all messages from $\mathsf{H\text{-}Exec}_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_{i-1})-\mathrm{F}_b}$ to and from $\langle Sim'_f \mid \mathcal{A} \rangle$ until the first response after the $i-1$st output node. Then, R potentially responds with the out-metadata that the protocol would send for control nodes between the $i-1$st output node and the ith output node.[18] Then, R responds with jth protocol message to the attacker. Finally, upon a *guess* by the attacker, R also outputs *guess* as a distinguishing guess for $\mathsf{H\text{-}Exec}_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_{i-1})-\mathrm{F}_b}$.

   Observe that since no alarm was raised, the branching monitor, in particular, did not raise an alarm. Thus, since by Corollary 16 Π is uniformity-enforcing, the control flow of both programs is the same. Moreover, if this case (with $O_b = x_j$) is reached with non-zero probability, there is a non-zero probability that x_j is the response that the protocol would have given as well. As a consequence, if the attacker succeeds with non-zero probability to distinguish $\mathsf{H\text{-}Exec}_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_b}$ in this case, i.e., for $O_b = x_j$, then R also succeeds with non-zero probability against $\mathsf{H\text{-}Exec}_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_{i-1})-\mathrm{F}_b}$.

2. **O_b is a message that is in the attacker-knowledge against Π_{i-1}.** Messages that are in the attacker-knowledge are messages such that there is an extended symbolic operation (see Definition 27) that is a shared test (see Definition 32) O with the following property: the symbolic operation that characterizes the message $eval_O(tr)$ is O_b (see the discussion about the uniqueness in Section 5.6.6 and Lemma 15). We distinguish two cases: first, the message m characterized by O_b is not only known through a protocol-generated ciphertext that uses an attacker-generated key; second, m is only known through such a ciphertext.

   Formally, in the first case O does not contain *unaryDec*, i.e., there is a shared test O such that the symbolic operation that characterizes $eval_O(tr)$ is O_b we have that for all C such that $O = C[O']$

[18] This is efficiently computable since R is specific for the protocol Π and the out-metadata for all control nodes is efficiently computable.

we have that $O' \neq unaryDec(O'')$. In the second case for all O such that the symbolic operation that characterizes $eval_O(tr)$ is O_b we have that there is a C such that $O = C[O']$ we have that $O' = unaryDec(O'')$.

In the first case, the reduction to the induction hypothesis can evaluate O, i.e., compute $eval_O(tr)$, resulting in a projection to an input node after the jth output node, for $j \leq i-1$. Then, we construct a reduction against P_{i-1}, along the lines of the reduction from Case 1. Thus, the statement follows by the induction hypothesis.

In the second case, the reduction cannot directly evaluate O because it would need to evaluate *unaryDec*, which directly accesses a computation node. Recall that there is a C and a shared test O_c such that $O = C[unaryDec(O_c)]$. In particular, we know that in each run the bitstring c characterized by O_c is known by the attacker and is protocol-generated. In particular, by the construction of unrolled variants, we know that c is a visible sub-message of a message that has been sent to the attacker. Recall that in the unrolled variants sub-messages are always sent before their parent messages. By the construction of Construct-shape (in Line 16), we know that the symbolic operation that characterizes c is of the form $enc(O_1, O_2, O_3)$ with $O_2 \neq plaintextof(O'_2)$. Since m is a visible sub-message of c by the construction of unrolled variants (see Defintion 37), we know that then its plaintext m characterized by O_2 and by O_b has been sent in an earlier output node as well. In other words, there is a projection that points to the output node that output m. As a consequence, we can construct a reduction as in Case 1, and the statement follows by the induction hypothesis.

We stress that these cases in particular cover garbage message, garbage ciphertexts, garbage signatures, and attacker-generated keys. For attacker-generated ciphertexts and signatures, since we consider unrolled variants, all visible sub-messages do not lead to distinguishing symbolic operations by induction hypothesis. Hence, attacker-generated ciphertexts and signatures are covered by this case, as well. Consequently, we exclude such attacker-generated messages in the subsequent cases.

3. **The empty payload string:** $O_b = emp()$**.** The bitstring the corresponds to O_b is the same for $\mathrm{F}_{\mathsf{left}}$ and $\mathrm{F}_{\mathsf{right}}$. Along the lines of Case 1, the statement follows by the induction hypothesis.

4. **Protocol nonce:** $O_b = n_P$**.** This case cannot occur. A protocol nonce is always parsed as a projection.

5. **Protocol-generated encryption or verification key:** $O_b \in \{ek(nonceof(O_1)), vk(nonceof(O_1))\}$**.** For protocol-generated keys, equality of the symbolic operations O_{left} and O_{right} means that the corresponding bitstrings has the same distribution. Either the encryption key is fresh in the shared knowledge, i.e., never used before, then learning it (information theoretically) does not help distinguishing Π_{i-1}, or the encryption key was used earlier, then it already was in the shared knowledge for Π_{i-1} and along the lines of Case 1 we can show by induction hypothesis that the equality holds. The same argumentation holds for the case if O_b characterizes a verification key.

6. **Protocol-generated decryption key:** $O_b \in \{dk(nonceof(O_1)), dkofek(O_1)\}$**.** For protocol-generated keys, equality of the symbolic operations O_{left} and O_{right} means that the corresponding bitstrings have the same distribution. The decryption key can only be used to decryption ciphertexts with the corresponding encryption key. In an unrolled all plaintext messages that are in the symbolic knowledge after learning the decryption key (the message corresponding to) O_b are sent directly before (the message corresponding to) O_b is sent. Recall that in the unrolled variants sub-messages are always sent before their parent messages. Since we can consider an unbounded simulator and by Implementation Condition 30 there is exactly one decryption key for each encryption key, we can reduce this case to the induction hypothesis (along the lines of Case 1).

7. **Protocol-generated signing key:** $O_b \in \{sk(nonceof(O_1)), skofvk(O_1)\}$**.** For protocol-generated keys, equality of the symbolic operations O_{left} and O_{right} means that the corresponding bitstrings have the same distribution. If $O_b = skofvk(O_1)$, the equality follows from the induction hypothesis, i.e., because verification key is equally distributed, and because for every verification key there is exactly one signing key (Implementation Condition 31). If $O_b = sk(nonceof(O_1))$, then by the construction of Construct-shape, the signing key (characterized by O_b) was never used before. Hence, the equality follows from the fact that the corresponding bitstring have the same distribution.

8. **A protocol-generated, derived public-key:** $O_b \in \{vkofsk(O_1), ekofdk(O_1)\}$**.** For protocol-generated keys, the distribution of the bitstring that corresponds to O_b is equally distributed in $\Omega(\Pi_i, \mathcal{A})_{\mathsf{left}}$ and $\Omega(\Pi_i, \mathcal{A})_{\mathsf{right}}$. By induction hypothesis, the statement follows. These are public keys and $O_b = O_{\mathsf{left}} = O_{\mathsf{right}}$. Hence, either these keys are already in the adversary's knowledge, or they have never been used anywhere yet. Consequently, with an argument along the lines of Case 1 the indistinguishability follows from the indistinguishability of Π_{i-1}.

9. **A virtual term:** $O_b = plaintextof(O_1, O_2)$**.** This case cannot occur since $plaintextof(O_1, O_2)$ only occurs inside an encryption symbolic operation.

10. **Encryption:** $O_b = enc(O_1, plaintextof(O_2, O_3), O_4)$**.** Observe that by the construction of Construct-shape this case only happens for honestly generated ciphertexts for which the decryption key has not been leaked. More precisely, if the decryption key is not in the approximation of the symbolic knowledge of the adversary that the monitor internally computes. By Lemma 19, we know that if the monitor does not raise an alarm then there is symbolically no distinguishing symbolic operation. Hence, if the key is not in the approximated symbolic knowledge, then the key is also not in the full symbolic knowledge (otherwise a counterexample to Lemma 19 can be constructed). If the decryption key is not in the symbolic knowledge, then by Lemma 1 from [9] we know that the simulator (i.e., $\beta^\dagger$) never called $\texttt{getdk}_{\texttt{ch}}$.

    If in turn $\texttt{getdk}_{\texttt{ch}}$ was never called, then for computing a ciphertext with the respective encryption key only $\texttt{fakeenc}_{\texttt{ch}}(R, l)$ is used, where l equals the bitstring that corresponds to $plaintextof(O_2, O_3)$ and R is some internal register of the PROG-KDM challenger. In other words, the faking PROG-KDM challenger faked the encryption, i.e., it did not use the plaintext for constructing the ciphertext. Hence, by construction of the PROG-KDM challenger, the bitstrings corresponding to O_{left} and O_{right} are equally distributed. By induction hypothesis (for Π_{i-1}), we can conclude that the success probability for $\Omega(\Pi_i, \mathcal{A})_{\mathsf{left}}$ is thus the same as for $\Omega(\Pi_i, \mathcal{A})_{\mathsf{right}}$.

11. **Encryption:** $O_b = enc(O_1, O_2, O_3)$ **where** $O_2 \neq plaintextof(O_4, O_5)$**.** Since we consider unrolled variants (see Lemma 18), we know that with previous output nodes the bitstrings that correspond to O_1 and O_2 have been sent to the adversary. Recall that in the unrolled variants sub-messages are always sent before their parent messages. By induction hypothesis we know that for Π_{i-1} the statement holds. By the construction of Construct-shape, we know that the bitstring that corresponds to $enc(O_1, O_2, O_3)$ is an honestly generated ciphertext. As a consequence, Π_i did not leak more information that Π_{i-1} since the randomness of the encryption algorithm is chosen uniformly at random (Item 1 in Definition 26 and Item 3 in Section 5.2) Hence, the probabilities for $\mathsf{H\text{-}Exec}_{\mathbf{M}, Imp, \mathsf{Mon}(\Pi_i) - \mathrm{F}_{\mathsf{left}}}$ and $\mathsf{H\text{-}Exec}_{\mathbf{M}, Imp, \mathsf{Mon}(\Pi_i) - \mathrm{F}_{\mathsf{right}}}$ are equal.

12. **Signature:** $O_b \in \{sig(sk(O_1), O_2, O_3), sig(skofvk(O_1), O_2, O_3)\}$**.** Since we consider unrolled protocols, with in the last two output nodes the protocol already sent $vkof(O_b), O_2$ to the adversary. Recall that in the unrolled variants sub-messages are always sent before their parent messages. By induction hypothesis for Π_{i-1}, we know that with Π_{i-1} the distribution is the same as long as $\texttt{bad}$ has not been raised. Assume that there is an adversary $\mathcal{A}$ that can distinguish $\mathsf{H\text{-}Exec}_{\mathbf{M}, Imp, \mathsf{Mon}(\Pi_i) - \mathrm{F}_{\mathsf{left}}}$ from $\mathsf{H\text{-}Exec}_{\mathbf{M}, Imp, \mathsf{Mon}(\Pi_i) - \mathrm{F}_{\mathsf{right}}}$ but not the same scenario for $i - 1$. Information theoretically, if the two distributions that already send $vkof(O_b), O_2$ are the same, then also the two distributions that additionally send $vkof(O_b), O_2, O_b$ are the same.

13. **Garbage encryption or signature:** $O_b \in \{garbageEnc(O_1, O_2, l), garbageSig(O_1,\ O_2, l)\}$**.** By the construction of Construct-shape and the protocol conditions, we know that the protocol cannot have produced a bitstring with a garbage encryption or garbage signature as an output shape. Hence, O_b characterizes a previously observed attacker-generated ciphertext or signature. Hence, the equality immediately follows from induction hypothesis, with a reduction along the lines of Case 1.

14. **Pairs:** $pair(O_1, O_2)$**.** Since we solely consider unrolled variants of protocols (see Lemma 18), we know that both O_1 and O_2 have already been sent in Π_{i-1}. Recall that in the unrolled variants sub-messages are always sent before their parent messages. Hence, the equality immediately follows from induction hypothesis, with a reduction along the lines of Case 1.

15. **Payload strings:** $O_b \in \{string_0(O_1), string_1(O_1)\}$**.** This case follows by induction hypothesis along the lines of Case 1, since Lemma 18 shows that it suffices to consider unrolled variants of protocols and O_b can be computed out of O_1.

16. **Length term:** $O_b = S(O_1)$**.** This case follows by induction hypothesis along the lines of Case 1, since Lemma 18 shows that it suffices to consider unrolled variants of protocols and O_b can be computed out of O_1. □

Lemma 22 *For the honest simulator Sim and the faking simulator Sim_f (see Section 5.6.3), we have*

$$Sim \approx_{comp} Sim_f$$

Proof. As a corollary of Lemma 12, we know that if the execution interface is connected to a hybrid execution for some efficient randomness-safe CoSP protocol Π, the statement holds for all machines that are connected to the network interface of *Sim* or Sim_f. For distinguishers that are directly connected to the execution interface and behave differently, the only inputs that cause a distinguishing behavior are inputs of protocols that do not obey the protocol conditions, in particular if terms that are used in a randomness position, i.e., randomness messages, are additionally sent in clear. If such randomness messages would be sent in clear the distinguisher could, e.g., reconstruct an encryption and check whether this randomness message has been used (as in *Sim*), or an external oracle has been used to construct a known ciphertext (as in Sim_f). However, we constructed the honest simulator *Sim* and the faking simulator Sim_f such that it checks whether these protocol conditions are met (see the construction of the honest simulator in Section 5.6.3). If this check fails, the execution is aborted, both in *Sim* and in Sim_f. □

Lemma 23 (Computational self-monitoring of knowledge monitor) *The parametric CoSP protocol $f_{\texttt{bad-knowledge},\Pi}$ satisfies computational self-monitoring (see Definition 23).*

Proof. Recall that for computational self-monitoring, we have to show that if `bad-knowledge` in $\mathsf{Mon}(\Pi_i)$ occurs computationally with at most negligible probability, Π_{i-1} is computationally indistinguishable, and the ith node in Π_i is an output node, then Π_i is time-insensitively computationally indistinguishable:

$$\mathsf{Exec}_{\mathbf{M},Imp,\mathsf{left}(\Pi_i)} \approx_{tic} \mathsf{Exec}_{\mathbf{M},Imp,\mathsf{right}(\Pi_i)}$$

By assumption, we know that `bad-knowledge` occurs with at most negligible probability. Hence, the following indistinguishability relations hold (for $b \in \{\mathsf{left}, \mathsf{right}\}$):

$$\begin{array}{cl} & \mathsf{Exec}_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_b} \\ \overset{\text{Lemma 12 \& [9]}}{\approx_{time}} & \langle \mathsf{H\text{-}Exec}'_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_b} \mid Sim' \rangle \end{array}$$

By the computational soundness for trace properties (Theorem 11) of the symbolic model, Tr_{break} only happens with negligible probability. Furthermore, since `bad` only happens with negligible probability and $\mathsf{Exec}_{\mathbf{M},Imp,\mathsf{left}(\Pi_{i-1})}$ is indistinguishable from $\mathsf{Exec}_{\mathbf{M},Imp,\mathsf{right}(\Pi_{i-1})}$, Lemma 21 implies that $\langle \mathsf{H\text{-}Exec}_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{left}}} \mid Sim'_f \rangle$ is indistinguishable from $\langle \mathsf{H\text{-}Exec}_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{right}}} \mid Sim'_f \rangle$.

$$\langle \mathsf{H\text{-}Exec}'_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{left}}} \mid Sim'_f \rangle \quad \approx_{tic} \quad \langle \mathsf{H\text{-}Exec}'_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{right}}} \mid Sim'_f \rangle$$

By Lemma 22, we know that $Sim'_f \approx_{comp} Sim'$. Unruh has proven that if $\langle C \mid A \rangle \approx_{tic} \langle D \mid A \rangle$ and $A \approx_{comp} B$ holds and A, B have polynomially bounded running-time then $\langle C \mid B \rangle \approx_{tic} \langle D \mid B \rangle$ holds [38, Lemma 22]. As a consequence, we get

$$\begin{array}{crcl} & \langle \mathsf{H\text{-}Exec}'_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{left}}} \mid Sim'_f \rangle & \approx_{tic} & \langle \mathsf{H\text{-}Exec}'_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{right}}} \mid Sim'_f \rangle \\ \wedge & Sim'_f & \approx_{comp} & Sim' \\ \Longrightarrow & \langle \mathsf{H\text{-}Exec}'_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{left}}} \mid Sim' \rangle & \approx_{tic} & \langle \mathsf{H\text{-}Exec}'_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{right}}} \mid Sim' \rangle \end{array}$$

Unruh has, furthermore, proven that if $C \approx_{time} A$ and $D \approx_{time} B$ and $A \approx_{tic} B$, then $C \approx_{tic} D$ [38, Corollary 13]. Thus, we conclude the proof as follows:

$$\begin{array}{crcl} & \mathsf{Exec}_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{left}}} & \approx_{time} & \langle \mathsf{H\text{-}Exec}'_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{left}}} \mid Sim' \rangle \\ \wedge & \mathsf{Exec}_{\mathbf{M},Imp,,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{right}}} & \approx_{time} & \langle \mathsf{H\text{-}Exec}'_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{right}}} \mid Sim' \rangle \\ \wedge & \langle \mathsf{H\text{-}Exec}'_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{left}}} \mid Sim' \rangle & \approx_{tic} & \langle \mathsf{H\text{-}Exec}'_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{right}}} \mid Sim' \rangle \\ \Longrightarrow & \mathsf{Exec}_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{left}}} & \approx_{tic} & \mathsf{Exec}_{\mathbf{M},Imp,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{\mathsf{right}}} \end{array}$$

With the application of Corollary 8, we conclude the proof:

$$\begin{array}{crcl} & \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)-\mathrm{F}_{\mathsf{left}}} & \approx_{tic} & \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{Mon}(\Pi)-\mathrm{F}_{\mathsf{right}}} \\ \Longrightarrow & \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{left}(\Pi)} & \approx_{tic} & \mathsf{Exec}_{\mathbf{M},\mathbf{A},\mathsf{right}(\Pi)} \end{array}$$

□

5.7 M allows for self-monitoring

In the previous sections, we have shown that there are distinguishing self-monitors for branching (`bad-branch`) and knowledge leaks (`bad-knowledge`). Hence, by Theorem 1 we conclude that $\mathbf{M}$ allows for self-monitoring.

Theorem 4. *Let* $\mathbf{M}$ *be the symbolic model from Section 5.1,* P *be the class of uniformity-enforcing of randomness-safe bi-protocols, and Imp an implementation that satisfies the conditions from Section 5.2. Then,* $\mathbf{M}, Imp, \mathrm{P}$ *allow for self-monitoring. In particular, for each bi-protocol* Π*,* $f_{\texttt{bad-knowledge},\Pi}$ *and* $f_{\texttt{bad-branch},\Pi}$ *as described above are distinguishing self-monitors (see Definition 23) for* $\mathbf{M}$ *and* P*.*

Proof. By construction of the self-monitors (see Definition 22) and the construction of the family of self-monitors $f_{\texttt{bad-knowledge},\Pi}$ and $f_{\texttt{bad-branch},\Pi}$ (see Section 5.5.1 and 5.6.1), and by inspection of the protocol conditions for the class P of uniformity-enforcing (see Definition 21) randomness-safe CoSP bi-protocols (see Definition 26), we can see that for all $\Pi \in \mathrm{P}$, $\mathsf{Mon}(\Pi) \in \mathrm{P}'$, where P' is the class of randomness-safe CoSP protocols (see Definition 26). By Lemma 6 and Lemma 9, we know that for any $\Pi \in \mathrm{P}$, each member of the family of branching monitors $f_{\texttt{bad-branch},\Pi}$ satisfies symbolic and computational self-monitoring. By Lemma 10 and Lemma 23, we know that for any $\Pi \in \mathrm{P}$, each member of the family of branching monitors $f_{\texttt{bad-knowledge},\Pi}$ satisfies symbolic and computational self-monitoring. □

5.8 CS for uniform bi-processes in the applied π-calculus

Combining our results, we conclude CS for protocols in the applied π-calculus that use signatures, public-key encryption, and corresponding length functions.

Theorem 24 (CS for encryption and signatures in the applied π-calculus) *Let* $\mathbf{M}$ *be as defined in Section 5. Let* Q *be a randomness-safe bi-process in the applied* π*-calculus, and let* $\mathbf{A}$ *of* $\mathbf{M}$ *be an implementation that satisfies the conditions from above. Let* e *be the embedding from bi-processes in the applied* π*-calculus to CoSP bi-protocols. If* Q *is uniform, then* $\mathsf{left}(e(Q)) \approx_c \mathsf{right}(e(Q))$*.*

Proof. By Theorem 4, there are for each bi-protocol Π distinguishing self-monitors $f_{\texttt{bad-knowledge},\Pi}$ and $f_{\texttt{bad-branch},\Pi}$ for $\mathbf{M}$. The class of the embedding of the applied π-calculus is uniformity-enforcing by Lemma 2; thus, Theorem 1 entails the claim. □

6 Conclusion

In this work, we provided the first result that allows to leverage existing CS results for trace properties to CS results for uniformity of bi-processes in the applied π-calculus. Our result, which is formulated in an extension of the CoSP framework to equivalence properties, holds for Dolev-Yao models that fulfill the property that all distinguishing computational tests are expressible as a process on the model. We exemplified the usefulness of our method by applying it to a Dolev-Yao model that captures signatures and public-key encryption.

We moreover discussed how computationally sound, automated analyses can still be achieved in those frequent situations in which ProVerif does not manage to terminate whenever the Dolev-Yao model supports a length function. We propose to combine ProVerif with the recently introduced tool APTE [20].

Our results are formulated in the CoSP framework, which decouples the CS of Dolev-Yao models from the calculi, such as the applied π-calculus. We extended this framework with a notion of CS for equivalence properties, which might be of independent interest. Moreover, we proved the existence of an embedding from the applied π-calculus to CoSP that preserves uniformity of bi-processes, using a slight variation of the embedding of the original CoSP paper.

We leave as a future work to prove for more comprehensive Dolev-Yao models (e.g., for zero-knowledge proofs) the sufficient conditions for deducing from CS results for trace properties the CS of uniformity. Another interesting direction for future work is the extension of our result to observational equivalence properties that go beyond uniformity.

Appendix

A Equivalence notions

ProVerif is not able to handle complicated destructors such as *length*, which are typically defined recursively, e.g., $\mathit{length}(\mathit{pair}(t_1, t_2)) = \mathit{length}(t_1) + \mathit{length}(t_2)$. Recent work by Cheval, Cortier, and Plet [20] extends the automated protocol verifier APTE [17, 18], which is capable of proving trace equivalence of two processes in the applied π-calculus (without replication), to support such length functions. However, trace equivalence is a weaker notion than uniformity, i.e., there are bi-processes that are trace equivalent but not uniform, our computational soundness result does not carry over to APTE.[19] Due to the lack of a tool that is able to check (only) uniformity as well as to handle length functions properly, we explain how APTE can be combined with ProVerif to make protocols on the symbolic model of our case study amendable to automated verification.

In [20], the notion of trace equivalence w.r.t. length is defined. Particularized for a bi-process Q, the definition states essentially that Q is trace-equivalent w.r.t. length if for any sequence of input or output actions that an attacker can reach with $\mathsf{left}(Q)$, the same sequence is reachable in $\mathsf{right}(Q)$ such that the resulting attacker knowledge is statically equivalent, and vice versa. Here, static equivalence means that the attacker has no test (built from destructors and constructors) that tells the knowledge reached in $\mathsf{left}(Q)$ apart from the knowledge reached in $\mathsf{right}(Q)$. This holds in particular for the test that returns the length associated with a term in the knowledge of the attacker. We refer the reader to [20] for a precise definition. Since any *length* destructor as introduced in the symbolic model $\mathbf{M}$ in Section 5 is linear, it can be regarded as the mentioned length test. Consequently, the analysis of APTE leads to guarantees that are meaningful with respect to the established computational soundness result.

On the other hand, ProVerif is able to decide whether the following holds on a input bi-process Q:

1. if $Q' \equiv K'[\overline{N}\langle M\rangle.P \mid N'(x).R]$ then $\mathsf{left}(N) = \mathsf{left}(N')$ if and only if $\mathsf{right}(N) = \mathsf{right}(N')$,
2. if $Q' \equiv K'[\mathit{let}\ x = d\ \mathit{in}\ P\ \mathit{else}\ R]$ then $\mathit{eval}(\mathsf{left}(d)) = \bot$ if and only if $\mathit{eval}(\mathsf{right}(d)) = \bot$.

In this case, we say that Q is *PV-uniform*. Blanchet, Abadi, and Fournet [14] prove that PV-uniformity implies uniformity.

Our goal is to show that if a bi-process Q is PV-uniform w.r.t. $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$ with $\mathbf{D}'_{pi} = \mathbf{D}_{pi} \setminus \{\mathit{length}/1\}$, and its left and right variant are trace equivalent w.r.t. $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$, then it is also PV-uniform w.r.t $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$. To this end, we prove the following slightly more general lemma.

Lemma 25 *Let Q be a bi-process in the applied π-calculus and $\mathbf{C}_{pi}$ be a set of constructors and $\mathbf{D}_{pi}$ be a set of destructors. If*

- (i) *Q only uses the constructors $\mathbf{C}_{pi}$ and the destructors $\mathbf{D}'_{pi} \subseteq \mathbf{D}_{pi}$,*
- (ii) *Q is PV-uniform with $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$, and*
- (iii) *for all $d \in \mathbf{D}_{pi} \setminus \mathbf{D}'_{pi}$ and for all terms t, the existence of a term t' such that $\mathit{eval}(d(t)) = t'$ implies that t' can be built using constructors only, and*
- (iv) *$\mathsf{left}(Q)$ and $\mathsf{right}(Q)$ are trace equivalent,*

then Q is PV-uniform against an attacker that uses $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$.

Proof. Assume towards contradiction that Q is not PV-uniform against an attacker that uses $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$, but Q is PV-uniform against an attacker that uses $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$, Q only uses $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$, and assume that $\mathsf{left}(Q)$ and $\mathsf{right}(Q)$ are trace equivalent.

If Q is not PV-uniform (against an attacker that uses $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$), then there are evaluation contexts K, K' and a bi-process Q' such that $K[Q] \rightarrow^* Q'$ and one of the following two properties are violated:

1. if $Q' \equiv K'[\overline{N}\langle M\rangle.P \mid N'(x).R]$ then $\mathsf{left}(N) = \mathsf{left}(N')$ if and only if $\mathsf{right}(N) = \mathsf{right}(N')$,
2. if $Q' \equiv K'[\mathit{let}\ x = d\ \mathit{in}\ P\ \mathit{else}\ R]$ then $\mathit{eval}(\mathsf{left}(d)) = \bot$ if and only if $\mathit{eval}(\mathsf{right}(d)) = \bot$.

[19] Cheval, Cortier, and Delaune [19] show that trace equivalence implies observational equivalence for so-called determinate processes. Recall that the even stronger notion of uniformity, which is only defined for bi-processes, was introduced as a means to prove observational equivalence.

W.l.o.g. we can assume that Q' is the first process in the reduction sequence $K[Q] \to^* Q'$ that violates the PV-uniformity-properties.

Case 1: In this case, we have w.l.o.g. $\mathsf{left}(N) = \mathsf{left}(N')$ but $\mathsf{right}(N) \neq \mathsf{right}(N')$. Since Q' is the first process that violates the PV-uniformity properties, 1 and 2 hold in all previous reduction steps. We distinguish two cases. First, *the protocol communicates with the attacker*, i.e., either $\overline{N}\langle M\rangle.P$ or $N'(x).R$ is not a residue process of the protocol Q. Then, the trace of $\mathsf{left}(Q)$ and $\mathsf{right}(Q)$ shows whether the communication over N and N' succeeds. Consequently, $\mathsf{right}(N) \neq \mathsf{right}(N')$ contradicts the trace equivalence of $\mathsf{left}(Q)$ and $\mathsf{right}(Q)$, i.e., assumption (iv).

Second, *the protocol performs an internal communication*, i.e., both $\overline{N}\langle M\rangle.P$ and $N'(x).R$ are residues processes of the protocol Q. In this case, we know by assumption (iii) that the attacker context K (which uses $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$) together with the protocol process Q cannot produce more terms than a corresponding context K'' that only uses $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$ together with Q. Even though K with Q can branch differently than K'' with Q, we know by assumption that all previous reduction steps satisfied the PV-uniformity property 2. Hence, there is a context K'' that only uses $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$ but has a reduction sequence $K'' \to^* Q''$ such that

$$Q'' \equiv K'''[\overline{N}\langle M\rangle.P \mid N'(x).R]$$

Then, however, $\mathsf{right}(N) \neq \mathsf{right}(N')$ contradicts the PV-uniformity of Q against an attacker that only uses $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$, i.e., assumption (ii).

Case 2: In this case, we assume w.l.o.g. that $eval(\mathsf{left}(d)) = \bot$ and $eval(\mathsf{right}(d)) \neq \bot$. We distinguish two cases. First, *a protocol test fails in* $\mathsf{left}(Q)$ *but not in* $\mathsf{right}(Q)$, i.e.,

$$let\ x = d\ in\ P\ else\ R$$

is inside a residue process of the protocol Q. In this case, (with the same argumentation as above) there is, by assumption iii, a context K'' that only uses $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$ but has a reduction sequence $K'' \to^* Q''$ such that

$$Q'' = K'''[let\ x = d\ in\ P\ else\ R]$$

$eval(\mathsf{left}(d)) = \bot$ and $eval(\mathsf{right}(d)) \neq \bot$, however, contradicts the PV-uniformity of Q against attackers that only use $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$, i.e., assumption (ii).

Second, *the attacker distinguishes* $\mathsf{left}(Q)$ *and* $\mathsf{right}(Q)$ *with the test* d, i.e.,

$$let\ x = d\ in\ P\ else\ R$$

is not inside a residue process of the protocol Q and $eval(\mathsf{left}(d)) = \bot$ and $eval(\mathsf{right}(d)) \neq \bot$. Then, $eval(\mathsf{left}(d)) = \bot$ and $eval(\mathsf{right}(d)) \neq \bot$ breaks the static equivalence property that the trace equivalence requires, i.e., assumption (iv). □

The lemma allows us to prove that the combined analysis using ProVerif and APTE is sound.

Theorem 26 *Let $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$ be the symbolic model defined in Section 5. Let Q be a bi-process in the applied π-calculus that uses only $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$ with $\mathbf{D}'_{pi} = \mathbf{D}_{pi} \setminus \{length\}$. If ProVerif proves uniformity for Q with $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$ and APTE proves trace-equivalence w.r.t. length with $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$, then Q is uniform with $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$.*

Proof. The theorem follows from Lemma 25. Condition (iii) of the lemma is satisfied, because the destructor *length* outputs only length specifications, which can be built using the constructors O and S. The other conditions hold by assumption. □

Note. Assumption (iii) in Lemma 25, i.e., that the destructors in $\mathbf{D}_{pi} \setminus \mathbf{D}'_{pi}$ do not allow the attacker to learn new terms is essential, because otherwise the following counter-example breaks the uniformity of Q with $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$:

$$\begin{aligned} Q :=\ & \overline{c}\langle hidden(magic)\rangle.c(x). \\ & \quad let\ y = equals(x, magic)\ in \\ & \quad\quad let\ z = equals(a, choice[a, b]) \\ & \quad\quad\quad in\ 0\ else\ 0 \\ & \quad else\ P \\ K[\bullet] :=\ & \nu c.\big(c(x).let\ y = showme(hidden(x))\ in\ \overline{c}\langle y\rangle \mid \bullet\big) \end{aligned}$$

Here, $showme(hidden(m)) = m$ and $showme \in \mathbf{D}_{pi} \setminus \mathbf{D}'_{pi}$.

References

1. Backes, M., Hritcu, C., Maffei, M.: Automated Verification of Remote Electronic Voting Protocols in the Applied Pi-Calculus. In: CSF, pp. 195–209. IEEE (2008)
2. Backes, M., Bendun, F., Unruh, D.: Computational Soundness of Symbolic Zero-knowledge Proofs: Weaker Assumptions and Mechanized Verification. In: POST, pp. 206–225. Springer (2013)
3. Backes, M., Hofheinz, D., Unruh, D.: CoSP: A General Framework for Computational Soundness Proofs. In: CCS, pp. 66–78. ACM (2009)
4. Backes, M., Jacobi, C., Pfitzmann, B.: Deriving Cryptographically Sound Implementations Using Composition and Formally Verified Bisimulation. In: FME, pp. 310–329. Springer (2002)
5. Backes, M., Laud, P.: Computationally Sound Secrecy Proofs by Mechanized Flow Analysis. In: Proc. 13th ACM Conference on Computer and Communications Security, pp. 370–379. ACM (2006)
6. Backes, M., Maffei, M., Mohammadi, E.: Computationally Sound Abstraction and Verification of Secure Multi-Party Computations. In: (FSTTCS), pp. 352–363. Schloss Dagstuhl (2010)
7. Backes, M., Maffei, M., Unruh, D.: Computationally Sound Verification of Source Code. In: CCS, pp. 387–398. ACM (2010)
8. Backes, M., Maffei, M., Unruh, D.: Zero-Knowledge in the Applied Pi-calculus and Automated Verification of the Direct Anonymous Attestation Protocol. In: S&P, pp. 202–215. IEEE (2008)
9. Backes, M., Malik, A., Unruh, D.: Computational Soundness without Protocol Restrictions. In: CCS, pp. 699–711. ACM (2012)
10. Backes, M., Pfitzmann, B.: Symmetric Encryption in a Simulatable Dolev-Yao Style Cryptographic Library. In: CSFW, pp. 204–218. IEEE (2004)
11. Backes, M., Pfitzmann, B., Waidner, M.: A Composable Cryptographic Library with Nested Operations (Extended Abstract). In: CCS, pp. 220–230 (2003)
12. Backes, M., Unruh, D.: Computational soundness of symbolic zero-knowledge proofs. J. of Comp. Sec. 18(6), 1077–1155 (2010)
13. Blanchet, B., Abadi, M., Fournet, C.: Automated Verification of Selected Equivalences for Security Protocols. In: LICS, pp. 331–340 (2005)
14. Blanchet, B., Abadi, M., Fournet, C.: Automated Verification of Selected Equivalences for Security Protocols. Journal of Logic and Algebraic Programming 75, 3–51 (2008)
15. Böhl, F., Cortier, V., Warinschi, B.: Deduction Soundness: Prove One, Get Five for Free. In: CCS, (2013)
16. Canetti, R., Herzog, J.: Universally Composable Symbolic Security Analysis. Journal of Cryptology 24(1), 83–147 (2011)
17. Cheval, V.: APTE (Algorihm for Proving Trace Equivalence). URL: projects.lsv.ens-cachan.fr/APTE/
18. Cheval, V., Comon-Lundh, H., Delaune, S.: Trace Equivalence Decision: Negative Tests and Non-determinism. In: CCS, pp. 321–330. ACM (2011)
19. Cheval, V., Cortier, V., Delaune, S.: Deciding equivalence-based properties using constraint solving. Theoretical Computer Science 492 (2013)
20. Cheval, V., Cortier, V., Plet, A.: Lengths May Break Privacy – Or How to Check for Equivalences with Length. In: CAV, pp. 708–723. Springer (2013)
21. Comon-Lundh, H., Cortier, V.: Computational soundness of observational equivalence. In: CCS, pp. 109–118. ACM (2008)
22. Comon-Lundh, H., Cortier, V., Scerri, G.: Security Proof with Dishonest Keys. In: POST, pp. 149–168. Springer (2012)
23. Comon-Lundh, H., Hagiya, M., Kawamoto, Y., Sakurada, H.: Computational soundness of indistinguishability properties without computable parsing. In: ISPEC, pp. 63–79. Springer (2012)
24. Cortier, V., Kremer, S., Küsters, R., Warinschi, B.: Computationally Sound Symbolic Secrecy in the Presence of Hash Functions. In: FSTTCS, pp. 176–187 (2006)
25. Cortier, V., Warinschi, B.: A composable computational soundness notion. In: CCS, pp. 63–74. ACM (2011)
26. Cortier, V., Warinschi, B.: Computationally Sound, Automated Proofs for Security Protocols. In: Proc. 14th European Symposium on Programming (ESOP), pp. 157–171 (2005)
27. Cortier, V., Wiedling, C.: A formal analysis of the Norwegian E-voting protocol. In: POST, pp. 109–128. Springer (2012)
28. Delaune, S., Kremer, S., Ryan, M.: Verifying privacy-type properties of electronic voting protocols. J. Comput. Secur. 17(4), 435–487 (2009)
29. Delaune, S., Kremer, S., Ryan, M.D., Steel, G.: Formal Analysis of Protocols Based on TPM State Registers. In: CSF, pp. 66–80. IEEE (2011)
30. Dolev, D., Yao, A.C.: On the Security of Public Key Protocols. IEEE Transactions on Information Theory 29(2), 198–208 (1983)
31. Even, S., Goldreich, O.: On the Security of Multi-Party Ping-Pong Protocols. In: FOCS, pp. 34–39. IEEE (1983)
32. Galindo, D., Garcia, F.D., Van Rossum, P.: Computational soundness of non-malleable commitments. In: ISPEC, pp. 361–376. Springer (2008)

33. Janvier, R., Lakhnech, Y., Mazaré, L.: Completing the Picture: Soundness of Formal Encryption in the Presence of Active Adversaries. In: ESOP, pp. 172–185. Springer (2005)
34. Kemmerer, R., Meadows, C., Millen, J.: Three Systems for Cryptographic Protocol Analysis. Journal of Cryptology 7(2), 79–130 (1994)
35. Micciancio, D., Warinschi, B.: Soundness of Formal Encryption in the Presence of Active Adversaries. In: TCC, pp. 133–151. Springer (2004)
36. Sprenger, C., Backes, M., Basin, D., Pfitzmann, B., Waidner, M.: Cryptographically Sound Theorem Proving. In: CSFW, pp. 153–166. IEEE (2006)
37. Unruh, D.: Programmable encryption and key-dependent messages. IACR ePrint Archive: 2012/423 (2012)
38. Unruh, D.: Termination-Insensitive Computational Indistinguishability (and Applications to Computational Soundness). In: CSF, pp. 251–265. IEEE (2011)