



A Decision Procedure for Alpha-Beta Privacy for a Bounded Number of Transitions

Fernet, Laouen Pablo Killian; Mödersheim, Sebastian Alexander; Viganò, Luca

Published in:

Proceedings of the 37th IEEE Computer Security Foundations Symposium (CSF 2024)

Publication date:

2024

Document Version

Early version, also known as pre-print

[Link back to DTU Orbit](#)

Citation (APA):

Fernet, L. P. K., Mödersheim, S. A., & Viganò, L. (in press). A Decision Procedure for Alpha-Beta Privacy for a Bounded Number of Transitions. In *Proceedings of the 37th IEEE Computer Security Foundations Symposium (CSF 2024)* IEEE.




General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

A Decision Procedure for Alpha-Beta Privacy for a Bounded Number of Transitions

Laouen Fernet¹ , Sebastian Mödersheim¹  and Luca Viganò² 

¹ Danmarks Tekniske Universitet, Kgs. Lyngby, Danmark

² King's College London, London, United Kingdom
{lpkf,samo}@dtu.dk, luca.vigano@kcl.ac.uk

Abstract

We present a decision procedure for verifying whether a protocol respects privacy goals, given a bound on the number of transitions. We consider *multi message-analysis problems*, where the intruder does not know exactly the structure of the messages but rather knows several possible structures and that the real execution corresponds to *one* of them. This allows for modeling a large class of security protocols, with standard cryptographic operators, non-determinism and branching. Our main contribution is the definition of a decision procedure for a fragment of alpha-beta privacy. Moreover, we have implemented a prototype tool as a proof-of-concept and a first step towards automation.

Keywords Privacy, Security Protocols, Unlinkability, Formal Methods, Automated Verification.

1 Introduction

The concept of (α, β) -privacy was introduced as an alternative way to define privacy-type properties in security protocols [1, 2]. The most widespread models of privacy use an equivalence notion between two processes to describe the goal that the intruder cannot distinguish between two possible realities. In contrast, (α, β) -privacy considers states that each represent one possible reality, and what the intruder knows about the reality in that state. This knowledge is not only in form of messages as in classic intruder models, but also in form of relations between messages, agents, etc. Together with a notion of what the intruder is allowed to know in a given state, we define a privacy violation if the intruder in any reachable state knows more than allowed. Privacy is then a question of reachability — a safety property — which is often easier to reason about and to specify than classical equivalence notions. First, one does not have to boil the privacy goal down to a distinction between two situations, which is often unnatural for more complicated properties. Second, one specifies goals positively by what the intruder is allowed to know rather than what they are

not allowed to know (and thus unable to distinguish). This essentially means that in the worst case one is erring on the safe side, i.e., allowing less than the protocol actually reveals, and thus can be alerted by a counterexample. The expressive power of equivalence notions and of (α, β) -privacy is actually hard to relate in general, due to the different nature of the approaches. However, on concrete examples it seems one can always give reasonable adaptations from one approach to the other [1, 2].

(α, β) -privacy shifts the problem from a notion of equivalence (that is a challenge for automation) to a simple reachability problem where however the privacy check for each reached state is more involved. So far, there is only one work [3] that considers a solution to checking a given state in (α, β) -privacy. However, that work is only applicable to specifications without conditional branching and it is based on an exploration of all concrete messages that the intruder can send, which are infinitely many unless one bounds the intruder.

Our main contribution in this paper is a decision procedure for the full notion of transaction processes defined by [2] for constructor/destructor theories [4, 5, 6, 7, 8]. This notion in fact entails that the intruder performs a symbolic execution of the transaction that in general yields several possibilities (due to conditional branching if the intruder does not know the truth value of the condition) and the intruder can then contrast this with all observations and experiments (constructing different messages and comparing them) to potentially rule out some possibilities. The core of our work is in a procedure to model this intruder analysis without bounding the number of steps that the intruder can make in this process. To that end, we use a popular constraint-based technique to represent the intruder symbolically, i.e., without exploring infinite sets of possibilities. In fact, we use several layers of symbolic representation to make the approach feasible.

Our decision procedure tells us whether from a given state we can reach a state that violates privacy for a fixed bound on the number of transitions. Our procedure is limited to such a bound on transitions, corresponding to the restriction to a bounded number of sessions in many approaches [9]. This is similar to the bounds needed in tools like APTE [10], AKiSs [11], SPEC [12, 13] and DeepSec [6]. In fact, this paper draws from the techniques used in these approaches, such as the symbolic representation of the intruder, a notion of an analyzed intruder knowledge, and methods for deciding the equivalence of frames. There are, however, several basic differences and generalizations. In particular, we use a symbolic handling of privacy variables (that in the equivalence-based approaches are simply one binary choice) and this is linked to logical formulas about relations between elements of the considered universe. In fact, in the prototype implementation of our decision procedure that we provide as a further contribution, we employ the SMT solver *cvc5* [14] to handle these logical evaluations. Moreover, we have multiple frames with constraints for the different possibilities resulting from conditional branching and we analyze if the intruder can rule out any possibilities in any instance.

In contrast, the tools ProVerif [4] and Tamarin [15] do handle unbounded sessions but require the restriction to so-called *diff-equivalence* [16, 8], which

drastically limits the use of branching in security protocols, though [17] recently relaxes these restrictions a bit. It seems thus in general that one has to choose between expressive power and unbounded sessions, and our approach is decidedly on the side of expressive power.

We proceed as follows. In §2, we present the notion of (α, β) -privacy in transition systems and define the problem that our procedure decides. In §3, we define how we symbolically represent messages sent by the intruder and how to solve constraints with the *lazy intruder* rules. In §4, we introduce the notion of symbolic states with their semantics. In §5, we explain how the intruder can perform experiments and make logical deductions relevant for privacy by comparing messages in their knowledge. In §6, we summarize how the different parts of the procedure are integrated. In §7, we discuss the prototype tool we have developed and its application to some examples. In §8, we discuss related and future work.

2 Preliminaries and Problem Definition

[1] introduces (α, β) -privacy as a reachability problem in a state transition system, where each state contains two formulas α and β . Intuitively, α represents what the intruder may know (e.g., the result of an election) and β what the intruder has seen (e.g., the encrypted votes). Then, a state (α, β) violates privacy iff some model of α can be excluded by the intruder knowing β , i.e., the intruder in that state can rule out more than allowed. The entire transition system violates (α, β) -privacy iff some reachable state does.

2.1 (α, β) -Privacy for a State

[1] focuses on how to define (α, β) pairs for a fixed state, and describes a state transition relation only briefly by an example. Let us also start with a fixed state. The formulas α and β are in *Herbrand logic* [18], a variant of First-Order Logic (FOL), with the difference that the universe is the quotient algebra of the *Herbrand universe* (the set of all terms that can be built with the function symbols) modulo a congruence relation \approx . This congruence specifies algebraic properties of cryptographic operators. For concreteness, we use the congruence defined in Fig. 1; for the class of properties supported by our result see Definition 6.1. The quotient algebra consists of the \approx -equivalence classes of terms.

Given an alphabet Σ , an *interpretation* \mathcal{I} interprets variable and relation symbols as usual (the interpretation of the function symbols is determined by the Herbrand universe) and we have a model relation \models_Σ as expected. By construction, $\mathcal{I} \models_\Sigma s \doteq t$ iff $\mathcal{I}(s) \approx \mathcal{I}(t)$. We say that ϕ *entails* ψ , and write $\phi \models_\Sigma \psi$, when all models of ϕ are models of ψ . We write $\phi \equiv \psi$ when $\phi \models_\Sigma \psi$ and $\psi \models_\Sigma \phi$; we may also use \equiv to define formulas.

We now fix the alphabet Σ that contains all symbols we use, namely cryptographic functions, a countable set of constants representing agents, nonces and

$\text{dcrypt}(s_1, s_2) \approx t$	if $s_1 \approx \text{inv}(k)$ and $s_2 \approx \text{crypt}(k, t, r)$
$\text{dscrypt}(k, s) \approx t$	if $s \approx \text{scrypt}(k, t, r)$
$\text{open}(k, s) \approx t$	if $s \approx \text{sign}(\text{inv}(k), t)$
$\text{pubk}(s) \approx k$	if $s \approx \text{inv}(k)$
$\text{proj}_1(s) \approx t_1$	if $s \approx \text{pair}(t_1, t_2)$
$\text{proj}_2(s) \approx t_2$	if $s \approx \text{pair}(t_1, t_2)$
and $\dots \approx \mathbf{ff}$	otherwise

Figure 1: The congruence used in this paper: `crypt` and `dcrypt` are asymmetric encryption and decryption, `scrypt` and `dscrypt` are symmetric encryption and decryption, `sign` and `open` are signing and verification/opening, `pair` is a transparent function and the `proji` are the projections, `inv` gives the private key corresponding to a public key, and `pubk` gives the public key from a private key. Here k , t , r and the t_i are variables standing for arbitrary messages. When the conditions are not met, the functions give \mathbf{ff} , which is a constant indicating failure of decryption or parsing. If `crypt` and `scrypt` are used as binary functions, we consider their deterministic variants where the random factor r has been fixed and is omitted for simplicity.

so on, and some relation symbols. We also have the set of variable symbols \mathcal{V} . Each protocol specification will fix a sub-alphabet $\Sigma_0 \subset \Sigma$ of *payload symbols*; we call $\Sigma \setminus \Sigma_0$ the *technical symbols*. All α formulas use only symbols in Σ_0 (besides variables). In the rest of the paper, we often omit the alphabet and just write \models to mean \models_{Σ_0} .

The main idea of (α, β) -privacy is that we distinguish between the actual privacy goal (e.g., an unlinkability goal talking only about agents) and the means to achieve it (e.g., the cryptographic messages exchanged).

Definition 2.1 (Adapted from [1]). *Given two formulas α over Σ_0 and β over Σ with $fv(\alpha) \subseteq fv(\beta)$, where fv denotes the free variables, we say that (α, β) -privacy holds iff for every $\mathcal{I} \models_{\Sigma_0} \alpha$ there exists $\mathcal{I}' \models_{\Sigma} \beta$ such that \mathcal{I} and \mathcal{I}' agree on the variables in $fv(\alpha)$ and on the relation symbols in Σ_0 .*

Payload. We call the formula α the *payload*, defining the privacy goal. For example, for unlinkability in an RFID-tag protocol, we may have a fixed set $\{t_1, t_2, t_3\}$ of tags and in a concrete state, the intruder has observed that two tags have run a session. Then α in that state may be $x_1, x_2 \in \{t_1, t_2, t_3\}$, meaning that the intruder is only allowed to know that both x_1 and x_2 are indeed tags, but not, for instance, whether $x_1 \doteq x_2$. In our approach, the formulas α that can occur fall into a fragment where we can always compute a finite representation of all models, in particular the variables like the x_i in the example will always be from a fixed finite domain.

Frames. For the formula β , we employ the concept of frames: a *frame*

has the form $F = l_1 \mapsto t_1 \cdots l_n \mapsto t_n$, where the l_i are distinguished constants called *labels* and the t_i are *messages* (that do not contain labels). This represents that the intruder has observed (or initially knows) messages t_1, \dots, t_n and we give each message a unique label. We call the set $\{l_1, \dots, l_n\}$ the *domain* of F . A frame can be used as a substitution, mapping labels to messages.

Recipes. To describe intruder deductions, we define a subset Σ_{pub} of the function symbols to be *public*: they represent operations the intruder can perform on known messages. For instance, all symbols used in Fig. 1 are public except for *inv*, since getting the private key is not an operation that everyone can do themselves. A *recipe* (in the context of a frame F) is any term that consists of only labels (in the domain of F) and public function symbols, so it represents a computation that the intruder can perform on F . We write $F \llbracket r \rrbracket$ for the message generated by the recipe r with the frame F .

Static equivalence. Two frames F_1 and F_2 with the same domain are *statically equivalent*, written $F_1 \sim F_2$, iff for every pair (r_1, r_2) of recipes, we have $F_1 \llbracket r_1 \rrbracket \approx F_1 \llbracket r_2 \rrbracket \Leftrightarrow F_2 \llbracket r_1 \rrbracket \approx F_2 \llbracket r_2 \rrbracket$. This means that the intruder cannot distinguish F_1 and F_2 , since any experiment they can make (i.e., compare the outcome of two computations r_1, r_2) either gives in both frames the same result or in both frames not.

Message-analysis problem. While static equivalence is typically used to formulate that two states are indistinguishable for the intruder, [1] employs instead *two* frames in each state: *concr* representing the concrete knowledge of the intruder and *struct* the structural knowledge. The messages in *struct* contain the privacy variables from α and *concr* is one concrete instance of *struct*, representing what is actually the case in that state. A *message-analysis problem* is then defined to have the form $\beta \equiv \alpha \wedge \text{concr} \sim \text{struct}$ (see [1] for details on formalizing frames in Herbrand logic), where *struct* contains only variables from α and *concr* = $\mathcal{I}(\text{struct})$ for one interpretation $\mathcal{I} \models \alpha$.

As an example, let $\alpha \equiv x_1, x_2 \in \{0, 1\}$, $\text{struct} = l_1 \mapsto h(k, x_1).l_2 \mapsto h(k, x_2)$ and $\text{concr} = l_1 \mapsto h(k, 0).l_2 \mapsto h(k, 1)$. Observe that there are four models $\mathcal{I} \models \alpha$, and in two of them $\text{concr} \sim \mathcal{I}(\text{struct})$ while $\text{concr} \not\sim \mathcal{I}(\text{struct})$ in the other two. The goal of the intruder is to rule out models that are not consistent with β . Note that β *requires* $\text{concr} \sim \text{struct}$: the intruder knows that *concr* is an instance of *struct* and thus, if an experiment distinguishes *concr* and $\mathcal{I}(\text{struct})$ then the intruder can rule out model \mathcal{I} . Thus, at this point, the intruder can exclude two models (namely those in which $x_1 \neq x_2$), so (α, β) -privacy does not hold.

Automation. A naive way to decide (α, β) -privacy for a message-analysis problem (in an algebra where static equivalence is decidable) is to compute all models $\mathcal{I}_1, \dots, \mathcal{I}_n$ of α and check whether $\mathcal{I}_1(\text{struct}) \sim \dots \sim \mathcal{I}_n(\text{struct})$ (note that in such problems $fv(\alpha) = fv(\beta)$). [3] gives a more efficient procedure that avoids the enumeration of all models: it generalizes the classical procedure for static equivalence of frames to deal with privacy variables, namely checking whether any experiment or decryption step works for every instance of the variables.

2.2 (α, β) -Privacy for a Transition System

So far we have been talking about only a single (α, β) pair, i.e., a single state of a larger transition system. [2] defines a language for specifying transition systems where the reachable states and their (α, β) pairs are defined by executing atomic transactions. We present their formalization with some minor adaptations to ease our further development.

For the presentation in this paper, we drop the concept of memory cells, which are necessary for many examples since actors need to remember what happened in previous transactions. While they are conceptually easy to integrate in the decision procedure, this takes a lot of space and distracts from the main points; so we have taken them out here and discuss them only in the extended version of this paper [19].

We distinguish two sorts of variables: the *privacy variables* $\mathcal{V}_{privacy}$, which are denoted with lower-case letters like x and are all introduced in the form $x \in D$ for a finite domain D of public constants from Σ_0 , and the *intruder variables* $\mathcal{V}_{intruder}$, which are denoted with upper-case letters like X for messages received in a transaction.

We also distinguish destructor and constructor function symbols. In Fig. 1 we have that `dcrypt`, `dscrypt`, `open`, `pubk`, `proj1` and `proj2` are destructors whereas the rest are constructors. Moreover, we call `pair` and `inv` transparent functions, because one can get all their arguments without any key (but recall that `inv` is not a public function).

Definition 2.2 (Protocol specification). *A protocol specification consists of a number of transaction processes P_i that are left processes in the syntax below, describing the atomic transactions that participants in the protocol can execute.*

We define left, center, and right processes as follows:

P_l		<i>Left process</i>
$::=$	<code>mode $x \in D.P_l$</code>	<i>Non-deterministic choice</i>
	<code> rcv(X).P_l</code>	<i>Receive</i>
	<code> P_c</code>	<i>Center process</i>
P_c		<i>Center process</i>
$::=$	<code>try $X \doteq d(t, t)$</code>	<i>Destructor application</i>
	<code>in P_c catch P_c</code>	
	<code> if ϕ then P_c else P_c</code>	<i>Conditional statement</i>
	<code> $\nu n_1, \dots, n_k.P_r$</code>	<i>Fresh constants</i>
P_r		<i>Right Process</i>
$::=$	<code>snd(t).P_r</code>	<i>Send</i>
	<code> $\star \phi.P_r$</code>	<i>Release</i>
	<code> 0</code>	<i>Terminate (nil process)</i>

where `mode` is either \star or \diamond , ϕ is a quantifier-free Herbrand formula, and d is a destructor. Destructors cannot occur elsewhere in terms. For simplicity, we have denoted destructors as binary functions, but we may similarly use unary destructors (like `proji` and `pubk` in the example).

We require that a transaction P is a closed left process, i.e., $fv(P) = \emptyset$ — we define the free variables $fv(P)$ of a process P as expected, where the non-deterministic choices, receives and fresh constants are binding. Moreover, for destructor applications:

$$fv(\text{try } X \doteq d(k, t) \text{ in } P_1 \text{ catch } P_2) = \\ fv(d(k, t)) \cup (fv(P_1) \setminus \{X\}) \cup fv(P_2)$$

Finally, a bound variable cannot be instantiated a second time and the only place destructors are allowed is in a destructor application with `try`.

Example 2.1 (Running example). Let us consider the following transaction where a server non-deterministically chooses an agent x and a yes/no-decision y , receives a message, tries to decrypt it with their own private key and then sends the decision encrypted with the public key of x :

```

★  $x \in \text{Agent}$ . ★  $y \in \{\text{yes}, \text{no}\}$ .
rcv( $M$ ).
try  $N \doteq \text{dcrypt}(\text{inv}(\text{pk}(s)), M)$  in
  if  $y \doteq \text{yes}$  then
     $\nu r. \text{snd}(\text{crypt}(\text{pk}(x), \text{pair}(\text{yes}, N), r)).0$ 
  else  $\nu r. \text{snd}(\text{crypt}(\text{pk}(x), \text{no}, r)).0$ 
catch 0

```

Here the \star means that the choice of x and y is privacy relevant and the intruder may (at least for now) only learn that $x \in \text{Agent}$ and $y \in \{\text{yes}, \text{no}\}$. The outgoing message has a different form depending on y : in the positive case the server also includes the content N of the encrypted message M they received (and if the message is not of the right format, then the transaction simply terminates); in either case the encryption is randomized with a fresh r . We may omit r if we want to model non-randomized encryption. `pk` is a public function (modeling a fixed public-key infrastructure known to everybody). \triangleleft

Much of these processes thus follows standard process calculus constructs. The special constructs of (α, β) -privacy are the non-deterministic choice and release. Choice comes in two flavors: \star if the choice is privacy relevant (as in the example), and \diamond if not. The latter means that the intruder does not a priori learn the choice, but if they find it out, it is not a violation of privacy as such. Accordingly the formula $x \in D$ is added to α when it is marked \star , and to β when it is marked \diamond . The release is used to declare that a certain fact ϕ may now be known to the intruder; we discuss this construct and what formulas can be released in the extended version.

Semantics. The semantics follows [2], with small adaptations. It is defined as a state transition system where each transition corresponds to the execution of one transaction. Thus, transactions are atomic: they cannot run concurrently with another transaction. A transaction thus consists in receiving input,

checking this input (possibly reading from memory), then making a decision (possibly updating the memory), and finally sending an output and releasing information.

The atomicity of transactions has an advantage: we can easily formalize how the intruder can reason about what is happening. In particular, we assume that the intruder at each point knows which transaction is executed and what process a transaction contains. What the intruder does not know in general are the concrete values of the variables and the truth values of conditions, and thus in which branch of an if-then-else or try-catch we are. However, the intruder can always contrast this knowledge with the observations about incoming and outgoing messages: if an observed sent message does not fit with one branch of the transaction, then the intruder knows that branch was not taken, and thus they also learn something about the truth value of the corresponding conditions. In other cases, the intruder may know what is in a received message and thus know the truth value of some condition. The intruder thus performs a symbolic execution of the transaction, leaving open what they do not know, keeping a list of possibilities, and in fact the semantics of transactions formally models this symbolic execution by the intruder.

To formalize the symbolic execution by the intruder, let a *possibility* be a triple $(P, \phi, struct)$, where P is the transaction being executed, ϕ is the conditions under which this possibility was reached and $struct$ is the structural knowledge about the messages in this possibility.

A *state* is a tuple $(\alpha, \beta_0, \gamma, \mathcal{P})$ where α, β_0 and γ are Σ_0 -formulas, and \mathcal{P} is a non-empty finite set of possibilities $\mathcal{P} = \{(P_1, \phi_1, struct_1), \dots, (P_n, \phi_n, struct_n)\}$, where one of the possibilities is marked by underlining as the possibility that is actually the case in the real execution (but the intruder does not know which one, in general). In this paper, we consider only well-formed states, where a *state* is *well-formed* iff all $struct_i$ have the same domain, γ describes a unique model of $\alpha \wedge \beta_0$ and the ϕ_i both are mutually exclusive, i.e., $\models \neg(\phi_j \wedge \phi_k)$, for $j \neq k$, and cover all models, i.e., $\alpha \wedge \beta_0 \models \bigvee_{j=1}^n \phi_j$. We define the concrete frame *concr* as the instantiation of the $struct_i$ from the underlined possibility by the model γ .

Definition 2.3 (Multi message-analysis problem (MMA)). *Given a well-formed state $S = (\alpha, \beta_0, \gamma, \mathcal{P})$, let $concr = \gamma(struct_i)$ where $(P_i, \phi_i, struct_i)$ is the marked possibility in \mathcal{P} . Define*

$$MMA(S) = \alpha \wedge \beta_0 \wedge \bigvee_{i=1}^n \phi_i \wedge concr \sim struct_i .$$

We say that S satisfies privacy iff $(\alpha, MMA(S))$ -privacy holds.

The possibilities will be used to represent that the intruder in the symbolic execution of a transaction cannot tell which conditions are true, and thus which path the execution actually takes. The $struct_i$ will contain the structural messages (i.e., containing privacy variables) that the transaction sends in the respective case, and ϕ_i is the condition under which this case was entered. In

contrast, *concr* contains the actually observed concrete messages (i.e., privacy variables are instantiated according to their true value described by γ). The intruder knows that exactly one of the ϕ_i is the case, and that $concr \sim struct_i$, i.e., the concrete messages are an instance of the messages sent in the execution path actually taken.

A state is called *finished* when all processes P_i are 0. The semantics thus defines an evaluation relation \rightarrow on states that work off the processes in each possibility until a finished state is reached. This represents the symbolic execution by the intruder of a given transaction. The branching of \rightarrow represents the non-deterministic choices of the process as well as choices of messages by the intruder.

To give a gentle introduction to (α, β) -privacy in transition systems, we present the symbolic execution at hand of the running example from Example 2.1. For the complete definition of the rules, see the extended version. As a starting point for the symbolic execution, we use the singleton set of possibilities $\{(P, \text{true}, \square)\}$ where P is the process from the running example. Let α , β_0 , and γ be true; \square is the empty frame.

2.2.1 Non-Deterministic Choice

The first step in P in the example are two non-deterministic choices of privacy variables $\star x \in \text{Agent}$. $\star y \in \{\text{yes}, \text{no}\}$. For this, the \rightarrow -relation has actually several successors, one for each possible choice of x and y . (In the decision procedure below we use a more clever way to handle all these successors as one.) The general rule defines for every $c \in D$ the following successor:

$$\begin{aligned} & \{(\text{mode } x \in D.P_1, \phi_1, struct_1), \dots, \\ & (\text{mode } x \in D.P_n, \phi_n, struct_n)\} \\ & \rightarrow \{(P_1, \phi_1, struct_1), \dots, (P_n, \phi_n, struct_n)\} \end{aligned}$$

where γ is augmented with $x \doteq c$, and if $\text{mode} = \star$ (resp. $\text{mode} = \diamond$) then α (resp. β_0) is augmented with $x \in D$. γ thus represents what really happened (which the intruder cannot see) and the information about the domain is released to α or β_0 , depending on whether x is privacy relevant. Note that x is not replaced in the P_i — this is a symbolic execution by the intruder. Also note that this rule assumes that all possibilities start with the same $\text{mode } x \in D$; this is ensured since this choice can only occur in the left part of the transaction, before any branching on conditions and tries can occur.

For the example, let us follow $x = a$ and $y = \text{yes}$; this is added to γ , and we add to α that $x \in \text{Agent}$ and $y \in \{\text{yes}, \text{no}\}$.

2.2.2 Receive

The next step is $\text{rcv}(M)$. Again the construction ensures that every process in the possibilities starts with a receive step (with the same variable). Here, the intruder can choose an arbitrary recipe r (over the domain of the $struct_i$) for

the message that should be received as M . In fact, in general, we have here infinitely many possible r and thus infinitely many successors. (Our decision procedure below uses a constraint-based approach to handle this in a finite way.) The general rule allows for every r over the domain of the $struct_i$ the following transition:

$$\begin{aligned} & \{(\text{rcv}(X).P_1, \phi_1, struct_1), \dots, (\text{rcv}(X).P_n, \phi_n, struct_n)\} \\ & \rightarrow \{(P_1[X \mapsto struct_1 \llbracket r \rrbracket], \phi_1, struct_1), \dots, \\ & \quad (P_n[X \mapsto struct_n \llbracket r \rrbracket], \phi_n, struct_n)\} \end{aligned}$$

Observe that the message that is being received depends on the possibility: it is $struct_i \llbracket r \rrbracket$ in the i th possibility, i.e., whatever the recipe r yields in the respective intruder knowledge $struct_i$.

As the intruder knowledge at this point is empty in the example, r can only be a recipe built from public constants and functions. Let us consider $r = \text{crypt}(\text{pk}(s), a, h(a))$, which then replaces M in the process.

2.2.3 Conditional Statement

The next step in the running example is $\text{try } N \doteq \dots \text{ in } P_0 \text{ catch } 0$. For the sake of this semantics, we can just consider $\text{try } X \doteq t \text{ in } P_1 \text{ catch } P_2$ as syntactic sugar for $\text{if } (t \doteq \text{ff}) \text{ then } P_2 \text{ else } P_1[X \mapsto t]$. (For the decision procedure it is important that destructors only occur in this **try-catch** form, however.)

We have a general rule that can fire when the next step in one of the possibilities is an **if-then-else**. In this case we split that possibility into two, one for the case that the condition is true and we go into the then branch, and one for the else branch:

$$\begin{aligned} & \{(\text{if } \psi \text{ then } P_1 \text{ else } P_2, \phi, struct)\} \uplus \mathcal{P} \\ & \rightarrow \{(P_1, \phi \wedge \psi, struct), (P_2, \phi \wedge \neg\psi, struct)\} \cup \mathcal{P} \end{aligned}$$

In our example, we thus have to evaluate the condition $\text{dcrypt}(\text{inv}(\text{pk}(s)), \text{crypt}(\text{pk}(s), a, h(a))) \doteq \text{ff}$, which we can simplify to **false**, i.e., the intruder knows that the received message will decrypt correctly. We thus have the two possibilities $\{(0, \text{false}, \llbracket \rrbracket), (\text{if } \dots, \text{true}, \llbracket \rrbracket)\}$, of which the second is underlined, and an evaluation rule allows removing possibilities with the condition **false**. The underlined possibility is what really happened (which is here obvious).

We thus apply a second time the condition rule, again splitting into two possibilities:

$$\begin{aligned} & \{(\nu r. \text{snd}(\dots(\text{yes}, N), r).0, y \doteq \text{yes}, \llbracket \rrbracket), \\ & \quad (\nu r. \text{snd}(\dots \text{no}, r).0, y \neq \text{yes}, \llbracket \rrbracket)\} \end{aligned}$$

Here the first possibility is what really happens (as stated by γ) and is thus underlined, but here the intruder does not know which one is the case.

The ν operator can be implemented by replacing the placeholder by a fresh non-public constant, say r_1 . We can in fact do this as a preparation before executing the transaction.

2.2.4 Send

When all the rules for the other constructs have been applied as far as possible, each of the remaining processes must be either a send or 0. If the intruder observes that a message is sent, this rules out all possibilities where the remaining process is 0. For all others, each $struct_i$ is augmented by the message sent in the respective possibility:

$$\begin{array}{c} \{(\text{snd}(t_1).P_1, \phi_1, struct_1), \dots, (\text{snd}(t_k).P_k, \phi_k, struct_k)\} \\ \vdash \mathcal{P} \\ \rightarrow \{(P_1, \phi_1, struct_1.l \mapsto t_1), \dots, (P_k, \phi_k, struct_k.l \mapsto t_k)\} \end{array}$$

where $\beta_0 \leftarrow \beta_0 \wedge \bigvee_{i=1}^k \phi_i$, l is a fresh label and all the processes in \mathcal{P} must be the 0 process. This requirement forbids applying the send rule as long as the next step of any possibility is different from send and 0 (so some of the other rules has to be applied first).

In our example we thus reach the state with

$$\begin{array}{c} \{(0, y \doteq \text{yes}, l \mapsto \text{crypt}(\text{pk}(x), \text{pair}(\text{yes}, a), r_1)), \\ (0, y \neq \text{yes}, l \mapsto \text{crypt}(\text{pk}(x), \text{no}, r_1))\} \end{array}$$

and $\text{concr}\{l\} = \text{crypt}(\text{pk}(a), \text{pair}(\text{yes}, a), r_1)$ which is a finished state, and the intruder has thus finished the symbolic execution of this transaction.

Example 2.2. Let us point out a few more interesting features of our running example. At the finished state, without further knowledge, the intruder is unable to tell which of the two possibilities is the case. This would be different if the encryption were not randomized: suppose we drop the third argument of crypt . Then the intruder could now construct $\text{crypt}(\text{pk}(x'), \text{no})$ for each value $x' \in \text{Agent}$ and compare the result with the learned message. Since this does not succeed in any case, the intruder learns that the second possibility is excluded, thus $y \doteq \text{yes}$, violating (α, β) -privacy. Even worse, if we look at the state where the non-deterministic choice was $y = \text{no}$, the intruder would find out x because exactly one of the guesses succeeds.

Reverting to randomized encryption, suppose that there had been an earlier transaction where the intruder learned $l \mapsto \text{crypt}(\text{pk}(z), \text{no}, r_2)$ for some privacy variable $z \in \text{Agent}$. If the intruder uses this as input for the next transaction, then the decryption works iff $z \doteq s$. Thus, we have a third possibility at the final sending step, namely $(0, z \neq s, l \mapsto \text{crypt}(\text{pk}(z), \text{no}, r_2))$. Then from the fact that a message was sent, the intruder can rule out this third possibility and thus deduce that $z \doteq s$, again violating (α, β) -privacy. \triangleleft

Similar to the *Send* rule, there is also a rule for the case where the process of the underlined possibility has terminated. Moreover, there is a rule for α -releasing a formula: when in the underlined possibility, the formula is added to α , otherwise it has no effect. Finally, there are rules for the reading and writing of memory cells that we have left out in this paper. Full details can be found in the extended version.

2.2.5 The Problem

We have defined the relation \rightarrow that works off the steps of a transaction, modeling an intruder's symbolic execution of a transaction P . We now define a transition relation \longrightarrow on finished states (i.e., the process in every possibility is 0) such that $S \longrightarrow S'$ iff there is a transaction P such that $\text{init}(P, S) \rightarrow^* S'$, where $\text{init}(P, S)$ denotes replacing the 0-process in every possibility of S by process P . Let the initial state be $S_0 = (\text{true}, \text{true}, \text{true}, \{(0, \text{true}, \square)\})$.

Definition 2.4 (The problem). *A protocol specification satisfies privacy iff (α, β) -privacy holds for every S s.t. $S_0 \longrightarrow^* S$.*

The contribution of the present paper is a procedure to decide whether for a given bound k , a violation is reachable in at most k steps, under the restriction of the algebraic properties to constructor/destructor theories of Definition 6.1.

3 FLICs: Framed Lazy Intruder Constraints

The semantics of the transition system says that, in a state where the processes are receiving a message, the intruder can choose any recipe built on the domain of *concr* (respectively, the *struct_i*: they all have the same domain). The problem is that there are in general infinitely many recipes the intruder can choose from. A classic technique for deciding such infinite spaces of intruder possibilities is a constraint-based approach that we call the *lazy intruder* [20, 9, 21]: it is lazy in that it avoids, as long as possible, instantiating the variables of receive steps like $\text{rcv}(X)$. The concrete intruder choice at this point does not matter; only when we check a condition that depends on X , we consider possible instantiations of X as far as needed to determine the outcome of the condition. Note that this is another symbolic layer of our approach: a symbolic state with variable X represents all concrete states where X is replaced with a message that the intruder can construct. In fact what the intruder can construct depends on the messages the intruder knew at the time when the message represented by X was sent. Due to the symbolic execution, in a state there are in general several *struct_i*, and thus we need not only to represent the messages sent by the intruder with variables but also the recipes that they have chosen, because a given recipe can produce different messages in each *struct_i*.

To keep track of this, we define an extension of frames called *framed lazy intruder constraints (FLICs)*: the entries of a standard frame represent messages that the intruder received and we write them now with a minus sign: $-l \mapsto t$. We extend this by also writing entries for messages the intruder sends with a plus sign: $+R \mapsto t$, where R is a *recipe variable* (disjoint from privacy and intruder variables). When solving the constraints, R may be instantiated with a more concrete recipe, but only using labels that occurred in the FLIC before this receive step; the order of the entries is thus significant. The messages t can contain variables representing intruder choices that we have not yet made concrete. We require that the intruder variables first occur in positive entries as they represent intruder choices made when sending a message.

Since we deal with several possibilities in parallel, we will have several FLICs in parallel, replacing the $struct_i$ in the ground model. Each FLIC has the same sequence of incoming labels and outgoing recipes. The intruder does not know in general which possibility is the case, but knows how they constructed the message from their knowledge, i.e., the recipe, which may result in a different message in each possibility.

A FLIC is a constraint, namely that the intruder can indeed produce messages of the form needed to reach a particular state of the execution. We show that we can *solve* such FLICs, i.e., find a finite representation of all solutions (as said before, there are in general infinitely many possible concrete choices) using the lazy intruder technique, similarly to other works doing constraint-based solving with frames such as [22, 6]. In the rest of this section, we will focus first on defining and solving constraints by considering just one FLIC and not the rest of the possibilities, and we explain afterwards how the lazy intruder is used for the transition system with several possibilities.

3.1 Defining Constraints

Definition 3.1 (FLIC). *A framed lazy intruder constraint (FLIC) \mathcal{A} is a sequence of mappings of the form $-l \mapsto t$ or $+R \mapsto t$, where each label l and recipe variable R occurs at most once, each term t is built from function symbols, privacy variables, and intruder variables. The first occurrence of each intruder variable must be in a message sent.*

We write $-l \mapsto t \in \mathcal{A}$ if $-l \mapsto t$ occurs in \mathcal{A} , and similarly $+R \mapsto t \in \mathcal{A}$. The domain $\text{dom}(\mathcal{A})$ is the set of labels of \mathcal{A} and $\text{vars}(\mathcal{A})$ are the privacy and intruder variables that occur in \mathcal{A} ; similarly, we write $\text{rvars}(\mathcal{A})$ for the recipe variables.

The message $\mathcal{A}\llbracket r \rrbracket$ produced by r in \mathcal{A} is: $\mathcal{A}\llbracket l \rrbracket = t$ if $-l \mapsto t \in \mathcal{A}$, $\mathcal{A}\llbracket R \rrbracket = t$ if $+R \mapsto t \in \mathcal{A}$ and $\mathcal{A}\llbracket f(r_1, \dots, r_n) \rrbracket = f(\mathcal{A}\llbracket r_1 \rrbracket, \dots, \mathcal{A}\llbracket r_n \rrbracket)$.

We also define an ordering between recipes and labels: $r <_{\mathcal{A}} l$ iff every label l' in r occurs before l in \mathcal{A} .

Example 3.1. Consider the transaction from Example 2.1, step $\text{rcv}(M)$. Using FLICs, we add $+R \mapsto M$ to the FLIC (where both R and M are fresh variables). We are lazy in the sense that we do not explore at this point what R and M might be, because any value would do. Now the server checks whether M can be decrypted with the private key $\text{inv}(\text{pk}(s))$. This is the case iff M has the form $\text{crypt}(\text{pk}(s), \cdot, \cdot)$. In the positive case, M is instantiated with $\text{crypt}(\text{pk}(s), X, Y)$ for two fresh intruder variables X and Y , thus requiring that R indeed yields a message of this form. The constraint solving in §3.2 computes a finite representation of all solutions for R . The negative case is considered separately, where we remember the negated equality $M \neq \text{crypt}(\text{pk}(s), \cdot, \cdot)$. \triangleleft

Definition 3.2 (Semantics of FLICs). *Let \mathcal{A} be a FLIC such that $\text{vars}(\mathcal{A}) = \emptyset$, i.e., the messages in \mathcal{A} are ground, so \mathcal{A} has only recipe variables. \mathcal{A} is constructable iff there exists a ground substitution of recipe variables ρ_0 such that $\mathcal{A}_1\llbracket \rho_0(R) \rrbracket \approx t$ for every recipe variable R where $\mathcal{A} = \mathcal{A}_1. +R \mapsto t. \mathcal{A}_2$.*

(This implies that only labels from $\text{dom}(\mathcal{A}_1)$ can occur in $\rho_0(R)$.) We then say that ρ_0 constructs \mathcal{A} .

Let \mathcal{A} be an arbitrary FLIC and \mathcal{I} be an interpretation of all privacy and intruder variables. We say that \mathcal{I} is a model of \mathcal{A} , written $\mathcal{I} \models \mathcal{A}$, iff $\mathcal{I}(\mathcal{A})$ is constructable. We say that \mathcal{A} is satisfiable iff it has a model.

A FLIC is thus satisfiable if there exist a suitable interpretation for the variables in messages and intruder choice for the variables in recipes such that all the constraints are satisfied.

Example 3.2. Suppose that Alice sent a signed message m to the intruder i and the constraint is to send some signed message to Bob. This is recorded in the following FLIC \mathcal{A} :

$$\begin{aligned} -l_1 &\mapsto \text{inv}(\text{pk}(i)). -l_2 \mapsto \text{crypt}(\text{pk}(i), \text{sign}(\text{inv}(\text{pk}(a)), m)). \\ +R &\mapsto \text{crypt}(\text{pk}(b), \text{sign}(\text{inv}(\text{pk}(X)), Y)) \end{aligned}$$

Here $\mathcal{I}_1 = [X \mapsto a, Y \mapsto m]$ is a model, since $\mathcal{I}_1(\mathcal{A})$ is constructable using $R = \text{crypt}(\text{pk}(b), \text{dcrypt}(l_1, l_2))$. For every ground recipe r over $\text{dom}(\mathcal{A})$ also $\mathcal{I}_r = [X \mapsto i, Y \mapsto \mathcal{A}\{r\}]$ is a model, using $R = \text{crypt}(\text{pk}(b), \text{sign}(l_1, r))$; note there are infinitely many such r . \triangleleft

3.2 Solving Constraints

We now present how to solve constraints when the intruder does not have access to destructors, i.e., as if all destructors were private functions and thus cannot occur in recipes. Hence the only place where destructors can occur are in transactions using `try-catch`. This allows us to work in the free algebra *for now* and with only destructor-free terms. We show in §6 how to integrate the lazy intruder without destructors and special transactions, so that the correctness of our decision procedure is valid for the intruder model with access to destructors.

Definition 3.3 (Simple FLIC). *A FLIC \mathcal{A} is called simple iff every message sent is an intruder variable, and each intruder variable is sent only once, i.e., every message sent is of the form $+R_i \mapsto X_i$ and the X_i are pairwise distinct.*

Simple FLICs are always satisfiable, since there are no more constraints on the messages, and the intruder can choose any recipes they want. In order to solve constraints in a non-simple FLIC, we instantiate privacy, intruder and recipe variables until we reach a simple FLIC. Computing a finite representation of all solutions is then done by keeping track of the substitutions applied to instantiate the variables.

Definition 3.4. *Let σ be a substitution that does not contain recipe variables. We define $\sigma(-l \mapsto t.\mathcal{A}) = -l \mapsto \sigma(t).\sigma(\mathcal{A})$ and $\sigma(+R \mapsto t.\mathcal{A}) = +R \mapsto \sigma(t).\sigma(\mathcal{A})$.*

For the substitutions of recipe variables, however, we cannot directly define the instantiation of recipe variables for an arbitrary FLIC, because we always

Table 1: Lazy Intruder Rules

Unification	$(\rho, \mathcal{A}_1.-l \mapsto s.\mathcal{A}_2.+R \mapsto t.\mathcal{A}_3, \sigma) \rightsquigarrow (\rho', \sigma'(\mathcal{A}_1.-l \mapsto s.\mathcal{A}_2.\mathcal{A}_3), \sigma')$ where $\rho' = [R \mapsto l]\rho$ and $\sigma' = mgu(\sigma \wedge s \doteq t)$	if $\mathcal{A}_1.-l \mapsto s.\mathcal{A}_2$ is simple, $s, t \notin \mathcal{V}$ and $\sigma' \neq \perp$
Composition	$(\rho, \mathcal{A}_1.+R \mapsto f(t_1, \dots, t_n).\mathcal{A}_2, \sigma) \rightsquigarrow (\rho', \mathcal{A}_1.+R_1 \mapsto t_1. \dots .+R_n \mapsto t_n.\mathcal{A}_2, \sigma)$ where the R_i are fresh recipe variables and $\rho' = [R \mapsto f(R_1, \dots, R_n)]\rho$	if \mathcal{A}_1 is simple, $f \in \Sigma_{pub}$ and $\sigma \neq \perp$
Guessing	$(\rho, \mathcal{A}_1.+R \mapsto x.\mathcal{A}_2, \sigma) \rightsquigarrow (\rho', \sigma'(\mathcal{A}_1.\mathcal{A}_2), \sigma')$ where $\rho' = [R \mapsto c]\rho$ and $\sigma' = mgu(\sigma \wedge x \doteq c)$	if \mathcal{A}_1 is simple, $c \in dom(x)$ and $\sigma' \neq \perp$
Repetition	$(\rho, \mathcal{A}_1.+R_1 \mapsto X.\mathcal{A}_2.+R_2 \mapsto X.\mathcal{A}_3, \sigma) \rightsquigarrow (\rho', \mathcal{A}_1.+R_1 \mapsto X.\mathcal{A}_2.\mathcal{A}_3, \sigma)$ where $\rho' = [R_2 \mapsto R_1]\rho$	if $\mathcal{A}_1.+R_1 \mapsto X.\mathcal{A}_2$ is simple and $\sigma \neq \perp$

need to make sure we instantiate both the recipe and the intruder variables according to the constraints. We thus define how to apply a substitution of recipe variables for simple FLICs.

Definition 3.5 (Choice of recipes). *A choice of recipes for a simple FLIC \mathcal{A} is a substitution ρ mapping recipe variables to recipes, where $dom(\rho) \subseteq rvars(\mathcal{A})$.*

Let $[R \mapsto r]$ be a choice of recipes for \mathcal{A} that maps only one recipe variable, where $\mathcal{A} = \mathcal{A}_1.+R \mapsto X.\mathcal{A}_2$. Let R_1, \dots, R_n be the fresh variables in r , i.e., $\{R_1, \dots, R_n\} = rvars(r) \setminus rvars(\mathcal{A})$, taken in a fixed order (e.g., the order in which they first occur in r). Let X_1, \dots, X_n be fresh intruder variables. The application of $[R \mapsto r]$ to the FLIC \mathcal{A} is defined as $[R \mapsto r](\mathcal{A}_1.+R \mapsto X.\mathcal{A}_2) = \mathcal{A}'.\sigma(\mathcal{A}_2)$ where $\mathcal{A}' = \mathcal{A}_1.+R_1 \mapsto X_1. \dots .+R_n \mapsto X_n$ and $\sigma = [X \mapsto \mathcal{A}'\{r\}]$.

For the general case, let ρ be a choice of recipes for \mathcal{A} . Then we define $\rho(\mathcal{A})$ recursively where one recipe variable is substituted at a time, and we follow the order in which the recipe variables occur in \mathcal{A} : if $\rho = [R \mapsto r]\rho'$, where R occurs in \mathcal{A} before any $R' \in dom(\rho')$, then $\rho(\mathcal{A}) = \rho'([R \mapsto r](\mathcal{A}))$. Every application $[R \mapsto r](\mathcal{A})$ corresponds to a substitution $\sigma = [X \mapsto \mathcal{A}'\{r\}]$ (as defined above), and we denote with $\sigma_\rho^{\mathcal{A}}$ the idempotent substitution aggregating all these substitutions σ from applying ρ to \mathcal{A} .

Remark. If ρ is a choice of recipes for a simple FLIC \mathcal{A} , then $\rho(\mathcal{A})$ is simple, because the fresh recipe variables added in $\rho(\mathcal{A})$ map to fresh intruder variables.

◁

Unification. We use an adapted version of syntactic unification, where we orient so that privacy variables are never substituted with intruder variables, e.g., an equality $x \doteq X$ of a privacy variable x and an intruder variable X yields the unifier $[X \mapsto x]$. We denote with $mgu(s_1 \doteq t_1 \wedge \dots \wedge s_n \doteq t_n)$ the result, called *most general unifier* (*mgu*), of unifying the s_i and t_i , which is either some substitution or \perp whenever no unifier exists. Slightly abusing notation, we consider a substitution $[x_1 \mapsto t_1, \dots, x_n \mapsto t_n]$ as the formula $x_1 \doteq t_1 \wedge \dots \wedge x_n \doteq t_n$ and \perp as false. Moreover, every privacy variable is associated to a domain by a formula $x \in D$, defining $dom(x) = D$. Thus, we filter out the mgus that are inconsistent w.r.t. the domain specifications (e.g., $[x \mapsto a]$ is filtered out if $a \notin dom(x)$).

The lazy intruder rules. In order to solve the constraints, we define a reduction relation \rightsquigarrow on FLICs. The idea is that \rightsquigarrow is Noetherian and a FLIC that cannot be further reduced is either simple or unsatisfiable. Moreover, \rightsquigarrow is not confluent, but rather is meant to explore different ways for the intruder to satisfy constraints, and thus we will consider the set of all simple FLICs that are reachable from a given one: the simple FLICs together will be equivalent to the given FLIC. Since \rightsquigarrow is not only Noetherian, but also finitely branching, the set of reachable simple FLICs is always finite by König's lemma.

Definition 3.6 (Lazy intruder rules). *The relation \rightsquigarrow is a relation on triples $(\rho, \mathcal{A}, \sigma)$ of a choice of recipes ρ , a FLIC \mathcal{A} and a substitution σ , where ρ and σ keep track of all variable substitutions performed in the reduction steps so far. We require that $\text{dom}(\rho) \cap \text{rvars}(\mathcal{A}) = \emptyset$ and $\text{dom}(\sigma) \cap \text{vars}(\mathcal{A}) = \emptyset$. The rules are defined in Table 1.*

Unification When the intruder has to send a message, they can use any message previously received and that unifies, by choosing a label for the recipe variable. Then there is one less message to send, but the unifier might make other constraints non-simple. This rule is *not* applicable for variables: the intruder is *lazy*.

Composition When the intruder has to send a composed message $f(t_1, \dots, t_n)$, they can generate it themselves if f is public and they can generate the t_i . The intruder thus chooses to compose the message themselves, so the recipe R is the application of f to other recipes.

Guessing When the intruder has to send a privacy variable x , they can guess the actual value of x , say c . In fact, this is a guessing attack as we let the privacy variables range over small domains of public constants. This rule represents the case that the intruder guesses correctly, and the variable x is replaced by the guessed value c . Note that using the **Guessing** rule does *not* yet mean that the intruder knows that c is the correct guess: in the rest of the procedure, whenever there is such a guess we model both the right and wrong guesses, and the intruder may not be able to tell what is the case.

Repetition If the intruder has to send an intruder variable that they have already sent earlier, they use the same recipe. Since there may be several ways to generate the same message, one may wonder if this is actually complete: could there be an attack where constructing the same messages in two different ways would tell the intruder anything more? In fact, for what concerns the behavior of the honest agents, it cannot make a difference, and comparing different ways to construct the same message is covered in the experiments later.

We now define the lazy intruder results as the set of recipe choices ρ that solve the constraint:

Definition 3.7 (Lazy intruder results). *Let \mathcal{A} be a FLIC and σ be a substitution. Let ε be the identity substitution. We define*

$$LI(\mathcal{A}, \sigma) = \{\rho \mid (\varepsilon, \sigma(\mathcal{A}), \sigma) \rightsquigarrow^* (\rho, \mathcal{A}', _), \mathcal{A}' \text{ is simple}\}.$$

Example 3.3. Following Example 3.1, let us assume that the intruder has already observed a message encrypted for the server from another agent x' , and is now

symbolically executing the transaction. With the constraint induced by the decryption from the server, the FLIC is now $-l \mapsto \text{crypt}(\text{pk}(s), x', r). + R \mapsto \text{crypt}(\text{pk}(s), X, Y)$. Since pk is public, the lazy intruder returns two choices of recipes: $\rho_1 = [R \mapsto l]$, meaning the intruder replays the message from the knowledge (since it unifies), and $\rho_2 = [R \mapsto \text{crypt}(\text{pk}(s), R_1, R_2)]$, meaning the intruder composes the message themselves where R_1 and R_2 stand for arbitrary recipes. \triangleleft

Definition 3.8 (Representation of choice of recipes). *Let \mathcal{A} be a FLIC, $\mathcal{I} \models \mathcal{A}$, ρ_0 be a ground choice of recipes and ρ be a choice of recipes. We say that ρ represents ρ_0 w.r.t. \mathcal{A} and \mathcal{I} iff there exists ρ'_0 such that ρ'_0 is an instance of ρ and for every $R \in \text{rvars}(\mathcal{A})$, $\mathcal{I}(\mathcal{A}) \Vdash \rho'_0(R) = \mathcal{I}(\mathcal{A}) \Vdash \rho_0(R)$ and:*

- If $\rho(R) \in \text{dom}(\mathcal{A})$, then $\rho_0(R) \in \text{dom}(\mathcal{A})$ and either $\rho'_0(R) = \rho_0(R)$ or $\rho'_0(R) <_{\mathcal{A}} \rho_0(R)$.
- If $\rho(R)$ is a composed recipe and $\rho_0(R) \in \text{dom}(\mathcal{A})$, then $\rho'_0(R) <_{\mathcal{A}} \rho_0(R)$.

This notion of representation gives the lazy intruder some “liberty”, namely to be lazy in not instantiating recipe variables that do not matter, and to replace subrecipes with equivalent ones (that may be smaller according to our ordering between recipes and labels). In the completeness proof we show that, despite all these liberties, every solution of the constraint is represented by some recipe choice that the lazy intruder finds. The lazy intruder rules are sound, complete and terminating:

Theorem 3.1 (Lazy intruder correctness). *Let \mathcal{A} be a FLIC, σ be a substitution, $\mathcal{I} \models \mathcal{A}$ such that $\mathcal{I} \models \sigma$ and let ρ_0 be a ground choice of recipes. Then ρ_0 constructs $\mathcal{I}(\mathcal{A})$ iff there exists $\rho \in \text{LI}(\mathcal{A}, \sigma)$ such that ρ represents ρ_0 w.r.t. \mathcal{A} and \mathcal{I} . Moreover, $\text{LI}(\mathcal{A}, \sigma)$ is finite.*

4 The Symbolic States

Our approach explores a symbolic transition system, i.e., transitions on symbolic states, where each symbolic state represents an infinite set of ground states. Our notion of ground states is an adaptation of the states defined in [2]. We denote symbolic states by \mathcal{S} , \mathcal{S}' , etc., and ground states by S , S' , etc.

A ground state may actually contain privacy variables, representing the possible uncertainty of the intruder in this state, but each variable has one concrete value that represents *the truth* in that state, which will be expressed by a formula γ that the intruder does not have access to (and the frame *concr* is an instance of one of the *struct_i* under γ). This is the reason why we call it a ground state, even though it contains variables. A symbolic state includes actually two symbolic layers. For the first symbolic layer, we define a symbolic state to merge all those ground states that differ only in the concrete γ and thus the concrete frame *concr*, i.e., where the intruder has the same uncertainty. Therefore, a symbolic state does not contain γ and *concr*, and has no underlined possibility.

Thus, we need to keep track of the released formula α_i for each possibility separately. A second symbolic layer is to use intruder variables and FLICs to avoid enumerating the infinite choices that the intruder has when sending messages, thus the frames $struct_i$ are generalized to FLICs \mathcal{A}_i in symbolic states.

Like a ground state, a symbolic state \mathcal{S} contains processes (one for each \mathcal{A}_i) that represent pending steps of a transaction being executed. Only when these steps have been worked off and we have only 0-processes remaining (and certain evaluations have been made), the resulting *finished* symbolic state is a reachable (symbolic) state of the transition system. This in particular ensures that transactions can only be executed atomically. Moreover, to keep track of the intruder experiments that have already been performed (i.e., comparing the outcome of two recipes — details in §5), in a symbolic state we have a set *Checked* that contains pairs of a label and a recipe.

Definition 4.1 (Symbolic state). *A symbolic state is a tuple $(\alpha_0, \beta_0, \mathcal{P}, \text{Checked})$ such that:*

- α_0 is a Σ_0 -formula, the common payload;
- β_0 is a Σ_0 -formula, the intruder reasoning about possibilities and privacy variables;
- \mathcal{P} is a set of possibilities, which are each of the form $(P, \phi, \mathcal{A}, \mathcal{X}, \alpha)$, where P is a process, ϕ is a Σ_0 -formula, \mathcal{A} is a FLIC, \mathcal{X} is a disequalities formula, and α is a Σ_0 -formula called partial payload;
- *Checked* is a set of pairs (l, r) , where l is a label and r is a recipe.

where disequalities formulas are of the following form:

$$\begin{aligned} \mathcal{X} &:= \mathcal{X} \wedge \mathcal{X} \mid \forall \bar{X}. \neg \mathcal{X}_0 && \text{Disequalities formula} \\ \mathcal{X}_0 &:= \mathcal{X}_0 \wedge \mathcal{X}_0 \mid t \doteq t && \text{Equalities formula} \end{aligned}$$

A symbolic state is finished iff all the processes in \mathcal{P} are 0.

We may write $\mathcal{S}[e \leftarrow e']$ to denote the symbolic state identical to \mathcal{S} except that e is replaced with e' .

We have augmented the FLICs \mathcal{A}_i here with disequalities \mathcal{X}_i , i.e., negated equality constraints, which allows us to restrict the choices of the intruder in a symbolic state. This is needed when we want to make a split between the case that the intruder makes a particular choice and the case that they choose anything else. This is formalized in the following definition of applying a recipe substitution which is only possible when all the respective \mathcal{X}_i are consistent with it:

Definition 4.2 (Choice of recipes for a symbolic state). *Let $\mathcal{S} = (_, _, \mathcal{P}, \text{Checked})$ be a symbolic state and ρ be a recipe substitution. We say that ρ is a choice of recipes for \mathcal{S} iff ρ is a choice of recipes for all FLICs in \mathcal{P} and for every FLIC*

\mathcal{A} and associated disequalities \mathcal{X} in \mathcal{P} , the formula $\sigma_\rho^{\mathcal{A}}(\mathcal{X})$ is satisfiable, i.e., ρ does not contradict the disequalities attached to any FLIC. Moreover, we define

$$\begin{aligned}\rho(\mathcal{P}) &= \{(\sigma_\rho^{\mathcal{A}}(P), \phi, \rho(\mathcal{A}), \sigma_\rho^{\mathcal{A}}(\mathcal{X}), \alpha) \mid \\ &\quad (P, \phi, \mathcal{A}, \mathcal{X}, \alpha) \in \mathcal{P}\} \\ \rho(\text{Checked}) &= \{(l, \rho(r)) \mid (l, r) \in \text{Checked}\} \\ \rho(\mathcal{S}) &= \mathcal{S}[\mathcal{P} \leftarrow \rho(\mathcal{P}), \text{Checked} \leftarrow \rho(\text{Checked})]\end{aligned}$$

When writing $\rho(\mathcal{S})$ in the following, we implicitly assume that all disequalities in \mathcal{S} are satisfiable under ρ , and that $\rho(\mathcal{S})$ is discarded otherwise. To decide whether disequality \mathcal{X} is satisfiable it suffices to replace the free variables with distinct fresh constants and check that the corresponding unification problems have no solution. Moreover, we will always use the lazy intruder in the context of a symbolic state, so we further assume that $LI(\cdot, \cdot)$ only returns choices of recipes for the current symbolic state, i.e., excluding any ρ that would contradict a disequality.

From a symbolic state we can define all the choices of recipes (instantiations of the recipe and intruder variables) for the messages sent by the intruder and all the concrete executions (instantiations of privacy variables) that the intruder considers possible. A symbolic state represents a set of ground states, where each ground state corresponds to one multi message-analysis problem. For every ground state, the common payload α_0 is augmented with the partial payload α_i released by the corresponding possibility. Moreover, every model γ of the privacy variables needs to be augmented by the interpretation of relation symbols. In our approach, we assume that the protocol specification contains a fixed interpretation of the relation symbols, formalized as a Σ_0 -formula γ_0 .

Meta-notation. In the specification of transactions, we allow in formulas released the use of the *meta-notation* $\gamma(t)$ for a message t : Recall that in every ground state, the real values of privacy variables is defined by a ground interpretation γ . Thus, for instance, releasing $\star x \doteq \gamma(x)$ means allowing the intruder to learn the true value of x . In the symbolic execution for ground states, the meta-notation can be implemented by using γ as a substitution before adding the formula to α .

Example 4.1. In Example 2.1, in case $x \doteq i$, then the intruder can decrypt the message and observe what was the decision. Thus they would learn both that $x \doteq i$ as well as the value of y (i.e., they know the server’s decision). This leads to a privacy violation, unless we “declassify” x and y with a release. If x is the intruder, we can release $\star x \doteq \gamma(x) \wedge y \doteq \gamma(y)$. Releasing this information is still not enough because in case $x \not\doteq i$ the intruder can also deduce that; so we additionally need to release $\star x \not\doteq i$ in that case to remove the privacy violation. \triangleleft

In a symbolic state, however, there is no γ since the symbolic state represents all possible γ at once. Hence, in order to define the semantics, we need to resolve the meta-notation that we allow in the α_i . Given α_i and the truth γ , let $[\alpha_i]^\gamma$ be the instantiation of the meta-notation in α_i , i.e., replacing every occurrence

of a term $\gamma(x)$ in α_i (for a variable x) with the actual value of x in the given γ . For instance, if $\gamma(x) = i$, then $[x \doteq \gamma(x)]^\gamma = x \doteq i$.

Definition 4.3 (Semantics of symbolic states). *Let $\mathcal{S} = (\alpha_0, \beta_0, \mathcal{P}, _)$ be a finished symbolic state. The ground states represented by \mathcal{S} are given by*

$$\begin{aligned} \llbracket \mathcal{S} \rrbracket = \{ & (\alpha_0 \wedge [\alpha_i]^\gamma, \beta_0, \gamma, \rho(\mathcal{P})) \mid (0, \phi_i, \mathcal{A}_i, _, \alpha_i) \in \mathcal{P}, \\ & \rho \text{ is a ground choice of recipes for } \mathcal{S}, \\ & \gamma \text{ is a model of } \alpha_0 \wedge \beta_0 \wedge \gamma_0 \wedge \phi_i \}, \end{aligned}$$

where $\rho(\mathcal{P})$ returns possibilities of the form $(0, \phi_j, \text{struct}_j)$, i.e., the additional components of symbolic possibilities are dropped because they are irrelevant for ground states (note that the α_i have already been used as part of the payload α); moreover, the possibility for which $\gamma \models \phi_i$ is underlined.

We say that a symbolic state \mathcal{S} satisfies privacy iff every ground state $S \in \llbracket \mathcal{S} \rrbracket$ satisfies privacy.

Remark. Given a symbolic state $\mathcal{S} = (\alpha_0, \beta_0, \mathcal{P}, _)$ and a possibility with formula ϕ_i . If $\alpha_0 \wedge \beta_0 \wedge \gamma_0 \wedge \phi_i$ is unsatisfiable, then the possibility can be removed from \mathcal{P} , as it corresponds to no ground state. In our procedure, we discard such possibilities whenever a transition is taken. \triangleleft

When computing the mgu between messages or solving constraints with the lazy intruder rules, we may deal with substitutions that contain both privacy and intruder variables. However, it is important to remember that the instantiation of privacy variables does not depend on the intruder, it is actually the goal of the intruder to learn about the privacy variables. On the other hand, intruder variables are instantiated according to the recipes chosen by the intruder. Thus, we distinguish substitutions that only substitute privacy variables.

Definition 4.4 (Privacy substitution). *Given a substitution σ , the predicate isPriv is defined as: $\text{isPriv}(\sigma)$ iff $\text{dom}(\sigma) \subseteq \mathcal{V}_{\text{privacy}}$. Moreover, define $\text{isPriv}(\perp) = \text{false}$.*

The intruder can make experiments on their knowledge by comparing the outcome of two recipes in every FLIC. It can happen that a pair of recipes gives the same message in one FLIC and different messages in another FLIC, allowing conclusions about the respective ϕ_i . In §5, we show how to extract all these conclusions and obtain a set of symbolic states in which every experiment either gives the same result in all FLICs or different results in all FLICs. This is formalized in the following equivalence relation between recipes:

Definition 4.5. *Let $\mathcal{S} = (\alpha_0, \beta_0, \mathcal{P}, _)$ be a symbolic state with $\mathcal{P} = \{(_, \phi_1, \mathcal{A}_1, _, _), \dots, (_, \phi_n, \mathcal{A}_n, _, _)\}$. Let r_1 and r_2 be two recipes and $\sigma_i = \text{mgu}(\mathcal{A}_i \upharpoonright r_1 \doteq \mathcal{A}_i \upharpoonright r_2)$ ($i \in \{1, \dots, n\}$). We define $r_1 \simeq r_2$ iff $r_1 \sqsubset r_2$ or $r_1 \bowtie r_2$, where*

$$\begin{aligned} r_1 \sqsubset r_2 \quad & \text{iff} \quad \text{for every } i \in \{1, \dots, n\}, \\ & \text{isPriv}(\sigma_i) \text{ and } \alpha_0 \wedge \beta_0 \wedge \phi_i \models \sigma_i \\ r_1 \bowtie r_2 \quad & \text{iff} \quad \text{for every } i \in \{1, \dots, n\}, \text{LI}(\mathcal{A}_i, \sigma_i) = \emptyset \\ & \text{or } (\text{isPriv}(\sigma_i) \text{ and } \alpha_0 \wedge \beta_0 \wedge \phi_i \models \neg \sigma_i) \end{aligned}$$

Intuitively, $r_1 \sqsubset r_2$ means that the two recipes produce the same message in every FLIC. Conversely, $r_1 \bowtie r_2$ means that the two recipes produce different messages in every FLIC, under any possible instantiation of the variables: either the unifier depends on intruder variables but the intruder cannot solve the constraints in any way, or the unifier depends only on privacy variables and its instances are already excluded by the intruder reasoning.

Example 4.2. Based on Example 2.1, suppose that we reached the following symbolic state containing two possibilities with $\phi_1 \equiv y \doteq \text{yes}$, $\phi_2 \equiv y \neq \text{yes} \wedge x \neq a$ and

$$\begin{aligned}\mathcal{A}_1 &= +R \mapsto N. -l \mapsto \text{crypt}(\text{pk}(x), \text{pair}(\text{yes}, N)) \\ \mathcal{A}_2 &= +R \mapsto N. -l \mapsto \text{crypt}(\text{pk}(x), \text{no})\end{aligned}$$

Here we again assume non-randomized encryption for the sake of the example. Then we have $l \bowtie \text{crypt}(\text{pk}(a), \text{no})$, because in \mathcal{A}_1 there is no unifier and in \mathcal{A}_2 the unifier $[x \mapsto a]$ is excluded by ϕ_2 . \triangleleft

We define well-formed symbolic states, where in particular what has been checked cannot distinguish the possibilities.

Definition 4.6 (Well-formed symbolic state). *Let $\mathcal{S} = (\alpha_0, \beta_0, \mathcal{P}, \text{Checked})$ be a symbolic state, with the possibilities $\mathcal{P} = \{(_, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1), \dots, (_, \phi_n, \mathcal{A}_n, \mathcal{X}_n, \alpha_n)\}$. We say that \mathcal{S} is well-formed iff*

- the ϕ_i are such that $\models \neg(\phi_i \wedge \phi_j)$ for $i \neq j$, $\text{fv}(\phi_i) \subseteq \text{fv}(\alpha_0) \cup \text{fv}(\beta_0)$ and $\alpha_0 \wedge \beta_0 \models \bigvee_{i=1}^n \phi_i$;
- the \mathcal{A}_i are simple FLICs with the same labels and same recipe variables, occurring in the same order;
- the disequalities \mathcal{X}_i are satisfiable;
- the α_i are such that $\text{fv}(\alpha_i) \subseteq \text{fv}(\alpha_0)$ and $\alpha_0 \wedge \beta_0 \wedge \gamma_0 \wedge \phi_i \models \alpha_i$; and
- for every $(l, r) \in \text{Checked}$, we have $l \simeq r$.

Recipe variables can only occur in the FLICs \mathcal{A}_i . Since $\text{dom}(\mathcal{A}_1) = \dots = \text{dom}(\mathcal{A}_n)$, we may write $\text{dom}(\mathcal{S})$ for the domain of the symbolic state.

The initially empty set *Checked* keeps track of which experiments the intruder has performed (cf. §5) and well-formedness requires that these experiments indeed no longer distinguish the possibilities. We now define a set of experiments $\text{Pairs}(\mathcal{S})$ that will be relevant: for every label l in the state and every FLIC \mathcal{A} , we try any other way to construct $\mathcal{A} \parallel l$ (except l). To that end, we use the lazy intruder to solve the constraint $\mathcal{A}.+R \mapsto \mathcal{A} \parallel l$ for a fresh recipe variable R . For each solution ρ , the experiment is the pair $(l, \rho(R))$:

Definition 4.7 (Pairs and normal symbolic state). *Let $\mathcal{S} = (_, _, \mathcal{P}, \text{Checked})$ be a symbolic state. The set of pairs of recipes to compare in \mathcal{S} is*

$$\begin{aligned}\text{Pairs}(\mathcal{S}) &= \{(l, \rho(R)) \mid l \in \text{dom}(\mathcal{S}), (_, _, \mathcal{A}, _, _) \in \mathcal{P}, \\ &\quad \rho \in \text{LI}(\mathcal{A}.+R \mapsto \mathcal{A} \parallel l, \varepsilon), \rho(R) \neq l\} \setminus \text{Checked}\end{aligned}$$

We say that \mathcal{S} is normal iff \mathcal{S} is finished and $\text{Pairs}(\mathcal{S}) = \emptyset$.

In a normal symbolic state, there are no more pairs of recipes that could distinguish the possibilities (they have all been checked). Thus, given a ground choice of recipes, all the concrete instantiations of frames are statically equivalent.

Lemma 4.1. *Let $\mathcal{S} = (\alpha_0, \beta_0, \mathcal{P}, _)$ be normal, where $\mathcal{P} = \{(0, \phi_1, \mathcal{A}_1, _, _), \dots, (0, \phi_n, \mathcal{A}_n, _, _)\}$. Let $S \in \llbracket \mathcal{S} \rrbracket$, ρ_0 be the ground choice of recipes defining S and concr be the concrete frame from S . Let $\theta \models \alpha_0 \wedge \beta_0 \wedge \phi_i$ for some $i \in \{1, \dots, n\}$ and $\text{concr}' = \theta(\rho_0(\mathcal{A}_i))$. Then $\text{concr} \sim \text{concr}'$.*

The idea is now that in a normal symbolic state, the FLICs do not contain any more insights for the intruder, and all remaining violations of (α, β) -privacy can only result from any other information β_0 that the intruder has gathered. We thus define that a symbolic state is consistent iff β_0 cannot lead to violations either:

Definition 4.8 (Consistent symbolic state). *We say that a finished symbolic state \mathcal{S} is consistent iff (α, β_0) -privacy holds for every $(\alpha, \beta_0, _, _) \in \llbracket \mathcal{S} \rrbracket$.*

Remark. By construction, β_0 can only contain symbols in Σ_0 . Even though $\llbracket \mathcal{S} \rrbracket$ is infinite, we need to consider only finitely many (α, β_0) pairs. This is because the corresponding α and β_0 in \mathcal{S} do not contain intruder variables and we only need to resolve the meta-notation if present. For truth γ , we also have only to consider finitely many instances of the privacy variables (as they range over finite domains). For each α and β_0 , the Σ_0 -models are computable as we show in the extended version. While that algorithm is based on an enumeration of models as a simple means to prove we are in a decidable fragment, our prototype tool uses the SMT solver `cvc5` to check consistency more efficiently. \triangleleft

Example 4.3. Let us consider again Example 2.1, where for now we assume that encryption is not randomized. Let $\mathcal{S} = (\alpha_0, \beta_0, \mathcal{P}, \emptyset)$ be the symbolic state such that:

$$\begin{aligned} \alpha_0 &\equiv x \in \text{Agent} \wedge y \in \{\text{yes}, \text{no}\} \\ \beta_0 &\equiv y \doteq \text{yes} \vee y \neq \text{yes} \\ \mathcal{P} &= \{(0, y \doteq \text{yes}, \mathcal{A}_1, \text{true}, \text{true}), \\ &\quad (0, y \neq \text{yes}, \mathcal{A}_2, \text{true}, \text{true})\} \\ \mathcal{A}_1 &= +R \mapsto N. -l \mapsto \text{crypt}(\text{pk}(x), \text{pair}(\text{yes}, N)) \\ \mathcal{A}_2 &= +R \mapsto N. -l \mapsto \text{crypt}(\text{pk}(x), \text{no}) \end{aligned}$$

Since there is no release in either possibility, we have that \mathcal{S} is consistent iff (α_0, β_0) -privacy holds, i.e., iff for every $\mathcal{I} \models x \in \text{Agent} \wedge y \in \{\text{yes}, \text{no}\}$, also $\mathcal{I} \models y \doteq \text{yes} \vee y \neq \text{yes}$. This clearly holds, so \mathcal{S} is consistent.

Note that if the intruder makes the experiment, e.g., of comparing l and $\text{crypt}(\text{pk}(a), \text{no})$ and considers the states where the recipes produce different messages, we would have $y \neq \text{yes} \wedge x \neq a$ for the second possibility and the symbolic state would then not be consistent (same payload but the new β_0 rules out the model $[x \mapsto a, y \mapsto \text{no}]$). \triangleleft

In a symbolic state that is both normal *and* consistent, we can combine the two properties to define, for each ground state in the semantics and model of the payload, a model of the full β and not just β_0 , using the static equivalence between concrete frames. Thus, to verify whether a normal symbolic state satisfies privacy, it suffices to verify consistency.

Theorem 4.2. *Let \mathcal{S} be a normal symbolic state. Then \mathcal{S} satisfies privacy iff \mathcal{S} is consistent.*

5 The Intruder Experiments

An *intruder experiment* is to compare pairs of recipes and the messages they produce in every frame: in a ground state, the intruder can check whether two messages are equal in the frame *concr*. In a symbolic state, each possibility considered by the intruder contains a different simple FLIC. When doing the comparison on the FLICs, the intruder may find out equalities that must hold (constraints on privacy and intruder variables) for messages to be equal. The intruder considers in separate symbolic states the possibilities where the two concrete messages are equal, and the possibilities where they are not. The result of such experiments can provide information about the values of privacy variables. Instead of comparing two arbitrary recipes, for every message t received, the intruder can try to compose t in a different way. We call these experiments *compose-checks*. We define a reduction relation \mapsto on symbolic states. Similarly to the lazy intruder rules, the idea is that \mapsto is Noetherian, but not confluent, and a symbolic state that cannot be reduced further is normal.

Definition 5.1 (Compose-checks). *The relation \mapsto is a binary relation on finished symbolic states. Let \mathcal{S} be a symbolic state $(_, \beta_0, \mathcal{P}, \text{Checked})$, with the possibilities $\mathcal{P} = \{(0, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1), \dots, (0, \phi_n, \mathcal{A}_n, \mathcal{X}_n, \alpha_n)\}$.*

Privacy split *When the intruder compares the messages produced by a label l and a recipe r , the messages may be equal under some unifiers, which depend only on privacy variables or which require a choice of recipes that has already been excluded. The formula β_0 is updated by considering in one symbolic state that the messages are equal ($l \sqsubset r$) and in the other symbolic state that the messages are unequal ($l \bowtie r$).*

$$\begin{aligned} \mathcal{S} &\mapsto \mathcal{S}[\beta_0 \leftarrow \beta_0 \wedge \bigwedge_{i=1}^n \left(\phi_i \Rightarrow \begin{cases} \sigma_i & \text{if } \text{isPriv}(\sigma_i) \\ \text{false} & \text{otherwise} \end{cases} \right) \\ \mathcal{P} &\leftarrow \{(0, \phi_i \wedge \sigma_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i) \mid i \in \{1, \dots, n\}, \text{isPriv}(\sigma_i)\} \\ \text{Checked} &\leftarrow \text{Checked} \cup \{(l, r)\} \end{aligned}$$

$$\begin{aligned}
\mathcal{S} &\mapsto \mathcal{S}[\beta_0 \leftarrow \beta_0 \wedge \bigwedge_{i=1}^n \left(\phi_i \Rightarrow \begin{cases} \neg\sigma_i & \text{if } isPriv(\sigma_i) \\ \text{true} & \text{otherwise} \end{cases} \right) \\
\mathcal{P} &\leftarrow \{(0, \phi_i \wedge \neg\sigma_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i) \mid i \in \{1, \dots, n\}, isPriv(\sigma_i)\} \\
&\quad \cup \{(0, \phi_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i) \mid i \in \{1, \dots, n\}, \text{not } isPriv(\sigma_i)\} \\
Checked &\leftarrow Checked \cup \{(l, r)\}
\end{aligned}$$

if $(l, r) \in Pairs(\mathcal{S})$ and for every $i \in \{1, \dots, n\}$, $isPriv(\sigma_i)$ or $LI(\mathcal{A}_i, \sigma_i) = \emptyset$, where $\sigma_i = mgu(\mathcal{A}_i \upharpoonright l \doteq \mathcal{A}_i \upharpoonright r)$.

Recipe split When the intruder compares the messages produced by a label l and a recipe r , the messages may be equal under some unifiers, which at least in one FLIC depend on intruder variables. Such a unifier makes one FLIC non-simple. For each lazy intruder result, there is one symbolic state in which the intruder takes a choice of recipes ρ and the whole symbolic state is updated accordingly. Additionally, there is one symbolic state in which the intruder chooses something else for the recipes so one unifier is excluded.

$$\mathcal{S} \mapsto \rho_1(\mathcal{S}), \dots, \mathcal{S} \mapsto \rho_k(\mathcal{S}), \mathcal{S} \mapsto \mathcal{S}[\mathcal{X}_i \leftarrow \mathcal{X}_i \wedge \neg\sigma_i]$$

if $(l, r) \in Pairs(\mathcal{S})$ and there exists $i \in \{1, \dots, n\}$ such that $\text{not } isPriv(\sigma_i)$ and $LI(\mathcal{A}_i, \sigma_i) = \{\rho_1, \dots, \rho_k\}$, where $\sigma_i = mgu(\mathcal{A}_i \upharpoonright l \doteq \mathcal{A}_i \upharpoonright r)$.

Example 5.1. The symbolic state \mathcal{S} from Example 4.3 is not normal since, e.g., $(l, \text{crypt}(\text{pk}(\text{a}), \text{no})) \in Pairs(\mathcal{S})$.

We can perform a compose-check, in this case by applying the privacy split rule. In \mathcal{A}_1 we have to unify $\text{crypt}(\text{pk}(x), \text{pair}(\text{yes}, N))$ and $\text{crypt}(\text{pk}(\text{a}), \text{no})$, which is not possible. In \mathcal{A}_2 we have to unify $\text{crypt}(\text{pk}(x), \text{no})$ and $\text{crypt}(\text{pk}(\text{a}), \text{no})$, which gives the mgu $\sigma = [x \mapsto \text{a}]$.

Then we get two symbolic states \mathcal{S}_1 and \mathcal{S}_2 , which have the same α_0 as \mathcal{S} but we update β_0 and \mathcal{P} . Moreover, in both \mathcal{S}_1 and \mathcal{S}_2 we have $Checked = \{(l, \text{crypt}(\text{pk}(\text{a}), \text{no}))\}$.

$$\begin{aligned}
\mathcal{S}_1 \quad \beta_0 &\equiv (y \doteq \text{yes} \vee y \not\doteq \text{yes}) \wedge (y \doteq \text{yes} \Rightarrow \text{false}) \\
&\quad \wedge (y \not\doteq \text{yes} \Rightarrow x \doteq \text{a}) \\
\mathcal{P} &= \{(0, y \not\doteq \text{yes} \wedge x \doteq \text{a}, \mathcal{A}_2, \text{true}, \text{true})\} \\
\mathcal{S}_2 \quad \beta_0 &\equiv (y \doteq \text{yes} \vee y \not\doteq \text{yes}) \wedge (y \doteq \text{yes} \Rightarrow \text{true}) \\
&\quad \wedge (y \not\doteq \text{yes} \Rightarrow x \not\doteq \text{a}) \\
\mathcal{P} &= \{(0, y \doteq \text{yes}, \mathcal{A}_1, \text{true}, \text{true}), \\
&\quad (0, y \not\doteq \text{yes} \wedge x \not\doteq \text{a}, \mathcal{A}_2, \text{true}, \text{true})\} \quad \triangleleft
\end{aligned}$$

Using the compose-checks, we can transform a symbolic state into a set of normal symbolic states, since by definition a symbolic state is normal when there are no more pairs to compare. Moreover, the compose-checks preserve the semantics of symbolic states by partitioning the ground states represented.

Theorem 5.1 (Compose-check correctness). *Let \mathcal{S} be a finished symbolic state, $(l, r) \in \text{Pairs}(\mathcal{S})$ and $\{\mathcal{S}_1, \dots, \mathcal{S}_n\}$ be the symbolic states after one rule application given the pair (l, r) . Then $\llbracket \mathcal{S} \rrbracket = \uplus_{i=1}^n \llbracket \mathcal{S}_i \rrbracket$, where \uplus denotes the disjoint union. Moreover, there is a finite number of \mathcal{S}' such that $\mathcal{S} \mapsto^* \mathcal{S}'$ and \mathcal{S}' is normal.*

6 Putting it All Together

We have so far just looked at a given symbolic state, how the intruder can solve constraints and make experiments on the FLICs — and that without destructors and algebraic properties. However, all important building blocks of the approach are now in place and we just have to use them.

We now first briefly summarize how all the rules defining the transitions from §2 can be adapted to our symbolic representation. This gives us a decision procedure for a very restricted intruder model: where the intruder has no access to destructors. As a second step we then lift the entire approach to an intruder model with a destructor theory. This is quite economical as we do not have to integrate the destructor reasoning into the constraint solving and experiments.

We outline here only the most important points of the adaptation of the symbolic execution rules. The details are found in the extended version. Let us follow the running example again. The non-deterministic choice is quite simple: instead of splitting into one successor state for each value in the domain, all these are handled in one symbolic state where we only add the domain constraint to α_0 or β_0 , respectively. For a receiving step $\text{rcv}(X)$, recall that the ground model has here an infinite branching over all the recipes that the intruder could use. This is the very reason for introducing the FLICs in the symbolic model: we simply choose a fresh recipe variable R and augment every FLIC with $+R \mapsto X$, saying that the intruder can choose any recipe R (over the labels of the FLIC so far) to form the input message X .

For conditions, we do not treat try as syntactic sugar in the symbolic approach. Consider a symbolic state \mathcal{S} where one possibility has process $\text{try } X \doteq d(t_1, t_2) \text{ in } P_1 \text{ catch } P_2$, condition ϕ , and FLIC \mathcal{A} . We define below the precise class of algebraic theories we can support, but for now it suffices that there is only one rule for the destructor d , say $d(s_1, s_2) \rightarrow s_3$ where s_3 is a proper sub-term of s_2 and let all variables in the s_i be renamed apart from the t_i and X . We compute the most general unifier σ for $s_1 \doteq t_1 \wedge s_2 \doteq t_2 \wedge s_3 \doteq X$ and use the lazy intruder to solve FLIC \mathcal{A} for it: $LI(\mathcal{A}, \sigma) = \{\rho_1, \dots, \rho_k\}$. Then we proceed with the following symbolic states $\mathcal{S}', \mathcal{S}_1, \dots, \mathcal{S}_k$, where \mathcal{S}' is the symbolic state that results from replacing the try process with P_2 and adding the disequality $\forall Y. \neg \sigma$ where Y are the intruder variables not bound by \mathcal{A} . This represents all grounds states where the intruder chose to send something for which the destructor would definitely fail. The other symbolic states \mathcal{S}_i result from applying the choice of recipes ρ_i to \mathcal{S} , and then resolving the try as follows. With ρ_i the intruder has chosen messages that have the right structure for the destructor to succeed, but it may still depend on the privacy variables in general. For

instance if the destructor is asymmetric description with key $\text{inv}(\text{pk}(\mathbf{a}))$ and the intruder chooses a message encrypted with $\text{pk}(x)$, this succeeds iff $x \doteq \mathbf{a}$. Let σ_0 be the substitution on privacy variables under which the destructor works. We can replace the possibility in question by the following two: one where we set condition $\phi \wedge \sigma_0$ and go to process P_1 ; and one where we set condition $\phi \wedge \neg\sigma_0$ and go to process P_2 . The if-then-else conditional is handled in a very similar way, obtaining a most general unifier σ under which the condition is true. (A condition composed with negation and disjunction can be first broken down into digestible pieces.) When the condition is a relation applied to some terms, the rule is the same as before: we simply split on whether the relation holds.

For releasing α information, recall that we have an α_i in each possibility that we can augment with the formula released. The rules for sending and 0 process require no changes. When all symbolic executions have terminated, we shall check whether the reached symbolic state satisfies privacy. For this, we apply the normalization procedure, i.e., perform all intruder experiments, and check consistency. Privacy is violated in the given symbolic state iff that check fails.

6.1 Lifting to Algebraic Properties

The above gives us a decision procedure for (α, β) -privacy (under a bound k on the number of transitions) as long as the intruder has no access to destructors. Note that transactions can apply destructors already. This allows for a very convenient and economical way to extend the intruder model with destructors as well without painfully extending all the above machinery to destructors: we define a set of special transactions called *destructor oracles*, one for each destructor. They receive a term and decryption key candidate, and send back the result of applying the destructor unless it fails. Note that these rules do not count towards the bound on the number of transitions, but rather we apply them to a reached symbolic state until destructors yield no further results.

6.1.1 The Supported Algebraic Theories

We have given in Fig. 1 a concrete example theory, but our result can be quite easily used for many similar theories. For instance, many modelers prefer for asymmetric cryptography that private keys are defined as atomic constants and the corresponding public key is obtained by a public function pub (so one can do without private functions). We like, in contrast, to start with public keys and have a private function inv to obtain the respective public key. This allows us to define a public function from agent names to public keys, which can be convenient in reasoning about privacy when the public-key infrastructure is fixed. Similarly, one may want to define further functions, in particular transparent functions like pair , i.e., functions that describe message serialization and where the intruder can extract every subterm. Finally, in some cases it is convenient to model some private extractor functions when we are dealing with messages where the recipient has to perform a small guessing attack. For instance, in a

protocol like Basic Hash [23] (found also in our examples `basic_hash.nn`) the reader actually needs to try out every shared key with a tag to find out which tag it is. Rather than describing transitions that iterate over all tags and try to decrypt, it is convenient to model a private extract function that “magically” extracts the name of the tag, if the message is of the correct form, and returns false otherwise. This extraction must be a private function since the intruder should not be able to see this unless they know the respective shared keys; if they do, then the experiments in our method automatically allow the intruder to perform the guessing attack.

We thus distinguish three kinds of algebraic properties of destructors that can be used arbitrarily in our approach:

Definition 6.1 (Algebraic theory). *A constructor/destructor rule is a rewrite rule of one of the following forms:*

- *Decryption:* $d(k, c(k', X_1, \dots, X_n)) \rightarrow X_i$ where d is a destructor symbol, c is a constructor symbol, $i \in \{1, \dots, n\}$, $fv(k) = fv(k')$ and the X_i are variables.
- *Transparency:* $d_i(c(X_1, \dots, X_n)) \rightarrow X_i$ where the d_i are destructors and c is a constructor c of arity n . We then say that c is transparent.
- *Private extractors:* $d(c(t_1, \dots, t_n)) \rightarrow t_0$ where d is a private destructor, c is a constructor and t_0 is a subterm of one of the t_i .

Let E be a set of such rules, where we require that every destructor d occurs in exactly one rule of E and E forms a convergent term-rewriting system. Moreover, each constructor c cannot occur both in decryption and transparency rules.

Define \approx to be the least congruence relation on ground terms such that

$$d(k, t) \approx \begin{cases} t_i & \text{if } t \approx c(k', t_1, \dots, t_n) \text{ and for some } \sigma, \\ & (d(k, c(k', t_1, \dots, t_n)) \rightarrow t_i) \in \sigma(E) \\ \mathbf{ff} & \text{otherwise} \end{cases}$$

and for unary destructors the definition is the same but k is omitted. Moreover, we require for every decryption rule $d(k, c(k', X_1, \dots, X_n)) \rightarrow X_i$ that $k = k'$ or $k \approx f(k')$ or $k' \approx f(k)$ for some public function f .

Remark. The requirement $k \approx f(k')$ or $k' \approx f(k)$ for some public f means that, given the decryption key k one can derive the encryption key k' , or the other way around. In particular, in most asymmetric encryption schemes, the public key can be derived from the private key; for signatures the private key takes the role of the “encryption key”. This requirement forces us to define in our example theory the rule $\text{pubk}(\text{inv}(k)) \rightarrow k$. Suppose that we omitted this rule, denying the intruder to derive the public key to a given private key. Suppose further that the intruder has received two messages $l_1 \mapsto \text{inv}(\text{pk}(x))$ and $l_2 \mapsto \text{pk}(y)$ and is wondering whether maybe $x \doteq y$. Then they could make the experiment whether $\text{dcrypt}(l_1, \text{crypt}(l_2, m, r)) \approx \mathbf{ff}$ and this would be the case iff $x \neq y$.

For our method, we want however to ensure that the intruder never needs to decrypt messages that they encrypted themselves. In the example, with the public-key extraction rule, the intruder can derive $\text{pubk}(\text{inv}(\text{pk}(x))) \approx \text{pk}(x)$ and now directly compare this with l_2 . The requirement allows us to show (proof in the extended version) that the intruder cannot learn anything new from decrypting terms that they have encrypted themselves. \triangleleft

Observe that every ground term t is equivalent to a unique destructor-free ground term t_0 (that we call the \approx -normal form) and that can be computed by applying a rewrite rule, when possible, to an inner-most destructor in t and replacing by ff if no rewrite rule is applicable, and repeating this until all destructors are eliminated.

6.1.2 Destructor Oracles

The idea is that transactions can already apply destructors and we can thus model oracles that provide decryption services for the intruder, namely the intruder has to provide a term to decrypt and the proposed decryption key and the oracle gives back the result of applying the destructor. More formally, given any decryption rule $(d(k, c(X_1, \dots, X_n)) \rightarrow X_i) \in E$, we define the transaction

$$\text{rcv}(X).\text{rcv}(Y).\text{try } Z \doteq d(X, Y) \text{ in } \text{snd}(Z).\text{snd}(X).0 \text{ catch } 0$$

and call it the *destructor oracle* for said rewrite rule. The handling of destructors for transparent functions is very similar to decryption rules (and easier since there is no key involved), so we will omit them in the following discussion. Finally, private extractors are not available to the intruder, anyway.

Obviously, such transactions are redundant if the intruder has access to the destructors and also it is sound to add such transactions. Also redundant is the output $\text{snd}(X)$, because X is already an input, but this ensures that different ways of composing the key will be considered by our compose-checks.

6.1.3 Term Marking and Analysis Strategy

In general the oracle rules are applicable without boundary. We use a special strategy in which to apply them that does not lead into non-termination, but covers all applications that are necessary for any attack. Note also that the application of oracle rules does not count towards the bound on the number of transitions.

All received terms and subterms in a FLIC shall be marked with one of three possible markings: \star for terms that may be decrypted but have not been; $+$ for terms that cannot be decrypted at the given intruder knowledge for any instance of the variables; and \checkmark for terms that either have already been decrypted or have been composed by the intruder himself (so the intruder knows already the subterms that may result from a decryption). We call a symbolic state *analyzed* if it does not contain any \star -marked terms anymore.

There is a strategy for applying the oracle rules to a given symbolic state \mathcal{S} to obtain a finite set of analyzed symbolic states $\mathcal{S}_1, \dots, \mathcal{S}_n$ that are together

equivalent to \mathcal{S} except that the FLICs are augmented with the results of decryptions, which we call *shorthands*. As this is quite standard we give the precise strategy only in the extended version as well as the proof of this theorem:

Theorem 6.1 (Analysis correctness). *For a symbolic state \mathcal{S} , the destructor oracle application strategy produces in finitely many steps a set $\{\mathcal{S}_1, \dots, \mathcal{S}_n\}$ of symbolic states that are analyzed. Further, for every ground state $S \in \llbracket \mathcal{S} \rrbracket$ there exists $S' \in \llbracket \mathcal{S}_i \rrbracket$, for some $i \in \{1, \dots, n\}$, such that S and S' are equivalent except that the frames in S' may contain further shorthands; and vice versa, for every $S' \in \llbracket \mathcal{S}_i \rrbracket$ there exists $S \in \llbracket \mathcal{S} \rrbracket$ such that S' is equivalent to S except for shorthands.*

For instance in the symbolic state reached after executing the transaction from Example 2.1, there is one FLIC that contains a received message $-l \mapsto \text{crypt}(\text{pk}(x), \text{pair}(\text{yes}, N), r)$ (marked \star) as well as $-l_0 \mapsto \text{inv}(\text{pk}(i))$, then the strategy will apply the oracle rule for asymmetric decryption for label l , and this gives two states \mathcal{S}_1 where for this possibility we unify $x \doteq i$ and the intruder has a new label $-l_1 \mapsto \text{pair}(\text{yes}, N)$ and \mathcal{S}_2 where we have $x \neq i$ and the intruder cannot decrypt l (given the intruder knows no other private keys $\text{inv}(\text{pk}(\cdot))$). The encrypted message at label l is now marked \checkmark in \mathcal{S}_1 and $+$ in \mathcal{S}_2 . In analyzed states, the intruder does not need any destructors anymore:

Lemma 6.2. *Let \mathcal{S} be a normal analyzed state, $S \in \llbracket \mathcal{S} \rrbracket$ and r be any recipe over the domain of S . Then there is a destructor-free recipe r' such that $\text{struct}\llbracket r \rrbracket \approx \text{struct}\llbracket r' \rrbracket$ in every frame struct of S .*

In the definition of normal state, all destructors are private functions, so the intruder can only make experiments using destructor-free recipes. Call a state *normal w.r.t. arbitrary recipes* when we allow destructors in the experiments:

Lemma 6.3. *Let \mathcal{S} be an analyzed state and normal. Then it is also normal w.r.t. arbitrary recipes.*

We can now conclude the correctness of our decision procedure. All the proofs are in the extended version. Note that we need a bound on the number of transitions, and this bound is restricting the number of transactions that are executed. All “internal” transitions taken by our compose-checks and analysis steps do not count towards that bound.

Theorem 6.4 (Procedure correctness). *Given a protocol specification for (α, β) -privacy, a bound on the number of transitions and an algebraic theory allowed by Definition 6.1, our decision procedure is sound, complete and terminating.*

7 Tool support

We have developed a prototype tool called **noname** implementing our decision procedure (available with the extended version [19]). The tool is a proof-of-concept showing that automation for (α, β) -privacy is achievable and practical.

The user must provide as input the protocol specification, consisting of the transactions that can be executed, and a bound on the number of transactions to execute. For the cryptographic operators, we make available by default primitives for asymmetric encryption/decryption, symmetric encryption/decryption, signatures and pairing (cf. Fig. 1). The user can define custom operators with the restriction to constructor/destructor theories (cf. Definition 6.1). We have also implemented an interactive running mode (the default is automatic, i.e., exploring of all reachable states) in which the user is prompted whenever there are multiple successor states, so that one can manually explore the symbolic transition system.

In case there is a privacy violation, the tool provides an attack trace that includes the sequence of atomic transactions executed and steps taken by the intruder (i.e., the recipes they have chosen) to reach an attack state, as well as a countermodel proving that the privacy goals in that state do not hold, i.e., a witness that the intruder has learned more in that state than what is allowed by the payload.

As case studies, we have focused on unlinkability goals: for the running example, we get a violation in presence of a corrupted agent. When permitting that in the corrupted case the intruder can learn the identity, the tool discovers another problem, namely that the intruder now also learns in the uncorrupted case that the involved agent is not corrupted. When releasing also that information, no more violations are found. This illustrates how the tool can help to discover all private information that is leaked, and thus either fix the protocol or permit that leak, and then finally verify that no additional information is leaked. We plan to strengthen the tool support further to make this exploration easier.

We also applied our approach to the Basic Hash [24] and the OSK [25] protocols, where OSK is particularly challenging as a stateful protocol. We have verified that the Basic Hash protocol satisfies unlinkability, but fails to provide forward privacy [23]. For the OSK protocol, we have modeled two variants where, respectively, no de-synchronization and one step de-synchronization is tolerated. For both versions the tool finds the known linkability flaws [26].

As further benchmarks we use the formalization in (α, β) -privacy of several variants of the BAC protocol by the ICAO [27] and the private authentication protocol by Abadi and Fournet [28] (denoted AF for short) that is found in [29]. For BAC, the tool finds the known problems in some implementations [30, 31, 32].

Table 2 gives an overview of the results of our tool. Finding a privacy violation is usually fast, because the tool stops as soon as it finds one without exploring the rest of the transition system. Most protocols take a few seconds to analyze, but when incrementing the bound on the number of transitions we can notice a steep increase in the verification time. Indeed, in our model, transactions can always be executed so there is in general a large number of possible interleavings. The tool seems thus to be limited by the substantial size of the search space, like earlier tools for deciding equivalence (APTE [10]). In our decision procedure, we are not deciding static equivalence between frames,

Table 2: Evaluation of the Tool

Protocol	Bound	Result	Time
Runex	2	⚡	0.35s
Runex (fix attempt)	2	⚡	0.42s
Runex (fixed)	2	✓	0.45s
Basic Hash	4	✓	1.28s
Basic Hash (compromised tag)	2	⚡	0.13s
OSK (no desynchronization)	3	⚡	0.21s
OSK (1 desynchronization step)	4	⚡	0.89s
BAC (different error messages)	3	⚡	0.14s
BAC (same error message)	4	✓	0.62s
BAC (parallel)	4	✓	0.80s
BAC (sequential)	4	✓	0.82s
AF0	2	⚡	0.98s
AF0 (fixed)	2	✓	3.14s
AF0 (fixed)	3	✓	3min52s
AF	2	✓	5.21s
AF	3	✓	8min41s

✓ = No violation, ⚡ = Violation
 Machine used: laptop with i7-4720HQ @ 2.60GHz, 8GB RAM
 GHC 9.6.2, cvc5 1.0.8

but the experiments made by the intruder to try and distinguish the different possibilities seem to have a comparable complexity. For unlinkability goals, in particular, our tool and others (for bounded sessions) essentially provide similar privacy guarantees. We share the challenges and techniques such as symbolic representation of constraints for the unbounded intruder. Thus, we believe that optimizations implemented in tools such as DeepSec [33], e.g., forms of symmetries and partial order reductions, could be adapted to our decision procedure.

8 Related and Future Work

It is a striking parallel between (α, β) -privacy and equivalence-based privacy models that the vast amount of possibilities leads to very high complexity for procedures, see, e.g., [16]. In equivalence-based approaches, the underlying problem is the static equivalence of (concrete) frames, representing two possible intruder knowledges. In (α, β) -privacy, we have instead the multi message-analysis problem: there is just one concrete frame *concr*, the observed messages, and one or more *struct_i* that result from a symbolic execution of the transactions by the intruder, where the privacy variables are not instantiated. Each

possibility has a corresponding condition ϕ_i , exactly one of which is actually true, and the intruder knows that *concr* is an instance of the corresponding *struct_i*, i.e., under the true instance of the privacy variables, $\text{concr} \sim \text{struct}_i$ for the true ϕ_i . Thus, evaluating the static equivalence can exclude several instantiations of privacy variables (even if there is just one *struct*) or rule out an entire possibility ϕ_i . The methods for solving these two problems bear many similarities, in particular one essentially in both cases looks for a pair of recipes that distinguishes the frames, i.e., the experiments that the intruder can do on their knowledge.

Like many other tools for a bounded number of sessions such as APTE [10] and DeepSec [6], we also use the symbolic representation of the lazy intruder, using variables for messages sent by the intruder that are instantiated only in a demand driven way when solving intruder constraints, turning frames into FLICs. This makes the frame distinction problems a magnitude harder (see for instance [34]). In recipes we have to also take into account variables that represent what the intruder has sent earlier and the actual choice may allow for different experiments now. We tackle this problem by first considering a model where the intruder cannot use destructors. It suffices then to check only if any message in any *struct_i* can be composed in a different way, which in turn can be solved with intruder constraint solving. This is the idea behind the notion of a *normal* state, i.e., where all said experiments have been done, and we can thus check if the results of the experiments exclude any model of α .

What makes the handling of destructors relatively easy is our requirement that all destructors yield a subterm or \mathbf{ff} , which the intruder and honest agents can see. Thus we have no problem with “garbage terms” like decryption of a nonce. This allows us to show that it is sufficient that the intruder has applied destructors as far as possible to their knowledge using the oracles — the notion of an *analyzed knowledge*: for any recipe that contains destructors, there is an equivalent recipe that uses the result of a destructor oracle.

One may wonder if a procedure for an unbounded number of steps is possible. If we look at the equivalence-based approaches, it seems the best option for this is the notion of diff-equivalence [16, 8] as used in ProVerif [4] and Tamarin [15]. Roughly speaking, diff-equivalence sidesteps the problem of the intruder’s uncertainty in branching by requiring that the conditions are either true in both executions or both false. This seems to correspond to the restriction in (α, β) -privacy that the intruder can always observe whether a condition was true or false, and we thus have just one *struct_i* in each state. We are currently investigating whether this can allow for a unbounded-step procedure similar to ProVerif for (α, β) -privacy. Again it is a striking similarity with equivalence-based approaches that one may either need a tight bound on the number of transitions or substantial restrictions on the processes one can model.

The main difference with other tools is in the properties being verified. Our tool looks at the reachable states from an (α, β) -privacy specification of a protocol, and the privacy goals are constructed by the tool when exploring the transition system. Instead of verifying whether a number of properties hold, we thus verify whether the intruder is ever able to learn more than the information

allowed (payload α). One advantage is that, in case of successful verification, we ensure that the intruder cannot learn *anything more* (about the privacy variables) than what the protocol is intentionally releasing. For some protocols such as AF, we believe that a characterization of the privacy goals with (α, β) -privacy can give a better understanding of which guarantees the protocol actually provides, as we do not see an obvious way of expressing all the privacy goals with equivalences between processes.

References

- [1] S. Mödersheim and L. Viganò, “Alpha-beta privacy,” *ACM Trans. Priv. Secur.*, vol. 22, no. 1, pp. 1–35, 2019.
- [2] S. Gondron, S. Mödersheim, and L. Viganò, “Privacy as reachability,” in *CSF 2022*. IEEE, 2022, pp. 130–146.
- [3] L. Fernet and S. Mödersheim, “Deciding a fragment of (alpha, beta)-privacy,” in *STM*, ser. LNCS, vol. 13075. Springer, 2021, pp. 122–142.
- [4] B. Blanchet, “An efficient cryptographic protocol verifier based on Prolog rules,” in *CSFW 2001*. IEEE, 2001, pp. 82–96.
- [5] B. Blanchet, M. Abadi, and C. Fournet, “Automated verification of selected equivalences for security protocols,” *J Log Algebr Program*, vol. 75, no. 1, pp. 3–51, 2008.
- [6] V. Cheval, S. Kremer, and I. Rakotonirina, “DEEPSEC: Deciding equivalence properties in security protocols theory and practice,” in *SP 2018*. IEEE, 2018, pp. 529–546.
- [7] D. Aparicio-Sánchez, S. Escobar, R. Gutiérrez, and J. Sapiña, “An optimizing protocol transformation for constructor finite variant theories in Maude-NPA,” in *ESORICS 2020*, ser. LNCS, vol. 12309. Springer, 2020, pp. 230–250.
- [8] V. Cheval, S. Kremer, and I. Rakotonirina, “The hitchhiker’s guide to decidability and complexity of equivalence properties in security protocols,” in *Logic, Language, and Security*, ser. LNCS. Springer, 2020, vol. 12300, pp. 127–145.
- [9] M. Rusinowitch and M. Turuani, “Protocol insecurity with a finite number of sessions and composed keys is NP-complete,” *Theor. Comput. Sci.*, vol. 299, no. 1, pp. 451–475, 2003.
- [10] V. Cheval, “APTE: An algorithm for proving trace equivalence,” in *TACAS 2014*, ser. LNCS, vol. 8413. Springer, 2014, pp. 587–592.

- [11] R. Chadha, V. Cheval, Ș. Ciobăcă, and S. Kremer, “Automated verification of equivalence properties of cryptographic protocols,” *ACM Trans. Comput. Logic*, vol. 17, no. 4, pp. 1–32, 2016.
- [12] A. Tiu and J. Dawson, “Automating open bisimulation checking for the spi calculus,” in *CSF*. IEEE, 2010, pp. 307–321.
- [13] A. Tiu, N. Nguyen, and R. Horne, “SPEC: An equivalence checker for security protocols,” in *APLAS*, ser. LNCS, vol. 10017. Springer, 2016, pp. 87–95.
- [14] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, “cvc5: A versatile and industrial-strength SMT solver,” in *TACAS 2022*, ser. LNCS, vol. 13243. Springer, 2022, pp. 415–442.
- [15] S. Meier, B. Schmidt, C. Cremers, and D. Basin, “The TAMARIN prover for the symbolic analysis of security protocols,” in *CAV 2013*, ser. LNCS, vol. 8044. Springer, 2013, pp. 696–701.
- [16] S. Delaune and L. Hirschi, “A survey of symbolic methods for establishing equivalence-based properties in cryptographic protocols,” *J. Log. Algebraic Methods Program.*, vol. 87, pp. 127–144, 2017.
- [17] V. Cheval and I. Rakotonirina, “Indistinguishability beyond diff-equivalence in ProVerif,” in *CSF 2023*. IEEE, 2023, pp. 184–199.
- [18] T. Hinrichs and M. Genesereth, “Herbrand logic,” Stanford University, USA, Tech. Rep. LG-2006-02, 2006. [Online]. Available: <http://logic.stanford.edu/reports/LG-2006-02.pdf>
- [19] L. Fernet, S. Mödersheim, and L. Viganò, “A decision procedure for alpha-beta privacy for a bounded number of transitions,” DTU Compute; KCL Informatics, Tech. Rep., 2023. [Online]. Available: <http://imm.dtu.dk/~samo/alphabeta/>
- [20] J. Millen and V. Shmatikov, “Constraint solving for bounded-process cryptographic protocol analysis,” in *CCS*. ACM, 2001, pp. 166–175.
- [21] D. Basin, S. Mödersheim, and L. Viganò, “OFMC: A symbolic model checker for security protocols,” *Int. J. Inf. Secur.*, vol. 4, no. 3, pp. 181–208, 2005.
- [22] V. Cheval, H. Comon-Lundh, and S. Delaune, “A procedure for deciding symbolic equivalence between sets of constraint systems,” *Inf Comput*, vol. 255, pp. 94–125, 2017.
- [23] M. Brusó, K. Chatzikokolakis, and J. den Hartog, “Formal verification of privacy for RFID systems,” in *CSF 2010*. IEEE, 2010, pp. 75–88.

- [24] S. A. Weis, S. E. Sarma, R. L. Rivest, and D. W. Engels, “Security and privacy aspects of low-cost radio frequency identification systems,” in *Security in Pervasive Computing*, ser. LNCS, vol. 2802. Springer, 2004, pp. 201–212.
- [25] M. Ohkubo, K. Suzuki, and S. Kinoshita, “Cryptographic approach to “privacy-friendly” tags,” in *RFID Privacy Workshop 2003*, 2003.
- [26] D. Baelde, S. Delaune, and S. Moreau, “A method for proving unlinkability of stateful protocols,” in *CSF 2020*. IEEE, 2020, pp. 169–183.
- [27] ICAO, “Machine readable travel documents,” Doc Series, Doc 9303, <https://www.icao.int/publications/pages/publication.aspx?docnum=9303>.
- [28] M. Abadi and C. Fournet, “Private authentication,” *Theor. Comput. Sci.*, vol. 322, no. 3, pp. 427–476, 2004.
- [29] L. Fernet and S. Mödersheim, “Private authentication with alphabet-privacy,” in *OID 2023*, ser. LNI. GI, 2023. [Online]. Available: <http://imm.dtu.dk/~samo/alphabet/>
- [30] M. Arapinis, T. Chothia, E. Ritter, and M. Ryan, “Analysing unlinkability and anonymity using the applied pi calculus,” in *CSF 2010*. IEEE, 2010, pp. 107–121.
- [31] T. Chothia and V. Smirnov, “A traceability attack against e-passports,” in *FC 2010*, ser. LNCS, vol. 6052. Springer, 2010, pp. 20–34.
- [32] I. Filimonov, R. Horne, S. Mauw, and Z. Smith, “Breaking unlinkability of the ICAO 9303 standard for e-passports using bisimilarity,” in *ESORICS 2019*, ser. LNCS, vol. 11735. Springer, 2019, pp. 577–594.
- [33] V. Cheval, S. Kremer, and I. Rakotonirina, “Exploiting symmetries when proving equivalence properties for security protocols,” in *CCS 2019*. ACM, 2019, pp. 905–922.
- [34] M. Baudet, “Deciding security of protocols against off-line guessing attacks,” in *CCS*. ACM, 2005, pp. 16–25.