Computationally Sound Verification of Source Code

Michael Backes
Saarland University and
MPI-SWS
backes@cs.uni-sb.de

Matteo Maffei
Saarland University
Saarbrücken, Germany
maffei@cs.uni-sb.de

Dominique Unruh
Saarland University
Saarbrücken, Germany
unruh@mmci.uni-saarland.de

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 typechecker 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.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol Verification*

General Terms

Security, theory, verification

Keywords

Computational soundness, verification, source code

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

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 [25, 26, 37] (see, e.g., [33, 40, 3, 36, 39, 14]). 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, [5] 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 [2]). 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., [28, 17, 16]. 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 [16] 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 [16]. 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 [7], 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.

Due to space constraints, we omit many details and proofs. These are given in the full version [9].

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 [7]. 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

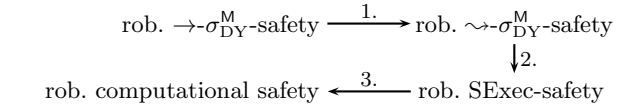

Figure 1: Main steps of the computational soundness proof

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 nondeterministic entity. Thus, for any possible behavior of the 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 $\sigma_{\mathrm{DY}}^{\mathsf{M}}$ (e.g., the code of one function is a subterm of the code of another). Since the library $\sigma_{\mathrm{DY}}^{\mathsf{M}}$ 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 $\sigma_{\mathrm{DY}}^{\mathsf{M}}$.

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 $\rightarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-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 $\sigma_{\mathrm{DY}}^{\mathsf{M}}$ is very similar in spirit to the one used in [18], we believe that the verification techniques used there can be applied to robust $\rightarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-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 $\sigma_{Highlevel}$ for $\sigma_{\mathrm{DY}}^{\mathsf{M}}$. A program that only invokes functions from $\sigma_{Highlevel}$ is guaranteed to satisfy these restrictions.

Sealing-based library. In the library $\sigma_{\mathrm{DY}}^{\mathsf{M}}$, we have internally represented symbolic cryptography as terms in some datatype. An alternative approach is used in the F7 verification framework [16] 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 $\rightsquigarrow$-$\sigma_{\mathrm{DY}}^{\mathsf{M}}$-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. 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 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 [24]; 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 al-

[1]The F# code of the library with typing annotations is available at [8].

[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.

lowed 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 [21]. 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. [19] pursue this approach; they do not, however, prove their encoding sound.

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 [7]. 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 [17]) 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 [5] for passive adversaries and symmetric encryption. The protocol language and security properties handled there were extended in [4, 34, 31, 15, 1], but still apply only to passive adversaries. Subsequent works studied computational soundness against active attacks (e.g., cf. [13, 11, 12, 10, 41, 35, 38, 32, 7]). Recent works also focused on computational soundness in the sense of observational equivalence of cryptographic realizations of processes (e.g., [6, 24, 23]). All these works do not tackle the computational soundness of protocol implementations. Concurrently with the announcement of this work at FCC 2009, [27] 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 [29] for secrecy properties of cryptographic protocols implemented in C. Chaki and Datta recently proposed a technique [22] based on software model checking for the automated verification of secrecy and authentication properties of protocols implemented in C. The analysis was applied to the SSL handshake protocol. Bhargavan et al. proposed a technique [17, 19] for the verification of F# protocol implementations by automatically extracting ProVerif models [20]. The analysis was used to verify implementations of real-world cryptographic protocols such as TLS [19]. None of these analysis techniques enjoys computational soundness guarantees. [19] also proposes an embedding of F# into the calculus of CryptoVerif. The embedding is not, however, proven to be sound; also, according to [19], 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 $t_1, \ldots, t_n$ 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 [16], a simple core calculus extending the Fixed Point Calculus [30] with refinement types and concurrency. Although very simple, this calculus is expressive enough to encode a large part of F# [16].

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 $\mathsf{match}\ M = N\ \mathsf{with}\ \mathsf{inl}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$ for some fresh x.

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* $\rightarrow$. 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 construc-

[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.

[4]The equivalence relation $\equiv$ considered in this paper is the extension of the heating relation $A \Rrightarrow B$ proposed in [16] where all heating rules are made symmetric. In the full version [9], we prove that making the heating relation symmetric is sound, i.e., it does not affect the safety of expressions.

a, b, c	name	$A, B ::=$	expression
h	constructor	M	value
		$M\ N$	function application
$M, N ::=$	value	$M = N$	syntactic equality
x, y, z	variable	$\mathsf{let}\ x = A\ \mathsf{in}\ B$	let
$()$	unit	$\mathsf{let}\ (x, y) = M\ \mathsf{in}\ A$	pair split
$\lambda x.A$	function	$\mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$	constructor match
(M, N)	pair	$\nu a.A$	restriction
$h\ M$	constructor application	$A \Rsh B$	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

tor 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 $\mathsf{assume}\ F$ and $\mathsf{assert}\ F$ 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 [16], a variant of first-order logic.

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 $\boldsymbol{S}$ is* statically safe *iff $P \vdash C$ where P are the active assumptions and C the active assertions of $\boldsymbol{S}$.*

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.* ◇

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.* ◇

The notion of robust $\rightarrow$-safety is the same as the robust safety defined in [16]. 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 pattern matches against private constructors occur only as subterms of A that are arguments of private constructors. We call an RCF-expression *pure* if it does not contain assumptions, assertions, outputs ($M!N$), inputs ($M?$), or forks ($M \Rsh N$).

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 [7], 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 $\mathsf{t}_1, \ldots, \mathsf{t}_n$, we write $\mathsf{d}(\mathsf{t}_1, \ldots, \mathsf{t}_n) = \bot$.* ◇

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

[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 [16].

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.

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 4 (Symbolic model). *A* symbolic model $\mathsf{M} = (\mathsf{C}, \mathsf{N}, \mathsf{T}, \mathsf{D}, \vdash_{\text{CoSP}})$ *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* $\vdash_{\text{CoSP}}$ *over* T*.* ◇

For a symbolic model, we specify a *computational implementation* Impl which assigns an algorithm to each constructor and destructor, and specifies the distribution of nonces.

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 5 (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 adversaries* E*, the node trace is in* $\wp$ *with overwhelming probability.* ◇

Definition 6 (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* $\Pi \in P$ *.* ◇

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 7 (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* $\mathsf{t}_1, \ldots, \mathsf{t}_n \in \mathsf{T}$ *satisfying* $\forall i \in [1, n]. S \vdash_{\text{CoSP}} \mathsf{t}_i$ *and* $\bot \neq \mathit{eval}_{\mathsf{f}}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \in \mathsf{T}$*, we have* $S \vdash_{\text{CoSP}} \mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$*.* ◇

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 $\mathit{lib}_{\mathsf{f}}$ such that $\sigma^{\mathsf{M}}_{\text{DY}}(\mathit{lib}_{\mathsf{f}})$ $(\mathsf{message}\ M_1, \ldots, \mathsf{message}\ M_n)$ returns $\mathsf{some\ message}\ M$ if $\mathit{eval}_{\mathsf{f}}(\mathsf{m}_1, \ldots, \mathsf{m}_n)$ returns m, or none if $\mathit{eval}_{\mathsf{f}}(\mathsf{m}_1, \ldots, \mathsf{m}_n)$ 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 $\sigma^{\mathsf{M}}_{\text{DY}}(\mathit{lib}_{\mathsf{dec}}) := \lambda x.\mathsf{match}\ x\ \mathsf{with}\ (\mathsf{message}(\mathsf{dk}(y)), \mathsf{message}(\mathsf{enc}(\mathsf{ek}(y), z, w)))\ \mathsf{then\ some}(\mathsf{message}(z))\ \mathsf{else\ none}$. A nonce would be represented as $\mathsf{message}(\mathsf{nonce}(\lambda x.a!x))$ for some fresh name a.[6]

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 $\{C_\iota[a] : a \neq a_{\mathit{chan}}$ is a name$\} = \iota(\mathsf{N})$.

Definition 8 (DY Library). *A DY library for* $\mathsf{M} = (\mathsf{C}, \mathsf{D}, \mathsf{N}, \mathsf{T}, \vdash_{\text{CoSP}})$ *is a substitution* $\sigma^{\mathsf{M}}_{\text{DY}}$ *from variables to RCF functions satisfying the following conditions:*

- *Let* $\mathsf{message}$ *be a private constructor.*
- $\operatorname{dom} \sigma^{\mathsf{M}}_{\text{DY}} = \{\mathit{lib}_{\mathsf{f}} \mid \mathsf{f} \in \mathsf{C} \cup \mathsf{D}\} \cup \{\mathit{nonce}, \mathit{send}, \mathit{recv}\}$.
- $\sigma^{\mathsf{M}}_{\text{DY}}(\mathit{lib}_{\mathsf{f}})$ *is a pure function such that the following holds for all* $\mathsf{m}_1, \ldots, \mathsf{m}_n \in \mathsf{T}$*: If* $\mathsf{m} := \mathit{eval}_{\mathsf{f}}(\mathsf{m}_1, \ldots, \mathsf{m}_n) \neq \bot$*, then* $\sigma^{\mathsf{M}}_{\text{DY}}(\mathit{lib}_{\mathsf{f}})$ $(\mathsf{message}\ \iota(\mathsf{m}_1), \ldots, \mathsf{message}\ \iota(\mathsf{m}_n)) \to^*$ $\mathsf{some\ message}\ \iota(\mathsf{m})$*. If* $\mathsf{m} = \bot$*, then* $\sigma^{\mathsf{M}}_{\text{DY}}(\mathit{lib}_{\mathsf{f}})$ $(\mathsf{message}\ \iota(\mathsf{m}_1), \ldots, \mathsf{message}\ \iota(\mathsf{m}_n)) \to^*$ none*. In all other cases,* $\sigma^{\mathsf{M}}_{\text{DY}}(\mathit{lib}_{\mathsf{f}})$ $(\ldots)$ *is stuck.*
- $\sigma^{\mathsf{M}}_{\text{DY}}(\mathit{nonce}) = \mathsf{fun}\ \_ \to \nu a.\mathsf{message}\ C_\iota[a]$.
- $\sigma^{\mathsf{M}}_{\text{DY}}(\mathit{send}) = (\mathsf{fun}\ x \to (\mathsf{match}\ x\ \mathsf{with\ message}\ \_\ \mathsf{then}\ a_{\mathit{chan}}!\mathsf{message}\ x\ \mathsf{else}\ \mathit{stuck}))$*. Here stuck is a pure diverging RCF expression.*

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

- $\sigma^{\mathsf{M}}_{\mathrm{DY}}(recv) = \mathsf{fun}\ \_ \to a_{chan}?$
- $fv(\mathrm{range}(\sigma^{\mathsf{M}}_{\mathrm{DY}})) = \emptyset$ *and* $fn(\mathrm{range}(\sigma^{\mathsf{M}}_{\mathrm{DY}})) = a_{\mathsf{chan}}$.
- *For any variable* $x \in \mathrm{dom}\,\sigma^{\mathsf{M}}_{\mathrm{DY}}$*, 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*.)* ◇

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.

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 $fn(A) = \emptyset$ and $fv(A) \subseteq \mathrm{dom}\,\sigma^{\mathsf{M}}_{\mathrm{DY}}$.

In $\sigma^{\mathsf{M}}_{\mathrm{DY}}$, all messages M are protected by the private constructor message. However, if an opponent would be allowed to perform a pattern match on message, he could get the internal representation of M and thus, e.g., extract the plaintext from an encryption. Similarly, an adversary applying 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 $\to$-safety models this.

Definition 9 (Robust $\to$-σ-safety). *Let σ be a substitution. We call an RCF expression a* σ-opponent *iff $fv(O) \subseteq \mathrm{dom}\,\sigma$ and O is pc-free and contains neither assertions nor assumptions and $a_{chan} \notin fn(O)$.*

An RCF expression A with $fv(A) \subseteq \mathrm{dom}\,\sigma$ is robustly $\to$-σ-safe *iff the application $(O\ A)\sigma$ is $\to$-safe for all σ-opponents O.* ◇

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 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 10. *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 $fv(A) \subseteq \{lib_{\mathsf{f}} : \mathsf{f} \in \mathsf{C} \cup \mathsf{D}\} \cup \{nonce, send, recv\}$ and $a_{chan} \notin fn(A)$ and A is pc-free and the FOL/F-formulae in A do not contain forbidden function symbols.* ◇

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 $A \rightsquigarrow B$, which differs from the standard reduction relation $A \to B$ in that cryptographic operations are atomically performed. The relation $\rightsquigarrow$ is defined like the normal reduction relation $\to$, but additionally satisfies the following rules:

$$send\ (\mathsf{message}\ M) \rightsquigarrow a_{chan}!\mathsf{message}\ M$$

$$recv\ N \rightsquigarrow a_{chan}?$$

$$\sigma^{\mathsf{M}}_{\mathrm{DY}}(f)\ M \to^* N \Longrightarrow f\ M \rightsquigarrow N \qquad (f \in \mathrm{dom}\,\sigma^{\mathsf{M}}_{\mathrm{DY}})$$

Note that these rules have been designed to be in one-to-one correspondence with the semantics of $\sigma^{\mathsf{M}}_{\mathrm{DY}}$ as defined in Definition 8.

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

Definition 11 ($\rightsquigarrow$-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.* ◇

In contrast to Definition 9, 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 12 (Robust $\rightsquigarrow$-σ-safety). *An RCF expression A with $fv(A) \subseteq \mathrm{dom}\,\sigma$ is* robustly $\rightsquigarrow$-σ-safe *iff the application $O\ A$ is $\rightsquigarrow$-σ-safe for all σ-opponents O.* ◇

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

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.*

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 $lib_{\mathsf{f}}(x)$ is executed, where lib_{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 $nonce()$. If the adversary requests a function evaluation $send(x)$ the adversary is given the bitstring $\eta(x)$; in the case $recv()$, 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[7] normal form (a structure) with

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

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 13 (Computational 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.* ⟨*Adv is a non-deterministic machine that only sends* m *if* $S \vdash_{\mathrm{CoSP}} \mathsf{m}$ *where* S *are the messages sent to Adv so far.*⟩ *We define the* computational ⟨symbolic⟩ RCF-execution *as an interactive machine* $\mathrm{Exec}^{\mathrm{Impl}}_A(1^k)$ ⟨SExec_A⟩ *that takes a security parameter* k *as argument* ⟨*that does not take any argument*⟩ *and interacts with Adv:*

Start*: Let* η *be a totally undefined partial function mapping variables to bitstrings* ⟨*terms*⟩. *(*η *provides an environment giving bitstring* ⟨*term*⟩ *interpretation to the variables occurring in the current expression.)*

Main Loop*: Let* $\boldsymbol{S} = \nu a_1 \ldots a_l.\ (\prod_{i\in 1\ldots m} \mathsf{assume}\ C_i \ \pi\ \prod_{j\in 1\ldots n} c_j!M_j \ \pi\ (\prod_{k\in 1\ldots o} \mathcal{L}_k\{e_k\}))$ *be the normal form of* A. *Here* $\mathcal{L}_k$ *stands for nested lets as given by the following grammar:* $\mathcal{L} := \{\} \mid \mathsf{let}\ x = \mathcal{L}\ \mathsf{in}\ B$. *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* $e_k = x\ (y_1, \ldots, y_n)$ *with* $x = lib_{\mathsf{f}}$ *for some constructor or destructor* f *of arity* n *and* $y_1, \ldots, y_n \in \mathrm{dom}\,\eta$: *Let* $m := \mathrm{Impl}_{\mathsf{f}}(\eta(y_1), \ldots, \eta(y_n))$ ⟨$m := eval_{\mathsf{f}}(\eta(y_1), \ldots, \eta(y_n))$⟩. *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* $\mathsf{n} \in \mathsf{N}_P$[8] ⟨*let* r *be a fresh protocol nonce*⟩ *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* ⟨*term*⟩ 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* $x \in \mathrm{dom}\,\eta$: *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* $x \in \mathrm{dom}\,\eta$, *let* $\rho(x)$ *be the lexicographically first* $y \in \mathrm{dom}\,\eta$ *with* $\eta(x) = \eta(y)$.[9] *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.* ◇

For a given polynomial-time interactive machine Adv, a closed expression A, and a polynomial p, we let $Trace^{\mathrm{Impl}}_{Adv,A,p}(k)$ denote the list of pairs $(\mathbf{S}, E)$ output by $\mathrm{Exec}^{\mathrm{Impl}}_A(1^k)$ (at the beginning of each loop iteration) within the first $p(k)$ computation steps (jointly counted for $Adv(1^k)$ and $\mathrm{Exec}^{\mathrm{Impl}}_A(1^k)$).

Definition 14 (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* $\boldsymbol{S}$, $vars := fv(E) \cup \mathrm{dom}\,\sigma$, *exterms is the set of all FOL/F-subterms* $h(t)$ *of* P, C *with* h *forbidden and* t *closed, and*

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

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 and x' are 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 $\rightarrow$-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safe: $A_0\sigma^{\mathsf{M}}_{\mathrm{DY}}$ reduces to $\mathsf{assume}\ (\sigma^{\mathsf{M}}_{\mathrm{DY}}(nonce) = \lambda z.z)\ \pi\ \mathsf{assert}\ (\mathsf{false})$ and we have $nonce\sigma^{\mathsf{M}}_{\mathrm{DY}} = (\lambda z.z) \vdash \mathsf{false}$ (this is implied by equality-friendliness). In the computational execution, however, we get the process $A = \mathsf{assume}\ (nonce = \lambda z.z)\ \pi\ \mathsf{assert}\ (\mathsf{false})$, 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 15 (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

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

[9] 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.

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-*$\sigma^{\mathsf{M}}_{\mathrm{DY}}$*-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 $\sigma^{\mathsf{M}}_{\mathrm{DY}}$. An inspection of Definition 14, however, reveals that the definition only depends on the *domain* of $\sigma^{\mathsf{M}}_{\mathrm{DY}}$, 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 13 (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 16 (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-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safe for all i.* ◇

We are now ready to establish the link between robust $\rightsquigarrow$-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safety and robust SExec-safety.

Lemma 2. *If $\mathsf{M} \vdash A_0$ and A_0 is robustly $\rightsquigarrow$-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safe, then A_0 is robustly* SExec*-safe.*

5.3 Computational soundness of the DY-library

We will now use the CoSP framework [7] 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. All these operations are available in CoSP protocols, too. We can therefore construct a CoSP protocol Π_{A_0} that, when executed symbolically, emulates the machine SExec_{A_0}. Furthermore, the same CoSP protocol Π_{A_0}, when executed computationally, emulates the computational execution $\mathrm{Exec}^{\mathrm{Impl}}_{A_0}$. We call *failure nodes* those nodes in the CoSP protocol Π_{A_0} which correspond to sending $(\mathbf{S}, E)$ to the adversary such that $(\mathbf{S}, E)$ is not statically equation-$\sigma^{\mathsf{M}}_{\mathrm{DY}}$-safe.

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 6). Let $\sigma^{\mathsf{M}}_{\mathrm{DY}}$ be a DY library for M.*

Let A_0 be an efficiently decidable[10] *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 2, 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. 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 [7], such a result is presented for a symbolic model supporting encryption, signatures, and arbitrary strings as payloads.

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

- Constructors: 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.
- The message type T describes which terms are considered well-formed. We omit the details here.
- Destructors: 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

[10] 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.

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.

- The deduction relation $\vdash_{\text{CoSP}}$ and the set N of nonces are as in Definition 7.

CoSP [7] 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 will call these conditions the "enc-sig-implementation conditions".

Furthermore, [7] 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*. Assuming that all these conditions are fulfilled, we get computational soundness for encryptions and signatures:

THEOREM 2 (COMPUTATIONAL SOUNDNESS [7]). *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 3. *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 $A_0\sigma_{\text{DY}}^{\mathsf{M}_{es}}$ *is robustly* $\to$*-safe or* A_0 *is robustly* $\rightsquigarrow$*-*$\sigma_{\text{DY}}^{\mathsf{M}_{es}}$*-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 $\sigma_{\text{DY}}^{\mathsf{M}_{es}}$ and that ensures that a program A_0 that never directly calls $\sigma_{\text{DY}}^{\mathsf{M}_{es}}$ 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 $\sigma_{\text{DY}}^{\mathsf{M}_{es}}(lib_{\mathsf{enc}})$. This ensures that the randomness-argument of $\sigma_{\text{DY}}^{\mathsf{M}_{es}}(lib_{\mathsf{enc}})$ 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 $\sigma_{\text{DY}}^{\mathsf{M}_{es}}(lib_{\mathsf{dk}})$ will only be used as the second argument to $\sigma_{\text{DY}}^{\mathsf{M}_{es}}(lib_{\mathsf{dec}})$.[11] 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). The source code of $\sigma_{Highlevel}$ is presented in the full version.

[11] 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.

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

LEMMA 4. *Let* A_0 *be an RCF expression with* $\mathsf{M}_{es} \vdash A_0$ *and* $fv(A_0) \cap \operatorname{dom}\sigma_{\text{DY}}^{\mathsf{M}_{es}} = \varnothing$ *and not containing the RCF-constructors DecKey and SigKey.*

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

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

THEOREM 3 (COMPUTATIONAL SOUNDNESS FOR $\sigma_{\text{DY}}^{\mathsf{M}_{es}}$). *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 $A_0\sigma_{Highlevel}\sigma_{\text{DY}}^{\mathsf{M}_{es}}$ *is robustly* $\to$*-safe, then* $A_0\sigma_{Highlevel}$ *is robustly computationally safe using* Impl.

Proof. Let $A_0' := A_0\sigma_{Highlevel}$. Since $fv(A_0) \subseteq \sigma_{Highlevel}$ and $\operatorname{dom}\sigma_{Highlevel} \cap \operatorname{dom}\sigma_{\text{DY}}^{\mathsf{M}_{es}} = \varnothing$, $fv(A_0) \cap \operatorname{dom}\sigma_{\text{DY}}^{\mathsf{M}_{es}} = \varnothing$. Thus by Lemma 4, $\Pi_{A_0'}$ 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 3, if $A_0'\sigma_{\text{DY}}^{\mathsf{M}_{es}}$ is robustly $\to$-safe, then A_0' is robustly computationally safe using Impl. □

5.5 Sealing-based library.

In the library $\sigma_{\text{DY}}^{\mathsf{M}_{es}}$, we have internally represented symbolic cryptography as terms in some datatype. An alternative approach is used in the F7 verification framework [16]. In this approach, a library σ_{S} based on seals is used (cf. Section 1.1). The main difficulty with such a library is that it uses a global state to keep track of encryptions and signatures produced by the protocol. We can, however, show that robust safety with respect to σ_{S} implies robust safety with respect to $\sigma_{\text{DY}}^{\mathsf{M}_{es}}$. From this, we immediately get a computational soundness result for σ_{S}; the theorem is exactly the same as Theorem 3, except that $\sigma_{\text{DY}}^{\mathsf{M}_{es}}$ is replaced by σ_{S}. We refer to the full version for details.

6. 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), the Miur'07 Project

SOFT (*Security Oriented Formal Techniques*), the ERC starting grant "End-to-end security", and the DFG grant 3194/1-1.

7. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

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

[7] 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.

[8] 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`.

[9] Michael Backes, Matteo Maffei, and Dominique Unruh. Computationally sound verification of source code (full version). IACR ePrint archive 2010/416, 2010.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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 as IACR ePrint 2005/401.

[22] 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.

[23] 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/`.

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

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

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

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

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

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

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

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

[36] 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.

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

[38] 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.

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

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

[41] 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.