A Typing Result for Alpha-Beta Privacy

Laouen Fernet
DTU Compute
Danmarks Tekniske Universitet
Kgs. Lyngby, Danmark
lpkf@dtu.dk

Sebastian Mödersheim
DTU Compute
Danmarks Tekniske Universitet
Kgs. Lyngby, Danmark
samo@dtu.dk

Luca Viganò
Department of Informatics
King's College London
London, United Kingdom
luca.vigano@kcl.ac.uk

***Abstract*—The privacy properties of security protocols can be specified with alpha-beta privacy as a reachability problem. Our main contribution is to show that, for a class of protocols satisfying certain syntactic conditions, it is correct to restrict the intruder model to a typed model, where the intruder only sends well-typed messages. Our result holds for an unbounded number of transitions and supports stateful protocols that can read from and write to memory.**

***Index Terms*—Privacy, Security Protocols, Formal Methods**

I. Introduction

a) Type Flaws: Type-flaw attacks occur when a security protocol uses several messages that have different meaning but have a similar shape so that an intruder can exploit it and send a message of one type where a message of another type is expected. For example, one message of the protocol is a signature on a nonce for challenge-response, say $\mathsf{sign}(\mathsf{inv}(\mathsf{pk}(x)), N)$ (where inv denotes the private key to the public key $\mathsf{pk}(x)$ of agent x), and another message is a signature on an encrypted message like $\mathsf{sign}(\mathsf{inv}(\mathsf{pk}(y)), \mathsf{crypt}(\mathsf{pk}(z), M, R))$. It is actually easy to prevent type-flaw attacks by good protocol design: messages should not sign or encrypt raw data, but rather include a few bits of information that specify what is the meaning of the message. In the example, the signatures should contain at least some kind of tag that distinguishes the different types of signed statements. Such a countermeasure is not only almost for free, it is completely in line with prudent engineering principles [1], [2].

b) Benefits of Typing: Formal verification of security protocols generally gets easier if we can rule out type-flaw attacks and analyze everything in a *typed model* where the intruder is restricted to sending well-typed messages. Then, many security problems become decidable (and, e.g., one can guarantee termination of tools like ProVerif [3]).

c) Typing Results: This motivates a relative soundness result of the form: "if a protocol that obeys certain type-flaw resistance requirements has an attack, then it has a well-typed attack." It is then sound to verify such a protocol in the typed model. This is particularly relevant in practice, if many existing protocols without modification already satisfy type-flaw requirements.

d) The Proof Idea: Most of the existing typing results, e.g., [4], [5], [6], [7], [8], use a constraint-based method for analyzing security protocols that is based on a symbolic approach, which we call here the *lazy intruder*: this technique avoids exploring all the messages that the intruder could generate at a given point, but instead uses a variable with the constraint that this variable represents any message that the intruder can generate from their current knowledge. This variable is only instantiated when the choice matters for the attack. One can then show that in a type-flaw resistant protocol, these instantiations are always well-typed, and that all remaining variables (that do not matter for the attack in the end) can be instantiated with something well-typed as well. Thus, if an attack exists, there exists a well-typed one. Although this method yields a decision procedure only for a bounded number of sessions, since the argument applies to an attack of arbitrary length, the typing result is not bounded to a fixed number of sessions and can be used in approaches/tools that do not use the lazy intruder (like ProVerif).

e) Typing for Privacy: A trend in protocol verification is the support for privacy-type properties such as unlinkability or vote-secrecy, i.e., secrecy of a choice over a small domain of intruder-known values. This is challenging for verification tools and thus many tools require a restriction like diff-equivalence [9], [10] where, roughly speaking, conditions—and thus control flow—cannot depend on the private choice.[1] It is thus very desirable to simplify the tools' lives by a typing result, but that is harder to obtain for privacy as well. For instance, a typing result needs to exclude that the intruder can gain any insight about a condition (and thus possibly private choices) by sending an ill-typed message. This is in fact related again to the problem of control flow (that classical diff-equivalence sidesteps): the intruder may not know in general what exactly is happening in the protocol, while in standard protocol verification the intruder is only unclear about the concrete value of some cryptographically strong secrets.

f) (α, β)-privacy: Another difficulty for typing (and verification in general) is that most approaches for privacy are based on observational equivalence notions, i.e., whether the intruder can distinguish between two variants of a process, e.g., unlinkability as the distinction between the scenario where every agent performs only one session and the scenario where they perform any number of sessions. This makes even the statement of a typing result rather involved. Gondron, Mödersheim and Viganò [12] proposed the notion of (α, β)-privacy

[1] There are, however, recent extensions of these concepts that considerably relax these restrictions [11].

that instead is based on a classical state-transition system, where each state has enough information to evaluate privacy questions. In a nutshell, the modeler can specify for each transaction (an atomic step of the protocol) a formula α that expresses information publicly released, e.g., in unlinkability one may simply release only the domain constraint that the actor belongs to a set of actors who can execute the protocol. Further, the semantics models the inference process of the intruder, who tries to analyze the messages observed and to relate them to the steps of the protocol, possibly learning the value of conditions. This yields a formula β for every state, and privacy is then defined as: β does not allow to exclude a model of α. A decision procedure for (α, β)-privacy for a bounded number of transitions, based on the lazy intruder technique, is given in [13].

g) Contributions: In this paper, we define a set of requirements for protocols and algebraic theories we can support, and prove that under these requirements the procedure from [13] performs only well-typed instantiations of variables and well-typed intruder experiments. This allows us to prove a typing result for (α, β)-privacy: "if there is an attack, then there is a well-typed one." As in previous typing results, this is independent of the number of transitions considered. This result is, to our knowledge, not only more general than previous typing results for privacy, since the requirements are less restrictive and a larger class of protocols is considered, but it also has a more declarative proof. We discuss this generality and the relation to other existing works in §VII. Before we do so, we summarize (α, β)-privacy and the decision procedure from [13] in §II, define the class of *type-flaw resistant* protocols that our result supports in §III, present the typing result on the constraint level in §IV and the full typing result for an unbounded number of transitions in §V, and report our experiments on case studies in §VI. All the proofs, together with additional details for the semantics and the models of the case study protocols, are given in the appendix.

II. Preliminaries

A. Term Algebra and All That

We consider terms over an alphabet Σ, containing function and relation symbols with their arity, and interpret the terms in the quotient algebra modulo a congruence relation $\approx$. Functions can be either *public* or *private* (accessible/not accessible to the intruder). For our purpose, the congruence allows for constructors and destructors like encryption and decryption, where decryption failure yields a distinguished constant $\mathtt{ff}$. Definition III.3 describes the precise class of algebraic theories that our result supports. Formulas (typically α, β, or ϕ) are in *Herbrand logic* [14], which is like standard First-Order Logic where the universe is said quotient algebra.

We use standard definitions like: $fv(\cdot)$ returning the free variables of a term or formula; linear terms (every variable occurs at most once); the interpretation $\mathcal{I}$ mapping all variables to the universe, and n-ary relations to a set of n-tuples of the universe; the models relation $\mathcal{I} \models \phi$ expressing that $\mathcal{I}$ is a satisfying interpretation for ϕ; $\equiv$ for logic equivalence of formulas (and for defining formulas).

B. (α, β)-Privacy

The main idea of (α, β)-privacy is that every state of the world contains a formula α that represents what the intruder is allowed to know and the formula β represents what the intruder has actually observed. A state violates (α, β)-privacy iff some model of α can be ruled out by the intruder knowledge in β, i.e., the intruder has learned more than what is allowed. We use a sub-alphabet $\Sigma_0 \subset \Sigma$ containing the *payload symbols*, which are used to express the privacy goals. The complement $\Sigma \setminus \Sigma_0$ contains the *technical symbols*, which are used to represent the intruder knowledge (e.g., the cryptographic messages observed).

Definition II.1 ((α, β)-privacy [13])**.** *Given two formulas α over Σ_0 and β over Σ with $fv(\alpha) \subseteq fv(\beta)$, we say that* (α, β)-privacy holds *iff for every $\mathcal{I} \models \alpha$ there exists $\mathcal{I}' \models \beta$ such that $\mathcal{I}$ and $\mathcal{I}'$ agree on the variables in $fv(\alpha)$ and on the relation symbols in Σ_0.*

C. Protocol Specification

To describe a state-transition system where the formulas α and β reflect what has been released and observed so far, respectively, we use the notion of (α, β)-*transaction* from [12], [13], where a transaction is an atomic step of a protocol participant that mainly consists of receiving a message, checking and modifying their long-term local state, and sending an answer. The transactions give rise to an infinite-state transition system and the question is whether every reachable state satisfies (α, β)-privacy.

We distinguish three sorts of variables: *privacy variables* $\mathcal{V}_{privacy}$ (typically denoted x, y), each chosen from a finite domain D of public constants in Σ_0; *intruder variables* $\mathcal{V}_{intruder}$ (typically denoted X, X_i), which come from messages received and reading the memory; and *recipe variables* (typically denoted R, R_i), used in the symbolic constraints to represent the choices made by the intruder.

Definition II.2 (Protocol specification, adapted from [13])**.** *A* protocol specification *consists of*

- *a number of* transaction processes P_l*, which are left processes according to the syntax below, describing the atomic transactions that protocol participants can execute;*
- *a number of* memory cells*, e.g.,* $\mathsf{cell}(\cdot)$*, together with a ground context $C[\cdot]$ for each memory cell defining the initial value of the memory, so that initially* $\mathsf{cell}(t) = C[t]$.

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

P_l			*Left process*
	::=	$\mathsf{mode}\ x \in D.P_l$	*Non-deterministic choice*
	|	$\mathsf{rcv}(X).P_l$	*Receive*
	|	$\mathsf{try}\ X := d(t, X)\ \mathsf{in}\ P_l$	*Destructor application*
	|	P_c	*Center process*
P_c			*Center process*
	::=	$X := \mathsf{cell}(t).P_c$	*Cell read*
	|	$\mathsf{if}\ \phi\ \mathsf{then}\ P_c\ \mathsf{else}\ P_c$	*Conditional statement*
	|	$\nu n_1, \ldots, n_k.P_r$	*Fresh constants*
P_r			*Right Process*
	::=	$\mathsf{snd}(t).P_r$	*Send*
	|	$\mathsf{cell}(t) := t.P_r$	*Cell write*
	|	$\star\ \phi.P_r$	*Release*
	|	0	*Terminate (nil process)*

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. We may omit writing* $\mathsf{else}\ 0$ *and trailing* 0*'s.*

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, cell reads and fresh constants are binding. Moreover, for destructor applications:*

$$fv(\mathsf{try}\ X := d(k, Y)\ \mathsf{in}\ P) = fv(d(k, Y)) \cup (fv(P) \setminus \{X\})$$

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

A transaction is thus partitioned into three distinguished sub-processes: the left part for receiving messages, making non-deterministic choices and applying destructors; the center part for performing checks on messages and memory; and the right part for modifying memory and sending messages.

The left part allows for choosing values of privacy variables like $\star\ x \in \{0, 1\}$ which means that x will be one of the two values and a priori the intruder does not know which; α is augmented with the conjunct $x \in \{0, 1\}$, so the intruder is allowed to know the domain of x. Unless further information about x is released, it is then a violation if the intruder learns more about x, e.g., $x \doteq x'$. We use $\doteq$ to denote equality between terms in formulas, and that is interpreted as equality modulo the congruence relation. The construct $\diamond\ x \in D$ is used when the choice of x is not privacy-relevant in itself: if the intruder learns anything about x it does not count as a privacy violation.

Moreover, we can apply destructors to messages; all destructors return either a subterm of the message being decomposed or $\mathtt{ff}$ for failure. In the case of failure, the process behaves as 0. In [13], there can be a process for handling failure, e.g., sending an error message, while we can only support transactions that silently stop in case of destructor failure. Moreover, try in [13] is part of the center process, while we require it as part of the left process, i.e., before branching, so that any destructor failure means that the entire transaction goes directly to 0. We discuss in more details why we make these changes in Remark III.1.

In our examples, we will use for instance the operators crypt and dcrypt for asymmetric encryption and decryption, and pair, proj_1 and proj_2 for pairing and projection. The relation between constructors and destructors is described by equations like $\mathsf{dcrypt}(\mathsf{inv}(k), \mathsf{crypt}(k, m, r)) \approx m$, where inv is a private constructor mapping public to private keys. The precise class of properties we support is found in Definition III.3.

In the right process, the release means that the respective formula ϕ becomes a new conjunct of the α formula in the successor state. All other constructs are standard. Note that we do not need a locking mechanism for reading and writing memory cells, because all transactions are atomic and cannot have race conditions among each other.

Example II.1. The following transaction illustrates several features of (α, β)-transactions.

$$\begin{aligned}
&\star\ x \in \{0, 1, 2\}.\\
&\mathsf{rcv}(M).\\
&\mathsf{try}\ K := \mathsf{dcrypt}(\mathsf{inv}(\mathsf{pk}(\mathsf{a})), M)\ \mathsf{in}\\
&\mathsf{if}\ x \doteq 0\ \mathsf{then}\ \star\ x \doteq 0.\,\mathsf{snd}(\mathsf{no})\\
&\qquad\qquad \mathsf{else}\ \star\ x \not\doteq 0.\,\mathsf{snd}(\mathsf{scrypt}(K, x))
\end{aligned}$$

A privacy variable x is chosen from a known domain, a message M is received and decrypted with the private key of agent a. If decryption is successful, there is a branching on the value of x. Different messages are sent in the two branches, and the intruder is able to find out whether the condition was true or not based on the reply observed, which would be a privacy violation, if that information were not released in each of the branches. $\triangleleft$

D. α and β and Frames

The formula α of a reached state is simply the conjunction of all ϕ for which $\star\ \phi$ has been executed, while β is more complicated. For this we first need to define frames.

A *frame* is a notion that is commonly used to characterize the knowledge of the intruder: we have a set of distinguished constants called *labels* that do not occur in normal messages; a frame F maps such labels to messages. The labels allow for describing intruder actions like encryption and decryption by a *recipe* r: a term built from labels and public functions. Here, $F(r)$ is the term that results by replacing all labels in r with the corresponding message from F; we ensure throughout the paper that $F(r)$ is only used when all labels in r occur in the domain of F.

We speak of an *experiment* for a frame F when the intruder checks whether two recipes r_1 and r_2 over the domain of F give the same result, i.e., whether $F(r_1) \approx F(r_2)$. We say two frames F_1 and F_2 are *statically equivalent*, written $F_1 \sim F_2$, iff there is no experiment to distinguish them, i.e., every experiment will either give the same result in both frames, or

different results in both frames. (This implicitly requires that the frames have the same domain.)

Even though we assume that the intruder knows which transactions are being executed, they do not know a priori the value of the privacy variables. For instance, the intruder may observe a concrete message $\mathsf{crypt}(k, 0, r)$ but only know that it has the form $\mathsf{crypt}(k, x, r)$ if that is the sent message according to the transaction. We thus distinguish here the *concrete* knowledge $concr$ and the *structural* knowledge $struct$. As a consequence, the intruder may not know the concrete value of any message being sent or received in a transaction, and thus may not know at a condition if ϕ, whether ϕ is true, and thus which branch is taken. Thus the intruder needs to make a case split in their analysis of what is happening. As a consequence, the intruder has in general a set $\{(\phi_1, struct_1), \ldots, (\phi_n, struct_n)\}$ of possibilities how the process could have executed, where ϕ_i are the conditions for arriving at the ith case, and $struct_i$ is the structure that the received messages would have in that case. There is only one concrete knowledge $concr$, however, as these are the truly observed messages. Note that the ϕ_i partition the space of models for the privacy variables: in each interpretation, exactly one ϕ_i is true.

The formula β is now $\alpha \wedge \psi \wedge \bigvee_{i=1}^{n} \phi_i \wedge struct_i \sim concr$, where ψ is the conjunction $\bigwedge x \in D_x$ over every variable that was chosen with $\diamond$ (recall that the domains of variables chosen with $\star$ are part of α). β thus expresses that the intruder knows that some ϕ_i is true and in that case the structural knowledge of that case is statically equivalent to the concrete knowledge.

E. Deciding (α, β)-Privacy

In [13], Fernet, Mödersheim and Viganò prove the correctness of a procedure that decides the problem for a given (symbolic) state and thus also for reachability with a given bound on the number of transitions. We only sketch the decision procedure from [13] here and in the following sections we then give those parts in detail that are relevant for the proof of the typing result.

The procedure uses an extension of the notion of frames called *FLICs*, which represent both sent messages and received messages, where the ordering is relevant. The messages can contain intruder variables that represent arbitrary messages from the intruder that the procedure has not yet determined.

Definition II.3 (FLIC and simple FLIC [13])**.** *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 *$dom(\mathcal{A})$ is the set of labels of $\mathcal{A}$ and $vars(\mathcal{A})$ are the privacy and intruder variables that occur in $\mathcal{A}$; similarly, we write $rvars(\mathcal{A})$ for the recipe variables.*

The message $\mathcal{A}(r)$ produced *by r in $\mathcal{A}$ is:*

$$\begin{aligned} \mathcal{A}(l) &= t \qquad \text{if } -l \mapsto t \in \mathcal{A} \\ \mathcal{A}(R) &= t \qquad \text{if } +R \mapsto t \in \mathcal{A} \\ \mathcal{A}(f(r_1, \ldots, r_n)) &= f(\mathcal{A}(r_1), \ldots, \mathcal{A}(r_n)) \end{aligned}$$

For recipes that use labels or recipe variables not defined in the FLIC, the result is undefined.

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.*

The FLIC is regarded as a *constraint* that asks for an instantiation of the intruder and privacy variables such that every message sent by the intruder can be derived from messages received by the intruder up to that point. At the core of the decision procedure is the *lazy intruder*: a set of rules to transform the FLICs into a finite set of equivalent FLICs in *simple* form; all remaining messages that the intruder has to send are distinct intruder variables, i.e., the intruder can choose arbitrary messages in these places.

The first step in executing a transaction is the symbolic execution of the different parts of the process, using FLICs and the lazy intruder whenever there are non-simple constraints to solve. Once the transaction has been fully executed, the next step is performing the intruder experiments on the messages observed by the intruder. This can lead to deductions about the privacy variables, if the intruder can rule out particular cases or particular instantiations of the privacy variables. This may, in fact, give an attack.

Finally, as a last step there is also analysis performed on top of this: the entire procedure up to this point considers recipes with constructors only. The destructors are then applied in a saturating way: the messages received by the intruder may yield subterms if they can be successfully decrypted, and there is a strategy to perform all relevant analyses (i.e., that add to the intruder knowledge) in finitely many steps.

III. Typed Model

The idea of the present paper is that we define a class of *type-flaw resistant* protocols and show for those protocols that the procedure given by [13] never performs any ill-typed substitutions. As a consequence, we show that, if there is an attack, then there is an attack that uses only well-typed messages. This is in fact proved for an arbitrary reachable state in a type-flaw resistant protocol, i.e., our typing result holds without any bound on the number of transitions. We will make two adaptations to the procedure: we extend semantics to support pattern matching, and we formulate analysis as built-in transitions instead of explicit application of destructors; we also show that these transformations are correct. For the typing result in §IV, we show that, for a type-flaw resistant protocol, the lazy intruder always returns well-typed solutions to the constraints. The intruder experiments of comparing pairs of recipes are never a typing issue. In §V, we obtain our main result for the transition system.

We first define a simple type system and we will require that the protocol specifies a type for each message. For a message received in a transaction, the type annotation expresses the intended type that should be used by the honest participants. We cannot a priori enforce that the intruder respects these types, i.e., the intruder is able to send ill-typed messages. Our main result is to show that, for the class of protocols we call *type-flaw resistant*, we can actually consider only well-typed choices by the intruder without loss of generality for finding privacy violations. The goal is to show that, if there exists a reachable state that violates privacy, then there exists a similar state that is reachable only using well-typed messages. We prove this by using the notion of symbolic states with constraints, where the ground solutions to the constraints define the ground states. We show that in the entire exploration of symbolic constraints, no ill-typed substitution of variables occurs, and that simple constraints always admit a well-typed solution. Thus, there is nothing the intruder can achieve using ill-typed messages that would not be similarly achieved with well-typed ones.

A. Type System

Types are defined similarly to terms. Instead of a set of variables, we use a set of *atomic types*, e.g., $\{\mathsf{agent}, \mathsf{nonce}\}$. The *composed types* are defined using the functions in Σ, with the restriction to constructors of non-zero arity, i.e., we forbid destructors and constants in composed types. The type system assigns an atomic or composed type to every message with the following requirements:

Definition III.1 (Typing function)**.** *A* typing function Γ *is s.t.:*

- $\Gamma(c)$ *is atomic for every function* $c \in \Sigma$ *of arity* 0.
- $\Gamma(f(t_1, \ldots, t_n)) = f(\Gamma(t_1), \ldots, \Gamma(t_n))$ *for every constructor* $f \in \Sigma$ *of arity* $n > 0$.
- $\Gamma(x)$ *is a type (atomic or composed) for every variable* $x \in \mathcal{V}$.

Our type system does not include terms containing destructors, because they represent terms that need to be evaluated and we rather want to give a type to the result. In a protocol specification, destructors can only occur as part of a destructor application of the form $\mathsf{try}\ Y := d(k, X)\ \mathsf{in}\ \ldots$ where either the result is $\mathtt{ff}$ and the transaction stops, or Y is bound to the respective subterm of X, and thus shall have the respective (destructor-free) type.

The fact that instantiations of variables are well-typed is defined with the notion of a substitution being well-typed.

Definition III.2 (Well-typed substitution)**.** *A substitution* σ *is* well-typed *iff for every* $x \in dom(\sigma)$*, we have* $\Gamma(x) = \Gamma(\sigma(x))$.

We need to ensure that the intruder is always able to make a well-typed choice, therefore they must be able to compose arbitrarily many messages of each type, even before receiving any message from honest agents. Hence, we require that, for each *atomic* type, there is an infinite set of public constants of that type, i.e., the intruder initially knows an unbounded number of constants of each atomic type. Suppose all function symbols were public, then the intruder would also immediately have access to an unbounded number of terms of every composed type. In fact, [7] observes that, even if all functions are public, one can still model a private function f of arity n by a public function f' of arity $n+1$, where the additional argument is filled with a distinct secret constant. Thus, private functions like f are just syntactic sugar. We adopt this suggestion and, for the rest of the paper, continue to use public and private functions, where we use a subset $\Sigma_{pub} \subseteq \Sigma$ to identify the public functions.

We first define the precise class of algebraic theories that our result supports. Here we slightly deviate from standard approaches for constructor-destructor theories that consider directly the congruence induced by a set of rewrite rules, because this can lead to "garbage terms", i.e., a failed destructor application like $\mathsf{dcrypt}(\mathsf{inv}(k), n)$ that does not reduce to anything. In our congruence, such terms yield $\mathtt{ff}$.

Definition III.3 (Algebraic theory, adapted from [13])**.** *A* constructor/destructor rule *is a rewrite rule of one of the following forms:*

- *Decryption:* $d(k, c(k', X_1, \ldots, X_n)) \to X_i$ *where* d *is a destructor,* c *is a constructor, the* X_j *are variables,* $c(k', X_1, \ldots, X_n)$ *is linear,* $i \in \{1, \ldots, n\}$*,* $fv(k) = fv(k')$ *and neither* k *nor* k' *contains a constant.*
- *Projection:* $d_i(c(X_1, \ldots, X_n)) \to X_i$ *where* $i \in \{1, \ldots, n\}$*,* d_i *is a public destructor called a* projector*,* c *is a constructor of arity* n*, the* X_j *are variables and* $c(X_1, \ldots, X_n)$ *is linear. There must be such a rule for every* $i \in \{1, \ldots, n\}$ *and* c *is then called* transparent*.*
- *Private extraction:* $d(c(t_1, \ldots, t_n)) \to t_0$ *where* d *is a private destructor called a* private extractor*,* c *is a constructor,* $c(t_1, \ldots, t_n)$ *is linear 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 projection rules.*

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

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

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

The theory allows for modeling usual cryptographic primitives such as asymmetric and symmetric cryptography, signatures and serialization of messages. We require that the algebraic theory is a constructor-destructor theory, not including properties like associative-commutative operators needed for Diffie-Hellman. The theory reveals destructor failure, so

honest agents can tell if a destructor like decryption fails and in this case just abort the transaction. Compared to the definition in [13], we have included the requirement of linearity for the constructor terms in the rewrite rules and excluded constants in decryption keys, and this will be used when proving the typing result for state transitions (see Definition V.2).

In a protocol specification, we write type annotations with a colon, i.e., $t : \tau$ specificies that $\Gamma(t) = \tau$. We further define what it means for a protocol specification to "type check". This does not yet include all the requirements for type-flaw resistance but simply ensures that the type annotations are consistent throughout the specification.

Definition III.4 (Type checking)**.** *For every constant c, one has to specify $\Gamma(c)$, i.e., the type of that constant. For every memory cell $\mathsf{cell}(\cdot)$, one has to specify $\Gamma(\mathsf{cell})$ which is the type of the argument for cell reads. The type annotations of constants and memory cells are global to the specification, while type checking a transaction uses local type annotations for the variables bound in that transaction. Every transaction must satisfy the following:*

- *For every choice $x \in D$, we have that D is a set of public constants of the same atomic type τ, and we then set $\Gamma(x) = \tau$.*
- *For every message received $\mathsf{rcv}(X : \tau)$, we have that τ is a type and we then set $\Gamma(X) = \tau$.*
- *For every destructor application $\mathsf{try}\ Y := d(t, X)$, where the rewrite rule for d is $d(k, c(k', X_1, \ldots, X_n)) \to X_i$, there exist types for the free variables of the left-hand side such that $\Gamma(d(k, c(k', X_1, \ldots, X_n)) = \Gamma(d(t, X))$. We then set $\Gamma(Y) = \Gamma(X_i)$.*
- *For every cell read $X := \mathsf{cell}(s)$, we have $\Gamma(s) = \Gamma(\mathsf{cell})$ and we then set $\Gamma(X) = \Gamma(C[s])$, where $C[\cdot]$ is the ground context for the initial value of $\mathsf{cell}(\cdot)$. For every cell write $\mathsf{cell}(s) := t$, we have $\Gamma(s) = \Gamma(\mathsf{cell})$ and $\Gamma(t) = \Gamma(C[s])$.*
- *For every equality $s \doteq t$ in a formula, we have $\Gamma(s) = \Gamma(t)$.*
- *For every step $\nu n_1 : \tau_1, \ldots, n_k : \tau_k$, the τ_i are atomic types and we then set $\Gamma(n_i) = \tau_i$.*

In the rest of the paper, we will only consider protocol specifications such that the type checking requirements above are satisfied.

We finally introduce several requirements on protocols, which we use to ensure that the intruder knows the types of the messages in their knowledge and to control the shapes of messages that can occur during the protocol execution.

Definition III.5 (Requirements)**.** *For some control flow requirements on transactions, consider the tree that is induced by the conditionals of the transactions (i.e., every if-then-else is a node with the respective subprocesses as children). We say two execution paths are* statically distinguishable *for the intruder, iff a different number of messages are sent along the paths.*[2] *Every transaction must satisfy the following:*

- *For any two execution paths that are not statically distinguishable (and thus have the same number of sent messages), and under any instantiation of the intruder variables (including ill-typed instantiations), the ith message sent in either path has the same type.*
- *In every cell write $\mathsf{cell}(s) := t$, the term t does not contain intruder variables.*
- *When a decryption destructor is applied to a variable, this variable does not occur in other destructor applications.*
- *If several projectors or several private extractors are applied to the same variable, then the rewrite rules for these destructors are defined over the same constructor term.*
- *For every message sent $\mathsf{snd}(t)$ and every subterm t' of t, if t' is composed with a constructor c occurring in a decryption rule $d(k, c(k', X_1, \ldots, X_n)) \to X_i$, we have that t' is an instance of $c(k', X_1, \ldots, X_n)$.*

Remark III.1. In a protocol that satisfies the requirements of Definition III.5, we have the invariant that the intruder knows the type of every message in their knowledge. Initially, the property holds because the intruder has not observed any message yet. Then whenever a transaction is receiving, the message is determined by the intruder and thus, if the intruder before the transaction knows the type of every message in their knowledge, then they know the types of the messages the transaction receives. They also know the type of every other variable in the transaction, because privacy variables are chosen from homogeneous domains, the type of messages in the memory cells never changes (only the content can), and the result of a destructor application has the type of a subterm of the input (if it does not fail anyway). Since destructor applications are in left processes and behave as 0 in case of failures, if any destructor fails then the entire transaction behaves as 0, i.e., it terminates immediately. Thus the intruder can determine the type of every message sent in a given execution path. Moreover, the intruder can observe how many messages the process sends and rule out all those execution paths that are not compatible with that. By the requirements, the remaining execution paths, being not statically distinguishable, must have the same type for corresponding messages for any given input messages from the intruder. Thus, the intruder may not know which of the remaining execution paths is the case, but they still know which types the respective messages have, so also after the transaction the intruder knows the type of every message in their knowledge. ◁

An example for a protocol that does not satisfy the first requirement immediately, i.e., that messages on two paths either are statically distinguishable or have the same type, is the model of Private Authentication found in [15]: here an agent B receives a message and performs a check on it. If

[2]One could use here a finer distinction criterion, but with a coarser relation one errs on the safe side as it excludes more protocols from being admitted.

the check succeeds, then B sends an encrypted reply as an answer. Otherwise, B sends a random nonce as a decoy to hide whether the check succeeded. Thus there are two paths where the messages sent have different types, and indeed the point is to hide from the intruder which message was really sent. In the original model by Abadi and Fournet [16], however, B instead of a random nonce as decoy sends an encrypted message with a fresh key and random contents of the same type as the positive case. In that formulation, the protocol satisfies our requirement. The only example we can think of that would resist a similar transformation are onion-routing protocols where the intruder should not be able to tell the number of encryption layers of a given message. For protocols that do not rely on hiding the taken branch from the intruder, one can of course easily make the messages of the branches statically distinguishable and thus can also use messages of different types.

The restriction on cell writes is significant, because it essentially means that we cannot update the memory with an arbitrary message sent by the intruder. Indeed, if the intruder was able to send some message to a transaction that writes this message in memory without doing any checks on it, then we could not maintain the invariant that the intruder always knows the types of the messages they observe.

The other requirements in Definition III.5 are not directly about the intruder knowing the types of the messages. We consider the requirements on the use of destructors as a reasonable restriction that ensures compatible destructor applications: whenever a variable is decomposed, we can instantiate the variable with a unique corresponding constructor term, because for this decryption there is a unique rewrite rule or for these projections/private extractions all rewrite rules are defined over the same term. The requirement on the use of constructor terms in messages sent will be useful when proving the well-typedness of analysis: if a subterm in a message sent is composed with a constructor that can be decomposed, it should be an instance of the constructor term in the corresponding rewrite rule. For instance, if we model signatures with the rewrite rule $\mathsf{open}(K, \mathsf{sign}(\mathsf{inv}(K), M)) \to M$, then signatures sent by honest agents must have a key starting with inv and cannot use a variable in this place, e.g., we do not allow sending $\mathsf{sign}(X, m)$, because $\mathsf{sign}(X, m)$ is not an instance of $\mathsf{sign}(\mathsf{inv}(K), M)$.

B. Message Patterns

To show the typing result, it is convenient to replace the try mechanism for handling destructors by pattern matching. In fact, the original (α, β)-privacy does not have a notion of pattern matching, because in a general untyped model, it is unclear how to define the semantics of such a construct in a suitable way. However, for a specification that satisfies the above restrictions, the intruder knows the type of every message, and thus also knows whether a given message will agree with a given pattern. Hence, we make a conservative extension of the receive construct with pattern matching (under the restrictions of Definition III.5).

Instead of $\mathsf{rcv}(X)$ for an intruder variable X, we now allow also $\mathsf{rcv}(t)$ where t is a *linear pattern term*: it contains fresh intruder variables, where each intruder variable can only occur once, and no constants. The meaning is that the agent only accepts an incoming ground message m, if m is an instance of t and then binds the variables of t with the respective subterms of m. (This ignores how an agent would be able to check that m is an instance of t.) We give a formal definition in Appendix A as a conservative extension of the semantics on ground states of (α, β)-privacy in [12]. In a nutshell, the original semantics transforms $\mathsf{rcv}(X).P$ for an incoming intruder message m into $P[X \mapsto m]$ while the extended semantics transforms $\mathsf{rcv}(t).P$ for message m into $\sigma(P)$ if $\sigma = mgu(m \doteq t) \neq \bot$ and into 0 otherwise.[3]

The idea is now that we can replace try by pattern matching and a condition, because the intruder already knows the type of every message in their knowledge and thus knows whether the messages they send will have the correct structures for every destructor application to succeed. (Of course, a message with the correct structure can still fail if it does not have the right key for instance.) Consider, for instance,

$$\mathsf{rcv}(X).\mathsf{try}\ Y := \mathsf{dscrypt}(k, X)\ \mathsf{in}\ \mathsf{try}\ Z := \mathsf{proj}_1(Y)\ \mathsf{in}\ P.$$

If the intruder sends for X any term that is not of the form $\mathsf{scrypt}(K, \mathsf{pair}(M, N))$ (for some K, M, and N) the destructors are going to fail. Thus we can split the try's into a structural check that we can describe by a linear pattern like $\mathsf{scrypt}(K, \mathsf{pair}(M, N))$ and a condition on the pattern variables. This transformation allows us to get rid of destructors in processes entirely. For the example, the transformation is

$$\mathsf{rcv}(\mathsf{scrypt}(K, \mathsf{pair}(M, N))).\mathsf{if}\ K \doteq k\ \mathsf{then}\ P'$$

where $P' = P[Y \mapsto \mathsf{pair}(M, N), Z \mapsto M]$. More generally:

Definition III.6 (Removing destructors)**.** *Let P be a transaction, from a protocol satisfying Definition III.5, that contains a destructor application for decryption, i.e., $P = C[\mathsf{try}\ Y := d(t, X)\ \mathsf{in}\ P']$ for some process context $C[\cdot]$ that does not contain any destructor applications. Let $d(k, c(k', X_1, \ldots, X_n)) \to X_i$ be the corresponding rewrite rule with all variables freshly renamed. Let $\sigma = [X \mapsto c(k', X_1, \ldots, X_n), Y \mapsto X_i]$. Then we replace the transaction P with $\sigma(C[\mathsf{if}\ t \doteq k\ \mathsf{then}\ P'])$.*

In case of projectors or private extractors, X may appear in m destructor applications: we have $\mathsf{try}\ Y^j := d^j(X)\ \mathsf{in}\ \ldots$, $j \in \{1, \ldots, m\}$, and rewrite rules of the form $d^j(c(t_1, \ldots, t_n)) \to t^j$. Then we remove all destructor applications for X since there are no keys, and we apply the substitution $[X \mapsto c(t_1, \ldots, t_n), Y^1 \mapsto t^1, \ldots, Y^m \mapsto t^m]$ to the transaction.

[3]The actual definition is more complicated since we model a symbolic execution by the intruder where we have several possibilities $\mathsf{rcv}(t).P_i$ and different frames $struct_i$ representing the intruder knowledge in each possibility, the intruder choosing one recipe r over the domain of $struct_i$ and $m = struct_i(r)$.

This transformation is repeated until the transaction does not contain any destructor application anymore. The result is denoted P_{pat}.

Example III.1. Consider a protocol in which a server chooses an agent and makes a binary decision, then they receive a message, try to decrypt it and send an encrypted reply containing the decision and a nonce from the received message. We omit type annotations for brevity but we will continue with this example later. We now apply the transformation to find the message patterns in the following transaction P:

$$\begin{aligned}
&\star\ x \in \mathsf{Agent}. \star\ y \in \{\mathsf{yes}, \mathsf{no}\}.\\
&\mathsf{rcv}(M).\\
&\mathsf{try}\ N := \mathsf{dcrypt}(\mathsf{inv}(\mathsf{pk}(\mathsf{s})), M)\ \mathsf{in}\\
&\mathsf{try}\ N_1 := \mathsf{proj}_1(N)\ \mathsf{in}\\
&\mathsf{try}\ N_2 := \mathsf{proj}_2(N)\ \mathsf{in}\\
&\mathsf{if}\ y \doteq \mathsf{yes}\ \mathsf{then}\\
&\quad \nu r.\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, N_1), r))\\
&\mathsf{else}\ \nu r.\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{no}, N_2), r))
\end{aligned}$$

The first step is to remove try $N := \mathsf{dcrypt}(\dots)$ with the substitution $[M \mapsto \mathsf{crypt}(X, Y, Z), N \mapsto Y]$, and the second step is to remove both projections with the substitution $[Y \mapsto \mathsf{pair}(Y_1, Y_2), N_1 \mapsto Y_1, N_2 \mapsto Y_2]$. We now have P_{pat}:

$$\begin{aligned}
&\star\ x \in \mathsf{Agent}. \star\ y \in \{\mathsf{yes}, \mathsf{no}\}.\\
&\mathsf{rcv}(\mathsf{crypt}(X, \mathsf{pair}(Y_1, Y_2), Z)).\\
&\mathsf{if}\ \mathsf{inv}(\mathsf{pk}(\mathsf{s})) \doteq \mathsf{inv}(X)\ \mathsf{then}\\
&\quad \mathsf{if}\ y \doteq \mathsf{yes}\ \mathsf{then}\\
&\qquad \nu r.\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, Y_1), r))\\
&\quad \mathsf{else}\ \nu r.\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{no}, Y_2), r))
\end{aligned}$$ ◁

Lemma III.1. *A protocol satisfying Definition III.5 and its transformation to use pattern matching according to Definition III.6 yield the same set of reachable ground states (up to logical equivalence of the contained formulas α and β).*

We now define how to compute the message patterns from a protocol specification using the P_{pat} version of transactions:

Definition III.7 (Protocol message patterns)**.** *For a protocol transaction P, we define $patterns(P)$ as the set of terms occurring in P_{pat}. For a memory cell $\mathsf{cell}(\cdot)$, we define the message pattern cell_{pat} as the message $C[X]$, where $C[\cdot]$ is the ground context for the initial value of $\mathsf{cell}(\cdot)$ and X is a variable of type $\Gamma(\mathsf{cell})$, i.e., the argument type for the cell. For a protocol Spec, we define $patterns(Spec)$ as the union of the $patterns(P)$ for every transaction P and of the cell_{pat} for every $\mathsf{cell}(\cdot)$ in the specification (up to α-renaming of variables so they are distinct in each transaction/cell).*

Example III.2. Continuing Example III.1, we write the type annotations in transaction P, where we assume that we have three atomic types agent, decision and nonce. Every constant in the set Agent and the constant s are of type agent, and the constants yes, no are of type decision.

$$\begin{aligned}
&\star\ x \in \mathsf{Agent}. \star\ y \in \{\mathsf{yes}, \mathsf{no}\}.\\
&\mathsf{rcv}(M : \mathsf{crypt}(\mathsf{pk}(\mathsf{agent}), \mathsf{pair}(\mathsf{nonce}, \mathsf{nonce}), \mathsf{nonce})).\\
&\mathsf{try}\ N := \mathsf{dcrypt}(\mathsf{inv}(\mathsf{pk}(\mathsf{s})), M)\ \mathsf{in}\\
&\mathsf{try}\ N_1 := \mathsf{proj}_1(N)\ \mathsf{in}\\
&\mathsf{try}\ N_2 := \mathsf{proj}_2(N)\ \mathsf{in}\\
&\mathsf{if}\ y \doteq \mathsf{yes}\ \mathsf{then}\\
&\quad \nu r : \mathsf{nonce}.\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, N_1), r))\\
&\mathsf{else}\ \nu r : \mathsf{nonce}.\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{no}, N_2), r))
\end{aligned}$$

This corresponds to the message patterns

$$\begin{aligned}
patterns(P) = \mathsf{Agent} \cup \{&x, y, r, \mathsf{yes}, \mathsf{no}, \mathsf{inv}(\mathsf{pk}(\mathsf{s}))\}\\
\cup \{&\mathsf{inv}(X), \mathsf{crypt}(X, \mathsf{pair}(Y_1, Y_2), Z),\\
&\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, Y_1), r),\\
&\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{no}, Y_2), r)\}
\end{aligned}$$

where x is of type agent, y is of type decision, X is of type $\mathsf{pk}(\mathsf{agent})$ and r, Y_1, Y_2, Z are of type nonce. ◁

C. Type-Flaw Resistance

The core part in the proof of our typing result is that variables can always be instantiated with messages of the same type. We first define the set of sub-message patterns, which includes all subterms, well-typed instantiations and key terms. To prove our result we will use the fact that every message in the symbolic execution of the protocol is in this set of sub-message patterns.

Definition III.8 (Sub-message patterns)**.** *The set of* sub-message patterns*, $SMP(M)$, of a set of terms M is the least set closed under the following rules:*

1) *If $t \in M$, then $t \in SMP(M)$.*
2) *If $t \in SMP(M)$ and t' is a subterm of t, then $t' \in SMP(M)$.*
3) *If $t \in SMP(M)$ and σ is a well-typed substitution, then $\sigma(t) \in SMP(M)$.*
4) *If $t \in SMP(M)$, k and t' are terms such that for some destructor d we have $d(k, t) \to t'$ as an instance of the rewrite rule for d, then $k \in SMP(M)$.*

With rule 4, we ensure that relevant decryption keys are in $SMP(M)$, because they may occur in the symbolic constraints when performing analysis steps.

We have now everything in place to formally define type-flaw resistance, which ensures that composed messages of different types cannot be unified.

Definition III.9 (Type-flaw resistance)**.** *A set of terms M is* type-flaw resistant *iff for all $s, t \in SMP(M) \setminus \mathcal{V}$ we have that $\Gamma(s) = \Gamma(t)$ if s and t are unifiable.*

A protocol Spec is type-flaw resistant *iff it satisfies Definition III.5 and the set $patterns(Spec)$ is type-flaw resistant.*

Remark III.2. Even though the set $SMP(patterns(Spec))$ is infinite in general, the condition for type-flaw resistance can be checked automatically using a finite representation. $\lhd$

Example III.3. The protocol from Example III.2 is not type-flaw resistant, because in the message patterns we have the input pattern $\mathsf{crypt}(X, \mathsf{pair}(Y_1, Y_2), Z)$ and an output pattern, e.g., $\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, Y_1), r)$, which can be unified even though they have different types: the mgu $[X \mapsto \mathsf{pk}(x), Y_1 \mapsto \mathsf{yes}, Y_2 \mapsto \mathsf{yes}, Z \mapsto r]$ is not well-typed since $\Gamma(Y_i) \neq \Gamma(\mathsf{yes})$.

One can make the protocol type-flaw resistant by using formats, for instance by replacing the function pair with f_1 in the input and with f_2 in the outputs, where the f_i are transparent functions (this requires also replacing the projectors in the process). $\lhd$

IV. Typing Result for Constraint Solving

Like many other works for checking security goals, the decision procedure for (α, β)-privacy [13] uses symbolic constraints to represent the messages sent by the intruder. To solve the constraints, the variables standing for the intruder-generated messages are instantiated in a demand-driven way, and this part of the procedure is called the *lazy intruder*. Our main result Theorem V.1 says that, for type-flaw resistant protocols, whenever the intruder is sending a message, we can without loss of generality consider that the message is well-typed. Thus, for the typing result to hold, we need to make sure that the solutions computed with the lazy intruder always perform well-typed instantiations.

The lazy intruder rules defined below compute the different recipes that can be used by the intruder in order to solve the constraints in the FLICs. As a preparation, we give here the definition of a *choice of recipes* and how to apply it to a simple FLIC, i.e., once the intruder is instantiating some recipes, the FLICs are updated to instantiate the recipe variables and corresponding intruder variables. Note that applying a choice of recipes is only defined for simple FLICs: whenever there are constraints to solve, this is done with the lazy intruder for just one FLIC and afterwards the results are applied to all the FLICs in the symbolic states (they are always simple).

Definition IV.1 (Choice of recipes [13])**.** *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 $R_1, \ldots, R_n$ be the fresh variables in r, i.e., $\{R_1, \ldots, R_n\} = rvars(r) \setminus rvars(\mathcal{A})$, taken in a fixed order (e.g., the order in which they first occur in r). Let $X_1, \ldots, X_n$ 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 $\mathcal{A}' = \mathcal{A}_1.{+}R_1 \mapsto X_1.\cdots.{+}R_n \mapsto X_n$ 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 $\mathcal{A}$.

Example IV.1. Consider the FLIC $\mathcal{A} = {+}R \mapsto M$. Let ρ be the choice of recipes $[R \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{pair}(R_1, R_2), R_3)]$. Then we remove the mapping for R and M, and we introduce new mappings with fresh intruder variables, so we have $\rho(\mathcal{A}) = {+}R_1 \mapsto X_1.{+}R_2 \mapsto X_2.{+}R_3 \mapsto X_3$. The substitution $[M \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{pair}(X_1, X_2), X_3)$ is also applied, but here there were no messages received containing M anyway. $\lhd$

The decision procedure actually considers at first an intruder who cannot apply destructors (and thus cannot analyze messages) but only constructors. The analysis is actually performed as a later step. Thus, the constraint solving with the lazy intruder works in the free algebra.

The lazy intruder reduces symbolic constraints expressed as a FLIC until all messages to send are distinct intruder variables. This reduction is defined as a set of rules, covering the different ways the intruder is able to send a message.

Definition IV.2 (Lazy intruder rules [13])**.** *The relation $\rightsquigarrow$ is a relation on triples $(\rho, \mathcal{A}, \sigma)$, where $\mathcal{A}$ is a FLIC, ρ is a choice of recipes such that $dom(\rho) \cap rvars(\mathcal{A}) = \emptyset$ and σ is a substitution such that $dom(\sigma) \cap vars(\mathcal{A}) = \emptyset$.*

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

The Unification rule means that to produce an outgoing term t, the intruder can use any term s previously received, if s and t are unifiable and s and t are not variables. If t is a variable it means that, at least at this moment, we have no constraints on t and the intruder can send any message they can construct. The lazy intruder technique avoids the blind and pointless exploration by simply leaving the variable as is as long as it is not substituted. If s is a variable, it means that s is a message sent by the intruder earlier that a transaction returned directly and thus also does not need to be considered. The Unification rule is in fact where the typing result is latching in: if the protocol is type-flaw resistant and s and t are not variables but unifiable, then they must have the same type and their most-general unifier thus be well-typed.

The other rules are just briefly explained: Composition means that the intruder can produce $f(t_1, \ldots, t_n)$ if f is public and the t_i are producible. When the intruder has to produce a privacy variable x, the Guessing rule models any guess that the intruder can take for it (right or wrong). The Repetition rule says that if the constraint for the intruder is to send the same message as a previous one, we can take the same recipe as the previous one. The soundness, completeness, and termination of the procedure, i.e., that from a given FLIC, $\rightsquigarrow$ reaches finitely many simple FLICs that cover the same set of solutions as the given FLIC, is proved in [13].

To see that the lazy intruder never performs ill-typed solutions, first observe that every lazy intruder rule preserves the well-typedness of the substitution σ of variables performed in previous lazy intruder reduction steps. Let $terms(\mathcal{A}) = \{t \mid -l \mapsto t \in \mathcal{A} \text{ or } +R \mapsto t \in \mathcal{A}\}$ be the set of terms occurring in a FLIC $\mathcal{A}$.

Lemma IV.1. *Let Spec be a type-flaw resistant protocol, $\mathcal{A}$ be a FLIC such that $terms(\mathcal{A}) \subseteq SMP(patterns(Spec))$, ρ be a choice of recipes such that $dom(\rho) \cap rvars(\mathcal{A}) = \emptyset$, σ be a well-typed substitution such that $dom(\sigma) \cap vars(\mathcal{A}) = \emptyset$, and $(\rho', \mathcal{A}', \sigma')$ be such that $(\rho, \mathcal{A}, \sigma) \rightsquigarrow (\rho', \mathcal{A}', \sigma')$. Then σ' is well-typed.*

The reduction with $\rightsquigarrow$ defines how to simplify FLICs. In the transition system, only the choices of recipes coming out of this simplification are used: the lazy intruder solves constraints and then the results are applied to a symbolic state.

Definition IV.3 (Lazy intruder results [13])**.** *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 IV.2. Suppose that the transaction from Example III.1 has been executed once already, where the intruder has sent $\mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{pair}(X_1, X_2), X_3)$ for the input, and that the transaction is executed a second time. Let $\mathcal{A} = +R_1 \mapsto X_1.+R_2 \mapsto X_2.+R_3 \mapsto X_3.-l \mapsto \mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, X_1), \mathsf{r}).+R \mapsto M$ be a simple FLIC. The label l corresponds to the reply from the server and the mapping $+R \mapsto M$ is the second input.

Let $\sigma = [M \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{pair}(Y_1, Y_2), Y_3)]$ represent constraints to solve. With the lazy intruder rules, we get the results $LI(\mathcal{A}, \sigma) = \{\rho_1, \rho_2\}$, where $\rho_1 = [R \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{pair}(R_4, R_5), R_6)]$ and $\rho_2 = [R \mapsto l]$. That is to say, either the intruder composes a message themselves or they reuse the message they received. Then applying the choice of recipes is done on the simple FLICs: $\rho_1(\mathcal{A}) = +R_1 \mapsto X_1.+R_2 \mapsto X_2.+R_3 \mapsto X_3.-l \mapsto \mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, X_1), \mathsf{r}).+R_4 \mapsto X_4.+R_5 \mapsto X_5.+R_6 \mapsto X_6$ and the substitution $[M \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{pair}(X_4, X_5), X_6)]$ is also applied in the process. This is done similarly for the FLIC where the message received contains no.

For the case where ρ_2 is applied instead, in the FLICs the last mapping is simply removed and then the substitution would be different in each possibility since the label maps to different messages: in one possibility we substitute $[M \mapsto \mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, X_1), \mathsf{r})]$ and in the other $[M \mapsto \mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{no}, X_2), \mathsf{r})]$. ◁

We extend the notion of well-typed substitutions to well-typed choices of recipes:

Definition IV.4 (Well-typed choice of recipes)**.** *Let $\mathcal{A}$ be a simple FLIC and ρ be a choice of recipes for $\mathcal{A}$. We say that ρ is* well-typed w.r.t. *$\mathcal{A}$ iff for every $+R \mapsto X \in \mathcal{A}$, we have $\Gamma(X) = \Gamma(\rho(\mathcal{A})(\rho(R)))$.*

Example IV.3. Continuing Example IV.2, note that $\rho_2 = [R \mapsto l]$ is not well-typed w.r.t. $\mathcal{A}$, because we have $\Gamma(M) = \mathsf{crypt}(\mathsf{pk}(\mathsf{agent}), \mathsf{pair}(\mathsf{nonce}, \mathsf{nonce}), \mathsf{nonce})$ but $\Gamma(\rho_2(\mathcal{A})(l)) = \mathsf{crypt}(\mathsf{pk}(\mathsf{agent}), \mathsf{pair}(\mathsf{decision}, \mathsf{nonce}), \mathsf{nonce})$. This is exactly the type-flaw vulnerability illustrated in Example III.3. If formats are added to achieve type-flaw resistance, i.e., pair in the input is replaced with f_1 and pair in the output is replaced with f_2, then $\rho_2 = [R \mapsto l]$ would not be returned by the lazy intruder. However, the lazy intruder composing the message themselves would still be a solution and the result would be $\rho_1 = [R \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{f}_1(R_4, R_5), R_6)]$. We then have $\rho(\mathcal{A})(\rho(R)) = \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{f}_1(X_4, X_5), X_6)$, where the variables X_4, X_5, X_6 are of type nonce, so ρ_1 is well-typed w.r.t. $\mathcal{A}$. ◁

We can now conclude that the lazy intruder results are doing only well-typed instantiations.

Theorem IV.1 (Lazy intruder well-typedness)**.** *Let Spec be a type-flaw resistant protocol, $\mathcal{A}$ be a simple FLIC such that $terms(\mathcal{A}) \subseteq SMP(patterns(Spec))$ and let σ be a well-typed substitution. Then every $\rho \in LI(\mathcal{A}, \sigma)$ is well-typed w.r.t. $\mathcal{A}$.*

V. Typing Result for State Transitions

In the transition system considered, each state is *symbolic* in that the privacy and intruder variables are used to represent infinitely many ground states that are instances of the symbolic states. In each symbolic state, we have formulas to represent the payload and the intruder deductions, a set of possibilities and a set of pairs (label, recipe) to keep track of the intruder experiments (comparisons between two recipes) already performed. In each possibility, there is a process for the steps of a transaction that remain to be executed, a formula representing the conditions under which the possibility can be reached, a FLIC (lifting the frame that would be in a ground state), some disequalities formula over intruder variables (for the cases where the intruder is not solving constraints), a formula for the releases done in that possibility and finally a sequence of memory updates.

Definition V.1 (Symbolic state [13])**.** *A* symbolic state *is a tuple $(\alpha_0, \beta_0, \mathcal{P}, Checked)$ 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 $(P, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)$, where P is a process, ϕ is a Σ_0-formula, $\mathcal{A}$ is a FLIC, $\mathcal{X}$ is a disequalities formula, α is a Σ_0-formula called* partial payload*, and δ is a sequence of memory updates of the form $\mathsf{cell}(s) := t$ for messages s and t;*
- *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:

$$\begin{array}{lcll} \mathcal{X} & := & \mathcal{X} \wedge \mathcal{X} \mid \forall \vec{X}.\ \neg \mathcal{X}_0 & \textit{Disequalities formula} \\ \mathcal{X}_0 & := & \mathcal{X}_0 \wedge \mathcal{X}_0 \mid s \doteq t & \textit{Equalities formula} \end{array}$$

A symbolic state is finished *iff all the processes in $\mathcal{P}$ are 0. Let $dom(\mathcal{S})$ be the domain of the FLICs in $\mathcal{S}$ (the domain is the same in every FLIC).*

[13] defines a relation $\Rightarrow$ on symbolic states, which describes how the intruder evaluates the processes in the different possibilities, and how they contrast this evaluation with their observations. They continue to show the correctness of this relation with respect to the semantics on ground states.

Since we have made an extension for pattern matching for ground states, we now define how to handle this construct for symbolic states, extending the relation $\Rightarrow$ and proving correctness for this case. To that end, we use the lazy intruder to consider all choices of recipes producing a linear pattern: the intruder can either use a label that produces a message of the same type, or compose the pattern themselves. We ensure that if a label is a solution in one FLIC, it is a solution in every FLIC. This is why the linearity requirement in rewrite rules is crucial, since the type information cannot distinguish variables of the same type. For instance, if a message $\mathsf{rcv}(f(X, X))$ was expected, it might be that in one FLIC a label l maps to $f(t, t)$ for some message t of type τ, and in another FLIC the label l maps to $f(t, s)$ where $s \neq\!\!\doteq t$ but s is still of type τ. We instead consider only linear patterns, so in the example we might have $\mathsf{rcv}(f(X, Y))$ (the transaction could still check whether $X \doteq Y$ after the receive). Similarly, we forbid constants in patterns because otherwise we cannot, using only the type information, know which value matches a pattern.

Definition V.2 (Receiving message patterns)**.** *We extend the semantics of receive steps to support linear patterns, with the following transition:*

$$\begin{array}{l} \{(\mathsf{rcv}(t).P_1, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots, \\ \ (\mathsf{rcv}(t).P_n, \phi_n, \mathcal{A}_n, \mathcal{X}_n, \alpha_n, \delta_n)\} \\ \Rightarrow \{(\sigma_1(P_1), \phi_1, \mathcal{A}'_1, \sigma_1(\mathcal{X}_1), \alpha_1, \sigma_1(\delta_1)), \ldots, \\ \qquad (\sigma_n(P_n), \phi_n, \mathcal{A}'_n, \sigma_n(\mathcal{X}_n), \alpha_n, \sigma_n(\delta_n))\} \end{array}$$

where R is a fresh recipe variable and X and a fresh intruder variable, $\rho \in LI(\mathcal{A}_1.{+}R \mapsto X, [X \mapsto t])$, $\mathcal{A}'_i = \rho(\mathcal{A}_i.{+}R \mapsto X)$, and $\sigma_i = \sigma_\rho^{\mathcal{A}_i.+R \mapsto X}$ – and every $\sigma_i(\mathcal{X}_i)$ is satisfiable.

Moreover, there is also the following transition if $t \notin \mathcal{V}_{intruder}$:

$$\begin{array}{l} \{(\mathsf{rcv}(t).P_1, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots, \\ \ (\mathsf{rcv}(t).P_n, \phi_n, \mathcal{A}_n, \mathcal{X}_n, \alpha_n, \delta_n)\} \\ \Rightarrow \{(0, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots, (0, \phi_n, \mathcal{A}_n, \mathcal{X}_n, \alpha_n, \delta_n)\} \end{array}$$

When receiving a pattern, the intruder can either reuse a label or compose the message themselves. The different choices of recipes are computed with the lazy intruder, and since a label maps to messages of the same type in every FLIC, if using a label is a solution in one FLIC, then it is a solution in every FLIC: the linear pattern t contains only fresh variables and no constants, so it can be unified with any message of the same type.

Example V.1. Continuing Example III.1, the transaction P_{pat} starts by receiving the pattern $\mathsf{rcv}(\mathsf{crypt}(X, \mathsf{pair}(Y_1, Y_2), Z)$. The intruder may compose that message themselves with the recipe $\mathsf{crypt}(R_X, \mathsf{pair}(R_{Y_1}, R_{Y_2}), R_Z)$. Another solution, assuming they have observed earlier a message $-l \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{pair}(\mathsf{n}_1, \mathsf{n}_2), \mathsf{r})$ sent by an honest agent, is to use the label l to instantiate the pattern (if there are multiple possibilities, this same label could map in other FLICs to other messages of the same type, e.g., where the nonces are different, and the substitutions are done in each process). $\lhd$

This conservative extension means that even when receiving patterns, we keep FLICs simple. The rest of the semantics is the same as in [13] (we give the full definitions in Appendix A).

Lemma V.1. *Given a type-flaw resistant specification, then the set of reachable states in the symbolic semantics represents exactly the reachable states of the ground semantics.*

During the exploration of reachable states, the intruder can perform experiments by comparing the outcome of two recipes. An important notion is that of *normal* symbolic state, meaning that the intruder has performed all relevant experiments and cannot distinguish the different possibilities anymore.

Definition V.3 (Pairs and normal symbolic state [13])**.** *Let $\mathcal{S} = (\_, \_, \mathcal{P}, Checked)$ be a symbolic state. The set of pairs of recipes to compare in $\mathcal{S}$ is*

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

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

The lazy intruder is used in two ways for the experiments: (i) to compute recipes that can be compared to labels, and (ii) to solve constraints whenever the outcome of an experiment depends on messages sent earlier. Since we have already shown that the lazy intruder results are well-typed, we have the guarantee that in a experiment with a pair (l, r), the label l and the recipe r produce messages of the same type and all

transitions to determine the outcome of an experiment are only doing well-typed instantiations. Thus, there is nothing more to show for intruder experiments.

It remains to show that analysis also never introduces ill-typed instantiations. In a normal symbolic state, the intruder can perform analysis by decrypting messages in their knowledge, if they know the appropriate key. The analysis is always done in *normal* states, i.e., after the experiments.

In [13], the analysis is performed through *destructor oracles*, which are defined as transactions available to the intruder: for every public destructor d, the corresponding oracle is the transaction $\mathsf{rcv}(K).\mathsf{rcv}(M).\mathsf{try}\ N := d(K,M)\ \mathsf{in}\ \mathsf{snd}(N).\mathsf{snd}(K)$. This transaction receives a candidate key and a message, tries to decrypt the message with the key and sends back the result if the decryption succeeded. (The oracles for projectors are omitted here as the handling is similar. Private extractors are not accessible to the intruder so there are no oracles for them.)

These destructor oracles do not work directly with the typing result: since they are defined as transactions, the computation of the sub-message patterns set SMP would need to include the patterns from the destructor oracles. This prevents us from achieving type-flaw resistance even for reasonable protocols and when formats are used. For instance, consider a protocol that uses several times crypt but with contents of different types, e.g., $\mathsf{crypt}(\mathsf{pk}(\mathsf{agent}), f_1(\mathsf{agent}), \mathsf{nonce})$ and $\mathsf{crypt}(\mathsf{pk}(\mathsf{agent}), f_2(\mathsf{nonce}), \mathsf{nonce})$. To compute the message patterns, we have to consider the transformed destructor oracle that uses pattern matching instead of try; for dcrypt, this would yield the transaction $\mathsf{rcv}(K).\mathsf{rcv}(\mathsf{crypt}(X,Y,Z)).\mathsf{if}\ K \doteq \mathsf{inv}(X)\ \mathsf{then}\ \mathsf{snd}(Y).\mathsf{snd}(K)$. We have the pattern $\mathsf{crypt}(X,Y,Z)$, because the decryption does not care about the actual content of the message but just about whether the key is correct. If we assume that there are multiple instances of this transaction where only the type annotations change (to cover all possible types), we would have $\mathsf{crypt}(X_1,Y_1,Z_1)$ and $\mathsf{crypt}(X_2,Y_2,Z_2)$ in SMP, with for instance $\Gamma(X_i) = \mathsf{pk}(\mathsf{agent})$, $\Gamma(Z_i) = \mathsf{nonce}$, $\Gamma(Y_1) = f_1(\mathsf{agent})$ and $\Gamma(Y_2) = f_2(\mathsf{nonce})$. These two message patterns are unifiable but have different types, so type-flaw resistance is not achieved.

However, the procedure from [13] does not blindly apply destructor oracles but always restrict the step $\mathsf{rcv}(M)$ to using a label l as recipe for message M, where l maps to a message composed with the top-level constructor corresponding to the oracle. Therefore, we can be more precise and specialize the processes coming out of the destructor oracles: instead of a general pattern like $\mathsf{crypt}(X,Y,Z)$, we only consider instances of that pattern with messages that the intruder has observed, e.g., $\mathsf{crypt}(\mathsf{pk}(\mathsf{a}), f_1(A), R)$, so that all terms in the FLICs remain in the set SMP.

We define analysis steps as part of the transition system instead of special transactions. We will show that, for a type-flaw resistant protocol, this alternative way of performing analysis is equivalent to using destructor oracles. The benefit of our formulation of analysis is that we ensure all messages are instances of the protocol message patterns, and thus we can obtain the typing result.

Definition V.4 (Analysis transition)**.** *Let $\mathcal{S}$ be a reachable symbolic state in a type-flaw resistant protocol. The transition for analysis is:*

$$\{(0,\phi_1,\mathcal{A}_1,\mathcal{X}_1,\alpha_1,\delta_1),\ldots,(0,\phi_n,\mathcal{A}_n,\mathcal{X}_n,\alpha_n,\delta_n)\}$$
$$\Rightarrow^{\bullet}\{(P_1,\phi_1,\mathcal{A}_1,\mathcal{X}_1,\alpha_1,\delta_1),\ldots,(P_n,\phi_n,\mathcal{A}_n.,\mathcal{X}_n,\alpha_n,\delta_n)\}$$

if $\mathcal{S}$ is normal and there exist a label $l \in dom(\mathcal{S})$ and a public destructor $d \in \Sigma_{pub}$ such that l may be analyzed with d, i.e., for every $i \in \{1,\ldots,n\}$, $-l \mapsto c(k_i', t_i^1,\ldots,t_i^m) \in \mathcal{A}_i$ where $d(k_i, c(k_i', t_i^1,\ldots,t_i^m)) \to t_i^j$ (for some $j \in \{1,\ldots,m\}$) is an instance of the rewrite rule for d and for every $i \in \{1,\ldots,n\}$, let $P_i = \mathsf{rcv}(X).\mathsf{if}\ X \doteq k_i\ \mathsf{then}\ \mathsf{snd}(t_i^j).\mathsf{snd}(k_i)$. In case c is transparent, we define $P_i = \mathsf{snd}(t_i^1).\cdots.\mathsf{snd}(t_i^m)$.

Remark V.1. The processes P_i that we put in each possibility are exactly the instances of the corresponding destructor oracle, after transformation to pattern matching and substitution of the message to analyze with the respective message that the label maps to in each FLIC. ◁

Example V.2. Continuing Example III.1, suppose the intruder is an agent with their own private key, they have sent the message $\mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{pair}(X_1,X_2), X_3)$ to the server and received a reply. The intruder knowledge is represented with two possibilities, where one of them contains the following FLIC:

$$-l_1 \mapsto \mathsf{inv}(\mathsf{pk}(\mathsf{i})).+R_1 \mapsto X_1.+R_2 \mapsto X_2.+R_3 \mapsto X_3.$$
$$-l_2 \mapsto \mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, X_1), \mathsf{r})$$

The other possibility contains a similar FLIC with $\mathsf{pair}(\mathsf{no}, X_2)$ in the reply. Applying the transition for analysis means that we execute the process $\mathsf{rcv}(X).\mathsf{if}\ X \doteq \mathsf{inv}(\mathsf{pk}(x))\ \mathsf{then}\ \mathsf{snd}(\mathsf{pair}(\mathsf{yes}, X_1)).\mathsf{snd}(\mathsf{inv}(\mathsf{pk}(x))$ (respectively $\mathsf{snd}(\mathsf{pair}(\mathsf{no}, X_2))$). Assuming the intruder does not know any other private key, the lazy intruder would return the label l_1 for instantiating the message X, which means $X = \mathsf{inv}(\mathsf{pk}(\mathsf{i}))$.

This yields two symbolic states, one in which decryption succeeded and one in which it failed. If it succeeded, the intruder would learn $x \doteq \mathsf{i}$ and receive the decrypted pair; projecting the pair, the intruder would learn whether $y \doteq \mathsf{yes}$. If it failed, they would learn $x \not\doteq \mathsf{i}$ but nothing about y. ◁

We can now consider two transition relations on states. $\Longrightarrow$ is induced by the original semantics from [13], i.e., processing a transaction with the evaluation of the process and then normalizing a state, where the transactions include the destructor oracle rules. In contrast, we define in this paper a new relation $\Longrightarrow^{\bullet}$ that replaces the destructor oracle rules by the analysis transitions $\Rightarrow^{\bullet}$ of Definition V.4.[4] The formal definitions of these relations are given in Appendix A.

[4]In this form, both transition systems admit infinite sequences of analysis steps (e.g., attempting repeatedly to decrypt the same message), but [13] shows that there is a terminating strategy to saturate the intruder knowledge after every transaction. This strategy can here be applied in the same way, but this is orthogonal to our result.

We have made two changes changes to the procedure of [13]: first, we have replaced explicit destructor applications with pattern matching (Definitions III.6 and V.2) and second, we have replaced destructor oracles with analysis transitions (Definition V.4). In Lemmas III.1 and V.1, we have already shown that, for type-flaw resistant protocols, using pattern matching instead of explicit destructor applications is correct. Thus for every transaction P in the protocol specification, we now consider that the transaction P_{pat} is executed instead of P. That way, we ensure that the messages in the symbolic constraints are always in the set of sub-message patterns SMP of the protocol. For type-flaw resistant protocols, the analysis transitions are equivalent to destructor oracles, and thus the two transition relations are the same:

Lemma V.2. *Given a type-flaw resistant protocol,* $\Longrightarrow = \Longrightarrow^{\bullet}$.

Moreover, for type-flaw resistant protocols, all instantiations performed by the transition $\Longrightarrow^{\bullet}$ are well-typed. Thus we conclude and obtain our main typing result, which holds for an unbounded number of transitions:

Theorem V.1 (Typing result)**.** *Given a type-flaw resistant protocol, it is correct to restrict the intruder model to well-typed recipes/messages for verifying privacy.*

VI. Case Studies

We use the protocols modeled in [13], [15] as case studies for our typing result: we show for each how to achieve type-flaw resistance requirements or why that is not possible in a reasonable way in case of one of the protocols. We summarize our results here; the models, together with more details, are in Appendix C.

a) Basic Hash [17]: In this RFID protocol, a tag sends to a reader a pair of messages containing a nonce and a MAC, using a secret key shared between tag and reader. Then the reader tries to recompute the MAC with every secret key they know to identify the tag (this behavior of the reader is modeled with a private extractor that retrieves the tag name from the MAC). Basic Hash is type-flaw resistant, where for the type annotations, we consider that we have the following atomic types: tag, used for the names of the tags and the privacy variable representing some tag name; nonce, used for the fresh number created by the tag; and ok, used for the reply from the reader when identification succeeds.

b) OSK [18]: This protocol is out of the scope of our typing result. In OSK, similarly to Basic Hash, a reader tries to identify a tag. However, in OSK, both the tag and the reader use memory cells as ratchets (initialized with a shared secret), instead of a MAC. The processes contain steps like $S := \mathsf{cell}[x].\mathsf{cell}[x] := \mathsf{h}(S)$ representing a turn of the ratchet with the application of a hash function, and thus the updates change the type of the content stored in memory, which is not allowed by Definition III.4.

c) BAC [19]: This standard RFID protocol is used to read data from passports. A tag and a reader perform a challenge-response, where the tag sends a nonce and an encrypted message containing that nonce, and the reader receives both and verifies that the nonces match. In the model from [15], there is a non-empty catch branch and thus it violates our requirements. However, the interesting aspect of the try in this case—namely that it can reveal whether it is the right agent—is independent of whether it is an encryption in the first place (which the intruder knows). Thus with the pattern matching notation introduced in this paper, we can equivalently formulate this as a pattern match and an if condition with a non-empty else branch and achieve the requirements of type-flaw resistance.[5]

d) Private Authentication: This protocol from [16] models agents that encrypt messages using a public-key infrastructure. The initiator sends a message containing their name and a nonce, and the responder either sends back a message with a fresh nonce or sends a decoy message. The model of [15] violates our requirements in three regards. First, the decoy message is a fresh nonce, while a normal reply is an encrypted message. It is intended that the intruder in general cannot tell which one is the case, violating our requirement that in this case the messages must have the same type. However, the original model from [16] actually ensures that the decoy message is of the same type as the regular message: it is an encryption of a fresh nonce with a fresh key. Following this, the requirement is actually met, as the intruder now in each case knows the type of each message (just not whether its content and key are dummy or regular). Second, there are non-empty catch branches which however can now be solved using our pattern matching notation as in the case of BAC. Third, the message from the initiator and the reply from the responder are unifiable but do not have the same type. Like for Runex, we can use formats to solve this third issue and thus achieve type-flaw resistance.

In Table I, we report the execution time of the noname tool, where we compare the models from [13], [15] to our models that include reasonable adaptations to achieve type-flaw resistance. In all cases, we only considered the variants of the protocols that do not have any privacy violation (at least until the bounds verified). For Private Authentication, the variant where agents always want to talk to other agents is denoted AF0, while the variant where agents might not want to talk to some other agents is denoted AF.

For BAC and Private Authentication, we have been able to solve the issue of non-empty catch branches by using pattern matching. Thus, one may wonder if we could not do that in general and drop some restrictions on our typing result. In fact here is an example that we would not be able to transform to pattern matching:

$$\begin{aligned}&\mathsf{rcv}(X).\\&\mathsf{try}\ Y := \mathsf{dscrypt}(k, X)\ \mathsf{in}\ \mathsf{snd}(\mathsf{h}(Y))\\&\mathsf{catch}\ \mathsf{snd}(\mathsf{sign}(\mathsf{inv}(pk), X))\end{aligned}$$

[5]Currently the noname tool of [15] does not support the pattern matching notation, but it can be simulated using private extractors.

TABLE I
EVALUATION OF EXECUTION TIME FOR CASE STUDIES

Protocol	Bound	Untyped	Type-flaw resistant	Ratio
Basic Hash	4	1.87s	1.87s	1
BAC	4	1.08s	1.08s	1
AF0	2	5.88s	4.05s	1.45
AF0	3	4min38.12s	2min45.84s	1.68
AF	2	10.24s	7.63s	1.34
AF	3	12min47.76s	8min51.26s	1.45

Untyped = models from [13], [15]
Type-flaw resistant = models described in this paper
Machine used: laptop with i7-4720HQ @ 2.60GHz, 8GB RAM
GHC 9.8.1, cvc5 1.1.1

where the message in the catch is a signed (error) message on the input X. Even if the intruder knows a priori that a particular message is not decipherable, obtaining the signature on it may be relevant in an attack that cannot be done in a well-typed way.

VII. RELATED WORK AND CONCLUSION

Why a typing result? There are in our view four benefits: robust engineering, efficiency, decidability, and simplifying interactive theorem proving. First, *robust engineering*: we spend a few extra bits (if not already present) to explicitly say what messages *mean* and thereby "solve" type-flaw attacks. In fact, the intruder can still take an encrypted message and send it in a place where a nonce is expected (thus still sending an "ill-typed" message), but due to the clear annotation of the meaning, every honest agent will always treat this bitstring as a nonce and never try to decrypt it. Hence, if there is an attack, the same attack would work if the intruder had indeed sent a random nonce, and it is thus sound to consider an intruder model with only well-typed messages.

This leads to *efficiency*. The first typing result was by Heather et al. [2] and supports the Casper tool based on the model checker FDR2 to explore the state space. This requires bounds on the number of steps honest agents and the intruder can perform; restricting the intruder to well-typed messages drastically cuts down the search space. Similarly, the model checker SATMC of the AVISPA Tool and AVANTSSAR Platform requires a typed model [20], [21], [22]. The result of Heather et al. [2] and several that followed are based on inserting tags into messages. This has a disadvantage when we consider existing protocols, say TLS, that do use some tagging but do not follow the precise tagging scheme of the typing result in question—then that result is simply not applicable. We follow the approach of Hess and Mödersheim [7] and model the concrete formatting of messages in a protocol implementation by using transparent functions, where different functions represent disjoint formats in the implementation. This style of typed model is compatible, e.g., with TLS 1.2. Several results have shown how to apply typing result to larger classes of protocols and properties, e.g., Arapinis and Duflot [4] show how to extend beyond secrecy goals, and Hess and Mödersheim [7] how to extend to stateful protocols.

For several typing results, including the present one, the proof is based on a constraint-based representation of protocol executions, sometimes called the lazy intruder. The core of the proof is to show that the constraint solving procedure for the lazy intruder constraints never performs an ill-typed substitution when applied to a constraint that originates from a type-flaw resistant protocol. Originally, the lazy intruder was however devised not as a proof technique but as a symbolic model-checking technique, namely in tools like OFMC and CL-Atse of AVISPA and AVANTSSAR [23], [24], [25], [21], [22], and the noname tool for (α, β)-privacy [13]. It is in the nature of the matter that these tools, for a type-flaw resistant protocol, will not consider any ill-typed messages, so the restriction to a typed model does not further cut down the size of the state space they explore.

The mentioned model-checking approaches are concerned with bounded number of steps of the honest agents. However, for verifying protocol security without such bounds, one of the most popular tools is ProVerif [26], based on abstract interpretation, basically abstracting the fresh messages into a coarser set of abstract values, while maintaining unbounded steps of both honest agents and intruder. This is in general still undecidable, but with a typed model, it becomes decidable as shown in [3]: essentially, we will have finitely many equivalence classes and thereby a finite set of well-typed messages that can occur in the saturation of Horn clauses that represent "what can happen". A similar tool based on abstract interpretation is PSPSP [27], which relies on a typed model and computes a finite fixedpoint for stateful protocols. While the abstraction is in general an over-approximation, PSPSP implements a decision procedure for the resulting abstraction under a typed model.

There are currently no tools and methods for (α, β)-privacy that perform verification for an unbounded number of sessions; therefore we currently cannot demonstrate how a typed model can help here and possibly allow for a decision procedure here, as well, but this seems very likely.

Finally, concerning interactive theorem proving, the first results in Isabelle/HOL by Paulson [28] in fact use a typed model (without any typing result). It underlines how the typed model allows for easier reasoning than dealing with ill-typed messages in manual proofs. Similarly, the compositionality result of Hess et al. [29] in Isabelle relies on typed model. We envision that a similar compositionality result is possible for (α, β)-privacy and will also largely benefit from a typed model.

A major challenge, and in fact the focus of this paper, is to give a typing result for privacy-type properties, where the most common approaches work with models based on trace-equivalence. Chrétien et al. [8], [30] are, to our knowledge, the only major results for this question, and thus also the related work closest to ours. Since our approach is based on (α, β)-privacy, it plays a quite different game but results in [12, §V] suggest that the two notions have similar expressive power.

Our work is more general than Chrétien et al. in the following three regards. First, they require that protocols are deterministic and they do not support if-then-else branching. In contrast, we allow non-deterministic choice of privacy variables by honest agents and if-then-else with conditions that can refer to all messages in scope (including privacy variables). This generalization is significant because it allows for protocols where the privacy also depends on the control flow, e.g., where the intruder does not know whether a recipient accepted a message (and sent a legitimate answer) or not (and sent a dummy answer). Note also that a common restriction for verification tools is the notion of diff-equivalence which (at least in its original form) forbids dependence on conditions.

A second generalization is the handling of constructors and destructors. [30] does not model destructors in the processes (only in the intruder model) and rather obtains decryption by pattern matching. We instead support the explicit application of destructors by honest agents that (α, β)-privacy uses in try-catch statements, where we only require the catch branch to be the nil process, i.e., honest agents just abort when decryption fails. We in fact turn this into a pattern-matching problem, but it is part of the method (and its soundness proof) rather than being part of the model. Note that we assume that failure of a destructor is detectable; this is significant as an intruder may learn something from this failure. It seems reasonable to assume for the constructor-destructor theories supported here as most standard cryptographic implementations of primitives like AES and RSA indeed reveal if decryption failed. An interesting question is how to handle more algebraic properties like those of exponentiation with inverses that does not allow to detect failure to "decrypt" in general. However, such algebraic properties are not supported by any of the mentioned typing results.

A final generalization is that our approach supports protocol with long-term state (the memory cells). An interesting aspect of this is that there are several results concerning decidability based on typing and bounding the number of fresh nonces; one may wonder if this is also applicable in our case. However, there is an obstacle since our argument requires an infinite supply of constants of all types for the intruder to solve the disequalities that arise, among other things, from handling the long-term state. We thus leave this question to future work.

Another closely related problem is that of compositionality, which is also regarded as a relative soundness result: given that several protocols are secure in isolation, can we show that also their composition is secure? It works indeed also by similar methods, namely transforming an attack against the composed system to an attack against one component. Since here one of the key problems is when the attacker can use messages from one component in another component, and a solution can be similarly some form of tagging, one could call compositionality a form of "typing" for a family of protocols. In fact, typing can thus be a stepping stone for a compositionality result [7] and we plan to investigate if this is possible for (α, β)-privacy.

References

[1] M. Abadi and R. Needham, "Prudent engineering practice for cryptographic protocols," *IEEE Trans. Softw. Eng.*, vol. 22, no. 1, pp. 6–15, 1996.

[2] J. Heather, G. Lowe, and S. Schneider, "How to prevent type flaw attacks on security protocols," *J. Comput. Secur.*, vol. 11, no. 2, pp. 217–244, 2003.

[3] B. Blanchet and A. Podelski, "Verification of cryptographic protocols: tagging enforces termination," *Theor. Comput. Sci.*, vol. 333, no. 1, pp. 67–90, 2005.

[4] M. Arapinis and M. Duflot, "Bounding messages for free in security protocols – extension to various security properties," *Inf Comput*, vol. 239, pp. 182–215, 2014.

[5] O. Almousa, S. Mödersheim, P. Modesti, and L. Viganò, "Typing and compositionality for security protocols: A generalization to the geometric fragment," in *ESORICS 2015*, ser. LNCS, vol. 9327. Springer, 2015, pp. 209–229.

[6] A. Hess and S. Mödersheim, "Formalizing and proving a typing result for security protocols in Isabelle/HOL," in *CSF 2017*. IEEE, 2017, pp. 451–463.

[7] ——, "A typing result for stateful protocols," in *CSF 2018*. IEEE, 2018, pp. 374–388.

[8] R. Chrétien, V. Cortier, and S. Delaune, "Typing messages for free in security protocols: The case of equivalence properties," in *CONCUR 2014*, vol. 8704. Springer, 2014, pp. 372–386.

[9] 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.

[10] 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.

[11] V. Cheval and I. Rakotonirina, "Indistinguishability beyond diff-equivalence in ProVerif," in *CSF 2023*. IEEE, 2023, pp. 184–199.

[12] S. Gondron, S. Mödersheim, and L. Viganò, "Privacy as reachability," in *CSF 2022*. IEEE, 2022, pp. 130–146.

[13] L. Fernet, S. Mödersheim, and L. Viganò, "A decision procedure for alpha-beta privacy for a bounded number of transitions," in *CSF 2024 (to appear)*. IEEE, 2024, extended version at https://people.compute.dtu.dk/lpkf.

[14] 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

[15] L. Fernet and S. Mödersheim, "Private authentication with alpha-beta-privacy," in *OID 2023*, ser. LNI. GI, 2023.

[16] M. Abadi and C. Fournet, "Private authentication," *Theor. Comput. Sci.*, vol. 322, no. 3, pp. 427–476, 2004.

[17] 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.

[18] M. Ohkubo, K. Suzuki, and S. Kinoshita, "Cryptographic approach to "privacy-friendly" tags," in *RFID Privacy Workshop 2003*, 2003.

[19] ICAO, "Machine readable travel documents," Doc Series, Doc 9303, https://www.icao.int/publications/pages/publication.aspx?docnum=9303.

[20] A. Armando, R. Carbone, and L. Compagna, "SATMC: A sat-based model checker for security-critical systems," in *TACAS 2014*, ser. LNCS, vol. 8413. Springer, 2014, pp. 31–45.

[21] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. Hankes Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron, "The AVISPA tool for the automated validation of internet security protocols and applications," in *CAV 2005*, ser. LNCS, vol. 3576. Springer, 2005, pp. 281–285.

[22] A. Armando, W. Arsac, T. Avanesov, M. Barletta, A. Calvi, A. Cappai, R. Carbone, Y. Chevalier, L. Compagna, J. Cuéllar, G. Erzse, S. Frau, M. Minea, S. Mödersheim, D. von Oheimb, G. Pellegrino, S. E. Ponta, M. Rocchetto, M. Rusinowitch, M. Torabi Dashti, M. Turuani, and L. Viganò, "The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures," in *TACAS 2012*, ser. LNCS, vol. 7214. Springer, 2012, pp. 267–282.

[23] 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.
[24] S. Mödersheim and L. Viganò, "The open-source fixed-point model checker for symbolic analysis of security protocols," in *FOSAD 2007/2008/2009 Tutorial Lectures*, ser. LNCS, vol. 5705. Springer, 2009, pp. 166–194.
[25] M. Turuani, "The CL-Atse protocol analyser," in *RTA 2006*, ser. LNCS, vol. 4098. Springer, 2006, pp. 277–286.
[26] B. Blanchet, "An efficient cryptographic protocol verifier based on Prolog rules," in *CSFW 2001*. IEEE, 2001, pp. 82–96.
[27] A. Hess, S. Mödersheim, A. Brucker, and A. Schlichtkrull, "Performing security proofs of stateful protocols," in *CSF 2021*. IEEE, 2021, pp. 1–16.
[28] L. Paulson, "The inductive approach to verifying cryptographic protocols," *J. Comput. Secur.*, vol. 6, no. 1, pp. 85–128, 1998.
[29] A. Hess, S. Mödersheim, and A. Brucker, "Stateful protocol composition in isabelle/HOL," *ACM Trans. Priv. Secur.*, vol. 26, no. 3, pp. 1–36, 2023.
[30] R. Chrétien, V. Cortier, A. Dallon, and S. Delaune, "Typing messages for free in security protocols," *ACM Trans. Comput. Logic*, vol. 21, no. 1, pp. 1–52, 2020.

APPENDIX

A. Semantics

We give here the rules for executing a transaction and normalizing symbolic states (i.e., performing intruder experiments). The semantics below are taken directly from [13], except for the extension to receiving message patterns that we explicitly say is introduced in this paper. We add brief text explanations to the semantics, and we also give our formulation of the relations inducing the overall state transition system.

1) Execution of a Transaction: The following rules define the symbolic execution done by the intruder. For each rule, we give two versions: the rule for ground states, and the rule for symbolic states. A ground state contains a formula γ called the truth formula, which records the actual value of every privacy variable. This formula γ is not known by the intruder, but it uniquely determines in the ground state one possibility. This possibility, which is underlined in the rules, represents the possibility that really is the case, i.e., the concrete observations by the intruder are an instance of the structural frame in that case.

a) Non-Deterministic Choice: A privacy variable is chosen from a set of public constants. This step happens at the same time in every possibility, since the choices are in the left part of processes.

On the ground level, the rule is:

$$\begin{aligned}&\{(\mathsf{mode}\ x \in D.P_1, \phi_1, \mathit{struct}_1, \delta_1), \ldots,\\ &\ (\mathsf{mode}\ x \in D.P_n, \phi_n, \mathit{struct}_n, \delta_n)\}\\ &\to \{(P_1, \phi_1, \mathit{struct}_1, \delta_1), \ldots, (P_n, \phi_n, \mathit{struct}_n, \delta_n)\}\end{aligned}$$

where γ is augmented with $x \doteq c$, and if $\mathsf{mode} = \star$ (resp. $\mathsf{mode} = \diamond$) then α (resp. β_0) is augmented with $x \in D$.

On the symbolic level, the rule is:

$$\begin{aligned}&\{(\mathsf{mode}\ x \in D.P_1, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots,\\ &\ (\mathsf{mode}\ x \in D.P_n, \phi_n, \mathcal{A}_n, \mathcal{X}_n, \alpha_n, \delta_n)\}\\ &\Rightarrow \{(P_1, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots, (P_n, \phi_n, \mathcal{A}_n, \mathcal{X}_n, \alpha_n, \delta_n)\}\end{aligned}$$

and if $\mathsf{mode} = \star$ (resp. $\mathsf{mode} = \diamond$) then α_0 (resp. β_0) is augmented with $x \in D$.

b) Receive: A message is received. Again, this step happens at the same time in every possibility, since the receives are in the left part of processes.

On the ground level, there is a transition for every recipe r over the domain of the frames struct_i:

$$\begin{aligned}&\{\underline{(\mathsf{rcv}(X).P_1, \phi_1, \mathit{struct}_1, \delta_1)}, \ldots,\\ &\ (\mathsf{rcv}(X).P_n, \phi_n, \mathit{struct}_n, \delta_n)\}\\ &\to \{\underline{(P_1[X \mapsto \mathit{struct}_1(r)], \phi_1, \mathit{struct}_1, \delta_1)}, \ldots,\\ &\ (P_n[X \mapsto \mathit{struct}_n(r)], \phi_n, \mathit{struct}_n, \delta_n)\}\end{aligned}$$

On the symbolic level, a simple constraint is added to every FLIC because at this point any message from the intruder would do. It is only later in the process that there can be non-simple constraints on the message, which would then be solved with the lazy intruder.

$$\begin{aligned}&\{(\mathsf{rcv}(X).P_1, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots,\\ &\ (\mathsf{rcv}(X).P_n, \phi_n, \mathcal{A}_n, \mathcal{X}_n, \alpha_n, \delta_n)\}\\ &\Rightarrow \{(P_1, \phi_1, \mathcal{A}_1.{+}R \mapsto X, \mathcal{X}_1, \alpha_1, \delta_1), \ldots,\\ &\ (P_n, \phi_n, \mathcal{A}_n.{+}R \mapsto X, \mathcal{X}_n, \alpha_n, \delta_n)\}\end{aligned}$$

where R is a fresh recipe variable.

c) Extension to Pattern Matching: In this paper, we extend the semantics to allow receive steps where instead of a variable, the processes contain a linear pattern t that contains only fresh variables and no constants.

On the ground level, the intruder chooses some recipe and then the pattern is instantiated with the message produced in the respective frame, or the process behaves as 0 if the message does not match the pattern. The following transition can be taken for every recipe r over the domain of the struct_i:

$$\begin{aligned}&\{\underline{(\mathsf{rcv}(t).P_1, \phi_1, \mathit{struct}_1, \delta_1)}, \ldots,\\ &\ (\mathsf{rcv}(t).P_n, \phi_n, \mathit{struct}_n, \delta_n)\}\\ &\to \{\underline{(P'_1, \phi_1, \mathit{struct}_1, \delta_1)}, \ldots, (P'_n, \phi_n, \mathit{struct}_n, \delta_n)\}\end{aligned}$$

where for every $i \in \{1, \ldots, n\}$, $\sigma_i = \mathit{mgu}(t \doteq \mathit{struct}_i(r))$, and $P'_i = \sigma_i(P_i)$ if $\sigma_i \neq \bot$ or $P'_i = 0$ otherwise.

On the symbolic level, we have introduced the rule in Definition V.2 and give it here again for convenience:

$$\begin{aligned}&\{(\mathsf{rcv}(t).P_1, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots,\\ &\ (\mathsf{rcv}(t).P_n, \phi_n, \mathcal{A}_n, \mathcal{X}_n, \alpha_n, \delta_n)\}\\ &\Rightarrow \{(\sigma_1(P_1), \phi_1, \mathcal{A}'_1, \sigma_1(\mathcal{X}_1), \alpha_1, \sigma_1(\delta_1)), \ldots,\\ &\ (\sigma_n(P_n), \phi_n, \mathcal{A}'_n, \sigma_n(\mathcal{X}_n), \alpha_n, \sigma_n(\delta_n))\}\end{aligned}$$

where R is a fresh recipe variable and X and a fresh intruder variable, $\rho \in LI(\mathcal{A}_1.{+}R \mapsto X, [X \mapsto t])$, $\mathcal{A}'_i = \rho(\mathcal{A}_i.{+}R \mapsto X)$, and $\sigma_i = \sigma_\rho^{\mathcal{A}_i.+R \mapsto X}$ – and every $\sigma_i(\mathcal{X}_i)$ is satisfiable.

Moreover, there is also the following transition if $t \notin \mathcal{V}_{intruder}$:

$$\{(\mathsf{rcv}(t).P_1, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots, (\mathsf{rcv}(t).P_n, \phi_n, \mathcal{A}_n, \mathcal{X}_n, \alpha_n, \delta_n)\} \Rightarrow \{(0, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots, (0, \phi_n, \mathcal{A}_n, \mathcal{X}_n, \alpha_n, \delta_n)\}$$

d) Cell Read: A memory cell is read, and the process is substituted according to the value read. The evaluation is done in exactly one possibility at a time, because there is no guarantee that other possibilities have the same cell read at this point (the cell reads are in the center part of processes, so there could have been branching already). The memory δ contains the sequence $\mathsf{cell}(s_1) := t_1. \cdots .\mathsf{cell}(s_k) := t_k$ for the given cell, and the initial value is given with ground context $C[\cdot]$.

On the ground level, the rule is:

$$\{(X := \mathsf{cell}(s).P, \phi, \mathit{struct}, \delta)\} \uplus \mathcal{P} \rightarrow \{(\mathsf{if}\ s \doteq s_1\ \mathsf{then}\ P[X \mapsto t_1]\ \mathsf{else} \ldots \mathsf{if}\ s \doteq s_k\ \mathsf{then}\ P[X \mapsto t_k]\ \mathsf{else}\ P[X \mapsto C[s]], \phi, \mathit{struct}, \delta)\} \cup \mathcal{P}$$

On the symbolic level, the rule is the same:

$$\{(X := \mathsf{cell}(s).P, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \uplus \mathcal{P} \Rightarrow \{(\mathsf{if}\ s \doteq s_1\ \mathsf{then}\ P[X \mapsto t_1]\ \mathsf{else} \ldots \mathsf{if}\ s \doteq s_k\ \mathsf{then}\ P[X \mapsto t_k]\ \mathsf{else}\ P[X \mapsto C[s]], \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \cup \mathcal{P}$$

e) Cell Write: A memory cell is written, and the update is prepended to the sequence of memory updates in the possibilities. Again the evaluation is done in exactly one possibility at a time, since the cell writes are in the right part of processes.

On the ground level, the rule is:

$$\{(\mathsf{cell}(s) := t.P, \phi, \mathit{struct}, \delta)\} \uplus \mathcal{P} \rightarrow \{(P, \phi, \mathit{struct}, \mathsf{cell}(s) := t.\delta)\} \cup \mathcal{P}$$

On the symbolic level, the rule is the same:

$$\{(\mathsf{cell}(s) := t.P, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \uplus \mathcal{P} \Rightarrow \{(P, \phi, \mathcal{A}, \mathcal{X}, \alpha, \mathsf{cell}(s) := t.\delta)\} \cup \mathcal{P}$$

f) Destructor Application: A destructor is applied to a message. On the ground level, try-catch is syntactic sugar around if-then-else, so there is only the rule for conditional statements.

On the symbolic level, try-catch is handled in a particular way. Note that in the protocol specification, only variables are decomposed, but since other evaluations rules perform substitutions, the rule must account for messages that are not necessarily variables. The evaluation is done in exactly one possibility at a time, since the destructor applications are in the center part of processes in the grammar from [13] (it is introduced in this paper that we consider them in the left part and with always empty catch branches). The rule is where the lazy intruder may be used to solve constraints: the rewrite rule corresponding to the destructor is looked up, and a unifier representing the constraints is computed. The symbolic execution may then lead to multiple successor symbolic states, representing different choices of recipes by the intruder. In each, the possibility is split between the two branches, in one case decryption succeeded and in the other it failed.

Let $d(k, c(k', X_1, \ldots, X_n)) \rightarrow X_i$ (for some $i \in \{1, \ldots, n\}$) be the rewrite rule for d, assuming the variables in k, k' and the X_j have been renamed with fresh intruder variables. Let $\sigma = \mathit{mgu}(t_1 \doteq k \wedge t_2 \doteq c(k', X_1, \ldots, X_n) \wedge X \doteq X_i)$.

- Case $\sigma = \bot$:

  $$\{(\mathsf{try}\ X := d(t_1, t_2)\ \mathsf{in}\ P_1\ \mathsf{catch}\ P_2, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \uplus \mathcal{P} \Rightarrow \{(P_2, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \cup \mathcal{P}$$

- Case $\sigma \neq \bot$: Let $\{\rho_1, \ldots, \rho_m\} = LI(\mathcal{A}, \sigma)$ and σ_0 be the substitution of privacy variables for which the decryption succeeds.

  $$\{(\mathsf{try}\ X := d(t_1, t_2)\ \mathsf{in}\ P_1\ \mathsf{catch}\ P_2, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \uplus \mathcal{P} \Rightarrow \rho_i(\{(P_1, \phi \wedge \sigma_0, \mathcal{A}, \mathcal{X}, \alpha, \delta), (P_2, \phi \wedge \neg\sigma_0, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \cup \mathcal{P})$$

  Moreover:

  $$\{(\mathsf{try}\ X := d(t_1, t_2)\ \mathsf{in}\ P_1\ \mathsf{catch}\ P_2, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \uplus \mathcal{P} \Rightarrow \{(P_2, \phi, \mathcal{A}, \mathcal{X} \wedge \forall \bar{Y}.\ \neg\sigma, \alpha, \delta)\} \cup \mathcal{P}$$

  where $\bar{Y} = \mathit{ivars}(\sigma) \setminus \mathit{ivars}(\mathcal{A})$. The function ivars gives the intruder variables of a FLIC, i.e., $\mathit{ivars}(\mathcal{A}) = \mathit{vars}(\mathcal{A}) \cap \mathcal{V}_{intruder}$; we extend this function to substitutions.

g) Conditional Statement: A process has a conditional statement, which leads to branching. Again the evaluation is done in exactly one possibility at a time, since the conditional statements are in the center part of processes.

On the ground level, a possibility is split into two, one for the case that the condition is true and the process goes into the then branch, and one for the else branch. By construction, if the marked possibility is split then there is only one branch that is consistent with the current truth γ and it is marked accordingly.

$$\{(\mathsf{if}\ \psi\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2), \phi, \mathit{struct}\} \uplus \mathcal{P} \rightarrow \{(P_1, \phi \wedge \psi, \mathit{struct}), (P_2, \phi \wedge \neg\psi, \mathit{struct})\} \cup \mathcal{P}$$

On the symbolic level, this is handled similarly to destructor applications, where the unifier is computed from the condition.

- If the condition is a relation:

  $$\{(\mathsf{if}\ R(t_1, \ldots, t_n)\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \uplus \mathcal{P} \Rightarrow \{(P_1, \phi \wedge R(t_1, \ldots, t_n), \mathcal{A}, \mathcal{X}, \alpha, \delta), (P_2, \phi \wedge \neg R(t_1, \ldots, t_n), \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \cup \mathcal{P}$$

- If the condition is an equality: We first compute the unifier $\sigma = mgu(s \doteq t)$ and then the transitions are just like for destructor application.
- If the condition uses conjunction or negation: the branches are nested or swapped until one of the previous cases is reached.

h) Send: A message is sent, which adds a new label to record that message in the intruder knowledge. This rule can only be applied if all possibilities start either with $\mathsf{snd}(\cdot)$ or 0; otherwise another rule must be applied. Note that this rule eliminates all possibilities which are terminating and thus not sending a message. This corresponds to the static distinguishability of paths: the intruder can tell whether they observe a message or the transaction is finished.

On the ground level, every $struct_i$ is augmented by the message sent in the respective possibility:

$$\begin{aligned}&\{\underline{(\mathsf{snd}(t_1).P_1, \phi_1, struct_1, \delta_1)}, \ldots,\\ &\ (\mathsf{snd}(t_k).P_k, \phi_k, struct_k, \delta_k)\} \uplus \mathcal{P}\\ &\to \{\underline{(P_1, \phi_1, struct_1.l \mapsto t_1, \delta_1)}, \ldots,\\ &\qquad (P_k, \phi_k, struct_k.l \mapsto t_k, \delta_k)\}\end{aligned}$$

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.

On the symbolic level, a new mapping is added to the FLICs.

$$\begin{aligned}&\{(\mathsf{snd}(t_1).P_1, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots,\\ &\ (\mathsf{snd}(t_k).P_k, \phi_k, \mathcal{A}_k, \mathcal{X}_k, \alpha_k, \delta_k)\} \uplus \mathcal{P}\\ &\Rightarrow \{(P_1, \phi_1, \mathcal{A}_1.{-}l \mapsto t_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots,\\ &\qquad (P_k, \phi_k, \mathcal{A}_k.{-}l \mapsto t_k, \mathcal{X}_k, \alpha_k, \delta_k)\}\end{aligned}$$

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.

i) Release: A formula is released, which happens done in exactly one possibility at a time, since the releases are in the right part of processes.

On the ground level, the formula released by the marked possibility is added to the payload or the truth (depending on the mode), and formulas released by other possibilities are ignored.

$$\{\underline{(\mathsf{mode}\ \psi.P, \phi, struct)}\} \uplus \mathcal{P} \to \{\underline{(P, \phi, struct)}\} \cup \mathcal{P}$$

and $\alpha \leftarrow \alpha \wedge \psi$ if $\mathsf{mode} = \star$ or $\gamma \leftarrow \gamma \wedge \psi$ if $\mathsf{mode} = \diamond$.

On the symbolic level, the formula released is added to the partial payload of the possibility. [13] say that their decision procedure does not support releases with $\mathsf{mode} = \diamond$, and thus the rule is only defined for releases with $\mathsf{mode} = \star$.

$$\{(\star\ \psi.P, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta) \uplus \mathcal{P} \Rightarrow \{(P, \phi, \mathcal{A}, \mathcal{X}, \alpha \wedge \psi, \delta)\} \cup \mathcal{P}$$

j) Terminate: A process is terminating, which leads to a finished state. Note that this rule eliminates all possibilities which are sending a message and thus not terminating. It is the counterpart of the Send rule.

On the ground level, the rule is:

$$\begin{aligned}&\{\underline{(0, \phi_1, struct_1, \delta_1)}, \ldots, (0, \phi_k, struct_k, \delta_k)\} \uplus \mathcal{P}\\ &\to \{\underline{(0, \phi_1, struct_1, \delta_1)}, \ldots, (0, \phi_k, struct_k, \delta_k)\}\end{aligned}$$

where every process in $\mathcal{P}$ starts with a send step and $\beta_0 \leftarrow \beta_0 \wedge \bigvee_{i=1}^{k} \phi_i$.

On the symbolic level, the rule is the same:

$$\begin{aligned}&\{(0, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots, (0, \phi_k, \mathcal{A}_k, \mathcal{X}_k, \alpha_k, \delta_k)\} \uplus \mathcal{P}\\ &\Rightarrow \{(0, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots, (0, \phi_k, \mathcal{A}_k, \mathcal{X}_k, \alpha_k, \delta_k)\}\end{aligned}$$

where every process in $\mathcal{P}$ starts with a send step and $\beta_0 \leftarrow \beta_0 \wedge \bigvee_{i=1}^{k} \phi_i$.

2) Normalization of a Symbolic State: After the symbolic execution of a transaction, the decision procedure normalizes symbolic states by performing all relevant intruder experiments, i.e., comparing recipes to check whether they produce the same message. Given two recipes, in each FLIC the unifier of the messages produced is computed. If the outcome of the experiment depends only on privacy variables, then the symbolic state is split to consider in one case that the recipes produced the same message and in the other case that they produce different. Otherwise there is some unifier that requires solving constraints, and this is done with the lazy intruder.

We define $isPriv(\sigma)$ iff $dom(\sigma) \subseteq \mathcal{V}_{privacy}$ and $isPriv(\bot) = \mathsf{false}$.

a) Privacy Split:

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

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

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

b) Recipe Split:

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

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

3) Transition Relations: The relations $\Rightarrow$ and $\rightarrowtail$ defined above correspond to the evaluation of processes and normalization of symbolic state, respectively. In the transition system for the protocol execution, symbolic states are reached by executing one transaction and then normalizing. Given a transaction P and a finished symbolic state $\mathcal{S}$, let $init(P, \mathcal{S})$ denote the symbolic state where all the 0 processes in $\mathcal{S}$ are replaced with P.

The transition system using destructor oracles from [13] is defined with the relation $\Longrightarrow$, where given two normal symbolic states $\mathcal{S}$ and $\mathcal{S}'$, we have $\mathcal{S} \Longrightarrow \mathcal{S}'$ iff there exists a transaction P (including the destructor oracles) such that $init(P, \mathcal{S}) \Rightarrow^* \rightarrowtail^* \mathcal{S}'$.

In contrast, the transition system using built-in analysis transitions from this paper is defined with the relation $\Longrightarrow^\bullet$, where given two normal symbolic states $\mathcal{S}$ and $\mathcal{S}'$, we have $\mathcal{S} \Longrightarrow^\bullet \mathcal{S}'$ iff either there exists a transaction P such that $init(P, \mathcal{S}) \Rightarrow^* \rightarrowtail^* \mathcal{S}'$ or $\mathcal{S} \Rightarrow^\bullet \Rightarrow^* \rightarrowtail^* \mathcal{S}'$. This corresponds to either executing exactly one transaction (where there are no destructor oracles) or performing exactly one analysis transition (followed by execution of the processes and normalization, as is done for transactions).

B. Proofs

We start with the proof that using pattern matching is correct w.r.t. destructor applications. This proof refers to the semantics for ground states, while our later proofs will work directly on symbolic states.

Lemma III.1. *A protocol satisfying Definition III.5 and its transformation to use pattern matching according to Definition III.6 yield the same set of reachable ground states (up to logical equivalence of the contained formulas α and β).*

Proof. The ground semantics defined in [13] (which we give in Appendix A) of $\mathsf{try}\ X := d(k,t)\ \mathsf{in}\ P\ \mathsf{catch}\ Q$ is syntactic sugar for a conditional $\mathsf{if}\ \phi\ \mathsf{then}\ P[X \mapsto d(k,t)]\ \mathsf{else}\ \mathsf{catch}\ Q$ where $Q = 0$ in our paper and ϕ is a formula expressing that the destructor application is successful. By construction, if ϕ is true, then $d(k,t)$ yields the respective subterm of t.

Since in this work, $Q = 0$ and try is only allowed in the left part of processes (i.e., before branching), then a sequence of try can be written as a single if condition ϕ (the conjunction of the conditions of the individual try) and that can again be split $\phi = \phi_1 \wedge \phi_2$ into conditions ϕ_1 on the structure (that the transformed process handles as a pattern) and a condition on the values ϕ_2.

In the transformed specification, if the pattern is satisfied, then the pattern variables are bound to the corresponding subterms of t as the destructor terms $d(k,t)$ mentioned above. This also leads to the same possibilities in both models: in the original process, each possibility splits into two, namely whether ϕ is satisfied or not. In the transformed specification, if ϕ_1 holds there is also a split into two by whether ϕ_2 holds or not. Otherwise, if $\neg\phi_1$ holds, there is no split (we arrive at 0 for sure). Now in each model, the intruder knows the typing of the messages and thus whether ϕ_1 holds. Thus, if ϕ_1 holds, the intruder in the original model can simplify the conditions ϕ and $\neg\phi$ to ϕ_2 and $\neg\phi_2$, respectively, yielding exactly the same conditions as in the new model. Conversely, if $\neg\phi_1$ holds, then the intruder in the original model can rule out the $\neg\phi$ case and knows that ϕ_2 holds, exactly as in the new model. □

We now prove that the lazy intruder rules only return well-typed solutions when used on a type-flaw resistant protocol.

Lemma IV.1. *Let Spec be a type-flaw resistant protocol, $\mathcal{A}$ be a FLIC such that $terms(\mathcal{A}) \subseteq SMP(patterns(Spec))$, ρ be a choice of recipes such that $dom(\rho) \cap rvars(\mathcal{A}) = \emptyset$, σ be a well-typed substitution such that $dom(\sigma) \cap vars(\mathcal{A}) = \emptyset$, and $(\rho', \mathcal{A}', \sigma')$ be such that $(\rho, \mathcal{A}, \sigma) \rightsquigarrow (\rho', \mathcal{A}', \sigma')$. Then σ' is well-typed.*

Proof. To show that the constraint solving always makes well-typed instantiations of intruder and privacy variables, we proceed by distinguishing which lazy intruder rule has been applied.

Unification: Then $\mathcal{A} = \mathcal{A}_1.-l \mapsto s.\mathcal{A}_2.+R \mapsto t.\mathcal{A}_3$, $\rho' = [R \mapsto l]\rho$ and $\sigma' = mgu(\sigma \wedge s \doteq t)$. We have that $s, t \in SMP(patterns(Spec)) \setminus \mathcal{V}$. Since *Spec* is type-flaw resistant and s and t are unifiable, $\Gamma(s) = \Gamma(t)$. Thus, σ' is well-typed.

Guessing: Then $\mathcal{A} = \mathcal{A}_1.+R \mapsto x.\mathcal{A}_2$, $\rho' = [R \mapsto c]\rho$ and $\sigma' = mgu(\sigma \wedge x \doteq c)$, for some $c \in dom(x)$. The guess c is a constant in the domain of the privacy variable x so $\Gamma(x) = \Gamma(c)$. Thus, σ' is well-typed.

Composition or **Repetition**: Then $\sigma' = \sigma$, i.e., no intruder or privacy variables are instantiated. Thus, σ' is well-typed.

Note that since σ' is well-typed and $SMP(patterns(Spec))$ is closed under well-typed instantiations, then $terms(\mathcal{A}') \subseteq SMP(patterns(Spec))$. □

Theorem IV.1 (Lazy intruder well-typedness)**.** *Let Spec be a type-flaw resistant protocol, $\mathcal{A}$ be a simple FLIC such that $terms(\mathcal{A}) \subseteq SMP(patterns(Spec))$ and let σ be a well-typed substitution. Then every $\rho \in LI(\mathcal{A}, \sigma)$ is well-typed w.r.t. $\mathcal{A}$.*

Proof. By induction and using Lemma IV.1, for every σ' such that $(\varepsilon, \sigma(\mathcal{A}), \sigma) \rightsquigarrow^* (\rho, \mathcal{A}', \sigma')$ and $\mathcal{A}'$ is simple, we have that σ' is well-typed. For every $+R \mapsto X \in \mathcal{A}$, we have $\sigma'(\mathcal{A}'(\rho(R))) = \sigma'(X)$, so $\Gamma(\mathcal{A}'(\rho(R))) = \Gamma(X)$. Moreover, $\rho(\mathcal{A})(\rho(R))$ and $\mathcal{A}'(\rho(R))$ are unifiable, because they only differ in the variables introduced by applying ρ to $\mathcal{A}$. Since there are infinitely many variables of each type, then without loss of generality the fresh intruder variables introduced by ρ can be taken of the appropriate types such that $\Gamma(X) = \Gamma(\rho(\mathcal{A})(\rho(R)))$. Thus, ρ is well-typed w.r.t. $\mathcal{A}$. □

As an intermediate result, we show that, given a set of FLICs with the same domain and constraints, solving the constraints to send a message pattern in one FLIC is equivalent to solving the constraints in any other FLIC.

Lemma A.1. *Let Spec be a type-flaw resistant protocol and $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be FLICs such that:*

- $dom(\mathcal{A}_1) = \cdots = dom(\mathcal{A}_n)$ *and for every label, the messages in the different FLICs have the same type.*
- *The messages sent in each FLIC are equal.*
- *For every* $i \in \{1, \ldots, n\}$, $terms(\mathcal{A}_i) \subseteq SMP(patterns(Spec))$ *and for every* $+R \mapsto t \in \mathcal{A}_i$, t *is linear, does not contain constants and the variables in* $fv(t)$ *do not occur in any other message sent.*

Then a lazy intruder rule is applicable in $\mathcal{A}_1$ *iff that rule is applicable in every* $\mathcal{A}_i$.

Proof. Let us consider the first non-simple constraints, say it is to send a message t. First, we assume that Unification is applicable in $\mathcal{A}_1$. Then it means that t can be unified with another message observed earlier, i.e., there is a label l that maps to a message unifying with t. Since t contains only fresh variables and no constants, then t can be unified with any message of the same type. Since the label l maps to messages of the same type in every FLIC, then l is a solution in every FLIC so Unification is applicable in the same way in every FLIC. Note that the variables in t that are substituted do not make any other constraints non-simple, since these variables do not occur in any other message sent.

Second, we assume that Compose is applicable. Then it means that t is composed with a public function at the top-level. The intruder can produce t with a composed recipe, using the same function at the top-level and subrecipes for the arguments, and this is applicable in every FLIC.

Since every message to send is linear and contains only fresh variables, the rules of Guessing and Repetition are not applicable. Moreover, after one rule application, the updated FLICs still form a set of FLICs with identical messages to send. This means that Guessing and Repetition will never be applicable when solving the constraints. □

Lemma A.2. *Given a type-flaw resistant protocol, the transitions for receiving message patterns always perform well-typed instantiations.*

Proof. First, we consider the case that the intruder makes some choice of recipes computed with the lazy intruder. The transition is:

$$\begin{aligned}&\{(\mathsf{rcv}(t).P_1, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots,\\ &\ (\mathsf{rcv}(t).P_n, \phi_n, \mathcal{A}_n, \mathcal{X}_n, \alpha_n, \delta_n)\}\\ &\Rightarrow \{(\sigma_1(P_1), \phi_1, \mathcal{A}'_1, \sigma_1(\mathcal{X}_1), \alpha_1, \sigma_1(\delta_1)), \ldots,\\ &\qquad (\sigma_n(P_n), \phi_n, \mathcal{A}'_n, \sigma_n(\mathcal{X}_n), \alpha_n, \sigma_n(\delta_n))\}\end{aligned}$$

where R is a fresh recipe variable and X is a fresh intruder variable, $\rho \in LI(\mathcal{A}_1.{+}R \mapsto X, [X \mapsto t])$, $\mathcal{A}'_i = \rho(\mathcal{A}_i.{+}R \mapsto X)$, and $\sigma_i = \sigma_\rho^{\mathcal{A}_i.+R \mapsto X}$ – and every $\sigma_i(\mathcal{X}_i)$ is satisfiable. By Theorem IV.1, ρ is well-typed w.r.t. $\mathcal{A}_1.{+}R \mapsto X$, and thus also w.r.t. $\mathcal{A}_1$. By induction using Lemma A.1, the lazy intruder gives the same results in every FLIC. Therefore ρ is actually well-typed w.r.t. $\mathcal{A}_i.{+}R \mapsto X$, and thus $\mathcal{A}_i$, for every $i \in \{1, \ldots, n\}$.

Next, we consider the case that the intruder sends a message that does not match a pattern $t \notin \mathcal{V}_{intruder}$. Then there is no instantiation of variables. □

Next we show that our rules for pattern matching for symbolic states are a correct representation on pattern matching for ground states. This is an update of the correctness result in [13], Proposition 1, when adding the new pattern matching construct.

Lemma V.1. *Given a type-flaw resistant specification, then the set of reachable states in the symbolic semantics represents exactly the reachable states of the ground semantics.*

Proof. We use the fact that this is already proved for all previous constructs in [13] and just show it for the newly added rules for receiving with pattern matching (found in Definition V.2 and Appendix A).

Given a symbolic state $\mathcal{S}$ where the possibilities have the form:

$$\begin{aligned}&\{(\mathsf{rcv}(t).P_1, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \delta_1), \ldots,\\ &\ (\mathsf{rcv}(t).P_n, \phi_n, \mathcal{A}_n, \mathcal{X}_n, \alpha_n, \delta_n)\}\end{aligned}$$

i.e., so that new the pattern-matching receive rules are applicable (the second rule only if $t \notin \mathcal{V}_{intruder}$).

For the positive case (i.e., satisfying the pattern) for a type-flaw resistant protocol, it follows from Lemma A.2 that all $\mathcal{A}_i$ have the same set ρ of solutions for producing t, i.e., for $\mathcal{A}_i.{+}R \mapsto t$.

Completeness (all reachable ground states are represented on the symbolic level): The new ground rule for pattern matching receive (Appendix A) is applicable for an arbitrary recipe r in every state ground S represented by $\mathcal{S}$. If r produces an instance of t, then by the correctness of the lazy intruder, r is an instance of $\rho(R)$ for one of the ρ that solve $\mathcal{A}_i.{+}R \mapsto t$. The resulting ground state S' is thus covered by the symbolic state that uses the positive rule with ρ.

If r does not produce an instance of t, then we go directly to 0 process in all possibilities, and this is covered by the second rule of the symbolic level, since in this case the pattern cannot be a variable.

Soundness (only reachable ground states are represented on the symbolic level): If ρ is a solution in $\mathcal{A}_i.{+}R \mapsto t$, then all ground states represented by $\rho(\mathcal{S})$ allow for applying the pattern rule with $r = \rho(R)$. Thus every successor state $\mathcal{S}'$ resulting from the first symbolic applying ρ is represented by a ground state. Moreover, if the second rule is applicable (when $t \notin \mathcal{V}_{intruder}$) then there is a ground recipe r that is not an instance of t, thus the transition that makes all processes 0 is also possible on the ground level. □

Before proving, for type-flaw resistant protocols, the equivalence between analysis transitions and destructor oracles, we show an intermediate result. Whenever a label maps to a composed term, then in every FLIC the label maps to a composed term with the same top-level function. This will be useful to make sure that if a destructor oracle can be applied, then also the transition for analysis can be applied.

Lemma A.3. *Let* $\mathcal{S}$ *be a normal symbolic state with FLICs* $\mathcal{A}_1, \ldots, \mathcal{A}_n$ *and* $l \in dom(\mathcal{S})$ *such that* $-l \mapsto c(t_1^1, \ldots, t_1^m) \in$

$\mathcal{A}_1$, *where* $m > 0$. *Then for every* $i \in \{1, \ldots, n\}$, *we have* $-l \mapsto c(t_i^1, \ldots, t_i^m) \in \mathcal{A}_i$ *for some terms* t_i^j.

Proof. Assume that in some FLIC $\mathcal{A}_j$ ($j \neq 1$) we have $-l \mapsto X$, where $X \in \mathcal{V}_{intruder}$. Since $\mathcal{S}$ is normal, the experiment (l, R) must have been done already, i.e., $(l, R) \in \mathit{Checked}$, where $+R \mapsto X \in \mathcal{A}_j$. We will show that this contradicts the assumption that $\mathcal{S}$ is well-formed. All states in the symbolic execution are well-formed by construction. The complete definition of well-formedness is found in [13] but for our purpose the following suffices: let $\sigma_i = mgu(\mathcal{A}_i(l) \doteq \mathcal{A}_i(R))$ for i in $\{1, \ldots, n\}$, since $(l, R) \in \mathit{Checked}$, then either for every $i \in \{1, \ldots, n\}$, $\mathit{isPriv}(\sigma_i)$ and $\alpha_0 \wedge \beta_0 \wedge \phi_i \models \sigma_i$, or for every $i \in \{1, \ldots, n\}$, $LI(\mathcal{A}_i, \sigma_i) = \emptyset$ or ($\mathit{isPriv}(\sigma_i)$ and $\alpha_0 \wedge \beta_0 \wedge \phi_i \models \neg\sigma_i$).

However, we have not $\mathit{isPriv}(\sigma_1)$. We also have that $\sigma_j = \varepsilon$, so $LI(\mathcal{A}_j, \sigma_j) \neq \emptyset$ and $\not\models \neg\sigma_j$. This contradicts well-formedness, so we conclude that in every FLIC, the label does not map to an intruder variable. Recall that, since the messages sent in different branches have the same types, every label maps to messages of the same type. Moreover, since $m > 0$, it cannot be the type of a privacy variable. Therefore the message mapped by l has the same constructor in every FLIC. □

Lemma V.2. *Given a type-flaw resistant protocol,* $\Longrightarrow = \Longrightarrow^\bullet$.

Proof. Let $\mathcal{S}, \mathcal{S}'$ be reachable symbolic states in a type-flaw resistant protocol. For executing normal transactions, the same transitions are possible in both relations. The only thing we have to show is that destructor oracles and analysis transitions are equivalent. Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be the FLICs in the state $\mathcal{S}$.

First, we assume that $\mathcal{S} \Longrightarrow \mathcal{S}'$, where some destructor oracle is executed. Then it means that $\mathcal{S}$ is normal and there exist a label $l \in dom(\mathcal{S})$ and a public destructor $d \in \Sigma_{pub}$ such that l can be analyzed with d, i.e., $-l \mapsto c(k', t_1, \ldots, t_m)$ in some FLIC, where c occurs in the rewrite rule for d. By Lemma A.3, in all the other FLICs also l maps to composed messages with the same constructor c, i.e., for every $i \in \{1, \ldots, n\}$, we have $-l \mapsto c(k_i', t_i^1, \ldots, t_i^m) \in \mathcal{A}_i$. By Definition III.5, for every $i \in \{1, \ldots, n\}$, we have $d(k_i, c(k_i', t_i^1, \ldots, t_i^m)) \to t_i^j$ as an instance of the rewrite rule for d, for some term k_i and some $j \in \{1, \ldots, m\}$. Executing the destructor oracle specialized with label l means that in FLIC $\mathcal{A}_i$, the process is now $\mathsf{rcv}(X).\mathsf{if}\ X \doteq k_i\ \mathsf{then}\ \mathsf{snd}(t_i^j).\mathsf{snd}(k_i)$, which is exactly what we get from the corresponding analysis transition. Thus $\mathcal{S} \Longrightarrow^\bullet \mathcal{S}'$.

Second, we assume that $\mathcal{S} \Longrightarrow^\bullet \mathcal{S}'$, where some analysis transition is taken. Then it means that $\mathcal{S}$ is normal and there exist a label $l \in dom(\mathcal{S})$ and a public destructor $d \in \Sigma_{pub}$ such that l can be analyzed with d, i.e., $-l \mapsto c(k', t_1, \ldots, t_n)$ in some FLIC, where c occurs in the rewrite rule for d. The corresponding destructor oracle transaction can be executed, leading to the same state. Thus $\mathcal{S} \Longrightarrow \mathcal{S}'$. □

Theorem V.1 (Typing result)**.** *Given a type-flaw resistant protocol, it is correct to restrict the intruder model to well-typed recipes/messages for verifying privacy.*

Proof. The only way that variables are instantiated during the transitions is by applying some lazy intruder result. For every transition, we ensure that all messages in the FLICs are in the set of sub-message patterns of the protocol. By Definition III.4, all the constraints occurring during the symbolic execution are well-typed, and thus by Theorem IV.1, the lazy intruder only performs well-typed instantiations. In a reachable state, all constraints are simple, i.e., all the messages sent are pairwise distinct intruder variables. Since the intruder can produce an unbounded number of messages of each type, then they can instantiate the intruder variables in a well-typed way. □

C. Models and Details for Case Studies

Note that in some of the models below, we write steps of right processes before steps of center processes. This is syntactic sugar to avoid repetition, for instance if we write $\mathsf{snd}(t).\mathsf{if}\ \phi\ \mathsf{then}\ P\ \mathsf{else}\ Q$ it means that $\mathsf{snd}(t)$ happens in both branches. We also use a wildcard "_" instead of a binding name when the value is not used in the transaction.

1) Basic Hash: We take the model found in [13] and add type annotations.

```
Sigma0: public t1/0 t2/0 t3/0
Sigma: public h/2 ok/0
       private sk/1 extract/1
Types: t1:tag t2:tag t3:tag ok:reply
Algebra: extract(h(sk(X),Y))->X

Transaction Tag:
* x in {t1,t2,t3}.
new N:nonce.
send pair(N,h(sk(x),N))

Transaction Reader:
receive M:pair(nonce,h(sk(tag),nonce)).
try N:=proj1(M) in
try H:=proj2(M) in
try X:=extract(H) in
if H=h(sk(X),N) then
  send ok
```

Then we have that this protocol satisfies Definition III.5. There is no destructor application to remove in the tag transaction. However, for the reader, we apply Definition III.6 to get the following transaction with pattern matching:

```
Transaction ReaderPat:
receive pair(N:nonce,
             h(sk(_:tag),N':nonce)).
if N=N' then
  send ok
```

Thus, we have the following message patterns:

$$M = \{x, \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3, N, \mathsf{pair}(N, \mathsf{h}(\mathsf{sk}(x), N)), \mathsf{pair}(N', \mathsf{h}(\mathsf{sk}(X), N'')), N', N'', \mathsf{ok}\}$$

with the following types for variables and constants:

$$\Gamma(x) = \Gamma(\mathsf{t}_1) = \Gamma(\mathsf{t}_2) = \Gamma(\mathsf{t}_3) = \Gamma(X) = \mathsf{tag}$$
$$\Gamma(N) = \Gamma(N') = \Gamma(N'') = \mathsf{nonce}$$
$$\Gamma(\mathsf{ok}) = \mathsf{reply}$$

The set M is type-flaw resistant, and thus Basic Hash is type-flaw resistant.

2) BAC: The model found in [15] contains, in the response transaction, a non-empty catch branch, which is not supported by the typing result. Therefore we change the model by replacing the symmetric decryption with private extractors. Note that the original model is slightly different from the model below: in case the message M received by the tag is not of the form $\mathsf{scrypt}(\mathsf{sk}(\cdot),\cdot,\cdot)$, the original model returns a format error, while the model here does not send anything in case M does not have the right form. However, in the original model, even if the intruder sends a message that does not have the right form message, the tag will respond with a message of type reply. Thus, the intruder knows the types of the messages in their knowledge. Therefore, the intruder also knows, before sending, whether a message matches the pattern, so they would not learn anything by sending a message that is not an encryption of the right form. This is why we consider our changes reasonable.

```
Sigma0: public t1/0 t2/0
Sigma: public ok/0 formatErr/0 fixedR/0
       private sk/1 fresh/0 spent/0
               session/2 sfst/1 ssnd/1
               recipient/1 content/1
Types: t1:tag t2:tag ok:reply
       nonceErr:reply formatErr:reply
       fixedR:nonce fresh:state
       spent:state
Algebra: sfst(session(X,Y))->X
         ssnd(session(X,Y))->Y
         recipient(scrypt(sk(X),M,R))->X
         content(scrypt(sk(X),M,R))->M
Cells: noncestate[N:nonce]:=fresh

Transaction Challenge:
* x in {t1,t2}.
new N:nonce.
send session(x,N).
send N.
send scrypt(sk(x),N,fixedR)

Transaction Response:
receive Session:session(tag,nonce).
receive M:scrypt(sk(tag),nonce,nonce).
try X:=sfst(Session) in
try N:=ssnd(Session) in
try Y:=recipient(M) in
try NN:=content(M) in
if Y=X then
  State:=noncestate[N].
  if N=NN and State=fresh then
    noncestate[N]:=spent.
    send ok
  else send formatErr
else send formatErr
```

Then we have that this protocol satisfies Definition III.5. There is no destructor application to remove in the challenge transaction. However, for the response transaction, we apply Definition III.6 to get the following transaction with pattern matching:

```
Transaction ResponsePat:
receive session(X:tag,N:nonce).
receive scrypt(sk(Y:tag),
               NN:nonce,
               _:nonce).
if Y=X then
  State:=noncestate[N].
  if N=NN and State=fresh then
    noncestate[N]:=spent.
    send ok
  else send formatErr
else send formatErr
```

Thus we have the following message patterns:

$$M = \{x, \mathsf{t}_1, \mathsf{t}_2, N, \mathsf{session}(x, N), \mathsf{scrypt}(\mathsf{sk}(x), N, \mathsf{fixedR}),$$
$$\mathsf{session}(X, N'), \mathsf{scrypt}(\mathsf{sk}(Y), NN, R), Y, X, \mathit{State},$$
$$N', NN, \mathsf{fresh}, \mathsf{spent}, \mathsf{ok}, \mathsf{formatErr}\}$$

with the following types for variables and constants:

$$\Gamma(x) = \Gamma(\mathsf{t}_1) = \Gamma(\mathsf{t}_2) = \Gamma(X) = \Gamma(Y) = \mathsf{tag}$$
$$\Gamma(N) = \Gamma(N') = \Gamma(\mathsf{fixedR}) = \Gamma(NN) = \Gamma(R) = \mathsf{nonce}$$
$$\Gamma(\mathit{State}) = \Gamma(\mathsf{fresh}) = \Gamma(\mathsf{spent}) = \mathsf{state}$$
$$\Gamma(\mathsf{ok}) = \Gamma(\mathsf{formatErr}) = \mathsf{reply}$$

The set M is type-flaw resistant, and thus BAC is type-flaw resistant.

3) Private Authentication: The model found in [15] contains, in the responder transaction, a non-empty catch branch, which is not supported by the typing result. Moreover, the reply sent by the responder is either an encryption or a fresh nonce as decoy. In general, the intruder does not know which is the case so when they observe that such a reply is sent, they do not know a priori its type. Therefore we change the model: first by replacing the asymmetric decryption with private destructors, and second by replacing the fresh nonce as decoy with a fresh encryption. In this formulation, the protocol is still not type-flaw resistant because a reply from the responder can be confused with the message sent by initiator, even though these have different types. Thus, our final change is replacing the pairing function by distinct formats.

```
Sigma0: public a/0 b/0 i/0
Sigma: public f1/2 f2/1 df11/1 df12/1
            df2/1
```

```
       private recipient/1 sender/1
Types: a:agent b:agent i:agent
Algebra:
recipient(crypt(pk(B),f1(A,NA),R))->B
sender(crypt(pk(B),f1(A,NA),R))->A
df11(f1(X,Y))->X
df12(f1(X,Y))->Y
df2(f2(X))->X

Transaction ReceivePrivateKey:
send inv(pk(i))

Transaction Initiator:
* xA in {a,b}.
* xB in {a,b,i}.
new NA:nonce,R:nonce.
send crypt(pk(xB),f1(xA,NA),R).
if xB=i then
  * xA=gamma(xA) and xB=gamma(xB)
else
  * xB in {a,b}

Transaction Responder:
* xB in {a,b}.
receive M:crypt(pk(agent),
                f1(agent,nonce),
                nonce).
try C:=recipient(M) in
try A:=sender(M) in
new NB:nonce,R:nonce.
if C=xB and A in {a,b,i} then
  send crypt(pk(A),f2(NB),R).
  if A=i then
    * xB=gamma(xB)
else
  new AA:agent.
  send crypt(pk(AA),f2(NB),R).
  if A in {a,b,i} and C in {a,b} then
    * not (C=xB and A=i)
```

In this model, it is convenient to make use of a meta-notation. The details of the meta-notation are defined in [13] and we only explain the idea behind it here: a formula released like `* xB=gamma(xB)` means that the intruder is now allowed to learn the true value of `xB`, say `a`. This meta-notation is not to be confused with an actual function symbol: when we compute the message patterns, we do consider the terms that occur inside the meta-notation, but `gamma` itself is not a function and therefore it is not part of the set of message patterns.

Then we have that this protocol satisfies Definition III.5. There is no destructor application to remove in the initiator transaction. However, for the responder transaction, we apply Definition III.6 to get the following version with pattern matching:

```
Transaction ResponderPat:
* xB in {a,b}.
receive crypt(pk(C:agent),
              f1(A:agent,_:nonce)),
              _:nonce).
new NB:nonce,R:nonce.
if C=xB then
  send crypt(pk(A),f2(NB),R).
  if A=i then
    * xB=gamma(xB)
else
  new AA:agent.
  send crypt(pk(AA),f2(NB),R).
  if A in {a,b,i} and C in {a,b} then
    * not (C=xB and A=i)
```

Thus we have the following message patterns:

$$\begin{aligned} M = \{&\mathsf{inv}(\mathsf{pk}(\mathsf{i})), xA, \mathsf{a}, \mathsf{b}, xB, \mathsf{i}, NA, R,\\ &\mathsf{crypt}(\mathsf{pk}(xB), \mathsf{f}_1(xA, NA), R), xB',\\ &\mathsf{crypt}(\mathsf{pk}(C), \mathsf{f}_1(A, NA'), R'), NB, R'',\\ &C, \mathsf{crypt}(\mathsf{pk}(A), \mathsf{f}_2(NB), R''), A, AA,\\ &\mathsf{crypt}(\mathsf{pk}(AA), \mathsf{f}_2(NB), R'')\} \end{aligned}$$

with the following types for variables and constants:

$$\begin{aligned} &\Gamma(\mathsf{i}) = \Gamma(xA) = \Gamma(\mathsf{a}) = \Gamma(\mathsf{b}) = \Gamma(xB) = \Gamma(xB') = \Gamma(C)\\ &\quad = \Gamma(A) = \Gamma(AA) = \mathsf{agent}\\ &\Gamma(NA) = \Gamma(R) = \Gamma(NA') = \Gamma(R') = \Gamma(NB) = \Gamma(R'')\\ &\quad = \mathsf{nonce} \end{aligned}$$

The set M is type-flaw resistant, and thus Private Authentication (AF0 variant) is type-flaw resistant.

As is done in [15], we can extend AF0 to include a relation talk, where an agent sends a decoy when they do not want to talk to the claimed sender.

```
Sigma0: public a/0 b/0 i/0
        rel talk/2
Sigma: public f1/2 f2/1 df11/1 df12/1
              df2/1
       private recipient/1 sender/1
Types: a:agent b:agent i:agent
gamma0: talk: (a,b),(a,i),(b,a)
Algebra:
recipient(crypt(pk(B),f1(A,NA),R))->B
sender(crypt(pk(B),f1(A,NA),R))->A
df11(f1(X,Y))->X
df12(f1(X,Y))->Y
df2(f2(X))->X

Transaction ReceivePrivateKey:
send inv(pk(i))

Transaction Initiator:
* xA in {a,b}.
* xB in {a,b,i}.
if talk(xA,xB) then
```

```
  new NA:nonce,R:nonce.
  send crypt(pk(xB),f1(xA,NA),R).
  * talk(xA,xB).
  if xB=i then
    * xA=gamma(xA) and xB=gamma(xB)
  else
    * xB in {a,b}
else
  * not talk(xA,xB)

Transaction Responder:
* xB in {a,b}.
receive M:crypt(pk(agent),
                f1(agent,nonce),
                nonce).
try C:=recipient(M) in
try A:=sender(M) in
new NB:nonce,AA:agent,R:nonce.
if C=xB then
  if A=i then
    if talk(xB,A) then
      send crypt(pk(A),f2(NB),R).
      * talk(xB,A) and xB=gamma(xB)
    else
      send crypt(pk(AA),f2(NB),R).
      * not talk(xB,A)
  else
    if A in {a,b} then
      if talk(xB,A) then
        send crypt(pk(A),f2(NB),R)
      else
        send crypt(pk(AA),f2(NB),R)
  else
    send crypt(pk(AA),f2(NB),R)
else
  send crypt(pk(AA),f2(NB),R).
  if A in {a,b,i} and C in {a,b} then
    * not (C=xB and A=i and talk(xB,A))

Transaction ResponderPat:
* xB in {a,b}.
receive crypt(pk(C:agent),
              f1(A:agent,_:nonce),
              _:nonce).
new NB:nonce,AA:agent,R:nonce.
if C=xB then
  if A=i then
    if talk(xB,A) then
      send crypt(pk(A),f2(NB),R).
      * talk(xB,A) and xB=gamma(xB)
    else
      send crypt(pk(AA),f2(NB),R).
      * not talk(xB,A)
  else
    if A in {a,b} then
      if talk(xB,A) then
        send crypt(pk(A),f2(NB),R)
      else
        send crypt(pk(AA),f2(NB),R)
  else
    send crypt(pk(AA),f2(NB),R)
else
  send crypt(pk(AA),f2(NB),R).
  if A in {a,b,i} and C in {a,b} then
    * not (C=xB and A=i and talk(xB,A))
```