Computational Soundness of Passively Secure Encryption in Presence of Active Adversaries

Master Thesis

Sebastian Meiser

Supervisor: Dr. Dominique Unruh
Additional reviewer: Prof. Michael Backes

December 2010

Abstract

The goal of this thesis is to modify the CoSP framework [1] in a way that allows the usage of encryption schemes that are not necessarily secure under IND-CCA, but under IND-CPA. To ensure the soundness, however, we show how to slightly modify some constructions in order to distinguish between messages sent by the adversary and messages constructed by the protocol. We add a new protocol condition which ensures that the protocol members do not decrypt messages sent by the adversary. Basically, we design a computational soundness proof for IND-CPA secure encryption schemes based on CoSP. It turns out that to achieve this, we only need one additional assumption that can be checked easily for a given protocol. Following the example of [1] we embed the applied π-calculus into the modified CoSP, extend the given implementation of public-key cryptography and digital signatures and prove it sound in our setting. Finally we present how to check the additional assumption automatically using automated tools like ProVerif.

I hereby declare that I have written this thesis myself
and that I did not use sources and aids other than those listed.

I hereby give the Saarland University the authorization to
publish the thesis in networks and/or data bases.

Contents

1 Introduction

1.1 Motivation

1.1.1 The computational world and the symbolic world

When proving them secure, cryptographic protocols are usually expressed using probabilistic Turing machines. The messages floating around over different channels are considered to be bitstrings. To prove the security of a protocol, one often defines a game between the protocol and an adversary. When there is an adversary that is able to win the game (i.e. reaches its goal) with some noticeable probability, the protocol is considered to be insecure, otherwise it is considered to be secure. The adversaries typically are only restricted in terms of running time and channel access. There may be secure channels that the adversary cannot see, mildly insecure ones where the adversary can read messages but not change them and completely insecure chanels where the adversary even can exchange messages.

The resulting proofs are long, complicated, hard to write, and even harder to verify.

For many years now, there have been ongoing attempts to reduce the complexity of these proofs by abstracting away probabilities, bitstring operations, and computational power. So called Dolev-Yao models [5] follow this approach. Here, the adversaries follow simple rules that allow them to deduce new information out of the messages they have seen so far. Instead of the bitstring representation that allows for lots of unintended manipulation, the messages are represented as symbolic terms.

While proofs in a Dolev-Yao model are easier to make and can even be automatised quite well, in the beginning it was not clear, whether the abstractions are sound. If a symbolic model is not sound, proving properties by using it might not be enough to show that the corresponding computational protocol also fulfills the properties. Thus, proving the computational soundness of a symbolic model is an important task.

Naturally, doing such proofs is not a new thing and thus there are lots of proofs for computational soundness. One of the problems is, it seems to be that for every proof, there is also a new formalism, suiting just this situation perfectly, but failing or being more complicated in other situations.

1.1.2 CoSP

The CoSP paper from M.Backes, D.Hofheinz and D. Unruh [1] provides a framework for constructing computational soundness proofs for arbitrary symbolic models. A simple but powerful way for describing the protocols is included. To prove a symbolic model computationally sound, it is embedded into the CoSP model: A mapping between the original symbolic protocols and symbolic CoSP protocols is defined. Then the power of the CoSP model in combination with its already established results can be used to prove the symbolic model to be computationally sound. While it is necessary to prove the connection between the (symbolic) CoSP protocol and the original (symbolic) protocol to be secure, the considerably more ugly step of establishing the connection from the symbolic to the computational world is provided by CoSP.

In order to obtain computational soundness results using the CoSP framework, several requirements for symbolic protocols and for their computational implementation must be met. The "flaw" that we want to adress with this thesis is the following one: CoSP requires the (computational) implementation to use an encryption scheme which is secure under IND-CCA2.[1] While this may seem reasonable for some protocols, depending on their structure and possible attacks, it is less reasonable when highly secure encryptions are not even relevant for a given protocol. Our idea is to change the proof in such a way that the (weaker) IND-CPA property suffices.

1.2 Our contribution

Our contribution is to improve the CoSP proof for public key encryption schemes and signatures in such a way that it suffices for the computational implementation to use an IND-CPA secure encryption scheme. We show that by adding an additional assumption called **garbage free decryption** this goal can be reached. Every encryption scheme that can be used for the implementation of protocols with the original result still remains an option. Additionally, all the schemes that are only secure against IND-CPA but not against IND-CCA2 can be used. Among others, this includes every stream cipher that uses the quite popular CBC mode. Even on the level of research it brings some benefits. When investigating protocols containing zero knowledge proofs, using simple encryption schemes like ElGamal [6] make proving certain properties easier, compared to complex IND-CCA2 secure schemes that may contain hashes.

We also embed the applied π-calculus into our new version of CoSP and prove it to have computational soundness. This has been done for the original CoSP framework with IND-CCA2 security as a necessary implementation condition. We show, that out new condition can easily be assured in the world of the applied π-calculus and transferred to the corresponding (symbolic) CoSP protocol. Finally, we show how to automatically prove that a protocol has **garbage free decryption** by using ProVerif.

2 CoSP framework

The CoSP paper [1] provides a general framework for computational soundness proofs. Since our work is an extension of their result, we will describe the notation and the main structure of the CoSP proof framework in this section. For more details we recommend having a look at the original paper.

The notation and the basic definitions in this section are taken from [1]. Additional comments from me are marked.

[1] For an overview over some commonly used security notions, have a look at Section 6.1

2.1 Notation

- Substitutions:
  For a term t and a substitution φ we write $t\varphi$ to denote that φ is applied to t.

- Functions:
  For a function f we write $f(x := y)$ when we want to use the function f, but overwrite the resulting value for x by y. Formally, $f(x := y)$ equals a function g defined by:

  - $g(x) = y$
  - $g(z) = f(z)$ for $z \neq x$

  We call a non-negative function f *negligible*, iff for every c and sufficiently large n, $f(n) < n^{-c}$.
  We call a function f *overwhelming* iff $1 - f$ is negligible.

  We call an n-ary function f *length regular* iff $|m_i| = |m'_i|$ for $i = 1, \ldots, n$ implies $|f(\underline{m})| = |f(\underline{m'})|$.

- Families of variables:
  When n is clear from the context, we write $\underline{x}$ instead of $x_1, \ldots, x_n$.

- Sets:
  For a function f and a set M we denote the preimage of f as $f^{-1}(M) := \{x : f(x) \in M\}$.
  When there is a deterministic polynomial-time algorithm that decides membership for a set M, we call this set *efficiently decidable*.
  We call M *prefix-closed* iff $x \in M$ implies $x' \in M$ for all prefixes x' of x.

Definition 1 (Constructors, destructors, nonces, and messages types)**.** A *constructor* C is a symbol with a (possibly zero) arity. A *nonce* N is a symbol with zero arity. We write $C/n \in \mathbf{C}$ to denote that $\mathbf{C}$ contains a constructor C with arity n. A *message type* $\mathbf{T}$ *over* $\mathbf{C}$ *and* $\mathbf{N}$ is a set of terms over constructors $\mathbf{C}$ and nonces $\mathbf{N}$. A *destructor* D of arity n, written D/n, over a message type $\mathbf{T}$ is a partial map $\mathbf{T}^n \to \mathbf{T}$. If D is undefined on $\underline{t}$, we write $D(\underline{t}) = \bot$.

For evaluation, we define a partial function $eval_F : \mathbf{T}^n \to \mathbf{T}$ for every constructor or nonce F/n as: $eval_F(t_1, \ldots, t_n) := F(\underline{t})$ if $F(\underline{t}) \in \mathbf{T}$ and $eval_F(\underline{t}) := \bot$ otherwise. For every destructor F/n we also define a partial function $eval_F : \mathbf{T}^n \to \mathbf{T}$ as: $eval_F(\underline{t}) := F(\underline{t})$ if $F(\underline{t}) \neq \bot$ and $eval_F(\underline{t}) := \bot$ otherwise. We name the functions in the same way to unify the notation.

We also need to define, which terms can be deduced from a set of other terms. Thus, we define a deduction relation $\vdash$ over such a set of terms $\mathbf{T}$ here. Intuitively, writing $S \vdash m$ for some set $S \subseteq \mathbf{T}$ and a term $m \in \mathbf{T}$ means that m can be deduced from S by anyone, including the adversary.

Definition 2 (Deduction relation)**.** A *deduction relation* $\vdash$ over a message type $\mathbf{T}$ is a relation between $2^{\mathbf{T}}$ and $\mathbf{T}$.

The deduction relation $\vdash$ models the knowledge and the possibilities of the adversary. Usually the adversary can apply all constructors and destructors. We define $S \vdash \underline{t} \Rightarrow S \vdash C(\underline{t})$ for every constructor C and $S \vdash \underline{t} \wedge D(\underline{t}) \neq \bot \Rightarrow S \vdash D(\underline{t})$ for every destructor D, respectively.

Definition 3 (Symbolic model)**.** A *symbolic model* $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$ consists of a set of constructors $\mathbf{C}$, a set of nonces $\mathbf{N}$, a message type $\mathbf{T}$ over $\mathbf{C}$ and $\mathbf{N}$ with $\mathbf{N} \subseteq \mathbf{T}$, a set of destructors $\mathbf{D}$ over $\mathbf{T}$, and a deduction relation $\vdash$ over $\mathbf{T}$.

2.2 Symbolic protocols

Definition 4 (CoSP protocol)**.** A *CoSP protocol* Π_{s} is a tree with a distinguished root and labels on both edges and nodes. Each node has a unique identifier N and one of the following types:

- *Computation nodes* are annotated with a constructor, nonce, or destructor 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.)
- *Nondeterministic nodes* have no further annotation. Nondetermininistic nodes have at least one and at most finitely many successors; the corresponding edges are labeled with distinct bitstrings.

We can define a probabilistic CoSP protocol by assigning the nondeterministic nodes a probability distribution over its successors. Probabilistic protocols are an important step between the symbolic world and the computational world. While they still work on symbolic terms, their behaviour can be analysed more closely and it is possible to talk about the probability for certain events to take place.

Definition 5 (Probabilistic CoSP protocol)**.** A *probabilistic CoSP protocol* Π_{p} is a CoSP protocol, where each nondeterministic node is additionally annotated with a probability distribution over the labels of the outgoing edges.

Transforming a probabilistic CoSP protocol Π_{p} back into to a CoSP protocol Π_{s} can easily be done by just removing the probability distributions again. We call Π_{s} the symbolic protocol that *corresponds to* Π_{p}.

Definition 6 (Efficient protocol)**.** We call a probabilistic CoSP protocol *efficient* if:

- There is a polynomial p such that for any node N, the length of the identifier of N is bounded by $p(m)$ where m is the length (including the total length of the edge-labels) of the path from the root to N.
- There is a deterministic polynomial-time algorithm that, given the identifiers of all nodes and the edge labels on the path to a node N, computes the label of N.

In the following definition of the symbolic execution, we specify a *full trace* as a list of triples (S, ν, f). Here S contains all messages, the adversary has seen during the execution and thus represents his "knowledge", ν represents the current node identifier in the protocol and f represents a mapping from previous node identifiers to messages. For a node N, intuitively $f(N)$ stands for the term that was computed or received at node N.

Definition 7 (Symbolic execution)**.** Let a symbolic model $(\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$ and a CoSP protocol Π_{s} 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_{s}, 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 $m := \mathit{eval}_F(\underline{\tilde{t}}) \neq \bot$, ν' is the *yes*-successor of ν in Π_{s}, 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_{s} and there exists an m with $S \vdash m$ 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_{s} and $f' = f$.
  - If ν is a control or a nondeterministic node, then ν' is a successor of ν and $f' = f$ and $S' = S$.

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

2.3 Computational model

After defining symbolic protocols and their (symbolic) behaviour, we now define their computational counterpart. It consists of a computational implementation that is modeled as a family of functions, one for every constructor, destructor, and nonce.

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

All functions A_F have to be computable in deterministic polynomial-time, and all A_N have to be computable in probabilistic polynomial-time.[2]

The requirement that A_C and A_D are deterministic is without loss of generality. When they have to behave randomly we can "outsource" the randomness by adding an explicit randomness argument, taking a nonce as input.

The representation of bitstrings will be assumed to be a canonical representation of symbols. The exact requirements that we will need later on can be seen in the implementation conditions in Appendix 11.1. In this section we will not require the bitstring representation itself to ensure any secrecy property like, e.g., hiding the message inside of a ciphertext.

Definition 9 (Computational execution)**.** Let a symbolic model $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$, a computational implementation A of $\mathbf{M}$, and a probabilistic CoSP protocol Π_{p} 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 $Nodes^p_{\mathbf{M},A,\Pi_{\mathsf{p}},E}(k)$, 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_{p}. Let f be an initially empty partial function from node identifiers to bitstrings, and let n be an initially empty partial function from $\mathbf{N}$ 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 $N \in \mathbf{N}$: Let $m' := n(N)$ if $n(N) \neq \bot$ and sample m' according to $A_N(k)$ 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 $m' := A_F(k, \underline{\tilde{m}})$. 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 $f := f'$.

[2] More precisely, there has to exist a single uniform probabilistic polynomial-time algorithm A that, given the name of $C \in \mathbf{C}$, $D \in \mathbf{D}$, or $N \in \mathbf{N}$, together with an integer k and the inputs $\underline{m}$, computes the output of A_C, A_D, and A_N or determines that the output is undefined. This algorithm must run in polynomial-time in $k + |\underline{m}|$ and may not use random coins when computing A_C and A_D.

* 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 $\nu := \nu'$.
* If ν is a nondeterministic node, let $\mathcal{D}$ be the probability distribution in the annotation of ν. Pick ν' according to the distribution $\mathcal{D}$, and 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.

When facing a nondeterministic node, the computational execution uses the probability annotation to decide which successor to choose. Thus, the overall execution leads to a probability distribution over the reachable node paths. We use them to define trace properties. Note that in the computational setting we are not interested in a full trace (including adversary knowledge). This is due to the fact that our (computational) adversary is treated like a black box.

2.4 Computational Soundness

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

Let $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$ be a symbolic model and Π_s a CoSP protocol. Then Π_s *symbolically satisfies* a trace property $\mathcal{P}$ in $\mathbf{M}$ iff every node trace of Π_s is contained in $\mathcal{P}$. Let A be a computational implementation of $\mathbf{M}$ and let Π_p be a probabilistic CoSP protocol. Then (Π_p, A) *computationally satisfies* a trace property $\mathcal{P}$ in $\mathbf{M}$ iff for all probabilistic polynomial-time interactive machines E and all polynomials p, the probability is overwhelming that $Nodes^p_{\mathbf{M},A,\Pi_\mathsf{p},E}(k) \in \mathcal{P}$.

Definition 11 (Computational soundness)**.** A computational implementation A of a symbolic model $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$ is *computationally sound* for a class P of CoSP protocols iff for every trace property $\mathcal{P}$ and for every efficient probabilistic CoSP protocol Π_p, we have that (Π_p, A) computationally satisfies $\mathcal{P}$ whenever the corresponding CoSP protocol Π_s of Π_p symbolically satisfies $\mathcal{P}$ and $\Pi_\mathsf{s} \in P$.

2.5 The hybrid execution

To simplify the proofs, we define yet another type of execution that stands between the symbolic and the computational execution. With the help of a so called simulator, we define a way for a symbolic protocol to communicate with a (simulated) computational

adversary. The simulator is equiped with a bunch of properties that lead to an interesting fact: The pure existence of a simulator, which has all the defined properties, directly implies computational soundness.

In the following, we fix a symbolic model $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$ and a computational implementation A of $\mathbf{M}$. Additionally we assume that for every term or node that is sent by a machine, this term/node can be suitably encoded as a bitstring.

Definition 12 (Simulator)**.** A *simulator* is an interactive machine *Sim* that satisfies the following syntactic requirements:

- When activated without input, it replies with a term $m \in \mathbf{T}$. (This corresponds to the situation that the protocol expects some message from the adversary.)
- When activated with some $t \in \mathbf{T}$, it replies with an empty output. (This corresponds to the situation that the protocol sends a message to the adversary.)
- When activated with $(\texttt{info}, \nu, t)$ where ν is a node identifier and $t \in \mathbf{T}$, it replies with $(\texttt{proceed})$.
- At any point (in particular instead of sending a reply), it may terminate.

Although a simulator is meant to communicate with a symbolic protocol, it internally simulates a computational adversary. A simulator functions as a translater between these worlds, especially translating terms from the protocol to bitstrings for its internal adversary and vice versa.

The hybrid execution with a simulator is hybrid in the sense of being a mix of the symbolic and the computational execution. It is defined as follows:

Definition 13 (Hybrid execution)**.** Let Π_{p} be a probabilistic CoSP protocol, and let *Sim* be a simulator. We define a probability distribution $H\text{-}Trace_{\mathbf{M},\Pi_{\mathsf{p}},Sim}(k)$ on (finite) lists of tuples (S_i, ν_i, f_i) called the *full hybrid trace* according to the following probabilistic algorithm Π^C, run on input k, that interacts with *Sim*. (Π^C is called the hybrid protocol machine associated with Π_{p} and internally runs a symbolic simulation of Π_{p} as follows:)

- *Start*: $S_1 := S := \varnothing$, $\nu_1 := \nu$ is the root of Π_{p}, and $f_1 := f$ is a totally undefined partial function mapping node identifiers to $\mathbf{T}$. Run Π_{p} on ν.
- *Transition*: For $i = 2, 3, \ldots$ do the following:
  - Let $\underline{\tilde{\nu}}$ be the node identifiers in the label of ν. Define $\underline{\tilde{t}}$ through $\tilde{t}_j := f(\tilde{\nu}_j)$.
  - Proceed depending on the type of ν:
    - If ν is a computation node with constructor, destructor, or nonce F, then let $m := eval_F(\underline{\tilde{t}})$. If $m \neq \bot$, let ν' be the *yes*-successor of ν and let $f' := f(\nu := m)$. If $m = \bot$, let ν' be the *no*-successor of ν and let $f' := f$.
    - If ν is an output node, send $\tilde{t}_1$ to *Sim* (but without handing over control to *Sim*). Let ν' be the unique successor of ν. Set $\nu := \nu'$.

* If ν is an input node, hand control to *Sim*, and wait to receive $m \in \mathbf{T}$ from *Sim*. Let $f' := f(\nu := m)$, and let ν' be the unique successor of ν. Set $f := f'$ and $\nu := \nu'$.
* If ν is a control node labeled with out-metadata l, send l to *Sim*, hand control to *Sim*, and wait to receive a bitstring l' from *Sim*. 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 $\nu := \nu'$.
* If ν is a nondeterministic node, sample ν' according to the probability distribution specified in ν. Let $\nu := \nu'$.

– Send $(info, \nu, t)$ to *Sim*. When receiving an answer (*proceed*) from *Sim*, continue.

– If *Sim* has terminated, stop. Otherwise let $(S_i, \nu_i, f_i) := (S, \nu, f)$.

The probability distribution of the (finite) list $\nu_1, \ldots$ produced by this algorithm we denote $\mathit{H\text{-}Nodes}_{\mathbf{M},\Pi_\mathsf{p},\mathit{Sim}}(k)$. We call this distribution the *hybrid node trace*.

With $\mathit{Sim} + \Pi^C$ we denote the execution of *Sim* and Π^C.

Now we define the previously mentioned properties that will make the simulator a key element in the proof.

The first such property is the one of Dolev-Yao style. It binds the simulator to the symbolic world, intuitively stating, that the simulated (computational) adversary is not more powerful than a symbolic one. More precisely, whenever *Sim* sends a term t to the probabilistic CoSP protocol, t has to be derivable from the messages that *Sim* has seen so far.

Definition 14 (Dolev-Yao style simulator)**.** A simulator *Sim* is *Dolev-Yao style* (short: *DY*) for $\mathbf{M}$ and Π_p, if with overwhelming probability the following holds:

In an execution of $\mathit{Sim} + \Pi^C$, for each ℓ, let $m_\ell \in \mathbf{T}$ be the ℓ-th term sent (during processing of one of Π^C's input nodes) from *Sim* to Π^C in that execution. Let $T_\ell \subseteq \mathbf{T}$ the set of all terms that *Sim* has received from Π^C (during processing of output nodes) prior to sending m_ℓ. Then we have $T_\ell \vdash m_\ell$.

The next property, called indistinguishability, intuitively says that the hybrid execution can not be distinguished from the computational execution. More precisely, the hybrid node traces are compuationally indistinguishable[3] from the computational node traces. With $\overset{\mathrm{c}}{\approx}$ we denote computational indistinguishability.

Definition 15 (Indistinguishable simulator)**.** A simulator *Sim* is *indistinguishable* for $\mathbf{M}$, Π_p, an implementation A, an adversary E, and a polynomial p, if

$$\mathit{Nodes}^p_{\mathbf{M},A,\Pi_\mathsf{p},E}(k) \overset{\mathrm{c}}{\approx} \mathit{H\text{-}Nodes}_{\mathbf{M},\Pi_\mathsf{p},\mathit{Sim}}(k),$$

i.e., if the computational node trace and the hybrid node trace are computationally indistinguishable.

[3] The corresponding random variables cannot be distinguished by any probabilistic algorithm that runs in polynomial time in the security parameter.

Definition 16 (Good simulator)**.** A simulator is *good* for $\mathbf{M}$, Π_{p}, A, E, and p if it is Dolev-Yao style for $\mathbf{M}$, and Π_{p}, and indistinguishable for $\mathbf{M}$, Π_{p}, A, E, and p.

The following theorem concludes the point of defining the notion of a good simulator: A good simulator implies computational soundness.

Theorem 1 (Good simulator implies soundness)**.** *Let* $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$ *be a symbolic model, let* P *be a class of CoSP protocols, and let* A *be a computational implementation of* $\mathbf{M}$*. Assume that for every efficient probabilistic CoSP protocol* Π_{p} *(whose corresponding CoSP protocol is in* P*), every probabilistic polynomial-time adversary* E*, and every polynomial* p*, there exists a good simulator for* $\mathbf{M}$*,* Π_{p}*,* A*,* E*, and* p*. Then* A *is computationally sound for protocols in* P*.*

3 Symbolic Model

This specification is taken from [1] and will be used for our proof as well. Additional comments are marked.

The symbolic model is defined as $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$:

- Constructors and nonces: Let $\mathbf{C} := \{enc/3, ek/1, dk/1, sig/3, vk/1, sk/1, pair/2, string_0/1, string_1/1, empty/0, garbageSig/2, garbage/1, garbageE/2\}$ and $\mathbf{N} := \mathbf{N}_P \cup \mathbf{N}_E$. Here $\mathbf{N}_P$ and $\mathbf{N}_E$ are countably infinite sets representing protocol and adversary nonces, respectively. Intuitively, encryption, decryption, verification, and signing keys are represented as $ek(r), dk(r), vk(r), sk(r)$ with a nonce r (the randomness used when generating the keys). $enc(ek(r'), m, r)$ encrypts m using the encryption key $ek(r')$ and randomness r. $sig(sk(r'), m, r)$ is a signature of m using the signing key $sk(r')$ and randomness r. The constructors $string_0$, $string_1$, and $empty$ are used to model arbitrary strings used as payload in a protocol (e.g., a bitstring 010 would be encoded as $string_0(string_1(string_0(empty)))$). $garbage$, $garbageE$, and $garbageSig$ are constructors necessary to express certain invalid terms the adversary may send, these constructors are not used by the protocol. However $garbageE$ will become quite important later in the proof.

- Message type: We define $\mathbf{T}$ as the set of all terms M matching the following grammar:

$$\begin{aligned} M ::= {} & enc(ek(N), M, N) \mid ek(N) \mid dk(N) \mid \\ & sig(sk(N), M, N) \mid vk(N) \mid sk(N) \mid \\ & pair(M, M) \mid S \mid N \mid \\ & garbage(N) \mid garbageE(M, N) \mid \\ & garbageSig(M, N) \\ S ::= {} & empty \mid string_0(S) \mid string_1(S) \end{aligned}$$

  where the nonterminal N stands for nonces.

- Destructors: $\mathbf{D} := \{dec/2, isenc/1, isek/1, ekof/1, verify/2, issig/1, isvk/1, vkof/2, fst/1, snd/1, unstring_0/1, unstring_1/1, equals/2\}$. 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. $dec(dk(r), c)$ decrypts the ciphertext c. $verify(vk(r), s)$ verifies the signature s with respect to the verification key $vk(r)$ and returns the signed message if successful. The destructors *fst* and *snd* are used to destruct pairs, and the destructors $unstring_0$ and $unstring_1$ allow to parse payload-strings. (Destructors *ispair* and *isstring* are not necessary, they can be emulated using *fst*, $unstring_i$, and $equals(\cdot, empty)$.)

  The behavior of the destructors is given by the following rules; an application matching none of these rules evaluates to $\bot$:

$$
\begin{aligned}
dec(dk(t_1), enc(ek(t_1), m, t_2)) &= m \\
isenc(enc(ek(t_1), t_2, t_3)) &= enc(ek(t_1), t_2, t_3) \\
isenc(garbageE(t_1, t_2)) &= garbageE(t_1, t_2) \\
isek(ek(t)) &= ek(t) \\
ekof(enc(ek(t_1), m, t_2)) &= ek(t_1) \\
ekof(garbageE(t_1, t_2)) &= t_1 \\
verify(vk(t_1), sig(sk(t_1), t_2, t_3)) &= t_2 \\
issig(sig(sk(t_1), t_2, t_3)) &= sig(sk(t_1), t_2, t_3) \\
issig(garbageSig(t_1, t_2)) &= garbageSig(t_1, t_2) \\
isvk(vk(t_1)) &= vk(t_1) \\
vkof(sig(sk(t_1), t_2, t_3)) &= vk(t_1) \\
vkof(garbageSig(t_1, t_2)) &= t_1 \\
fst(pair(x, y)) &= x \\
snd(pair(x, y)) &= y \\
unstring_0(string_0(s)) &= s \\
unstring_1(string_1(s)) &= s \\
equals(t_1, t_1) &= t_1
\end{aligned}
$$

- Deduction relation: $\vdash$ is the smallest relation satisfying the rules in Figure 1.

$$\frac{m \in S}{S \vdash m} \qquad \frac{N \in \mathbf{N}_E}{S \vdash N} \qquad \frac{S \vdash \underline{t} \quad \underline{t} \in \mathbf{T} \quad F \in \mathbf{C} \cup \mathbf{D} \quad eval_F(\underline{t}) \neq \bot}{S \vdash eval_F(\underline{t})}$$

Figure 1: Deduction rules for the symbolic model of the applied π-calculus

4 Simulator Types

In this section, some variations of the hybrid execution are defined. The definitions are taken from [1] and only changed slightly: The oracle simulator, which is called Sim' in the original proof in [1], is called Sim_o here. Additionally, the case where the decryption oracle is called does not occur anymore and has been removed.

4.1 The Simulator

In the following, we define distinct nonces $N^m \in \mathbf{N}_E$ for each $m \in \{0,1\}^*$. In a hybrid execution, we call a term t *honestly generated* if it occurs as a subterm of a term sent by the protocol Π^C to the simulator before it has occurred as a subterm of a term sent by the simulator to the protocol Π^C.

For an adversary E and a polynomial p, we construct the simulator *Sim* as follows: In the first activation, it chooses $r_N \in \text{Nonces}_k$ for every $N \in \mathbf{N}_P$. It maintains an integer *len*, initially 0. At any point in the execution, $\mathcal{N}$ denotes the set of all nonces $N \in \mathbf{N}_P$ that occurred in terms received from Π^C. $\mathcal{R}$ denotes the set of *randomness nonces* (i.e., the nonces associated with all randomness nodes[4] of Π^C passed through up to that point).

Sim internally simulates the adversary E. When receiving a term $\tilde{t} \in \mathbf{T}$ from Π^C, it passes $\beta(\tilde{t})$ to E where the partial function $\beta : \mathbf{T} \to \{0,1\}^*$ is defined in Appendix 11.3. When E answers with $m \in \{0,1\}^*$, the simulator sends $\tau(m)$ to Π^C where the function $\tau : \{0,1\}^* \to \mathbf{T}$ is defined in Appendix 11.3. The bitstrings sent from the protocol at control nodes are passed through to E and vice versa. When the simulator receives $(info, \nu, t)$, the simulator increases *len* by $\ell(t) + 1$ where $\ell : \mathbf{T} \to \{0,1\}^*$ is defined below. If $len > p(k)$, the simulator terminates, otherwise it answers with $(proceed)$.

The function $\ell : \mathbf{T} \to \{0,1\}^*$ is defined as $\ell(t) := |\beta(t)|$. Note that $\ell(t)$ does not depend on the actual values of r_N because of the length-regularity of A_{enc}, A_{ek}, A_{dk}, A_{sig}, A_{vk}, A_{sk}, A_{pair}, A_{string_0}, and A_{string_1}. Hence $\ell(t)$ can be computed without accessing r_N.

4.2 The simulator with an oracle (Sim_o)

The simulator Sim_o is defined exactly like *Sim*, except that it makes use of an encryption and a signing oracle (these oracles also supply keypairs (ek_N, dk_N), resp. (vk_N, sk_N)).

[4]Randomness nodes are defined in the protocol conditions, see Appendix 11.2

When computing $\beta(ek(N))$ or $\beta(dk(N))$ with $N \in \mathcal{N}$, it instructs the encryption oracle to generate a new encryption/decryption key pair (ek_N, dk_N) (unless (ek_N, dk_N) are already defined) and retrieves ek_N or dk_N from the oracle, respectively. When computing $\beta(enc(ek(N), t, M))$ with $N, M \in \mathcal{N}$, instead of computing $A_{enc}(A_{ek}(r_N), \beta(t), r_M)$, Sim_o requests the encryption $enc(ek_N, \beta(t))$ of $\beta(t)$ from the encryption oracle (that is, Sim_o has to compute $\beta(t)$ but does not need to retrieve ek_N). However, the resulting ciphertext is stored and when later computing $\beta(enc(ek(N), t, M))$ with the same arguments, the stored ciphertext is reused. When computing $\beta(enc(ek(N^e), t_2, M))$ with $M \in \mathcal{N}$, Sim_o requests the encryption $enc(e, \beta(t))$ from Sim_o. (In this case, the oracle encrypts $\beta(t)$ using its own randomness but using the encryption key e provided by Sim_o.)

Similarly, to compute $\beta(vk(N))$ or $\beta(sk(N))$, Sim_o retrieves keys vk_N or sk_N from the signing oracle. To compute $\beta(sig(sk(N), t, M))$, Sim_o invokes the signing oracle with message $\beta(t)$ to get a signature under the signing key sk_N. However, the resulting signature is stored and when later computing $\beta(sig(sk(N), t, M))$ with the same arguments, the stored ciphertext is reused. Sim_o does not invoke the signing oracle for verifying signatures, instead Sim_o executes A_{verify} directly (as does Sim).

4.3 The faking simulator (Sim_f)

The simulator Sim_f is defined like Sim_o, except that when computing $\beta(enc(ek(N), t, M))$ with $N, M \in \mathcal{N}$, instead of invoking the encryption oracle with plaintext $\beta(t)$, it invokes it with plaintext $0^{\ell(t)}$. (But in a computation $\beta(enc(ek(N^e), t, M))$ with $M \in \mathcal{N}$, the simulator Sim_f still uses $\beta(t)$ as plaintext.)

5 The original proof

The main idea of the proof is the following: It shows that the hybrid execution with the faking simulator cannot be distinguished from a computational execution with the real protocol. Thus a link between the computational execution and the more idealized execution, consisting of the symbolic protocol and a simulator that hides the messages of some encryptions is established. Then it shows that the faking hybrid execution preserves the symbolic properties that are of interest.

5.1 Computational implementation

The computational implementation uses an IND-CCA2 secure encryption scheme as well as a strongly unforgeable signature scheme. Up to the IND-CCA2 property, the implementation conditions are the ones that we use for our model. These can be found in section 11.1 while the original conditions are listed in [1].

5.2 Key-safe protocols

On the protocol side there are also some requirements that may seem to be quite restrictive. Here, the notion of "Key-safe protocols" is defined. In short, *key-safe* means that for key generation, encryption and signing the protocol always has to use a fresh randomness, that it does not produce garbage by itself and that no secret keys are sent over the network. Just like the implementation conditions, the conditions for the protocol are important for our model and listed in Section 11.2. They equal the original conditions up to our new condition 10.

5.3 Lemma: Indistinguishability of Sim and Sim_f

The full traces $H\text{-}Trace_{\mathbf{M},\Pi_p,Sim}$ *and* $H\text{-}Trace_{\mathbf{M},\Pi_p,Sim_f}$ *are computationally indistinguishable.*

This lemma states that a run of the faking simulator cannot be distinguished from a run of the original simulator. Thus, one can use the nice properties of the faking simulator in the later proofs, where encryptions, from the point of view of the adversary, are not even related to the messages they hide. Using the IND-CCA2 property this Lemma is easy to show.

5.4 Lemma: Indistinguishability of Sim (and the computational execution)

Sim is indistinguishable for $\mathbf{M}$, Π, A, *and for every polynomial* p.

Using the previous lemma (Indistinguishability of Sim and Sim_f) it is shown that the hybrid execution with Sim, where the simulated adversary and the symbolic protocol interact, cannot be distinguished from the computational execution, where the computational adversary interacts with the implementation of the protocol. In combination with the previous lemma there is now a bridge between the (real) computational world and a idealized view on this world with some aspects from the symbolic world.

The proof is done by showing that the translation functions cancel out and that moving the computational effort from the actual implementation to the simulator is not noticeable.

Now that the connection between the computational and the (faking) hybrid world is established, it is left to connect the (faking) hybrid simulation to the purely symbolic one. To do so, is is shown that the faking simulator has the DY property.[5] When the (faking) simulator has this property, then every such hybrid execution also directly corresponds to a symbolic execution.[6] To show the DY-ness, the notion of a "bad"

[5] DY-ness: Whenever the simulator sends a term t to the protocol, this term could have been deduced from the set S of terms that the adversary received before, i.e. $S \vdash t$.

[6] Note that the only restriction for messages (terms) m from an "adversary" in a symbolic execution is the fact that they have to be deducible from the set of messages S that were sent beforehand, namely $S \vdash m$.

subterm is introduced. Intuitively, the idea is that whenever a term is sent from the simulator to the protocol, that does not fulfill the DY property, then there is a "bad subterm" which cannot exist.

5.5 Lemma: A bad subterm

In a given step of the hybrid execution with Sim_f, let S be the set of messages sent from Π^c to Sim_f. Let $u' \in \mathbf{T}$ be the message sent from Sim_f to Π^c in that step. Let $\mathcal{C}$ be a context and $u \in \mathbf{T}$ such that $u' = \mathcal{C}[u]$ and $S \nvdash u$ and $\mathcal{C}$ does not contain a subterm of the form $sig(\Box, \cdot, \cdot)$. ($\Box$ denotes the hole of the context $\mathcal{C}$.)

Then there exists a term t_{bad} and a context $\mathcal{D}$ such that $\mathcal{D}$ obeys the following grammar

$$\begin{aligned}\mathcal{D} ::= \Box \mid & \mathit{pair}(t, \mathcal{D}) \mid \mathit{pair}(\mathcal{D}, t) \mid \mathit{enc}(\mathit{ek}(N), \mathcal{D}, M) \\ \mid & \mathit{enc}(\mathcal{D}, t, M) \mid \mathit{sig}(\mathit{sk}(M), \mathcal{D}, M) \\ \mid & \mathit{garbageE}(\mathcal{D}, M) \mid \mathit{garbageSig}(\mathcal{D}, M) \\ & \text{with } N \in \mathbf{N}_P, M \in \mathbf{N}_E, t \in \mathbf{T}\end{aligned}$$

and such that $u = \mathcal{D}[t_{bad}]$ and such that $S \nvdash t_{bad}$ and such that one of the following holds: $t_{bad} \in \mathbf{N}_P$, or $t_{bad} = \mathit{enc}(p, m, N)$ with $N \in \mathbf{N}_P$, or $t_{bad} = \mathit{sig}(k, m, N)$ with $N \in \mathbf{N}_P$, or $t_{bad} = \mathit{sig}(\mathit{sk}(N), m, M)$ with $N \in \mathbf{N}_P$, $M \in \mathbf{N}_E$ or $t_{bad} = \mathit{ek}(N)$ with $N \in \mathbf{N}_P$, or $t_{bad} = \mathit{vk}(N)$ with $N \in \mathbf{N}_P$.

This lemma states that whenever the simulator sends some term t with $S \nvdash t$ then there is a subterm t_{bad} of t which is "bad" in the sense that it contradicts some other properties.

The lemma follows from the structure of terms and thus can be proven by structural induction on $\mathbf{M}$.

5.6 Lemma: β does not leak information

For any (direct or recursive) invocation of $\beta(t)$ performed by Sim_f, we have that $S \vdash t$ where S is the set of all terms sent by Π^c to Sim_f up to that point.

The lemma basically states that the translation function β itself is not applied to some (sub-)terms that are not deducible from the messages that were sent before. Thus, β is not applied to secret terms. This is needed in the next proof to exclude that the adversary has learned a secret that was leaked by β.

The lemma is shown by distinguishing the cases where β is called: When it is called directly by the faking simulator Sim_f with a message t, then this message was sent by Π^c to the simulator and thus $t \in S$ must hold. When β is not called directly but as a recursive call from β itself, there are only a few cases possible. By investigating the definition of β (to be found in Appendix 11.3) it is shown that all of the cases fulfill the lemma.

5.7 Lemma: Sim_f is DY

Sim_f is DY for $\mathbf{M}$ *and* Π.

The idea of the proof is the following one: If Sim_f is not DY, then there will be a term u sent from Sim_f to the protocol, such that $S \nvdash u$ where S again is the set of messages sent from the protocol to Sim_f before. By the *bad subterm lemma* (Section 5.5) we know that there must be a term t_{bad} and a context $\mathcal{D}$ s.t. $u = \mathcal{D}[t_{bad}]$. Now it is shown that the existence of such a term t_{bad} is only possible when the simulator came up with a corresponding bitstring m_{bad} before.

Then it is shown that the bitstring m_{bad} can only be found with negligible probability.

5.8 Theorem (computational soundness)

The implementation A (satisfying the implementation conditions [...]) is a computationally sound implementation of the symbolic model $\mathbf{M}$ *[...] for the class of key-safe protocols.*

The previous lemma basically concluded the proof: We have that a hybrid execution with *Sim* is indistinguishable from an execution of the computational protocol and also indistinguishable from the faking hybrid execution. Since this faking hybrid execution now has been shown to behave like a symbolic one, the main theorem follows directly.

6 Constructing the proof

The CoSP result, that has been described so far, defines a framework that allows for nice computational soundness proofs. In the given proof for public key encryption schemes, the requirements for the computational implementation are strict. Even if the protocols we are interested in do not even care about the security of their encryptions or if they use signatures in a way that allows for simpler encryption algorithms, the proofs we have seen so far only work for very strong implementations of encryption schemes.

Our goal was to come up with ideas for weakening these constraints. What would be necessary in order to archieve computational soundness when only having an IND-CPA secure encryption scheme? Starting from the original proof, we constructed a new proof, based on the new assumptions. We then simplified the proof again and again until we surprisingly ended up with a version that only required few additional definitions and lemmas.

6.1 Security notions of public key encryption schemes

In our process of proving computational soundness, security notions play a significant role. In this work we will change the requirements imposed by the CoSP framework on the computational implementations of encryption schemes, allowing the usage of more, even weaker schemes. First, before investigating how the proof structure has evolved, we will briefly explain the terms and definitions that are used. For a more detailed explanation and a comparison between the different notions we refer to [2].

When talking about the security of public key encryptions, there are several approaches to model properties and resistance against specific attacks. The concept of *non-malleability* states that an adversary is not able to modify a given ciphertext in a way such that it decrypts to a similar plaintext as the original one. For example, flipping one or more bits in the ciphertext should not result in the flipping of just a few bits of the outcome of that ciphertext's decryption. A different approach is the concept of *plaintext awareness*. Here a scheme is considered secure, if the creator of a valid ciphertext must always be "aware" of the underlying plaintext. Awareness in this sense intuitively means the knowledge of the underlying plaintext. This is even harder to prove and requires the encryption schemes to be of a certain form in order to be secure under this notion. Nonetheless the concept of plaintext awareness has some nice properties and is an interesting approach.

Another, more common approach, that we will focus on in our work is using the notion of **indistinguishability** of ciphertexts. An encryption scheme is considered secure if an attacker is not able to distinguish between the encryption of two messages, even if he can chose them himself.

Formally, we define a "game" between the adversary and a challenger. The adversary is considered to be an arbitrary probabilistic Turing machine that is allowed to run for an amount of steps polynomial in the security parameter. The challenger behaves according to the given security definition. During the game, the adversary may send queries to the challenger who responds in a defined way. At a certain point, the adversary can send two messages to the challenger. The challenger will then pick one of the messages at random (each with probability $\frac{1}{2}$) and encrypt it, using the given encryption scheme. The resulting ciphertext is sent to the adversary and is called the "challenge". If the adversary is able to guess which of the messages has been encrypted, he wins the game. If he guesses the wrong one, he loses the game. The **advantage** of an adversary is the difference between its probability to win and to $\frac{1}{2}$ (pure guessing).

If the advantage of all adversaries is negligible in the security parameter, the scheme is considered secure. By changing the definition of the challenger machine and by giving the adversary access to special oracles, different security definitions can be created. Some quite commonly used examples are:

- **IND-CPA**
  The adversary is only allowed to have the encryption key or, alternatively, access to an encryption oracle. Thus he can encrypt arbitrary messages using this key, but he has no access to the decryption key or a decryption oracle.

- **IND-CCA1** The adversary additionally has access to a decryption oracle and is able to decrypt arbitrary ciphertexts before he receives the challenge. After receiving the challenge, he loses his access to the decryption oracle.

- **IND-CCA2** The adversary has access to a decryption oracle and is able to decrypt arbitrary ciphertexts before and after he receives the challenge. The only restriction is, that the decryption oracle does not decrypt the challenge-ciphertext for him.

Two of those definitions, namely IND-CPA and IND-CCA2, are central for our work. It is easy to see that the first security restriction is weaker than the second one. While in the IND-CPA setting the adversary only has access to an encryption oracle, in the second case he additionally has access to a decryption oracle.

Every encryption scheme that is secure under IND-CCA2 also is secure under IND-CPA. Thus, by weakening the requirements for the encryption schemes, all previously possible schemes still can be used while additional schemes can be taken into consideration. Since those schemes do not have to fulfill the (strong) requirements for IND-CCA2, they can be less complex. This might especially be interesting when investigating protocols containing zero knowledge proofs, where using simple encryption schemes like ElGamal makes proving properties easier, compared to complex IND-CCA2 secure schemes. Additionally, some commonly used schemes like *CBC Mode* for stream ciphers are not IND-CCA2 secure, but IND-CPA secure.

6.2 Approach

Our idea is the following one: We slightly change the way that messages from the adversary are handled.

In the symbolic world we add a protocol condition, stating that the (symbolic) protocol will never try to decrypt garbage encryptions. Note that the protocol is not able to distinguish a garbage encryption from a normal encryption without trying to decrypt it (there is no destructor that checks if a term is a garbage encryption term). Thus we make sure that no encryption that originally came from the adversary will be decrypted. What the adversary still can do is send a cipher to the protocol that he has already observed in a previous step. This new protocol condition, that we call *garbage free decryption*, is defined below. Since this property is basically the one that will be used to restrict the possibilities of the adversary, it is crucial for the overall proof.

Definition *A symbolic CoSP protocol* Π_s *has* ***garbage free decryption*** *(GFD) if in every full trace of* Π_s *the second argument of every dec destructor node is not annotated with a garbage encryption term.*

Since the only possibility to introduce a *garbage encryption term* is via an input node, this definition captures the notion of not decrypting any term that the adversary has constructed on his own.

In the hybrid execution, the simulator handles the flow of information both from the protocol to the adversary and vice versa. Since we do not want the protocol to decrypt ciphertexts from the adversary, we just define that every cipher that has not been generated by the protocol itself is considered to be garbage. To do this, we change the translation function τ that translates bitstrings from the adversary into terms.

Definition *Sim has* ***garbage free decryption*** *(GFD) for* $\boldsymbol{M}, \Pi$ *and A if for the full trace* $H - Trace_{\boldsymbol{M},\Pi_p,Sim}$ *the probability that there is a dec destructor node whose second argument is annotated with a garbage encryption term is negligible.*

Note that in the hybrid execution we allow for a negligible probability of failure.

Changing the simulator

As described in Section 2, the proof makes use of a simulator that connects the symbolic and the computational world. In this so called hybrid execution we make use of two functions: β, which translates the symbolic terms from the protocol into bitstrings, and τ, which translates bitstrings from the adversary into terms. While the first one can remain untouched, the second one is changed slightly as follows:

Change of τ. The following cases handle the translation of ciphertext bitstrings to terms. In the original definition of τ we have:

- $\tau(c) := enc(ek(M), t, N)$ if c has earlier been output by $\beta(enc(ek(M), t, N))$ for some $M \in \mathbf{N}, N \in \mathcal{N}$.
- $\tau(c) := enc(ek(M), m, N^c)$ if c is of type ciphertext and $\tau(A_{ekof}(c)) = ek(N)$ for some $N \in \mathcal{N}$ and $m := A_{dec}(A_{dk}(r_N), c) \neq \perp$.
- $\tau(c) := garbageE(\tau(A_{ekof}(c), N^c)$ if c is of type ciphertext

We modified these lines according to our idea. Whenever the adversary sends a ciphertext that has not been output by β before, we simply consider it to be a garbage encryption. Thus we have:

- $\tau(c) := enc(ek(M), t, N)$ if c has earlier been output by $\beta(enc(ek(M), t, N))$ for some $M \in \mathbf{N}, N \in \mathcal{N}$.
- $\tau(c) := garbageE(\tau(A_{ekof}(c), N^c)$ if c is of type ciphertext

This change ensures that every encryption that has been generated by the adversary himself is handled as a garbage encryption and will not be decrypted when the simulator has **GFD**. For a more detailed look on the functions see Appendix 11.3

6.3 Developing the proof structure

This section can be regarded as a "historical overview". While it is not necessary for understanding the proof itself, the goal of the section is to show how the proof evolved over the time we worked on the thesis.

In order to prove the new theorem of CoSP with IND-CPA, we tried to stay as close to the original proof from [1] as possible. We added our assumption of garbage free decryption in the symbolic case (you will find a formal definition later on on page 27) and slightly changed the definitions of the translation functions used in Sim (as can be seen in Section 6.2). It seemed as if most of the original proof structure could be preserved.

On a second look, the following problem occured: Even in the very first lemma of the encryption related part of the proof, the IND-CCA2 property was used. In order to apply the weaker IND-CPA property, we had to use the assumption of garbage free decryption for the simulator (which is very close to the notion of garbage free decryption for the symbolic protocol as stated above). While this does follow trivially from the fact that the symbolic protocol has GFD by using the computational soundness theorem for IND-CPA secure encryption (the one we want to prove), showing this property without using the theorem seemed a lot harder.

In order to overcome this technical difficulty, we started to design an inductive proof. The idea was to perform a induction over the number of steps that the symbolic protocol makes. When showing the computational soundness step by step the GFD property could be shown for all previous steps. We then could use the property to prove the computational soundness for the actual step.

Analysing the new proof structure, we realised that there is a better approach. For transfering the GFD property from the symbolic execution to the hybrid execution, it suffices to show that Sim was Dolev-Yao first. After that, we have a connection between the symbolic and the hybrid world and this connection then can be used to transfer the GFD property. By rearranging the parts of the proof, we managed to reduce the overall complexity: While we did not get rid of the induction using this idea, we managed to reduce its size: Now we planned to do an induction for proving that Sim is Dolev-Yao and has GFD. The rest of the proof, though restructured and expanded, would at least not be a giant induction.

While restructuring the proof, trying to come up with good proof ideas for the new and old lemmas, at some point we realised something quite interesting: We had a sketch of the overall proof at our board, with lots of arrows, annotations and ideas, that was astonishing due to one fact: There was no loop.

We would now first show that Sim_f is Dolev-Yao and thus fulfills our notion of garbage free decryption. Then we would show Sim_f and Sim to be indistinguishable

and transfer the GFD property from Sim_f to Sim. Finally, we would use the fact that Sim has GFD to prove it to be indistinguishable from the computational execution.

For showing that Sim_f is DY, no additional requirements have to be met. The indistinguishability however appeared to be a different thing. To prove that $Sim_f \approx Sim$, we first introduced another type of hybrid execution: The **stopping** hybrid execution. This behaves exacltly like the hybrid execution but whenever the GFD property would be violated, it just halts:

For a Simulator Sim, *the* ***stopping hybrid execution*** *(Sim^{STOP}) is defined like the hybrid execution with the difference that whenever a dec destructor node has a garbage encryption term as input, the stopping simulator sends* ***"STOP"*** *and then stops immediately.*

This definition allowed us to first analyse a world, where the GFD property is not violated.

Together with our change to τ (as defined in Section 6.2), we could be sure, that no ciphertexts generated by the adversary can be decrypted. After showing the indistinguishability of Sim_f^{STOP} and Sim^{STOP}, the indistinguishability of Sim_f and Sim follows directly. Since Sim_f had been shown to have GFD, we could also prove that the stopping faking hybrid execution did not stop and from the previous indistinguishability result transfer the GFD property from Sim_f to Sim.

In fact, when analysing the proof again, we realised that the stopping hybrid execution was not even necessary. Our change in the translation functions made sure that there was no real decryption (outside the symbolic protocol). Additionally, τ would translate encryptions that had not been output by β before, into garbage encryption terms. Although the GFD property was not shown for Sim, these terms could not be decrypted by the (symbolic) protocol as the *dec* destructor would simply fail. Again, we removed the redundant definitions and lemmas and simplified the proof structure.

In the end we came up with a proof that seems to be quite similar to the original one. Many of the lemmas from [1] could be reused and most of the proofs only had to be changed slightly in order to work in our setting. This is a surprising result. The main work we did here was not to just copy the original proofs, but to generate a new one and simplify it again and again. We did so until we reached a point where we could prove the computational soundness result with IND-CPA by just introducing a few new definitions and were able to use the same proof techniques that were used for the original CoSP proof, although ordered differently.

7 Our proof

7.1 Overview

The proof is structured into the following steps:

- **Proving that Sim_f is Dolev Yao.** In the hybrid execution with Sim as defined in Section 4.1, a computational adversary is simulated. This simulated adversary interacts with the symbolic protocol. In this interaction messages from the protocol to the adversary and vice versa are translated via translation functions β and τ. In the faking hybrid execution with Sim_f, when translating messages from the protocol to the adversary, every encryption is a zero-encryption. This scenario results in an adversary that is quite similar to the one existing in the idealised symbolic execution. The (simulated) adversary is unable to get information out of encryptions himself and has to either just guess or deduce information out of the data he receives.

  To prove this, we follow the idea and use the lemmas from [1]. First it is shown that whenever a subterm of a message sent by the faking simulator Sim_f to the protocol is not (symbolically) deducible from the messages sent from the protocol to the simulator in previous steps, then it must contain some kind of secret, or *bad* subterm. Additionally, we show that the undeducible subterm must be of a specific form.

  Then we show that whenever the translation function β is called with a term t, this term also must be (symbolically) deducible from the previously received terms. This ensures that there is no secret revealed to the adversary as a result of the translation function. Whenever the adversary comes up with a string that corresponds to some secret, he must have created it by himself.

  Finally these lemmas are combined to prove that whenever a message sent from the simulator to the protocol contains such a bad subterm t_{bad}, then the simulator must have created a corresponding bitstring m_{bad} before. By the definition of t_{bad}, this is only possible by pure guessing and the probability for this is negligible. Thus, every message sent from Sim_f to the protocol is deducible from the set of previously received messages.

- **Proving that Sim_f has Garbage Free Decryption.** For our proof, it is important that the notion of Garbage Free Decryption, which is a necessary property of the symbolic protocol, also holds in the hybrid case. This property ensures that an IND-CPA secure encryption scheme suffices for the computational soundness result. Since the faking hybrid execution is quite similar to the symbolic execution, the GFD property can be shown easily.

  To prove it we use the previous results: We know that Sim_f is Dolev Yao and thus with an overwhelming probability behaves like a symbolic adversary. Since the symbolic protocol fulfills the GFD property for symbolic adversaries, it must also fulfill it in the faking hybrid execution with overwhelming probability.

- **Proving Sim_f and Sim to be indistinguishable.** Until now, we only had a look at the faking hybrid execution and have shown some nice properties for it. The next step is relating the faking hybrid execution and the normal, non-faking hybrid execution. Although this has also been done in [1], it seems to be more complicated in this context, because we can not just use the IND-CCA2 security of the encryption scheme.

  The changes in τ lead to an interesting fact: In our faking hybrid execution, the decryption oracle is never called. Thus, there will never be a real decryption of ciphers sent by the adversary. This allows us to apply the IND-CPA property to show the indistinguishability of Sim_f and Sim.

- **Proving properties of Sim.** After we have shown all the previously mentioned results, we can now use them to prove some interesting properties of Sim which are needed for proving the computational soundness theorem.

  First, we show that since Sim and Sim_f are indistinguishable from each other, the GFD property must hold for Sim too. If it would not hold, they could be distinguished.

  Now Sim has to finally meet its destiny: Designed as an hybrid execution that stands between the purely symbolic one and the computational one, it has to function as a bridge between these two worlds. In order to achieve this, we show that on the one hand it also fulfills the Dolev Yao property. This can be shown easily using the fact that Sim and Sim_f are indistinguishable and that Sim_f is Dolev Yao. On the other hand we show that the hybrid execution with Sim is indistinguishable from the computational execution. To do so, we fix the randomness of the adversary (both the real one and the simulated one) and by comparing the behaviour of both executions we show that we yield indistinguishable node traces.

- **Proving Computational Soundness with IND-CPA.** In this final part of the proof, we plug together the individual lemmas and conclude the overall proof. From the properties of Sim (shown in the previous part) it follows directly that Sim is a good simulator in the sense of Definition 16. Using Theorem 1, this directly implies computational soundness.

7.2 Definitions

We will use the following definitions in the proof. Some of them have already been mentioned and formulated in the previous sections. Nevertheless, we repeat them here.

Definition 17. *A symbolic CoSP protocol Π_s has **garbage free decryption** (GFD) iff in every full trace of Π_s the second argument of every dec destructor node is not annotated with a garbage encryption term.*

Since the only possibility to introduce a *garbage encryption term* is via an input node, this definition captures the notion of not decrypting any term that the adversary has constructed on his own. Note that although normal encryptions are not included, no encryption term generated by the adversary will be decrypted. If the protocol would try to decrypt an encryption term that was sent by the adversary, then the adversary could have sent a garbage encryption term instead. With the given set of destructors, the protocol can not distinguish encryptions and garbage encryptions without applying the *dec* destructor. Trying (and failing) to decrypt nonces, encryption keys or signature keys etc. from the adversary is still possible, but does not harm us.

Definition 18. *Sim has* ***garbage free decryption*** *(GFD) for* $\boldsymbol{M}, \Pi$*, and A iff for the full trace* $H\text{-}Trace_{\boldsymbol{M},\Pi_p,Sim}$ *the probability that there is a dec destructor node whose second argument is annotated with a garbage encryption term is negligible.*

Note that in the hybrid execution we allow for a negligible probability of failure.

7.3 Proving the DY property of the faking simulator

To prove that Sim_f is DY, we follow the original lemmas. First we introduce a so called *bad subterm* (t_{bad}) and show that when the DY property is violated, such a term must exist. We also show that the translation function β does not leak secret information. More precisely, we show that whenever β is applied to a term, the adversary could already deduce it. Having eliminated this possibility, we show that the existence of a bad subterm leads to a contradiction.

Lemma 1. *In a given step of the hybrid execution with* Sim_f*, let* S *be the set of messages sent from* Π^c *to* Sim_f*. Let* $u' \in \mathbf{T}$ *be the message sent from* Sim_f *to* Π^c *in that step. Let* $\mathcal{C}$ *be a context and* $u \in \mathbf{T}$ *such that* $u' = \mathcal{C}[u]$ *and* $S \nvdash u$ *and* $\mathcal{C}$ *does not contain a subterm of the form* $sig(\Box, \cdot, \cdot)$*. (*$\Box$ *denotes the hole of the context* $\mathcal{C}$*.)*

Then there exists a term t_{bad} *and a context* $\mathcal{D}$ *such that* $\mathcal{D}$ *obeys the following grammar*

$$\begin{aligned}\mathcal{D} ::= \Box \mid & \mathit{pair}(t, \mathcal{D}) \mid \mathit{pair}(\mathcal{D}, t) \mid \mathit{enc}(\mathit{ek}(N), \mathcal{D}, M) \\ \mid & \mathit{enc}(\mathcal{D}, t, M) \mid \mathit{sig}(\mathit{sk}(M), \mathcal{D}, M) \\ \mid & \mathit{garbageE}(\mathcal{D}, M) \mid \mathit{garbageSig}(\mathcal{D}, M) \\ & \text{with } N \in \mathbf{N}_P, M \in \mathbf{N}_E, t \in \mathbf{T}\end{aligned}$$

and such that $u = \mathcal{D}[t_{bad}]$ *and such that* $S \nvdash t_{bad}$ *and such that one of the following holds:* $t_{bad} \in \mathbf{N}_P$*, or* $t_{bad} = enc(p, m, N)$ *with* $N \in \mathbf{N}_P$*, or* $t_{bad} = sig(k, m, N)$ *with* $N \in \mathbf{N}_P$*, or* $t_{bad} = sig(sk(N), m, M)$ *with* $N \in \mathbf{N}_P$*,* $M \in \mathbf{N}_E$ *or* $t_{bad} = ek(N)$ *with* $N \in \mathbf{N}_P$*, or* $t_{bad} = vk(N)$ *with* $N \in \mathbf{N}_P$*.*

Proof. The lemma is proven by structural induction on M. The following cases remain exactly the same as in the original proof of this lemma with the exception of case 15.

Case 1: *"$u = garbage(u_1)$".*
By protocol condition 9 the protocol does not contain *garbage*-computation nodes. Thus u is not an honestly generated term.[7] Hence it was produced by an invocation $\tau(m)$ for some $m \in \{0,1\}^*$, and hence $u = garbage(N^m)$. Hence $S \vdash u$ in contradiction to the premise of the lemma.

Case 2: *"$u = garbageE(u_1, u_2)$".*
By protocol condition 9 the protocol does not contain *garbageE*-computation nodes. Thus u is not an honestly generated term. Hence it was produced by an invocation $\tau(c)$ for some $c \in \{0,1\}^*$, and hence $u = garbageE(u_1, N^m)$. Since $S \vdash N^m$ and $S \nvdash u$, we have $S \nvdash u_1$. Hence by the induction hypothesis, there exists a subterm t_{bad} of u_1 and a context $\mathcal{D}$ satisfying the conclusion of the lemma for u_1. Then t_{bad} and $\mathcal{D}' := garbageE(\mathcal{D}, N^m)$ satisfy the conclusion of the lemma for u.

Case 3: *"$u = garbageSig(u_1, u_2)$".*
By protocol condition 9 the protocol does not contain *garbageSig*-computation nodes. Thus u is not an honestly generated term. Hence it was produced by an invocation $\tau(c)$ for some $c \in \{0,1\}^*$, and hence $u = garbageSig(u_1, N^m)$. Since $S \vdash N^m$ and $S \nvdash u$, we have $S \nvdash u_1$. Hence by the induction hypothesis, there exists a subterm t_{bad} of u_1 and a context $\mathcal{D}$ satisfying the conclusion of the lemma for u_1. Then t_{bad} and $\mathcal{D}' := garbageSig(\mathcal{D}, N^m)$ satisfy the conclusion of the lemma for u.

Case 4: *"$u = dk(u_1)$".*
By protocol condition 5, any *dk*-computation node occurs only as the first argument of a *dec* destructor node. The output of the destructor *dec* only contains a subterm $dk(u_1)$ if its second argument already contained such a subterm. Hence a term $dk(u_1)$ cannot be honestly generated. But subterms of the form $dk(\cdot)$ are not in the range of τ. (Except if $dk(\cdot)$ was given as argument to a call to β. However, as β is only invoked with terms sent by Π^c, this can only occur if $dk(\cdot)$ was honestly generated or produced by τ.) Thus no term sent by Sim_f contains $dk(\cdot)$. Hence u cannot be a subterm of u'.

Case 5: *"$u = ek(u_1)$ with $u_1 \notin \mathbf{N}_P$".*
By protocol condition 1, the argument of an *ek*-computation node is an N-computation node with $N \in \mathbf{N}_P$. Hence u is not honestly generated. Hence it was produced by an invocation $\tau(e)$ for some $e \in \{0,1\}^*$, and hence $u = ek(N^e)$. Hence $S \vdash u$ in contradiction to the premise of the lemma.

Case 6: *"$u = ek(N)$ with $N \in \mathbf{N}_P$".*
The conclusion of the lemma is fulfilled with $\mathcal{D} := \Box$ and $t_{bad} := u$.

Case 7: *"$u = vk(u_1)$ with $u_1 \notin \mathbf{N}_P$".*
By protocol condition 1, the argument of a *vk*-computation node is an N-computation

[7] as defined in Section 4.1

node with $N \in \mathbf{N}_P$. Hence u is not honestly generated. Hence it was produced by an invocation $\tau(e)$ for some $e \in \{0,1\}^*$, and hence $u = vk(N^e)$. Hence $S \vdash u$ in contradiction to the premise of the lemma.

Case 8: *"$u = vk(N)$ with $N \in \mathbf{N}_P$".*
The conclusion of the lemma is fulfilled with $\mathcal{D} := \Box$ and $t_{bad} := u$.

Case 9: *"$u = sk(N)$".*
Say a subterm $sk(N)$ *occurs free* in some term t' if an occurrence of $sk(N)$ in t' is not the first argument of a *sig* constructor in t'. Since $\mathcal{C}$ is not of the form $sig(\Box, \cdot, \cdot)$, we have that u occurs free in u'. However, by protocol condition 7, Π^c only sends a free $sk(N)$ if Sim_f first sends one. And by construction of τ, Sim_f sends a free $sk(N)$ only if $sk(N)$ was given as an argument to a call to β. And $sk(N)$ is given as an argument to β only if it is sent by Π^c. Hence Sim_f cannot have sent u' in contradiction to the premise of the lemma.

Case 10: *"$u = pair(u_1, u_2)$".*
Since $S \nvdash u$, we have $S \nvdash u_i$ for some $i \in \{1,2\}$. Hence by induction hypothesis, there exists a subterm t_{bad} of u_i and a context $\mathcal{D}$ satisfying the conclusion of the lemma for u_i. Then t_{bad} and $\mathcal{D}' = pair(\mathcal{D}, u_2)$ or $\mathcal{D}' = pair(u_1, \mathcal{D})$ satisfy the conclusion of the lemma for u.

Case 11: *"$u = string_i(u_1)$ with $i \in \{0,1\}$ or $u = empty$".*
Then, since $u \in \mathbf{T}$, u contains only the constructors $string_0, string_1, empty$. Hence $S \vdash u$ in contradiction to the premise of the lemma.

Case 12: *"$u \in \mathbf{N}_P$".*
The conclusion of the lemma is fulfilled with $\mathcal{D} := \Box$ and $t_{bad} := u$.

Case 13: *"$u \in \mathbf{N}_E$".*
Then $S \vdash u$ in contradiction to the premise of the lemma.

Case 14: *"$u = enc(u_1, u_2, N)$ with $N \in \mathbf{N}_P$".*
The conclusion of the lemma is fulfilled with $\mathcal{D} := \Box$ and $t_{bad} := u$.

Case 15: *"$u = enc(u_1, u_2, u_3)$ with $u_3 \notin \mathbf{N}_P$".*
By protocol condition 1, Π^c does not generate such a term, so it must have been produced by τ, but by definition of τ, such a term is not generated. Thus, this case does not occur.

Case 16: *"$u = sig(u_1, u_2, N)$ with $N \in \mathbf{N}_P$".*
The conclusion of the lemma is fulfilled with $\mathcal{D} := \Box$ and $t_{bad} := u$.

Case 17: *"$u = sig(sk(N), u_2, u_3)$ with $u_3 \notin \mathbf{N}_P$ and $N \in \mathbf{N}_P$".*
Since $u \in \mathbf{T}$ we have $u_3 \in \mathbf{N}$, hence $u_3 \in \mathbf{N}_E$. The conclusion of the lemma is fulfilled with $\mathcal{D} := \Box$ and $t_{bad} := u$.

Case 18: *"$u = sig(u_1, u_2, u_3)$ with $S \vdash u_1$ and $u_3 \notin \mathbf{N}_P$ and u_1 is not of the form $sk(N)$ with $N \in \mathbf{N}_P$".*
By protocol condition 1, the third argument of a Sig-computation node is an N-computation node with $N \in \mathbf{N}_P$. Hence u is not honestly generated. Hence it was produced by an invocation $\tau(s)$ for some $s \in \{0,1\}^*$, and hence $u = sig(sk(N), u_2, N^s)$ for some $N \in \mathbf{N}$. Since u_1 is not of the form $sk(N)$ with $N \in \mathbf{N}_P$, we have $N \in \mathbf{N}_E$. From $S \vdash u_1$, $S \vdash N^c$, and $S \nvdash u$ we have $S \nvdash u_2$. Hence by induction hyposthesis, there exists a subterm t_{bad} of u_2 and a context $\mathcal{D}$ satisfying the conclusion of the lemma for u_2. Then Then t_{bad} and $\mathcal{D}' = sig(sk(N), \mathcal{D}, N^s)$ satisfy the conclusion of the lemma for u.

Case 19: *"$u = sig(u_1, u_2, N)$ with $S \nvdash u_1$ and $u_3 \notin \mathbf{N}_P$".*
As in the previous case, $u = sig(sk(N), u_2, N^s)$ for some $N \in \mathbf{N}$. Since $S \nvdash u_1$, $N \notin \mathbf{N}_E$. Hence $N \in \mathbf{N}_P$. Thus conclusion of the lemma is fulfilled with $\mathcal{D} := \Box$ and $t_{bad} := u$.

□

Lemma 2. *For any (direct or recursive) invocation of $\beta(t)$ performed by Sim_f, we have that $S \vdash t$ where S is the set of all terms sent by Π^c to Sim_f up to that point.*

Proof. The proof can remain almost as it was in the original paper. The only change is, that one of the later cases does not occur anymore and thus can be dropped. The other cases still have to be checked carefully.

We show the lemma by performing an induction on the point in time at which $\beta(t)$ has been invoked. Assume that Lemma 2 holds for all invocations prior to the current invocation $\beta(t)$. We can distinguish the following two cases:

The first is that $\beta(t)$ was directly invoked by Sim_f. This implies that t is a message sent from the protocol to Sim_f. Thus, $t \in S$ holds, which directly implies $S \vdash t$.

The second case is that $\beta(t)$ was not directly invoked but as a recursive call from β itself. Assume that $beta(t)$ has been invoked from a call $\beta(t')$ for some term t'. By definition of β this leaves the following cases for t', where $N, M \in \mathcal{N}$, $u \in \mathbf{T}$:

- $t' = enc(t, u, M)$
- $t' = sig(sk(N), t, M)$
- $t' = pair(t, u)$ or $t' = pair(u, t)$
- $t' = string_0(t)$ or $t' = string_1(t)$
- (The case $t' = enc(ek(N), t, M)$ with $N, M \in \mathcal{N}$, where t occurs as the plaintext inside an encryption does not lead to a recursive call, since we are talking about the faking simulator that encrypts $0^{|\ell(t)|}$ instead of $\beta(t)$.)

- (The original definition of β included a recursive call for an adversary generated encryption term. This case does not occur anymore and thus is omitted in our proof.)

In all these cases from the definition of $\vdash$ and the fact that $S \vdash t'$ it follows that $S \vdash t$. Thus, the lemma holds. □

Lemma 3. *Sim_f is DY for M and Π.*

Proof. The proof for this lemma can stay as it was in the original paper. The only difference is, that our change in the translation functions slightly changes the argumentation of one of the cases. This part is marked. Although the proof syntactically stayed as it was after all, it had to be checked carefully. Since there were changes in the definitions, the semantics of the proof have changed.

Let $a_1, \ldots, a_n$ be terms sent by the protocol to Sim_f. Let $u_1, \ldots, u_n$ be the terms sent by Sim_f to the protocol. Let $S_i := \{a_1, \ldots, a_i\}$. If Sim_f is not DY, then with non-negligible probability there exists an i such that $S_i \nvdash u_i$. Fix the smallest such i_0 and set $S := S_{i_0}$ and $u := u_{i_0}$. By Lemma 1 (with $u' := u$ and $\mathcal{C} := \Box$), we have that there is a term t_{bad} and a context $\mathcal{D}$ obeying the grammar given in Lemma 1 and such that $u = \mathcal{D}[t_{bad}]$ and such that $S \nvdash t_{bad}$ and such that one of the following holds:

- $t_{bad} \in \mathbf{N}_P$, or
- $t_{bad} = enc(p, m, N)$ with $N \in \mathbf{N}_P$, or
- $t_{bad} = sig(k, m, N)$ with $N \in \mathbf{N}_P$, or
- $t_{bad} = sig(sk(N), m, M)$ with $N \in \mathbf{N}_P$, $M \in \mathbf{N}_E$ or
- $t_{bad} = ek(N)$ with $N \in \mathbf{N}_P$, or
- $t_{bad} = vk(N)$ with $N \in \mathbf{N}_P$.

By construction of the simulator, if the simulator outputs u, we know that the simulated adversary E has produced a bitstring m such that $\tau(m) = u = \mathcal{D}[t_{bad}]$. By definition of τ, during the computation of $\tau(m)$, some recursive invocation of τ has returned t_{bad}. Hence the simulator has computed a bitstring m_{bad} with $\tau(m_{bad}) = t_{bad}$.

We are left to show that such a bitstring m_{bad} can be found only with negligible probability.

We distinguish the possible values for t_{bad} (as listed in 1):

Case 1: *"$t_{bad} = N \in \mathbf{N}_P$".*

By definition of β and using the fact that Sim_f uses the signing and encryption oracle for all invocations of β except $\beta(N)$ that involve r_N (such as $\beta(dk(N))$), we have that Sim_f accesses r_N only when computing $\beta(N)$ and in τ. Since $S \nvdash t_{bad} = N$, by Lemma 2 we have that $\beta(N)$ is never invoked, thus r_N is never accessed

through β. In τ, r_N is only used in comparisons. More precisely, $\tau(r)$ checks for all $N \in \mathcal{N}$ whether $r = r_N$. Such checks do not help in guessing r_N since when such a check succeeds, r_N has already been guessed. Thus the probability that $m_{bad} = r_N$ occurs as input of τ is negligible.

Case 2: *"$t_{bad} = enc(p, m, N)$ with $N \in \mathbf{N}_P$".*
Then $\tau(m_{bad})$ returns t_{bad} only if m_{bad} was the output of an invocation of $\beta(enc(p, m, N)) = \beta(t_{bad})$. But by Lemma 2, $\beta(t_{bad})$ is never invoked, so this case does not occur.

Case 3: *"$t_{bad} = sig(k, m, N)$ with $N \in \mathbf{N}_P$".*
Then $\tau(m_{bad})$ returns t_{bad} only if m_{bad} was the output of an invocation of $\beta(sig(k, m, N)) = \beta(t_{bad})$. But by Lemma 2, $\beta(t_{bad})$ is never invoked, so this case does not occur.

Case 4: *"$t_{bad} = sig(sk(N), m, M)$ with $N \in \mathbf{N}_P$, $M \in \mathbf{N}_E$".*
Then $\tau(m_{bad})$ returns t_{bad} only if m_{bad} was not the output of an invocation of β. In particular, m_{bad} was not produced by the signing oracle. Furthermore, $\tau(m_{bad})$ returns t_{bad} only if m_{bad} is a valid signature with respect to the verification key vk_N. Hence m_{bad} is a valid signature that was not produced by the signing oracle. Such a bitstring m_{bad} can only be produced with negligible probability by E because of the strong existential unforgeability of $(\mathsf{SKeyGen}, \mathsf{Sig}, \mathsf{Verify})$ (implementation condition 20).

Case 5: *"$t_{bad} = ek(N)$ with $N \in \mathbf{N}_P$".*
Then by Lemma 2, $\beta(ek(N))$ is never computed and hence ek_N never requested from the encryption oracle. Furthermore, from protocol conditions 5 and 2, we have that no term sent by Π^c contains $dk(N)$, and all occurrences of N in terms sent by Π^c are of the form $ek(N)$. Thus $S \nvdash dk(N)$. Hence by Lemma 2, $\beta(dk(N))$ is never computed and dk_N is never requested from the encryption oracle. Furthermore, since $S \nvdash ek(N)$, for all terms of the form $t = enc(ek(N), \dots, \dots)$, we have that $S \nvdash t$. Thus $\beta(t)$ is never computed and hence no encryption using ek_N is ever requested from the encryption oracle.

In the original proof, it was possible, that decryption queries with respect to dk_N were sent to the encryption oracle. Since there are no decryption queries in our definition, this case does not occur anymore.

Case 6: *"$t_{bad} = vk(N)$ with $N \in \mathbf{N}_P$".*
Then by 2, $\beta(vk(N))$ is never computed and hence vk_N is never requested from the signing oracle. Furthermore, since $S \nvdash vk(N)$, we also have $S \nvdash sk(N)$ and $S \nvdash t$ for $t = sig(sk(N), \dots, \dots)$. Thus $\beta(sk(N))$ and $\beta(t)$ never computed and hence neither sk_N nor a signature with respect to sk_N is requested from the signing oracle. Hence the probability that $vk_N = m_{bad}$ occurs as output of τ is negligible.

Summarizing, we have shown that if the simulator Sim_f is not DY, then with non-negligible probability Sim_f performs the computation $\tau(m_{bad})$, but m_{bad} can only occur with negligible probability as an argument of τ. Hence we have a contradiction to the assumption that Sim_f is not DY. □

7.4 Garbage Free Simulator

Lemma 4. *Sim_f has garbage free decryption.*

Proof. We have to show that in the full trace $H - Trace_{M,\Pi_p,Sim_f}$, the probability that a *garbageEnc* term occurs as second argument of a *dec* destructor node is negligible.

Since Sim_f is DY, it holds that in an execution of $Sim_f + \Pi_p$, the probability is overwhelming, that for every l s.t. $m_l \in \mathbf{T}$ sent as l-th message (input node of Π_p),

$$T_l \vdash m_l$$

Where T_l is the set of terms that Sim_f has received from Π_p prior to sending this message.

Now let a hybrid execution with a full trace $(S_1, \nu_1, f_1), (S_2, \nu_2, f_2), \ldots$ be fixed. Assume the DY-property holds for this execution. Furthermore assume that in this execution there is a *dec* destructor node with *garbageEnc* as second input.
Since the DY-property holds for this execution, we have that every term sent from Sim_f to Π_p was deducible from the terms sent to Sim_f before. Thus the full hybrid trace also constitutes a full symbolic trace for Π_s where there is a *dec* destructor node with *garbageEnc* as second input. This, however, is a contradiction to the fact that Π_s has GFD.

Thus, with overwhelming probability, there is no *dec* destructor node whose second argument is annotated with a garbage encryption term, so Sim_f has garbage free decryption. □

7.5 Sim and Sim_f indistinguishable

Lemma 5. *The full traces $H\text{-}Trace_{\mathbf{M},\Pi_\mathrm{p},Sim}$ and $H\text{-}Trace_{\mathbf{M},\Pi_\mathrm{p},Sim_f}$ are computationally indistinguishable.*

Proof. One might expect this proof to be completely different from the original one. In fact, we first thought that some additional constructs and lemmas are necessary. Our change to the translation functions however reduces this additional complexity. As in the original proof, we first have a look at *Sim* and Sim_o.

The difference between *Sim* and Sim_o lies in the handling of the randomness. While *Sim* uses randomness nonces (r_N) for key generation, encryption and signing, in Sim_o the randomness is chosen by the oracles for key generation, encryption and signing themselves. From protocol conditions 1, 2, 3, 4, it follows that *Sim* never uses a given randomness r_N twice (note that, since $N \in \mathcal{R}$, τ does not access r_N either). Hence the full traces $H\text{-}Trace_{\mathbf{M},\Pi_\mathrm{p},Sim}$ and $H\text{-}Trace_{\mathbf{M},\Pi_\mathrm{p},Sim_o}$ are indistinguishable. Note that by definition of τ, Sim_o never invokes $dec(dk_N, c)$.

Our definition of τ ensures that the decryption oracle is never queried (since there is no call for decryption inside of τ). While ciphertexts that were generated by the protocol still can be decrypted (symbolically), no encryptions from the adversary will

be decrypted. Note that they may be translated to *garbageE* terms but when the symbolic protocol tries to apply the *dec* destructor to them it will fail. So even if the GFD property is violated, they will not be decrypted by either the simulator-oracle or the protocol. This allows to apply the IND-CPA property.

Since $|\beta(t)| = |0^{\ell(t)}|$ by definition of ℓ, the IND-CPA property of $(\mathsf{KeyGen}, enc, dec)$ (implementation condition 19) implies that the full traces $H\text{-}Trace_{\mathbf{M},\Pi_\mathsf{p},Sim_o}$ and $H\text{-}Trace_{\mathbf{M},\Pi_\mathsf{p},Sim_f}$ are indistinguishable. Using the transitivity of computational indistinguishability, the lemma follows. □

7.6 Properties of Sim

Lemma 6. *Sim is DY.*

Proof. By Lemma 3, Sim_f is DY for key-safe protocols. Whether a full trace satisfies the conditions from Definition 14 can be efficiently verified (since $\vdash$ is efficiently decidable). Hence, Lemma 5 implies that *Sim* is DY for key-safe protocols too. □

Lemma 7. *Sim has garbage free decryption.*

Proof. By Lemma 4 we know that Sim_f has GFD. Thus the probability, that there will be a garbage encryption as an input of a *dec* destructor node is negligible. If the probability would be non neglogible for *Sim*, this would constitute a contradiction to Lemma 5. Thus *Sim* has garbage free decryption. □

Lemma 8. *Sim is indistinguishable for* $\mathbf{M}$, Π, *A, and for every polynomial p.*

Proof. The proof can remain as it was in the original paper. We have to add the assumption that the GFD property holds. The rest of the proof had to be checked carefully but with the exception of one of the cases in Claim 4, nothing changed. The changes are marked again.

We will first show that when fixing the randomness of the adversary and the protocol, the node trace $Nodes^p_{\mathbf{M},A,\Pi_\mathsf{p},E}$ in the computational execution and the node trace $H\text{-}Nodes_{\mathbf{M},\Pi_\mathsf{p},Sim}$ in the hybrid execution are equal. Hence, fix the variables r_N for all $N \in \mathbf{N}_P$, fix a random tape for the adversary, and for each node ν fix a choice e_ν of an outgoing edge.

We assume that the randomness is chosen such that all bitstrings r_N, $A_{ek}(r_N)$, $A_{dk}(r_N)$, $A_{vk}(r_N)$, $A_{sk}(r_N)$, $A_{enc}(e, m, r_N)$, and $sig(s, m, r_N)$ are all pairwise distinct for all $N \in \mathcal{N}$ and all bitstrings e of type encryption key, s of type signing key, and $m \in \{0,1\}^*$ that result from some evaluation of β in the execution and that the GFD condition holds for this trace.

Note that this is the case with overwhelming probability: For terms of different types this follows from implementation condition 5. For keys, this follows from the fact that if two randomly chosen keys would be equal with non-negligible probability, the adversary could guess secret keys and thus break the IND-CPA property or the strong existential unforgeability (implementation conditions 19 and 20). For nonces, if two random nonces

r_N, r_M would be equal with non-negligible probability, so would encryption keys $A_{ek}(r_N)$ and $A_{ek}(r_M)$. For encryptions, by implementation condition 21, the probability that $A_{enc}(e, m, r_N)$ for random $r_N \in \text{Nonces}_k$ matches any given string is negligible. Since by protocol condition 4, each $A_{enc}(e, m, r_N)$ computed by β uses a fresh nonce r_N, this implies that $A_{enc}(e, m, r_N)$ equals a previously computed encryption is negligible. Analogously for signatures (implementation condition 22, protocol conditions 4 and 8).

In the following, we designate the values f_i and ν_i in the computational execution by f'_i and ν'_i, and in the hybrid execution by f_i^C and ν_i^C. Let s'_i denote the state of the adversary E in the computational model, and s_i^C the state of the simulated adversary in the hybrid model.

Claim 1: In the hybrid execution, for any $b \in \{0,1\}^*$, $\beta(\tau(b)) = b$.

This claim follows by induction over the length of b and by distinguishing the cases in the definition of τ.

Claim 2: In the hybrid execution, for any term t stored at a node ν, $\beta(t) \neq \bot$.

By induction on the structure of t.

Claim 3: For all terms $t \notin \mathcal{R}$ that occur in the hybrid execution, $\tau(\beta(t)) = t$.

By induction on the structure of t and using the assumption that r_N, $A_{ek}(r_N)$, $A_{dk}(r_N)$, $A_{vk}(r_N)$, $A_{ek}(r_N)$, as well as all occuring encryptions and signatures are pairwise distinct for all $N \in \mathcal{N}$. For terms t that contain randomness nonces, note that by protocol condition 4, randomness nonces never occur outside the last argument of *enc*-, *sig*-, *ek*-, *dk*-, *vk*-, or *sk*-terms.

Claim 4: In the hybrid execution, at any computation node $\nu = \nu_i$ with constructor or destructor F and arguments $\bar{\nu}_1, \ldots, \bar{\nu}_n$ the following holds: Let t_i be the term stored at node $\bar{\nu}_i$ (i.e., $t_j = f'_i(\bar{\nu}_j)$). Then $\beta(eval_F(\underline{t})) = A_F(\beta(t_1), \ldots, \beta(t_n))$. Here the left hand side is defined iff the right hand side is.

We show Claim 4. We distinguish the following cases:

Case 1: *"$F = ek$".*
Note that by protocol condition 1, we have $t_1 \in \mathbf{N}_P$. Then $\beta(ek(t_1)) = A_{ek}(r_{t_1}) = A_{ek}(\beta(t_1))$.

Case 2: *"$F \in \{dk, vk, sk\}$".*
Analogous to the case $F = ek$.

Case 3: *"$F \in \{pair, fst, snd, string_0, string_0, unstring_0, unstring_1, empty\}$".*
Claim 4 follows directly from the definition of β.

Case 4: *"$F = isek$".*
If $t_1 = ek(t'_1)$, we have that $t'_1 = N \in \mathcal{N}$ or $t'_1 = N^m$ where m is of type ciphertext (as other subterms of the form $ek(\cdot)$ are neither produced by the protocol nor by τ). In both cases, $\beta(ek(t'_1))$ is of type encryption key. Hence $\beta(isek(t_1)) = \beta(ek(t'_1)) = A_{isek}(\beta(ek(t'_1))) = A_{isek}(\beta(t_1))$. If t_1 is not of the form $ek(\cdot)$, then $\beta(t_1)$ is not of type public key (this uses that τ only uses N^m with m of type public key inside a term $ek(N^m)$). Hence $\beta(isek(t_1)) = \bot = A_{isek}(\beta(t_1))$.

Case 5: *"$F \in \{isvk, isenc, issig\}$".*
Similar to the case $F = isek$.

Case 6: *"$F = ekof$".*
If $t_1 = enc(ek(u_1), u_2, M)$ with $M \in \mathcal{N}$, we have that $\beta(t_1) = A_{enc}(\beta(ek(u_1)), \beta(u_2), r_M)$. By implementation condition 8, $A_{ekof}(\beta(t_1)) = \beta(ek(u_1))$. Furthermore, $ekof(t_1) = ek(u_1)$, hence $A_{ekof}(\beta(t_1)) = \beta(ekof(t_1))$. If $t_1 = enc(ek(u_1), u_2, N^m)$, by protocol condition 9, t_1 was not honestly generated. Hence, by definition of τ, m is of type ciphertext, and $ek(u_1) = \tau(A_{ekof}(m))$. Thus with Claim 1, $\beta(ek(u_1)) = A_{ekof}(m)$. Furthermore, we have $\beta(t_1) = m$ by definition of β and thus $A_{ekof}(\beta(t_1)) = \beta(ek(u_1)) = \beta(ekof(t_1))$. If $t_1 = garbageE(u_1, u_2)$, the proof is analogous. In all other cases for t_1, $\beta(t_1)$ is not of type ciphertext, hence $A_{ekof}(\beta(t_1)) = \bot$ by implementation condition 8. Furthermore $ekof(t_1) = \bot$. Thus $\beta(ekof(t_1)) = \bot = A_{ekof}(\beta(t_1))$.

Case 7: *"$F = vkof$".*
If $t_1 = sig(sk(N), u_1, M)$ with $N, M \in \mathcal{N}$, we have that $\beta(t_1) = A_{sig}(A_{sk}(r_N), \beta(u_2), r_M)$. By implementation condition 9, $A_{ekof}(\beta(t_1)) = A_{vk}(r_N)$. Furthermore, $vkof(t_1) = vk(N)$, hence $A_{vkof}(\beta(t_1)) = A_{vk}(r_N) = \beta(vk(N)) = \beta(vkof(t_1))$. All other cases for t_1 are handled like in the case of $F = ekof$.

Case 8: *"$F = enc$".*
By protocol condition 1, $t_3 =: N \in \mathcal{N}$. If $t_1 = ek(u_1)$ we have $\beta(enc(t_1, t_2, t_3)) = A_{enc}(\beta(t_1), \beta(t_2), r_N)$ by definition of β. Since $\beta(N) = r_N$, we have $\beta(enc(t_1, t_2, t_3)) = A_{enc}(\beta(t_1), \beta(t_2), \beta(t_3))$. If t_1 is not of the form $ek(u_1)$, then $enc(t_1, t_2, t_3) = \bot$ and by definition of β, $\beta(t_1)$ is not of type encryption key and hence by implementation condition 10, $\beta(enc(t_1, t_2, t_3)) = A_{ek}(\beta(t_1), \dots) = \bot = \beta(enc(t_1, t_2, t_3))$.

Case 9: *"$F = dec$".*
By protocol condition 6, $t_1 = dk(N)$ with $N \in \mathcal{N}$. We distinguish the following cases for t_2:

Case 9.1: *"$t_2 = enc(ek(N), u_2, M)$ with $M \in \mathcal{N}$".*
Then $A_{dec}(\beta(t_1), \beta(t_2)) = A_{dec}(A_{dk}(r_N), A_{enc}(A_{ek}(N), \beta(u_2), r_M)) = \beta(u_2)$ by implementation condition 12. Furthermore $\beta(dec(t_1, t_2)) = \beta(u_2)$ by definition of dec.

Case 9.2: *"$t_2 = enc(ek(N), u_2, N^c)$".*
Then t_2 was produced by τ and hence c is of type ciphertext and $\tau(A_{dec}(A_{dk}(r_N), c)) = u_2$. Then by Claim 1, $A_{dec}(A_{dk}(r_N), c) = \beta(u_2)$ and hence $A_{dec}(\beta(t_1), \beta(t_2)) = A_{dec}(A_{dk}(r_N), c) = \beta(u_2) = \beta(dec(t_1, t_2))$.

Case 9.3: *"$t_2 = garbageE(u_1, N^c)$".*
By assumption this case does not occur.

Case 9.4: *"All other cases".*

Then $\beta(t_2)$ is not of type ciphertext. By implementation condition 8, $A_{ekof}(\beta(t_2)) = \bot$. Hence $A_{ekof}(\beta(t_2)) \neq A_{ek}(r_N)$ and by implementation condition 11, $A_{dec}(\beta(t_1), \beta(t_2)) = A_{dec}(A_{dk}(r_N), \beta(t_2)) = \bot = \beta(dec(t_1, t_2))$.

Case 10: *"$F = sig$".*

By protocol conditions 8 and 1 we have that $t_1 = sk(N)$ and $t_3 = M$ with $N, M \in \mathcal{N}$. Then $\beta(sig(\underline{t})) = A_{sig}(A_{sk}(r_N), \beta(t_3), r_M) = A_{sig}(\beta(sk(N)), \beta(t_2), \beta(M)) = A_{sig}(\beta(t_1), \beta(t_2), \beta(t_3))$.

Case 11: *"$F = verify$".*

We distinguish the following subcases:

Case 11.1: *"$t_1 = vk(N)$ and $t_2 = sig(sk(N), u_2, M)$ with $N, M \in \mathcal{N}$".*

Then $A_{verify}(\beta(t_1), \beta(t_2)) = A_{verify}(A_{vk}(r_N), A_{sig}(A_{sk}(r_N), \beta(u_2), r_M)) \overset{(*)}{=} \beta(u_2) = \beta(verify(\underline{t}))$ where $(*)$ uses implementation condition 13.

Case 11.2: *"$t_2 = sig(sk(N), u_2, M)$ and $t_1 \neq vk(N)$ with $N, M \in \mathcal{N}$".*

By Claim 3, $\beta(t_1) \neq \beta(vk(N))$ Furthermore $A_{verify}(\beta(vk(N)), \beta(t_2)) = A_{verify}(\beta(t_1), A_{sig}(A_{sk}(r_N), \beta(u_2), r_M)) \overset{(*)}{=} \beta(u_2) \neq \bot$. Hence with implementation condition 14, $A_{verify}(\beta(t_1), \beta(t_2)) = \bot = \beta(\bot) = verify(t_1, t_2)$.

Case 11.3: *"$t_1 = vk(N)$ and $t_2 = sig(sk(N), u_2, M^s)$".*

Then t_2 was produced by τ and hence s is of type signature with $\tau(A_{vkof}(s)) = vk(N)$ and $m := A_{verify}(A_{vkof}(s), s) \neq \bot$ and $u_2 = \tau(m)$. Hence with Claim 1 we have $m = \beta(\tau(m)) = \beta(u_2)$ and $\beta(t_1) = \beta(vk(N)) = \beta(\tau(A_{vkof}(s))) = A_{vkof}(s)$. Thus $A_{verify}(\beta(t_1), \beta(t_2)) = A_{verify}(A_{vkof}(s), s) = m = \beta(u_2)$. And $\beta(verify(t_1, t_2)) = \beta(verify(vk(N), sig(sk(N), u_2, M^s))) = \beta(u_2)$.

Case 11.4: *"$t_2 = sig(sk(N), u_2, M^s)$ and $t_1 \neq vk(N)$".*

As in the previous case, $A_{verify}(A_{vkof}(s), s) \neq \bot$ and $\beta(vk(N)) = A_{vkof}(s)$. Since $t_1 \neq vk(N)$, by Claim 3, $\beta(t_1) \neq \beta(vk(N)) = A_{vkof}(s)$. From implementation condition 14 and $A_{verify}(A_{vkof}(s), s) \neq \bot$, we have $A_{verify}(\beta(t_1), \beta(t_2)) = A_{verify}(\beta(t_1), s) = \bot = \beta(\bot) = \beta(verify(t_1, t_2))$.

Case 11.5: *"$t_2 = garbageSig(u_1, N^s)$".*

Then t_2 was produced by τ and hence s is of type signature and either $A_{verify}(A_{vkof}(s), s) = \bot$ or $\tau(A_{vkof}(s))$ is not of the form $vk(\dots)$. The latter case only occurs if $A_{vkof}(s) = \bot$ as otherwise $A_{vkof}(s)$ is of type verification key and hence $\tau(A_{vkof}(s)) = vk(\dots)$. Hence in both cases $A_{verify}(A_{vkof}(s), s) = \bot$. If $\beta(t_1) = A_{vkof}(s)$ then $A_{verify}(\beta(t_1), \beta(t_2)) = A_{verify}(A_{vkof}(s), s) = \bot = \beta(verify(t_1, t_2))$. If $\beta(t_1) \neq A_{vkof}(s)$ then by implementation condition 14, $A_{verify}(\beta(t_1), \beta(t_2)) = A_{verify}(\beta(t_1), s) = \bot$. Thus in both cases, with $verify(t_1, t_2) = \bot$ we have $A_{verify}(\beta(t_1), \beta(t_2)) = \bot = \beta(verify(t_1, t_2))$.

Case 11.6: *"All other cases".*

Then $\beta(t_2)$ is not of type signature, hence by implementation condition 9,

$A_{vkof}(\beta(t_2)) = \bot$, hence $\beta(t_1) \neq A_{vkof}(\beta(t_2))$, and by implementation condition 14 we have $A_{verify}(\beta(t_1), \beta(t_2)) = \bot = \beta(verify(t_1, t_2))$.

Case 12: *"$F = equals$".*
If $t_1 = t_2$ we have $\beta(equals(t_1, t_2)) = \beta(t_1) = A_{equals}(\beta(t_1), \beta(t_1)) = A_{equals}(\beta(t_1), \beta(t_2))$. If $t_1 \neq t_2$, then $t_1, t_2 \notin \mathcal{R}$. To see this, let N_1 be the node associated with t_1. If N_1 is a nonce computation node, then $t_1 \notin \mathcal{R}$ follows from protocol conditions 2, 3, and 4. In case N_1 is an input node, $t_1 \notin \mathcal{R}$ follows by definition of τ. Finally, if N_1 is a destructor computation node, $t_1 \notin \mathcal{R}$ follows inductively. (Similarly for t_2.) By Claim 3, $t_1, t_2 \notin \mathcal{R}$ implies $\beta(t_1) \neq \beta(t_2)$ and hence $\beta(equals(t_1, t_2)) = \bot = A_{equals}(\beta(t_1), \beta(t_2))$ as desired.

Case 13: *"$F \in \{garbage, garbageE, garbageSig\} \cup \mathbf{N}_E$".*
By protocol condition 9, the constructors *garbage*, *garbageE*, *garbageSig*, and $N \in \mathbf{N}_E$ do not occur in the protocol.

Thus Claim 4 holds.

We will now show that for the random choices fixed above,

$$Nodes^p_{\mathbf{M},A,\Pi_\mathrm{p},E} = H\text{-}Nodes_{\mathbf{M},\Pi_\mathrm{p},Sim}.$$

To prove this, we show the following invariant: $f'_i = \beta \circ f^C_i$ and $\nu'_i = \nu^C_i$ and $s_i = s'_i$ for all $i \geq 0$. We show this by induction on i.

We have $f'_0 = f^C_0 = \varnothing$ and $\nu'_0 = \nu^C_0$ is the root node, so the invariant is satisfied for $i = 0$. Assume that the invariant holds for some i. If ν'_i is a nondeterministic node, $\nu'_{i+1} = \nu^C_{i+1}$ is determined by $e_{\nu'_i} = e_{\nu^C_i}$. Since a nondeterministic node does not modify f and the adversary is not activated, $f'_{i+1} = f'_i = \beta \circ f^C_i = \beta \circ f^C_{i+1}$ and $s_i = s'_i$. Hence the invariant holds for $i + 1$ if ν'_i is a nondeterministic node.

If ν'_i is a computation node with constructor or destructor F, we have that $f'_{i+1}(\nu'_i) = A_F(f'_i(\bar{\nu}_1), \ldots, f'_i(\bar{\nu}_n)) = A_F(\beta(f^C_i(\bar{\nu}_1)), \ldots, \beta(f^C_i(\bar{\nu}_n)))$ for some nodes $\bar{\nu}_s$ depending on the label of ν'_i. And $f^C_{i+1}(\nu'_i) = f^C_{i+1}(\nu^C_i) = eval_F(f^C_i(\bar{\nu}_1), \ldots, f^C_i(\bar{\nu}_n))$. From Claim 4 it follows that $\beta(f^C_{i+1}(\nu'_i)) = f'_{i+1}(\nu'_i)$ where the lhs is defined iff the rhs is. Hence $\beta \circ f^C_{i+1} = f'_{i+1}$.

By Claim 2, $\beta(f^C_{i+1}(\nu^C_i))$ is defined if $f^C_{i+1}(\nu'_i)$ is. Hence $f^C_{i+1}(\nu^C_i)$ is defined iff $f'_{i+1}(\nu'_i)$ is. If $f^C_{i+1}(\nu^C_i)$ is defined, then ν^C_{i+1} is the yes-successor of ν^C_i and the no-successor otherwise. If $f'_{i+1}(\nu'_i)$ is defined, then ν'_{i+1} is the yes-successor of $\nu'_i = \nu^C_i$ and the no-successor otherwise. Thus $\nu^C_{i+1} = \nu'_{i+1}$.

The adversary E is not invoked, hence $s'_{i+1} = s^C_{i+1}$. So the invariant holds for $i + 1$ if ν'_i is a computation node with a constructor or destructor.

If ν'_i is a computation node with nonce $N \in \mathbf{N}_P$, we have that $f'_{i+1}(\nu'_i) = r_N = \beta(N) = \beta(f^C_{i+1}(\nu'_i))$. Hence $\beta \circ f^C_{i+1} = f'_{i+1}$. By 9, the ν'_{i+1} is the yes-successor of ν'_i. Since $N \in \mathbf{T}$, ν^C_{i+1} is the yes-successor of $\nu^C_i = \nu'_i$. Thus $\nu'_{i+1} = \nu^C_{i+1}$. The adversary E is not invoked, hence $s'_{i+1} = s^C_{i+1}$. So the invariant holds for $i + 1$ if ν'_i is a computation node with a nonce.

In the case of a control node, the adversary E in the computational execution and the simulator in the hybrid execution get the out-metadata l of the node ν'_i or ν^C_i, respectively. The simulator passes l on to the simulated adversary. Thus, since $s'_i = s^C_i$, we have that $s'_{i+1} = s^C_{i+1}$, and in the computational and the hybrid execution, E answer with the same in-metadata l'. Thus $\nu'_{i+1} = \nu^C_{i+1}$. Since a control node does not modify f we have $f'_{i+1} = f'_i = \beta \circ f^C_i = \beta \circ f^C_{i+1}$. Hence the invariant holds for $i+1$ if ν'_i is a control node.

In the case of an input node, the adversary E in the computational execution and the simulator in the hybrid execution is asked for a bitstring m' or bitstring t^C, respectively. The simulator produces this string by asking the simulated adversary E for a bitstring m^C and setting $t^C := \tau(m^C)$. Since $s'_i = s^C_i$, $m' = m^C$. Then by definition of the computational and hybrid executions, $f'_{i+1}(\nu'_i) = m'$ and $f^C_{i+1}(\nu'_i) = t^C = \tau(m')$. Thus $f'_{i+1}(\nu'_i) = m' \overset{(*)}{=} \beta(\tau(m')) = \beta(f^C_{i+1}(\nu'_i))$ where $(*)$ follows from Claim 1. Since $f'_{i+1} = f'_i$ and $f^C_{i+1} = f^C$ everywhere else, we have $f'_{i+1} = \beta \circ f^C_{i+1}$. Furthermore, since input nodes have only one successor, $\nu'_{i+1} = \nu^C_{i+1}$. Thus the invariant holds for $i+1$ in the case of an input node.

In the case of an output node, the adversary E in the computational execution gets $m' := f'_i(\bar{\nu}_1)$ where the node $\bar{\nu}_1$ depends on the label of ν'_i. In the hybrid execution, the simulator gets $t^C := f^C_i(\bar{\nu}_1)$ and sends $m^C := \beta(t^C)$ to the simulated adversary E. By induction hypothesis we then have $m' = m^C$, so the adversary gets the same input in both executions. Thus $s'_{i+1} = s^C_{i+1}$. Furthermore, since output nodes have only one successor, $\nu'_{i+1} = \nu^C_{i+1}$. And $f'_{i+1} = f'_i$ and $f^C_{i+1} = f^C$, so $f'_{i+1} = \beta \circ f^C_{i+1}$. Thus the invariant holds for $i+1$ in the case of an output node.

From the invariant it follows, that the node trace is the same in both executions.

Since random choices with all nonces, keys, encryptions, and signatures being pairwise distinct occur with overwhelming probability (as discussed above), the node traces of the real and the hybrid execution are indistinguishable. □

7.7 Computational Soundness Theorem

Theorem 2. *The implementation A (satisfying the implementation conditions listed in Appendix 11.1) is a computationally sound implementation of the symbolic model* **M** *defined in Section 3 for the class of key-safe protocols with garbage free decryption.*

Proof. By Lemma 3, Sim_f is DY for key-safe protocols. Whether a full trace satisfies the conditions from Definition 14 can be efficiently verified (since $\vdash$ is efficiently decidable). Hence Lemma 5 implies that *Sim* is DY for key-safe protocols, too. By Lemma 8, *Sim* is indistinguishable. Hence *Sim* is a good simulator for **M**, key-safe Π, A, and polynomials p. By Theorem 1, the computational soundness of A for key-safe protocols follows. □

8 Computational soundness of the applied π-calculus

We have seen now that for protocols that have GFD (see Definition 17), we can apply the somputational soundness theorem even if the encryption scheme is secure only under IND-CPA. In order to make use of this new property, we now show that it can be transferred to other calculi. Also we come up with a way to test automatically if a protocol satisfies the GFD property.

The definitions of the calculus have been taken from [1], since they can be used for our proof as well. In contrast to our main proof, here we can use some of the lemmas from the original proof, since we do not change the definition of the calculus.

8.1 Syntax and Semantics of the applied π-calculus

The process calculus that we use is the same that was used in the CoSP paper [1]. The syntax is listed in Figure 2. The calculus corresponds to the one considered in [4], extended only by adding event processes $event(e).P$ for strings e. Such an event process raises the event e and then proceeds to execute the process P. Although technically this defines a variant of the applied π-calculus, it is not given a new name for simplicity reasons.

Notation

- To avoid ambiguity, we add the prefix **CoSP-** to terms, constructors, destructors, nodetraces etc. that are defined in the sense of the symbolic Model of CoSP in Section 2.2. Equivalently, terms, constructors, traces etc. that originate in the π-calculus receive the prefix **π-**.
- The set of ground π-terms is denoted T_π.
- For a process P, the set of free names[8] is denoted by $\mu(P)$, while the set of free variables[9] is denoted by $\eta(P)$.
- A process P is called **closed** if it has no free variables (it may have free names).

The applied π-calculus is parametrized over:

- A (possibly infinite) set of π-constructors $\mathbf{C}_\pi$ like the ones from Section 3.
- A (possibly infinite) set of π-destructors $\mathbf{D}_\pi$ also like the ones from Section 3.
- An equivalence relation $\approx$ over ground π-terms that we will call *equational theory*. We require $\approx$ to fulfill the following equations for all π-constructors f and π-destructors d of arity n, for all ground π-terms $M_1, \ldots, M_n, M'_1, \ldots, M'_n$ with $M_i \approx M'_i$ for $i = 1, \ldots, n$

[8] A free name in the sense of this notation is a name n that is not protected by a restriction.
[9] We call a variable x free, if it is not protected by a *let* or an *input* statement.

- $f(\underline{M}) \approx f(\underline{M'})$
- $d(\underline{M}) = \bot$ iff $d(\underline{M'}) = \bot$
- $d(\underline{M}) \approx d(\underline{M'})$
- $d(\underline{M}\tau) = d(\underline{M})\tau$ for any renaming τ of names.

$M, N ::=$	terms
x, y, z	variables
a, b, c	names
$f(M_1, \ldots, M_n)$	constructor application
$D ::=$	destructor terms
M	terms
$d(D_1, \ldots, D_n)$	destructor application
$f(D_1, \ldots, D_n)$	constructor application
$P, Q ::=$	processes
$\bar{M}\langle N\rangle.P$	output
$M(x).P$	input
0	nil
$P \mid Q$	parallel composition
$!P$	replication
$\nu a.P$	restriction
$\mathit{let}\ x = D$	let
$\quad \mathit{in}\ P\ \mathit{else}\ Q$	
$\mathit{event}(e).P$	event

Figure 2: Syntax of the applied π-calculus.

Although there is no explicit definition of an *if-statement* in the syntax of the applied π-calculus, we can emulate such a statement. To do so, we use an additional destructor *equals*. It is defined by $\mathit{equals}(x, y) = x$ for $x \approx y$. When we want to express an if-statement like *if M=N then P else Q*, we write it as Let $x = \mathit{equals}(M, N)$ in P else Q for some $x \notin \eta(P)$. In the following we will assume $\mathit{equals} \in \mathbf{D}_\pi$.

To simplify notation we will use the *if-statement* instead of the longer let-expression and we will write Let $x = D$ in P for Let $x = D$ in P else 0 and analogously for *if*.

Note that, in contrast to the CoSP model from Section 3, we here have *destructor terms*. When dealing with them, we can evaluate a ground destructor π-term D to a ground π-term $\mathrm{eval}^\pi(D)$ by evaluating all the π-destructors. If one of them returns $\bot$,

$$\frac{}{P \mid 0 \equiv P} \qquad \frac{}{P \equiv P} \qquad \frac{}{P \mid Q \equiv Q \mid P}$$

$$\frac{}{(P \mid Q) \mid R \equiv P \mid (Q \mid R)} \qquad \frac{P \equiv Q \qquad Q \equiv R}{P \equiv R}$$

$$\frac{}{\nu a.\nu b.P \equiv \nu b.\nu a.P} \qquad \frac{P \equiv Q}{P \mid R \equiv Q \mid R} \qquad \frac{a \notin \mu(P)}{\nu a.(P \mid Q) \equiv P \mid \nu a.Q}$$

$$\frac{P \equiv Q}{\nu a.P \equiv \nu a.Q} \qquad \frac{N \approx N'}{\overline{N}\langle M\rangle..Q \mid N'(x)..P \to Q \mid P\{M/x\}}$$

$$\frac{\mathrm{eval}^\pi\ D \neq \bot}{\text{Let } x = D \text{ in } P \text{ else } Q \to P\{\mathrm{eval}^\pi\ D/x\}} \qquad \frac{\mathrm{eval}^\pi\ D = \bot}{\text{Let } x = D \text{ in } P \text{ else } Q \to Q}$$

$$\frac{}{!P \to P \mid !P} \qquad \frac{P \to Q}{P \mid R \to Q \mid R \qquad \nu a.P \to \nu a.Q}$$

$$\frac{P' \equiv P \qquad P \to Q \qquad Q \equiv Q'}{P' \to Q'} \qquad \frac{}{event(e).P \xrightarrow{e} P}$$

$$\frac{P \xrightarrow{e} Q}{P \mid R \xrightarrow{e} Q \mid R \qquad \nu a.P \xrightarrow{e} \nu a.Q} \qquad \frac{P' \equiv P \qquad P \xrightarrow{e} Q \qquad Q \equiv Q'}{P' \xrightarrow{e} Q'}$$

Figure 3: Semantics of the applied π-calculus with events.

we set $\mathrm{eval}^\pi(D) := \bot$. Similary we can talk about destructor CoSP terms D including CoSP-destructors, -constructors and -nonces. We define $\mathrm{eval}^{\mathsf{CoSP}}(D)$ analogously to eval^π.

The semantics of the calculus corresponds to the one defined in [4] except for a new type of transitions. In addition to the normal transition $\to$ we define an event transition $\xrightarrow{e}$, denoting that the event e has occured. The rules of the semantics are formally defined in Figure 3.

We use the event transitions to define trace properties as sequences of events that occur during the execution of a process. The formal definition is as follows:

Definition 19 (π-trace properties)**.** A list of strings $e_1, \ldots, e_n$ is an *event trace* of P if there is a process Q that does not contain events such that $P \mid Q \to^* \xrightarrow{e_1} \to^* \xrightarrow{e_2} \to^* \cdots \to^* \xrightarrow{e_n}$. A *$\pi$-trace property* is an efficiently decidable and prefix-closed set of strings. A process P *symbolically satisfies a π-trace property* $\wp$ if we have $e \in \wp$ for all event traces e of P.

8.2 A computational π-execution

In analogy to the computational CoSP implementation, we specify a computational π-implementation as a set of partial deterministic polynomial time algorithms. To each π-constructor or π-destructor f, we assign an algorithm A^π_f. For our equality check function we require that $A^\pi_{equals}(1^k, x, x) = x$ and $A^\pi_{equals}(1^k, x, y) = \bot$ for $x \neq y$. Thus, our computational interpretation of the $\approx$ relation is the equality of bitstrings.

We also fix a set Nonces_k depending on a security parameter k and require it to be efficiently sampleable.

For the computational π execution there are still a few things to define:

- **names and variables:**
  We define two injective functions μ from names to bitstrings and η from variables to bitstrings. Their task is to assign a bitstring to each name or variable respectively.

- **a computational π-evaluation:**
  Analogously to the symbolic case we define $\text{ceval}_{\eta,\mu} D$. Also, as in the symbolic case we define $\text{ceval}_{\eta,\mu} D := \bot$ if at least one of the algorithms A^π_f fails. Note that ceval formally has the security parameter k as an additional input.

- **nondeterminism:**
  We model the nondeterminism by giving the adversary total control over the scheduling. This gives the adversary a lot of power and can be considered a worst-case assumption and thus a safe approximation.

- **information flow to the adversary:**
  In the applied π-calculus, an adversary can receive every message on any channel that he knows. To model this, we allow the (computational) adversary to query for messages. He receives them if he is able to produce the bitstring corresponding to the channels name, which corresponds to the (symbolic) knowledge.

- **evaluation contexts:**
  For the computational implementation of a process we make use of evaluation contexts, defined as follows: *An evaluation context is a context with either one hole, or with two (distinguished) holes where each hole occurs only once and is located only below parallel compositions.*[10] *In the case of two holes, we write $E[P][Q]$ to denote the replacement of the first hole by P and of the second hole by Q.*

Definition 20 (Computational π-execution)**.** Let P_0 be a closed process, and let C be an interactive machine called the adversary. We define the *computational π-execution* as an interactive machine $\text{Exec}_{P_0}(1^k)$ that takes a security parameter k as argument and interacts with C:

[10] Traditionally, one considers evaluation contexts where the hole may also be protected by a restriction. However, computationally the evaluation of a restriction has to be considered a proper reduction step (it corresponds to choosing a nonce).

- *Start*: Let $P := P_0$ (where we rename all bound variables and names such that they are pairwise distinct and distinct from all unbound ones). Let η be a totally undefined partial function mapping variables to bitstrings, let μ be a totally undefined partial function mapping names to bitstrings. Let $a_1, \ldots, a_n$ denote the free names in P_0. For each i, pick $r_i \in \text{Nonces}_k$ at random. Set $\mu := \mu(a_1 := r_1, \ldots, a_n := r_n)$. Send $(r_1, \ldots, r_n)$ to C.[11]
- *Main loop:* Send P to the adversary and expect an evaluation context E from the adversary. Distinguish the following cases:
  - $P = E[M(x).P_1]$: Request two bitstrings c, m from the adversary. If $c = \text{ceval}_{\eta,\mu}(M)$, set $\eta := \eta(x := m)$ and $P := E[P_1]$.
  - $P = E[\nu a.P_1]$: Pick $r \in \text{Nonces}_k$ at random, set $P := E[P_1]$ and $\mu := \mu(a := r)$.
  - $P = E[\overline{M_1}\langle N\rangle..P_1][M_2(x)..P_2]$: If $\text{ceval}_{\eta,\mu}(M_1) = \text{ceval}_{\eta,\mu}(M_2)$ then set $P := E[P_1][P_2]$ and $\eta := \eta(x := \text{ceval}_{\eta,\mu}(N))$.
  - $P = E[\text{Let } x = D \text{ in } P_1 \text{ else } P_2]$: If $m := \text{ceval}_{\eta,\mu}(D) \neq \bot$, set $\eta := \eta(x := m)$ and $P := E[P_1]$. Otherwise set $P := E[P_2]$.
  - $P = E[\mathit{event}(e).P_1]$: Let $P := E[P_1]$ and raise the event e.
  - $P = E[!P_1]$: Rename all bound variables of P_1 such that they are pairwise distinct and distinct from all variables and names in P and in the domains of η and μ, yielding a process $\tilde{P}_1$. Set $P := E[\tilde{P}_1 \mid !P_1]$.
  - $P = E[\overline{M}\langle N\rangle..P_1]$: Request a bitstring c from the adversary. If $c = \text{ceval}_{\eta,\mu}(M)$, set $P := E[P_1]$ and send $\text{ceval}_{\eta,\mu}(N)$ to the adversary.
  - In all other cases, do nothing.

In order to check if trace properties are fulfilled, we again use the events that are raised from our event transitions. We gather the events raised during the computational execution in a list. More precisely, for a given polynomial-time interactive machine C, a closed process P_0 and a polynomial p we define $\mathit{Events}_{C,P_0,p}(k)$ as the list of events e raised within the first $p(k)$ computation steps (where we count the steps from both $C(1^k)$ and $\text{Exec}_{P_0}(1^k)$). Based on these lists, we define, if a π-trace property is computationally fulfilled:

Definition 21 (Computational π-trace properties)**.** Let P_0 be a closed process, and p a polynomial. We say that P_0 *computationally satisfies a π-trace property* $\wp$ if for all polynomial-time interactive machines C and all polynomials p, we have that $\Pr[\mathit{Events}_{C,P_0,p}(1^k) \in \wp]$ is overwhelming in k.

8.3 Towards Computational Soundness

We now want to show the computational soundness of the applied π-calculus using our result from Theorem 2. Since we did not change the applied π-calculus from [1], we will be able to apply the lemmas proven there. However, we have to find a condition for π

[11] In the applied π-calculus, free names occurring in the initial process represent nonces that are honestly chosen but known to the attacker.

$$\frac{m \in S}{S \vdash m} \qquad \frac{N \in \mathbf{N}_E}{S \vdash N} \qquad \frac{S \vdash \underline{M} \qquad f \in \mathbf{C} \setminus \mathbf{N}}{S \vdash f(\underline{M})}$$

$$\frac{S \vdash \underline{M} \qquad d \in \mathbf{D} \qquad d(\underline{M}) \neq \bot}{S \vdash d(\underline{M})}$$

Figure 4: Deduction rules for the symbolic model of the applied π-calculus

protocols that corresponds to the GFD property in the sense of Definition 17. Then we have to prove that this condition implies that the (symbolic) CoSP protocol which we derive from the given π protocol fulfills our GFD property.

In order to make this thesis self contained, we first include the definitions and lemmas that are needed to prove the computational soundness of the applied π-calculus. They are taken from [1] and do not need to be changed.

Definition 22 (Symbolic model of the applied π-calculus)**.** For a π-destructor d, we define d' by $d'(\underline{t}) := d(\underline{t}\rho)\rho^{-1}$ where ρ is any injective map from the nonces occurring in the CoSP-terms $\underline{t}$ to names.[12] Let $\mathbf{N}_E$ and $\mathbf{N}_P$ be countably infinite sets.

The *symbolic model of the applied π-calculus* is given by $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$, where $\mathbf{N} := \mathbf{N}_E \cup \mathbf{N}_P$, $\mathbf{C} := \mathbf{C}_\pi$, $\mathbf{D} := \{d' : d \in \mathbf{D}_\pi\}$, and where $\mathbf{T}$ consists of all terms over $\mathbf{C}$ and $\mathbf{N}$, and where $\vdash$ is defined by the rules in Figure 4.

We fix $\mathbf{M}, \mathbf{C}, \mathbf{N}, \mathbf{D}, \vdash$ as in Definition 22. The destructor *equals'* induces an equivalence relation $\cong$ on the set of CoSP-terms with $x \cong y$ iff $\mathit{equals}'(x, y) \neq \bot$. The relation $\cong$ is the analogue to the equivalence relation $\approx$ describing the equational theory of the applied π-calculus.

Definition 23 (Computational implementation of Def. 22)**.** The computational implementation A of the symbolic model $\mathbf{M}$ of the applied π-calculus is given by $A_f := A_f^\pi$ for all $f \in \mathbf{C}$ and $A_d := A_d^\pi$ for all $d \in \mathbf{D}$. A_N for $N \in \mathbf{N}$ picks $r \in \mathrm{Nonces}_k$ uniformly at random and returns r.

The following definition of the symbolic execution is a technical tool, designed to relate the symbolic and the computational semantics. It is defined analogously to the computational execution while constituting a safe approximation of the original semantics.

Note that in the following definition we have $\mathrm{eval}^{\mathsf{CoSP}}\, M\eta\mu = M\eta\mu$ for π-terms M that are no destructor terms. To show the analogy to Definition 20 the redundant application is not removed.

Definition 24 (Symbolic execution of a π-process)**.** Let P_0 be a closed process, and let C be an interactive machine called the adversary. We define the *symbolic π-execution* as an interactive machine SExec_{P_0} that interacts with C:

[12] This is well-defined and independent of ρ since for any renaming of names τ, we have $d(\underline{M}\tau) = d(\underline{M})\tau$; intuitively d' behaves as d except that it uses nonces instead of names.

- *Start*: Let $P := P_0$ (where we rename all bound variables and names such that they are pairwise distinct and distinct from all unbound ones). Let η be a totally undefined partial function mapping variables to **terms**, let μ be a totally undefined partial function mapping names to **terms**. Let $a_1, \ldots, a_n$ denote the free names in P_0. For each i, **choose a different** $\boldsymbol{r_i \in \mathbf{N}_P}$ Set $\mu := \mu(a_1 := r_1, \ldots, a_n := r_n)$. Send $(r_1, \ldots, r_n)$ to C.
- *Main loop:* Send P to the adversary and expect an evaluation context E from the adversary. Distinguish the following cases:
  - $P = E[M(x).P_1]$: Request two **CoSP-terms** c, m from the adversary. If $\boldsymbol{c \cong \mathrm{eval}^{\mathsf{CoSP}}(M\eta\mu)}$, set $\eta := \eta(x := m)$ and $P := E[P_1]$.
  - $P = E[\nu a.P_1]$: **Choose** $\boldsymbol{r \in \mathbf{N}_P \setminus \mathrm{range}\mu}$, set $P := E[P_1]$ and $\mu := \mu(a := r)$.
  - $P = E[\overline{M_1}\langle N\rangle..P_1][M_2(x)..P_2]$: If $\boldsymbol{\mathrm{eval}^{\mathsf{CoSP}}(M_1)\eta\mu \cong \mathrm{eval}^{\mathsf{CoSP}}(M_2\eta\mu)}$ then set $P := E[P_1][P_2]$ and $\eta := \eta(x := \boldsymbol{\mathrm{eval}^{\mathsf{CoSP}}(N\eta\mu)})$.
  - $P = E[\text{Let } x = D \text{ in } P_1 \text{ else } P_2]$: If $m := \boldsymbol{\mathrm{eval}^{\mathsf{CoSP}}(D\eta\mu)} \neq \bot$, set $\eta := \eta(x := m)$ and $P := E[P_1]$. Otherwise set $P := E[P_2]$.
  - $P = E[event(e).P_1]$: Let $P := E[P_1]$ and raise the event e.
  - $P = E[!P_1]$: Rename all bound variables of P_1 such that they are pairwise distinct and distinct from all variables and names in P and in the domains of η and μ, yielding a process $\tilde{P}_1$. Set $P := E[\tilde{P}_1 \mid !P_1]$.
  - $P = E[\overline{M}\langle N\rangle..P_1]$: Request a **CoSP-term** c from the adversary.
    If $\boldsymbol{c \cong \mathrm{eval}^{\mathsf{CoSP}}(M\eta\mu)}$, set $P := E[P_1]$ and send $\boldsymbol{\mathrm{eval}^{\mathsf{CoSP}}(N\eta\mu)}$ to the adversary.
  - In all other cases, do nothing.

Defined like this, the symbolic π-execution is quite similar to the computational π-execution (Definition 20). The only differences are:

- The symbolic execution uses CoSP-terms instead of bitstrings as messages.
- The symbolic execution compares its messages using $\cong$ instead of checking for bitstring equality.
- The symbolic execution computes $\mathrm{eval}^{\mathsf{CoSP}}\, X\eta\mu$ instead of $\mathrm{ceval}_{\eta,\mu} X$.
- As value for a restricted name, the symbolic execution chooses a fresh CoSP-nonce instead of a random bitstring.

The interactive machine SExec_{P_0} performs only basic operations on CoSP-terms: the application of CoSP-constructors (including nonces) and CoSP destructors [13] , sending and receiving terms. Thus we can realise the machine as a CoSP protocol in the sense of Definition 4. For sending P to the adversary and receiving an evaluation context E, we use control nodes. If an event e is raised, we model this by sending $(event, e)$ to the adversary, again using a control node (but with only one successor). We call these nodes event nodes, and given a sequence of nodes $\underline{\nu}$, by $events(\underline{\nu})$ we denote the events e raised by the event nodes in $\underline{\nu}$.

[13] This includes comparing terms using $\cong$, which can be done by applying the destructor *equals'*.

For an interactive machine SExec_{P_0}, we call the protocol resulting from applying this construction Π_{P_0}. Since Π_{P_0} does not contain nondeterministic nodes, it is a CoSP protocol and a probabilistic protocol simultaneously. For the technical details of the construction we recommend having a look at [1].

Definition 25. A nondeterministic interactive machine C is a *Dolev-Yao adversary* if the following holds in an interaction with any interactive machine M in each step of the interaction: Let S be the set of all CoSP-terms sent by M up to the current step. Let m be the term sent by C in the current step. Then $S \vdash m$.

SExec_{P_0} satisfies a π-trace property $\wp$ if in a finite interaction with any Dolev-Yao adversary, the sequence of events raised by SExec_{P_0} is contained in $\wp$.

The following lemmas that originated in the CoSP paper will be needed for our result but we omit the proofs here. Since we did not change the applied π-calculus, we can apply the lemmas as they are. If you are interested in the proofs, you can find them in [1].

Lemma 9. *Let $\wp$ be a trace property. Then SExec_{P_0} satisfies $\wp$ iff Π_{P_0} symbolically satisfies $events^{-1}(\wp)$ (in the sense of Definition 10). Moreover, P_0 computationally satisfies $\wp$ iff (Π_{P_0}, A) computationally satisfies $events^{-1}(\wp)$ (in the sense of Definition 10).*

Lemma 10. *The probabilistic CoSP protocol Π_{P_0} is efficient.*

Lemma 11. *If a closed process P_0 symbolically satisfies a π-trace property $\wp$, then SExec_{P_0} satisfies $\wp$.*

The previous lemmas constitute a basis for showing the computational soundness of the calculus. For a trace property $\wp$, a closed process P and a computationally sound implementation A of the calculus, we can show: If P symbolically satisfies $\wp$, then by Lemma 11 the corresponding SExec_P also satisfies $\wp$. By Lemma 9 it follows that Π_P symbolically satisfies $events^{-1}(\wp)$. By Lemma 10 Π_P is efficient. If we now could apply Theorem 2 to show that Π_P, together with A, computationally satisfies $events^{-1}(\wp)$, we could apply Lemma 9 again to show that P computationally satisfies $\wp$.

Unfortunately, we can only apply Theorem 2 when the CoSP protocol fulfills all of the protocol conditions listed in Appendix 11.2, especially our new condition of garbage free decryption in the sense of Definition 17.

In the following, we specify what condition the π process P has to fulfill. Also we show that when the process fulfills this condition, then the coresponding CoSP protocol Π_P has garbage free decryption in the sense of Defintition 17.

8.4 A related protocol

For a π-process P we create a new protocol P_{gfd} that is a copy of P but whenever in P the destructor *dec* is applied to a *garbageEnc*-term, in P_{gfd} the event *badDec* is generated. Basically, every decryption term $dec(c)$ is replaced by *If isGarbageEnc(c) then event badDec Else dec (c)*. We then require the following condition to hold in order to apply our results:
Additional condition: *The process P_{gfd} has to satisfy the trace property φ_{gfd} (as defined below), stating that the event badDec does not occur.*

Our condition for the process might seem a bit strange, since it acutally does not speak about our original process, but about a related one. We will however show that in practice the condition can be checked easily and that this process can even be automatised (see Section 9 for more details).

Formally we state that P_{gfd} is a π-protocol related to P by a new relation R, defined as follows: Two states $s = (P, \eta, \mu)$ and $s' = (P', \eta', \mu')$ are related by R (we also write $s \approx^R s'$ or $(s, s') \in R$) iff $\mu = \mu'$, $\eta \subset \eta' \circ \pi$ for a variable renaming π and $(P, P') \in R^P$ (we also write $P \approx^P P'$), where R^P is the following relation on processes: $(P, P') \in R^P$ holds iff one of the following cases is met

- $P = M(x).P_1$
  $P' = M(x).P_1'$
  and $(P_1, P_1') \in R^P$
- $P = \nu a.P_1$
  $P' = \nu a.P_1'$
  and $(P_1, P_1') \in R^P$
- $P = \overline{M_1}\langle N\rangle.P_1$
  $P' = \overline{M_1}\langle N\rangle.P_1'$
  and $(P_1, P_1') \in R^P$
- $P = M_1\langle x\rangle.P_1$
  $P' = M_1\langle x\rangle.P_1'$
  and $(P_1, P_1') \in R^P$
- $P =$ let $x = dec(x_1, D)$ in P_1 else P_2
  $P' =$ let $y = isGarbageEnc(D)$ in event $badDec.L$ else L
  with $L =$ let $\pi(x) = dec(x_1, D)$ in P_1' else P_2'
  and $(P_1, P_1') \in R^P$, $(P_2, P_2') \in R^P$, $y \notin \text{domain}(\eta)$.
- let $x = D$ in P_1 else P_2
  $P' =$ let $\pi(x) = D$ in P_1' else P_2'
  and $(P_1, P_1') \in R^P$, $(P_2, P_2') \in R^P$, D not of the form $dec(\ldots)$.

- $P = P_1|P_2$
  $P' = P'_1|P'_2$
  and $(P_1, P'_1) \in R^P$, $(P_2, P'_2) \in R^P$.
- $P = \text{event}(e).P_1$
  $P' = \text{event}(e).P'_1$
  and $(P_1, P'_1) \in R^P$
- $P = !P_1$
  $P' = !P'_1$
  and $(P_1, P'_1) \in R^P$
- $P = \overline{M}\langle N\rangle.P_1$
  $P' = \overline{M}\langle N\rangle.P'_1$
  and $(P_1, P'_1) \in R^P$
- P is not of the form "let $x = D$ in P_1 else P_2" and $P = P'$.

Note that for every evaluation context E and for processes P_1, P'_1 with $(P_1, P'_1) \in R^P$ and for $E'[\_]$ with $(E[\_], E'[\_]) \in R^P$ it holds that $(E[P_1], E'[P'_1]) \in R^P$.

Definition 26. The trace property φ_{gfd} is the set of all lists of events that do not contain the event *badDec*.

Definition 27. A machine $SExec_P$ has GFD if it has no *bad* state in the following sense:
A state (P, η, μ) is called *bad* if $P = E[let\ x = dec(x_1, D)\ in\ P_1]$ with $eval_\eta(D)$ is of the form $gE(\ldots)$

Lemma 12. *P_{gfd} symbolically satisfies $\varphi_{\text{gfd}} \Rightarrow SExec_{P_{\text{gfd}}}$ satisfies φ_{gfd}.*

Proof. This follows directly by applying Lemma 11. □

8.5 Transfering GFD-ness

In order to apply Theorem 2, we need that Π_P has garbage free decryption in the sense of Definition 17. Intuitively this follows from the fact that P_{gfd} symbolically satisfies φ_{gfd}: We have defined that whenever a garbage decryption would occur in P, P_{gfd} raises the event *badDec*. If we assume that this event was not raised, no garbage decryptions can occur in the original protocol. If there is no garbage decryption in P, there should be no garbage decryption in Π_P. The proof however is a bit more complicated.

Lemma 13. *$SExec_{P_{\text{gfd}}}$ satisfies $\varphi_{\text{gfd}} \Rightarrow SExec_P$ has GFD.*

Proof. To show this we assume that there is an execution of $SExec_P$ in interaction with a DY adversary E that contradicts the GFD property. We then show that this leads to the existence of a execution of $SExec_{P_{\text{gfd}}}$ in interaction with a DY adversary E_g where the event *badDec* is raised.

First we connect the states of $SExec_P$ and $SExec_{P_{\text{gfd}}}$ in the following way:

Let $s' = (P', \eta', \mu')$ be the state of $SExec_P$, initially being the state before the first iteration but after the first message $(r_1, \ldots, r_n)$ has been sent. Analogously let $s'_g = (P'_g, \eta'_g, \mu'_g)$ be the state of $SExec_{P_{\text{gfd}}}$, also initially being the state before the first iteration but after the first message has been sent.

Since by changing P to P_{gfd} no names are changed, dropped or additionally introduced (see Section 8.4), the free names in P are also exactly the free names in P_{gfd}. Thus we can choose the same first message $(r_1, \ldots, r_n)$ for $SExec_{P_{\text{gfd}}}$.

We now use the relation from Section 8.4 that links the states of $SExec_P$ to states of $SExec_{P_{\text{gfd}}}$. Observe that, like it was intended, now $s' \approx^R s'_g$ holds with s' being the state of $SExec_P$ and s'_g being the state of $SExec_{P_{\text{gfd}}}$:

- At this point, $P'_g = P_{\text{gfd}}$ and $P' = P$. Since P_{gfd} is derived from P as described in Section 8.4, $P \approx^P P_{\text{gfd}}$ holds by definition of P_{gfd} and thus $P' \approx^P P'_g$.
- $\eta' = \eta'_g$ since both are totally undefined partial functions from variables to terms. Thus $\eta' \subset \eta'_g \circ \pi$ holds for arbitrary π.
- By definition of the symbolic execution, $\mu' = \mu_0(a_1 := r_1, \ldots, a_n := r_n)$ where μ_0 is totally undefined, $a_1, \ldots a_n$ are the free names in P and $(r_1, \ldots, r_n)$ is the first message sent to the adversary in $SExec_P$.

  Since, as stated above, $a_1, \ldots a_n$ also are the free names in P_{gfd} and $(r_1, \ldots, r_n)$ also is the first message sent to the adversary in $SExec_{P_{\text{gfd}}}$,
  $\mu'_g = \mu_0(a_1 := r_1, \ldots, a_n := r_n) = \mu'$.

Showing the following three claims will conclude the proof:

Claim 1: For all states $s = (P, \eta, \mu)$ with $P = E'[\text{let } y = isGarbageEnc(D) \text{ in } event\ badDec.P_1 \text{ else } P_1]$ where E' is given by the adversary and $eval_\eta(D)$ is of type $gE(\ldots)$, there is a state s^* s.t. $s \longrightarrow \stackrel{badDec}{\longrightarrow} s^*$ is possible in interaction with some adversary E_g.

By applying the rules from Definition 24 we see that in the next iteration of the main loop, $s \longrightarrow s'$ for some $s' = (P', \eta', \mu')$ with $P' = E'[event\ badDec.P_1]$. When, in the next iteration, the adversary sends the same E' again, we have that by applying the same rules $s' \stackrel{badDec}{\longrightarrow} s^*$ for some s^*.

Claim 2: $s' \approx^R t' \wedge s' \longrightarrow s'' \Rightarrow \exists t'' : s'' \approx^R t'' \wedge t' \longrightarrow^* t''$

Let there be an iteration step in $SExec_P$ that transforms the state from $s' = (P', \eta', \mu')$ to $s'' = (P'', \eta'', \mu'')$ and a state $t' = (P'_t, \eta'_t, \mu'_t)$ with $s' \approx^R t'$.

In $SExec_P$, P' is sent to the adversary that answers with an evaluation context E. For $SExec_{P_{\text{gfd}}}$ we choose an evaluation context E_t with $E \approx^R E_t$. This is possible because $P' \approx^P P'_t$. So for every context E s.t. $P' = E[P_1]$ we will find a context E_t with $P'_t = E_t[P_{t1}]$, $E \approx^P E_t$ and $P_1 \approx^P P_{t1}$. We now distunguish the following cases depending on P':

- $P' = E[\text{let } x = dec(x_1, D) \text{ in } P_1 \text{ else } P_2]$ with $eval_\eta(D)$ not of type $gE(\ldots)$

  Since $P' \approx^P P'_t$ and $E \approx^P E_t$, P'_t must be of the form $E_t[$ let $y = isGarbageEnc(D)$ in event $badDec.L$ else $L]$ for $L =$ let $x_t = dec(x_1, D)$ in P_{t1} else P_{t2}.

  We also know that $P_1 \approx^P P_{t1}$ and $P_2 \approx^P P_{t2}$ since this is necessary for $P' \approx^P P'_t$.

  By applying the rules from Definition 24 we have that $s'_t \longrightarrow s^*_t = (P^*_t, \eta^*_t, \mu^*_t)$ with $P^*_t = E'[L]$, $\eta^*_t = \eta'_t$, $\mu^*_t = \mu'_t$.

  Now, depending on $dec(x_1, D)$, both machines make a step:

  - If $l := eval^{COSP}(dec(x_1, D)) \neq \bot$
    $P^*_t = E'[P_{t1}]$, $P'' = E[P_1]$,
    We know that $\eta' = \eta'_t \circ \pi$ for a variable renaming π. For $\pi' := \pi[x \to x_t]$ we have that $\eta'' = \eta'(x := l) = \eta'_t(x_t := l) \circ \pi' = \eta^*_t \circ \pi'$. Thus there again exists a variable renaming π' such that $\eta'' = \eta^* \circ \pi'$.
    $\mu'' = \mu' = \mu'_t = \mu^*_t$.
    Thus $s'' \approx^R s^*_t$.
  - If $eval^{COSP}(dec(x_1, D)) = \bot$
    $P^*_t = E'[P_{t2}]$, $P'' = E[P_2]$.
    $\eta'' = \eta' \overset{(1)}{=} \eta'_t \circ \pi = \eta^*_t \circ \pi$, where (1) follows from $s' \approx^R t'$ for some variable renaming π.
    $\mu'' = \mu' = \mu'_t = \mu^*_t$.
    Thus $s'' \approx^R s^*_t$.

  Thus $s'_t \longrightarrow^* s^*_t$ with $s'' \approx^R s^*_t$.

- $P' = E[\text{let } x = dec(x_1, D) \text{ in } P_1 \text{ else } P_2]$ with $eval_\eta(D)$ of type $gE(\ldots)$

  By Claim 1 we have that
  $s'_t \longrightarrow \overset{badDec}{\longrightarrow} s^*_t = (P^*_t, \eta^*_t, \mu^*_t)$ with P^*_t, η^*_t, μ^*_t as in the previous case.

  We now proceed as above.

- The other cases follow directly from the definition of R^P.

Claim 3:For an evaluation context E and Processes P' and P'_t with $P' \approx^P P'_t$ and $P' = E[P'']$ we can create another evaluation context E_g with $P'_t = E_g[P''_t]$ such that $P'' \approx^P P''_t$.

The only differences of the protocols lies in variable names and the extension of *Let* constructs with decryption terms. Traversing down the structure of both P' and P'_t until reaching P'', seeing that for the relation to hold the subterms have to be related as well, the claim follows immediately with P''_t being the pendant to P'' and E_g being the rest.

Claim 4: $s' = (P', \eta', \mu')$ *bad* $\Rightarrow \forall s'_t = (P'_t, \eta'_t, \mu'_t)$ with $s' \approx^R s'_t$ there is a process P^* and an evaluation context E_g such that $P'_t = E_g[\text{let } y = isGarbageEnc(D) \text{ in event } badDec.P^* \text{ else } P^*]$ with $eval_\eta(D)$ is of the form $gE(\ldots)$.

Let $s'_t = (P'_t, \eta'_t, \mu'_t)$ be a state with $s' \approx^R s'_t$. By Definition 27 $s' = (P', \eta', \mu')$ is *bad* implies that $P' = E[let\ x = dec(x_1, D)\ in\ P_1 else\ P_2]$ with $eval'_\eta(D)$ is of the form $gE(\ldots)$ for some evaluation context E.

Since $s' \approx^R s'_t$ we know that $P' \approx^P P'_t$. For an evaluation context E with $P' = E[P'']$ by Claim 3 we know that we can create another evaluation context E_g with $P'_t = E_g[P''_t]$ such that $P'' \approx^P P''_t$. But since in our case, for the chosen E, the inner process P'' is of the form let $x = dec(x_1, D)$ in P_1 else P_2, by the definition of R^P (see Section 8.4) we know that P''_t must be of the form let $y = isGarbageEnc(D)$ in event $badDec.P^*$ else P^* for some P^*.

□

Lemma 14. *$SExec_P$ has GFD $\Rightarrow \Pi_P$ has GFD.*

Proof. This follows directly from the construction of Π_P. When in Π_P a *dec* destructor node is reached with the second argument being of type *garbageEnc*, then in $SExec_P$ there is a *bad* state (in the sense of Definition 27) that is also reachable. □

This Lemma concludes the proof. We can now show the computational soundness of the calculus:

Theorem 3. ***Computational soundness with IND-CPA in the applied π-calculus*** *Assume that the computational implementation A of the applied π-calculus (Definition 23) is a computationally sound implementation (in the sense of Definition 11) of the symbolic model of the applied π-calculus (Definition 22) for a class $\mathbf{P}$ of protocols.*

If a closed process P symbolically satisfies a π-trace property $\wp$, $\Pi_P \in \mathbf{P}$, and the related process P_{gfd} with $P \approx^P P_{\text{gfd}}$ symbolically satisfies φ_{gfd} then P computationally satisfies $\wp$.

Proof. Assume that P symbolically satisfies $\wp$. Then by Lemma 11 the corresponding SExec_P also satisfies $\wp$. By Lemma 9 it follows that Π_P symbolically satisfies $events^{-1}(\wp)$. By Lemma 10 Π_P is efficient. Since $\wp$ is an efficient decidable, prefix closed set, so is $events^{-1}(\wp)$. Thus, $events^{-1}(\wp)$ is a CoSP trace property in the sense of Definition 10. Since by assumption P_{gfd} symbocically satisfies φ_{gfd}, by Lemma 12 we have that $SExec_{P_{\text{gfd}}}$ satisfies φ_{gfd}. Thus, by Lemma 13 $SExec_P$ has GFD in the sense of Definition 27. By Lemma 14 it follows that Π_P has GFD.

Together with the assumption that the computational implementation A of the applied π-calculus is computationally sound, we can apply Theorem 2 and yield that (Π_P, A) computationally satisfies $events^{-1}(\wp)$. We apply Lemma 9 again to show that P computationally satisfies $\wp$.

□

9 Case Study

In the previous section, we have shown how the computational soundness result with IND-CPA can be used for the applied π-calculus. We stated that the required condition for our new symbolic property, the concept of garbage free decryption, can even be checked automatically.

In this section we will present an example of how to check for GFD and how to use ProVerif to automatise this.

9.1 ProVerif

ProVerif is a tool designed to automatically verify cryptographic protocols. It is based on Prolog rules. The definitions of functions like destructors are stated as facts. The protocol itself is written in a π-calculus alike syntax.

By writing queries the user can ask ProVerif to check if certain properties are fulfilled. Some of the possible queries include:

- Checking if an adversary is able to deduce some nonce n
- Checking if some event A is raised
- Checking if events are raised in a specific order. (e.g. event B is never raised before event A is raised)

ProVerif uses some abstractions and approximations but they are proven to be sound. If ProVerif states that it does not find a violation to the queries, there is none. ProVerif is not complete though. For more information about ProVerif, please have a look at [3].

In the CoSP paper it is shown that and how ProVerif can be used to check whether a certain property is fulfilled. We wanted to extend this by showing that also the fact that a protocol has symbolic GFD can be shown. This is a necessary criterion for a protocol in order to apply our computational soundness theorem. In Section 8 we have shown how this can be verified: A related protocol is created like described in Section 8.4. If it can be shown that the (new) event `badDec()` is never raised, the original protocol has GFD.

Our example directly follows this approach. In order to modify the protocol we make use of the preprocessor $m4$. In our protocol we use the pattern

```
DEC(a,b,c)
```

instead of the ProVerif line

```
let a = dec(b,c) in
```

When we want to check for the GFD property, $m4$ replaces the line by

```
let y = isgarbageenc(c) in event badDec() else let a = dec(b,c) in
```

where *isgarbageenc* checks, if the input is a garbage encryption term. Thus, whenever a garbage encryption would be decrypted, the event `badDec()` will be raised. Now when ProVerif can prove that the event is never raised, from Section 8 we know that the corresponding CoSP protocol will have GFD.

9.2 Some words about m4

We did not want to leave applying the changes from Section 8.4 by hand to the user. Thus we made use of the GNU $m4$ preprocessor [7] and defined a few simple rules for it, that are stated below. For the ProVerif file we simply require, that decryption is applied in the previously mentioned syntax (DEC(a,b,c)) and that the following lines are included:

```
GFDDEF
GFDQUERY
```

The GNU $m4$ preprocessor is a macro processor that copies its input to the output while expanding both built in and user defined macros. We barely use the power that comes with the many features of $m4$, using it to simply replace a few words by other words. If interested in the features, we recommend having a look at the extensive manual (see [7]).

As mentioned before, here you will se the $m4$ definitions:

Our definitions follow a very simple pattern.

```
define(f, this is a text including $1 and $2)
```

defines a function symbol f. Every occurence of f is replaced by the text that follows the comma. The symbols \$1 and \$2 are replaced by the parameters of the function symbol. The text *f(cats,dogs)* would be replaced by *this is a text including cats and dogs.*

First, you will se our $m4$ definition file for the normal check. The resulting file includes the user defined queries that allow to check for the security properties, but it excludes the GFD check.

```
define(DEC,let $1 = dec($2,$3) in)

define(GFDQUERY,)
define(GFDDEF,)
define(QUERY,query $1.)
```

As you can see, in this "normal" setting, we replace the `DEC` symbol by a normal decryption. The placeholders for the GFD query and for the definition of the garbage check are removed from the file. The normal, user defined queries stay in the file and are converted to the ProVerif syntax.

In our GFD check definition we exclude the user defined queries that allow to check for the security properties. Instead we include the check for GFD and the additional definition for checking if a term is a garbage encryption.

```
define(DEC,let y = isgarbageenc($3) in event badDec() else let $1 = dec($2,$3) in)

define(GFDQUERY,query ev:badDec().)
define(GFDDEF,reduc isgarbageenc(garbageenc(r1,r2)) = garbageenc(r1,r2).)
define(QUERY,)
```

Here, whenever a decryption occurs, we want to check if it would be a garbage encryption first. Thus, we replace the `DEC` symbol by a garbage encryption check, followed by the actual decryption.

Additionally the definition for the garbage encryption check is given and a new query is included. It simply checks if the event `badDec` can be send. All other queries are removed.

When applying $m4$ to the example lines listed in Figure 5, we will see some differences.

```
GFDDEF

QUERY(evinj:end() ==> evinj:begin())
GFDQUERY

[...]

let e1 = verify(vkA,s1) in
DEC(nA',dkB,e1)
new r1; new r2;
```

Figure 5: original file "replay2.pv"

When using `m4defGFD`, the resulting file will look like shown in Figure 6.
When using `m4defNORMAL` instead, we will get a file as shown in Figure 7.
As you can see, in the first proverif file we can check if the GFD property is fulfilled, while the second file allows to check for the user defined properties.

9.3 The example protocol

Our example protocol models a communication between Alice and her bank. She wants her bank to execute a transaction for her (for example transfering money from her bank account to another one). We assume that both she and the bank already have exchanged public keys for encryptions and verification keys for signatures, via some secret channel but now are communicating using an insecure channel.

```
reduc isgarbageenc(garbageenc(r1,r2)) = garbageenc(r1,r2).


query ev:badDec().

[...]

let e1 = verify(vkA,s1) in
let y = isgarbageenc(e1) in event badDec() else let nA' = dec(dkB,e1) in
new r1; new r2;
```

Figure 6: gfd check file "replay2_gfd.pv"

```
query evinj:end() ==> evinj:begin().


[...]

let e1 = verify(vkA,s1) in
let nA' = dec(dkB,e1) in
new r1; new r2;
```

Figure 7: normal check file "replay2_normal.pv"

We want our protocol to not only be secure against an eavesdropping adversary, or an adversary that changes the messages directly, but also against so called *replay attacks*. In our setting, a replay attack occurs, if an adversary can make the bank execute a transaction more often than Alice requested it. Lets assume that Alice wants to transfer 50$ from her account to the account of Eve. If by copying some messages and sending them to the bank over and over again, Eve can make the bank transfer the 50$ not once but 10 times, she made a replay attack.

Our protocol is modelled like this:

$$A \xrightarrow{sig(sigk_A,(A,enc(pk_B,N_A)))} B$$
$$A \xleftarrow{sig(sigk_B,(B,enc(pk_B,(N_A,N_B))))} B$$
$$A \xrightarrow{sig(sigk_A,(enc(pk_B,(N_B,m))))} B$$

Alice sends to the bank a new nonce N_A, encrypted with the public key of the bank, and signs the message with her signing key $sigk_A$. The bank verifies the signature and then decrypts the nonce. Then it picks a new nonce N_B and sends to Alice an encryption of both nonces, signed with its signing key $sigk_B$ Upon receiving this, Alice first verifies the signature of the message and then decrypts it. If the result of the decryption is a tuple with her own nonce (N_A) and another one (N_B), she sends an encryption of N_B and her actual transaction message (e.g. "Transfer 50$ to Eve") to the bank, again signed with $sigk_A$. The bank, upon receiving the message, again verifies the signature

and then decrypts. If N_B matches the nonce of the current transaction, it executes the transaction.

Since both Alice and the bank pick a new nonce and check whether this nonce is used correctly, we claim that a replay attack is impossible. We prove our claim using ProVerif and our $m4$ macros. We model our protocol in the ProVerif syntax. The full protocol description can be found in Appendix 11.4.1. In order to check for our protocol to be secure against replay attacks, we do the following:

We allow the adversary to specify for which message $textm$ he wants to host a replay attack. Whenever Alice starts the protocol, we allow the adversary to even specify which transaction message she picks. Additionally, the adversary can chose how often and in which order the protocol members are invoked (He could for example let Alice start the protocol twice and combine the messages he sees in some way). This is an overapproximation of the adversaries capabilities.

When Alice starts the protocol and her transaction message is $textm$, she raises the event `begin()`. When the bank executes the transaction $textm$, it raises the event `end()`.

The idea is the following one: If a replay attack is possible, then it will be possible that the event `end()` is raised more often than the event `begin()`. We ask ProVerif to check if this can happen be checking the query
`evinj:end() ==> evinj:begin().`
This query states that for every event `end()` there has to exist a distinct event `begin()` that was raised before.

In Appendix 11.4 you can find the full description of a protocol and the query in ProVerif syntax, where the placeholders are at the right places. Additionally you can find the files that are produced by $m4$.

When invoking ProVerif with the "gfd check" file (see Appendix 11.4.2), the output looks as follows:

```
-- Query not ev:badDec()
Completing...
Starting query not ev:badDec()
RESULT not ev:badDec() is true.
```

Proverif states that the event `badDec()` is never raised. By Lemmas 12, 13 and 14 we know that the CoSP protocol, corresponding to our original protocol has garbage free decryption. Thus we can safely analyse it.

When invoking ProVerif with the "normal" file (see Appendix 11.4.4), the output looks as follows:

```
-- Query evinj:end() ==> evinj:begin()
Completing...
Starting query evinj:end() ==> evinj:begin()
RESULT evinj:end() ==> evinj:begin() is true.
```

ProVerif states that our query holds. Thus, the event `end()` can not be raised more often than the event `begin()`. A replay attack is impossible.

We have shown now that our protocol is secure against the replay attack. By Theorem 3 we know that a computational implementation, satisfying all implementation conditions, will also be secure against the replay attack. Following this example we can analyse arbitrary protocols that satisfy our requirements.

10 Conclusion and Future Work

We have shown how to change the requirements for protocols in order to prove a symbolic model computationally sound using the CoSP framework when requiring the implementation of the encryption scheme to be IND-CPA secure. We have presented the simple condition of *garbage free decryption* and shown how to check if a protocol fulfills this property. We have presented a way to automatically check if a protocol in the applied π-calculus has *garbage free decryption* using ProVerif.

We plan to investigate which security flaws might arise from the fact that the GFD property was violated at a certain point in the protocol. A goal might be to expand the overall result from stating "We cant prove anything, since the GFD property is not fulfilled" to something like "We cant prove that nonce N is secure after point x".

We also plan to have a look at the computational soundness of zero knowledge proofs in combination with IND-CPA secure encryption schemes.

11 Appendix

11.1 Implementation Conditions

We require that the implementation A of the symbolic model $\mathbf{M}$ has the following properties:

1. A is an implementation of $\mathbf{M}$ in the sense of 8 (in particular, all functions A_f ($f \in \mathbf{C} \cup \mathbf{D}$) are polynomial-time computable).
2. There are disjoint and efficiently recognizable sets of bitstrings representing the types nonces, ciphertexts, encryption keys, decryption keys, signatures, verification keys, signing keys, pairs, and payload-strings. The set of all bitstrings of type nonce we denote Nonces_k.[14] (Here and in the following, k denotes the security parameter.)
3. The functions A_{enc}, A_{ek}, A_{dk}, A_{sig}, A_{vk}, A_{sk}, A_{pair}, A_{string_0}, and A_{string_1} are length-regular. All $m \in \text{Nonces}_k$ have the same length.
4. A_N for $N \in \mathbf{N}$ returns a uniformly random $r \in \text{Nonces}_k$.
5. Every image of A_{enc} is of type ciphertext, every image of A_{ek} and A_{ekof} is of type encryption key, every image of A_{dk} is of type decryption key, every image of A_{sig} is of type signature, every image of A_{vk} and A_{vkof} is of type verification key, every image of A_{empty}, A_{string_0}, and A_{string_1} is of type payload-string.
6. For all $m_1, m_2 \in \{0,1\}^*$ we have $A_{fst}(A_{pair}(m_1, m_2)) = m_1$ and $A_{snd}(A_{pair}(m_1, m_2)) = m_2$. Every m of type pair is in the range of A_{pair}. If m is not of type pair, $A_{fst}(m) = A_{snd}(m) = \bot$.
7. For all m of type payload-string we have that $A_{unstring_i}(A_{string_i}(m)) = m$ and $A_{unstring_i}(A_{string_j}(m)) = \bot$ for $i, j \in \{0,1\}$, $i \neq j$. For $m = empty$ or m not of type payload-string, $A_{unstring_0}(m) = A_{unstring_1}(m) = \bot$. Every m of type payload-string is of the form $m = A_{unstring_0}(m')$ or $m = A_{unstring_1}(m')$ or $m = empty$ for some m' of type payload-string.
8. $A_{ekof}(A_{enc}(p, x, y)) = p$ for all p of type encryption key, $x \in \{0,1\}^*$, $y \in \text{Nonces}_k$. $A_{ekof}(e) \neq \bot$ for any e of type ciphertext and $A_{ekof}(e) = \bot$ for any e that is not of type ciphertext.
9. $A_{vkof}(A_{sig}(A_{sk}(x), y, z)) = A_{vk}(x)$ for all $y \in \{0,1\}^*$, $x, z \in \text{Nonces}_k$. $A_{vkof}(e) \neq \bot$ for any e of type signature and $A_{vkof}(e) = \bot$ for any e that is not of type signature.
10. $A_{enc}(p, m, y) = \bot$ if p is not of type encryption key.
11. $A_{dec}(A_{dk}(r), m) = \bot$ if $r \in \text{Nonces}_k$ and $A_{ekof}(m) \neq A_{ek}(r)$. (This implies that the encryption key is uniquely determined by the decryption key.)
12. $A_{dec}(A_{dk}(r), A_E(A_{ek}(r), m, r')) = m$ for all $r, r' \in \text{Nonces}_k$.
13. $A_{verify}(A_{vk}(r), A_{sig}(A_{sk}(r), m, r')) = m$ for all $r, r' \in \text{Nonces}_k$.
14. For all $p, s \in \{0,1\}^*$ we have that $A_{verify}(p, s) \neq \bot$ implies $A_{vkof}(s) = p$.
15. $A_{isek}(x) = x$ for any x of type encryption key. $A_{isek}(x) = \bot$ for any x not of type encryption key.
16. $A_{isvk}(x) = x$ for any x of type verification key. $A_{isvk}(x) = \bot$ for any x not of type

[14] This would typically be the set of all k-bit strings with a tag denoting nonces.

verification key.

17. $A_{isenc}(x) = x$ for any x of type ciphertext. $A_{isenc}(x) = \bot$ for any x not of type ciphertext.
18. $A_{issig}(x) = x$ for any x of type signature. $A_{issig}(x) = \bot$ for any x not of type signature.
19. We define an encryption scheme $(\mathsf{KeyGen}, enc, dec)$ as follows: KeyGen picks a random $r \leftarrow \text{Nonces}_k$ and returns $(A_{ek}(r), A_{dk}(r))$. $enc(p, m)$ picks a random $r \leftarrow \text{Nonces}_k$ and returns $A_{enc}(p, m, r)$. $dec(k, c)$ returns $A_{dec}(k, c)$. We require that then $(\mathsf{KeyGen}, enc, dec)$ is IND-CPA secure.
20. We define a signature scheme $(\mathsf{SKeyGen}, \mathsf{Sig}, \mathsf{Verify})$ as follows: $\mathsf{SKeyGen}$ picks a random $r \leftarrow \text{Nonces}_k$ and returns $(A_{vk}(r), A_{sk}(r))$. $\mathsf{Sig}(p, m)$ picks a random $r \leftarrow \text{Nonces}_k$ and returns $A_{sig}(p, m, r)$. $\mathsf{Verify}(p, s, m)$ returns 1 iff $A_{verify}(p, s) = m$. We require that then $(\mathsf{SKeyGen}, \mathsf{Sig}, \mathsf{Verify})$ is strongly existentially unforgeable.
21. For all e of type encryption key and all $m \in \{0,1\}^*$, the probability that $A_{enc}(e, m, r) = A_{enc}(e, m, r')$ for uniformly chosen $r, r' \in \text{Nonces}_k$ is negligible.
22. For all $r_s \in \text{Nonces}_k$ and all $m \in \{0,1\}^*$, the probability that $A_{sig}(A_{sk}(r_s), m, r) = A_{sig}(A_{sk}(r_s), m, r')$ for uniformly chosen $r, r' \in \text{Nonces}_k$ is negligible.

11.2 Protocol Conditions

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 $N \in \mathbf{N}_P$. (Here and in the following, we call the nodes referenced by a protocol node its arguments.) We call these N-computation nodes *randomness nodes*. Any two randomness nodes on the same path are annotated with different nonces.
2. Every computation node that is the argument of an *ek*-computation node or of a *dk*-computation node on some path p occurs only as argument to *ek*- and *dk*-computation nodes on that path p.
3. Every computation node that is the argument of a *vk*-computation node or of an *sk*-computation node on some path p occurs only as argument to *vk*- and *sk*-computation nodes on that path p.
4. Every computation node that is the third argument of an *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 *garbage*, *garbageE*, *garbageSig*, or $N \in \mathbf{N}_E$.
10. The protocol has **garbage free decryption** in the sense of Definition 17.

11.3 Translation Functions

Translation functions. The partial function $\beta : \mathbf{T} \to \{0,1\}^*$ is defined as follows (where the first matching rule is taken):

- $\beta(N) := r_N$ if $N \in \mathcal{N}$.
- $\beta(N^m) := m$.
- $\beta(enc(ek(t_1), t_2, M)) := A_{enc}(\beta(ek(t_1)), \beta(t_2), r_M)$ if $M \in \mathcal{N}$.
- $\beta(ek(N)) := A_{ek}(r_N)$ if $N \in \mathcal{N}$.
- $\beta(ek(N^m)) := m$.
- $\beta(dk(N)) := A_{dk}(r_N)$ if $N \in \mathcal{N}$.
- $\beta(sig(sk(N), t, M)) := A_{sig}(A_{sk}(r_N), \beta(t), r_M)$ if $N, M \in \mathcal{N}$.
- $\beta(sig(sk(M), t, N^s)) := s$.
- $\beta(vk(N)) := A_{vk}(r_N)$ if $N \in \mathcal{N}$.
- $\beta(vk(N^m)) := m$.
- $\beta(sk(N)) := A_{sk}(r_N)$ if $N \in \mathcal{N}$.
- $\beta(pair(t_1, t_2)) := A_{pair}(\beta(t_1), \beta(t_2))$.
- $\beta(string_0(t)) := A_{string_0}(\beta(t))$.
- $\beta(string_1(t)) := A_{string_1}(\beta(t))$.
- $\beta(empty) := A_{empty}()$.
- $\beta(garbage(N^c)) := c$.
- $\beta(garbageE(t, N^c)) := c$.
- $\beta(garbageSig(t_1, t_2, N^s)) := s$.
- $\beta(t) := \perp$ in all other cases.

The total function $\tau : \{0,1\}^* \to \mathbf{T}$ is defined as follows (where the first matching rule is taken):

- $\tau(r) := N$ if $r = r_N$ for some $N \in \mathcal{N} \setminus \mathcal{R}$.
- $\tau(r) := N^r$ if r is of type nonce.
- $\tau(c) := enc(ek(M), t, N)$ if c has earlier been output by $\beta(enc(ek(M), t, N))$ for some $M \in \mathbf{N}$, $N \in \mathcal{N}$.

- $\tau(e) := ek(N)$ if e has earlier been output by $\beta(ek(N))$ for some $N \in \mathcal{N}$.
- $\tau(e) := ek(N^e)$ if e is of type encryption key.
- $\tau(s) := sig(sk(M), t, N)$ if s has earlier been output by $\beta(sig(sk(M), t, N))$ for some $M, N \in \mathcal{N}$.
- $\tau(s) := sig(sk(M), \tau(m), N^s)$ if s is of type signature and $\tau(A_{vkof}(s)) = vk(M)$ for some $M \in \mathbf{N}$ and $m := A_{verify}(A_{vkof}(s), s) \neq \bot$.
- $\tau(e) := vk(N)$ if e has earlier been output by $\beta(vk(N))$ for some $N \in \mathcal{N}$.
- $\tau(e) := vk(N^e)$ if e is of type verification key.
- $\tau(m) := pair(\tau(A_{fst}(m)), \tau(A_{snd}(m)))$ if m of type pair.
- $\tau(m) := string_0(m')$ if m is of type payload-string and $m' := A_{unstring_0}(m) \neq \bot$.
- $\tau(m) := string_1(m')$ if m is of type payload-string and $m' := A_{unstring_1}(m) \neq \bot$.
- $\tau(m) := empty$ if m is of type payload-string and $m = A_{empty}()$.
- $\tau(c) := garbageE(\tau(A_{ekof}(c)), N^c)$ if c is of type ciphertext.
- $\tau(s) := garbageSig(\tau(A_{vkof}(s)), N^s)$ if s is of type signature.
- $\tau(m) := garbage(N^m)$ otherwise.

The function $\ell : \mathbf{T} \to \{0,1\}^*$ is defined as $\ell(t) := |\beta(t)|$. Note that $\ell(t)$ does not depend on the actual values of r_N because of the length-regularity of A_E, A_{ek}, A_{dk}, A_{sig}, A_{vk}, A_{sk}, A_{pair}, A_{string_0}, and A_{string_1}. Hence $\ell(t)$ can be computed without accessing r_N.

11.4 Case Study Files

11.4.1 file "replay.pv"

```
free c.
free adv.

fun enc/3. fun ek/1. fun dk/1.
fun sig/3. fun vk/1. fun sk/1.

fun pair/2. fun garbage/1. fun garbageenc/2. fun garbagesig/2.

fun string0/1. fun string1/1. fun empty/0.

reduc fst(pair(x,y)) = x.
reduc snd(pair(x,y)) = y.
reduc equals(x,x) = x.
reduc unstring0(string0(s)) = s.
reduc unstring1(string1(s)) = s.

reduc dec(dk(r1),enc(ek(r1),y,r2)) = y.
```

```
reduc isenc(enc(ek(r1),m,r2)) = enc(ek(r1),m,r2);
  isenc(garbageenc(r1,r2)) = garbageenc(r1,r2).
reduc ekof(enc(ek(r1),m,r2)) = ek(r1);
  ekof(garbageenc(r1,r2)) = r1.
reduc isek(ek(r)) = ek(r).

reduc verify(vk(r1),sig(sk(r1),m,r2)) = m.
reduc issig(sig(sk(r1),m,r2)) = sig(sk(r1),m,r2);
  issig(garbagesig(r1,r2)) = garbagesig(r1,r2).
reduc vkof(sig(sk(r1),m,r2)) = vk(r1);
  vkof(garbagesig(r1,r2)) = r1.
reduc isvk(vk(r)) = vk(r).

GFDDEF

QUERY(evinj:end() ==> evinj:begin())

GFDQUERY

let A' =
out(c, sig(skA, enc(ekA, nA, r), r'));
in (c, s);
let e = verify(vkB, s) in
let p = dec(dkA, e) in
if nA = fst(p) then
let nB' = snd(p) in new r1; new r2;
let en = enc(ekB, pair(nB',m), r1) in
out(c, sig(skA, en, r2) ).

let A = new nA; new r; new r'; in(adv,m);
    if m = testm then event begin(); A' else A'.

let B = new nB; in (c, s1);
let e1 = verify(vkA,s1) in
DEC(nA',dkB,e1)
new r1; new r2;
let en = enc(ekA, pair(nA',nB), r1) in
let si = sig(skB, en, r2) in
out (c, si );
in (c, s2);
let e2 = verify(vkA,s2) in
DEC(p,dkB,e2)
if fst(p) = nB then
if snd(p) = testm then event end().

let C = event false().

process new rA;  let ekA = ek(rA)  in let dkA = dk(rA)  in out(c, ekA);
new rA'; let vkA = vk(rA') in let skA = sk(rA') in out(c, vkA);
new rB;  let ekB = ek(rB)  in let dkB = dk(rB)  in out(c, ekB);
new rB'; let vkB = vk(rB') in let skB = sk(rB') in out(c, vkB);
in(adv,testm);
(!A | !B)
```

11.4.2 file "replay_gfd.pv"

```
free c.
free adv.

fun enc/3. fun ek/1. fun dk/1.
fun sig/3. fun vk/1. fun sk/1.
```

```
fun pair/2. fun garbage/1. fun garbageenc/2. fun garbagesig/2.

fun string0/1. fun string1/1. fun empty/0.

reduc fst(pair(x,y)) = x.
reduc snd(pair(x,y)) = y.
reduc equals(x,x) = x.
reduc unstring0(string0(s)) = s.
reduc unstring1(string1(s)) = s.

reduc dec(dk(r1),enc(ek(r1),y,r2)) = y.
reduc isenc(enc(ek(r1),m,r2)) = enc(ek(r1),m,r2);
  isenc(garbageenc(r1,r2)) = garbageenc(r1,r2).
reduc ekof(enc(ek(r1),m,r2)) = ek(r1);
  ekof(garbageenc(r1,r2)) = r1.
reduc isek(ek(r)) = ek(r).

reduc verify(vk(r1),sig(sk(r1),m,r2)) = m.
reduc issig(sig(sk(r1),m,r2)) = sig(sk(r1),m,r2);
  issig(garbagesig(r1,r2)) = garbagesig(r1,r2).
reduc vkof(sig(sk(r1),m,r2)) = vk(r1);
  vkof(garbagesig(r1,r2)) = r1.
reduc isvk(vk(r)) = vk(r).

reduc isgarbageenc(garbageenc(r1,r2)) = garbageenc(r1,r2).



query ev:badDec().

let A' =
out(c, sig(skA, enc(ekA, nA, r), r'));
in (c, s);
let e = verify(vkB, s) in
let p = dec(dkA, e) in
if nA = fst(p) then
let nB' = snd(p) in new r1; new r2;
let en = enc(ekB, pair(nB',m), r1) in
out(c, sig(skA, en, r2) ).

let A = new nA; new r; new r'; in(adv,m);
    if m = testm then event begin(); A' else A'.

let B = new nB; in (c, s1);
let e1 = verify(vkA,s1) in
let y = isgarbageenc(e1) in event badDec() else let nA' = dec(dkB,e1) in
new r1; new r2;
let en = enc(ekA, pair(nA',nB), r1) in
let si = sig(skB, en, r2) in
out (c, si );
in (c, s2);
let e2 = verify(vkA,s2) in
let y = isgarbageenc(e2) in event badDec() else let p = dec(dkB,e2) in
if fst(p) = nB then
if snd(p) = testm then event end().

let C = event false().

process new rA;  let ekA = ek(rA)  in let dkA = dk(rA)  in out(c, ekA);
new rA'; let vkA = vk(rA') in let skA = sk(rA') in out(c, vkA);
new rB;  let ekB = ek(rB)  in let dkB = dk(rB)  in out(c, ekB);
```

```
new rB'; let vkB = vk(rB') in let skB = sk(rB') in out(c, vkB);
in(adv,testm);
(!A | !B)
```

11.4.3 ProVerif output for "replay_GFD.pv"

```
Process:
new rA_41;
{1}let ekA_42 = ek(rA_41) in
{2}let dkA_43 = dk(rA_41) in
{3}out(c, ekA_42);
new rA'_44;
{4}let vkA_45 = vk(rA'_44) in
{5}let skA_46 = sk(rA'_44) in
{6}out(c, vkA_45);
new rB_47;
{7}let ekB_48 = ek(rB_47) in
{8}let dkB_49 = dk(rB_47) in
{9}out(c, ekB_48);
new rB'_50;
{10}let vkB_51 = vk(rB'_50) in
{11}let skB_52 = sk(rB'_50) in
{12}out(c, vkB_51);
{13}in(adv, testm_53);
{14}!
(
new nA_67;
new r_68;
new r'_69;
{32}in(adv, m_70);
{33}if m_70 = testm_53 then
(
    {42}event begin();
    {43}out(c, sig(skA_46,enc(ekA_42,nA_67,r_68),r'_69));
    {44}in(c, s_78);
    {45}let e_79 = verify(vkB_51,s_78) in
    {46}let p_80 = dec(dkA_43,e_79) in
    {47}if nA_67 = fst(p_80) then
    {48}let nB'_81 = snd(p_80) in
    new r1_82;
    new r2_83;
    {49}let en_84 = enc(ekB_48,pair(nB'_81,m_70),r1_82) in
    {50}out(c, sig(skA_46,en_84,r2_83));
    0
)
else
(
    {34}out(c, sig(skA_46,enc(ekA_42,nA_67,r_68),r'_69));
    {35}in(c, s_71);
    {36}let e_72 = verify(vkB_51,s_71) in
    {37}let p_73 = dec(dkA_43,e_72) in
    {38}if nA_67 = fst(p_73) then
    {39}let nB'_74 = snd(p_73) in
    new r1_75;
    new r2_76;
    {40}let en_77 = enc(ekB_48,pair(nB'_74,m_70),r1_75) in
    {41}out(c, sig(skA_46,en_77,r2_76));
    0
)
) | (
{15}!
```

```
new nB_54;
{16}in(c, s1_55);
{17}let e1_56 = verify(vkA_45,s1_55) in
{18}let y_57 = isgarbageenc(e1_56) in
(
    {31}event badDec();
    0
)
else
(
    {19}let nA'_58 = dec(dkB_49,e1_56) in
    new r1_59;
    new r2_60;
    {20}let en_61 = enc(ekA_42,pair(nA'_58,nB_54),r1_59) in
    {21}let si_62 = sig(skB_52,en_61,r2_60) in
    {22}out(c, si_62);
    {23}in(c, s2_63);
    {24}let e2_64 = verify(vkA_45,s2_63) in
    {25}let y_65 = isgarbageenc(e2_64) in
    (
        {30}event badDec();
        0
    )
    else
    (
        {26}let p_66 = dec(dkB_49,e2_64) in
        {27}if fst(p_66) = nB_54 then
        {28}if snd(p_66) = testm_53 then
        {29}event end();
        0
    )
)
)

-- Query not ev:badDec()
Completing...
Starting query not ev:badDec()
RESULT not ev:badDec() is true.
```

11.4.4 file "replay_norm.pv"

```
free c.
free adv.

fun enc/3. fun ek/1. fun dk/1.
fun sig/3. fun vk/1. fun sk/1.

fun pair/2. fun garbage/1. fun garbageenc/2. fun garbagesig/2.

fun string0/1. fun string1/1. fun empty/0.

reduc fst(pair(x,y)) = x.
reduc snd(pair(x,y)) = y.
reduc equals(x,x) = x.
reduc unstring0(string0(s)) = s.
reduc unstring1(string1(s)) = s.

reduc dec(dk(r1),enc(ek(r1),y,r2)) = y.
reduc isenc(enc(ek(r1),m,r2)) = enc(ek(r1),m,r2);
  isenc(garbageenc(r1,r2)) = garbageenc(r1,r2).
reduc ekof(enc(ek(r1),m,r2)) = ek(r1);
```

```
  ekof(garbageenc(r1,r2)) = r1.
reduc isek(ek(r)) = ek(r).

reduc verify(vk(r1),sig(sk(r1),m,r2)) = m.
reduc issig(sig(sk(r1),m,r2)) = sig(sk(r1),m,r2);
  issig(garbagesig(r1,r2)) = garbagesig(r1,r2).
reduc vkof(sig(sk(r1),m,r2)) = vk(r1);
  vkof(garbagesig(r1,r2)) = r1.
reduc isvk(vk(r)) = vk(r).



query evinj:end() ==> evinj:begin().



let A' =
out(c, sig(skA, enc(ekA, nA, r), r'));
in (c, s);
let e = verify(vkB, s) in
let p = dec(dkA, e) in
if nA = fst(p) then
let nB' = snd(p) in new r1; new r2;
let en = enc(ekB, pair(nB',m), r1) in
out(c, sig(skA, en, r2) ).

let A = new nA; new r; new r'; in(adv,m);
    if m = testm then event begin(); A' else A'.

let B = new nB; in (c, s1);
let e1 = verify(vkA,s1) in
let nA' = dec(dkB,e1) in
new r1; new r2;
let en = enc(ekA, pair(nA',nB), r1) in
let si = sig(skB, en, r2) in
out (c, si );
in (c, s2);
let e2 = verify(vkA,s2) in
let p = dec(dkB,e2) in
if fst(p) = nB then
if snd(p) = testm then event end().

let C = event false().

process new rA;  let ekA = ek(rA)  in let dkA = dk(rA)  in out(c, ekA);
new rA'; let vkA = vk(rA') in let skA = sk(rA') in out(c, vkA);
new rB;  let ekB = ek(rB)  in let dkB = dk(rB)  in out(c, ekB);
new rB'; let vkB = vk(rB') in let skB = sk(rB') in out(c, vkB);
in(adv,testm);
(!A | !B)
```

11.4.5 ProVerif output for "replay_norm.pv"

```
Process:
new rA_39;
{1}let ekA_40 = ek(rA_39) in
{2}let dkA_41 = dk(rA_39) in
{3}out(c, ekA_40);
new rA'_42;
{4}let vkA_43 = vk(rA'_42) in
{5}let skA_44 = sk(rA'_42) in
{6}out(c, vkA_43);
new rB_45;
{7}let ekB_46 = ek(rB_45) in
{8}let dkB_47 = dk(rB_45) in
{9}out(c, ekB_46);
new rB'_48;
{10}let vkB_49 = vk(rB'_48) in
{11}let skB_50 = sk(rB'_48) in
{12}out(c, vkB_49);
{13}in(adv, testm_51);
{14}!
(
new nA_63;
new r_64;
new r'_65;
{28}in(adv, m_66);
{29}if m_66 = testm_51 then
(
    {38}event begin();
    {39}out(c, sig(skA_44,enc(ekA_40,nA_63,r_64),r'_65));
    {40}in(c, s_74);
    {41}let e_75 = verify(vkB_49,s_74) in
    {42}let p_76 = dec(dkA_41,e_75) in
    {43}if nA_63 = fst(p_76) then
    {44}let nB'_77 = snd(p_76) in
    new r1_78;
    new r2_79;
    {45}let en_80 = enc(ekB_46,pair(nB'_77,m_66),r1_78) in
    {46}out(c, sig(skA_44,en_80,r2_79));
    0
)
else
(
    {30}out(c, sig(skA_44,enc(ekA_40,nA_63,r_64),r'_65));
    {31}in(c, s_67);
    {32}let e_68 = verify(vkB_49,s_67) in
    {33}let p_69 = dec(dkA_41,e_68) in
    {34}if nA_63 = fst(p_69) then
    {35}let nB'_70 = snd(p_69) in
    new r1_71;
    new r2_72;
    {36}let en_73 = enc(ekB_46,pair(nB'_70,m_66),r1_71) in
    {37}out(c, sig(skA_44,en_73,r2_72));
    0
)
) | (
{15}!
new nB_52;
{16}in(c, s1_53);
{17}let e1_54 = verify(vkA_43,s1_53) in
{18}let nA'_55 = dec(dkB_47,e1_54) in
new r1_56;
```

```
new r2_57;
{19}let en_58 = enc(ekA_40,pair(nA'_55,nB_52),r1_56) in
{20}let si_59 = sig(skB_50,en_58,r2_57) in
{21}out(c, si_59);
{22}in(c, s2_60);
{23}let e2_61 = verify(vkA_43,s2_60) in
{24}let p_62 = dec(dkB_47,e2_61) in
{25}if fst(p_62) = nB_52 then
{26}if snd(p_62) = testm_51 then
{27}event end();
0
)

-- Query evinj:end() ==> evinj:begin()
Completing...
Starting query evinj:end() ==> evinj:begin()
RESULT evinj:end() ==> evinj:begin() is true.
```

References

[1] Michael Backes, Dennis Hofheinz, and Dominique Unruh. CoSP: A general framework for computational soundness proofs. In *Proc. 16th ACM Conference on Computer and Communications Security (CCS)*. 2009.

[2] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45. Springer Berlin / Heidelberg, 1998.

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

[4] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75:3–51, 2008. Online available at `http://www.di.ens.fr/~blanchet/publications/BlanchetAbadiFournetJLAP07.pdf`.

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

[6] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In George Blakley and David Chaum, editors, *Advances in Cryptology*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer Berlin / Heidelberg, 1985.

[7] GNU Project. Gnu m4 - gnu macro processor. Online available at http://www.gnu.org/software/m4/manual/index.html.