Computationally Sound Verification of Source Code

Michael Backes
Saarland University, MPI-SWS

Matteo Maffei
Saarland University

Dominique Unruh
Saarland University

July 26, 2010

Abstract

Increasing attention has recently been given to the formal verification of the source code of cryptographic protocols. The standard approach is to use symbolic abstractions of cryptography that make the analysis amenable to automation. This leaves the possibility of attacks that exploit the mathematical properties of the cryptographic algorithms themselves. In this paper, we show how to conduct the protocol analysis on the source code level (F# in our case) in a computationally sound way, i.e., taking into account cryptographic security definitions.

We build upon the prominent F7 verification framework (Bengtson et al., CSF 2008) which comprises a security type-checker for F# protocol implementations using symbolic idealizations and the concurrent lambda calculus RCF to model a core fragment of F#.

To leverage this prior work, we give conditions under which symbolic security of RCF programs using cryptographic idealizations implies computational security of the same programs using cryptographic algorithms. Combined with F7, this yields a computationally sound, automated verification of F# code containing public-key encryptions and signatures.

For the actual computational soundness proof, we use the CoSP framework (Backes, Hofheinz, and Unruh, CCS 2009). We thus inherit the modularity of CoSP, which allows for easily extending our proof to other cryptographic primitives.

Contents

1 Introduction

Proofs of security protocols are known to be error-prone and, owing to the distributed-system aspects of multiple interleaved protocol runs, difficult for humans to generate. Hence, work towards the automation of such proofs started soon after the first protocols were developed. From the start, the actual cryptographic operations in such proofs were idealized into so-called symbolic or Dolev-Yao models, following [DY83, EG83, Mer83] (see, e.g., [KMM94, Sch96, AG97, Low96, Pau98, BMV04]). This idealization simplifies proofs by freeing them from cryptographic details such as computational restrictions, probabilistic behavior, and error probabilities. Unfortunately, these idealizations also abstract away from the algebraic properties a cryptographic algorithm may exhibit. Therefore a symbolic analysis may overlook attacks based on these properties. In other words, symbolic security does not imply computational security. In order to remove this limitation, [AR02] introduced the concept of computational soundness. We call a symbolic abstraction computationally sound when symbolic security implies computational security. A computational soundness result allows us to get the best of two worlds: The analysis can be performed (possibly automatically) using symbolic abstractions, but the final results hold with respect to the realistic security models used by cryptographers.

A drawback common to the existing computational soundness results, is, however, that they work on abstract protocol representations (e.g., the applied π-calculus [AF01]). That is, although the analysis takes into account the actual cryptographic algorithms, it still abstracts away from the actual protocol implementation. Thus, even if we prove the protocol secure, the implementation that is later deployed may contain implementation errors that introduce new vulnerabilities. To avoid this issue, recent work has tackled the problem of verifying security directly on the source code, e.g., [GLP05a, BFGT06, BBF$^+$08]. Yet, this verification is again based on symbolic idealizations.

Thus, we are left with the choice between verification techniques that abstract away from the cryptographic algorithms, and verification techniques that abstract from the protocol implementation. To close this gap, we need a computational soundness result that applies directly to protocol implementations.

Our result. We present a computational soundness result for F# code. For this, we use the RCF calculus proposed by [BBF$^+$08] as semantics for (a core fragment of) F#. RCF allows for encoding implementation in F# by offering a lambda-abstraction constructor that allows for reasoning about higher-order languages. Moreover, it supports concurrency primitives, inductive datastructures, recursion, and an expressive treatment of symbolic cryptography using sealing mechanisms. Furthermore, RCF supports very general trace-based security properties that are expressed in first-order logic, using assumptions and assertions. (Previous computational soundness results are restricted to calculi like the applied π-calculus which lack these features.) We specify a cryptographic library that internally uses symbolic abstractions, and prove that if a protocol is symbolically secure when linked to that library, it is computationally secure when using actual cryptographic algorithms. Our approach enables the use of existing symbolic verification tools, such as the type-checker F7 [BBF$^+$08]. The requirement to use these tools in particular ruled out potential changes to the RCF semantics that would have simplified to establish a computational soundness result. We stress, however, that our result does not depend on any particular symbolic verification technique.

We have derived computational soundness for encryptions and digital signatures. Our result is, however, extensible: most of our theorems are parametric in the set of cryptographic primitives and the remaining theorems can be easily extended. Furthermore, by basing on the so-called CoSP framework [BHU09], our proof solely concerns the semantics of RCF programs and does not involve any cryptographic arguments; thus extending our proofs to additional cryptographic abstractions supported by CoSP does not require a deep knowledge of cryptography, which makes such an extension accessible to a more general audience.

1.1 Our techniques

CoSP (Section 3). The main idea of our work is to reduce computational soundness of RCF to computational soundness in the CoSP framework [BHU09]. Thus, we first give an overview of the ideas underlying CoSP. All definitions in CoSP are relative to a *symbolic model* that specifies a set of constructors and destructors that symbolically represent computational operations, and a *computational implementation* that specifies cryptographic algorithms for these constructors and destructors. In CoSP, a protocol is represented by an infinite tree that describes the protocol as a labeled transition system. Such a CoSP protocol contains actions for performing abstract computations (applying constructors and destructors to messages) and for communicating with an adversary. A CoSP protocol is endowed with two semantics, a symbolic execution and a computational execution. In the symbolic execution, messages are represented by terms. In the computational execution, messages are bitstrings, and the computational implementation is used instead of applying constructors and destructors. A computational implementation is *computationally sound* if any symbolically secure CoSP protocol is also computationally secure. The advantage of expressing computational soundness results in CoSP is that the protocol model in CoSP is very general. Hence the semantics of other calculi can be embedded therein, thus transferring the computational soundness results from CoSP to these calculi.

DY library (Sections 4, 5). To apply CoSP to RCF, we first define a library $\sigma_{\mathrm{DY}}^{\mathsf{M}}$ that encodes an arbitrary symbolic model. This library internally represents all messages as terms in some datatype. Manipulation of these terms is possible only through the library, neither the program nor the adversary can directly manipulate messages. $\sigma_{\mathrm{DY}}^{\mathsf{M}}$ also provides functions for sending and receiving messages. Given the library $\sigma_{\mathrm{DY}}^{\mathsf{M}}$, we can define a notion of symbolic security. A program A contains certain events and security policies specified in first-order logic. We call *A robustly $\rightarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-safe* if the security policies are satisfied in every step of the execution when A runs in parallel with an arbitrary opponent and is linked to the library $\sigma_{\mathrm{DY}}^{\mathsf{M}}$.

Next, we specify a probabilistic computational semantics for RCF programs A. In these semantics, we specify an algorithm (the *computational RCF-execution*) that executes A. In each step of the execution, the adversary is asked what reduction rule to apply to A. Letting the adversary make these scheduling decisions resolves the non-determinism in the RCF program and simultaneously makes our result stronger by making the worst-case assumption that the adversary has total control over the scheduling. All messages are represented as bitstrings, and any invocation of $\sigma_{\mathrm{DY}}^{\mathsf{M}}$ is replaced by the corresponding computation from the computational implementation. Notice that in the computational RCF-execution, the adversary is not limited to invoking library routines; since messages are bitstrings, the adversary can perform arbitrary polynomial-time operations on them. If all security policies are satisfied in each step of the computational RCF-execution, we call *A robustly computationally safe.*

Our goal is to show that, if an RCF program is robustly $\rightarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-safe, then it is robustly computationally safe. To prove this, we introduce two intermediate semantics.

- The reduction relation $\rightsquigarrow$: This semantics is very similar to the original semantics of RCF, except that all invocations of $\sigma_{\mathrm{DY}}^{\mathsf{M}}$ are internalized, i.e., symbolic cryptographic operations are atomic operations with respect to $\rightsquigarrow$. This leads to the notion of *robust $\rightsquigarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-safety.*
- The *symbolic RCF-execution* SExec: This semantics is defined by taking the definition of the computational RCF-execution, and by replacing all computational operations by the corresponding symbolic operations. That is, the symbolic and the computational RCF-execution are essentially the same algorithm, one operating on terms, the other doing the corresponding operations from the computational implementation. This leads to the notion of *robust* SExec-*safety.*

In the first step (cf. Figure 1), we show that robust $\rightarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-safety implies robust $\rightsquigarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-safety. This proof is fairly straightforward, because $\rightsquigarrow$ just internalizes the definition of $\sigma_{\mathrm{DY}}^{\mathsf{M}}$.

In the second step, we show that robust $\rightsquigarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-safety implies robust SExec-safety. The first technical difficulty here lies in the fact that robust $\rightsquigarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-safety is defined with respect to adversaries that are expressed as RCF-programs and that run interleaved with the program A, while robust SExec-safety models the adversary as an external non-deterministic entity. Thus, for any possible behavior of the

$$\begin{tikzcd}
\text{rob. } \rightarrow\text{-}\sigma^{\mathsf{M}}_{\mathrm{DY}}\text{-safety} \arrow[r, "1."] & \text{rob. } \rightsquigarrow\text{-}\sigma^{\mathsf{M}}_{\mathrm{DY}}\text{-safety} \arrow[d, "2."] \\
\text{rob. computational safety} & \text{rob. SExec-safety} \arrow[l, "3."']
\end{tikzcd}$$

Figure 1: Main steps of the computational soundness proof

SExec-adversary, we have to construct an RCF-program Q that performs the same actions when running in parallel with A. The second difficulty lies in the fact that the logic for describing security properties is quite general. In particular, it allows for expressing facts about the actual code of $\sigma^{\mathsf{M}}_{\mathrm{DY}}$ (e.g., the code of one function is a subterm of the code of another). Since the library $\sigma^{\mathsf{M}}_{\mathrm{DY}}$ is not present in the symbolic RCF-execution, we need to identify criteria that ensure that the policies do not depend on the actual code of $\sigma^{\mathsf{M}}_{\mathrm{DY}}$.

In the third step, we use the fact that the symbolic and the computational RCF-execution of A have essentially the same definition, except that one performs symbolic and the other computational operations. Thus, if we express these executions by a labeled transition system that treats operations on messages as atomic steps, we get the same transition system for both executions, only with a different interpretation of these atomic steps. This transition system is a CoSP protocol Π_A, and the symbolic and the computational execution of that protocol are equivalent to the symbolic and the computational RCF-execution of A. Thus, assuming a computational soundness result in CoSP, we get that robust SExec-safety implies robust computational safety. Combining this with the previous steps, we have that robust $\rightarrow$-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safety implies robust computational safety (Theorem 1).

Note that this argumentation is fully generic, it does not depend on any particular symbolic model. Once we have a new computational soundness result in CoSP, this directly translates into a result for RCF. Note further that no actual cryptographic proofs need to be done; all cryptographic details are outsourced to CoSP. The library $\sigma^{\mathsf{M}}_{\mathrm{DY}}$ is very similar in spirit to the one used in [BFG10], we believe that the verification techniques used there can be applied to robust $\rightarrow$-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safety as well.

Encryption and signatures (Section 5.4). Our results so far are fully generic. In the CoSP framework, a computational soundness result exists for public-key encryption and signatures. Combining this result with our generic result, we get a self-contained computational soundness result for encryptions and signatures in RCF (Theorem 2). The result in the CoSP framework imposes certain restrictions on the use of the cryptographic primitives (e.g., one is not allowed to send secret keys around). To ensure that these restrictions are met, we introduce a wrapper-library $\sigma_{\mathit{Highlevel}}$ for $\sigma^{\mathsf{M}}_{\mathrm{DY}}$. A program that only invokes functions from $\sigma_{\mathit{Highlevel}}$ is guaranteed to satisfy these restrictions.

Sealing-based library (Section 6). In the library $\sigma^{\mathsf{M}}_{\mathrm{DY}}$, we have internally represented symbolic cryptography as terms in some datatype. An alternative approach is used in the F7 verification framework [BBF$^+$08] for analyzing RCF/F#-code. In this approach, a library based on seals is used. Roughly, a seal consists of a mutable reference and accessor functions. An encryption key pair, e.g., is modeled as a sealed map. The encryption key is a function that inserts the plaintext into that map and returns the index of the plaintext. The decryption key is a function that retrieves the plaintext given the index. Seals have proven well-suited for security analysis by type-checking, since they allow for polymorphic types. We present a sealing-based library σ_{S} modeling encryptions and signatures. We show that robust safety with respect to σ_{S} implies robust $\rightsquigarrow$-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safety by proving the existence of a simulation between executions with respect to the two libraries. Combined with Theorem 2, this immediately returns a computational soundness result for the sealing-based library (Theorem 4). The advantage of this result is that programs using σ_{S} can be analyzed using the F7 type-checker, since the library itself is type-checked with polymorphic typing annotations[1].

Note that this part of our paper is specific to the case of encryptions and signatures. We believe, however, that the proof can be easily extended to other primitives on a case-by-case basis. Furthermore

[1]The F# code of the library with typing annotations is available at [BMU].

our proof also gives an additional justification to the approach of seals: We reduce security with respect to seals to security with respect to a term-based abstraction that is considerably simpler because it does not rely on a shared state.

Restrictions. We briefly discuss the limitations of our result and explain why they are present.

Security properties. We only consider safety properties (described by authorization policies) that are efficiently decidable (in the sense that for any given trace, it is efficiently decidable whether the safety property is fulfilled). Both the restriction to safety properties (as opposed to liveness properties) and the restriction to efficiently decidable properties[2] are state of the art in computational soundness results. Computational soundness results for properties based on observational equivalence exist [CLC08]; applying these to RCF would constitute an interesting extension to our work.

Protocol conditions. We impose certain conditions on our protocols. Most prominently, we forbid to encrypt or send secret keys. (As a side effect, this also avoids so-called key-cycles.) Again, these conditions are state of the art in computational soundness results, and, if removed there, they can also be removed from our result.

Authorization policies. Constructors that represent cryptographic operations (such as encryptions) may not occur in the formulae used to express authorization properties. This is due to the fact that a statement such as $\exists xyz.c = \mathsf{enc}(x, y, z)$ does not have a sensible computational interpretation (there is no efficient way to check it). Since our treatment is generic, also constructors that represent "harmless" primitives such as pairs are excluded from authorization policies; allowing them should be possible but would considerably complicate our treatment. We believe, however, that disallowing these constructors in authorization policies does not constitute a big restriction. In most cases, an authorization policy will define high-level rules (such as "if P has paid for x, then P may download x"). Statements about the actual format of messages (e.g., "m is a pair") will only be used during the symbolic verification of the high-level properties, e.g., as part of a refinement type. We do not impose any restrictions on the symbolic verification techniques; arbitrary formulae can be used there as long as they do not appear in the final authorization policy.

Network channels. We assume that there is only a single public network channel (i.e., only a single channel to the adversary). This is done for simplicity only, our results could be easily extended to a setting with more channels. Or, one might emulate several channels by adding a header to all messages sent over the public channel.

Assumptions and assertions in libraries. One is not allowed to add assumptions and assertions (i.e., authorization policies) in the code of the symbolic libraries themselves. This is, however, not really a restriction since one may use a wrapper library that adds these assumptions and assertions.

Alternative approaches. We briefly discuss several possible alternatives to our approach and explain their difficulties.

Using CryptoVerif. Instead of doing a symbolic security verification and then applying a computational soundness result, one could perform the analysis directly in the computational setting using a tool such as CryptoVerif [Bla06]. CryptoVerif is a tool that performs a security analysis directly in the computational model. To follow this approach in our setting, one would have to describe an encoding of RCF into CryptoVerif's calculus. Although this can easily be done for a fragment of RCF, many features of RCF such as recursion, authorization policies in first-order logic, and concurrency[3] are probably beyond what CryptoVerif can handle. Also, CryptoVerif's approach probably does not scale well to complex programs. Finally, one needs to prove that the encoding of RCF into CryptoVerif preserves all required security properties; such a proof might be not much simpler than the proofs in the present paper. [BCFZ08] pursue this approach; they do not, however, prove their encoding sound.

[2]By efficiently decidable properties, we do not mean that it can be efficiently decided whether a protocol guarantees that the property is satisfied, we only mean that it can be efficiently decided whether in a given execution, the property was satisfied.

[3]CryptoVerif does support concurrency natively, but its model of concurrency assumes a uniform random choice in each scheduling decision which arguably is an unrealistic assumption in most settings.

Reducing to the applied π-calculus. An alternative approach to obtain computational soundness would be to embed F# into the applied pi-calculus and to exploit the computational soundness result for the applied pi-calculus established in [BHU09]. However, establishing this embedding would arguably not be easier than our approach: it requires to encode datastructures, recursion, the sealing mechanism, and assertions/assumptions into the applied pi-calculus, including the whole FOL/F logic. Moreover, the correctness of the encoding has to be proven twice – once symbolically (the proof would follow the same lines as the proof in [BFGT06]) and once with respect to the computational semantics.

Removing equality tests on lambda-expressions. A large part of the technical difficulties in our proofs stem from the fact that RCF allows to do syntactic equality tests on lambda-abstractions. It is not, however, easily possible to remove these tests: If we change the semantics of RCF, our results become incompatible with existing tools like the F7 framework. A syntactic restriction that disallows comparisons of lambda-abstractions does not seem to be possible either; which variables are instantiated with lambda-abstractions only becomes clear at runtime.

1.2 Related work

The problem of computational soundness was first addressed by Abadi and Rogaway in [AR02] for passive adversaries and symmetric encryption. The protocol language and security properties handled there were extended in [AJ01, Lau01, HLM03, BCK05, ABW06], but still apply only to passive adversaries. Subsequent works studied computational soundness against active attacks (e.g., cf. [BPW07, BPW03a, BPW03b, BP04, SBB$^+$06, Lau04, MW04, JLM05, BHU09]). Recent works also focused on computational soundness in the sense of observational equivalence of cryptographic realizations of processes (e.g., [AF06, CLC08, CL08]). All these works do not tackle the computational soundness of protocol implementations. Concurrently with the announcement of this work at FCC 2009, [Fou09] reported independent work in progress on a type system for RCF that entails computational soundness.

The analysis of the source code of protocol implementations has recently received increasing attention. Goubault-Larrecq and Parrennes developed a static analysis technique [GLP05b] based on pointer analysis and clause resolution for cryptographic protocols implemented in C. The analysis is limited to secrecy properties. Chaki and Datta recently proposed a technique [CD09] based on software model checking for the automated verification of secrecy and authentication properties of protocols implemented in C. The analysis provides security proofs for a bounded number of sessions and is effective in discovering attacks. It was used to check secrecy and authentication properties of the SSL handshake protocol for configurations of up to three servers and three clients. Bhargavan et al. proposed a technique [BFGT06, BCFZ08] for the verification of F# protocol implementations by automatically extracting ProVerif models [Bla01]. The analysis provides security proofs and, despite its non-compositional nature, scales remarkably well and was successfully used to verify implementations of real-world cryptographic protocols such as TLS [BCFZ08]. None of these analysis techniques enjoys computational soundness guarantees. [BCFZ08] also proposes an embedding of F# into the calculus of CryptoVerif. The embedding is not, however, proven to be sound; also, according to [BCFZ08], it is difficult to analyze recursive functions with CryptoVerif.

1.3 Notation

Given a term t, we write $t\{t'/x\}$ for the result of substituting all free occurrences of x by t'. We assume that substitutions are capture avoiding, i.e., bound names are renamed when necessary. We write $\underline{t}$ for a list $t_1, \dots, t_n$ where the length n of the list is left implicit. Given sets P, C of logical formulae, we write $P \vdash C$ iff for all $F \in C$, F is entailed by P.

2 RCF (review)

This section outlines the Refined Concurrent FPC [BBF$^+$08], a simple core calculus extending the Fixed Point Calculus [Gun92] with refinement types and concurrency. Although very simple, this calculus is

a, b, c	name
h	constructor
$M, N ::=$	value
x, y, z	variable
$()$	unit
$\lambda x.A$	function
(M, N)	pair
$h\ M$	constructor application

$A, B ::=$	expression
M	value
$M\ N$	function application
$M = N$	syntactic equality
$\mathsf{let}\ x = A\ \mathsf{in}\ B$	let
$\mathsf{let}\ (x, y) = M\ \mathsf{in}\ A$	pair split
$\mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$	constructor match
$\nu a.A$	restriction
$A \Rsh B$	fork
$a!M$	transmission of M on channel a
$a?$	receive message off channel
$\mathsf{assume}\ F$	assumption of formula F
$\mathsf{assert}\ F$	assertion of formula F

Figure 2: Syntax of RCF values and expressions

Struct Refl	$A \equiv A$
Struct Trans	$A \equiv A''$, if $A \equiv A'$ and $A' \equiv A''$
Struct Let	$\mathsf{let}\ x = A\ \mathsf{in}\ B \equiv \mathsf{let}\ x = A'\ \mathsf{in}\ B$, if $A \equiv A'$
Struct Res	$\nu a.A \equiv \nu a.A'$, if $A \equiv A'$
Struct Fork 1	$A \Rsh B \equiv A' \Rsh B$, if $A \equiv A'$
Struct Fork 2	$B \Rsh A \equiv B \Rsh A'$, if $A \equiv A'$
Struct Fork ()	$() \Rsh A \equiv A$
Struct Msg ()	$a!M \equiv a!M \Rsh ()$
Struct Assume ()	$\mathsf{assume}\ C \equiv \mathsf{assume}\ C \Rsh ()$
Struct Res Fork 1	$A' \Rsh (\nu a.A) \equiv \nu a.A' \Rsh A$, if $a \notin fn(A')$
Struct Res Fork 2	$\nu a.A \Rsh A' \equiv \nu a.A \Rsh A'$, if $a \notin fn(A')$
Struct Res Let	$\mathsf{let}\ x = \nu a.A\ \mathsf{in}\ B \equiv \nu a.\mathsf{let}\ x = A\ \mathsf{in}\ B$, if $a \notin fn(B)$
Struct Fork Assoc	$(A \Rsh A') \Rsh A'' \equiv A \Rsh (A' \Rsh A'')$
Struct Fork Comm	$(A \Rsh A') \Rsh A'' \equiv (A' \Rsh A) \Rsh A''$
Struct Fork Let	$\mathsf{let}\ x = (A \Rsh A')\ \mathsf{in}\ B \equiv A \Rsh (\mathsf{let}\ x = A'\ \mathsf{in}\ B)$

Figure 3: Structural equivalence relation $A \equiv A'$

expressive enough to encode a large part of F# [BBF$^+$08].

2.1 Syntax and semantics

The set of *values* is composed of names, variables, unit, functions, pairs, and type constructors (cf. Figure 2). Names are generated at run-time and are only used as channel identifiers, while variables are place-holders for values. Unit, functions, and pairs are standard. While RCF originally includes only three type constructors (namely, introduction forms for union and recursive types), we extend the syntax of the calculus to an arbitrary set of constructors.

Conditionals are encoded using the following syntactic sugar: $\mathsf{true} := \mathsf{inl}()$, $\mathsf{false} := \mathsf{inr}()$, and $\mathsf{if}\ M = N\ \mathsf{then}\ A\ \mathsf{else}\ B$ abbreviates $\mathsf{let}\ y = (M = N)\ \mathsf{in}\ \mathsf{match}\ y\ \mathsf{with}\ \mathsf{inl}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$ for some fresh x, y.

An *expression* represents a concurrent computation that may reduce to a value, or may diverge. The semantics of expressions is defined by a *structural equivalence relation* $\equiv$[4] and a *reduction relation* $\rightarrow$.

[4]The equivalence relation $\equiv$ considered in this paper is the extension of the heating relation $A \Rrightarrow B$ proposed in [BBF$^+$08] where all heating rules are made symmetric. In Appendix A, we prove that making the heating relation symmetric is sound, i.e., it does not affect the safety of expressions.

Red Fun	$(\lambda x.A)\ N$	$\rightarrow A\{N/x\}$	
Red Split	let $(x_1, x_2) = (N_1, N_2)$ in A	$\rightarrow A\{N_1/x_1\}\{N_2/x_2\}$	
Red Match	match M with $h\ x$ then A else B	$\rightarrow \begin{cases} A\{N/x\} & \text{if } M = h\ N \\ B & \text{otherwise} \end{cases}$	
Red Eq	$M = N$	$\rightarrow \begin{cases} \mathsf{true} & \text{if } M = N \\ \mathsf{false} & \text{otherwise} \end{cases}$	
Red Comm	$a!M \Rsh a?$	$\rightarrow M$	
Red Assert	assert C	$\rightarrow ()$	
Red Let Val	let $x = M$ in A	$\rightarrow A\{M/x\}$	
Red Let	let $x = A$ in B	$\rightarrow$ let $x = A'$ in B,	if $A \rightarrow A'$
Red Res	$\nu a.A$	$\rightarrow \nu a.A'$,	if $A \rightarrow A'$
Red Fork 1	$A \Rsh B$	$\rightarrow A' \Rsh B$,	if $A \rightarrow A'$
Red Fork 2	$B \Rsh A$	$\rightarrow B \Rsh A'$,	if $A \rightarrow A'$
Red Struct	A	$\rightarrow A'$,	if $A \equiv B, B \rightarrow B', B' \equiv A'$

Figure 4: Reduction relation $A \rightarrow A'$

The former enables convenient rearrangements of expressions, while the latter describes the semantics of RCF commands.

Values are irreducible. The semantics of function applications, conditionals, let commands, pair splits, and constructor matches is standard. Intuitively, the restriction $\nu a.A$ generates a globally fresh channel a that can only be used in A and the name a is bound in A. The expression $A \Rsh B$ evaluates A and B in parallel, and returns the result of B (the result of A is discarded). The expression $a!M$ outputs M on channel a and reduces to the unit value $()$. The evaluation of $a?$ blocks until some message M is available on channel a, removes M from the channel, and then returns M.

The expressions assume F and assert F represent logical assumptions and assertions for modeling security policies. The intended meaning is that at any point of the execution, the assertions are entailed by the assumptions. The formulae F are specified in FOL/F [BBF$^+$08], a variant of first-order logic.

More precisely, the formulae F occurring in an RCF-expressions are formulae in the logic FOL/F, a variant of first order logic extended with the concept of *syntactic* function symbols. For syntactic function symbols $f \neq f'$, we have the additional axioms $\vdash (f(\underline{x}) = f(\underline{x}')) \Rightarrow x = x'$ (F Injective) and $\vdash f(\underline{x}) \neq f'(\underline{x}')$ (F Distinct). All syntactic elements of RCF except for variables (e.g., lambda-abstractions, names, constructors) are encoded as (possibly nullary) syntactic function symbols in FOL/F-formulae. RCF-variables are identified with FOL/F-variables. For details, see [BBF$^+$08].

The equivalence relation $\equiv$ introduce a normal form for RCF-expressions, a *structure*. A structure is an expression of the form

$$\mathbf{S} := \nu\underline{a}.\big(\Pi_{i\in[1,m]}\mathsf{assume}\ F_i \Rsh \Pi_{j\in[1,n]}c_j!M_j \Rsh \Pi_{k\in[1,o]}\mathcal{L}_k\{e_k\}\big)$$

where e_k is any expression apart from a let, restriction, fork, message send, or an assumption and $\mathcal{L} := \{\} \mid \mathsf{let}\ x = \mathcal{L}\ \mathsf{in}\ B$. Notice that any expression is structurally equivalent to a structure.

The FOL/F-formulae F_i in $\mathbf{S}$ we call the *active assumptions*, and any FOL/F-formula F with $e_i =$ assert F for some i we call the *active assertions* of $\mathbf{S}$. We can now formalize the fact that the assumptions follow from the assertions in the execution of an RCF expression:

RCF expressions can be transformed by structural equivalence into a normal form, which is called a *structure* and consists of a sequence of restrictions followed by a parallel composition of assumptions, outputs, and lets. These assumptions and the assertions ready to be reduced are called *active.* Intuitively, an expression is safe if all active assertions are entailed by the active assumptions.

Definition 1 ($\rightarrow$-safety) *A structure **S** is* statically safe *iff $P \vdash C$ where P are the active assumptions and C the active assertions of **S**.*

An expression A is $\rightarrow$-safe *if for all structures $\boldsymbol{S}$ such that $A \rightarrow^* \boldsymbol{S}$, we have that $\boldsymbol{S}$ is statically safe.* $\diamond$

When reasoning about implementations of cryptographic protocols, we are interested in the safety of programs executed in parallel with an arbitrary attacker. This property is called *robust safety.*

Definition 2 (Opponents and robust $\rightarrow$-safety) *An expression O is an* opponent *if and only if O is closed and O contains no assertions. A closed expression A is* robustly $\rightarrow$-safe *if and only if the application $O\ A$ is $\rightarrow$-safe for all opponents O.* $\diamond$

The notion of robust $\rightarrow$-safety is the same as the robust safety defined in [BBF$^+$08]. Robust $\rightarrow$-safety can be automatically verified using the F7 type checker.

In the following, we will sometimes need to restrict our attention to programs that only use a certain subset of the set of all constructors. For this, we assume that the set of RCF constructors is partitioned into *public constructors* and *private constructors*. Private constructors are usually used inside a library. Note however, that the semantics of RCF treats private and public constructors in the same way. An RCF expression that does not contain private constructors (neither in constructor applications nor in pattern-matches) is called *pc-free*. We call an RCF-expression A *mpc-free* (for match-private-constructor-free) iff $A = C[h_1 t_1, \dots, h_n t_n]$ where h_i are private constructors and C is a context that does not contain subterms of the form $\mathsf{match}\ \cdot\ \mathsf{with}\ h\ \cdot\ \mathsf{then}\ \cdot\ \mathsf{else}$ for private constructors h. (That is, a mpc-free expression may have pattern matches using private constructors only below private constructors.) We call an RCF-expression *pure* if it does not contain assumptions, assertions, outputs ($M!N$), inputs ($M?$), or forks ($M \upharpoonleft N$).

Furthermore, we call a FOL/F-function symbol *forbidden* if it is the function symbol representing an RCF-lambda-abstraction[5] or a private RCF-constructor.

3 CoSP Framework (review)

The computational soundness proof developed in this paper follows CoSP [BHU09], a general framework for conducting computational soundness proofs of symbolic cryptography and for embedding these proofs into process calculi. CoSP enables proving computational soundness results in a conceptually modular and generic way: every computational soundness proof for a cryptographic abstraction phrased in CoSP automatically holds for all embedded calculi, and the process of embedding process calculi is conceptually decoupled from computational soundness proofs.

CoSP provides a general symbolic model for expressing cryptographic abstractions. We first introduce some central concepts such as constructors, destructors, and deduction relations.

Definition 3 (CoSP terms) *A* constructor f *is a symbol with a (possibly zero) arity. We write $\mathsf{f}/n \in \mathsf{C}$ to denote that C contains a constructor f with arity n. A* nonce n *is a symbol with zero arity. A* message type T over C and N *is a set of terms over constructors C and nonces N. A* destructor d *of arity n, written d/n, over a message type T is a partial map $\mathsf{T}^n \rightarrow \mathsf{T}$. If d is undefined on $\mathsf{t}_1, \dots, \mathsf{t}_n$, we write $\mathsf{d}(\mathsf{t}_1, \dots, \mathsf{t}_n) = \bot$.* $\diamond$

To unify the notations for constructors, destructors, and nonces, we define the partial function $\mathit{eval}_{\mathsf{f}} : \mathsf{T}^n \rightarrow \mathsf{T}$ as follows: If f is a constructor or nonce, $\mathit{eval}_{\mathsf{f}}(\mathsf{t}_1, \dots, \mathsf{t}_n) := \mathsf{f}(\mathsf{t}_1, \dots, \mathsf{t}_n)$ if $\mathsf{f}(\mathsf{t}_1, \dots, \mathsf{t}_n) \in \mathsf{T}$ and $\mathit{eval}_{\mathsf{f}}(\mathsf{t}_1, \dots, \mathsf{t}_n) := \bot$ otherwise. If f is a destructor, $\mathit{eval}_{\mathsf{f}}(\mathsf{t}_1, \dots, \mathsf{t}_n) := \mathsf{f}(\mathsf{t}_1, \dots, \mathsf{t}_n)$ if $\mathsf{f}(\mathsf{t}_1, \dots, \mathsf{t}_n) \neq \bot$ and $\mathit{eval}_{\mathsf{f}}(\mathsf{t}_1, \dots, \mathsf{t}_n) := \bot$ otherwise.

A *deduction relation* $\vdash_{\text{CoSP}}$ between 2^{T} and T formalizes which terms can be deduced from other terms. The intuition of $S \vdash_{\text{CoSP}} \mathsf{m}$ for $S \subseteq \mathsf{T}$ and $\mathsf{m} \in \mathsf{T}$ is that the term m can be deduced from the terms in S.

[5] In RCF, every construct from the language (including lambda-abstractions) is represented in FOL/F formulae by a special function symbol; see the full version of [BBF$^+$08].

Definition 4 (Deduction relation) *A* deduction relation $\vdash_{\mathrm{CoSP}}$ *over a message type* T *is a relation between* 2^{T} *and* T. $\diamond$

The constructors, destructors, and nonces, together with the message type and the deduction relation form a symbolic model. Such a symbolic model describes a particular Dolev-Yao-style theory.

Definition 5 (Symbolic model) *A* symbolic model $\mathsf{M} = (\mathsf{C}, \mathsf{N}, \mathsf{T}, \mathsf{D}, \vdash_{\mathrm{CoSP}})$ *consists of a set of constructors* C*, a set of nonces* N*, a message type* T *over* C *and* N *with* $\mathsf{N} \subseteq \mathsf{T}$*, a set of destructors* D *over* T*, and a deduction relation* $\vdash_{\mathrm{CoSP}}$ *over* T. $\diamond$

A *CoSP protocol* Π is defined as a tree with labelled nodes and edges. We distinguish *computation nodes*, which describe constructor applications, destructors applications, and nonce creations, *output* and *input nodes*, which describe communication, and *control nodes*, which allow the adversary to influence the control flow of the protocol. Computation and output nodes refer to earlier computation and input nodes; the messages computed at these earlier nodes are then taken as arguments by the constructor/destructor applications or sent to the adversary.

For CoSP protocols, both a symbolic and a computational execution are defined by traversing the tree. In the symbolic execution, the computation nodes operate on terms, and the input/output nodes receive/send terms to the (symbolic) adversary. The successors of control nodes are chosen non-deterministically. In the computational execution, the computation nodes operate on bitstrings (using a computational implementation Impl), and the input/output nodes receive/send bitstrings to the (polynomial-time) adversary. The adversary chooses the successors of control nodes.

Definition 6 (CoSP protocol) *A* CoSP protocol Π *is a tree with a distinguished root and labels on both edges and nodes. Each node has a unique identifier* ν *and one of the following types:*[6]

- Computation nodes *are annotated with a constructor, destructors, or nonce* f/n *together with the identifiers of* n *(not necessarily distinct) nodes. Computation nodes have exactly two successors; the corresponding edges are labeled with yes and no, respectively.*
- Output nodes *are annotated with the identifier of one node. An output node has exactly one successor.*
- Input nodes *have no further annotation. An input node has exactly one successor.*
- Control nodes *are annotated with a bitstring* l*. A control node has at least one and up to countably many successors annotated with distinct bitstrings* $l' \in \{0,1\}^*$*. (We call* l *the out-metadata and* l' *the in-metadata.)*

If a node ν *contains an identifier* ν' *in its annotation, then* ν' *has to be on the path from the root to* ν *(including the root, excluding* ν*), and* ν' *must be a computation node or input node. In case* ν' *is a computation node, the path from* ν' *to* ν *has to additionally go through the outgoing edge of* ν' *with label yes.* $\diamond$

Definition 7 (Symbolic execution) *Let a symbolic model* $(\mathsf{C}, \mathsf{N}, \mathsf{T}, \mathsf{D}, \vdash_{\mathrm{CoSP}})$ *and a CoSP protocol* Π *be given. A* full trace *is a (finite) list of tuples* (S_i, ν_i, f_i) *such that the following conditions hold:*

- Correct start*:* $S_1 = \varnothing$*,* ν_1 *is the root of* Π*,* f_1 *is a totally undefined partial function mapping node identifiers to terms.*
- Valid transition*: For every two consecutive tuples* (S, ν, f) *and* (S', ν', f') *in the list, let* $\underline{\tilde{\nu}}$ *be the node identifiers in the annotation of* ν *and define* $\underline{\tilde{t}}$ *through* $\tilde{t}_j := f(\tilde{\nu}_j)$*. We have:*
  - *If* ν *is a computation node with constructor, destructor or nonce* f*, then* $S' = S$*. If* $m := eval_{\mathsf{f}}(\underline{\tilde{t}}) \neq \bot$*,* 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* $f' = f$*.*
  - *If* ν *is an input node, then* $S' = S$ *and* ν' *is the successor of* ν *in* Π *and there exists an* m *with* $S \vdash_{\mathrm{CoSP}} m$ *and* $f' = f(\nu := m)$*.*
  - *If* ν *is an output node, then* $S' = S \cup \{\tilde{t}_1\}$*,* ν' *is the successor of* ν *in* Π *and* $f' = f$*.*

[6]Note in [BHU09], there is an additional type of node, the non-deterministic node. We have omitted the non-deterministic nodes here because we do not use them in the CoSP protocols constructed in this paper.

A list of node identifiers (ν_i) is a node trace *if there is a full trace with these node identifiers.* $\diamond$

Definition 8 (Computational implementation) *Let a symbolic model* $\mathsf{M} = (\mathsf{C}, \mathsf{N}, \mathsf{T}, \mathsf{D}, \vdash_{\text{CoSP}})$ *be given. A* computational implementation of M *is a family of functions* $\text{Impl} = (\text{Impl}_x)_{x \in \mathsf{C} \cup \mathsf{D} \cup \mathsf{N}}$ *such that* Impl_{f} *for* $\mathsf{f}/n \in \mathsf{C} \cup \mathsf{D}$ *is a partial deterministic function* $\mathbb{N} \times (\{0,1\}^*)^n \to \{0,1\}^*$, *and* Impl_{n} *for* $\mathsf{n} \in \mathbf{N}$ *is a total probabilistic function with domain* $\mathbb{N}$ *and range* $\{0,1\}^*$ *(i.e., it specifies a probability distribution on bitstrings that depends on its argument). The first argument of* Impl_{f} *and* Impl_{n} *represents the security parameter.*

All functions Impl_{f} *have to be computable in deterministic polynomial-time, and all* Impl_{n} *have to be computable in probabilistic polynomial-time.* $\diamond$

Definition 9 (Computational execution) *Let a symbolic model* $\mathsf{M} = (\mathsf{C}, \mathsf{N}, \mathsf{T}, \mathsf{D}, \vdash_{\text{CoSP}})$, *a computational implementation* Impl *of* M, *and a CoSP protocol* Π *be given. Let a probabilistic polynomial-time interactive machine E (the adversary) be given (polynomial-time in the sense that the number of steps in all activations are bounded in the length of the first input of E), and let p be a polynomial. We define a probability distribution* $\mathit{Nodes}^p_{\mathsf{M},\text{Impl},\Pi,E}(k)$, *the* computational node trace, *on (finite) lists of node identifiers (ν_i) according to the following probabilistic algorithm (both the algorithm and E are run on input k):*

- *Initial state: $\nu_1 := \nu$ is the root of Π. Let f be an initially empty partial function from node identifiers to bitstrings, and let n be an initially empty partial function from* N *to bitstrings.*
- *For $i = 2, 3, \ldots$ do the following:*
  - *Let $\underline{\tilde{\nu}}$ be the node identifiers in the annotation of ν. $\tilde{m}_j := f(\tilde{\nu}_j)$.*
  - *Proceed depending on the type of node ν:*
    - *If ν is a computation node with nonce* $\mathsf{n} \in \mathsf{N}$*: Let $m' := n(N)$ if $n(N) \neq \bot$ and sample m' according to* $\text{Impl}_{\mathsf{n}}(k)$ *otherwise. Let ν' be the yes-successor of ν, $f' := f(\nu := m')$, and $n' := n(N := m')$. Let $\nu := \nu'$, $f := f'$ and $n := n'$.*
    - *If ν is a computation node with constructor or destructor* f, *then* $m' := \text{Impl}_{\mathsf{f}}(k, \underline{\tilde{m}})$. *If $m' \neq \bot$, then ν' is the yes-successor of ν, if $m' = \bot$, then ν' is the no-successor of ν. Let $f' := f(\nu := m')$. Let $\nu := \nu'$ and $f := f'$.*
    - *If ν is an input node, ask for a bitstring m from E. Abort the loop if E halts. Let ν' be the successor of ν. Let $f := f(\nu := m)$ and $\nu := \nu'$.*
    - *If ν is an output node, send $\tilde{m}_1$ to E. Abort the loop if E halts. Let ν' be the successor of ν. Let $\nu := \nu'$.*
    - *If ν is a control node, annotated with out-metadata l, send l to E. Abort the loop if E halts. Upon receiving an answer l', let ν' be the successor of ν along the edge labeled l' (or the lexicographically smallest edge if there is no edge with label l'). Let $\nu := \nu'$.*
  - *Let $\nu_i := \nu$.*
  - *Let len be the number of nodes from the root to ν plus the total length of all bitstrings in the range of f. If len $> p(k)$, stop.*

$\diamond$

Definition 10 (Trace property) *A* trace property $\wp$ *is an efficiently decidable and prefix-closed set of (finite) lists of node identifiers.*

Let $\mathsf{M} = (\mathsf{C}, \mathsf{N}, \mathsf{T}, \mathsf{D}, \vdash_{\text{CoSP}})$ *be a symbolic model and Π a CoSP protocol. Then* Π symbolically satisfies *a trace property $\wp$ iff every node trace of Π is in $\wp$.*

Let Impl *be a computational implementation of* M *and let Π be a CoSP protocol. Then* (Π, Impl) computationally satisfies *a trace property $\wp$ iff for all probabilistic polynomial-time interactive machines E and all polynomials p,* $\mathit{Nodes}^p_{\mathsf{M},\text{Impl},\Pi,E}(k) \in \wp$ *with overwhelming probability.* $\diamond$

Definition 11 (Computational soundness) *A computational implementation* Impl *of a symbolic model* $\mathsf{M} = (\mathsf{C}, \mathsf{N}, \mathsf{T}, \mathsf{D}, \vdash_{\text{CoSP}})$ *is* computationally sound *for a class P of CoSP protocols iff for every trace property $\wp$ and for every efficient CoSP protocol Π, we have that* (Π, Impl) *computationally satisfies $\wp$ whenever Π symbolically satisfies $\wp$ and $\Pi \in P$.* $\diamond$

4 The Dolev-Yao library

In this paper, we do not restrict our attention to a specific symbolic library. We instead provide a computational soundness result for any symbolic library fulfilling certain conditions that we detail in this section.

4.1 The library

We first define a general Dolev-Yao model, which is a symbolic model subject to certain natural restrictions.

Definition 12 (DY Model) *We say that a symbolic model* $\mathsf{M} = (\mathsf{C}, \mathsf{D}, \mathsf{N}, \mathsf{T}, \vdash_{\text{CoSP}})$ *is a* DY model *if* $\mathsf{N} = \mathsf{N_E} \uplus \mathsf{N_P}$ *for countably infinite* $\mathsf{N_E}, \mathsf{N_P}$*, and* $\mathsf{equals}/2 \in \mathsf{D}$ *where* $\mathsf{equals}(x, x) := x$ *and* $\mathsf{equals}(x, y) := \bot$ *for* $x \neq y$*, and* $\vdash_{\text{CoSP}}$ *is the smallest relation such that* $\mathsf{m} \in S \Rightarrow S \vdash_{\text{CoSP}} \mathsf{m}$*,* $\mathsf{n} \in \mathsf{N}_E \Rightarrow S \vdash_{\text{CoSP}} \mathsf{n}$*, and such that for any constructor or destructor* $\mathsf{f}/n \in \mathsf{C} \cup \mathsf{D}$ *and for any* $\mathsf{t}_1, \dots, \mathsf{t}_n \in \mathsf{T}$ *satisfying* $\forall i \in [1, n]. S \vdash_{\text{CoSP}} \mathsf{t}_i$ *and* $\bot \neq \mathit{eval}_\mathsf{f}(\mathsf{t}_1, \dots, \mathsf{t}_n) \in \mathsf{T}$*, we have* $S \vdash_{\text{CoSP}} \mathsf{f}(\mathsf{t}_1, \dots, \mathsf{t}_n)$. ◇

In the following, we will only reason about DY models. Intuitively, in a DY library each CoSP term m is represented by $\mathsf{message}\ M$, where $\mathsf{message}$ is a private constructor that tags all values which the library operates on and M is an encoding of m. CoSP constructors are represented by RCF constructors and nonces are represented by RCF names. For each constructor and destructor f, the library exports a function lib_f such that $\sigma^\mathsf{M}_\mathrm{DY}(\mathit{lib}_\mathsf{f})$ $(\mathsf{message}\ M_1, \dots, \mathsf{message}\ M_n)$ returns $\mathsf{some}\ \mathsf{message}\ M$ if $\mathit{eval}_\mathsf{f}(\mathsf{m}_1, \dots, \mathsf{m}_n)$ returns m, or none if $\mathit{eval}_\mathsf{f}(\mathsf{m}_1, \dots, \mathsf{m}_n)$ returns $\bot$. In addition, the library exports a function *nonce* that picks a fresh name (to be used as a nonce) and functions *send* and *recv* for sending and receiving terms of the form $\mathsf{message}\ M$ over a public channel.

For example, if we have a symbolic model containing encryptions and decryptions (such as the one presented in Section 5.4), we would represent a ciphertext with key $\mathsf{ek}(k)$ and randomness r as $\mathsf{message}(\mathsf{enc}(\mathsf{ek}(k), m, r))$. The decryption function in the library would then be defined by $\sigma^\mathsf{M}_\mathrm{DY}(\mathit{lib}_\mathsf{dec}) := \lambda x.\mathsf{match}\ x\ \mathsf{with}\ (\mathsf{message}(\mathsf{dk}(y)), \mathsf{message}(\mathsf{enc}(\mathsf{ek}(y), z, w)))\ \mathsf{then}\ \mathsf{some}(\mathsf{message}(z))\ \mathsf{else}\ \mathsf{none}$. A nonce would be represented as $\mathsf{message}(\mathsf{nonce}(\lambda x.a!x))$ for some fresh name a.[7]

Instead of giving a definition that is specific to a particular DY model, we will give a general definition of a DY library for a DY model. In the following, we assume an arbitrary embedding ι of terms T into the set of closed RCF values. We further assume a fixed name a_{chan} used internally by the library for communication and we assume that there is a value-context $C_\iota[]$ (a value with a hole) such that $\{C_\iota[a] : a \neq a_{chan} \text{ is a name}\} = \iota(\mathsf{N})$.

Definition 13 (DY Library) *A DY library for* $\mathsf{M} = (\mathsf{C}, \mathsf{D}, \mathsf{N}, \mathsf{T}, \vdash_{\text{CoSP}})$ *is a substitution* $\sigma^\mathsf{M}_\mathrm{DY}$ *from variables to RCF functions satisfying the following conditions:*

- *Let* $\mathsf{message}$ *be a private constructor.*
- $\operatorname{dom} \sigma^\mathsf{M}_\mathrm{DY} = \{\mathit{lib}_\mathsf{f} \mid \mathsf{f} \in \mathsf{C} \cup \mathsf{D}\} \cup \{\mathit{nonce}, \mathit{send}, \mathit{recv}\}$.
- $\sigma^\mathsf{M}_\mathrm{DY}(\mathit{lib}_\mathsf{f})$ *is a pure function such that the following holds for all* $\mathsf{m}_1, \dots, \mathsf{m}_n \in \mathsf{T}$*: If* $\mathsf{m} := \mathit{eval}_\mathsf{f}(\mathsf{m}_1, \dots, \mathsf{m}_n) \neq \bot$*, then* $\sigma^\mathsf{M}_\mathrm{DY}(\mathit{lib}_\mathsf{f})$ $(\mathsf{message}\ \iota(\mathsf{m}_1), \dots, \mathsf{message}\ \iota(\mathsf{m}_n)) \rightarrow^* \mathsf{some}\ \mathsf{message}\ \iota(\mathsf{m})$. *If* $\mathsf{m} = \bot$*, then* $\sigma^\mathsf{M}_\mathrm{DY}(\mathit{lib}_\mathsf{f})$ $(\mathsf{message}\ \iota(\mathsf{m}_1), \dots, \mathsf{message}\ \iota(\mathsf{m}_n)) \rightarrow^* \mathsf{none}$. *In all other cases,* $\sigma^\mathsf{M}_\mathrm{DY}(\mathit{lib}_\mathsf{f})\ (\dots)$ *is stuck.*
- $\sigma^\mathsf{M}_\mathrm{DY}(\mathit{nonce}) = \mathsf{fun}\ \_ \rightarrow \nu a.\mathsf{message}\ C_\iota[a]$.
- $\sigma^\mathsf{M}_\mathrm{DY}(\mathit{send}) = (\mathsf{fun}\ x \rightarrow (\mathsf{match}\ x\ \mathsf{with}\ \mathsf{message}\ \_\ \mathsf{then}\ a_{chan}!\mathsf{message}\ x\ \mathsf{else}\ \mathit{stuck}))$. *Here stuck is a pure diverging RCF expression.*
- $\sigma^\mathsf{M}_\mathrm{DY}(\mathit{recv}) = \mathsf{fun}\ \_ \rightarrow a_{chan}?$
- $\mathit{fv}(\operatorname{range}(\sigma^\mathsf{M}_\mathrm{DY})) = \emptyset$ *and* $\mathit{fn}(\operatorname{range}(\sigma^\mathsf{M}_\mathrm{DY})) = a_\mathsf{chan}$.
- *For any variable* $x \in \operatorname{dom} \sigma^\mathsf{M}_\mathrm{DY}$*, and any mpc-free value* $M \neq x$*, we have* $\sigma^\mathsf{M}_\mathrm{DY}(x) \neq M\sigma^\mathsf{M}_\mathrm{DY}$. *(We call a substitution satisfying this condition condition* equality-friendly*.)* ◇

[7]For syntactic reasons, RCF forbids to simply write $\mathsf{message}(\mathsf{nonce}(a))$ if a is a name.

The requirement that $\sigma^{\mathsf{M}}_{\mathrm{DY}}$ is equality-friendly is a technical condition to ensure that the outcome of equality-tests in a program execution does not depend on the internal code of the library functions. For example, if we had that $\sigma^{\mathsf{M}}_{\mathrm{DY}}(\mathit{lib}_{\mathsf{f}_1}) = (\lambda x.\sigma^{\mathsf{M}}_{\mathrm{DY}}(\mathit{lib}_{\mathsf{f}_2})x)$, the test $\mathit{lib}_{\mathsf{f}_1} = (\lambda x.\mathit{lib}_{\mathsf{f}_2}x)$ would succeed in a program linked to $\sigma^{\mathsf{M}}_{\mathrm{DY}}$. To avoid such dependencies on the internal code of the library, we introduce equality-friendliness. Note that equality-friendliness is only necessary because RCF allows syntactic equality tests on lambda-abstractions.

Equality-friendliness can be enforced, e.g., by requiring that all $\sigma(f)$ are expressions of the form $\lambda x.(\mathit{magic}; A)$ for some RCF expression A where $\mathit{magic} := (\lambda z.\mathsf{match}\ z\ \mathsf{with}\ \mathsf{message}\ y\ \mathsf{then}\ ()\ \mathsf{else}\ ())$.

To interface an expression A with a library $\sigma^{\mathsf{M}}_{\mathrm{DY}}$, we use the expression $A\sigma^{\mathsf{M}}_{\mathrm{DY}}$. We will only consider programs A such that $\mathit{fn}(A) = \emptyset$ and $\mathit{fv}(A) \subseteq \operatorname{dom} \sigma^{\mathsf{M}}_{\mathrm{DY}}$.

In $\sigma^{\mathsf{M}}_{\mathrm{DY}}$, all messages M are protected by the private constructor $\mathsf{message}$. However, if an opponent would be allowed to perform a pattern match on $\mathsf{message}$, he could get the internal representation of M and thus, e.g., extract the plaintext from an encryption. Similarly, an adversary applying $\mathsf{message}$ could produce invalid messages. Thus, when using $\sigma^{\mathsf{M}}_{\mathrm{DY}}$, we have to restrict ourselves to pc-free opponents. The following variant of robust $\rightarrow$-safety models this.

Definition 14 (Robust $\rightarrow$-σ-safety) *Let σ be a substitution. We call an RCF expression a* σ-opponent *iff $\mathit{fv}(O) \subseteq \operatorname{dom}\sigma$ and O is pc-free and contains neither assertions nor assumptions and $a_{chan} \notin \mathit{fn}(O)$.*

An RCF expression A with $\mathit{fv}(A) \subseteq \operatorname{dom}\sigma$ is robustly $\rightarrow$-σ-safe *iff the application $(O\ A)\sigma$ is $\rightarrow$-safe for all σ-opponents O.* $\diamond$

Note that in contrast to Definition 2, we explicitly apply the substitution σ representing the library to the opponent. This is because a pc-free opponent has to invoke library functions in order to perform encryptions, outputs, etc. Furthermore, we will also need that the programs we analyze operate on terms tagged by $\mathsf{message}$ only through the library. In order to enforce this (and other invariants that will be used in various locations in the proofs) we introduce the following well-formedness condition:

Definition 15 *Let A be an RCF expression and $\mathsf{M} = (\mathsf{C}, \mathsf{N}, \mathsf{T}, \mathsf{D}, \vdash)$ a DY model. We say $\mathsf{M} \vdash A$ iff $\mathit{fv}(A) \subseteq \{\mathit{lib}_{\mathsf{f}} : \mathsf{f} \in \mathsf{C} \cup \mathsf{D}\} \cup \{\mathit{nonce}, \mathit{send}, \mathit{recv}\}$ and $a_{chan} \notin \mathit{fn}(A)$ and A is pc-free and the FOL/F-formulae in A do not contain forbidden function symbols.* $\diamond$

4.2 Dolev-Yao transition relation

The COSP framework assumes the atomicity of cryptographic operations. In general, however, Dolev-Yao libraries may define these operations by a sequence of commands, which may lead to non-atomic computations. For this reason, a convenient tool for the embedding of a language in COSP is the definition of a symbolic semantics where cryptographic operations are executed atomically. This is achieved by defining a new reduction relation $A \rightsquigarrow B$ (cf. Figure 5), which differs from the standard reduction relation $A \rightarrow B$ in that cryptographic operations are atomically performed.

Using the definition of $\rightsquigarrow$, we can reformulate the notion of safety. Our formulation is justified by Lemma 1 below.

Definition 16 ($\rightsquigarrow$-σ-Safety) *A structure $\boldsymbol{S}$ is* statically σ-safe *iff $P\sigma \vdash C\sigma$ where P are the active assumptions and C the active assertions of $\boldsymbol{S}$.*

An expression A is $\rightsquigarrow$-σ-safe *if for all $\boldsymbol{S}$ such that $A \rightsquigarrow^* \boldsymbol{S}$ we have that $\boldsymbol{S}$ is statically σ-safe.* $\diamond$

In contrast to Definition 14, when defining robust safety with respect to $\rightsquigarrow$, we to not apply σ to the opponent or the program, because σ is hard-coded into $\rightsquigarrow$:

Definition 17 (Robust $\rightsquigarrow$-σ-safety) *An RCF expression A with $\mathit{fv}(A) \subseteq \operatorname{dom}\sigma$ is* robustly $\rightsquigarrow$-σ-safe *iff the application $O\ A$ is $\rightsquigarrow$-σ-safe for all σ-opponents O.* $\diamond$

A necessary ingredient for the computational soundness result is the proof that if a program is $\rightarrow$-safe then it is also $\rightsquigarrow$-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safe.

$$
\begin{array}{rcl}
(\lambda x.A)N & \rightsquigarrow & A\{N/x\} \\
\mathsf{let}\ (x_1,x_2) = (N_1,N_2)\ \mathsf{in}\ A & \rightsquigarrow & A\{N_1/x_1, N_2/x_2\} \\
\mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ A\ \mathsf{else}\ B & \rightsquigarrow & \begin{cases} A\{N/x\} & \text{if } M = h\ N \text{ for some } N \\ B & \text{otherwise} \end{cases} \\
M = N & \rightsquigarrow & \begin{cases} \mathsf{true} & \text{if } M = N \\ \mathsf{false} & \text{otherwise} \end{cases} \\
a!M \upharpoonright a? & \rightsquigarrow & M \\
\mathsf{assert}\ C & \rightsquigarrow & () \\
\mathsf{let}\ x = M\ \mathsf{in}\ A & \rightsquigarrow & A\{M/x\} \\
A \rightsquigarrow A' \Rightarrow \mathsf{let}\ x = A\ \mathsf{in}\ B & \rightsquigarrow & \mathsf{let}\ x = A'\ \mathsf{in}\ B \\
A \rightsquigarrow A' \Rightarrow \nu a.A & \rightsquigarrow & \nu a.A' \\
A \rightsquigarrow A' \Rightarrow (A \upharpoonright B) & \rightsquigarrow & (A' \upharpoonright B) \\
A \rightsquigarrow A' \Rightarrow (B \upharpoonright A) & \rightsquigarrow & (B \upharpoonright A') \\
A \equiv B \rightsquigarrow B' \equiv A' \Longrightarrow A & \rightsquigarrow & A' \\
\mathit{send}\ (\mathsf{message}\ M) & \rightsquigarrow & a_{chan}!\mathsf{message}\ M \\
\mathit{recv}\ N & \rightsquigarrow & a_{chan}? \\
\mathit{nonce}\ M & \rightsquigarrow & \nu a.\mathsf{message}\ C_\iota[a] \\
\sigma^{\mathsf{M}}_{\mathrm{DY}}(\mathit{lib}_{\mathsf{f}})\ M \to^* N \Longrightarrow \mathit{lib}_{\mathsf{f}}\ M & \rightsquigarrow & N \qquad (\mathit{lib}_{\mathsf{f}} \in \mathrm{dom}\,\sigma^{\mathsf{M}}_{\mathrm{DY}} \setminus \{\mathit{send}, \mathit{recv}, \mathit{nonce}\})
\end{array}
$$

Figure 5: Reduction relation $A \rightsquigarrow B$

Lemma 1 *Let A be pc-free. If $A\sigma^{\mathsf{M}}_{\mathrm{DY}}$ is $\to$-safe then A is $\rightsquigarrow$-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safe.*

Proof. By definition of $\to$-safety and $\rightsquigarrow$-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safety, we only have to show that $A \rightsquigarrow^* B$ implies $A\sigma^{\mathsf{M}}_{\mathrm{DY}} \to^* B\sigma^{\mathsf{M}}_{\mathrm{DY}}$. The proof is by structural induction on the derivation of $A \rightsquigarrow^* B$. The base case is when $A = B$ and no reduction step is applied, and the proof is straightforward. The induction cases are also straightforward, except for the equality, match, and library function application rules.

Equality. Since $\to$ tests $M\sigma^{\mathsf{M}}_{\mathrm{DY}} = N\sigma^{\mathsf{M}}_{\mathrm{DY}}$ while $\rightsquigarrow$ tests $M = N$, we have to prove that $M\sigma^{\mathsf{M}}_{\mathrm{DY}} = N\sigma^{\mathsf{M}}_{\mathrm{DY}} \Leftrightarrow M = N$. The $\Leftarrow$ direction is straightforward. For proving the $\Rightarrow$ direction, we actually prove that $M \neq N \Rightarrow M\sigma^{\mathsf{M}}_{\mathrm{DY}} \neq N\sigma^{\mathsf{M}}_{\mathrm{DY}}$.

Since A is pc-free, we can easily see that B is mpc-free and therefore M and N are mpc-free. Now we proceed by structural induction on M. We first reason on the base cases:

$M = a$ The only interesting case is when $N = x$. By Definition 13, $\mathrm{range}\,\sigma^{\mathsf{M}}_{\mathrm{DY}}$ is a set of functions, hence $\sigma^{\mathsf{M}}_{\mathrm{DY}}(x) \neq a$.

$M = ()$ Analogous to the previous item.

$M = x$ The proof follows by observing that $\sigma^{\mathsf{M}}_{\mathrm{DY}}$ is equality-friendly.

We now reason on the induction cases:

$M = h\ M'$ & $N = h'\ N'$ & $h \neq h'$ We clearly have $M\sigma^{\mathsf{M}}_{\mathrm{DY}} \neq N\sigma^{\mathsf{M}}_{\mathrm{DY}}$.

$M = h\ M'$ & $N = h\ N'$ If h is a public constructor, then the proof follows directly from the induction hypothesis. If h is a private constructor, then we do not know whether M' and N' are mpc-free or not. We do know, however, that they are closed (by an inspection of the $\rightsquigarrow$-semantics and by Definition 13). Therefore $M\sigma^{\mathsf{M}}_{\mathrm{DY}} = M$ and $N\sigma^{\mathsf{M}}_{\mathrm{DY}} = N$.

The remaining cases follow straightforwardly from the induction hypothesis.

Match. We have to show that (i) if $\mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ C\ \mathsf{else}\ D \rightsquigarrow C\{N/x\}$ (i.e., $M = h\ N$ for some N and $B = C\{N/x\}$) then $\mathsf{match}\ M\sigma^{\mathsf{M}}_{\mathrm{DY}}\ \mathsf{with}\ h\ x\ \mathsf{then}\ C\sigma^{\mathsf{M}}_{\mathrm{DY}}\ \mathsf{else}\ D\sigma^{\mathsf{M}}_{\mathrm{DY}} \to C\{N/x\}\sigma^{\mathsf{M}}_{\mathrm{DY}}$ and (ii) if $\mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ C\ \mathsf{else}\ D \rightsquigarrow D$ (i.e., $\nexists N.M = h\ N$) then $\mathsf{match}\ M\sigma^{\mathsf{M}}_{\mathrm{DY}}\ \mathsf{with}\ h\ x\ \mathsf{then}\ C\sigma^{\mathsf{M}}_{\mathrm{DY}}\ \mathsf{else}\ D\sigma^{\mathsf{M}}_{\mathrm{DY}} \to D\sigma^{\mathsf{M}}_{\mathrm{DY}}$. For this, we must show (i) $M = h\ N \Rightarrow M\sigma^{\mathsf{M}}_{\mathrm{DY}} = h\ N\sigma^{\mathsf{M}}_{\mathrm{DY}}$ and (ii) $\nexists N.M = h\ N \Rightarrow \nexists N.M\sigma^{\mathsf{M}}_{\mathrm{DY}} = h\ N$. The proof for (i) is straightforward (since $\sigma^{\mathsf{M}}_{\mathrm{DY}}$ is applied on both sides). For proving (ii), we actually prove that $\forall N.M \neq h\ N \Rightarrow \forall N.M\sigma^{\mathsf{M}}_{\mathrm{DY}} \neq h\ N$.

Since A is pc-free, we can easily see that B is mpc-free and therefore M is mpc-free and h is a public constructor. Now we proceed by structural induction on M. We first reason on the base cases:

$M = a$ Straightforward.

$M = ()$ Straightforward.

$M = x$ Assume by contradiction that $\exists N.x\sigma^{\mathsf{M}}_{\mathrm{DY}} = h\ N$. By Definition 13, $\operatorname{range}\sigma^{\mathsf{M}}_{\mathrm{DY}}$ is a set of functions, therefore $h\ N \notin \operatorname{range}\sigma^{\mathsf{M}}_{\mathrm{DY}}$, which yields a contradiction.

We now reason on the induction cases:

$M = h\ M'$ This case is obvious, since the hypothesis $\forall N.M \neq h\ N$ does not hold.

The remaining cases are straightforward.

Application of library functions. We have to show that if $f\ M \rightsquigarrow N$, with $f \in \operatorname{dom}\sigma^{\mathsf{M}}_{\mathrm{DY}}$, then $(f\sigma^{\mathsf{M}}_{\mathrm{DY}})(M\sigma^{\mathsf{M}}_{\mathrm{DY}}) \to^* N\sigma^{\mathsf{M}}_{\mathrm{DY}}$. We focus on the case $f \notin \{\mathit{send}, \mathit{recv}, \mathit{nonce}\}$, since the other cases are straightforward. By definition of $\rightsquigarrow$, we have $f\sigma^{\mathsf{M}}_{\mathrm{DY}}\ M \to^* N$. By definition of DY library, this reduction takes place only if $M = (\mathtt{message}\ \iota(\mathsf{m}_1), \ldots, \mathtt{message}\ \iota(\mathsf{m}_n))$. By definition of ι, $\iota(\mathsf{m}_i)$ and hence also M is closed. Similar reasoning shows that N is closed. Hence $M\sigma^{\mathsf{M}}_{\mathrm{DY}} = M$ and $N\sigma^{\mathsf{M}}_{\mathrm{DY}} = N$. This concludes the proof. □

5 Computational soundness

In this section, we present the computational soundness result for Dolev-Yao libraries.

5.1 Definitions

Since RCF only has semantics in the symbolic model (without probabilism and without the notion of a computational adversary) we need to introduce the notion of a computational execution of RCF expressions. In the computational execution, we let the adversary have the full control over the scheduling and all non-deterministic decisions. This models the worst case; a setting in which scheduling decisions are taken randomly can be reduced to this setting. Our computational execution maintains a state that consists of the current process $\mathbf{S}$ and an environment η. Cryptographic messages (i.e., bitstrings received by the adversary or computed by cryptographic operations) are represented in $\mathbf{S}$ by free variables. The bitstrings corresponding to these variables are maintained in the environment η. In each step of the execution, the adversary is given the process $\mathbf{S}$ (together with a set of equations E that tell him for which x, y we have $\eta(x) = \eta(y)$), and then can decide which of the different reduction rules from the RCF semantics should be applied to $\mathbf{S}$. Note that giving $\mathbf{S}$ to the adversary does not leak any secrets since these are only contained in η. If the adversary requests that a function application $\mathit{lib}_{\mathsf{f}}(x)$ is executed, where $\mathit{lib}_{\mathsf{f}}$ is a function in the DY library, the computational implementation $\mathrm{Impl}_{\mathsf{f}}$ is used to compute the result of this function application; that bitstring is then stored in η. Similarly for an application $\mathit{nonce}()$. If the adversary requests a function evaluation $\mathit{send}(x)$ the adversary is given the bitstring $\eta(x)$; in the case $\mathit{recv}()$, the adversary provides a bitstring that is then stored in η.

The following definition formalizes the computational execution of RCF expressions. We assume that each RCF expression has a unique[8] normal form (a structure) with the property that bound names are distinct from free names (and similarly for variables). We also assume that the bound names of the normal form are distinct from the free names of $\sigma^{\mathsf{M}}_{\mathrm{DY}}$. We follow the convention that "fresh variable" or "name" means a variable or name that does not occur in any of the variables maintained by the algorithm, nor in $\sigma^{\mathsf{M}}_{\mathrm{DY}}$. The parts in angle brackets ($\langle \cdots \rangle$) can be ignored, as they define the symbolic RCF-execution which will be discussed in the next section.

Definition 18 (Computational $\langle$symbolic$\rangle$ RCF-execution) *Let* M *be a DY model and let* Impl *be a computational implementation for* M. *Let A be an expression such that $\mathsf{M} \vdash A$, and let Adv be an interactive machine called the adversary. $\langle$Adv is a non-deterministic machine that only sends* m *if $S \vdash_{\mathrm{CoSP}}$* m *where S are the messages sent to Adv so far.$\rangle$ We define the* computational $\langle$symbolic$\rangle$ RCF-execution *as an interactive machine* $\mathrm{Exec}^{\mathrm{Impl}}_{A}(1^k)$ $\langle \mathrm{SExec}_A \rangle$ *that takes a security parameter k as argument $\langle$that does not take any argument$\rangle$ and interacts with Adv:*

- Start*: Let η be a totally undefined partial function mapping variables to bitstrings $\langle$terms$\rangle$. (η provides an environment giving bitstring $\langle$term$\rangle$ interpretation to the variables occurring in the current expression.)*
- Main Loop*: Let $\boldsymbol{S} = \nu a_1 \dots a_l.\ (\prod_{i\in 1\dots m} \mathsf{assume}\ C_i \upharpoonright \prod_{j \in 1\dots n} c_j!M_j \upharpoonright (\prod_{k\in 1 \dots o} \mathcal{L}_k\{e_k\}))$ be the normal form of A. Let $E := \{x = y : x \neq y,\ \eta(x) = \eta(y)\}$ be a set of formulae. Send $(\boldsymbol{S}, E)$ to the adversary and proceed depending on the type of message received from Adv as follows:*
  - *When receiving $(sync, j, k)$ from Adv, if $e_k = c_j?$, then set $A := B$, where B is the expression obtained from $\boldsymbol{S}$ by removing $c_j!M_j$ and replacing $\mathcal{L}_k\{e_k\}$ by $\mathcal{L}_k\{M_j\}$;*
  - *When receiving $(step, k)$:*
    - *If $e_k = x\ (y_1, \dots, y_n)$ with $x = lib_{\mathsf{f}}$ for some constructor or destructor f of arity n and $y_1, \dots, y_n \in \operatorname{dom} \eta$: Let $m := \mathrm{Impl}_{\mathsf{f}}(\eta(y_1), \dots, \eta(y_n))$ $\langle m := eval_{\mathsf{f}}(\eta(y_1), \dots, \eta(y_n))\rangle$. If $m \neq \bot$, set $\eta := \eta \uplus (z := m)$ for fresh z and $m' := \mathsf{some}\, z$. If $m = \bot$, set $\eta := \eta$ and $m' := \mathsf{none}$. Set $A := \boldsymbol{S}\{\mathcal{L}_k\{m'\}/\mathcal{L}_k\{e_k\}\}$;*
    - *If $e_k = nonce\ M$, then pick $r \leftarrow \mathrm{Impl}_{\mathsf{n}}(1^k)$ for some $\mathsf{n} \in \mathsf{N}_P$[9] $\langle$let r be a fresh protocol nonce$\rangle$ and set $\eta := \eta \uplus (z := r)$ for fresh z and $A := \boldsymbol{S}\{\mathcal{L}_k\{z\}/\mathcal{L}_k\{e_k\}\}$.*
    - *If $e_k = recv\ M$, then request a bitstring $\langle$term$\rangle$ m from the adversary and set $\eta := \eta \uplus (z := m)$ for fresh z and $A := \boldsymbol{S}\{\mathcal{L}_k\{z\}/\mathcal{L}_k\{e_k\}\}$.*
    - *If $e_k = send\ x$ with $x \in \operatorname{dom}\eta$: Send $\eta(x)$ to the adversary and set $A := \boldsymbol{S}\{\mathcal{L}_k\{()\}/\mathcal{L}_k\{e_k\}\}$.*
    - *If $e_k = (\lambda x.B)\ N$, let $A := \boldsymbol{S}\{\mathcal{L}_k\{B\{N/x\}\}/\mathcal{L}_k\{e_k\}\}$.*
    - *If $\mathcal{L}_k\{e_k\} = \mathcal{L}'\{\mathsf{let}\ x = M\ \mathsf{in}\ B\}$: Set $A := \boldsymbol{S}\{\mathcal{L}_k\{B\{M/x\}\}/\mathcal{L}_k\{e_k\}\}$.*
    - *If $e_k = (M = N)$: For every $x \in \operatorname{dom}\eta$, let $\rho(x)$ be the lexicographically first $y \in \operatorname{dom}\eta$ with $\eta(x) = \eta(y)$.*[10] *If $M\rho\sigma^{\mathsf{M}}_{\mathrm{DY}} = N\rho\sigma^{\mathsf{M}}_{\mathrm{DY}}$, let $b := \mathsf{true}$, otherwise let $b := \mathsf{false}$. Set $A := \boldsymbol{S}\{\mathcal{L}_k\{b\}/\mathcal{L}_k\{e_k\}\}$.*
    - *If $e_k = \mathsf{let}\ (x, y) = (M_1, M_2)\ \mathsf{in}\ B$: Set $A := \boldsymbol{S}\{\mathcal{L}_k\{B\{M_1/x, M_2/y\}\}/\mathcal{L}_k\{e_k\}\}$.*
    - *If $e_k = \mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2$: If M is of the form $h\,N$, let $B := B_1\{N/x\}$, otherwise let $B := B_2$. Set $A := \boldsymbol{S}\{\mathcal{L}_k\{B\}/\mathcal{L}_k\{e_k\}\}$.*
    - *If $e_k = \mathsf{assert}\ \ C$: Set $A := \boldsymbol{S}\{\mathcal{L}_k\{()\}/\mathcal{L}_k\{e_k\}\}$.*
  - *If none of these cases apply, do nothing.* $\diamond$

[8]The uniqueness of normal forms can be achieved, for instance, by imposing a lexicographical order on structures.

[9]The enc-sig-implementation conditions ensure that $\mathrm{Impl}_{\mathsf{n}}(1^k)$ does not depend on the choice of n.

[10]We use ρ to unify variables that refer to the same messages. This is necessary because the test $M\sigma^{\mathsf{M}}_{\mathrm{DY}} = N\sigma^{\mathsf{M}}_{\mathrm{DY}}$ without ρ would treat these variables as distinct terms.

Notice that the execution of $\mathrm{Exec}_A^{\mathrm{Impl}}(1^k)$ maintains the invariant that all bound variables and names in A are pairwise distinct and that they are distinct from the variables in the domain of η. For a given polynomial-time interactive machine Adv, a closed expression A, and a polynomial p, we let $Trace_{Adv,A,p}^{\mathrm{Impl}}(k)$ denote the list of pairs $(\mathbf{S}, E)$ output by $\mathrm{Exec}_A^{\mathrm{Impl}}(1^k)$ (at the beginning of each loop iteration) within the first $p(k)$ computation steps (jointly counted for $Adv(1^k)$ and $\mathrm{Exec}_A^{\mathrm{Impl}}(1^k)$).

Definition 19 (Statical equation-σ-safety) *Let σ be a substitution. A pair $(\boldsymbol{S}, E)$ of a structure $\boldsymbol{S}$ and a set E of equalities between variables is* statically equation-σ-safe *iff $P, eqs \vdash C$ where P and C are the active assumptions and assertions of $\boldsymbol{S}$, $vars := fv(E) \cup \operatorname{dom}\sigma$, exterms is the set of all FOL/F-subterms $h(t)$ of P, C with h forbidden and t syntactic closed, and*

$$\begin{aligned} eqs := E &\cup \{x \neq x' : x, x' \in vars,\ x \neq x',\ (x = x') \notin E\} \\ &\cup \{\forall \underline{y}.x \neq c(\underline{y}) : x \in vars,\ c \textit{ non-forbidden syntactic function symbol}\} \\ &\cup \{x \neq t : x \in vars,\ t \in exterms\}. \end{aligned}$$ ◇

We add the facts *eqs* in order to tell the logic what is known about the environment η in the computational execution. More precisely, we have $x = x'$ whenever $\eta(x) = \eta(x')$ and $x \neq x'$ otherwise. Furthermore, we have equations $x \neq x'$ when $x \neq x'$ refer to library functions (intuitively, this is justified because we assume all our libraries to be equality-friendly), and $x \neq x'$ when x is a library function and x' refers to the environment (i.e., represents a bitstring). The equations $x \neq t$ with $t \in exterms$ are best explained by an example: Let $A_0 := \mathsf{let}\ x = nonce\ \mathsf{in}\ \mathsf{let}\ x' = (\lambda z.z)\ \mathsf{in}\ \mathsf{assume}\ (x = x'); \mathsf{assert}\ (\mathsf{false})$. Then A_0 is robustly $\rightarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-safe: $A_0\sigma_{\mathrm{DY}}^{\mathsf{M}}$ reduces to $\mathsf{assume}\ (\sigma_{\mathrm{DY}}^{\mathsf{M}}(nonce) = \lambda z.z) \Rsh \mathsf{assert}\ (\mathsf{false})$ and we have $nonce\sigma_{\mathrm{DY}}^{\mathsf{M}} = (\lambda z.z) \vdash \mathsf{false}$ (this is implied by equality-friendliness). In the computational execution, however, we get the process $A = \mathsf{assume}\ (nonce = \lambda z.z) \Rsh \mathsf{assert}\ (\mathsf{false})$, thus for robust computational safety, we need that $nonce = (\lambda z.z), eqs \vdash \mathsf{false}$ holds. For this, we need the inequalities $x \neq t$ in *eqs*. Notice that these extra inequalities are necessary only because the logic allows us to compare lambda-abstractions syntactically.

Definition 20 (Robust computational safety) *Let* Impl *be a computational implementation. Let A be an expression with $\mathsf{M} \vdash A$. We say that A* is robustly computationally safe using Impl *if for all polynomial-time interactive machines Adv and all polynomials p, we have that* $\Pr[$*all components of $Trace_{Adv,A,p}(1^k)$ are statically equation-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-safe*$]$ *is overwhelming in k.* ◇

At the first glance, it may seem strange that the definition of robust computational safety is parametrized by the symbolic library $\sigma_{\mathrm{DY}}^{\mathsf{M}}$. An inspection of Definition 19, however, reveals that the definition only depends on the *domain* of $\sigma_{\mathrm{DY}}^{\mathsf{M}}$, i.e., on the set of cryptographic operations available to A.

5.2 Symbolic vs. computational execution

As described in Section 1.1, we now introduce an intermediate semantics, the symbolic RCF-execution. This execution is specified in Definition 18 (by reading the parts inside the $\langle \dots \rangle$), and is the exact analogue to the computational RCF-execution, except that it performs symbolic operations instead of computational ones.

We write $SExec_A$ in the set of lists of pairs $(\mathbf{S}, E)$ that can be sent in the symbolic execution. Like for the computational RCF-execution, these pairs $(\mathbf{S}, E)$ contain the information needed to check whether the active assumptions entail the active assertions. This allows us to express robust safety in terms of the symbolic RCF-execution:

Definition 21 (Robust SExec-safety) *Let A be an expression and M a DY model such that $\mathsf{M} \vdash A$. We say that A is* robustly *SExec*-safe *iff for any $((\boldsymbol{S}_1, E_1), \dots) \in SExec_A$, we have that $(\boldsymbol{S}_i, E_i)$ is statically equation-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-safe for all i.* ◇

We now proceed to show that robust $\rightsquigarrow$-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safety implies robust SExec-safety. For this, we first need a bit of notation:

We call a name a a *protocol name* if $C_\iota[a] \in \iota(\mathsf{N}_P)$ and an *adversary name* if $C_\iota[a] \in \iota(\mathsf{N}_E)$. Note that every name is either a protocol name or an adversary name. For an expression Q and a substitution φ from variables to CoSP terms, we say that Q is valid for φ if Q does not contain assume or assert, Q is pc-free, $fn(Q) = \varnothing$, and $fv(Q) \subseteq \operatorname{dom}\varphi \cup \operatorname{dom}\sigma^{\mathsf{M}}_{\mathrm{DY}}$, and all its free names are adversary names. Let A denote the process from the symbolic execution before the current main loop, let φ denote a substitution mapping $x_1, \dots, x_k$ to the messages sent to the adversary. Let $\iota_m(x) := \mathsf{message}\ \iota(x)$. For $\mathsf{n} \in \mathsf{N}$, let $\iota_N(\mathsf{n})$ be the name a with $C_\iota(a) = \iota(\mathsf{n})$. Let $\underline{\mathsf{n}}$ be the list of all nonces chosen by the protocol, and $\underline{n} := \iota_N(\underline{\mathsf{n}})$.

Let $A', \varphi', \underline{N}', \underline{n}'$ denote the values of $A, \varphi, \underline{N}, \underline{n}$ after the current iteration, and let $A_0, \varphi_0, \underline{N}_0, \underline{n}_0$ denote the values of $A, \varphi, \underline{N}, \underline{n}$ before the first iteration. We now show that each iteration of the symbolic execution can be simulated in the RCF semantics by choosing a suitable $\sigma^{\mathsf{M}}_{\mathrm{DY}}$-opponent.

Lemma 2 *Consider an iteration of the main loop of the symbolic execution with* $\mathsf{M} \vdash A_0$*. Then for all RCF expressions* Q' *valid for* φ'*, there is a substitution* φ *and an RCF expression* Q *valid for* φ *such that*

$$\nu\underline{n}.(Q(\iota_m \circ \varphi) \upharpoonleft A(\iota_m \circ \eta)) \rightsquigarrow^* \nu\underline{n}'.(Q'(\iota_m \circ \varphi') \upharpoonleft A'(\iota_m \circ \eta')).$$

Proof. By induction over the number of iterations of the main loop, and using the fact that $\mathsf{M} \vdash A_0$, we get that A is closed, pc-free, and does not contain a_{chan}.

We distinguish the cases in Definition 18.

- The adversary sends $(sync, j, k)$: Then $\underline{n} = \underline{n}'$, $\varphi = \varphi'$, $\eta = \eta'$. And $A(\iota_m \circ \eta) \rightsquigarrow A'(\iota_m \circ \eta)$. So the lemma holds with $Q := Q'$.

- The adversary sends $(step, k)$ and $e_k = x\ (y_1, \dots, y_n)$ with $x = lib_{\mathsf{f}}$ for some constructor or destructor f of arity n and $y_1, \dots, y_n \in \operatorname{dom}\eta$ and $m := eval_{\mathsf{f}}(\eta(y_1), \dots, \eta(y_n)) \neq \bot$: Then $\eta' = \eta \uplus (z \mapsto m)$ for fresh z and $A' := A\{\mathcal{L}_k\{\mathsf{some}\ z\}/\mathcal{L}_k\{e_k\}\}$. Furthermore $\underline{n}' = \underline{n}$ and $\varphi = \varphi'$. Let $m_i := \eta(y_i)$. Then

$$\begin{aligned}(x(y_1, \dots, y_n))(\iota_m \circ \eta) &= lib_{\mathsf{f}}(\mathsf{message}\ \iota(m_1), \dots, \mathsf{message}\ \iota(m_n)) \\ &\rightsquigarrow \mathsf{some}(\mathsf{message}\ \iota(m)) = (\mathsf{some}\ z)(\iota_m \circ \eta')\end{aligned}$$

  and hence $A(\iota_m \circ \eta) \rightsquigarrow A'(\iota_m \circ \eta)$. So the lemma holds with $Q := Q'$.

- The adversary sends $(step, k)$ and $e_k = x\ (y_1, \dots, y_n)$ with $x = lib_{\mathsf{f}}$ for some constructor or destructor f of arity n and $y_1, \dots, y_n \in \operatorname{dom}\eta$ and $eval_{\mathsf{f}}(\eta(y_1), \dots, \eta(y_n)) = \bot$: Then $\eta' = \eta$, $\varphi = \varphi'$, $\underline{n} = \underline{n}'$, and $A' = A\{\mathcal{L}_k\{\mathsf{none}\}/\mathcal{L}_k\{e_k\}\}$. Let $m_i := \eta(y_i)$. Then

$$(x(y_1, \dots, y_n))(\iota_m \circ \eta) = lib_{\mathsf{f}}(\mathsf{message}\ \iota(m_1), \dots, \mathsf{message}\ \iota(m_n)) \rightsquigarrow \mathsf{none} = \mathsf{none}(\iota_m \circ \eta')$$

  and hence $A(\iota_m \circ \eta) \rightsquigarrow A'(\iota_m \circ \eta)$. So the lemma holds with $Q := Q'$.

- The adversary sends $(step, k)$ and $e_k = nonce\ M$: Then $\varphi = \varphi'$ and $\eta' = \eta \uplus (z := \mathsf{r})$ for a fresh variable z and a fresh protocol nonce r. Furthermore $A' = A\{\mathcal{L}_k\{z\}/\mathcal{L}_k\{e_k\}\}$. Since r is a protocol nonce, $\iota(\mathsf{r}) = C_\iota[a]$ for some fresh protocol name a. Hence $a \notin fn(\sigma^{\mathsf{M}}_{\mathrm{DY}})$ and $\underline{n}' = \underline{n}a$. We have

$$e_k(\iota_m \circ \eta) = nonce\ (M(\iota_m \circ \eta)) \rightsquigarrow \nu a.\mathsf{message}\ C_\iota[a] = \nu a.(z(\iota_m \circ \eta'))$$

  and hence $A(\iota_m \circ \eta)\sigma^{\mathsf{M}}_{\mathrm{DY}} \rightsquigarrow \nu a.A'(\iota_m \circ \eta')\sigma^{\mathsf{M}}_{\mathrm{DY}}$. Since Q is valid, Q does not contain the protocol name a, so the lemma holds with $Q := Q'$.

- The adversary sends (step, k) and $e_k = \mathit{recv}\ M$ and the adversary sends the term m: Then $\eta' = \eta \uplus (z := m)$ and $\varphi' = \varphi$ and $\underline{n}' = \underline{n}$ and $A' = A\{\mathcal{L}_k\{z\}/\mathcal{L}_k\{e_k\}\}$. Furthermore $\operatorname{range}\varphi \vdash m$. By induction over the rules defining $\vdash_{\text{CoSP}}$, we have that for any term t with $\operatorname{range}\varphi \vdash t$, there is a RCF expression e such that $e(\iota_m \circ \varphi) \rightsquigarrow^* \iota_m(t)$ where e obeys the following grammar: $e ::= x_i | \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \dots \mathsf{let}\ x_n = e_n\ \mathsf{in}\ \mathit{lib}_{\mathsf{f}}(x_1, \dots, x_n) | a$ where a is an adversary name, $1 \le i \le |\varphi|$, and n is the arity of the constructor or destructor f. Thus there is an RCF expression e_m such that $e_m(\iota_m \circ \varphi) \rightsquigarrow^* \iota_m(m)$. Let y, y' be variables that are not free in Q'. Let $Q := \mathsf{let}\ y = e_m\ \mathsf{in}\ \mathsf{let}\ y' = \mathit{send}\ y\ \mathsf{in}\ Q'$. Since e_m only contains adversary nonces, Q is valid for φ. We have

$$\begin{aligned}
&Q(\iota_m \circ \varphi) \upharpoonright e_k(\iota_m \circ \eta)\\
&= \mathsf{let}\ y = e_m(\iota_m \circ \varphi)\ \mathsf{in}\ \mathsf{let}\ y' = \mathit{send}\ y\ \mathsf{in}\ Q'(\iota_m \circ \varphi) \upharpoonright \mathit{recv}\ (M(\iota_m \circ \eta))\\
&\rightsquigarrow^* \mathsf{let}\ y' = \mathit{send}\ \iota_m(m)\ \mathsf{in}\ Q'(\iota_m \circ \varphi) \upharpoonright \mathit{recv}\ (M(\iota_m \circ \eta))\\
&\rightsquigarrow^* \mathsf{let}\ y' = a_{\mathit{chan}}!\iota_m(m)\ \mathsf{in}\ Q'(\iota_m \circ \varphi) \upharpoonright a_{\mathit{chan}}?\\
&\rightsquigarrow \mathsf{let}\ y' = ()\ \mathsf{in}\ Q'(\iota_m \circ \varphi) \upharpoonright \iota_m(m)\\
&\rightsquigarrow Q'(\iota_m \circ \varphi) \upharpoonright \iota_m(m)\\
&= Q'(\iota_m \circ \varphi) \upharpoonright z(\iota_m \circ \eta')
\end{aligned}$$

and thus

$$\nu\underline{n}.(Q(\iota_m \circ \varphi) \upharpoonright A(\iota_m \circ \eta)) \rightsquigarrow^* \nu\underline{n}.(Q'(\iota_m \circ \varphi) \upharpoonright A'(\iota_m \circ \eta)) = \nu\underline{n}'.(Q'(\iota_m \circ \varphi') \upharpoonright A'(\iota_m \circ \eta')).$$

- The adversary sends (step, k) and $e_k = \mathit{send}\ M$ and $M \in \operatorname{dom}\eta$: Let $m := \eta(M)$. Then $\varphi' = \varphi \uplus (x_{n+1} \mapsto m)$ where $n := |\operatorname{dom}\varphi|$. Furthermore $\underline{n}' = \underline{n}$ and $\eta' = \eta$ and $A' = A\{\mathcal{L}_k\{()\}/\mathcal{L}_k\{e_k\}\}$. Let $Q := \mathsf{let}\ x_{n+1} = \mathit{recv}()\ \mathsf{in}\ Q'$. Then

$$\begin{aligned}
&Q(\iota_m \circ \varphi) \upharpoonright e_k(\iota_m \circ \eta)\\
&= \mathsf{let}\ x_{n+1} = \mathit{recv}()\ \mathsf{in}\ Q'(\iota_m \circ \varphi) \upharpoonright \mathit{send}(\mathsf{message}\ \iota(m))\\
&\rightsquigarrow^* \mathsf{let}\ x_{n+1} = a_{\mathit{chan}}?\ \mathsf{in}\ Q'(\iota_m \circ \varphi) \upharpoonright a_{\mathit{chan}}!(\mathsf{message}\ \iota(m))\\
&\rightsquigarrow^* \mathsf{let}\ x_{n+1} = \mathsf{message}(\iota(m))\ \mathsf{in}\ Q'(\iota_m \circ \varphi) \upharpoonright ()\\
&\rightsquigarrow Q'(\iota_m \circ \varphi') \upharpoonright ()
\end{aligned}$$

and thus

$$\nu\underline{n}.(Q(\iota_m \circ \varphi) \upharpoonright A(\iota_m \circ \eta)) \rightsquigarrow^* \nu\underline{n}'.(Q'(\iota_m \circ \varphi') \upharpoonright A'(\iota_m \circ \eta')).$$

- The adversary sends (step, k) and $e_k = (\lambda x.B)\ N$: Then $\underline{n} = \underline{n}'$, $\varphi = \varphi'$, $\eta = \eta'$. And $A(\iota_m \circ \eta) \rightsquigarrow A'(\iota_m \circ \eta)$. So the lemma holds with $Q := Q'$.

- The adversary sends (step, k) and $\mathcal{L}_k\{e_k\} = \mathcal{L}'\{\mathsf{let}\ x = M\ \mathsf{in}\ B\}$: Then $\underline{n} = \underline{n}'$, $\varphi = \varphi'$, $\eta = \eta'$. And $A(\iota_m \circ \eta) \rightsquigarrow A'(\iota_m \circ \eta)$. So the lemma holds with $Q := Q'$.

- The adversary sends (step, k) and $e_k = (M = N)$ and $M\rho = N\rho$ where $\rho(x)$ is the lexicographically first $y \in \operatorname{dom}\eta$ with $\eta(x) = \eta(y)$: Then $M(\iota_m \circ \eta) = M\rho(\iota_m \circ \eta) = N\rho(\iota_m \circ \eta) = N(\iota_m \circ \eta)$. Thus $e_k(\iota_m \circ \eta) \rightsquigarrow \mathsf{true}$ and hence $A(\iota_m \circ \eta) \rightsquigarrow A'(\iota_m \circ \eta)$. So the lemma holds with $Q := Q'$.

- The adversary sends (step, k) and $e_k = (M = N)$ and $M\rho \neq N\rho$ where ρ is as in the previous case. Assume for contradiction that $M(\iota_m \circ \eta) = N(\iota_m \circ \eta)$. Since all terms in the range of $\iota_m \circ \eta$ are of the form $\mathsf{message}\ M'$ and M, N are mpc-free (because A is mpc-free), we have that M and N differ only in their variables, i.e., that there is a context C with $M = C[x_1, \dots, x_n]$ and $N = C[x'_1, \dots, x'_n]$. Furthermore, for all i we have $\iota_m \circ \eta(x_i) = \iota_m \circ \eta(x'_i)$. Since ι_m is injective, $\eta(x_i) = \eta(x'_i)$. By definition of ρ, this implies $\rho(x_i) = \rho(x'_i)$. Thus $M\rho = N\rho$ in contradiction to $M\rho \neq N\rho$. Thus $M(\iota_m \circ \eta) \neq N(\iota_m \circ \eta)$. It follows that $e_k(\iota_m \circ \eta) \rightsquigarrow \mathsf{false}$ and hence $A(\iota_m \circ \eta) \rightsquigarrow A'(\iota_m \circ \eta)$. So the lemma holds with $Q := Q'$.

- The adversary sends (step, k) and $e_k = \mathsf{let}\ (x, y) = (M_1, M_2)\ \mathsf{in}\ B$: Then $\underline{n} = \underline{n}'$, $\varphi = \varphi'$, $\eta = \eta'$. And $A(\iota_m \circ \eta) \rightsquigarrow A'(\iota_m \circ \eta)$. So the lemma holds with $Q := Q'$.

- The adversary sends (step, k) and $e_k = \mathsf{match}\ h\ N\ \mathsf{with}\ h\ x\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2$: Then $\underline{n} = \underline{n}'$, $\varphi = \varphi'$, $\eta = \eta'$. And $A(\iota_m \circ \eta) \rightsquigarrow A'(\iota_m \circ \eta)$. So the lemma holds with $Q := Q'$.

- The adversary sends (step, k) and $e_k = \mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2$ and M is not of the form $h\ N$ for any N: If $M \notin \operatorname{dom} \eta$, then $M(\iota_m \circ \eta)$ is not of the form $h\ N$. If $M \in \operatorname{dom} \eta$, then $M(\iota_m \circ \eta)$ is of the form $\mathsf{message}\ N'$, and since A is mpc-free, $h \neq \mathsf{message}$ and thus $\mathsf{message}\ N'$ is not of the form $h\ N$.

  Thus $(\mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2)(\iota_m \circ \eta) \to B_2(\iota_m \circ \eta)$ and hence $A(\iota_m \circ \eta) \to A'(\iota_m \circ \eta)$. So the lemma holds with $Q := Q'$.

- The adversary sends (step, k) and $e_k = \mathsf{assert}\ C$: Then $\underline{n} = \underline{n}'$, $\varphi = \varphi'$, $\eta = \eta'$. Furthermore $e_k \rightsquigarrow ()$ and hence $A(\iota_m \circ \eta) \rightsquigarrow A'(\iota_m \circ \eta)$. So the lemma holds with $Q := Q'$.

- All other cases: $A' = A$, $\eta' = \eta$, $\varphi' = \varphi$, $\underline{n}' = \underline{n}$. Thus with $Q := Q'$,

$$\nu\underline{n}.(Q(\iota_m \circ \varphi) \upharpoonright A(\iota_m \circ \eta))\sigma_{\mathrm{DY}} = \nu\underline{n}'.(Q'(\iota_m \circ \varphi') \upharpoonright A'(\iota_m \circ \eta'))\sigma_{\mathrm{DY}}.$$

□

Lemma 3 *Let P, C be sets of FOL/F-formulae. Assume that P and C contain no forbidden function symbols. Let γ be a substitution mapping variables to closed FOL/F-terms. Assume that $\operatorname{dom}\gamma \subseteq \mathit{fv}(P, C)$. Assume that for all $x \in \operatorname{dom}\gamma$ we have $\gamma(x) = h(t)$ for forbidden h and syntactic t. Let*

$$\begin{aligned} \mathit{eqs} := \{x = x' : x, x' \in \operatorname{dom}\gamma,\ x \neq x',\ \gamma(x) = \gamma(x')\} \cup \{x \neq x' : x, x' \in \operatorname{dom}\gamma, \gamma(x) \neq \gamma(x')\} \\ \cup \{\forall \underline{y}.\ x \neq c(\underline{y}) : x' \in \operatorname{dom}\gamma,\ c \text{ non-forbidden syntactic function symbol}\}. \end{aligned}$$

Then $P\gamma \vdash C\gamma \Longrightarrow P, \mathit{eqs} \vdash C$.

Proof. Without loss of generality, we assume that the bound variables and the free variables of P, C, eqs are disjoint. In the following, we will use x to denote variables in $\operatorname{dom}\gamma$, y to denote variables not free in P, C, f for function symbols, h for forbidden function symbols, c for non-forbidden function symbols, t for FOL/F-terms, and u for members of the universe U (defined below).

To show that $P, \mathit{eqs} \vdash C$, it is sufficient to show that for any model $\mathcal{M}$ and any environment η with $\operatorname{dom}\eta = \operatorname{dom}\gamma$, if $\mathcal{M}, \eta \vDash P, \mathit{eqs}$ then $\mathcal{M}, \eta \vDash C$. Thus, fix such a model $\mathcal{M}$ and such an environment η. We are left to show $\mathcal{M}, \eta \vDash C$. Let U be the universe of $\mathcal{M}$, and for a non-forbidden function symbol f, let $\mathcal{M}_f$ denote the interpretation of f in $\mathcal{M}$, and analogously for forbidden function symbols h. (We treat constants as nullary function symbols, thus we do not need to treat them separately.) We write $\mathcal{M}_\eta(t)$ for the interpretation of t in the model $\mathcal{M}$ under environment η. If t is closed, we also write $\mathcal{M}(t)$ instead of $\mathcal{M}_\eta(t)$.

By definition of FOL/F, we have that for any syntactic function symbol f, M_f is injective. Thus for closed terms t, t' containing only syntactic function symbols, $\mathcal{M}(t) = \mathcal{M}(t')$ iff $t = t'$. Furthermore, since $\mathcal{M}, \eta \vDash \mathit{eqs}$, $\eta(x) = \eta(x')$ iff $\gamma(x) = \gamma(x')$. Hence $\mathcal{M}(\gamma(x)) = \mathcal{M}(\gamma(x'))$ iff $\eta(x) = \eta(x')$.

Since $\mathcal{M}(\gamma(x)) = \mathcal{M}(\gamma(x'))$ iff $\eta(x) = \eta(x')$ for all x, x', there is a permutation π on $\{\eta(x), \mathcal{M}(\gamma(x)) : x \in \operatorname{dom}\gamma\} \subseteq U$ such that $\pi(\eta(x)) = \mathcal{M}(\gamma(x))$ for all x. Fix such a permutation π. We extend π to a permutation on U by setting $\pi(u) := u$ for $u \notin \{\eta(x), \mathcal{M}(\gamma(x)) : x \in \operatorname{dom}\gamma\} \subseteq U$. We abbreviate $\pi(\underline{u})$ for $\pi(u_1), \pi(u_2), \ldots$ and analogously for π^{-1}.

For any c, x, we have $(\forall \underline{y}.\ x \neq c(\underline{y})) \in \mathit{eqs}$. Since $\mathcal{M}, \eta \vDash \mathit{eqs}$ by assumption, it follows that $\forall \underline{u} \in U.\ \eta(x) \neq \mathcal{M}_c(\underline{u})$. Furthermore, since $\mathcal{M}$ is a FOL/F-model, $\forall \underline{u}, \underline{u}' \in U.\ \mathcal{M}_h(\underline{u}') \neq \mathcal{M}_c(\underline{u})$. Since for all x, $\gamma(x) = h(t)$ for some h, t, it follows that $\forall \underline{u} \in U.\ \gamma(x) \neq \mathcal{M}_c(\underline{u})$. Thus $\forall \underline{u} \in U.\ \mathcal{M}_c(\underline{u}) \notin \{\eta(x), \mathcal{M}(\gamma(x)) : x \in \operatorname{dom}\gamma\}$. By definition of π, this implies

$$\forall \underline{u} \in U.\ \pi(\mathcal{M}_c(\underline{u})) = \mathcal{M}_c(\underline{u}). \tag{1}$$

We define a model $\mathcal{M}'$. This model $\mathcal{M}'$ has universe U, $\mathcal{M}'_c(\underline{u}) := \pi(\mathcal{M}_c(\pi^{-1}(\underline{u})))$, and $\mathcal{M}'_p(\underline{u}) := \mathcal{M}_p(\pi^{-1}(\underline{u}))$ for predicate symbols p.

It is left to define $\mathcal{M}'_h$. First note that $\mathcal{M}_h$ is injective. (By definition of FOL/F, for syntactic h we have $\mathcal{M} \vDash \forall \underline{y}, \underline{y}'.\ h(\underline{y}) \neq h(\underline{y}') \Rightarrow \underline{y} \neq \underline{y}'$ for any model $\mathcal{M}$.) Thus $U \times \cdots \times U$ and $\operatorname{range} \mathcal{M}_h$ have the same cardinality. Furthermore, as seen above, $\mathcal{M}(t) = \mathcal{M}(t')$ iff $t = t'$ for closed terms t, t' only containing syntactic function symbols. Thus we can fix $\mathcal{M}'_h$ to be some injective function with range $\operatorname{range} \mathcal{M}'_h = \operatorname{range} \mathcal{M}_h$ and satisfying: For all $h(t) \in \operatorname{range} \gamma$, we have $\mathcal{M}'_h(\mathcal{M}'(t)) = \mathcal{M}_h(\mathcal{M}(t))$.

Claim 1 *$\mathcal{M}'$ is a FOL/F-model.*

Obviously, $\mathcal{M}'$ is a FOL-model. Thus, to show Claim 1, we only need to show that for all syntactic function symbols $f \neq f'$, the axioms $\forall \underline{y}, \underline{y}'. f(\underline{y}) \neq f'(\underline{y}')$ (F Disjoint) and $\forall \underline{y}, \underline{y}'. f(\underline{y}) \neq f(\underline{y}') \Rightarrow \underline{y} = \underline{y}'$ (F Injective) are satisfied by $\mathcal{M}'$. Since $\mathcal{M}'_f$ is injective by definition both for forbidden and non-forbidden f, (F Injective) is satisfied. To show that (F Disjoint) is satisfied, we distinguish the cases $(f, f') = (c, c')$, $(f, f') = (h, h')$, and $(f, f') = (c, h')$ (the case $(f, f') = (h, c')$ is analogous). If $(f, f') = (c, c')$, then $\operatorname{range} \mathcal{M}_c \cap \operatorname{range} \mathcal{M}_{c'} = \varnothing$ (since $\mathcal{M}$ is a FOL/F-model and $c \neq c'$ are syntactic) and thus $\operatorname{range} \mathcal{M}'_c \cap \operatorname{range} \mathcal{M}'_{c'} = \pi(\operatorname{range} \mathcal{M}_c) \cap \pi(\operatorname{range} \mathcal{M}_{c'}) = \pi(\varnothing) = \varnothing$ (since π is a permutation on U). If $(f, f') = (h, h')$, we have $\operatorname{range} \mathcal{M}'_h \cap \operatorname{range} \mathcal{M}'_{h'} = \operatorname{range} \mathcal{M}_h \cap \operatorname{range} \mathcal{M}_{h'} = \varnothing$ by definition of $\mathcal{M}'_h, \mathcal{M}'_{h'}$ and since $\mathcal{M}$ is a FOL/F-model and $h \neq h'$ are syntactic. If $(f, f') = (c, h')$, we have that $\operatorname{range} \mathcal{M}_c \cap \operatorname{range} \mathcal{M}_{h'} = \varnothing$ since $\mathcal{M}$ is a FOL/F-model and $c \neq h'$ are syntactic. By (1), we have that $\operatorname{range} \mathcal{M}_c = \pi(\operatorname{range} \mathcal{M}_c)$. Thus $\operatorname{range} \mathcal{M}'_c \cap \operatorname{range} \mathcal{M}'_{h'} \overset{(*)}{=} \pi(\operatorname{range} \mathcal{M}_c) \cap \operatorname{range} \mathcal{M}_{h'} = \operatorname{range} \mathcal{M}_c \cap \operatorname{range} \mathcal{M}_{h'} = \varnothing$ where $(*)$ uses the definition of $\mathcal{M}'_c$ and $\mathcal{M}'_{h'}$. Thus, for all syntactic function symbols $f \neq f'$, we have $\operatorname{range} \mathcal{M}'_f \cap \operatorname{range} \mathcal{M}'_{f'} = \varnothing$. Thus (F Disjoint) is satisfied and Claim 1 holds.

Claim 2 *For all environments ζ with $\operatorname{dom} \zeta \cap \operatorname{dom} \eta = \varnothing$, and all terms t not containing forbidden function symbols and $fv(t) \subseteq \operatorname{dom} \eta \cup \operatorname{dom} \zeta$, we have that $\pi(\mathcal{M}_{\eta\cup\zeta}(t)) = \mathcal{M}'_{\pi\circ\zeta}(t\gamma)$.*

We show this by structural induction on t. We distinguish the following cases:

- Case "$t = c(t')$": $\pi(\mathcal{M}_{\eta\cup\zeta}(t)) = \pi(\mathcal{M}_c(\mathcal{M}_{\eta\cup\zeta}(t'))) \overset{\text{IH}}{=} \pi(\mathcal{M}_c(\pi^{-1}(\mathcal{M}'_{\pi\circ\zeta}(t'\gamma)))) \overset{(*)}{=} \mathcal{M}'_c(\mathcal{M}'_{\pi\circ\zeta}(t'\gamma)) = \mathcal{M}'_{\pi\circ\zeta}(c(t'\gamma)) = \mathcal{M}'_{\pi\circ\zeta}(t\gamma)$. Here $(*)$ uses the definition of $\mathcal{M}'_c$.
- Case "$t = h(t')$": This case does not occur because t does not contain forbidden function symbols.
- Case "$t = x \in \operatorname{dom} \eta$": Then $\pi(\mathcal{M}_{\eta\cup\zeta}(t)) = \pi(\eta(x))$ since $x \in \operatorname{dom} \eta$ and $\mathcal{M}'_{\pi\circ\zeta}(t\gamma) = \mathcal{M}'(\gamma(x))$ since $x \in \operatorname{dom} \gamma = \operatorname{dom} \eta$. By definition of π, $\pi(\eta(x)) = \mathcal{M}(\gamma(x))$. Furthermore, since $\gamma(x)$ is of the form $h(t')$, we have that $\mathcal{M}(\gamma(x)) = \mathcal{M}_h(\mathcal{M}(t')) = \mathcal{M}'_h(\mathcal{M}'(t')) = \mathcal{M}'(\gamma(x))$. Summarizing, $\pi(\mathcal{M}_{\eta\cup\zeta}(t)) = \pi(\eta(x)) = \mathcal{M}(\gamma(x)) = \mathcal{M}'(\gamma(x)) = \mathcal{M}'_{\pi\circ\zeta}(t\gamma)$.
- Case "$t = y \in \operatorname{dom} \zeta$": $\pi(\mathcal{M}_{\eta\cup\zeta}(t)) = \pi(\eta(y)) = \mathcal{M}'_{\pi\circ\zeta}(y)$.

We have shown Claim 2.

Claim 3 *For all environments ζ with $\operatorname{dom} \zeta \cap \operatorname{dom} \eta = \varnothing$, and all FOL/F-formulae Q not containing forbidden function symbols and $fv(Q) \subseteq \operatorname{dom} \eta \cup \operatorname{dom} \zeta$, we have that $\mathcal{M}, \zeta \cup \eta \vDash Q$ iff $\mathcal{M}', \pi \circ \zeta \vDash Q\gamma$.*

We show this by structural induction on t. We distinguish the following cases:

- Case "$Q = Q_1 \cdot Q_2$ for $\cdot \in \{\wedge, \vee, \Rightarrow\}$": Then $\mathcal{M}, \zeta \cup \eta \vDash Q \Leftrightarrow (\mathcal{M}, \zeta \cup \eta \vDash Q_1) \cdot (\mathcal{M}, \zeta \cup \eta \vDash Q_2) \overset{\text{IH}}{\Leftrightarrow} (\mathcal{M}', \pi \circ \eta \vDash Q_1\gamma) \cdot (\mathcal{M}', \pi \circ \eta \vDash Q_2\gamma) \Leftrightarrow \mathcal{M}', \pi \circ \zeta \vDash Q\gamma$. (This also covers $Q = \neg Q'$ because $\neg Q'$ is syntactic sugar for $Q' \Rightarrow \mathsf{false}$.)
- Case "$Q = \square y.Q'$ with $\square \in \{\forall, \exists\}$": For an environment ζ and a value $u \in U$, we abbreviate $\zeta(y := u)$ by ζ_u. Then

$$\begin{aligned}
&\mathcal{M}, \zeta \cup \eta \vDash Q \iff \square u \in U.\ (\mathcal{M}, \zeta_u \cup \eta \vDash Q') \\
&\overset{\text{IH}}{\iff} \square u \in U.\ (\mathcal{M}, \pi \circ \zeta_u \vDash Q'\gamma) \iff \square u \in U.\ (\mathcal{M}, \pi \circ \zeta_u \vDash Q'\gamma) \\
&\iff \square u \in U.\ (\mathcal{M}, (\pi \circ \zeta)_{\pi(u)} \vDash Q'\gamma) \iff \mathcal{M}', \pi \circ \zeta \vDash \square z.\ Q'\gamma \\
&\iff \mathcal{M}', \pi \circ \zeta \vDash Q\gamma.
\end{aligned}$$

- Case "$Q = \mathsf{false}$": Then both $\mathcal{M}, \zeta \cup \eta \vDash Q$ and $\mathcal{M}', \pi \circ \zeta \vDash Q$ do not hold.
- Case "$Q = (t = t')$":

$$\begin{aligned}
&\mathcal{M}, \eta \cup \zeta \vDash Q \iff \mathcal{M}_{\eta\cup\zeta}(t) = \mathcal{M}_{\eta\cup\zeta}(t') \\
&\iff \pi(\mathcal{M}_{\eta\cup\zeta}(t)) = \pi(\mathcal{M}_{\eta\cup\zeta}(t')) \overset{\text{Claim 2}}{\iff} \mathcal{M}'_{\pi\circ\zeta}(t\gamma) = \mathcal{M}'_{\pi\circ\zeta}(t'\gamma) \\
&\iff \mathcal{M}', \pi \circ \zeta \vDash t\gamma = t'\gamma. \iff \mathcal{M}', \pi \circ \zeta \vDash Q\gamma.
\end{aligned}$$

- Case "$Q = p(\underline{t})$ for a predicate symbol p": We use the abbreviation $\mathcal{M}_p(\underline{t})$ for $\mathcal{M}_p(t_1, t_2, \dots)$.

$$\begin{aligned}
&\mathcal{M}, \eta \cup \zeta \vDash Q \iff \mathcal{M}_p(\mathcal{M}_{\eta\cup\zeta}(\underline{t})) \\
&\overset{(*)}{\iff} \mathcal{M}'_p(\pi(\mathcal{M}_{\eta\cup\zeta}(\underline{t}))) \overset{\text{Claim 2}}{\iff} \mathcal{M}'_p(\pi(\mathcal{M}'_{\pi\circ\zeta}(\underline{t}\gamma))) \\
&\iff \mathcal{M}', \pi \circ \zeta \vDash p(\underline{t}\gamma). \iff \mathcal{M}', \pi \circ \zeta \vDash Q\gamma.
\end{aligned}$$

  Here $(*)$ uses the definition of $\mathcal{M}'_p$.

This shows Claim 3.

We can now conclude the proof of Lemma 3. Since $\mathcal{M}, \eta \vDash P$, we have $\mathcal{M}', \pi \circ \eta \vDash P\gamma$ by Claim 3 (special case with $\zeta = \varnothing$). Since $P\gamma \vdash C\gamma$ by assumption, it follows that $\mathcal{M}', \pi \circ \eta \vDash C\gamma$. By Claim 3 we get $\mathcal{M}, \eta \vDash C$. Since $\mathcal{M}, \eta \vDash C$ was the only remaining goal, $P, \mathit{eqs} \vdash C$ and thus Lemma 3 follows. $\square$

Lemma 4 *Let P, C be sets of message-free FOL/F-formulae. Assume that $\mathit{fv}(P,C) \cap \mathit{bv}(P,C) = \varnothing$. Let γ be a substitution mapping variables to closed FOL/F-terms. Assume that $\mathit{fv}(P,C) \subseteq \operatorname{dom}\gamma$. Assume that for all x, we have $\gamma(x) = h(t)$ for forbidden h and syntactic closed t. Let exterms be the set of subterms $h(t)$ of P, C such that h is forbidden. Assume that $\mathit{fv}(\mathit{exterms}) \cap \mathit{bv}(P,C) = \varnothing$ (i.e., the variables in exterms are not bound). Assume that all $t \in \mathit{exterms}$ are syntactic. Assume that for all $x \in \operatorname{dom}\gamma$ and all message-free FOL/F-terms $t \notin \operatorname{dom}\gamma$, we have $\gamma(x) \neq t\gamma$. Let*

$$\begin{aligned}
\mathit{eqs} := {}&\{x = x' : x \neq x',\ x, x' \in \operatorname{dom}\gamma, \gamma(x) = \gamma(x')\} \cup \{x \neq x' : x, x' \in \operatorname{dom}\gamma, \gamma(x) \neq \gamma(x')\} \\
&\cup \{\forall \underline{y}.\ x \neq c(\underline{y}) : x \in \operatorname{dom}\gamma,\ c \textit{ non-forbidden syntactic function symbol}\} \\
&\cup \{x \neq t : x \in \operatorname{dom}\gamma, t \in \mathit{exterms}\}.
\end{aligned}$$

Then $P\gamma \vdash C\gamma$ iff $P, \mathit{eqs} \vdash C$.

Proof. We first show the direction $P, \mathit{eqs} \vdash C \Rightarrow P\gamma \vdash C\gamma$. Since FOL/F is an authorization logic, $P, \mathit{eqs} \vdash C$ implies $P\gamma, \mathit{eqs}\,\gamma \vdash C\gamma$. Since for all x, $\gamma(x)$ is a closed term $h(t)$ containing only syntactic function symbols, we have $\vdash \gamma(x) = \gamma(x')$ for $\gamma(x) = \gamma(x')$ by (FOL Refl), and $\vdash \gamma(x) \neq \gamma(x')$ for $\gamma(x) \neq \gamma(x')$ by (F Disjoint) and (F Injective), and $\vdash \forall \underline{y}.\ \gamma(x) \neq c(\underline{y})$ for all c, x by (F Disjoint), and $\vdash \gamma(x) \neq t$ for all $t \in \mathit{exterms}$ by (F Disjoint) and (F Injective) and $\gamma(x) \notin \mathit{exterms}$. Thus $\vdash \mathit{eqs}\,\gamma$. Since FOL/F is an authorization logic, $P\gamma, \mathit{eqs}\,\gamma \vdash C\gamma$ and $\vdash \mathit{eqs}\,\gamma \vdash C\gamma$ implies $P\gamma \vdash C\gamma$. Thus we have shown $P, \mathit{eqs} \vdash C \Rightarrow P\gamma \vdash C\gamma$.

We proceed to show the direction $P\gamma \vdash C\gamma \Rightarrow P, \mathit{eqs} \vdash C$. Assume that $P\gamma \vdash C\gamma$ holds.

Let $\{t_1, \dots, t_n\} := \mathit{exterms}$ where the t_i are distinct. Let $x_1, \dots, x_n$ be fresh variables. Let $\sigma := \{t_1/x_1, \dots, t_n/x_n\}$ and $\bar\sigma := \{x_1/t_1, \dots, x_n/t_n\}$. Let $P' := P\bar\sigma$, $C' := C\bar\sigma$, $\gamma' := \gamma \cup \sigma\gamma$. Let

$$\begin{aligned}
\mathit{eqs}_0 := {}&\{x_i \neq x_j : i \neq j,\ t_i\gamma \neq t_j\gamma\} \cup \{x_i = x_j : i \neq j,\ t_i\gamma = t_j\gamma\} \\
&\cup \{\forall \underline{y}. x_i \neq c(\underline{y}) : c \text{ non-forbidden syntactic}\} \\
\mathit{eqs}' := {}&\{x = x' : x, x' \in \operatorname{dom}\gamma', \gamma'(x) = \gamma'(x')\} \cup \{x \neq x' : x, x' \in \operatorname{dom}\gamma', \gamma'(x) \neq \gamma'(x')\} \\
&\cup \{\forall \underline{y}.\ x \neq c(\underline{y}) : x \in \operatorname{dom}\gamma',\ c \text{ non-forbidden syntactic function symbol}\}.
\end{aligned}$$

Since $P'\gamma' = P\gamma$ and $C'\gamma' = C\gamma$, from $P\gamma \vdash C\gamma$ we get $P'\gamma' \vdash C'\gamma'$. P', C' contain no forbidden function symbols. Then, by Lemma 3, $P', \mathit{eqs}' \vdash C'$. Since FOL/F is an authorization logic, $P'\sigma, \mathit{eqs}'\sigma \vdash C'\sigma$.

Furthermore, we have that $\mathit{eqs}' = \mathit{eqs}\,\bar\sigma \cup \mathit{eqs}_0$. Thus $P'\sigma, \mathit{eqs}'\sigma \vdash C'\sigma$ implies $P, \mathit{eqs}, \mathit{eqs}_0\,\sigma \vdash C$.

We will now show that $\mathit{eqs} \vdash \mathit{eqs}_0\,\sigma$. For this, we first need the following two facts:

Claim 1 *For all message-free subterms t_1, t_2 of P, C (not only those of the form $h(\cdot)$) with $t_1\gamma = t_2\gamma$ and $fv(t_1, t_2) \subseteq \operatorname{dom}\gamma$, we have $eqs \vdash t_1 = t_2$.*

Claim 2 *For all subterms t_1, t_2 of P, C (not only those of the form $h(\cdot)$) with $t_1\gamma \neq t_2\gamma$ and $fv(t_1, t_2) \subseteq \operatorname{dom}\gamma$, we have $eqs \vdash t_1 \neq t_2$.*

We prove Claim 1 by structural induction on t_1, t_2 and distinguish the following cases: Case "$t_1 = f(t_1')$ and $t_2 = f'(t_2')$": Since $f(t_1'\gamma) = t_1\gamma = t_2\gamma = f'(t_2'\gamma)$, we have $f = f'$ and $t_1'\gamma = t_2'\gamma$. By induction hypothesis, $eqs \vdash t_1' = t_2'$. Thus $eqs \vdash f(t_1') = f(t_2')$ which is the same as $eqs \vdash t_1 = t_2$. Case "$t_1 = x$ and $t_2 = f(t_2')$": Since $t_2 \notin \operatorname{dom}\gamma$ is message-free, we have that $t_1\gamma = \gamma(x) \neq t_2\gamma$ in contradiction to the assumption $t_1\gamma = t_2\gamma$. Case "$t_1 = f(t_1')$ and $t_2 = x$": Analogous. Case "$t_1 = x$ and $t_2 = x'$": Since $\gamma(x) = t_1\gamma = t_2\gamma = \gamma(x')$, we have $(x_1 = x_2) \in eqs$. Thus $eqs \vdash t_1 = t_2$. This shows Claim 1.

We prove Claim 2 by structural induction on t_1, t_2 and distinguish the following cases: Case "$t_1 = f_1(t_1')$ and $t_2 = f_2(t_2')$ with $f_1 \neq f_2$": Then $eqs \vdash t_1 \neq t_2$ by (F Disjoint). Case "$t_1 = f(\underline{t}')$ and $t_2 = f(\underline{t}'')$": From $f(\underline{t}'\gamma) = t_1\gamma \neq t_2\gamma = f(\underline{t}''\gamma)$ we get $t_i'\gamma \neq t_i''\gamma$ from some i. Thus, by induction hypothesis, $eqs \vdash t_i' \neq t_i''$. By (F Injective), it follows that $eqs \vdash f(\underline{t}') \neq f(\underline{t}'')$ and thus $eqs \vdash t_1 \neq t_2$. Case "$t_1 = x$ and $t_2 = h(\underline{t}')$": Since t_2 is a subterm of P, C, $t_2 \in exterms$. Thus $(x \neq t_2) \in eqs$ and hence $eqs \vdash t_1 \neq t_2$. Case "$t_1 = x$ and $t_2 = c(\underline{t}')$": Since $(\forall \underline{y}.\ x \neq c(\underline{y})) \in eqs$, we have $eqs \vdash x \neq c(\underline{t}')$, thus $eqs \vdash t_1 \neq t_2$. Case "$t_1 = x_1$ and $t_2 = x_2$": Since $t_1\gamma \neq t_2\gamma$, we have $\gamma(x_1) \neq \gamma(x_2)$. Thus $(x_1 \neq x_2) \in eqs$ and hence $eqs \vdash t_1 \neq t_2$. The remaining cases are symmetric to the ones above. This shows Claim 2.

Now $eqs \vdash eqs_0\,\sigma$ follows: By Claim 1, we have $eqs \vdash t_i = t_j$ for all $i \neq j$ with $t_i\gamma = t_j\gamma$. By Claim 2, we have $eqs \vdash t_i \neq t_j$ for all i, j with $t_i\gamma \neq t_j\gamma$. And since $t_i = h(t')$ for some $h \neq c$ by definition, we get $eqs \vdash \forall \underline{y}.\ t_i \neq c(\underline{y})$ by (F Distinct). Thus $eqs \vdash eqs_0\,\sigma$ holds. Together with $P, eqs, eqs_0\,\sigma \vdash C$, since FOL/F is an authorization logic, this implies that $P, eqs \vdash C$. □

Lemma 5 *If $\mathsf{M} \vdash A_0$ and A_0 is robustly $\rightsquigarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-safe, then A_0 is robustly* SExec*-safe.*

Proof. Assume that A_0 is not SExec-safe. Then, at some step of the symbolic execution for A_0, we have that $P, eqs \nvdash C$ where P and C are the active assumes and assertions of A and eqs is as in Definition 19. By Lemma 2 and induction on the descending number of the iteration of the main loop of the symbolic execution (starting with $Q := ()$), we get that there is a RCF expression Q_0 valid for φ_0 such that

$$(Q_0 \upharpoonright A_0) \rightsquigarrow^* \nu\underline{n}'.() \upharpoonright A(\iota_m \circ \eta).$$

Since Q_0 is valid for $\varphi_0 = \varnothing$, we have that Q_0 does not contain assumptions or assertions, is pc-free and satisfies $fv(Q_0) \subseteq \operatorname{dom}\sigma_{\mathrm{DY}}^{\mathsf{M}}$ and $a_{chan} \notin fn(Q_0)$. With $O := (\lambda x.Q_0 \upharpoonright x)$ it follows that $OA \rightsquigarrow^* A(\iota_m \circ \eta)$ and that O is a $\sigma_{\mathrm{DY}}^{\mathsf{M}}$-opponent. Hence, since A_0 is robustly $\rightsquigarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-safe, $A(\iota_m \circ \eta)$ is statically $\sigma_{\mathrm{DY}}^{\mathsf{M}}$-safe. Hence $C(\iota_m \circ \eta)\sigma_{\mathrm{DY}}^{\mathsf{M}} \vdash P(\iota_m \circ \eta)\sigma_{\mathrm{DY}}^{\mathsf{M}}$. Let $\gamma := \sigma_{\mathrm{DY}}^{\mathsf{M}} \cup (\iota_m \circ \eta)$. (Note that η and $\sigma_{\mathrm{DY}}^{\mathsf{M}}$ have disjoint domains by construction.) Then $P\gamma \vdash C\gamma$.

Since all $\sigma_{\mathrm{DY}}^{\mathsf{M}}(x)$ are lambda-abstractions (by Definition 13), and all $\iota_m \circ \eta(x)$ are of the form $\mathsf{message}(\dots)$, we have that for all $x \in \operatorname{dom}\gamma$, $\gamma(x) = h(t)$ for forbidden h and closed t. Let $exterms$ be the set of subterms $h(t)$ of P, C such that h is forbidden. Since $\mathsf{M} \vdash A_0$, A_0 is message-free and, by induction over the number of iterations of the main loop of the symbolic execution, we get that P, C are message-free, that $fv(P, C) \subseteq \operatorname{dom}\gamma$, that $fv(P, C) \cap bv(P, C) = \varnothing$, that $fv(exterms) \cap bv(P, C) = \varnothing$, and that all $t \in exterms$ are syntactic. (For the last fact, note that all RCF-values are encoded as syntactic terms in FOL/F-formulae.)

We have that $\sigma_{\mathrm{DY}}^{\mathsf{M}}$ is equality-friendly, and $\iota_m \circ \eta(x) = \mathsf{message}(\cdot)$ is message-match-free for all x. Thus for all $x \in \operatorname{dom}\gamma$ and all message-free FOL/F-terms $t \notin \operatorname{dom}\gamma$, we have $\gamma(x) \neq t\gamma$.

Thus we can apply Lemma 4 and get $P, eqs' \vdash C$ where

$$\begin{aligned} eqs' := {} & \{x = x' : x \neq x',\ x, x' \in \operatorname{dom}\gamma, \gamma(x) = \gamma(x')\} \cup \{x \neq x' : x, x' \in \operatorname{dom}\gamma, \gamma(x) \neq \gamma(x')\} \\ & \cup \{\forall \underline{y}.\ x \neq c(\underline{y}) : x \in \operatorname{dom}\gamma,\ c \text{ non-forbidden syntactic function symbol}\} \\ & \cup \{x \neq t : x \in \operatorname{dom}\gamma, t \in exterms\}. \end{aligned}$$

Since $\gamma(x) \neq \gamma(x')$ for $x \notin \operatorname{dom} \iota_m \circ \eta$ or $x' \notin \operatorname{dom} \iota_m \circ \eta$ (this follows from the fact that $\sigma^{\mathsf{M}}_{\mathrm{DY}}$ is equality-friendly and $\iota_m \circ \eta(x) = \mathsf{message}(\dots))$, we get $eqs = eqs'$.

Thus $P, eqs \vdash C$. $\square$

5.3 Computational soundness of the DY-library

We will now use the CoSP framework [BHU09] to derive conditions under which robust SExec-safety implies robust computational safety. In order to do so, we first define a CoSP protocol Π_{A_0} that simultaneously captures the behavior of the symbolic execution and the one of the computational execution. Then, computational soundness results in the CoSP framework guarantee that the security of Π_{A_0} (interpreted symbolically) implies security of Π_{A_0} (interpreted computationally). Hence robust SExec-safety implies robust computational soundness. Together with the fact that $\rightarrow$-safety implies SExec-safety, we get our first computational soundness result for RCF.

Notice that the algorithm describing the symbolic execution performs only the following operations on CoSP-terms: Applying CoSP-constructors (this includes nonces) and CoSP-destructors, doing equality tests on terms, and sending and receiving terms. Hence this interactive machine can be realized as a CoSP protocol in the sense of Definition 6: The state of the machine SExec_{A_0} is used as a node identifier. However, CoSP-terms (i.e., the images of η) are not encoded directly into the node identifier; the node in which they were created (or received) is referenced instead. This is due to the fact that a CoSP protocol allows one to treat CoSP-terms only as black boxes. Note that the current program A will be encoded in the node identifier (as a bitstring). Operations on CoSP-terms can then be performed by using constructor and destructor nodes, and the input and output of CoSP-terms is handled using input/output nodes. Equality tests can be performed using the *equals*-destructor. Sending $(\mathbf{S}, E)$ to the adversary and receiving $(sync, j, k)$ and $(step, k)$ is realized using control nodes (assuming a suitable encoding of these values as bitstrings). A control node that sends $(\mathbf{S}, E)$ such that $(\mathbf{S}, E)$ is not statically equation-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safe is called a *failure node*. We call the resulting CoSP protocol Π_{A_0}.

Definition 22 (Efficiently decidable RCF expressions) *Let A_0 be an RCF expression. We call a formula F a* possible assertion *of A_0 iff there is an assertion F' in A_0 such that $F = F'\varphi$ for some substitution φ. Analogously we define* possible assumptions.

We call an RCF expression A_0 efficiently decidable *if for any set P of possible assumptions and any possible assertion C, it can be decided in polynomial-time whether $P \vdash C$.* $\diamond$

Theorem 1 *Assume a DY model* M *and a computational implementation* Impl. *Assume that* Impl *is a computationally sound implementation of* M *for a class $\mathcal{P}$ of CoSP protocols (Definition 11). Let $\sigma^{\mathsf{M}}_{\mathrm{DY}}$ be a DY library for* M.

Let A_0 be an efficiently decidable[11] *RCF expression with $\mathsf{M} \vdash A_0$ and $\Pi_{A_0} \in \mathcal{P}$.*

If $A_0\sigma^{\mathsf{M}}_{\mathrm{DY}}$ is robustly $\rightarrow$-safe or A_0 is robustly $\rightsquigarrow$-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safe, then A_0 is robustly computationally safe using Impl.

Proof. By Lemma 1, A_0 is robustly $\rightsquigarrow$-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safe. By Lemma 5, A_0 is robustly SExec-safe. By construction of Π_{A_0}, we have that A_0 is robustly SExec-safe iff the symbolic CoSP-execution of Π_{A_0} reaches failure nodes only with negligible probability. Let $\wp$ be the set of all sequences of node identifiers that do not contain failure nodes. Then Π_{A_0} symbolically satisfies the CoSP-trace property $\wp$. Since A_0 is efficiently decidable, it can be decided in polynomial-time whether a node is a failure node. Thus $\wp$ is an efficiently decidable trace property. Since Impl is a computationally sound implementation of M for a class $\mathcal{P}$ of CoSP protocols, and $\Pi_{A_0} \in \mathcal{P}$, Π_{A_0} computationally satisfies the CoSP-trace property $\wp$. Then, again by construction of Π_{A_0}, we have that A_0 is robustly computationally safe iff the computational CoSP-execution of Π_{A_0} never reaches a failure node. (For this, note that the computational execution is defined like the symbolic execution, except that it stores/sends bitstrings instead of terms,

[11] A_0 is efficiently decidable if, at runtime, no assertions occur for which it cannot be decided in polynomial-time whether they are entailed. A precise definition is given in the full version.

and applies the computational implementation of the constructors/destructors/nonces instead of the constructors/destructors/nonces themselves. The difference between the computational CoSP-execution and the symbolic CoSP-execution is the same.) Thus A_0 is robustly computationally safe with respect to Impl. □

5.4 Encryption and signatures

In the preceding section, we derived a generic computational soundness result for RCF programs (Theorem 1), parametric in the symbolic model. To apply that result to a specific symbolic model, we need a computational soundness result in CoSP for that particular model. In [BHU09], such a result is presented for a symbolic model supporting encryption, signatures, and arbitrary strings as payloads.

The symbolic model. We first specify the symbolic model $\mathsf{M}_{es} = (\mathsf{C}, \mathsf{N}, \mathsf{T}, \mathsf{D}, \vdash_{\text{CoSP}})$:

- Constructors and nonces: Let $\mathsf{C} := \{\mathsf{enc}/3, \mathsf{ek}/1, \mathsf{dk}/1, \mathsf{sig}/3, \mathsf{vk}/1, \mathsf{sk}/1, \mathsf{pair}/2, \mathsf{string}_0/1, \mathsf{string}_1/1, \mathsf{empty}/0, \mathsf{garbageSig}/2, \mathsf{garbage}/1, \mathsf{garbageEnc}/2\}$ and $\mathsf{N} := \mathsf{N}_P \cup \mathsf{N}_E$. Here N_P and N_E are countably infinite sets representing protocol and adversary nonces, respectively. Intuitively, encryption, decryption, verification, and signing keys are represented as $\mathsf{ek}(\mathsf{r}), \mathsf{dk}(\mathsf{r}), \mathsf{vk}(\mathsf{r}), \mathsf{sk}(\mathsf{r})$ with a nonce r (the randomness used when generating the keys). $\mathsf{enc}(\mathsf{ek}(\mathsf{r}'), \mathsf{m}, \mathsf{r})$ encrypts m using the encryption key $\mathsf{ek}(\mathsf{r}')$ and randomness r. $\mathsf{sig}(\mathsf{sk}(\mathsf{r}'), \mathsf{m}, \mathsf{r})$ is a signature of m using the signing key $\mathsf{sk}(\mathsf{r}')$ and randomness r. The constructors string_0, string_1, and empty are used to model arbitrary strings used as payload in a protocol (e.g., a bitstring 010 would be encoded as $\mathsf{string}_0(\mathsf{string}_1(\mathsf{string}_0(\mathsf{empty})))$). $\mathsf{garbage}$, $\mathsf{garbageEnc}$, and $\mathsf{garbageSig}$ are constructors necessary to express certain invalid terms the adversary may send, these constructors are not used by the protocol.
- Message type: We define $\mathbf{T}$ as the set of all terms M matching the following grammar:

$$\begin{aligned}\mathsf{M} ::= {} & E(\mathsf{ek}(\mathsf{n}), \mathsf{M}, \mathsf{n}) \mid \mathsf{ek}(\mathsf{n}) \mid \mathsf{dk}(\mathsf{n}) \mid \\ & \mathsf{sig}(\mathsf{sk}(\mathsf{n}), \mathsf{M}, \mathsf{n}) \mid \mathsf{vk}(\mathsf{n}) \mid \mathsf{sk}(\mathsf{n}) \mid \\ & \mathsf{pair}(\mathsf{M}, \mathsf{M}) \mid \mathsf{S} \mid \mathsf{n} \mid \\ & \mathsf{garbage}(\mathsf{n}) \mid \mathsf{garbageEnc}(\mathsf{M}, \mathsf{n}) \mid \\ & \mathsf{garbageSig}(\mathsf{M}, \mathsf{n}) \\ \mathsf{S} ::= {} & \mathsf{empty} \mid \mathsf{string}_0(\mathsf{S}) \mid \mathsf{string}_1(\mathsf{S})\end{aligned}$$

  where the nonterminal n stands for nonces.
- Destructors: $\mathsf{D} := \{\mathsf{dec}/2, \mathsf{isenc}/1, \mathsf{isek}/1, \mathsf{ekof}/1, \mathsf{verify}/2, \mathsf{issig}/1, \mathsf{isvk}/1, \mathsf{vkof}/2, \mathsf{fst}/1, \mathsf{snd}/1, \mathsf{unstring}_0/1, \mathsf{unstring}_1/1, \mathsf{equals}/2\}$. The destructors isek, isvk, isenc, and issig realize predicates to test whether a term is an encryption key, verification key, ciphertext, or signature, respectively. ekof extracts the encryption key from a ciphertext, vkof extracts the verification key from a signature. $\mathsf{dec}(\mathsf{dk}(\mathsf{r}), \mathsf{c})$ decrypts the ciphertext c. $\mathsf{verify}(\mathsf{vk}(\mathsf{r}), \mathsf{s})$ verifies the signature s with respect to the verification key $\mathsf{vk}(\mathsf{r})$ and returns the signed message if successful. The destructors fst and snd are used to destruct pairs, and the destructors $\mathsf{unstring}_0$ and $\mathsf{unstring}_1$ are used to parse payload-strings. (Destructors *ispair* and *isstring* are not necessary, they can be emulated using fst, $\mathit{unstring}_i$, and $\mathsf{equals}(\cdot, \mathit{empty})$.) The behavior of the destructors is given by the following rules; an application matching none of these rules evaluates to $\perp$:

$$\begin{aligned}
\mathsf{dec}(\mathsf{dk}(t_1), \mathsf{enc}(\mathsf{ek}(t_1), m, t_2)) &= m\\
\mathsf{isenc}(\mathsf{enc}(\mathsf{ek}(t_1), t_2, t_3)) &= \mathsf{enc}(\mathsf{ek}(t_1), t_2, t_3)\\
\mathsf{isenc}(\mathsf{garbageEnc}(t_1, t_2)) &= \mathsf{garbageEnc}(t_1, t_2)\\
\mathsf{isek}(\mathsf{ek}(t)) &= \mathsf{ek}(t)\\
\mathsf{ekof}(\mathsf{enc}(\mathsf{ek}(t_1), m, t_2)) &= \mathsf{ek}(t_1)\\
\mathsf{ekof}(\mathsf{garbageEnc}(t_1, t_2)) &= t_1\\
\mathsf{verify}(\mathsf{vk}(t_1), \mathsf{sig}(\mathsf{sk}(t_1), t_2, t_3)) &= t_2\\
\mathsf{issig}(\mathsf{sig}(\mathsf{sk}(t_1), t_2, t_3)) &= \mathsf{sig}(\mathsf{sk}(t_1), t_2, t_3)\\
\mathsf{issig}(\mathsf{garbageSig}(t_1, t_2)) &= \mathsf{garbageSig}(t_1, t_2)\\
\mathsf{isvk}(\mathsf{vk}(t_1)) &= \mathsf{vk}(t_1)\\
\mathsf{vkof}(\mathsf{sig}(\mathsf{sk}(t_1), t_2, t_3)) &= \mathsf{vk}(t_1)\\
\mathsf{vkof}(\mathsf{garbageSig}(t_1, t_2)) &= t_1\\
\mathsf{fst}(\mathsf{pair}(x, y)) &= x\\
\mathsf{snd}(\mathsf{pair}(x, y)) &= y\\
\mathsf{unstring}_0(\mathsf{string}_0(s)) &= s\\
\mathsf{unstring}_1(\mathsf{string}_1(s)) &= s\\
\mathsf{equals}(t_1, t_1) &= t_1
\end{aligned}$$

- Deduction relation: $\vdash_{\text{CoSP}}$ is the smallest relation such that $\mathsf{m} \in S \Rightarrow S \vdash_{\text{CoSP}} \mathsf{m}$, $\mathsf{n} \in \mathsf{N}_E \Rightarrow S \vdash_{\text{CoSP}} \mathsf{n}$, and such that for any constructor or destructor $\mathsf{f}/n \in \mathsf{C} \cup \mathsf{D}$ and for any $\mathsf{t}_1, \dots, \mathsf{t}_n \in \mathsf{T}$, with $\forall i \in [1, n]. S \vdash_{\text{CoSP}} \mathsf{t}_i$ and $\bot \neq \mathit{eval}_\mathsf{f}(\mathsf{t}_1, \dots, \mathsf{t}_n) \in \mathsf{T}$, we have $S \vdash_{\text{CoSP}} \mathsf{f}(\mathsf{t}_1, \dots, \mathsf{t}_n)$.

It is easy to see that M_{es} is a DY model in the sense of Definition 12.

CoSP [BHU09] also specifies conditions a computational implementation Impl for M_{es} should fulfill. Essentially, these conditions ensure that the encryption scheme used is IND-CCA secure, the signature scheme is strongly existentially unforgeable, and that certain conventions for tagging the different kinds of bitstrings are observed. We do not reproduce these conditions here but instead refer to [BHU09]. We will call these conditions the "enc-sig-implementation conditions".

Furthermore, [BHU09] imposes conditions on the CoSP protocol. These ensure that all encryptions and signatures are produced using fresh randomness and that secret keys are not sent around. A protocol satisfying these conditions is called *key-safe*.

Definition 23 *A CoSP protocol is* key-safe *if it satisfies the following conditions:*

1. *The argument of every* ek-*,* dk-*,* vk-*, and* sk-*computation node and the third argument of every* enc- *and* sig-*computation node is an* n-*computation node with* $\mathsf{n} \in \mathsf{N}_P$*. (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* enc-*computation node or of a* sig-*computation node on some path p occurs exactly once as an argument in that path p.*
5. *Every* dk-*computation node occurs only as the first argument of a* dec-*destructor node.*
6. *The first argument of a* dec-*destructor node is a* dk-*computation node.*
7. *Every* sk-*computation node occurs only as the first argument of a* sig-*computation node.*

8. *The first argument of a* sig*-computation node is an* sk*-computation node.*
9. *There are no computation nodes with the constructors* $\mathsf{garbage}$*,* $\mathsf{garbageEnc}$*,* $\mathsf{garbageSig}$*, or* $\mathsf{n} \in \mathsf{N}_E$*.* $\diamond$

Assuming that all these conditions are fulfilled, we get computational soundness for encryptions and signatures:

Theorem 2 (Computational soundness of encryptions and signatures [BHU09]) *If* Impl *satisfies the enc-sig-implementation conditions, then* Impl *is a computationally sound implementation of* M_{es} *for the class of key-safe protocols.*

When combining Theorem 2 with Theorem 1, we immediately get the following lemma:

Lemma 6 *Let* Impl *be a computational implementation satisfying the enc-sig-implementation conditions. Let* A_0 *be an efficiently decidable RCF expression such that* $\mathsf{M} \vdash A_0$ *and* Π_{A_0} *is key-safe.*

If $A_0\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ *is robustly* $\rightarrow$*-safe or* A_0 *is robustly* $\rightsquigarrow$*-*$\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$*-safe, then* A_0 *is robustly computationally safe using* Impl*.*

This lemma still has the drawback that one has to check whether Π_{A_0} is key-safe. To be able to simplify the lemma, we introduce a library $\sigma_{Highlevel}$ that serves as a wrapper for $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ and that ensures that a program A_0 that never directly calls $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ but only the wrappers from $\sigma_{Highlevel}$ will result in a key-safe Π_{A_0}. For example, $\sigma_{Highlevel}$ exports a function $\sigma_{Highlevel}(encrypt)$ that takes an encryption key and a plaintext, chooses a fresh nonce for randomness, and then invokes $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}(lib_{\mathsf{enc}})$. This ensures that the randomness-argument of $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}(lib_{\mathsf{enc}})$ is always a fresh nonce. Furthermore, the function $\sigma_{Highlevel}(enckeypair)$ picks a fresh nonce and uses that nonce to generate an encryption and a decryption key. The decryption key is wrapped using a private constructor *DecKey* so that it can only be used as an argument of $\sigma_{Highlevel}(decrypt)$. This ensures that keys are generated with fresh randomness and that the output of $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}(lib_{\mathsf{dk}})$ will only be used as the second argument to $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}(lib_{\mathsf{dec}})$.[12] For signatures and signing keys, we proceed similarly. "Harmless" functions such as pairs are simply exported by $\sigma_{Highlevel}$ (possibly with modified calling conventions for more convenient use, in particular for the functions related to payload strings). Functions that may never be called from the protocol, such as $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}(lib_{\mathsf{garbage}})$ are not exported by $\sigma_{Highlevel}$.

The exact definition of $\sigma_{Highlevel}$ is given in Figure 6. For increased readability, we use F#-syntax for the presentation of $\sigma_{Highlevel}$.

The next lemma states that $\sigma_{Highlevel}$ can be used to enforce key-safety.

Lemma 7 *Let* A_0 *be an RCF expression with* $\mathsf{M}_{es} \vdash A_0$ *and* $fv(A_0) \cap \operatorname{dom} \sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = \varnothing$ *and not containing the RCF-constructors DecKey and SigKey.*

Then $\Pi_{A_0\sigma_{Highlevel}}$ *is key-safe.*

Proof sketch. The thesis follows directly from an inspection of the code of $\sigma_{Highlevel}$. $\square$

Finally, we get computational soundness for encryptions and signatures with respect to programs using the DY library:

Theorem 3 (Computational soundness for $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$) *Let* Impl *be a computational implementation satsfying the enc-sig-implementation conditions. Let* A_0 *be an efficiently decidable RCF expression such that* $fv(A_0) \subseteq \sigma_{Highlevel}$*, A is pc-free, A does not contain the RCF-constructor DecKey or SigKey, and the FOL/F-formulae in A do not contain forbidden function symbols.*

Then, if $A_0\sigma_{Highlevel}\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ *is robustly* $\rightarrow$*-safe, then* $A_0\sigma_{Highlevel}$ *is robustly computationally safe using* Impl*.*

[12]Notice that this has the effect that keys may not be corrupted during the protocol execution (no adaptive corruption). It is, however, possible to model statically corrupted parties by subsuming them into the adversary and letting him choose their keys.

```
type bitstring = bool list
type 'a deckey = DecKey of 'a Lib.deckey
type 'a sigkey = SigKey of 'a Lib.sigkey

let valOf x = match x with Some m -> m

let encrypt (k,m) = valOf (Lib.enc (k,m,(Lib.nonce())))
let sign (SigKey k) m  = let n = Lib.nonce() in valOf (Lib.sign(k,m,n))

let nonce _ = Lib.nonce ()
let enckeypair usage = let r = nonce usage in (valOf (Lib.ek r), DecKey (valOf (Lib.dk r)))

let sigkeypair usage = let r = nonce usage in (SigKey (valOf (Lib.sk r)), valOf(Lib.vk r))
let decrypt ((DecKey dk),m) = Lib.dec (dk,m)
let verify vk s = Lib.verify (vk,s)

let pair x y = valOf (Lib.pair (x,y))
let frst x = Lib.frst x
let scnd x = Lib.scnd x

(* Instead of directly exporting payloadEmpty, payload0/1, unpayload0/1,
   we export the following more convenient wrappers *)
let empty_payload = valOf(Lib.payloadEmpty())
let rec payload str =
    match str with
        [] -> empty_payload
        | true::cs  -> let pcs = payload cs in valOf (Lib.payload1 pcs)
        | false::cs  -> let pcs = payload cs in valOf (Lib.payload0 pcs)
let rec unpayload msg =
    if msg = empty_payload then Some []
    else (match Lib.unpayload0 msg with
              Some m -> (match unpayload m with Some m' -> Some(false::m') | None -> None)
              | None -> (match Lib.unpayload1 msg with
                            Some m -> (match unpayload m with Some m' -> Some(true::m')
                                                                | None -> None)
                            | None -> None))

let send m = Lib.send m
let recv () = Lib.recv () : message

let ekof x = Lib.ekof x
let isenc x = Lib.isenc x
let isek x = Lib.isek x
let issig x = Lib.issig x
let isvk x = Lib.isvk x
let vkof x = Lib.vkof x
```

Figure 6: Definition of $\sigma_{Highlevel}$ using F# syntax. The domain of $\sigma_{Highlevel}$ consists of all functions defined here.

Proof. Let $A_0' := A_0\sigma_{Highlevel}$. Since $fv(A_0) \subseteq \sigma_{Highlevel}$ and $\operatorname{dom}\sigma_{Highlevel} \cap \operatorname{dom}\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = \varnothing$, $fv(A_0) \cap \operatorname{dom}\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = \varnothing$. Thus by Lemma 7, $\Pi_{A_0'}$ is key-safe. Furthermore, $\mathsf{M}_{es} \vdash A_0'$ since $\mathsf{M}_{es} \vdash \sigma_{Highlevel}(x)$ for all $x \in \operatorname{dom}\sigma_{Highlevel}$. Hence by Lemma 6, if $A_0'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ is robustly $\rightarrow$-safe, then A_0' is robustly computationally safe using Impl. $\square$

6 The sealing-based library

We first review the RCF sealing-based library and then show that programs that are robustly safe when linked to the sealing-based library are also robustly safe when linked to the Dolev-Yao library described in the previous sections.

6.1 Dynamic Sealing

The notion of *dynamic sealing* was initially introduced by Morris [Mor73] as a protection mechanism for programs. Later, Sumii and Pierce [SP03, SP07] studied the semantics of dynamic sealing within a λ-calculus, observing a close correspondence with symmetric-key cryptographic primitives.

In RCF [BBF+08] seals are encoded using pairs, functions, references[13], and lists. A seal is a pair of a *sealing function* and an *unsealing function* sharing a secret reference to a list. The sealing function takes as input a term M and checks whether the pair (M, N) is already stored in the list for some N. If it is not, then the sealing function returns a fresh value N, after adding the pair (M, N) to the secret list. Otherwise, the sealing function returns the value N that was previously stored in the list. The unsealing function takes as input a value N, scans the list in search of a pair (M, N), and returns M. Only the sealing function and the unsealing function can access this secret list. Each key-pair is (symbolically) implemented by means of a seal. In the case of public-key cryptography, the sealing function is used for encrypting (resp. signing), the unsealing function is used for decrypting (resp. verifying), and the fresh value N represents the encryption of (resp. signature on) M.[14]

From a computational point of view, the conceptual difference between the sealing-based library and the Dolev-Yao library is that the former relies on a global state (i.e., references to lists of pairs (M, N)).

The full code of the sealing-based library, in the following denoted as σ_{S}, is reported in Figure 7 and Figure 8. The following proposition recalls some important facts about σ_{S}.

Proposition 1 (Sealing-based Library) *The sealing-based library σ_{S} satisfies the following conditions:*

- *The range of σ_{S} only contains lambda-abstractions.*
- $\operatorname{dom}\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = \operatorname{dom}\sigma_{\mathrm{S}}$
- $\sigma_{\mathrm{S}}(nonce) = \mathsf{fun}\ \_ \rightarrow \nu a.\mathsf{message}\ C_\iota[a]$.
- $\sigma_{\mathrm{S}}(send) = (\mathsf{fun}\ x \rightarrow (\mathsf{match}\ x\ \mathsf{with}\ \mathsf{message}\ x'\ \mathsf{then}\ a_{chan}!x\ \mathsf{else}\ stuck))$. *Here stuck is a pure diverging RCF expression.*
- $\sigma_{\mathrm{S}}(recv) = \mathsf{fun}\ \_ \rightarrow a_{chan}?$
- $fv(\mathrm{range}(\sigma_{\mathrm{S}})) = \emptyset$ *and* $fn(\mathrm{range}(\sigma_{\mathrm{S}})) = a_{\mathsf{chan}}$.
- σ_{S} *is equality-friendly.*
- *The constructors* $\mathsf{N}, \mathsf{EK}, \ldots$ *used in the library are private.*

[13]As shown below, references are implemented via secret channels.

[14]The main advantage of the sealing-based library is polymorphism: the type of a seal is $\forall\alpha.(\alpha \rightarrow \mathsf{Un}) * (\mathsf{Un} \rightarrow \alpha)$, which states that the sealing function takes as input a message of an arbitrary type α and returns a message of type Un (the type of messages possibly known to the attacker) and, conversely, the unsealing function takes as input a message of type Un and returns a message of type α. Dolev-Yao libraries are not polymorphic but, as shown in [BFG10], they can be typed with refinement types that are expressive enough to verify a large number of protocol implementations.

```
// ********** Copied from list.fs from F7 package:
let rec mem x u = match u with
                    | y::v -> if x = y then true else mem x v
                    | _ -> false
let rec find p m = match m with
                    | x::xs -> if p x then x else find p xs
                    | [] -> failwith "not found"
let rec first f xs = match xs with
                        | x::xs -> (let r = f x in match r with
                                       Some(y) -> r
                                     | None -> first f xs)
                        | [] -> None
let left z (x,y) = if z = x  then Some y else None
let right z (x,y) = if z = y  then Some x else None
let rec map f xs = match xs with
                    | x::xs -> f x :: map f xs
                    | [] -> []
// ************* End: list.fs

let magic x = match x with Message x -> ()

// auxiliary functions for helping the F7-type checking
let cast_enc x = x
let cast_sig x = x
let cast_enckey x = x
let cast_verkey x = x

// Sealing
let deref: 'a ref -> 'a = fun x -> !x

let seal = fun s m ->
  let state =  deref s in match first (left m) state with
                          | Some(a) -> a
                          | None ->
                             let a = ref () in
                             s := ((m,a)::state); a

let unseal = fun s a ->
  let state = deref s in
  match first (right a) state with Some t' -> t'

let mkSeal() =
   let s = ref[] in
      (seal s, unseal s)
```

Figure 7: Definition of the SB-library σ_S. The exported functions are (i.e., dom σ_S) are `recv`, `send`, `nonce`, `pair`, `frst`, `scnd`, `ek`, `dk`, `ekof`, `isek`, `isenc`, `enc`, `dec`, `sign`, `verify`, `sk`, `vk`, `issig`, `isvk`, `vkof`, `payloadEmpty`, `payload0`, `payload1`, `unpayload0`, `unpayload1`, `garbage`, `garbageEnc`, `garbageSig`. Continued in Figure 8

```
let nonce () = magic; Nonce (mkSeal ())
let rec recv () = Pi.recv Pi.achan
let send m = Pi.send Pi.achan m

let pair p = magic; match p with (x,y) -> Some (Pair(x,y))
let frst pair = magic; match pair with (Pair (x,y)) -> Some x | _ -> None
let scnd pair = magic; match pair with (Pair (x,y)) -> Some y | _ -> None

let ek args = magic; match args with Nonce (s,u) -> Some (EK s) | _ -> None
let dk args = magic; match args with Nonce (s,u) -> Some (DK (s,u)) | _ -> None
let ekof msg      = magic; match cast_enc msg with Enc(k,e) -> Some k
                                                 | GarbageEnc(k,x) -> Some k | _ -> None
let isek msg      = magic; match cast_enckey msg with EK k -> Some msg | _ -> None
let isenc msg     = magic; match cast_enc msg with Enc(x,y) -> Some msg
                                                 | GarbageEnc(x,y) -> Some msg | _ -> None
let enc args = magic; match args with
     (EK key,msg,Nonce r)  -> Some (Enc (EK key, key (msg,Nonce r))) | _ -> None
let dec msg = magic; match msg with (DK (s,u), Enc(k,ciph)) ->
         if s=k then let msgrand = u ciph in (Some (fst msgrand)) else None
         | _ -> None

let sign args = magic; match args with
        (SK (sk,vk),msg, Nonce r) -> Some (Sign (VK vk, sk (msg,Nonce r)))
        | _ -> None
let verify args = magic; match args with
    ((VK key),(Sign(vk,sign))) -> if key=vk then let msgrand = key sign in  Some (fst msgrand) else
    | _ -> None
let sk args = magic; match args with (Nonce (s,u)) -> Some  (SK (s,u)) | _ -> None
let vk args = magic; match args with (Nonce (s,u)) -> Some  (VK u) | _ -> None
let issig msg = magic; match cast_sig msg with Sign(x,y) -> Some msg
                                             | GarbageSig(x,y) -> Some msg | _ -> None
let isvk msg  = magic; match cast_verkey msg with VK k -> Some msg | _ -> None
let vkof msg  = magic; match cast_sig msg with Sign(k,e) -> Some k
                                             | GarbageSig(k,e) -> Some k | _ -> None

let payloadEmpty () = magic; Some (Payload [])
let payload0' s = Some (Payload (false::s))
let payload0 msg = magic; match msg with Payload s -> payload0' s | _ -> None
let payload1' s = Some (Payload (true::s))
let payload1 msg = magic; match msg with Payload s -> payload1' s | _ -> None
let unpayload0 m = magic; match m with Payload (false::m') -> Some (Payload m') | _ -> None
let unpayload1 m = magic; match m with Payload (true::m') -> Some (Payload m') | _ -> None

let garbage N       = magic; match N with Nonce n -> Some (Garbage N) | _ -> None
let garbageEnc args = magic; match args with (EK k, Nonce n) -> Some (GarbageEnc (EK k, Nonce n))
                                             | _ -> None
let garbageSig args = magic; match args with (VK k, Nonce n) -> Some (GarbageSig (fst args, snd args
                                             | _ -> None
```

Figure 8: Definition of σ_S, continued.

6.2 Mapping DY-terms into SB-terms

In the next definition, we introduce some useful abbreviations and show the code implementing references and seals[15] A reference is a pair composed of two functions that read from and write to a private channel, respectively. Since each communication consumes one input and one output, the reading function returns the content of the reference after outputting it again on the private channel and, conversely, the writing function reads and discards the current content of the reference before updating it.

Definition 24 (References and Seals) *References are implemented using secret channels as follows:*

$$\begin{aligned}
C_{ref} &\overset{def}{=} \big((\lambda x.\mathsf{let}\ y = \Box?\ \mathsf{in}\ \Box!y; y), (\lambda x.\Box?; \Box!x)\big)\\
ref &\overset{def}{=} \lambda x.\nu a.(a!x \upharpoonleft C_{ref}[a])\\
!r &\overset{def}{=} \mathsf{let}\ (g,s) = r\ \mathsf{in}\ g()\\
r := v &\overset{def}{=} \mathsf{let}\ (g,s) = r\ \mathsf{in}\ s\ v
\end{aligned}$$

Seals are implemented using references and lists as follows:

$$\begin{aligned}
C'_{seal} &\overset{def}{=} \lambda m.\mathsf{let}\ state = !\Box\ \mathsf{in}\ (\mathsf{match}\ first\ (left\ m)\ state\ \mathsf{with\ some}\ x\ \mathsf{then}\ x\ \mathsf{else}\\
&\qquad \mathsf{let}\ x = ref()\ \mathsf{in}\ (\Box := ((m,x) :: state); x))\\
C'_{unseal} &\overset{def}{=} \lambda x.\mathsf{let}\ state = !\Box\ \mathsf{in}\ (\mathsf{match}\ first\ (right\ x)\ state\ \mathsf{with\ some}\ y\ \mathsf{then}\ y\ \mathsf{else}\ stuck)\\
C_{seal} &\overset{def}{=} C_{seal'}[C_{ref}[\Box]]\\
C_{unseal} &\overset{def}{=} C_{unseal'}[C_{ref}[\Box]]
\end{aligned}$$

where stuck is a pure diverging RCF expression, first (left m) state returns the first pair in the list state with m as first component, and first (right a) state returns the first pair in the list state with a as second component. The library functions seal and unseal are defined in terms of these contexts: $seal \overset{def}{=} \lambda x.C_{seal}[x]$ *and* $unseal \overset{def}{=} \lambda x.C_{unseal}[x]$. ◇

In the following, we will call DY-terms the RCF terms representing cryptographic messages in the DY-library and SB-terms the RCF terms representing cryptographic messages in the sealing-based library. In order to show that each execution of a program with the DY-library is matched by an execution with the sealing-based library, we need to map DY-terms to SB-terms. We could define this mapping directly, but this would make our result dependent on the specific implementation of the DY library. In order to make our result general, we decided instead to define a mapping from CoSP terms to SB-terms, which naturally induces a mapping from DY-terms to SB-terms via the embedding ι of CoSP terms into DY-terms (cf. Section 4.1).

We recall that the sealing-based library depends on a global hidden state, which tracks the cryptographic operations performed at run-time. For this reason, the mapping from CoSP terms to SB-terms has to depend on such a state.

A state ϕ is a pair of functions, denoted as (ϕ_S, ϕ_L). The former is a partial injective function from CoSP terms to RCF names, the latter is a partial injective function from CoSP terms to closed RCF values. Intuitively, each CoSP term (i.e., nonces, keys, ciphertexts, signatures, etc.) is implemented by means of a distinct seal and function ϕ_S is used to map CoSP terms to the name of the channel of the corresponding seal[16]. For instance, if a ciphertext M is implemented by means of *seal* a, then $\phi_S(\mathsf{M}) = a$

[15]Our implementation of references differs from the one proposed in [BBF+08], since the latter makes it possible to store multiple messages, which are then retrieved non-deterministically, and prevents one from reading several times from the same reference.

[16]Technically, the domain of ϕ_S consists of ciphertexts, signatures, and the randomness of keys and nonces. This asymmetry is due to the possible occurrence of different keys with the same randomness, which is not explicitly prevented in CoSP.

or if a key $\mathsf{ek}(\mathsf{k})$ is implemented by means of *seal* b, then $\phi_S(\mathsf{k}) = b$. The function ϕ_L is used to map each key to the secret list of pairs of the form $((M, R), N)$ stored in the corresponding reference, where M is the encrypted message, R is the randomness, and N is a fresh value representing the ciphertext. Storing the randomness along with the encrypted message allows for modelling probabilistic encryptions and probabilistic signatures. For instance, if M_1 and M_2 have been encrypted with randomness R_1 and R_2, respectively, and key k, then $\phi_L(\mathsf{k}) = [((M_1, R_1), N_1), ((M_2, R_2), N_2)]$, where N_1 and N_2 are the fresh values corresponding to the encryption of M_1 and M_2, respectively. In the following, we sometimes write M^{ϕ_S} for $\phi_S(\mathsf{M})$ and M^{ϕ_L} for $\phi_L(\mathsf{M})$.

Definition 25 (CoSP-terms to SB-terms) *Given a state ϕ, we define v-map$_\phi$ recursively as follows:*

$$\begin{aligned}
\textit{v-map}_\phi(\mathsf{n}) &= \mathsf{N}(C_{seal}[\mathsf{n}^{\phi_S}], C_{unseal}[\mathsf{n}^{\phi_S}]) \qquad\qquad (\mathsf{n} \in \mathsf{N})\\
\textit{v-map}_\phi(\mathsf{ek}(\mathsf{r})) &= \mathsf{EK}(C_{seal}[\mathsf{ek}(\mathsf{r}^{\phi_S})])\\
\textit{v-map}_\phi(\mathsf{dk}(\mathsf{r})) &= \mathsf{DK}(C_{seal}[\mathsf{r}^{\phi_S}], C_{unseal}[\mathsf{r}^{\phi_S}])\\
\textit{v-map}_\phi(\mathsf{vk}(\mathsf{r})) &= \mathsf{VK}(C_{unseal}[\mathsf{r}^{\phi_S}])\\
\textit{v-map}_\phi(\mathsf{sk}(\mathsf{r})) &= \mathsf{DK}(C_{seal}[\mathsf{r}^{\phi_S}], C_{unseal}[\mathsf{r}^{\phi_S}])\\
\textit{v-map}_\phi(\mathsf{enc}(\mathsf{ek}(\mathsf{k}), \mathsf{m}, \mathsf{r})) &= \mathsf{Enc}(\textit{v-map}_\phi(\mathsf{ek}(\mathsf{k})), C_{ref}[\mathsf{enc}(\mathsf{ek}(\mathsf{k}), \mathsf{m}, \mathsf{r})^{\phi_S}])\\
\textit{v-map}_\phi(\mathsf{sign}(\mathsf{sk}(\mathsf{k}), \mathsf{m}, \mathsf{r})) &= \mathsf{Sign}(\textit{v-map}_\phi(\mathsf{vk}(\mathsf{k})), C_{ref}[\mathsf{sign}(\mathsf{sk}(\mathsf{k}), \mathsf{m}, \mathsf{r})^{\phi_S}])\\
\textit{v-map}_\phi(\mathsf{pair}(\mathsf{m}_1, \mathsf{m}_2)) &= \mathsf{Pair}(\textit{v-map}_\phi(\mathsf{m}_1), \textit{v-map}_\phi(\mathsf{m}_2))\\
\textit{v-map}_\phi(\mathsf{garbage}(\mathsf{r})) &= \mathsf{Garbage}(\textit{v-map}_\phi(\mathsf{r}))\\
\textit{v-map}_\phi(\mathsf{garbageenc}(\mathsf{e}, \mathsf{r})) &= \mathsf{GarbageE}(\textit{v-map}_\phi(\mathsf{e}), \textit{v-map}_\phi(\mathsf{r}))\\
\textit{v-map}_\phi(\mathsf{garbagesign}(\mathsf{s}, \mathsf{r})) &= \mathsf{GarbageSig}(\textit{v-map}_\phi(\mathsf{s}), \textit{v-map}_\phi(\mathsf{r}))
\end{aligned}$$

and v-map$_\phi$(m) := $\mathsf{Payload}(f(\mathsf{m}))$ if m is a payload term, where $f(\mathsf{payloadEmpty}) = []$, $f(\mathsf{payload0}(\mathsf{m}')) = \mathsf{false} :: f(\mathsf{m}')$ and $f(\mathsf{payload1}(\mathsf{m}')) = \mathsf{true} :: f(\mathsf{m}')$.

◇

In the following, we focus on the DY library $\sigma^{\mathsf{M}_{es}}_{\mathrm{DY}}$ for the model M_{ES}.

The following definition introduces the notion of expression and state validity. Given an expression A, we let *CoSPterms*(A) denote $\{\iota^{-1}(M) : \mathsf{message}\ M \text{ is a subterm of } A\}$.

Definition 26 (Valid expressions and states) *An RCF expression A is valid if:*

- *A is a structure and $\nexists a, A'$ such that $A \equiv \nu a.A'$.*
- *$fv(A) \subseteq fv(\sigma^{\mathsf{M}_{es}}_{\mathrm{DY}})$.*
- *For every subterm* message *A' of A, we have that $A' \in \iota(\mathsf{T})$.*
- *A is mpc-free.*

Given a CoSP nonce k *and a state ϕ, we say that L is* k*-valid if L is a list of pairs of RCF values and all its entries are of the form $(N, C_{ref}[\mathsf{enc}(\mathsf{ek}(\mathsf{k}), \mathsf{M}, \mathsf{r})^{\phi_S}])$ or $(N, C_{ref}[\mathsf{sign}(\mathsf{sk}(\mathsf{k}), \mathsf{M}, \mathsf{r})^{\phi_S}])$ with $N = (\textit{v-map}_\phi(\mathsf{M}), \textit{v-map}_\phi(\mathsf{r}))$.*

A state ϕ is valid for an RCF expression A if for all CoSP nonces k, r, *and all CoSP terms* M *the following holds:*

- $\mathrm{dom}\,\phi_S = \mathrm{dom}\,\phi_L$.
- *If* $\mathsf{k} \in \mathrm{dom}\,\phi_L$, *then* k^{ϕ_L} *is a* k*-valid list. (Notice that this does not constrain* M^{ϕ_L} *for CoSP terms* M *other than nonces.)*
- *If* k *occurs in CoSPterms(A), then* $\mathsf{k} \in \mathrm{dom}\,\phi_S$.

- $\mathsf{enc}(\mathsf{ek}(\mathsf{k}),\mathsf{M},\mathsf{r}) \in \operatorname{dom}\phi_L$ *iff* $\mathsf{k} \in \operatorname{dom}\phi_L$ *and* $\exists N.(N, C_{\mathit{ref}}[a]) \in \mathsf{k}^{\phi_L}$ *with* $a := \mathsf{enc}(\mathsf{ek}(\mathsf{k}),\mathsf{M},\mathsf{r})^{\phi_S}$ *(and analogously for signatures).*
- *If* $\mathsf{enc}(\mathsf{ek}(\mathsf{k}),\mathsf{M},\mathsf{r})$ *is a subterm of CoSPterms(A), then* $\mathsf{enc}(\mathsf{ek}(\mathsf{k}),\mathsf{M},\mathsf{r}) \in \operatorname{dom}\phi_S$ *and* $\exists N.(N, C_{\mathit{ref}}[\mathsf{enc}(\mathsf{ek}(\mathsf{k}),\mathsf{M},\mathsf{r})^{\phi_S}]) \in \mathsf{k}^{\phi_L}$ *(and analogously for signatures).*

◇

We can finally formalize the mapping from a DY-expression to the corresponding SB-expression, which is obtained by replacing each DY-term with the corresponding SB-term and by adding the global state to the SB-expression. The state consists of the lists of encrypted values, each of them output on the private channel associated to the seal of the corresponding encryption key.

Definition 27 *Given a valid expression A and a state ϕ valid for A, let e-map$_\phi(A)$ be the result of replacing every* message *M occurring in A by v-map$_\phi(\iota^{-1}(M))$. Let s-map$_\phi(A) := \prod_{\mathsf{M}\in\operatorname{dom}\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonright$ e-map$_\phi(A)$. (Or s-map$_\phi(A) := \bot$ if e-map$_\phi(A) = \bot$).* ◇

6.3 Preservation of safety

In this section, we show that robust safety with respect to σ_{S} implies robust $\rightsquigarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$-safety. This is achieved by proving the existence of a simulation between executions with respect to the two libraries.

Lemma 8 (Preservation of Structural Equivalence) *Let A be valid and let ϕ be a valid state for A. If $A \equiv A'$, then s-map$_\phi(A) \equiv$ s-map$_\phi(A')$.*

Proof. The proof proceeds by induction on the derivation of $A \equiv A'$. We first consider the base cases:

- Struct Refl: straightforward.
- Struct Fork () : $A = () \upharpoonright B \equiv B = A'$.

  We have *s-map*$_\phi(() \upharpoonright B) = \prod_{\mathsf{M}\in\operatorname{dom}\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonright$ *e-map*$_\phi(() \upharpoonright B) = \prod_{\mathsf{M}\in\operatorname{dom}\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonright () \upharpoonright$ *e-map*$_\phi(B)$. By Struct Fork Assoc, Struct Fork Comm, and Struct Trans we get *s-map*$_\phi(() \upharpoonright B) \equiv () \upharpoonright \prod_{\mathsf{M}\in\operatorname{dom}\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonright$ *e-map*$_\phi(B)$. By Struct Fork () and Struct Trans, we get *s-map*$_\phi(() \upharpoonright B) \equiv \prod_{\mathsf{M}\in\operatorname{dom}\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonright$ *e-map*$_\phi(B) =$ *s-map*$_\phi(A')$.
- Struct Msg (): $A = a!M \equiv a!M \upharpoonright () = A'$.

  We have *s-map*$_\phi(a!M) = \prod_{\mathsf{M}\in\operatorname{dom}\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonright$ *e-map*$_\phi(a!M)$. By Struct Msg () and Struct Fork 2, we get *s-map*$_\phi(a!M) \equiv \prod_{\mathsf{M}\in\operatorname{dom}\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonright$ *e-map*$_\phi(a!M) \upharpoonright () =$ *s-map*$_\phi(A')$.
- Struct Assume : similar to the previous item.
- Struct Res Fork 1: $A = B' \upharpoonright \nu b.B \equiv \nu b.(B' \upharpoonright B) = A'$ if $b \notin fn(B')$.

  We have *s-map*$_\phi(B' \upharpoonright \nu b.B) = \prod_{\mathsf{M}\in\operatorname{dom}\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonright$ *e-map*$_\phi(B' \upharpoonright \nu b.B) = \prod_{\mathsf{M}\in\operatorname{dom}\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonright$ *e-map*$_\phi(B') \upharpoonright \nu b.$*e-map*$_\phi(B)$.

  By Struct Fork 2, Struct Res Fork 1, and Struct Trans, we get *s-map*$_\phi(B' \upharpoonright \nu b.B) \equiv \prod_{\mathsf{M}\in\operatorname{dom}\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonright \nu b.($*e-map*$_\phi(B') \upharpoonright$ *e-map*$_\phi(B)) =$ *s-map*$_\phi(A')$
- The remaining base cases follow similarly to the previous item.

We now discuss the induction step:

- Struct Res: $A = \nu b.B \equiv \nu b.B' = A'$ by $B \equiv B'$.

  By induction hypothesis, *s-map*$_\phi(B) \equiv$ *s-map*$_\phi(B')$. By Struct Res, we get *s-map*$_\phi(A) = \nu b.$*s-map*$_\phi(B) \equiv \nu b.$*s-map*$_\phi(B') =$ *s-map*$_\phi(A')$.

- The remaining induction cases follow by a similar argument.

$\square$

Lemma 9 *Let A be an expression such that $A \rightsquigarrow B$ and let ϕ be a valid state for A. Then there exist valid expressions A', B', a state ϕ' valid for B, lists of names $\underline{a}, \underline{b}, \underline{b}' \supseteq \underline{b}$ such that $a_{chan} \notin \underline{a}, \underline{b}'$, $A \equiv \nu\underline{a}.A'$, $B \equiv \nu\underline{b}.B'$, and $\nu\underline{a}.s\text{-}map_\phi(A')\sigma_\mathrm{S} \to^* \nu\underline{b}'.s\text{-}map_{\phi'}(B')\sigma_\mathrm{S}$.*

Proof. The proof proceeds by induction on the derivation of $A \rightsquigarrow A'$:

- *send*: $A = send\ M \rightsquigarrow a_{chan}!M = B$ and $s\text{-}map_\phi(A)\sigma_\mathrm{S} = \cdots \upharpoonleft \sigma_\mathrm{S}(send)\ e\text{-}map_\phi(M) \to \cdots \upharpoonleft a_{chan}!e\text{-}map_\phi(M) = s\text{-}map_\phi(B)$. Thus the lemma holds with $\phi' := \phi$, $A' := A$, $B' := B$, $\underline{a}, \underline{b}, \underline{b}' := \varnothing$.

- *recv*: $A = recv\ M \rightsquigarrow a_{chan}? = B$ and $s\text{-}map_\phi(A)\sigma_\mathrm{S} = \cdots \upharpoonleft \sigma_\mathrm{S}(recv)\ e\text{-}map_\phi(M) \to \cdots \upharpoonleft a_{chan}? = s\text{-}map_\phi(B)$. Thus the lemma holds with $\phi' := \phi$, $A' := A$, $B' := B$, $\underline{a}, \underline{b}, \underline{b}' := \varnothing$.

- *nonce*: $A = nonce\ M \rightsquigarrow \nu a.\mathsf{message}\ C_\iota[a] = B$.

  We choose $\underline{a} := \varnothing$, $A' = A$, $B' = \mathsf{message}\ C_\iota[a]$, $\underline{b} := a$. Notice that $A \equiv \nu\underline{a}.A'$ and $\nu\underline{b}.B' \equiv B$ by (Struct Refl).

  By Definition 27, we know that there exists M' and a such that $s\text{-}map_\phi(nonce\ M)\sigma_S = \prod_{\mathsf{M} \in \mathrm{dom}\,\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonleft \sigma_\mathrm{S}(nonce)M' \to^* \nu a. \prod_{\mathsf{M} \in \mathrm{dom}\,\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonleft (a![] \upharpoonleft \mathsf{N}(C_{seal}[a], C_{unseal}[a]))$. Notice that we applied the structural equivalence relation to move the restriction on top of the target expression.

  We set $\phi'_S := \phi_S[\mathsf{a} \mapsto a]$, $\phi'_L := \phi_L[\mathsf{a} \mapsto []]$, $\underline{b}' = a$. Notice that $s\text{-}map_{\phi'}(B')\sigma_\mathrm{S} = \prod_{\mathsf{M} \in \mathrm{dom}\,\phi'_S} \mathsf{M}^{\phi'_S}!\mathsf{M}^{\phi'_L} \upharpoonleft (\mathsf{N}(C_{seal}[a], C_{unseal}[a]))$.

- *enc* : $A = enc(\mathsf{message}\ \iota(\mathsf{M}_1), \mathsf{message}\ \iota(\mathsf{M}_2), \mathsf{message}\ \iota(\mathsf{M}_3)) \rightsquigarrow B$, where $B := \mathsf{some\ message}\ \iota(\mathsf{enc}(\mathsf{M}_1, \mathsf{M}_2, \mathsf{M}_3))$ or $B := \mathsf{none}$. (In all other cases, enc would be stuck.)

  We choose $\underline{a} = \varnothing$, $A' = A$, $B' = B$, and $\underline{b} = \varnothing$.

  If M_1 is not an encryption key or M_3 is not a nonce then $A \rightsquigarrow \mathsf{none}$; in this case $s\text{-}map_\phi(A)\sigma_\mathrm{S} \to^* \mathsf{none}$.

  Hence assume $\mathsf{M}_1 = \mathsf{ek}(\mathsf{k})$ and $\mathsf{M}_3 = \mathsf{r}$ for nonces k, r. Then $A \rightsquigarrow \mathsf{some\ message}\ \iota(\mathsf{enc}(\mathsf{ek}(\mathsf{k}), \mathsf{M}_2, \mathsf{r}))$. Let $t := (v\text{-}map_\phi(\mathsf{M}_2), v\text{-}map_\phi(\mathsf{r}))$.

  Case 1 "$(t, C_{ref}[a]) \in \mathsf{k}^{\phi_L}$ with $a := \mathsf{enc}(\mathsf{ek}(\mathsf{k}), \mathsf{M}_2, \mathsf{r})^{\phi_S}$". We set $\phi' := \phi$. By an inspection of Definition 26, we can easily see that ϕ' is valid for B.

  Notice that $s\text{-}map_\phi(A)\sigma_\mathrm{S} = \prod_{\mathsf{M} \in \mathrm{dom}\,\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonleft \sigma_\mathrm{S}(enc)(\mathsf{EK}(C_{seal}[\mathsf{k}^{\phi_S}]), v\text{-}map_\phi(\mathsf{M}_2), v\text{-}map_\phi(\mathsf{r}))$. Since $\mathsf{k}^{\phi_L} = [\ldots, (t, C_{ref}[a]), \ldots]$, the seal function will retrieve $C_{ref}[a]$. So $s\text{-}map_\phi(A) \to^*$ some $\prod_{\mathsf{M} \in \mathrm{dom}\,\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonleft \mathsf{Enc}(v\text{-}map_\phi(\mathsf{ek}(\mathsf{k})), C_{ref}[a]) = s\text{-}map_\phi(B)$. We finally get $s\text{-}map_\phi(A)\sigma_\mathrm{S} \to^* s\text{-}map_\phi(B)\sigma_\mathrm{S}$ by observing that $s\text{-}map_\phi(A)$ and $s\text{-}map_\phi(B)$ are closed.

  Case 2: "$\nexists a.(t, C_{ref}[a]) \in \mathsf{k}^{\phi_L}$". Notice that $s\text{-}map_\phi(A)\sigma_\mathrm{S} = \prod_{\mathsf{M} \in \mathrm{dom}\,\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \upharpoonleft \sigma_\mathrm{S}(enc)(\mathsf{EK}(C_{seal}[\mathsf{k}^{\phi_S}]), v\text{-}map_\phi(\mathsf{M}_2), v\text{-}map_\phi(\mathsf{r}))$. Since k^{ϕ_L} does not contain $[\ldots, (t, \ldots), \ldots]$, the seal function will append $(t, C_{ref}[a])$ for some fresh restricted name a and add $a!()$ to the state and will reduce to $\mathsf{some\ Enc}(v\text{-}map_\phi(\mathsf{ek}(k)), C_{ref}[a])$.

  Let $\underline{b}' := a$, $\phi'_S = \phi_S[\mathsf{enc}(\mathsf{ek}(\mathsf{k}), \mathsf{M}_2, \mathsf{r}) \mapsto a]$, $\phi'_L = \phi_L[\mathsf{k} \mapsto \mathsf{k}_L :: (t, C_{ref}[a]), \mathsf{enc}(\mathsf{ek}(\mathsf{k}), \mathsf{M}_2, \mathsf{r}) \mapsto ()]$. We have that ϕ' is valid for B. Then $s\text{-}map_{\phi'}(B')\sigma_\mathrm{S} = \prod_{\mathsf{M} \in \mathrm{dom}\,\phi'_S} \mathsf{M}^{\phi'_S}!\mathsf{M}^{\phi'_L} \upharpoonleft \mathsf{some\ Enc}(v\text{-}map_{\phi'}(\mathsf{ek}(\mathsf{k})), C_{ref}[a])$. Thus $s\text{-}map_\phi(A)\sigma_\mathrm{S} \to^* \nu a.s\text{-}map_{\phi'}(B')\sigma_\mathrm{S}$.

- *dec*: $A = \mathit{dec}(\mathsf{message}\ \iota(\mathsf{M}_1), \mathsf{message}\ \iota(\mathsf{M}_2)) \rightsquigarrow B$, where $B := \mathsf{some\ message}\ \iota(\mathsf{M})$ or $B := \mathsf{none}$. (In all other cases, dec would be stuck.)

  We choose $\underline{a} := \varnothing$, $A' := A'$, $B' = B$, $\underline{b} := \varnothing$. $A = \mathit{dec}(\mathsf{message}\ \iota(\mathsf{M}_1), \mathsf{message}\ \iota(\mathsf{M}_2))$.

  If M_1 is not a decryption key and M_2 is not an encryption with the corresponding encryption key, then $A \rightsquigarrow \mathsf{none}$ and $\mathit{s\text{-}map}_\phi(A)\sigma_{\mathrm{S}} \to^* \mathsf{none}$ and, by Struct Res, $\nu\underline{a}.\mathit{s\text{-}map}_\phi(A)\sigma_{\mathrm{S}} \to^* \nu\underline{a}.\mathsf{none}$ for all $\underline{a}$, as desired.

  Hence assume $\mathsf{M}_1 = \mathsf{dk}(\mathsf{k})$ and $\mathsf{M}_2 = \mathsf{enc}(\mathsf{ek}(\mathsf{k}), \mathsf{M}, \mathsf{r})$ for nonces k, r and a CoSP term M. Then $A \rightsquigarrow \mathsf{some\ message}\ \iota(\mathsf{M})$. Since ϕ is valid, $\mathsf{k}^{\phi_L} = [\dots, (t, C_{\mathit{ref}}[\mathsf{M}_2^{\phi_S}]), \dots]$ with $t := (\mathit{v\text{-}map}_\phi(\mathsf{M}), \mathit{v\text{-}map}_\phi(\mathsf{r}))$.

  By the definition of the decryption function of the sealing-based library, we have that $\mathit{s\text{-}map}_\phi(A)\sigma_{\mathrm{S}} = \mathit{s\text{-}map}_\phi(A)\sigma_{\mathrm{S}} = \prod_{\mathsf{M} \in \mathrm{dom}\,\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \Rsh \sigma_{\mathrm{S}}(\mathit{dec})(\mathsf{DK}(C_{\mathit{seal}}[\mathsf{k}^{\phi_S}], C_{\mathit{unseal}}[\mathsf{k}^{\phi_S}]), \mathsf{Enc}(\mathsf{EK}(C_{\mathit{seal}}[\mathsf{k}^{\phi_S}]), C_{\mathit{ref}}[\mathsf{M}_2^{\phi_S}])) \to^* \mathit{s\text{-}map}_\phi(B)\sigma_{\mathrm{S}} = \prod_{\mathsf{M} \in \mathrm{dom}\,\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \Rsh \mathsf{some}\ \mathit{v\text{-}map}_\phi(\mathsf{M}) = \mathit{s\text{-}map}_\phi(\mathsf{some\ message}\ \iota(\mathsf{M}))\sigma_{\mathrm{S}}$.

- *sign*: Analogous to enc.

- *verify*: Analogous to dec.

- *ek*: Straightforward, with $A = \mathit{ek}\ \mathsf{message}\ \iota(\mathsf{r})$, $B = \mathsf{some\ message}\ \iota(\mathsf{ek}(\mathsf{r}))$, $\underline{a} := \varnothing$, $b := \varnothing$, $\underline{b}' = \varnothing$. (If the argument of *ek* is not of the form $\mathsf{message}\ \iota(\mathsf{r})$, then $B := \mathsf{none}$.)

- *dk*: Analogous to ek.

- *isenc*: Straightforward, with $A = \mathit{isenc}(\mathsf{message}\ \iota(\mathsf{M}))$, $B = \mathsf{some\ message}\ \iota(\mathsf{M})$, $\underline{a} := \varnothing$, $b := \varnothing$, $\underline{b}' = \varnothing$. (If the argument of isenc is not of the form $\mathsf{message}\ \mathsf{E}(\mathsf{ek}(\iota(\mathsf{M}_1))), \iota(\mathsf{M}_2), \iota(\mathsf{M}_3))$ or $\mathsf{messageGarbageE}(\iota(\mathsf{M}_1), \iota(\mathsf{M}_2))$, then $B := \mathsf{none}$.)

- *ekof*: Analogous to isenc.

- *isek*: Analogous to isenc.

- sk: Analogous to ek.

- *vk*: Analogous to ek.

- *issig*: Analogous to isenc.

- *isvk*: Analogous to isenc.

- *vkof*: Analogous to isenc.

- *pair*: Straightforward, with $A = \mathit{pair}(\mathsf{message}\ \iota(\mathsf{M}_1), \mathsf{message}\ \iota(\mathsf{M}_2))$, $B = \mathsf{some\ message}\ \iota(\mathsf{pair}(\iota(\mathsf{M}_1), \iota(\mathsf{M}_2)))$, $\underline{a} := \varnothing$, $\underline{b} := \varnothing$, $\underline{b}' = \varnothing$.

- *frst*: $A = \mathit{frst}(\mathsf{message\ pair}(\iota(\mathsf{M}_1), \iota(\mathsf{M}_2)))$, $B = \mathsf{some\ message}\ \iota(\mathsf{M}_1)$, $\underline{a} := \varnothing$, $\underline{b} := \varnothing$, $\underline{b}' = \varnothing$.

- *scnd*: Similar to frst.

- *payloadEmpty*: $A = \mathit{payloadEmpty}()$, $B = \mathsf{some\ message\ PayloadEmpty}$, $\underline{a} := \varnothing$, $\underline{b} := \varnothing$, $\underline{b}' = \varnothing$.

- *payload0*: Similar to pair.

- *payload1*: Similar to pair.

- *unpayload0*: Similar to frst.

- *unpayload1*: Similar to frst.

- Red Fun: Straightforward, by an inspection of the reduction rule.
- Red Split: Straightforward, by an inspection of the reduction rule.
- Red Match: Straightforward, by observing that $h \neq \mathsf{message}$.
- Red Eq: $A = (M = N) \rightsquigarrow \mathsf{true} \mid \mathsf{false}$.

  We choose $\underline{a} := \varnothing$, $A' := A$, $B' = B$, $\underline{b} := \varnothing$.

  If $M\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = N\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$, $A \rightsquigarrow \mathsf{true}$, otherwise $A \rightsquigarrow \mathsf{false}$. By definition, $\textit{s-map}_\phi(A)\sigma_{\mathrm{S}} = \prod_{\mathsf{M} \in \mathrm{dom}\,\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \Rsh (\textit{e-map}_\phi(M)\sigma_{\mathrm{S}} = \textit{e-map}_\phi(N)\sigma_{\mathrm{S}})$, since $\textit{e-map}_\phi(A)\sigma_{\mathrm{S}}$ is equal to $(\textit{e-map}_\phi(M)\sigma_{\mathrm{S}} = \textit{e-map}_\phi(N)\sigma_{\mathrm{S}})$. Therefore $\textit{s-map}_\phi(A)\sigma_{\mathrm{S}} \rightsquigarrow \prod_{\mathsf{M} \in \mathrm{dom}\,\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \Rsh \mathsf{true}$ iff $\textit{e-map}_\phi(M)\sigma_{\mathrm{S}} = \textit{e-map}_\phi(N)\sigma_{\mathrm{S}}$. Thus all we need to show is that $M\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = N\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ iff $\textit{e-map}_\phi(M)\sigma_{\mathrm{S}} = \textit{e-map}_\phi(N)\sigma_{\mathrm{S}}$.

  Assume that this does not hold. Then there are subterms M', N' of M and N (at the same position), such that $M'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = N'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ not-iff $\textit{e-map}_\phi(M')\sigma_{\mathrm{S}} = \textit{e-map}_\phi(N')\sigma_{\mathrm{S}}$ and such that one of M', N' is a variable in $\mathrm{dom}\,\sigma_{\mathrm{S}} = \mathrm{dom}\,\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ or of the form $\mathsf{message}\ M''$. Wlog, we assume that M' has this property. Furthermore, $N' \neq M'$.

  Case 1 "$M' \in \mathrm{dom}\,\sigma_{\mathrm{S}}$": Since A is valid, N' is mpc-free. Then, since $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ is equality-friendly, $M'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = \sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}(M') \neq N'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$. $\mathrm{range}\,\textit{v-map}_\phi$ does not contain **match** ... **with** $\mathsf{message}\ldots$, so $\textit{e-map}_\phi(N')$ does not contain **match** ... **with** $\mathsf{message}\ldots$. Furthermore $\textit{e-map}_\phi(M') = M'$. Since σ_{S} is equality-friendly, $\textit{e-map}_\phi(M')\sigma_{\mathrm{S}} = \sigma_{\mathrm{S}}(M') \neq \textit{e-map}_\phi(N')\sigma_{\mathrm{S}}$. Thus $M'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = N'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ iff $\textit{e-map}_\phi(M')\sigma_{\mathrm{S}} = \textit{e-map}_\phi(N')\sigma_{\mathrm{S}}$.

  Case 2 "$M' = \mathsf{message}\ M''$ and $N' = \mathsf{message}\ N''$": Since A is valid, $M'', N'' \in \mathrm{range}\,\iota$, hence they are closed. Thus $M'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = M' \neq N' = N'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$. And $\textit{e-map}_\phi(M') = \textit{v-map}_\phi(\iota^{-1}(M')) \neq \textit{e-map}_\phi(N') = \textit{v-map}_\phi(\iota^{-1}(N'))$ since ι and $\textit{v-map}_\phi$ are injective. Thus $M'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = N'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ iff $\textit{e-map}_\phi(M')\sigma_{\mathrm{S}} = \textit{e-map}_\phi(N')\sigma_{\mathrm{S}}$.

  Case 3 "$M' = \mathsf{message}\ M''$ and N' is not of the form $\mathsf{message}\ N''$": Then $M'' \in \mathrm{range}\,\iota$ and M' is a closed value.

  If N' is a variable, $M'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = M' \neq \sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}(N')$ because $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ only contains lambda-expressions (this follows from the operational specification). If N' is not a variable, then $N'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ is not of the form $\mathsf{message}\ldots$, hence $M'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = M' \neq N'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$.

  If N' is a variable, $\textit{e-map}_\phi(M')\sigma_{\mathrm{S}} = \textit{v-map}_\phi(\iota^{-1}(M')) \neq \sigma_{\mathrm{S}}(N')$, since σ_{S} only contains lambda-expressions and $\mathrm{range}\,\textit{v-map}$ does not contain lambda-expressions, and thus $\textit{e-map}_\phi(M')\sigma_{\mathrm{S}} \neq \textit{e-map}_\phi(N')\sigma_{\mathrm{S}}$. If N' is not a variable, then the top-most syntactic construct of $\textit{e-map}_\phi(N')\sigma_{\mathrm{S}}$ is not an application of $\mathsf{message}$. Furthermore, since all constructors used in $\textit{v-map}$ are assumed to be encoded as constructor-chains starting with $\mathsf{message}$, $\textit{e-map}_\phi(M')\sigma_{\mathrm{S}} = \textit{v-map}_\phi(\iota^{-1}(M'))$ has a $\mathsf{message}$ constructor on top-level, and thus $\textit{e-map}_\phi(M')\sigma_{\mathrm{S}} \neq \textit{e-map}_\phi(N')\sigma_{\mathrm{S}}$. Thus $M'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = N'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ iff $\textit{e-map}_\phi(M')\sigma_{\mathrm{S}} = \textit{e-map}_\phi(N')\sigma_{\mathrm{S}}$.
- Red Comm: Straightforward, by an inspection of the reduction rule.
- Red Assert: Straightforward, by an inspection of the reduction rule.
- Red Let Val: Straightforward, by an inspection of the reduction rule.
- Red Let: We have $A = \mathsf{let}\ x = C\ \mathsf{in}\ D \rightsquigarrow \mathsf{let}\ x = C'\ \mathsf{in}\ D = B$ by $C \rightsquigarrow C'$.

  By induction hypothesis, we know that there exist valid RCF expressions C_*, C'_*, a state ϕ' valid for C', lists of names $\underline{c}_*$, $\underline{c}'_*, \underline{c}' \supseteq c'_*$ such that $C \equiv \nu\underline{c}_*.C_*$ and $\nu\underline{c}'_*.C'_* \equiv C'$ and $\nu\underline{c}_*.\textit{s-map}_\phi(C_*)\sigma_{\mathrm{S}} = \nu\underline{c}_*.\prod_{\mathsf{M} \in \mathrm{dom}\,\phi_S} \mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \Rsh \textit{e-map}_\phi(C_*)\sigma_{\mathrm{S}} \rightarrow^* \nu\underline{c}'.\prod_{\mathsf{M} \in \mathrm{dom}\,\phi'_S} \mathsf{M}^{\phi'_S}!\mathsf{M}^{\phi'_L} \Rsh \textit{e-map}_{\phi'}(C'_*)\sigma_{\mathrm{S}} = \nu\underline{c}'.\textit{s-map}_{\phi'}(C'_*)\sigma_{\mathrm{S}}$.

By Red Let, we have $\mathsf{let}\ x = \nu\underline{c}_*.\prod_{\mathsf{M}\in\mathrm{dom}\,\phi_S}\mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \Rsh \textit{e-map}_{\phi}(C_*)\sigma_{\mathrm{S}}\ \mathsf{in}\ \textit{e-map}_{\phi}(D)\sigma_{\mathrm{S}} \rightarrow^* \mathsf{let}\ x = \nu\underline{c}'.\prod_{\mathsf{M}\in\mathrm{dom}\,\phi'_S}\mathsf{M}^{\phi'_S}!\mathsf{M}^{\phi'_L} \Rsh \textit{e-map}_{\phi'}(C'_*)\sigma_{\mathrm{S}}\ \mathsf{in}\ \textit{e-map}_{\phi}(D)\sigma_{\mathrm{S}}$.

By Red Struct, Struct Fork Let, and Struct Res Let, we get $\nu\underline{c}_*.\mathsf{let}\ x = \prod_{\mathsf{M}\in\mathrm{dom}\,\phi_S}\mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \Rsh \textit{e-map}_{\phi}(C)\sigma_{\mathrm{S}}\ \mathsf{in}\ \textit{e-map}_{\phi}(D)\sigma_{\mathrm{S}} \rightarrow^* \nu\underline{c}'.\prod_{\mathsf{M}\in\mathrm{dom}\,\phi'_S}\mathsf{M}^{\phi'_S}!\mathsf{M}^{\phi'_L} \Rsh \mathsf{let}\ x = \textit{e-map}_{\phi'}(C'_*)\sigma_{\mathrm{S}}\ \mathsf{in}\ \textit{e-map}_{\phi}(D)\sigma_{\mathrm{S}}$.

The proof concludes by setting $\underline{a} := \underline{c}_*$, $\underline{b} := \underline{c}'_*$, $\underline{b}' := \underline{c}'$, $A' := \mathsf{let}\ x = C_*\ \mathsf{in}\ D$, and $B' := \mathsf{let}\ x = C'_*\ \mathsf{in}\ D$.

- Red Res: We have $A = \nu a.A' \rightsquigarrow \nu a.B' = B$ by $A' \rightsquigarrow B'$.

  By induction hypothesis, we know that there exist valid expressions A'', B'', a state ϕ' valid for B', lists of names $\underline{a}', \underline{b}'_*, \underline{b}''_* \supseteq \underline{b}'_*$ such that $A' \equiv \nu\underline{a}'.A''$, $\nu\underline{b}'_*.B'' \equiv B'$, and $\nu\underline{a}'.\textit{s-map}_{\phi}(A'')\sigma_{\mathrm{S}} \rightarrow^* \nu\underline{b}''_*.\textit{s-map}_{\phi'}(B'')\sigma_{\mathrm{S}}$.

  By Red Res, we have $\nu a, \underline{a}'.\textit{s-map}_{\phi}(A'')\sigma_{\mathrm{S}} \rightarrow^* \nu a, \underline{b}''_*.\textit{s-map}_{\phi}(B'')\sigma_{\mathrm{S}}$. The proof concludes by setting $\underline{a} := a, \underline{a}'$, $\underline{b} := a, \underline{b}'_*$, $\underline{b}' := a, \underline{b}''_*$, $A' := A''$, and $B' := B''$.

- Red Fork 1: We have $A = C \Rsh D \rightsquigarrow C' \Rsh D = B$ by $C \rightsquigarrow C'$.

  By induction hypothesis, we know that there exist valid expressions C_*, C'_*, a state ϕ' valid for C', lists of names $\underline{c}_*, \underline{c}'_*, \underline{c}' \supseteq \underline{c}'_*$ such that $C \equiv \nu\underline{c}_*.C_*$, $\nu\underline{c}'_*.C'_* \equiv C'$, and $\nu\underline{c}_*.\textit{s-map}_{\phi}(C_*)\sigma_{\mathrm{S}} \rightarrow^* \nu\underline{c}'.\textit{s-map}_{\phi'}(C'_*)\sigma_{\mathrm{S}}$.

  By Red Fork 1, we have $\nu\underline{c}_*.\prod_{\mathsf{M}\in\mathrm{dom}\,\phi_S}\mathsf{M}^{\phi_S}!\mathsf{M}^{\phi_L} \Rsh \textit{e-map}_{\phi}(C_*)\sigma_{\mathrm{S}} \Rsh \textit{e-map}_{\phi}(D)\sigma_{\mathrm{S}} \rightarrow^* \nu\underline{c}'.\prod_{\mathsf{M}\in\mathrm{dom}\,\phi'_S}\mathsf{M}^{\phi'_S}!\mathsf{M}^{\phi'_L} \Rsh \textit{e-map}_{\phi'}(C'_*)\sigma_{\mathrm{S}} \Rsh \textit{e-map}_{\phi}(D)\sigma_{\mathrm{S}}$.

  The proof concludes by setting $\underline{a} := c_*$, $\underline{b} := \underline{c}'_*$, $\underline{b}' := \underline{c}'$, $A' := C_* \Rsh D$, and $B' := C'_* \Rsh D$.

- Red Fork 2: Similar to Red Fork 1.

- Red Struct: We have $A \rightsquigarrow B$ by $A \equiv A'$, $A' \rightsquigarrow B'$, and $B' \equiv B$.

  By induction hypothesis, we know that there exist valid expressions A'', B'', a state ϕ' valid for B', lists of names $\underline{a}', \underline{b}', \underline{b}'' \supseteq \underline{b}'$ such that $A' \equiv \nu\underline{a}'.A''$, $B' \equiv \nu\underline{b}'.B''$, and $\nu\underline{a}'.\textit{s-map}_{\phi}(A'')\sigma_{\mathrm{S}} \rightarrow^* \nu\underline{b}''.\textit{s-map}_{\phi'}(B'')\sigma_{\mathrm{S}}$.

  By Struct Trans, $B \equiv \nu\underline{b}'.B''$. This concludes the proof, since by Struct Trans we get $A \equiv \nu\underline{a}'.A''$.

$\square$

Lemma 10 (Restrictions and Heating) *Forall a, A, B, C such that $C[\nu a.A] \equiv B$, there exists B', C' such that $B = C'[\nu a.B']$ and $C[A] \equiv C'[B']$.*

Proof. The proof is by straightforward induction on the derivation of $C[\nu a.A] \equiv B$. $\square$

Lemma 11 *Let A be an expression such that $A \rightsquigarrow^* B$. Assume that A is pc-free and $fv(A) \subseteq \mathrm{dom}\,\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$. Then there exists a valid expression B', a state ϕ valid for B', lists of names $\underline{b}$, $\underline{b}' \supseteq \underline{b}$ such that $a_{chan} \notin \underline{a}, \underline{b}'$, $\nu\underline{b}.B' \equiv B$, and $A\sigma_{\mathrm{S}} \rightarrow^* \nu\underline{b}'.\textit{s-map}_{\phi}(B')\sigma_{\mathrm{S}}$.*

Notice that the SB-execution introduces more restrictions than the DY-execution. The reason is that each SB-ciphertext is represented by a fresh value, which is implemented by a seal. Each of these seals introduces a restriction that does not have a counterpart in the DY-execution, where ciphertexts are represented by applying a private constructor to the encryption key, the encrypted value, and the randomness.

Proof. The proof is by induction on the length of the derivation of $A \rightsquigarrow^* B$. We first show the base case $A = B$. Let B' be a structure and $\underline{b}$ be a list of names such that $A \equiv \nu\underline{b}.B'$ and such that

$\nexists b'', B''.(B' \equiv \nu b''.B'')$. Let $\underline{b}' := \underline{b}$ and $\phi := \varnothing$. Since A is mpc-free and $fv(A) \subseteq \mathrm{dom}(\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}})$, the same holds for B'. Thus B' is valid by Definition 26. Since B' is mpc-free and $\phi = \varnothing$, we have that ϕ is valid for B' by Definition 26 and $\textit{s-map}_\phi(B') = B'$ by Definition 27. Hence $A\sigma_{\mathrm{S}} \equiv \nu\underline{b}.B'\sigma_{\mathrm{S}} = \nu\underline{b}'.B'\sigma_{\mathrm{S}} = \nu\underline{b}.\textit{s-map}_\phi(B')\sigma_{\mathrm{S}}$, so $A\sigma_{\mathrm{S}} \to^* \nu\underline{b}.\textit{s-map}_\phi(B')\sigma_{\mathrm{S}}$.

For the induction step, let us suppose that $A \rightsquigarrow^* B \rightsquigarrow C$. By induction hypothesis, we know that there exist a valid expression B', a state ϕ valid for B', lists of names $\underline{b}$ and $\underline{b}' \supseteq \underline{b}$ such that $\nu\underline{b}.B' \equiv B$, and $A\sigma_{\mathrm{S}} \to^* \nu\underline{b}'.\textit{s-map}_\phi(B')\sigma_{\mathrm{S}}$.

By Lemma 9, we know that there exist valid expressions B'_*, C', a state ϕ' valid for C, lists of names $\underline{b}_*$, $\underline{c}$, $\underline{c}' \supseteq \underline{c}$ such that $B \equiv \nu\underline{b}_*.B'_*$, $\nu\underline{c}.C' \equiv C$, and $\nu\underline{b}_*.\textit{s-map}_\phi(B'_*)\sigma_{\mathrm{S}} \to^* \nu\underline{c}'.\textit{s-map}_{\phi'}(C')\sigma_{\mathrm{S}}$.

By repeated application of Lemma 10 and by observing that heating does not cancel restrictions, it is easy to see that $\underline{b}$ is a permutation of $\underline{b}_*$ and $B' \equiv B'_*$.

By Lemma 8, $\textit{s-map}_\phi(B') \equiv \textit{s-map}_\phi(B'_*)$. By Struct Res, we get $\nu\underline{b}'.\textit{s-map}_\phi(B') \equiv \nu\underline{b}'.\textit{s-map}_\phi(B'_*)$. By an inspection of the heating rules, we can easily see that this implies $\nu\underline{b}'.\textit{s-map}_\phi(B')\sigma_{\mathrm{S}} \equiv \nu\underline{b}'.\textit{s-map}_\phi(B'_*)\sigma_{\mathrm{S}}$.

By repeated application of Lemma 15, it is easy to see that $\nu\underline{b}_*.\textit{s-map}_\phi(B'_*)\sigma_{\mathrm{S}} \to^* \nu\underline{c}'.\textit{s-map}_{\phi'}(C')\sigma_{\mathrm{S}}$ implies $\underline{c}' = \underline{b}_* \cup \underline{c}''$, for some fresh names $\underline{c}''$, and $\textit{s-map}_\phi(B'_*)\sigma_{\mathrm{S}} \to^* \nu\underline{c}''.\textit{s-map}_{\phi'}(C')\sigma_{\mathrm{S}}$. By Struct Res we get $\nu\underline{b}'.\textit{s-map}_\phi(B')\sigma_{\mathrm{S}} \to \nu\underline{b}', \underline{c}''.\textit{s-map}_{\phi'}(C')\sigma_{\mathrm{S}}$, as desired. $\square$

We now show that $\to$-safety with respect to σ_{S} implies $\rightsquigarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$-safety.

Lemma 12 *Fix an RCF expression A and a DY model M such that $\mathsf{M} \vdash A$. If $A\sigma_S$ is $\to$-safe then A is $\rightsquigarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$-safe.*

Proof. Assume that A is not $\rightsquigarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$-safe. Then $A \rightsquigarrow^* B$ for some structure B that is not statically $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$-safe. By Lemma 11, there exists a valid expression B', lists of names $\underline{b}$, $\underline{b}'$ with $a_{chan} \notin \underline{b}, \underline{b}'$ such that $\nu\underline{b}.B' \equiv B$ and $A\sigma_{\mathrm{S}} \to^* \nu\underline{b}'.\textit{s-map}_\phi(B')\sigma_{\mathrm{S}}$. For a structure $\mathbf{S}$, let $P(\mathbf{S})$ denote the active assumptions of $\mathbf{S}$, and $C(\mathbf{S})$ the active assertions of $\mathbf{S}$. Since B is not statically $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$-safe, $P(B)\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} \nvdash C(B)\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$. Since B' is valid, B' and $\nu\underline{b}.B'$ are structures. We have $P(\nu\underline{b}.B') = P(B')$ and $C(\nu\underline{b}.B') = C(B')$. Since $\nu\underline{b}.B' \equiv B$, $(P(\nu\underline{b}.B'), C(\nu\underline{b}.B')) = (P(B), C(B))$ up to renaming of names other than a_{chan} (the possibility of renaming stems from the fact that $\equiv$ allows for α-renaming of bound names). Hence also $(P(B')\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}, C(B')\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}) = (P(B)\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}, C(B)\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}})$ up to renaming of names. Hence $P(B')\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} \nvdash C(B')\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$.

By definition of $\rightsquigarrow$, and due to the fact that A is pc-free and $\mathsf{message}$ is private, we have that any FOL/F-subterm $\mathsf{message}\ t$ of B satisfies that t is syntactic and closed. By definition of $\equiv$, this implies that any FOL/F-subterm $\mathsf{message}\ t$ of B' and therefore of $P(B')$ and $C(B')$ is syntactic and closed. Let $\{t_1, \dots, t_n\}$ be the set of all t_i such that $\mathsf{message}\ t_i$ occurs in (C, P). Let $\eta_1 := \{\mathsf{message}\ t_1/x_1, \dots, \mathsf{message}\ t_n/x_n\}$ and $\bar{\eta}_1 := \{x_1/\mathsf{message}\ t_1, \dots, x_n/\mathsf{message}\ t_n\}$. Let $C^* := C(B')\bar{\eta}_1$ and $P^* := P(B')\bar{\eta}_1$. Then C^*, P^* do not contain the constructor $\mathsf{message}$. Since A is pc-free, and the only private constructor that $\rightsquigarrow$ can introduce is $\mathsf{message}$, we have that C^*, P^* are pc-free.

Let $\eta_2 := \{\textit{v-map}_\phi(\iota^{-1}(t_1))/x_1, \dots, \textit{v-map}_\phi(\iota^{-1}(t_n))/x_n\}$. Let $\gamma_1 := \sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} \cup \eta_1$. Let $\gamma_2 := \sigma_{\mathrm{S}} \cup \eta_2$. Then $P(B')\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = P^*\gamma_1$ and $C(B')\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = C^*\gamma_1$ and, by definition of $\textit{s-map}_\phi$, $P(\textit{s-map}_\phi(B'))\sigma_{\mathrm{S}} = P^*\gamma_2$ and $C(\textit{s-map}_\phi(B'))\sigma_{\mathrm{S}} = C^*\gamma_2$. We write $\mathrm{dom}\,\gamma$ for $\mathrm{dom}\,\gamma_1 = \mathrm{dom}\,\gamma_2$.

Thus from $P(B')\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} \nvdash C(B')\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$, we have $P^*\gamma_1 \nvdash C^*\gamma_1$. We will now apply Lemma 4 in order to show

$$\begin{aligned} & P^*\gamma_1 \vdash C^*\gamma_1 \iff P^*, \textit{eqs} \vdash C^* \qquad \text{and} && (2)\\ & P^*\gamma_2 \vdash C^*\gamma_2 \iff P^*, \textit{eqs} \vdash C^* && (3)\\ & \text{with} \qquad \textit{eqs} := \{x \neq x' : x, x' \in \mathrm{dom}\,\gamma,\ x \neq x'\} \\ & \qquad\qquad\qquad \cup \{\forall \underline{y}.\ x \neq c(\underline{y}) : x \in \mathrm{dom}\,\gamma,\ c \text{ non-forbidden syntactic}\} \\ & \qquad\qquad\qquad \cup \{x \neq t : x \in \mathrm{dom}\,\gamma,\ t \in \textit{exterms}\} \end{aligned}$$

where $\textit{exterms}$ is the set of subterms $h(t)$ of P^*, C^* with forbidden h.

To apply Lemma 4, we check the following:

- P^*, C^* are pc-free: This holds by definition of P^*, C^*, and $\bar{\eta}_1$.
- $fv(P^*, C^*) \cap bv(P^*, C^*) = \varnothing$: We can assume this without loss of generality since $\equiv$ is closed under α-renaming of bound variables.
- γ_1, γ_2 map variables to syntactic closed FOL/F-terms: This holds because the ranges of $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ and σ_{S} are closed by definition, and because the t_i are closed.
- For all x and $i = 1, 2$, $\gamma_i(x) = h(t)$ for forbidden h: $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}(x)$ and $\sigma_{\mathrm{S}}(x)$ are lambda-abstractions, and the function symbol representing lambda-abstractions is forbidden. $\eta_1(x) = \mathsf{message}(t)$, and $\mathsf{message}$ is forbidden. $\eta_2(x) = h(t)$ by definition of *v-map* where h is one of the private constructors listed in Definition 25. Thus $\eta_2(x) = h(t)$ for some forbidden h.
- $fv(exterms) \cap bv(P^*, C^*) = \varnothing$: Since A and $A\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ does not contain FOL/F-formulae that contain terms $h(t)$ with forbidden h, such terms can only be introduced in P^* and C^* by substituting a variable in a FOL/F-formula by an RCF-term. Hence the terms $h(t)$ do not contain variables that are bound in the FOL/F-formula.
- All $t \in exterms$ are syntactic: All terms $h(t)$ in P^*, C^* with forbidden h result from substituting a variable by an RCF-term (previous point), and RCF-terms are encoded as syntactic FOL/F-terms by definition.
- For $i = 1, 2$ and $x \in \operatorname{dom}\gamma$ and all pc-free FOL/F-terms $t \notin \operatorname{dom}\gamma_i$, we have $\gamma_i(x) \neq t\gamma_i$: If $x \in \operatorname{dom}\sigma_i$ (with $\sigma_1 = \sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ and $\sigma_2 = \sigma_{\mathrm{S}}$), we have that $\gamma_i(x) = \sigma_i(x) \neq t\eta_i\sigma_i = t\gamma_i$ since σ_i is equality-friendly and $t\eta_i$ is mpc-free. If t is a variable, then $t\gamma$ is a variable since $t \notin \operatorname{dom}\gamma$, and hence $\gamma_i(x) \neq t\gamma_i$ since $\gamma_i(x)$ is closed. If $x \notin \operatorname{dom}\sigma_i$ and t is not a variable, then $t = f(t')$ where f is not a private constructor, and $\gamma_i(x) = \eta_i(x)$. Furthermore, $\eta_1(x) = \mathsf{message}(t)$ by definition (and $\mathsf{message}$ is private), and $\eta_2(x) = \textit{v-map}_\phi(\dots) = h(t')$ where h is one of the forbidden constructors listed in Definition 25. Thus $\gamma_i(x) \neq t\gamma_i$.
- For $i = 1, 2$ and $x, x' \in \operatorname{dom}\gamma$ and $x \neq x'$, we have $\gamma_i(x) \neq \gamma_i(x')$: If $x, x' \in \operatorname{dom}\gamma$, we have $\gamma_i(x) \neq \gamma_i(x')$ because $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ and σ_{S} are equality-friendly. If $x, x' \notin \operatorname{dom}\gamma$, we have that $\gamma_i(x) = \eta_i(x) \neq \eta_i(x') = \gamma_i(x')$ because all t_i are distinct and $\textit{v-map}_\phi$ and ι are injective. If $x \in \operatorname{dom}\gamma$ and $x' \notin \operatorname{dom}\gamma$, we have that $\gamma_i(x') = \eta_i(x') = h(t)$ for a private constructor h and $\gamma_i(x')$ is closed, so $\gamma_i(x')$ is mpc-free. Since $\sigma_1 := \sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ and $\sigma_2 := \sigma_{\mathrm{S}}$ are equality-friendly, this implies that $\gamma_i(x) = \sigma_i(x) \neq h(t)\sigma_i = \gamma_i(x')$.

Thus the conditions of Lemma 4 are fulfilled and (2) and (3) follow.

From $P^*\gamma_1 \nvdash C^*\gamma_1$, (2), and (3), we get $P^*\gamma_2 \nvdash C^*\gamma_2$. Since $P(\nu\underline{b}'.\textit{s-map}_\phi(B')\sigma_{\mathrm{S}}) = P(\textit{s-map}_\phi(B'))\sigma_{\mathrm{S}} = P^*\gamma_2$ and $C(\nu\underline{b}'.\textit{s-map}_\phi(B')\sigma_{\mathrm{S}}) = C(\textit{s-map}_\phi(B'))\sigma_{\mathrm{S}} = C^*\gamma_2$, it follows that $P(\nu\underline{b}'.\textit{s-map}_\phi(B')\sigma_{\mathrm{S}}) \nvdash C(\nu\underline{b}'.\textit{s-map}_\phi(B')\sigma_{\mathrm{S}})$. Hence $\nu\underline{b}'.\textit{s-map}_\phi(B')\sigma_{\mathrm{S}}$ is not statically safe. Since $A\sigma_{\mathrm{S}} \rightarrow^* \nu\underline{b}'.\textit{s-map}_\phi(B')\sigma_{\mathrm{S}}$, this implies that $A\sigma_{\mathrm{S}}$ is not $\rightarrow$-safe. Thus, from the fact that A is not $\rightsquigarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$-safe, it follows that $A\sigma_{\mathrm{S}}$ is not $\rightarrow$-safe. By contraposition, the lemma follows. $\square$

We can finally state a main result of this section, i.e., safety with respect to σ_{S} implies $\rightsquigarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$-safety.

Lemma 13 *Fix an RCF expression A and a DY model M such that $\mathsf{M} \vdash A$. If $A\sigma_S$ is robustly $\rightarrow$-safe then A is robustly $\rightsquigarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$-safe.*

Proof. We first observe that for all $\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$-opponents O, $\mathsf{M} \vdash O$. The thesis follows directly from Lemma 12, Definition 14, and Definition 16. $\square$

6.4 Computational soundness

By combining the results from the previous section (relating the DY library and the SB library) with the computational soundness result for the DY library (Theorem 3), we get a computational soundness result for the SB library:

Theorem 4 (Computational soundness for σ_{S}) *Let* Impl *be a computational implementation satisfying the enc-sig-implementation conditions. Let A_0 be an efficiently decidable RCF expression such that*

$fv(A_0) \subseteq \sigma_{Highlevel}$, A is pc-free, A does not contain the RCF-constructor DecKey or SigKey, and the FOL/F-formulae in A do not contain forbidden function symbols.

Then, if $A_0\sigma_{Highlevel}\sigma_{\mathrm{S}}$ is robustly $\rightarrow$-safe, then $A_0\sigma_{Highlevel}$ is robustly computationally safe using Impl.

Proof. Let $A_0' := A_0\sigma_{Highlevel}$. We have that $\mathsf{M}_{es} \vdash A_0'$ since $\mathsf{M}_{es} \vdash \sigma_{Highlevel}(x)$ for all $x \in \operatorname{dom}\sigma_{Highlevel}$. Since $A_0'\sigma_{\mathrm{S}}$ is robustly $\rightarrow$-safe, by Lemma 13, $A_0'\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}}$ is robustly $\rightsquigarrow$-safe. Since $fv(A_0) \subseteq \sigma_{Highlevel}$ and $\operatorname{dom}\sigma_{Highlevel} \cap \operatorname{dom}\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = \varnothing$, $fv(A_0) \cap \operatorname{dom}\sigma_{\mathrm{DY}}^{\mathsf{M}_{es}} = \varnothing$. Thus by Lemma 7, $\Pi_{A_0'}$ is key-safe. By Lemma 6, if A_0' is robustly $\rightsquigarrow$-safe, then A_0' is robustly computationally safe using Impl. $\square$

We type-checked the library σ_{S} using F7. Exported functions are given polymorphic types as in [BBF+08], so we do not restrict the expressiveness of the verification technique. Since well-typed programs are robustly $\rightarrow$-safe [BBF+08], Theorem 4 implies that well-typed programs enjoy computational safety.

7 Conclusions

This paper presents a computational soundness result for F7, a type-checker for F# programs. We show the computational soundness of a generic DY library as well as the computational soundness of a sealing-based library. The proof is conducted in the CoSP framework and solely concerns the semantics of RCF programs, without involving any cryptographic arguments. This makes our result easily extensible to additional cryptographic primitives supported by CoSP. We remark that the proof does not depend on a specific verification technique, thus our computational soundness result would automatically apply to refinements of the type system, or even to a different analysis technique, as long as these use the same symbolic cryptographic libraries. To the best of our knowledge, this is the first computational soundness result for an automated verification technique of protocol implementations.

Acknowledgments. This work was partially funded by the Cluster of Excellence "Multimodel Computing and Interaction" (German Science Foundation), the Emmy Noether Programme (German Science Foundation), and Miur'07 Project SOFT (*Security Oriented Formal Techniques*).

A Symmetric semantics of RCF

We start this section by proving that making the heating relation symmetric does not affect the safety of programs. More formally let us define $\equiv$ according to the rules defining $\Rrightarrow$ plus the symmetric variant of the Heat Msg (), Heat Assume (), Heat Res Fork 1, Heat Res Fork 2 Heat Res Let, and Heat Fork Comm rules. Similarly, let us define $\rightarrow$ as $\rightarrow_a$, where the heating relation is defined by $\equiv$ instead of $\Rrightarrow$.

We let $C_0^{\mathrm{out}}[\cdot]$ range over the set of contexts defined by the following grammar:

$$\begin{array}{lcl} C_0^{\mathrm{out}}[\cdot] & = & [\cdot] \mid Z \Rsh C_0^{\mathrm{out}}[\cdot] \mid \mid C_0^{\mathrm{out}}[\cdot] \Rsh Z \\ Z & = & () \mid Z \Rsh Z \end{array}$$

Lemma 14 (Heating to output) *The set of processes ranged over by $C_0^{out}[a!M]$ is closed by $\equiv$.*

Proof. We prove that for all A and B such that $A \equiv B$, $A \notin C_0^{\mathrm{out}}[a!M]$ or $B \in C_0^{\mathrm{out}}[a!M]$. The proof is by induction on the derivation of $A \equiv B$.

The base cases (i.e., Heat Refl, Heat Fork (), Heat Msg (), Heat Fork Assoc, and Heat Fork Comm) follow directly from an inspection of the heating rule.

The inductive cases (i.e., Heat Trans, Heat Fork 1, and Heat Fork 2) follow straightforwardly from the induction hypothesis. $\square$

In the following, we let $|\mathcal{R}|_r$ denote the number of reduction rules used in the derivation of the relation $\mathcal{R} \in \{\rightarrow, \rightarrow_a\}$. We also let $|\mathcal{R}|_h$ denote the number of heating rules used in the derivation of $\mathcal{R} \in \{\equiv, \Rrightarrow\}$.

Heat Refl	$A \Rrightarrow A$
Heat Trans	$A \Rrightarrow A'', \text{if } A \Rrightarrow A' \text{ and } A' \Rrightarrow A''$
Heat Let	$\mathsf{let}\ x = A\ \mathsf{in}\ B \Rrightarrow \mathsf{let}\ x = A'\ \mathsf{in}\ B, \text{if } A \Rrightarrow A'$
Heat Res	$\nu a.TA \Rrightarrow \nu a.TA', \text{if } A \Rrightarrow A'$
Heat Fork 1	$A \Rsh B \Rrightarrow A' \Rsh B, \text{if } A \Rrightarrow A'$
Heat Fork 2	$B \Rsh A \Rrightarrow B \Rsh A', \text{if } A \Rrightarrow A'$
Equiv Fork ()	$() \Rsh A \equiv_a A$
Heat Msg ()	$a!M \Rrightarrow a!M \Rsh ()$
Heat Assume ()	$\mathsf{assume}\ C \Rrightarrow \mathsf{assume}\ C \Rsh ()$
Heat Res Fork 1	$A' \Rsh (\nu a.TA) \Rrightarrow \nu a.T(A' \Rsh A), \text{if } a \notin fn(A')$
Heat Res Fork 2	$(\nu a.TA) \Rsh A' \Rrightarrow \nu a.T(A \Rsh A'), \text{if } a \notin fn(A')$
Heat Res Let	$\mathsf{let}\ x = \nu a.TA\ \mathsf{in}\ B \Rrightarrow \nu a.T\mathsf{let}\ x = A\ \mathsf{in}\ B, \text{if } a \notin fn(B)$
Equiv Fork Assoc	$(A \Rsh A') \Rsh A'' \equiv_a A \Rsh (A' \Rsh A'')$
Heat Fork Comm	$(A \Rsh A') \Rsh A'' \Rrightarrow (A' \Rsh A) \Rsh A''$
Equiv Fork Let	$\mathsf{let}\ x = (A \Rsh A')\ \mathsf{in}\ B \equiv_a A \Rsh (\mathsf{let}\ x = A'\ \mathsf{in}\ B)$

Notation: We use $A \equiv_a A'$ to mean that both $A \Rrightarrow A'$ and $A' \Rrightarrow A$.

Figure 9: Heating relation $A \Rrightarrow A'$

Lemma 15 (Reduction, restriction, and heating) *For all a, A, A, B' such that $\nu a.A \equiv A'$ and $A' \to B'$, there exists B such that $A \to B$, $B' \equiv \nu a.B$, and $|A \to B|_r \leq |\nu a.A \to B'|_r$ (where $\nu a.A \to B'$ is proved by $A' \to B'$ and, if $\nu a.A \neq A'$, by $\nu a.A \equiv A'$ and Red Heat).*

Proof. The proof is by induction on $|A' \to B'|_r$.

The base case is trivial since $A' \to B'$ cannot be of length one assuming $\nu a.A \equiv A'$. This follows from an inspection of Red Fun,Red Split, Red Match, Red Eq, Red Comm, Red Assert, and Red Let Val and by observing that heating does not cancel restrictions.

For the induction step, we proceed by case analysis on the last rule applied:

Red Fun, Red Split, Red Match, Red Eq, Red Comm, Red Assert, Red Let Val These rules are not applicable.

Red Let $A' = \mathsf{let}\ x = C\ \mathsf{in}\ D \to \mathsf{let}\ x = C'\ \mathsf{in}\ D = B'$ and $C \to C'$ Since $A' \equiv \nu a.A$, there exists C'' such that $C \equiv \nu a.C''$ and $A \equiv \mathsf{let}\ x = C''\ \mathsf{in}\ D$.

Since $\nu a.C'' \to C'$, by induction hypothesis (the considered heating relation is $\nu a.C'' \equiv \nu a.C''$), we know that $C' \equiv \nu a.C'''$ for some C''' such that $C'' \to C'''$ and $|C'' \to C'''|_r \leq |\nu a.C'' \to C'|_r$. By Red Let, $\mathsf{let}\ x = C''\ \mathsf{in}\ D \to \mathsf{let}\ x = C'''\ \mathsf{in}\ D$. We set $B := \mathsf{let}\ x = C'''\ \mathsf{in}\ D$.

Notice that $|A \to B|_r \leq |C'' \to C'''|_r + 1$ (Red Let and Red Heat with $A \equiv \mathsf{let}\ x = C''\ \mathsf{in}\ D$) and $|C'' \to C'''|_r \leq |C \to C'|_r$ and $|\nu a.A \to B'|_r = |C \to C'|_r + 2$ (Red Let and Red Heat with $\nu a.A \equiv \mathsf{let}\ x = C\ \mathsf{in}\ D$). Therefore we preserve the invariant $|A \to B|_r \leq |\nu a.A \to B'|_r$.

Red Res $A' = \nu b.C \to \nu b.C' = B'$ and $C \to C'$ We have two cases, depending on whether $b = a$ or not. The former is trivial, the latter follows straightforwardly from the induction hypothesis.

Red Fork 1 $A' = C \Rsh D \to C' \Rsh D = B'$ and $C \to C'$ We must have either $C \equiv \nu a.C''$ or $D \equiv \nu a.D'$, for some C'', D'.

Assume that $D \equiv \nu a.D'$, for some D', i.e., $A \equiv C \Rsh D'$. We know that $C \to C'$. By Red Fork 1, $C \Rsh D' \to C' \Rsh D'$. We set $B := C' \Rsh D'$.

Notice that $|A \to B|_r \leq |C \to C'|_r + 2$ (Red Fork 1 and Red Heat with $A \equiv C \Rsh D'$) and $|\nu a.A \to B'|_r = |C \to C'|_r + 2$ (Red Fork 1 and Red Heat with $\nu a.A \equiv C \Rsh D$). Therefore the length invariant is fulfilled.

Assume that $C \equiv \nu a.C''$, for some C'', i.e., $A \equiv C'' \upharpoonleft D$.

Since $\nu a.C'' \rightarrow C'$, by induction hypothesis we know that there exists C''' such that $C'' \rightarrow C'''$, $C' \equiv \nu a.C'''$ and $|C'' \rightarrow C'''|_r \leq |\nu a.C'' \rightarrow C'|_r$.

By Red Fork 2, we get $C'' \upharpoonleft D \rightarrow C''' \upharpoonleft D$. We set $B := C''' \upharpoonleft D$.

Notice that $|A \rightarrow B|_r \leq |C'' \rightarrow C''''|_r + 2$ (Red Fork 2 and Red Heat with $A \equiv C'' \upharpoonleft D$) and $|C'' \rightarrow C'''|_r \leq |\nu a.C'' \rightarrow C'|_r$ and $|\nu a.A \rightarrow B'|_r = |\nu a.C'' \rightarrow C'|_r + 2$ (Red Fork 2 and Red Heat with $\nu a.A \equiv C \upharpoonleft D$). Therefore we have $|A \rightarrow B|_r \leq |\nu a.A \rightarrow B'|_r$.

Red Fork 2 $A' = C \upharpoonleft D \rightarrow C \upharpoonleft D' = B'$ and $D \rightarrow D'$ The reasoning is symmetric to Red Fork 1.

Red Heat $A' = C \rightarrow C' = B'$ and $C \equiv D$ and $D \rightarrow D'$ and $D' \equiv C'$ We have $\nu a.A \equiv C$, which implies $D \equiv \nu a.A$. By induction hypothesis, there exists B such that $D' \equiv \nu a.B$ and $|A \rightarrow B|_r \leq |\nu a.A \rightarrow D'|_r \leq |D \rightarrow D'|_r + 1 = |A' \rightarrow B'|_r$. Therefore the length invariant is fulfilled.

□

Lemma 16 (Symmetric reduction) *For every closed expressions A, A', B' such that $A \equiv A'$ and $A' \rightarrow B'$, there exists B such that $A \rightarrow_a B$, $B' \equiv B$, and one of the following conditions holds true:*

- *If $A = A'$, then $|A \rightarrow_a B|_r \leq |A' \rightarrow B'|_r$; otherwise $|A \rightarrow_a B|_r \leq |A' \rightarrow B'|_r + |A \equiv A'|_h$.*
- *If the derivation of $A' \rightarrow B'$ contains one application of Red Comm (say $a!M \upharpoonleft a? \rightarrow_a M$), then $A \rightarrow_a B$ is derived by Red Heat with hypotheses $A \Rrightarrow a!M \upharpoonleft a?$ and $a!M \upharpoonleft a? \rightarrow_a M$.*

Proof. The proof proceeds by simultaneous induction on $|A \equiv A'|_h$ and $|A' \rightarrow B'|_r$. The base case is when both the derivations have length one. We proceed by case analysis on the derivation of $A' \rightarrow B'$:

Red Fun $A' = (\lambda x.P)\ N \rightarrow P\{N/x\} = B'$ We proceed by case analysis on the derivation of $A \equiv A'$. The proof for Heat Refl is straightforward, since we know that $A = A'$. The proof for Heat Fork () follows by observing that this rule is symmetric even in $\Rrightarrow$.

Red Split, Red Match, Red Eq, Red Comm, Red Assert, Red Let Val The reasoning is similar since the only applicable heating rules are the same as those considered in Red Fun.

For the induction step, we proceed by case analysis on the last rule applied in the derivation of $A \rightarrow B$:

Red Fun $A' = (\lambda x.P)\ N \rightarrow P\{N/x\} = B'$ We proceed by case analysis on the last heating rule applied in the derivation of $A \equiv A'$. The only interesting case is Heat Trans. We know that $A \equiv A''$, $A'' \equiv A'$, and $|A \equiv A'|_h = |A \equiv A''|_h + |A'' \equiv A'|_h + 1$.

We can now apply the induction hypothesis, since $|A'' \equiv A'|_h + |A' \rightarrow B'|_r < |A \equiv A'|_h + |A' \rightarrow B'|_r$. By induction hypothesis, there exists B'' such that $A'' \rightarrow_a B''$ and $B' \equiv B''$. In addition, $|A'' \rightarrow_a B''|_r \leq |A' \rightarrow_a B'|_r + |A'' \equiv A'|_h$. (We are considering the only interesting case, i.e., $A \neq A' \neq A''$.) As $\rightarrow$ is larger than $\rightarrow$, we know that $A'' \rightarrow B''$ and $|A'' \rightarrow_a B''|_r = |A'' \rightarrow B''|_r$.

We can now apply the induction hypothesis, since $|A \equiv A''|_h + |A'' \rightarrow B''|_r < |A \equiv A'|_h + |A' \rightarrow B'|_r$. By induction hypothesis, there exists B such that $A \rightarrow_a B$ and $B'' \equiv B$, with $|A \rightarrow_a B|_r \leq |A'' \rightarrow B''|_r + |A \equiv A''|_h$. By Heat Trans, $B' \equiv B$.

Red Split, Red Match, Red Eq, Red Assert, Red Let Val The reasoning is similar to the previous case.

Red Comm $A' = a!M \Rsh a? \to M = B'$ We proceed by case analysis on the last heating rule applied in the derivation of $A \equiv A'$. The only interesting cases are Heat Fork 1 and Heat Trans.

Heat Fork 1. We know that $A = A'' \Rsh a? \equiv A'$, for some A'' such that $A'' \equiv a!M$. By Lemma 14, $A'' \in C_0^{\text{out}}[a!M]$. It is easy to see that by repeated application of Heat Fork Assoc, Heat Fork Comm, and Heat Fork () we can derive $A \Rrightarrow A'$. By Red Comm, $A' \to_a B'$. By Red Heat, we get the desired result. Notice that $|A \to_a B|_r = |A' \to B'|_r + 1$.

Heat Trans. We know that $A \equiv A''$, $A'' \equiv A'$, and $|A \equiv A'|_h = |A \equiv A''|_h + |A'' \equiv A'|_h + 1$.

We can now apply the induction hypothesis, since $|A'' \equiv A'|_h + |A' \to B'|_r < |A \equiv A'|_h + |A' \to B'|_r$. By induction hypothesis, $A'' \Rrightarrow A'$ and $A'' \to_a B'$ is proved by Red Heat. Since $\to$ is larger than $\to$, we know that $A'' \to B''$ and $|A'' \to_a B''|_r = |A'' \to B''|_r$.

We can now apply the induction hypothesis, since $|A \equiv A''|_h + |A'' \to B''|_r < |A \equiv A'|_h + |A' \to B'|_r$. By induction hypothesis, $A \Rrightarrow A''$. By Heat Trans, $A \Rrightarrow A'$. The result follows by Red Heat. Notice that $|A \to B'|_r = |A' \to B'|_r + 1$.

The remaining reduction rules are defined recursively on the reduction of a subprocess. By induction hypothesis, this reduction could contain one application of Red Comm or none. Since the proof is the same and the length invariant is preserved anyway, we assume that the derivation does not contain any application of Red Comm.

Red Let $A' = \mathsf{let}\ x = C\ \mathsf{in}\ D \to \mathsf{let}\ x = C'\ \mathsf{in}\ D = B'$ and $C \to C'$ The two interesting cases are when $A \equiv A'$ is proved by Heat Res Let or Heat Let.

Heat Res Let. Let us assume that $A \equiv A'$ is proved by (the symmetric variant of) Heat Res Let, i.e., there exists C'' such that $C = \nu a.C''$ and $A = \nu a.\mathsf{let}\ x = C''\ \mathsf{in}\ B$.

Since $\nu a.C'' \to C'$, by Lemma 15 we know that $C' \equiv \nu a.C'''$ for some C''' such that $C'' \to C'''$ and $|C'' \to C'''|_r \leq |\nu a.C'' \to C'|_r$. By induction hypothesis (we consider $C'' \equiv C''$), we know that there exists C'''' such that $C'' \to_a C''''$, $C''' \equiv C''''$, and $|C'' \to_a C''''|_r \leq |C'' \to C'''|_r$.

By Red Let, $\mathsf{let}\ x = C''\ \mathsf{in}\ D \to_a \mathsf{let}\ x = C''''\ \mathsf{in}\ D$ and, by Red Res, $A = \nu a.\mathsf{let}\ x = C''\ \mathsf{in}\ D \to \nu a.\mathsf{let}\ x = C''''\ \mathsf{in}\ D = B$. Notice that $|A \to B|_r = |C'' \to C'''|_r + 2$, while $|A' \to B'|_r = |\nu a.C'' \to C'|_r + 1$. Since $|C'' \to C'''|_r \leq |\nu a.C'' \to C'|_r$, we preserve the invariant $|A \to_a B|_r \leq |A' \to B'|_r + |A \equiv A'|_h$.

Since $C' \equiv \nu a.C'''$, by Heat Let we get $B' = \mathsf{let}\ x = C'\ \mathsf{in}\ D \equiv \mathsf{let}\ x = \nu a.C'''\ \mathsf{in}\ D$. Since $C''' \equiv C''''$, by Heat Let we get $\mathsf{let}\ x = \nu a.C'''\ \mathsf{in}\ D \equiv \mathsf{let}\ x = \nu a.C''''\ \mathsf{in}\ D$. By Heat Res Let, we get $\mathsf{let}\ x = \nu a.C''''\ \mathsf{in}\ D \equiv \nu a.\mathsf{let}\ x = C''''\ \mathsf{in}\ D$. Finally, by Heat Trans, we get $B' \equiv B$.

Heat Let. Let us assume now that $A \equiv A'$ is derived by Heat Let, i.e., there exists C'' such that $A = \mathsf{let}\ x = C''\ \mathsf{in}\ D$ and $C'' \equiv C$. By induction hypothesis, we know that there exists C''' such that $C'' \to_a C'''$ and $C' \equiv C'''$. By Red Let, $A = \mathsf{let}\ x = C''\ \mathsf{in}\ D \to \mathsf{let}\ x = C'''\ \mathsf{in}\ D = B$ and $B' \equiv B$ by Heat Let.

Red Res $A' = \nu a.C \to \nu a.C' = B'$ and $C \to C'$ The only interesting case is when $A \Rrightarrow A'$ is proved by Heat Res, i.e., there exists C'' such that $A = \nu a.C''$ and $C'' \equiv C$. By induction hypothesis, there exists C''' such that $C'' \equiv C'''$ and $C' \equiv C'''$. By Red Res, $A = \nu a.C'' \to_a \nu a.C''' = B$ and $B' \equiv B$ by Heat Res.

Red Fork 1 $A' = C \Rsh D \to C' \Rsh D = B'$ and $C \to C'$ The proof for Heat Fork 1 follows straightforwardly from the induction hypothesis. The proof for Heat Fork 2 follows from an inspection of the heating and reduction rules. We now reason on the two interesting cases, namely, Heat Res Fork 1 and Heat Res Fork 2.

Heat Res Fork 1. We know that $D = \nu a.D'$, for some D' and $A = \nu a.C \Rsh D'$. We know that $C \to_a C'$. By Red Res and Red Fork 1, $A = \nu a.C \Rsh D' \to \nu a.C' \Rsh D = B$. By Heat Res Fork 1, $B' \equiv B$. Notice that $|A \to_a B|_r = |A' \to B'|_r + 1$, which fulfills the length invariant.

Heat Res Fork 2. We know that $C = \nu a.C''$, for some C'', and $A = \nu a.C'' \Rsh D$.

Since $\nu a.C'' \to C'$, by Lemma 15 we know that there exists C''' such that $C'' \to C'''$ and $C' \equiv \nu a.C'''$ and $|C'' \to C'''|_r \leq |\nu a.C'' \to C'|_r$. By induction hypothesis (we consider $C'' \equiv C''$), we know that there exists C'''' such that $C'' \to_a C''''$ and $C''' \equiv C''''$ and $|C'' \to_a C''''|_r \leq |C'' \to_a C'''|_r$.

By Red Fork 1 and Red Res, $A = \nu a.C'' \Rsh D \to \nu a.C'' \Rsh D = B$. By Heat Res Fork 2, we get $B' \equiv B$.

Notice that $|A \equiv A'|_h = 1$, $|A' \to_a B'|_r = |C \to C'|_r + 1$, $|C \to C'|_r \geq |C'' \to C'''|_r \geq |C'' \to_a C''''|_r$, and $|A \to_a B|_r = |C'' \to_a C''''|_r + 2$. Therefore $|A \to B|_r \leq |A' \to_a B'|_r + |A \equiv A'|_h$.

Red Fork 2 $A' = C \Rsh D \to C \Rsh D' = B'$ and $D \to D'$ Symmetric to Red Fork 1.

Red Heat The proof follows directly from the induction hypothesis.

□

References

[ABW06] M. Abadi, M. Baudet, and B. Warinschi. Guessing attacks and the computational soundness of static equivalence. In *Proc. 9th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 3921 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2006.

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 104–115, New York, NY, USA, 2001. ACM Press.

[AF06] Pedro Adão and Cédric Fournet. Cryptographically sound implementations for communicating processes. In *Proc. ICALP*, pages 83–94, 2006.

[AG97] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. 4th ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

[AJ01] Martín Abadi and Jan Jürjens. Formal eavesdropping and its computational interpretation. In *Proc. 4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 82–94, 2001.

[AR02] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[BBF$^+$08] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *Proc. 21st IEEE Security Foundations Symposium (CSF)*, pages 17–32, 2008. Full version is Microsoft Research technical report MSR-TR-2008-118.

[BCFZ08] Karthikeyan Bhargavan, Ricardo Corin, Cédric Fournet, and Eugen Zălinescu. Cryptographically verified implementations for TLS. In *15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 459–468. ACM Press, 2008.

[BCK05] M. Baudet, V. Cortier, and S. Kremer. Computationally sound implementations of equational theories against passive adversaries. In *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 of *Lecture Notes in Computer Science*, pages 652–663. Springer, 2005.

[BFG10] K. Bhargavan, C. Fournet, and A.D. Gordon. Modular verification of security protocol code by typing. In *Proc. 37th Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2010.

[BFGT06] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *Proc. 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 139–152. IEEE, 2006.

[BHU09] Michael Backes, Dennis Hofheinz, and Dominique Unruh. CoSP: A general framework for computational soundness proofs. In *ACM CCS 2009*, pages 66–78. ACM Press, November 2009. Full version on IACR ePrint 2009/080. Some of the definitions we use only occur in the full version.

[Bla01] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 82–96. IEEE Computer Society Press, 2001.

[Bla06] Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy, Proceedings of SSP 2006*, pages 140–154. IEEE Computer Society, 2006. Extended version online available at `http://eprint.iacr.org/2005/401.ps`.

[BMU] Michael Backes, Matteo Maffei, and Dominique Unruh. Library source code with F7 type-checking annotations. Available at `http://crypto.m2ci.org/unruh/misc/rcf/library.zip`.

[BMV04] David Basin, Sebastian Mödersheim, and Luca Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 2004.

[BP04] Michael Backes and Birgit Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *Proc. 17th IEEE Computer Security Foundations Workshop (CSFW)*, pages 204–218, 2004.

[BPW03a] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A composable cryptographic library with nested operations (extended abstract). In *Proc. 10th ACM Conference on Computer and Communications Security*, pages 220–230, 2003. Full version in IACR Cryptology ePrint Archive 2003/015, Jan. 2003, `http://eprint.iacr.org/`.

[BPW03b] Michael Backes, Birgit Pfitzmann, and Michael Waidner. Symmetric authentication within a simulatable cryptographic library. In *Proc. 8th European Symposium on Research in Computer Security (ESORICS)*, volume 2808 of *Lecture Notes in Computer Science*, pages 271–290. Springer, 2003.

[BPW07] Michael Backes, Birgit Pfitzmann, and Michael Waidner. The reactive simulatability (RSIM) framework for asynchronous systems. *Information and Computation*, 205(12):1685–1720, 2007.

[CD09] Sagar Chaki and Anupam Datta. Aspier: An automated framework for verifying security protocol implementations. In *Proc. 22nd IEEE Computer Security Foundations Symposium (CSF)*, pages 172–185. IEEE, 2009.

[CL08] Hubert Comon-Lundh. About models of security protocols (abstract). In Ramesh Hariharan, Madhavan Mukund, and V Vinay, editors, *Proc. FSTTCS*, Dagstuhl, Germany, 2008. Schloss Dagstuhl. `http://drops.dagstuhl.de/opus/volltexte/2008/1766/`.

[CLC08] Hubert Comon-Lundh and Véronique Cortier. Computational soundness of observational equivalence. In *Proc. ACM CCS*, pages 109–118, 2008.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[EG83] Shimon Even and Oded Goldreich. On the security of multi-party ping-pong protocols. In *Proc. 24th IEEE FOCS*, pages 34–39, 1983.

[Fou09] Cédric Fournet. On the computational soundness of cryptographic verification by typing. Workshop on Formal and Computational Cryptography (FCC 2009), 2009.

[GLP05a] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real c code. In *Proc. 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 363–379. Springer-Verlag, 2005.

[GLP05b] Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic protocol analysis on real C code. In *6th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI 2005)*, pages 363–379. Springer, 2005.

[Gun92] A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.

[HLM03] Jonathan Herzog, Moses Liskov, and Silvio Micali. Plaintext awareness via key registration. In *Advances in Cryptology: CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 548–564. Springer, 2003.

[JLM05] Romain Janvier, Yassine Lakhnech, and Laurent Mazaré. Completing the picture: Soundness of formal encryption in the presence of active adversaries. In *Proc. ESOP*, pages 172–185, 2005.

[KMM94] Richard Kemmerer, Catherine Meadows, and Jon Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, 1994.

[Lau01] Peeter Laud. Semantics and program analysis of computationally secure information flow. In *Proc. 10th European Symposium on Programming (ESOP)*, pages 77–91, 2001.

[Lau04] Peeter Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *Proc. 25th IEEE Symposium on Security & Privacy*, pages 71–85, 2004.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[Mer83] Michael Merritt. *Cryptographic Protocols.* PhD thesis, Georgia Institute of Technology, 1983.

[Mor73] J. Morris. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.

[MW04] Daniele Micciancio and Bogdan Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Proc. 1st Theory of Cryptography Conference (TCC)*, volume 2951 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2004.

[Pau98] Lawrence Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Cryptology*, 6(1):85–128, 1998.

[SBB$^+$06] Christoph Sprenger, Michael Backes, David Basin, Birgit Pfitzmann, and Michael Waidner. Cryptographically sound theorem proving. In *Proc. 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 153–166, 2006.

[Sch96] Steve Schneider. Security properties and CSP. In *Proc. 17th IEEE Symposium on Security & Privacy*, pages 174–187, 1996.

[SP03] Eijiro Sumii and Benjamin C. Pierce. Logical relations for encryption. *Journal of Computer Security*, 11(4):521–554, 2003.

[SP07] E. Sumii and B. Pierce. A bisimulation for dynamic sealing. *Theoretical Computer Science*, 375(1-3):169–192, 2007.

Index