Stateful Protocol Composition in Isabelle/HOL

ANDREAS V. HESS, DTU Compute, Technical University of Denmark, Denmark
SEBASTIAN A. MÖDERSHEIM, DTU Compute, Technical University of Denmark, Denmark
ACHIM D. BRUCKER, University of Exeter, United Kingdom

Communication networks like the Internet form a large distributed system where a huge number of components run in parallel, such as security protocols and distributed web applications. For what concerns security, it is obviously infeasible to verify them all at once as one monolithic entity; rather, one has to verify individual components in isolation.

While many typical components like TLS have been studied intensively, there exists much less research on analyzing and ensuring the security of the composition of security protocols. This is a problem since the composition of systems that are secure in isolation can easily be insecure. The main goal of compositionality is thus a theorem of the form: given a set of components that are already proved secure in isolation and that satisfy a number of easy-to-check conditions, then also their parallel composition is secure. Said conditions should of course also be realistic in practice, or better yet, already be satisfied for many existing components. Another benefit of compositionality is that when one would like to exchange a component with another one, all that is needed is the proof that the new component is secure in isolation and satisfies the composition conditions—without having to re-prove anything about the other components.

This paper has three contributions over previous work in parallel compositionality. First, we extend the compositionality paradigm to *stateful systems*: while previous approaches work only for simple protocols that only have a local session state, our result supports participants who maintain long-term *databases* that can be *shared* among several protocols. This includes a paradigm for *declassification of shared secrets*. This result is in fact so general that it also covers many forms of *sequential composition* as a special case of stateful parallel composition. Second, our compositionality result is formalized and proved in Isabelle/HOL, providing a strong correctness guarantee of our proofs. This also means that one can prove, without gaps, the security of an entire system in Isabelle/HOL, namely the security of components in isolation, the composition conditions, and thus derive the security of the entire system as an Isabelle theorem. For the components one can also make use of our tool PSPSP that can perform automatic proofs for many stateful protocols. Third, for the compositionality conditions we have also implemented an automated check procedure in Isabelle.

CCS Concepts: • **Networks** → **Security protocols**; **Protocol testing and verification**; • **Security and privacy** → **Formal security models**; *Logic and verification.*

Additional Key Words and Phrases: protocol composition, stateful security protocol, Isabelle/HOL

ACM Reference Format:
Andreas V. Hess, Sebastian A. Mödersheim, and Achim D. Brucker. 2022. Stateful Protocol Composition in Isabelle/HOL. 1, 1 (December 2022), 35 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Andreas V. Hess, DTU Compute, Technical University of Denmark, Richard Petersens Plads, Building 324, Kgs. Lyngby, DK-2800, Denmark, avhe@dtu.dk; Sebastian A. Mödersheim, DTU Compute, Technical University of Denmark, Richard Petersens Plads, Building 324, Kgs. Lyngby, DK-2800, Denmark, samo@dtu.dk; Achim D. Brucker, University of Exeter, Streatham Campus, Innovation Centre, Rennes Drive, Exeter, EX4 4RN, United Kingdom, a.brucker@exeter.ac.uk.

XXXX-XXXX/2022/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The typical use of communication networks like the Internet is to run a wide variety of security protocols in parallel. While the security properties of many of these protocols, e.g., TLS, have been analyzed in great detail, much less research has been devoted to their parallel composition.

It is far from self-evident that the parallel composition of secure protocols is still secure, in fact one can systematically construct counter-examples. One such problem is if protocols have similar message structures of different meaning, so that an attacker may be able to abuse messages, or parts thereof, that they have learned in the context of one protocol, and use them in the context of another where the same structure has a different meaning. Thus, we have to exclude that the protocols in some sense "interfere" with each other. However, it is unreasonable to require that the developers of the different protocols have to work together and synchronize with each other. Similarly, we do not want to reason about the composition of several protocols as a whole, neither in manual nor automated verification. Instead, we want a set of sufficient conditions and a composition theorem of the form: every set of protocols that satisfies the conditions yields a secure composition, provided that each protocol is secure in isolation. The conditions should be realistic so that many existing protocols like TLS actually satisfy them, and they should be simple, in the sense that checking them is a static task that does not involve considering the reachable states.

One main contribution of this paper is the extension of the compositionality paradigm to *stateful* protocols, where participants may maintain a database (e.g., a list of valid public keys). Such databases do not necessarily grow monotonically during protocol execution—we allow, e.g., negative membership checks and deletion of elements from databases. Moreover, we allow databases to be *shared* between the protocols to be composed. For instance, in the example of public keys, there could be several protocols for registering, certifying, and revoking keys that all work on the same public-key database. Since such a shared database can potentially be exploited by the intruder to trigger harmful interference, an important part of our result is a clear coordination of the ways in which each protocol is allowed to access the database. This coordination is based on assumptions and guarantees on the transactions that involve the database. Moreover, this also allows us to support protocols with the declassification of long-term secrets (e.g., that the private key to a revoked public key may be learned by the intruder without breaking the security goals). The result is so general that it actually also covers many forms of *sequential composition* as a special case, since one can for instance model that one protocol inserts keys into a database of fresh session keys, and another protocol "consumes" and uses them. In more detail, our main contributions are:

(1) We extend the compositionality paradigm to *stateful protocols*. In particular, our result supports participants who maintain long-term *databases* that can be *shared* among several protocols, and a paradigm for *declassification of shared secrets*. Our result is so general that it also covers various forms of *sequential composition* as a special case.
(2) Our compositionality result is formalized and proved in the interactive theorem prover Isabelle/HOL, providing a strong correctness guarantee of our proofs. This means that one can prove the security of a composed protocol in Isabelle by proving the security of the components in isolation and checking the compositionality conditions, and thus derive the security of the composition as an Isabelle theorem.
(3) We implemented checks for the compositionality conditions in Isabelle/HOL, so that they can be checked automatically.
(4) We have connected the compositionality result to our tool PSPSP [25] that can perform automatic proofs for many stateful protocols in Isabelle. This extends PSPSP (for protocols supported by PSPSP) with a composable verification method.

All of our theory and proofs are published and maintained in the Archive of Formal Proofs (AFP) [28, 30]. The overall formalization is over 27000 lines of code (over 8000 more lines than the conference version) and took about 36 person months to develop.

The main advancements of this paper over the underlying conference version [31] are: Firstly, we have extended the Isabelle proofs; the result is now completely formalized in Isabelle. Secondly, we generalized our formalization to use an arbitrary set of labels, providing improved support for composing more than two protocols. Thirdly, we added support for a transaction-based protocol model and its protocol specification language by connecting our work to PSPSP [25]. Fourthly, we implemented automated checks in Isabelle/HOL of the compositionality requirements. Moreover, there are numerous smaller enhancements such as supporting a more general constraint syntax and an improved leakage definition.

The rest of the paper is organized as follows. In Sec. 2 we introduce our specification language for stateful protocols and define its semantics via intruder constraints. In Sec. 3 we summarize a typing result, namely that every satisfiable constraint has a *well-typed* model, provided that the protocol fulfills some simple conditions. In Sec. 4, we introduce composition of security protocols by an example. In Sec. 5 we formally define protocol composition, in particular the concept of shared terms, declassification, and the requirements on disjointness. We then present our main result for parallel composition. The idea is to show that, if an intruder constraint of a composition is satisfiable, then the projections to each component are satisfiable, again provided that the components fulfill a number of easy-to-check conditions. In Sec. 6, we explain how checking these compositionality conditions can be automated. As a last contribution of the paper, we discuss in Sec. 7 how our result for parallel composition can also be applied to other forms of protocol composition—in particular, sequential composition. Finally, we conclude in Sec. 8 and discuss related work.

2 LANGUAGE AND MODEL

We start with the first contribution: a slight generalization of the specification language for stateful protocols that we introduced in [25, 31]. With *stateful* we mean that the protocols do not necessarily consist of independent sessions but that participants may maintain databases that are modeled as sets of messages. In every transaction, besides sending and receiving messages, one can retrieve messages from the databases, or modify the databases. In [25] we have developed a first version of this language for automated verification in Isabelle/HOL. This first version was tailored to the needs of automated verification, in particular, it requires to fix the number of sets, and the elements of the sets have to be atomic values. For the present work, we do not need these restrictions and thus generalize the language. We also introduce some additional notation that is later useful for the parallel composition.

We now first define this language by example and then give its semantics by translating to a symbolic transition system with intruder constraints (which we also formally define). The constraint representation is in fact a key idea to proving the compositionality result.

A *protocol specification* consists of the following elements:

- Enumerations, which are named (potentially infinite) sets of constants like user = {a, b, i}.
- A set of atomic types like pkey. The types are initially only an annotation and the semantics does not take them into account. We will, however, in Sec. 3 use them in a typing result.
- A declaration of the available function symbols, e.g., sign. With every function symbol we also specify the arity, whether the function symbol is public or private (informally, *public* function symbols can be applied by the intruder, if they know the necessary arguments; in contrast, *private* function symbols cannot be applied), and an analysis theory (i.e., under

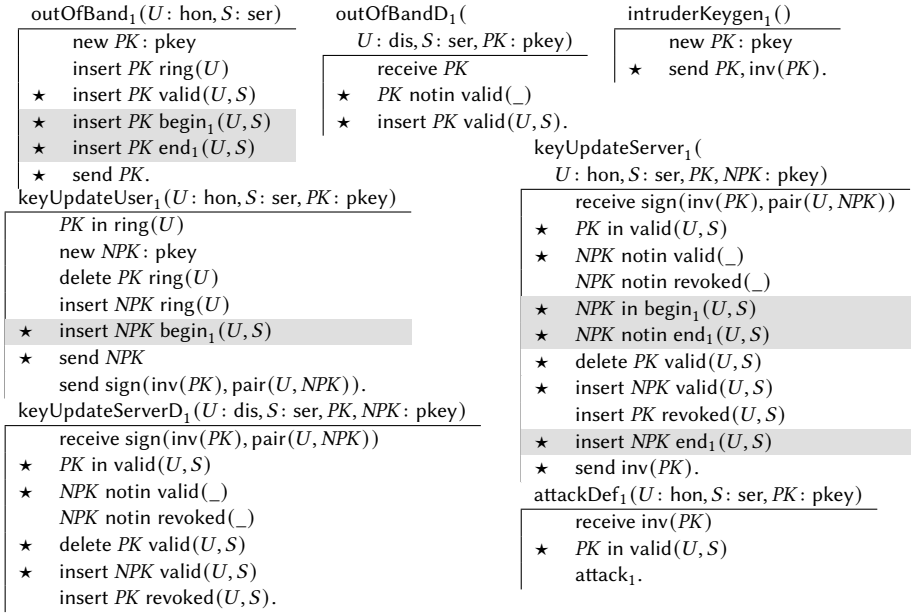

Fig. 1. A keyserver protocol consisting of seven transactions.

what circumstances the intruder can obtain parts of such messages). This will be defined in detail below in Sec. 2.2.2.

- A declaration of sets that the protocol participants can insert messages into, remove from, and check for containment. In order to allow an infinite number of such sets, e.g. key set ring(a) for every agent a, we declare a number of function symbols with their arities here (e.g. ring/1). While PSPSP [25] is limited to enumeration constants as parameters to the set function symbols, the compositionality result allows for arbitrary messages, both as arguments of the set function symbols and as contents of the sets.
- Last but not least, we have to specify a set of transactions.

A transaction consists of any combination of the following: input messages to receive, checks on the sets, modifications of the sets, and output messages to send. A transaction can only be executed atomically, i.e., it can only fire when input messages are present, such that the checks are satisfied, and then they produce all changes and the output messages in one state transition. Security goals are formulated by a special kind of *attack transactions*: they consist only of receiving messages and checks on sets and then reveal a special constant attack to the intruder. (In fact, later in protocol composition, we will instead have attack$_p$ where p is the name or index of the protocol specifying the attack rule, so we can easily talk about which protocol's goal has been violated.) We say a protocol is *secure* if the intruder cannot obtain attack in any protocol run.

2.1 A Keyserver Example

Before we proceed with the formal definitions we illustrate the transaction language through a simple keyserver example (adapted from [25, 31]) shown in Fig. 1. Here users can register public keys at trusted keyservers and these keys can later be revoked. The users are modeled with two enumerations: one named hon containing the honest users, and dis containing the dishonest users. The set of keyservers is defined as the enumeration ser. Each honest user U has an associated

keyring $\mathsf{ring}(U)$ with which it keeps track of its keys. (The elements of $\mathsf{ring}(U)$ are actually public keys; we implicitly assume that the user U knows the corresponding private key.)

First, in the protocol, we model a mechanism $\mathsf{outOfBand}_1$ by which an honest user U can register a new key PK at the keyserver S out-of-band, e.g., by physically visiting the keyserver. The user U first constructs a fresh public key PK and inserts PK into its keyring $\mathsf{ring}(U)$. We model that the keyserver—in the same transaction—learns the key and adds it to its database of valid keys for user U, i.e., into a set $\mathsf{valid}(U, S)$. Finally, PK is published. There are also several gray-shaded actions in the protocol; we explain these later in Sec. 4 when we compose the keyserver protocol with another protocol—similarly, we defer the explanation of the $\star$ annotations until later (in a nutshell, the $\star$-labeled actions are those that are relevant to all component protocols because they act on shared sets or terms). There is also a corresponding rule of $\mathsf{outOfBand}_1$ for dishonest users: $\mathsf{outOfBandD}_1$. Dishonest users may register in their name any key they know (hence the receive PK action), so the key is not necessarily freshly created; also we do not model a key ring for them. However, dishonest users can only register a key PK out of band if it is not yet known as valid at the server. The same check is not necessary in the honest out-of-band rule, because there the key is guaranteed to be freshly created. This rule allows the dishonest agents to use any known key that is currently not valid and register it, but so far they could only use keys that others have created; we thus enter also a rule that allows the dishonest agents to generate arbitrarily many fresh key pairs: $\mathsf{intruderKeygen}_1$.

Note that there is no built-in notion of set ownership, or who exactly is performing an action: we just specify with such transactions what can happen. The intuition in this example is that $\mathsf{ring}(U)$ is a set of public keys controlled by U (and U has the corresponding private key of each) while $\mathsf{valid}(U, S)$ is controlled by server S. By putting it into a single transaction we model that this happens in collaboration between a user and a server.

We model a key update mechanism that allows for registering a new key while simultaneously revoking an old one. Here we model this as three transactions, one for the (honest) users $\mathsf{keyUpdateUser}_1$ and two for the servers $\mathsf{keyUpdateServer}_1$ and $\mathsf{keyUpdateServerD}_1$ (one for each kind of user), since here we model a scenario where user and server communicate via an asynchronous network controlled by the intruder. To initiate the key revocation process with transaction $\mathsf{keyUpdateUser}_1$ the honest user U first picks and removes a key PK from its keyring to later revoke, then freshly generates a new key NPK and stores it in its keyring. (Again the corresponding private key $\mathsf{inv}(NPK)$ is known to U, but this is not explicitly described.) As a final step the user signs the new key with the private key $\mathsf{inv}(PK)$ of the old key and sends this signature to the server S by transmitting it over the network. The check PK in $\mathsf{ring}(U)$ represents here a nondeterministic choice of an element of $\mathsf{ring}(U)$. (A user can register any number of keys with the $\mathsf{outOfBand}_1$ transaction.) For $\mathsf{keyUpdateUser}_1$ there is no equivalent for the dishonest agents, since the intruder's ability to craft update request messages from their knowledge already subsumes this.

Continuing in the protocol execution with $\mathsf{keyUpdateServer}_1$ (which is defined for honest users—$\mathsf{keyUpdateServerD}_1$ is the pendant for dishonest users), when server S receives the signed message, it checks that PK is indeed a valid key, that NPK has not been registered earlier, and then revokes PK and registers NPK. To keep track of revoked keys, S maintains another database $\mathsf{revoked}(U, S)$ containing the revoked keys of U at S. In this transaction the notation x notin $s(\_)$ expresses that x must not be in any set of the form $s(c)$. As a last action, the old private key $\mathsf{inv}(PK)$ is revealed. This is of course not what one would do in a reasonable implementation, but it allows us to prove that the protocol is correct even if the intruder obtains all private keys to revoked public keys. (This could also be separated into a rule that just leaks private keys of revoked keys.) The difference in the dishonest-user version, $\mathsf{keyUpdateServerD}_1$, is only in the goals (that we will discuss later in Sec. 4) and that we do not reveal $\mathsf{inv}(PK)$ (because it is a key associated to the intruder anyway).

Finally, we define that there is an attack if the intruder learns a valid key of an honest user. This, again, can be modeled as a sequence of actions in which we check if the conditions for an attack hold, and, if so, transmit the constant attack_1 that acts as a signal for goal violations. This is all defined in $\mathsf{attackDef}_1$ where the last action attack_1 is just syntactic sugar for $\mathsf{send}\,\mathsf{attack}_1$.

2.2 Symbolic Constraints and Intruder Model

We now define our formal model in a bottom-up fashion, starting with the intruder constraints and conclude with a formal definition of the transactions we have seen in the previous section.

2.2.1 Terms and Substitutions. A transaction specification gives rise to a set Σ of function symbols with their arities (where constants are function symbols of arity 0): these are the functions and constants explicitly declared, the countable set of constants that arise from infinite enumerations like user = {..}, and freshly generated constants during protocol execution. Disjoint from Σ, we have a countable set $\mathcal{V}$ of variables (that includes the ones from the specification). By convention, in protocol specifications, elements of Σ start with a lower-case letter and are written in sans serif style, and elements of $\mathcal{V}$ start with an upper-case letter.

A *term* is either a variable $x \in \mathcal{V}$ or a composed term of the form $f(t_1, \ldots, t_n)$ where $f \in \Sigma^n$ and t_i are terms and Σ^n denotes the symbols in Σ of *arity n*. We also write C for Σ^0, the set of constants. The set of variables of a term t is denoted by $fv(t)$ and if $fv(t) = \emptyset$ then t is *ground*. Both of these notions are extended to sets of terms. By $\sqsubseteq$ we denote the *subterm* relation.

Substitutions are defined as functions from variables to terms. The *domain* of a substitution δ is denoted by $dom(\delta)$ and is defined as the set of variables that are not mapped to themselves by δ: $dom(\delta) \equiv \{x \in \mathcal{V} \mid \delta(x) \neq x\}$. The substitution range, $ran(\delta)$, is then defined as the range of $dom(\delta)$ under δ: $ran(\delta) \equiv \delta(dom(\delta))$. If the range of δ is ground then δ is said to be *ground*. If $ran(\delta) \subseteq \mathcal{V}$ and δ is injective then δ is a *variable-renaming*. Additionally, we define an *interpretation* to be a substitution that assigns a ground term to every variable: $\mathcal{I}$ is an interpretation iff $dom(\mathcal{I}) = \mathcal{V}$ and $ran(\mathcal{I})$ is ground. We extend substitutions to functions on terms and sets of terms as expected. The substitution *composition* $\delta \cdot \sigma$ of two substitutions δ and σ is defined as $(\delta \cdot \sigma)(t) \equiv \sigma(\delta(t))$. For δ with finite domain we usually use the common value mapping notation: $\delta = [x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$. Finally, a substitution δ is a *unifier of* terms t and t' iff $\delta(t) = \delta(t')$.

2.2.2 The Intruder Model. We use a Dolev-Yao-style intruder model, i.e., the intruder can encrypt and decrypt terms where they have the respective keys, but they cannot break the cryptography. We define the intruder deduction relation $\vdash$ as the least relation satisfying the rules of Definition 2.1:

Definition 2.1 (Intruder model).

$$\frac{}{M \vdash t}\ \begin{array}{l}(Axiom),\\ t \in M\end{array} \qquad \frac{M \vdash t_1 \quad \cdots \quad M \vdash t_n}{M \vdash f(t_1, \ldots, t_n)}\ \begin{array}{l}(Compose),\\ f \in \Sigma^n_{pub}\end{array}$$

$$\frac{M \vdash t \quad M \vdash k_1 \quad \cdots \quad M \vdash k_n}{M \vdash r}\ \begin{array}{l}(Decompose), \mathsf{Ana}(t) = (K, R),\\ r \in R, K = \{k_1, \ldots, k_n\}\end{array}$$

□

Here $M \vdash t$ means that an intruder who knows the set of terms M can derive the term t. The rule (*Axiom*) simply says that the intruder can derive any term that is already in their knowledge M. For composing messages it is common to give a rule for each operator, like encryption, that is available to the intruder. In order to have a result that is parameterized over an arbitrary signature Σ of functions—so one does not need to adapt the proofs of the compositionality result whenever a new operator shall be added—the modeler should simply declare a subset of Σ to be *public* functions, and we write Σ^n_{pub} for the public functions that take n arguments. The rule (*Compose*) now says

that the intruder can apply public functions to any derivable terms and the result is also derivable. In fact most functions on messages, like encryption and signing, will be public, while we will have a private function inv that maps public keys to their corresponding private keys. Obviously if inv were a public function, then the (*Compose*) rule would allow the intruder to know all private keys to which they know the public keys and trivially break all asymmetric cryptography.

For decryption, (*Decompose*), things are a bit more involved. Many approaches use algebraic reasoning here, e.g. with equations like $\mathsf{dcrypt}(\mathsf{inv}(x), \mathsf{crypt}(x, y)) = y$, but this is rather complex to formalize in Isabelle and it is not really necessary for many standard operators. The idea is that one does not necessarily use an explicit decryption operator, but rather has further decryption rules of the form: if the intruder knows $\mathsf{crypt}(x, y)$ and $\mathsf{inv}(x)$, then they also know y. To formulate such rules without specializing to particular operators, we ask the modeler to define a function Ana such that $\mathsf{Ana}(t) = (K, R)$ means: the given term t allows for decryption if the intruder knows all the keys in K, then they obtain every message in R. This is formalized by the rule (*Decompose*).

Example 2.2. For most of the paper, we will use the public function symbols crypt, sign, upd, and pair and the private symbol inv. In addition, we model the set-family symbols—ring, valid, and revoked—as private symbols. Decomposition of terms we model with the following Ana theory:

$$\begin{array}{lcllcl} \mathsf{Ana}(\mathsf{crypt}(k,m)) & = & (\{\mathsf{inv}(k)\},\{m\}), & \mathsf{Ana}(\mathsf{upd}(s,t,u,v)) & = & (\emptyset,\{s,t,u,v\}),\\ \mathsf{Ana}(\mathsf{sign}(k,m)) & = & (\emptyset,\{m\}), & \mathsf{Ana}(\mathsf{pair}(t,t')) & = & (\emptyset,\{t,t'\}), \end{array}$$

and where $\mathsf{Ana}(t) = (\emptyset, \emptyset)$ for all other terms t.

This means that for decrypting a message encrypted with public key k, one needs the corresponding private key $\mathsf{inv}(k)$. In a signature $\mathsf{sign}(k, m)$, m can directly be retrieved without knowing any key. This models signature schemes where m is provided in clear along with the signed hash of m. Note that this is independent of *signature verification* which is done by pattern matching in the transaction rules like in all free-algebra approaches. Consider for instance in the keyserver example the rule keyUpdateServer, where S receives $\mathsf{sign}(\mathsf{inv}(PK), \mathsf{pair}(U, NPK))$: here the server S accepts only a signature with *some* private key $\mathsf{inv}(PK)$ of a cleartext that has to be a pair and the components are any values U and NPK. Then the server checks that PK in $\mathsf{valid}(U, S)$ and that NPK is not yet known. Note that operationally, the server would first obtain the cleartext of the signature, obtain every public key PK associated with the (claimed) user U, and check if the signature verifies with some PK. With upd and pair we have *transparent* functions: they are public and one can obtain all their arguments without knowing a key. □

Our results rely on the following requirements on Ana:

(1) $\mathsf{Ana}(t) = (\emptyset, \emptyset)$ for $t \in \mathcal{V} \cup \mathcal{C}$, i.e. for variables and constants,
(2) $\mathsf{Ana}(f(t_1, \ldots, t_n)) = (K, R)$ implies $R \subseteq \{t_1, \ldots, t_n\}$, finite K, and $fv(K) \subseteq fv(f(t_1, \ldots, t_n))$,
(3) $\mathsf{Ana}(f(t_1, \ldots, t_n)) = (K, R)$ implies $\mathsf{Ana}(\delta(f(t_1, \ldots, t_n))) = (\delta(K), \delta(R))$.

Ana must be defined for arbitrary terms, including terms with variables (while the standard Dolev-Yao deduction is typically applied to ground terms). The three conditions regulate that Ana is also meaningful on symbolic terms. The first requirement says that we cannot analyze a variable or a constant. The second requirement says that the results of the analysis are *immediate* subterms of the term being analyzed, and the keys can be any finite set of terms (including composed terms, e.g., $\mathsf{inv}(k)$), but built with only variables that occur in the term being analyzed. We use the fact that for most constructor-destructor theories it holds that the intruder cannot learn something new from encrypting a term and then decrypting it again. Without the second condition, we would however allow for theories that violate this principle, e.g., $\mathsf{Ana}(f(g(x))) = (\emptyset, \{x\})$, an intruder who knows $g(c)$ could obtain c by first applying f. The third requirement says that analysis does not change its behavior when instantiating a term (that is not a variable).

2.2.3 Symbolic Constraints. To reason about protocol executions, we use *symbolic constraints* which essentially are a sequence of transactions (with their variables appropriately renamed to avoid clashes) from the point of view of the intruder: every message sent by a transaction is received by the intruder, and every message received by a transaction is sent by the intruder. This is a constraint in that every message the intruder sends must be generated from the messages that the intruder has received before that point. An attack is defined by satisfiability of a constraint in which the intruder produces a special secret.

Formally, *symbolic constraints* are defined as finite sequences $\mathcal{A}$ of *actions* built from the following grammar where t and t' range over terms and $\bar{x}$ over finite variable sequences $x_1, \ldots, x_n$:

$$\begin{array}{lll} \mathcal{A} & ::= & \mathsf{send}\ t_1, \ldots, t_n \cdot \mathcal{A} \mid \mathsf{receive}\ t_1, \ldots, t_n \cdot \mathcal{A} \mid t \doteq t' \cdot \mathcal{A} \mid t\ \mathsf{in}\ t' \cdot \mathcal{A} \mid \\ & & (\forall \bar{x}.\ \phi_{\mathrm{neg}}) \cdot \mathcal{A} \mid \mathsf{insert}\ t\ t' \cdot \mathcal{A} \mid \mathsf{delete}\ t\ t' \cdot \mathcal{A} \mid 0 \\ \phi_{\mathrm{neg}} & ::= & t \not\doteq t' \mid t\ \mathsf{notin}\ t' \mid \phi_{\mathrm{neg}} \vee \phi'_{\mathrm{neg}} \end{array}$$

Instead of $\forall \bar{x}.\ \phi_{\mathrm{neg}}$ we may write ϕ_{neg} whenever $\bar{x}$ is the empty sequence. We may also write $t\ \mathsf{notin}\ f(\_)$ for $f \in \Sigma^n$ as an abbreviation of $\forall x_1, \ldots, x_n.\ t\ \mathsf{notin}\ f(x_1, \ldots, x_n)$ where the x_i do not occur in t. The *intruder knowledge* of a constraint $\mathcal{A}$, denoted by $ik(\mathcal{A})$, is defined as the set of those terms that occur in receive actions of $\mathcal{A}$. The *bound variables* of $\mathcal{A}$ consist of its variable sequences while the remaining variables, $fv(\mathcal{A})$, are the *free variables*. Also, by $trms(\mathcal{A})$ we denote the set of terms occurring in $\mathcal{A}$ and the *set of set operations* of $\mathcal{A}$, called $setops(\mathcal{A})$, is defined as follows where $(\cdot,\cdot) \in \Sigma^2_{pub}$ is a fixed function symbol:

$$\begin{array}{l} setops(\mathcal{A}) \equiv \{(t,s) \mid \mathsf{insert}\ t\ s \text{ or } \mathsf{delete}\ t\ s \text{ or } t\ \mathsf{in}\ s \text{ occurs in } \mathcal{A}, \text{ or there exists} \\ \qquad\qquad \phi_{\mathrm{neg}} \text{ and } \bar{x} \text{ such that } t\ \mathsf{notin}\ s \text{ occurs in } \phi_{\mathrm{neg}} \text{ and } \forall \bar{x}.\ \phi_{\mathrm{neg}} \text{ occurs in } \mathcal{A}\} \end{array}$$

For the semantics of constraints we first define a predicate $[\![M, D; \mathcal{A}]\!]\ \mathcal{I}$, where M is a ground set of terms (the intruder knowledge), D is a ground set of tuples (the state of the sets), $\mathcal{A}$ is a constraint, and $\mathcal{I}$ is an interpretation as follows:

$$\begin{array}{rll} [\![D; t \not\doteq t']\!]\ \mathcal{I} & \text{iff} & \mathcal{I}(t) \neq \mathcal{I}(t') \\ [\![D; t\ \mathsf{notin}\ s]\!]\ \mathcal{I} & \text{iff} & \mathcal{I}((t,s)) \notin D \\ [\![D; \phi_1 \vee \phi_2]\!]\ \mathcal{I} & \text{iff} & [\![D; \phi_1]\!]\ \mathcal{I} \text{ or } [\![D; \phi_2]\!]\ \mathcal{I} \\ [\![M, D; 0]\!]\ \mathcal{I} & \text{iff} & \mathit{true} \\ [\![M, D; \mathsf{send}\ t_1, \ldots, t_n \cdot \mathcal{A}]\!]\ \mathcal{I} & \text{iff} & M \vdash \mathcal{I}(t_i) \text{ for all } i \in \{1, \ldots, n\}, \text{ and } [\![M, D; \mathcal{A}]\!]\ \mathcal{I} \\ [\![M, D; \mathsf{receive}\ t_1, \ldots, t_n \cdot \mathcal{A}]\!]\ \mathcal{I} & \text{iff} & [\![\{\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n)\} \cup M, D; \mathcal{A}]\!]\ \mathcal{I} \\ [\![M, D; t \doteq t' \cdot \mathcal{A}]\!]\ \mathcal{I} & \text{iff} & \mathcal{I}(t) = \mathcal{I}(t') \text{ and } [\![M, D; \mathcal{A}]\!]\ \mathcal{I} \\ [\![M, D; \mathsf{insert}\ t\ s \cdot \mathcal{A}]\!]\ \mathcal{I} & \text{iff} & [\![M, \{\mathcal{I}((t,s))\} \cup D; \mathcal{A}]\!]\ \mathcal{I} \\ [\![M, D; \mathsf{delete}\ t\ s \cdot \mathcal{A}]\!]\ \mathcal{I} & \text{iff} & [\![M, D \setminus \{\mathcal{I}((t,s))\}; \mathcal{A}]\!]\ \mathcal{I} \\ [\![M, D; t\ \mathsf{in}\ s \cdot \mathcal{A}]\!]\ \mathcal{I} & \text{iff} & \mathcal{I}((t,s)) \in D \text{ and } [\![M, D; \mathcal{A}]\!]\ \mathcal{I} \\ [\![M, D; (\forall \bar{x}.\ \phi) \cdot \mathcal{A}]\!]\ \mathcal{I} & \text{iff} & [\![D; \phi]\!]\ (\delta \cdot \mathcal{I}) \text{ for all ground substitutions } \delta \text{ with domain } \bar{x}, \\ & & \text{and } [\![M, D; \mathcal{A}]\!]\ \mathcal{I} \end{array}$$

We then define that $\mathcal{I}$ is a *model* of $\mathcal{A}$, written $\mathcal{I} \models \mathcal{A}$, iff $[\![\emptyset, \emptyset; \mathcal{A}]\!]\ \mathcal{I}$.

It is common in symbolic approaches to require a form of *well-formedness* for constraints. First, one typically requires that the intruder knowledge grows monotonically over time. This is already built-in in our formalism: observe that the intruder knowledge M in the above semantics has indeed this property. Second, we require that every variable first occurs in a message the intruder sends, or in a positive check like $t \doteq t'$ or $t\ \mathsf{in}\ s$. This is formalized as follows:

Definition 2.3 (Constraint well-formedness). A constraint $\mathcal{A}$ is *well-formed w.r.t.* the set of variables X (or just *well-formed* if $X = \emptyset$) iff the free variables and the bound variables of $\mathcal{A}$ are disjoint and

$wf_X(\mathcal{A})$ holds where:

$$\begin{array}{lll}
wf_X(0) & \text{iff} & \mathit{true} \\
wf_X(\mathsf{receive}\ t_1, \ldots, t_n \cdot \mathcal{A}) & \text{iff} & fv(\{t_1, \ldots, t_n\}) \subseteq X \text{ and } wf_X(\mathcal{A}) \\
wf_X(\mathsf{send}\ t_1, \ldots, t_n \cdot \mathcal{A}) & \text{iff} & wf_{X \cup fv(\{t_1, \ldots, t_n\})}(\mathcal{A}) \\
wf_X(t \doteq t' \cdot \mathcal{A}) & \text{iff} & \begin{cases} wf_{X \cup fv(t)}(\mathcal{A}) & \text{if } fv(t') \subseteq X \\ wf_X(\mathcal{A}) & \text{otherwise} \end{cases} \\
wf_X(\mathsf{insert}\ t\ t' \cdot \mathcal{A}) & \text{iff} & fv(t) \cup fv(t') \subseteq X \text{ and } wf_X(\mathcal{A}) \\
wf_X(t\ \mathsf{in}\ t' \cdot \mathcal{A}) & \text{iff} & wf_{X \cup fv(t) \cup fv(t')}(\mathcal{A}) \\
wf_X(\mathfrak{a} \cdot \mathcal{A}) & \text{iff} & wf_X(\mathcal{A}) \text{ otherwise}
\end{array}$$ □

This allows to "introduce" variables in a send action, on the left-hand side of an equation, or in a positive set-membership check. In this paper, we will work only with well-formed constraints.

Example 2.4. The constraints $\mathsf{send}\ x \cdot \mathsf{receive}\ f(x)$ and $\mathsf{receive}\ c \cdot x \doteq f(c) \cdot \mathsf{send}\ x$ and $x\ \mathsf{in}\ set \cdot \mathsf{send}\ x$ are all well-formed, while $\mathsf{receive}\ x$ is not: it would correspond to the intruder receiving an arbitrary (unconstrained) value. □

2.3 Syntax and Semantics of Protocols

We now define the syntax of transactions in our transaction-based language as follows:

Definition 2.5. A *transaction* has the form $\mathfrak{t}(x_1 : T_1, \ldots, x_n : T_n)\ \phi_t$ where ϕ_t is a *transaction strand* built from the following grammar where t, t', t_i range over terms, τ over atomic types, x over variables, $\bar{x}$ over variable sequences, and where $(\ \psi\ )^*$ denotes finite sequences of the form $\psi \cdot \ldots \cdot \psi$:

$$\begin{array}{llllll}
\phi_t & ::= & \phi_r \cdot \phi_c \cdot \phi_f \cdot \phi_m \cdot \phi_s & \phi_f & ::= & (\ \mathsf{new}\ x : \tau\ )^* \\
\phi_r & ::= & (\ \mathsf{receive}\ t_1, \ldots, t_n\ )^* & \phi_m & ::= & (\ \mathsf{insert}\ t\ t' \mid \mathsf{delete}\ t\ t'\ )^* \\
\phi_c & ::= & (\ t \doteq t' \mid t\ \mathsf{in}\ t' \mid \forall \bar{x}.\ \phi_n\ )^* & \phi_s & ::= & (\ \mathsf{send}\ t_1, \ldots, t_n\ )^* \\
\phi_n & ::= & t \not\doteq t' \mid t\ \mathsf{notin}\ t' \mid \phi_n \vee \phi_n' & & &
\end{array}$$

The prefix $\mathfrak{t}(x_1 : T_1, \ldots, x_n : T_n)$ gives the transaction a name $\mathfrak{t}$ and declares parameters x_i. The corresponding T_i are either enumerations or types from the type system we define in the following section. In the case of types, this is just an annotation irrelevant for the semantics for now, but in the case of enumerations, we consider the instantiation of the transaction with any substitution σ such that $\sigma(x_i) \in T_i$ for those i where T_i is an enumeration. The functions $trms(\cdot)$ and $setops(\cdot)$ are extended to transactions, rules, and protocols as expected. □

Example 2.6. Consider the $\mathsf{keyUpdateServer}_1$ transaction from Sec. 2.1. With the transaction syntax just defined, it could be written as follows:

$$\mathsf{keyUpdateServer}_1(U : \mathsf{hon}, S : \mathsf{ser}, PK : \mathsf{pkey}, NPK : \mathsf{pkey})\ \phi_r \cdot \phi_c \cdot \phi_m \cdot \phi_s$$

where its transaction strand $\phi_r \cdot \phi_c \cdot \phi_m \cdot \phi_s$ consists of the following action sequences:

$$\begin{array}{lll}
\phi_r & = & \mathsf{receive}\ \mathsf{sign}(\mathsf{inv}(PK), \mathsf{pair}(U, NPK)) \\
\phi_c & = & PK\ \mathsf{in}\ \mathsf{valid}(U, S) \cdot \forall X, Y.\ NPK\ \mathsf{notin}\ \mathsf{valid}(X, Y) \cdot \forall X, Y.\ NPK\ \mathsf{notin}\ \mathsf{revoked}(X, Y) \cdot \\
 & & NPK\ \mathsf{in}\ \mathsf{begin}_1(U, S) \cdot NPK\ \mathsf{notin}\ \mathsf{end}_1(U, S) \\
\phi_m & = & \mathsf{delete}\ PK\ \mathsf{valid}(U, S) \cdot \mathsf{insert}\ PK\ \mathsf{revoked}(U, S) \cdot \mathsf{insert}\ NPK\ \mathsf{valid}(U, S) \cdot \\
 & & \mathsf{insert}\ NPK\ \mathsf{end}_1(U, S) \\
\phi_s & = & \mathsf{send}\ \mathsf{inv}(PK)
\end{array}$$ □

Note that the actions that ϕ_t is composed of are very similar to the intruder constraints we have introduced before. In fact, we will now define the semantics of transactions in terms of the intruder

constraints that they induce: to execute a transaction, we will add a copy (with variables freshly renamed) to the intruder constraint with just two alterations: for every new $x\colon \tau$, we instantiate x with a fresh constant (by default, our model is untyped, and the use of typing is discussed in the next section). Every message that is sent by the transaction is received by the intruder and vice-versa. For that purpose, we define the *dual* of a transaction strand or constraint S, written $dual(S)$, as "swapping" the direction of the sent and received messages of S:

$$\begin{aligned} dual(\mathsf{send}\ t_1, \ldots, t_n \cdot S) &= \mathsf{receive}\ t_1, \ldots, t_n \cdot dual(S), \\ dual(\mathsf{receive}\ t_1, \ldots, t_n \cdot S) &= \mathsf{send}\ t_1, \ldots, t_n \cdot dual(S), \text{ and} \\ dual(\mathfrak{s} \cdot S) &= \mathfrak{s} \cdot dual(S) \text{ otherwise.} \end{aligned}$$

Note that we impose an order on the kind of actions in a transaction: we start with receiving terms (ϕ_r), then perform checks (ϕ_c), then generate fresh constants (ϕ_f), then manipulate sets (ϕ_m), and finally send out some messages (ϕ_s). In fact, transactions are atomic in the sense that if any of the requirements fail, the transaction is not performed at all (so the state of all sets remains unchanged), otherwise it is performed as a whole.

Definition 2.7. Given a protocol $\mathcal{P}$, the semantics is a transition relation $\Rightarrow_{\mathcal{P}}$ where states are constraints, the initial state is the empty constraint 0, and the transition relation $\Rightarrow_{\mathcal{P}}$ is defined as follows: $\mathcal{A} \Rightarrow_{\mathcal{P}} \mathcal{A} \cdot dual(\alpha(\sigma(\phi_r \cdot \phi_c \cdot \phi_m \cdot \phi_s)))$ holds iff there exist t, x_i, T_i, ϕ_f such that:

(1) $\mathsf{t}(x_1\colon T_1, \ldots, x_n\colon T_n)\ \phi_r \cdot \phi_c \cdot \phi_f \cdot \phi_m \cdot \phi_s \in \mathcal{P}$,
(2) σ has the following properties:
   (a) For every $x_i\colon T_i$ for which T_i is an enumeration set it holds that $\sigma(x_i) \in T_i$.
   (b) For every new $y\colon \tau$ occurring in ϕ_f it holds that $\sigma(y)$ is a fresh constant of type τ.
   (c) No other variables are in the domain of σ.
(3) α is a variable-renaming of the variables of $fv(\phi_r \cdot \phi_c \cdot \phi_f \cdot \phi_m \cdot \phi_s) \setminus dom(\sigma)$ such that the variables in $ran(\alpha)$ do not occur in $\sigma(\phi_r \cdot \phi_c \cdot \phi_f \cdot \phi_m \cdot \phi_s)$, and that preserves the type of variables (i.e., it is "well-typed") in a typed model. □

According to this semantics, each transition can only perform a transaction *in its entirety* (by being augmented to the intruder constraint accordingly). Thus, there cannot occur any race-conditions between transactions that are working on the same sets: every transaction first imposes constraints on the sets and then performs updates without other transactions disturbing it. This is equivalent to process calculi where each transaction first locks the sets it wants to access, and unlocks the sets at the end of the transaction. Note that each transaction rule can be executed arbitrarily often and so we support an unbounded number of "sessions". For instance, the transaction $\mathsf{outOfBand}_1(U\colon \mathsf{hon}, S\colon \mathsf{ser})$ of the keyserver example models that each honest agent $a \in \mathsf{hon}$ can register one fresh key at server $s \in \mathsf{ser}$ during each application of the transaction by inserting the fresh key into its keyring $\mathsf{ring}(a)$ while the server inserts it into the set of valid keys $\mathsf{valid}(a, s)$. The transaction can be executed any number of times with any agent $\sigma(U) \in \mathsf{hon}$, server $\sigma(S) \in \mathsf{ser}$, and a fresh value $\sigma(PK)$ for the key PK each time.

We say that a constraint $\mathcal{A}$ is *reachable* in protocol $\mathcal{P}$ iff $0 \Rightarrow_{\mathcal{P}}^{*} \mathcal{A}$ where $\Rightarrow_{\mathcal{P}}^{*}$ denotes the transitive reflexive closure of $\Rightarrow_{\mathcal{P}}$.

We need to ensure that these constraints are well-formed and we will therefore always assume the following sufficient requirements on the protocols $\mathcal{P}$ that we work with:

Definition 2.8 (Transaction well-formedness). For any transaction $\mathsf{t}(x_1\colon T_1, \ldots, x_n\colon T_n)\ \phi_r \cdot \phi_c \cdot \phi_f \cdot \phi_m \cdot \phi_s$ of $\mathcal{P}$ the constraint $dual(\phi_r \cdot \phi_c \cdot \phi_m \cdot \phi_s)$ must be well-formed w.r.t. the variables $\{z_1, \ldots, z_k, y_1, \ldots, y_m\}$ where $z_1, \ldots, z_k$ are exactly those variables among $x_1, \ldots, x_n$ declared as ranging over enumerations, and $y_1, \ldots, y_m$ are those variables that occur in ϕ_f.

Additionally, we require that the variables $x_1, \ldots, x_n$ in the transaction prefix are all distinct, and that the variables of ϕ_f are all distinct and that they do not occur in ϕ_r, ϕ_c, and $\{x_1, \ldots, x_n\}$ (i.e., they first occur in ϕ_f). Note also that well-formedness implies that the free variables and the bound variables of the transaction are disjoint. □

The first requirement means that each free variable must either first occur in a receive step or a positive check, be declared as ranging over an enumeration, or occur in a new action. Note that this still allows for non-determinism: for instance the check x in s allows for choosing an arbitrary element from s (that is compatible with the rest of the transaction); the requirement only forbids that variables can stand for completely arbitrary terms. The second requirement is just for convenience of further definitions and comes at no loss of generality.

Example 2.9. Continuing Example 2.6 we have for $\mathsf{keyUpdateServer}_1$ that $dual(\phi_r \cdot \phi_c \cdot \phi_m \cdot \phi_s) = \mathsf{send}\,\mathsf{sign}(\mathsf{inv}(PK), \mathsf{pair}(U, NPK)) \cdot \phi_c \cdot \phi_m \cdot \mathsf{receive}\,\mathsf{inv}(PK)$. This is a well-formed constraint w.r.t. $\{U, S\}$. The other well-formedness requirements are also satisfied for $\mathsf{keyUpdateServer}_1$. As another example, the constraint $\mathsf{receive}\ PK, \mathsf{inv}(PK)$ is well-formed w.r.t. $\{PK\}$, and so the transaction $\mathsf{intruderKeygen}_1$ of Fig. 1 is well-formed. In fact, all transactions of Fig. 1 are well-formed. □

To model goal violations of a protocol $\mathcal{P}$ we first fix a special private constant unique to $\mathcal{P}$, e.g., $\mathsf{attack}_{\mathcal{P}}$. These attack constants are supposed to be used only to signal that a goal violation has occurred, and so we require that they only occur in actions of transactions of the form $\mathsf{send}\,\mathsf{attack}_{\mathcal{P}}$. The protocol then has a (well-typed) attack if there exists a reachable constraint $\mathcal{A}$ such that $\mathcal{A} \cdot \mathsf{send}\,\mathsf{attack}_{\mathcal{P}}$ is (well-typed) satisfiable. A protocol with no attacks is *secure*.

With sets we can also model events, e.g., asserting an event e amounts to inserting e into a distinguished set of events while checking whether e has previously occurred (or not) corresponds to a positive (respectively negative) set-membership check. We therefore support all security properties expressible in the geometric fragment [2]. This covers many standard reachability goals such as authentication. It seems that any significantly richer fragment of first-order logic would be incompatible with our result. We do not currently support privacy-type properties, i.e., where goal violations occur if the observable behavior of protocols can be distinguished.

3 A TYPING RESULT

Type-flaw attacks are a nuisance in many formal arguments and there are relatively cheap ways to get rid of them once and for all. For instance, a classical example is in the Otway-Rees protocol [40]: here A sends first a message of the form

$$M, A, B, \mathsf{scrypt}(\mathsf{sk}(A, s), NA, M, A, B)$$

to the server s containing a fresh session identifier M, a fresh nonce NA, the name B, and a shared key $\mathsf{sk}(A, s)$ of A and s. Note that the names and session identifier are also transmitted in clear text. The only other action that A is involved in is receiving a message from the server of the form

$$M, A, B, \mathsf{scrypt}(\mathsf{sk}(A, s), NA, KAB)$$

containing a new shared key KAB for use with B. Due to the similarity of the messages, if pure string concatenation is used and if the key KAB can have the same length as M, A, B, there is a simple reflection attack where the intruder just sends back the message that A sent in the first step, so that A would accept as KAB the concatenation M, A, B which the intruder knows.

An easy way to prevent this is to use simple tags in the encrypted messages, e.g.,

$$\mathsf{scrypt}(\mathsf{sk}(A, s), 1, NA, M, A, B) \text{ and } \mathsf{scrypt}(\mathsf{sk}(A, s), 2, NA, KAB)\ .$$

This is for instance suggested by [5]. This, however, does not consider many other problems of plaintext structuring, e.g., pure string concatenation is associative and so it is not in general clear how to parse a string unless all terms have fixed size (which we do not have in practice). Moreover, this form of tagging is imposing a particular way of preventing type-flaw attacks, and there may be other ways to do it such as ASN.1 or XML. For this reason, we like to use a more general concept of *formats*, i.e., free function symbols that represent arbitrary real-world structuring mechanisms. Under the assumption that these are unambiguous (any string can be parsed in only one way for a given format) and pairwise disjoint (any string can be parsed for at most one format), there is a soundness result for this modeling [37]. The function symbols are transparent, i.e., they are both public and the intruder can destruct them like a pair. For instance, for Otway-Rees we may define formats $f_1(NA, M, A, B)$ and $f_2(NA, KAB)$. In this generality, many existing protocols (like TLS) do not require any changes in order to satisfy the type-flaw resistance requirement we define below. After this definition we can show a typing result: a type-flaw resistant protocol that has an attack has a so-called *well-typed* attack, i.e., where the intruder did not send any ill-typed messages. Thus, it is sound to verify the protocols under the restriction to a typed model: if they do not have a well-typed attack, then they have no attack at all. In this way we simply get rid of a lot of "garbage".

Type Expressions. Type expressions are terms built over the function symbols of $\Sigma \setminus C$ and a finite set $\mathfrak{T}_a$ of *atomic* types like enum and pkey. Further, we define a typing function Γ that assigns to every variable a type, to every constant an atomic type, and that is extended to composed terms as follows: $\Gamma(f(t_1, \ldots, t_n)) = f(\Gamma(t_1), \ldots, \Gamma(t_n))$ for every $f \in \Sigma^n \setminus C$ and terms t_i. We also require that $\{c \in C_{pub} \mid \Gamma(c) = \beta\}$ is infinite for each $\beta \in \mathfrak{T}_a$, thus giving the intruder access to an infinite supply of terms of each atomic type.[1]

We furthermore fix an atomic type attacktype $\in \mathfrak{T}_a$ and assign this type to all attack constants: $\Gamma(\mathsf{attack}_i) = \mathsf{attacktype}$. For the transactions $T = \mathsf{t}(x_1 : T_1, \ldots, x_n : T_n)\ \phi_r \cdot \phi_c \cdot \phi_f \cdot \phi_m \cdot \phi_s$ that we consider in this paper we will assume that they satisfy the following conditions:

(1) For each new $y : \tau$ action occurring in T the type τ is an atomic type.
(2) For each declared variable $x_i : T_i$ the following two conditions hold:
   (a) If T_i is a type then $\Gamma(x_i) = T_i$.
   (b) If T_i is an enumeration set then $\Gamma(T_i) = \{\Gamma(x_i)\}$ and $\Gamma(x_i)$ must be an atomic type.
(3) No variable of T contains in its type the atomic type attacktype.

The sufficient condition for a protocol to satisfy the typing result is now based on the following notions. A substitution δ is *well-typed* iff $\Gamma(x) = \Gamma(\delta(x))$ for all $x \in \mathcal{V}$. Given a set of messages that occur in a protocol we define the following set of sub-message patterns, intuitively the ones that may occur during constraint reduction:

Definition 3.1 (Sub-message patterns). The *sub-message patterns* $SMP(M)$ for a set of messages M is defined as the least set satisfying the following rules:

(1) $M \subseteq SMP(M)$.
(2) If $t \in SMP(M)$ and $t' \sqsubseteq t$ then $t' \in SMP(M)$.
(3) If $t \in SMP(M)$ and δ is a well-typed substitution then $\delta(t) \in SMP(M)$.
(4) If $t \in SMP(M)$ and $\mathsf{Ana}(t) = (K, R)$ then $K \subseteq SMP(M)$. □

The sufficient condition for the typing result is now that non-variable sub-message patterns have no unifier unless they have the same type:

[1] Note that [27] in contrast considers only public function symbols; one can simulate, however, a private function symbol of arity n by a public function symbol of arity $n + 1$ where the additional argument is used with a special constant that is never given to the intruder; in this way all results can be lifted to a model with both private and public function symbols. For instance, we can encode $\mathsf{inv} \in \Sigma^1$ in terms of a public symbol $\mathsf{inv}' \in \Sigma^2$ and a special secret constant $\mathsf{sec}_{\mathsf{inv}}$.

Definition 3.2 (Type-flaw resistance). We call a term t *generic for a set of variables* X, iff $t = f(x_1, \ldots, x_n)$ for some public f, $n > 0$, and $x_1, \ldots, x_n \in X$. We say that a set M of messages is *type-flaw resistant* iff $\forall t, t' \in SMP(M) \setminus \mathcal{V}.\ (\exists \delta.\ \delta(t) = \delta(t')) \longrightarrow \Gamma(t) = \Gamma(t')$. We may also apply the notion of type-flaw resistance to a constraint $\mathcal{A}$ to mean that:

(1) $trms(\mathcal{A}) \cup setops(\mathcal{A})$ is type-flaw resistant, and
(2) for all $t \doteq t'$ occurring in $\mathcal{A}$, if t and t' are unifiable then $\Gamma(t) = \Gamma(t')$, and
(3) for all $\forall \bar{x}.\ \phi_{\text{neg}}$ occurring in $\mathcal{A}$, and for all $t \not\doteq t'$ and t notin t' occurring in ϕ_{neg}, no subterm of (t, t') is generic for $\bar{x}$.[2]

Type-flaw resistance is extended to transactions as expected. A protocol $\mathcal{P}$ is type-flaw resistant iff $trms(\mathcal{P}) \cup setops(\mathcal{P})$ is type-flaw resistant and all transactions of $\mathcal{P}$ are type-flaw resistant. □

The main type-flaw resistance condition states that matching pairs of messages that might occur in a protocol run must have the same type. For equality checks $t \doteq t'$ the terms t and t' must have the same type if they are unifiable. For negative checks $\forall \bar{x}.\ \phi_{\text{neg}}$ we only need to require that there are no composed subterms of set-operations and inequalities in ϕ_{neg} whose immediate parameters are all bound variables (i.e., generic for $\bar{x}$). Together with the requirements on the typed model, this ensures that there always exist well-typed models of satisfiable type-flaw resistant negative checks. Note that the remaining actions on sets—insert, delete, and in—are also covered by condition 3.2(1): by requiring that the set $trms(\mathcal{A}) \cup setops(\mathcal{A})$ is type-flaw resistant we have that unifiable set operations in $\mathcal{A}$ must have the same type.

Example 3.3 (Type-flaw resistance of the keyserver protocol). As an example we show that the keyserver protocol is type-flaw resistant. That is, we show that the set of terms and set-operations of the keyserver protocol is type-flaw resistant (condition 3.2(1)), and that the actions of its transactions are type-flaw resistant. Condition 3.2(2) is vacuously satisfied since there are no equality checks in the protocol. For the last condition notice that each negative check in the protocol is either of the form x notin $f(c)$ or of the form $\forall y.\ x$ notin $f(y)$, where f is one of the set-family symbols. The only subterm of $(x, f(y))$ or $(x, f(c))$ that could potentially be generic is $f(y)$. Recall, however, from Example 2.2 that all set-family symbols are defined as private function symbols; thus the term $f(y)$ cannot be generic for y. Hence condition 3.2(3) is satisfied for the keyserver protocol.

To prove the main condition 3.2(1) one approach is to compute a finite set that subsumes the sub-message patterns of the protocol as well-typed instances. In fact, we will later show how to automatically compute such a set and prove that checking type-flaw resistance of this set is indeed sufficient. For the keyserver protocol notice that all terms and set-operations occurring in the protocol are well-typed instances of terms from the following set:

$$M = \{\mathsf{attack}, U, S, PK, NPK, \mathsf{ring}(U), \mathsf{valid}(U,S), \mathsf{revoked}(U,S), (PK, \mathsf{ring}(U)), (PK, \mathsf{valid}(U,S)), (PK, \mathsf{revoked}(U,S)), \mathsf{sign}(\mathsf{inv}(PK), \mathsf{pair}(U, NPK)), \mathsf{inv}(PK), \mathsf{pair}(U, NPK)\}$$

where $\Gamma(PK) = \mathsf{pkey}$, $\Gamma(NPK) = \mathsf{pkey}$, $\Gamma(U) = \mathsf{enum}$, and $\Gamma(S) = \mathsf{enum}$.

Since M is closed under subterms, and all variables in M are atomic, and since no term in M requires keys to analyze, then M subsumes all the sub-message patterns of the protocol as well-typed instances of terms in M. What remains to be shown is that each pair of matching composed terms in M (after sufficient variable-renaming) have the same type, and this is trivial to verify. Thus, the keyserver protocol is type-flaw resistant. □

[2]In fact, we have also proved in Isabelle that our typing result also works when allowing additionally the following form of negative checks: ϕ_{neg} is of the form $t_1 \not\doteq t_1' \vee \cdots \vee t_n \not\doteq t_n'$ and $\Gamma(fv(\phi_{\text{neg}}) \setminus \bar{x}) \subseteq \mathfrak{T}_a$. While this is in specifications of relatively little value, it may be helpful in other contexts.

With this definition we can now state and prove the actual typing result: that for type-flaw resistant protocols it is safe to only verify that no well-typed attack exists. A recurrent idea (that we also employ when proving the compositionality result) is to first prove the result on the constraint-level and then lift it to the protocol-level. One of the benefits of our constraint-based protocol model is actually that performing this lifting is relatively straightforward. The constraint-level typing result then says that satisfiable type-flaw resistant constraints always have well-typed models:

Theorem 3.4 (The typing result on the constraint level). *If $\mathcal{A}$ is a well-formed, type-flaw resistant constraint, and if $\mathcal{I} \models \mathcal{A}$, then there exists a well-typed interpretation $\mathcal{I}_\tau$ such that $\mathcal{I}_\tau \models \mathcal{A}$.*

This constraint-level theorem is then lifted to the protocol-level. First, a type-flaw resistance preservation lemma needs to be proven:

Lemma 3.5 (Well-formedness and type-flaw resistance preservation). *If $\mathcal{P}$ is a well-formed and type-flaw resistant protocol, and if $\mathcal{A}$ is a reachable constraint of $\mathcal{P}$, then both $\mathcal{A}$ and $\mathcal{A} \cdot \mathsf{send}\,\mathsf{attack}_{\mathcal{P}}$ are well-formed and type-flaw resistant.*

Finally, the protocol-level theorem follows from Lemma 3.5 and Theorem 3.4:

Theorem 3.6 (The typing result, for protocols). *Let $\mathcal{P}$ be a well-formed and type-flaw resistant protocol. If $\mathcal{P}$ is well-typed secure then $\mathcal{P}$ is also secure in the untyped model.*

Discussion. In contrast to with most other type-flaw prevention results like [5], we do not prescribe a particular tagging scheme but merely require that (sub-)messages of different types are always discernible. This makes the result directly applicable to many protocols without changes and the use of formats also takes care of most problems of parsing. Our result requires formats also on the top-level, i.e., unencrypted plaintext messages. Obviously tagging at that level does not give any additional security (nor does it hurt) since the intruder can easily manipulate plaintext messages, e.g., changing tags. We do this for uniformity: we do not build an exception into our definition, and in fact it is good practice to be clear also on the cleartext level how the message is meant. Note that [13] uses an idea similar to our typing system, except that they do not require formats on the plaintext level. Moreover, [13] can handle equivalence properties. On the other hand, our result supports protocols that use databases.

4 COMPOSITION: EXAMPLE AND ILLUSTRATION

In this section, we extend and adapt our keyserver example (Sec. 2.1) to illustrate the composition of security protocols: we define two keyserver protocols that share the same database of valid public keys registered at the keyserver. In a nutshell, the first protocol $\mathcal{P}_{ks,1}$ allows users to register public keys out of band and to update an existing key with a new one (revoking the old key in the process), while the second protocol $\mathcal{P}_{ks,2}$ uses a different mechanism to register new public keys. Thus, both protocols use the shared database valid.

There are three atomic types used in the example: the type of agents enum (which is also the type of the enumeration constants), public keys pkey, and the type attacktype of the attack_i constants. We partition the constants of type enum into the honest users hon, the dishonest users dis, and the keyservers ser. There are sets for authentication goals named begin_1, end_1, begin_2, and end_2, and all protocol actions related to these sets are highlighted in gray; let us first ignore these.

Protocol $\mathcal{P}_{ks,1}$. The protocol $\mathcal{P}_{ks,1}$ is an extension of the keyserver example from Sec. 2.1 and includes the transactions from Fig. 1. Observe that some actions in the transactions of Fig. 1 are labeled with a $\star$. In a nutshell those actions are relevant also to the other protocol in composition: this is because the set valid, e.g., will be shared with the other protocol, so an insertion is relevant;

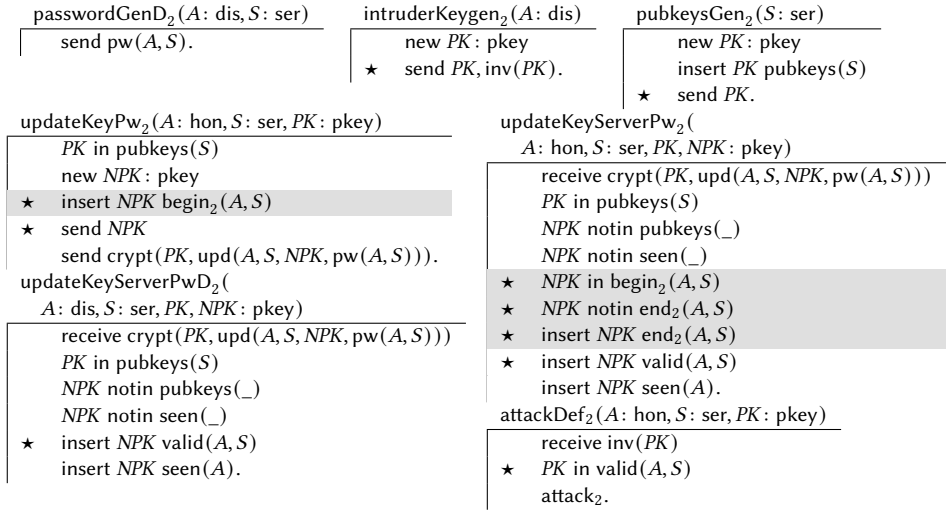

Fig. 2. The second keyserver protocol $\mathcal{P}_{ks,2}$. In addition to the transactions shown in this figure the protocol also contains transactions for authentication goals.

similarly, in outOfBand$_1$, all newly created keys like PK are by default secret, and by the ★ label we declare that it is immediately declassified (as it is expected for a public key), while the corresponding private key inv(PK) remains a secret. This will be explained in more detail below (as well as the gray-shaded actions related to the authentication goals). Note also that the intruder learns in outOfBandD$_1$ both PK and inv(PK) and they are both declassified, i.e., it does not count as an attack that the intruder knows key pairs that dishonest users generated. Also, we reveal in keyUpdateUser$_1$ the private key inv(PK), in order to specify that the protocol must ensure forward secrecy: even when old private keys are leaked, none of the security goals are violated. In fact, that this action is labeled ★ is a textbook example for declassification of a secret: after this transaction, the intruder does know inv(PK) and this does not count as an attack.

Protocol $\mathcal{P}_{ks,2}$. The second protocol—Fig. 2—has another mechanism to register new keys: every user has a password pw(A, S) with the server. The intruder gets the password of every dishonest agent A with rule passwordGenD$_2$.

Instead of using a (possibly weak) password for an encryption, the registration message is encrypted with the public key of the server. For honest agents this is defined in rule updateKeyPw$_2$.

For uniformity, we model the server's public keys in a set pubkeys(S) that is initialized with rule pubkeysGen$_2$ (in fact, S may thus have multiple public keys), and as in the first protocol we also have a rule in $\mathcal{P}_{ks,2}$ that gives the intruder arbitrarily many fresh key pairs. Note that thus pubkeys(S) is a set that is shared in the sense that all honest agents can read it, and S can insert new keys into it. This is an easy way to abstract from how the public key of the server itself is securely distributed. However, this set is local to $\mathcal{P}_{ks,2}$: the other keyserver protocol does not use it.

Similar to $\mathcal{P}_{ks,1}$ we give the dishonest agents the ability to generate their own fresh key pairs with rule intruderKeygen$_2$. This is necessary since we want to be able to run $\mathcal{P}_{ks,2}$ also in isolation.

Rule updateKeyServerPw$_2$ models how the server receives a registration request (in case of honest users—updateKeyServerPwD$_2$ is the pendant for the dishonest users). To protect against replay, the server uses a set seen of seen keys (this may in a real implementation be a buffer-timestamp mechanism). The difference to the honest-agent version is in the goal sets.

Secrecy. If a valid private key of an honest agent becomes known to the intruder then there is an attack. Such a secrecy goal is defined with rule $\mathsf{attackDef}_i$, where $i = 1$ in $\mathcal{P}_{ks,1}$ and $i = 2$ in $\mathcal{P}_{ks,2}$.

Authentication. Besides the secrecy goal $\mathsf{attackDef}_i$ that no valid private key of an honest agent may ever be known by the intruder, the crucial authentication goal is that all insertions into $\mathsf{valid}(A, S)$ for honest A are authenticated. This authentication goal is in particular relevant for the composition, because the valid set family is shared, and thus neither protocol can work properly if the other would insert unauthenticated keys into valid. Actually, compositionality requires even more about the shared sets, but at least some form of authentication is obviously needed.

We shall now look at the gray-shaded lines in all the previous rules that we skipped in previous explanations. First, in every rule where an honest agent A intends to introduce a key PK, they will insert this key into a special set $\mathsf{begin}_i(A, S)$ (for i being the protocol identifier). This is a set that would not exist in a real implementation and that we simply use here to formulate the security goals: in other words the $\mathsf{begin}_i(A, S)$ set contains the keys that A would like to use with S as public keys of A. We will never delete from this set, so it contains also past keys that A has actually already deleted. Note that for dishonest A we do not have any such sets, since the goals are concerned with guarantees for the honest agents only.

Symmetrically, the set $\mathsf{end}_i(A, S)$ represents the keys that server S has accepted as (apparently) coming from an honest agent A. In the out-of-band rule $\mathsf{outOfBand}_1$, we have that the key is directly inserted into both $\mathsf{begin}_i(A, S)$ and $\mathsf{end}_i(A, S)$, because this rule formalizes that A and S by some other means (e.g. physical visit) directly agree on a public key for A. In all other rules, the insertion into $\mathsf{end}_i(A, S)$ happens when the server receives a message that seems to indicate that A wants to introduce some key, and all conditions for accepting it are met. Observe, however, that these requirements include also some conditions on the authentication-sets themselves: for instance, the rule $\mathsf{keyUpdateServer}_1$ requires that the received NPK that A apparently wants to use as her new public key shall both be in $\mathsf{begin}_i(A, S)$ and not be in $\mathsf{end}_i(A, S)$. Suppose the situation that all conditions of rule $\mathsf{keyUpdateServer}_1$ were met, except that $NPK \notin \mathsf{begin}_i(A, S)$. That would mean that the server were to accept at this moment NPK (since, as far as the server can see, the key is legitimate) while A has never meant to use this key in this way. We would regard this as an authentication attack, more precisely as a violation of the goal of non-injective agreement in Lowe's hierarchy [35]. We formalize this by the following attack rules (rule $\mathsf{authAttack}_1$ and rule $\mathsf{authAttack}_2$ for the analogous situation of protocol 2):

$\mathsf{authAttack}_1(A\colon \mathsf{hon}, S\colon \mathsf{ser}, PK, NPK\colon \mathsf{pkey})$	
	$\mathsf{receive}\ \mathsf{sign}(\mathsf{inv}(PK), \mathsf{pair}(A, NPK))$
★	PK in $\mathsf{valid}(A, S)$.
★	NPK notin $\mathsf{valid}(\_)$
	NPK notin $\mathsf{revoked}(\_)$
★	NPK notin $\mathsf{begin}_1(A, S)$
	attack_1.

$\mathsf{authAttack}_2(A\colon \mathsf{hon}, S\colon \mathsf{ser}, PK, NPK\colon \mathsf{pkey})$	
	$\mathsf{receive}\ \mathsf{crypt}(PK, \mathsf{upd}(A, S, NPK, \mathsf{pw}(A, S)))$
	PK in $\mathsf{pubkeys}(S)$
	NPK notin $\mathsf{pubkeys}(\_)$
	NPK notin $\mathsf{seen}(\_)$
★	NPK notin $\mathsf{begin}_2(A, S)$
	attack_2.

Going back to $\mathsf{keyUpdateServer}_1$, there is also the condition $NPK \notin \mathsf{end}_i(A, S)$. Suppose now that all other conditions of the rule were met but $NPK \in \mathsf{end}_i(A, S)$. This would mean that, as far as the server S can tell, the key is legitimate and S would accept it, and indeed A at some point in the past has meant to use this key (and thus $NPK \in \mathsf{begin}_i(A, S)$). However, S has already accepted it before, because it is already in $\mathsf{end}_i(A, S)$. This would be a violation of injective agreement: the server has been made to accept a key more often than it was meant to be used. We can only do this, because the keys of an update request message are supposed to be fresh, i.e., an honest agent will never request an update to an old key; thus, when a server is made to accept the same key twice

from an honest agent it must be a replay attack. This is captured by the following two rules:

$\mathsf{authInjAttack}_1(A\colon \mathsf{hon}, S\colon \mathsf{ser}, PK, NPK\colon \mathsf{pkey})$	
	$\mathsf{receive}\ \mathsf{sign}(\mathsf{inv}(PK), \mathsf{pair}(A, NPK))$
$\star$	PK in $\mathsf{valid}(A, S)$
$\star$	NPK notin $\mathsf{valid}(\_)$
	NPK notin $\mathsf{revoked}(\_)$
$\star$	NPK in $\mathsf{begin}_1(A, S)$
$\star$	NPK in $\mathsf{end}_1(A, S)$
	attack_1.

$\mathsf{authInjAttack}_2(A\colon \mathsf{hon}, S\colon \mathsf{ser}, PK, NPK\colon \mathsf{pkey})$	
	$\mathsf{receive}\ \mathsf{crypt}(PK, \mathsf{upd}(A, S, NPK, \mathsf{pw}(A, S)))$
	PK in $\mathsf{pubkeys}(S)$
	NPK notin $\mathsf{pubkeys}(\_)$
	NPK notin $\mathsf{seen}(\_)$
$\star$	NPK in $\mathsf{begin}_2(A, S)$
$\star$	NPK in $\mathsf{end}_2(A, S)$
	attack_2.

Observe that for protocol 1, the rules $\mathsf{keyUpdateServer}_1$, $\mathsf{authAttack}_1$, and $\mathsf{authInjAttack}_1$ together handle all cases where the server would accept the incoming public key NPK: either everything is fine as far as the authentication goals are concerned ($\mathsf{keyUpdateServer}_1$) or there is a non-injective agreement violation ($\mathsf{authAttack}_1$), or an injective agreement violation ($\mathsf{authInjAttack}_1$) (and similarly for the second protocol). Notice that our rules thus do not allow for an unauthenticated key to be inserted into the shared set valid: if the protocol due to a security flaw could insert an unauthenticated key, then only the attack rule can fire. This is crucial for composition as this allows us a kind of "contract" between the protocols that they comply with certain rules, e.g. inserting only authenticated keys into the valid set and the verification of this compliance is then a problem we can check for each protocol in isolation.

5 THE COMPOSITIONALITY RESULT

The core definition of this section is rather simple: We define the *parallel composition* $\mathcal{P}_1 \parallel \mathcal{P}_2$ of two protocols $\mathcal{P}_1$ and $\mathcal{P}_2$ as their union: $\mathcal{P}_1 \parallel \mathcal{P}_2 \equiv \mathcal{P}_1 \cup \mathcal{P}_2$. Protocols $\mathcal{P}_1$ and $\mathcal{P}_2$ are also referred to as the *component protocols* of the composition $\mathcal{P}_1 \parallel \mathcal{P}_2$. To express composition of more than two protocols we parameterize our theory over a set $\mathcal{L}$ of indices. The only requirement on the set $\mathcal{L}$ is that it has at least two elements and we usually use natural numbers to denote the elements of $\mathcal{L}$. The parallel composition of all protocols indexed by $\mathcal{L}$ is then denoted by $\|_{i\in\mathcal{L}}\,\mathcal{P}_i$ and is again defined as the union of the component protocols: $\|_{i\in\mathcal{L}}\,\mathcal{P}_i \equiv \bigcup_{i\in\mathcal{L}}\mathcal{P}_i$. If $\mathcal{L} = \{1, \ldots, N\}$ for some N (i.e., if $\mathcal{L}$ is finite) then we may also use the notation $\mathcal{P}_1 \parallel \cdots \parallel \mathcal{P}_N$.

For composed protocols $\|_{i\in\mathcal{L}}\,\mathcal{P}_i$ the reachable constraints will in general contain actions originating from multiple component protocols. To keep track of where an action in a constraint originated we assign to each action a *label* ℓ, and we use the notation $\ell\colon a$ to denote that action a has label ℓ. The actions that are exclusive to the i-th component are marked with i. For that reason we also refer to the indices $\mathcal{L}$ as the *protocol-specific labels*. In addition to the protocol-specific labels $\mathcal{L}$ we also have the special label $\star$. Labels ℓ then range over $\mathcal{L} \cup \{\star\}$ unless otherwise specified. The formal transaction language defined in Sec. 2.3 is also extended to allow labeling of actions.

Let $\mathcal{A}$ be a constraint with labels and ℓ be a label. We define $\mathcal{A}|_\ell$ to be the projection of $\mathcal{A}$ to the steps labeled ℓ or $\star$ (so the $\star$-steps are kept in every projection). We extend projections to transaction rules and protocols as expected. We may also write $\mathcal{P}^\star$ instead of $\mathcal{P}|_\star$.

Example 5.1. In the composed keyserver example we have two protocol specific labels: $\mathcal{L} = \{1, 2\}$. Note that we omitted the protocol-specific labels when defining the transactions. This is because each protocol has a unique protocol-specific label and so it is unambiguous which label to assign to the actions that do not carry the $\star$-label. Note also that all new actions are implicitly labeled with a $\star$: it is necessary for our compositionality result that they occur in all projections, and so we usually omit explicitly labeling them. For instance, in the formal transaction syntax the $\mathsf{pubkeysGen}_2$ transaction is written as follows, and its projection $\mathsf{pubkeysGen}'_2$ to label 1 preserves

only the first and the last of its actions:

$$\mathsf{pubkeysGen}_2(S\colon \mathsf{ser})\ \star\colon \mathsf{new}\ PK\colon \mathsf{pkey} \cdot 2\colon \mathsf{insert}\ PK\ \mathsf{pubkeys}(S) \cdot \star\colon \mathsf{send}\ PK$$
$$\mathsf{pubkeysGen}'_2(S\colon \mathsf{ser})\ \star\colon \mathsf{new}\ PK\colon \mathsf{pkey} \cdot \star\colon \mathsf{send}\ PK \qquad \square$$

An important property of projections is that projected reachable constraints are reachable in projected protocols:

Lemma 5.2. *If $\mathcal{A}$ is a constraint reachable in the protocol $\mathcal{P}$ and ℓ is a label, then the projected constraint $\mathcal{A}|_\ell$ is reachable in the projected protocol $\mathcal{P}|_\ell$.*

With stateful protocols and parallel composition defined we can now formally define the concepts underlying our results and state our compositionality theorems. We first provide a result on the level of constraints and afterwards show our main theorems for stateful protocols.

5.1 Protocol Abstraction

All actions containing the valid set family in our keyserver example have been labeled with $\star$. Labeling operations on the shared sets with $\star$ is an important part of our compositionality result.

Essentially, compositionality results aim to prevent that attacks can arise from the composition itself, i.e., attacks that do not similarly work on the components in isolation. Thus, we want to show that attacks on the composed system can be sufficiently decomposed into attacks on the components. This, however, cannot directly work if the components have shared sets like valid in the example: if one protocol inserts something into a set and the other protocol reads from the set, then this trace in general does not have a counter-part in the second protocol alone. We thus need a kind of *interface* to how the two protocols can influence their shared sets. In the keyserver example, both protocols can insert public keys into the shared set valid; the first protocol can even remove them. The idea is now that we develop from each protocol an *abstract* version that subsumes all the modifications that the concrete protocol can perform on the shared sets. This can be regarded as a "contract" for the composition: each protocol *guarantees* that it will not make any modifications that are not covered by its abstract protocol, and it will *assume* that the other protocol only makes modifications covered by the other protocol's abstraction. We will still have to verify that each individual protocol is also secure when running together with the other abstract protocol, but this is in general much simpler than the composition of the two concrete protocols.

In general, the abstraction of a component protocol $\mathcal{P}$ is defined by restriction to those steps that are labeled $\star$, i.e., $\mathcal{P}^\star$. We require that at least the modifications of shared sets are labeled $\star$. In the keyserver example we have also labeled the operations on the authentication-related sets with a $\star$ (everything highlighted in gray): we need to ensure that we insert into the set of valid keys of an honest agent only those keys that really have been created by that agent and that have not been previously inserted. So the contract between the two protocols is that they only insert keys that are properly authenticated, but the abstraction ignores how each protocol achieves the authentication (e.g., signatures vs. passwords and seen-set). Some outgoing messages are also labeled with $\star$ which we discuss later.

Example 5.3. Consider the abstractions of rules $\mathsf{updateKeyPw}_2$ and $\mathsf{updateKeyServerPw}_2$:

$\mathsf{updateKeyPw}_2(A\colon \mathsf{hon}, S\colon \mathsf{ser})$	
	$\mathsf{new}\ NPK\colon \mathsf{pkey}$
$\star$	$\mathsf{insert}\ NPK\ \mathsf{begin}_2(A,S)$
$\star$	$\mathsf{send}\ NPK.$

$\mathsf{updateKeyServerPw}_2(A\colon \mathsf{hon}, S\colon \mathsf{ser}, NPK\colon \mathsf{pkey})$	
$\star$	$NPK\ \mathsf{in}\ \mathsf{begin}_2(A,S)$
$\star$	$NPK\ \mathsf{notin}\ \mathsf{end}_2(A,S)$
$\star$	$\mathsf{insert}\ NPK\ \mathsf{valid}(A,S)$
$\star$	$\mathsf{insert}\ NPK\ \mathsf{end}_2(A,S).$

The gray actions prevent unauthenticated key registration because keys can only be registered if inserted into begin_2 by an honest agent. If we *did not* ensure such authenticated key-registration

then the intruder would be able to register arbitrary keys in $\mathcal{P}^{\star}_{ks,2}$. This would lead to an attack on secrecy in the protocol $\mathcal{P}_{ks,1} \parallel \mathcal{P}^{\star}_{ks,2}$.

One may wonder why there is no similar specification for secrecy, i.e., that $\mathsf{inv}(NPK)$ is secret for every key NPK that is being inserted into valid. In fact, below we will declare all private keys to be secret by default. Thus, unless explicitly declassified, they are (implicitly) required to be secret. □

5.2 Shared Terms

The main restriction that most compositionality results require is that the messages of the component protocols are sufficiently disjoint, so that the intruder cannot use messages (or sub-messages) of one component in another. (As discussed in Sec. 3, one can achieve this using formats.) One typically needs to make at least some exceptions from disjointness, e.g., agent names and fixed long-term keys. However, as we have seen in the previous examples, we want in this work to also (for instance) consider freshly generated keys that are part of shared sets. In fact, any kind of data shared via the sets cannot be unique to a component. Therefore, our approach uses a generalized way of handling such shared terms.

First, let us call *basic public terms* those that the intruder can derive without any prior knowledge: $\emptyset \vdash t$. This could for instance contain agent names. Second, the modeler shall choose a set of ground terms $\mathcal{S}$, called *the shared terms*, and $\mathcal{S}$ is a parameter of our compositionality result. In fact, our only requirement is that every ground protocol message is either unique to a single protocol, or it is a basic public term, or it is in $\mathcal{S}$. For our keyserver example, we have the freshly generated public and private keys as part of $\mathcal{S}$, while the more complex messages like signatures are unique to one component. Note that one may well have also more complex terms in $\mathcal{S}$, e.g., key certificates if they are used by several protocols. For reasons that we explain below in more detail, one may want to keep $\mathcal{S}$ as small as possible, because all messages in there require special care in the composition. In Sec. 6 we discuss how a suitable representation of $\mathcal{S}$ can be computed automatically.

More formally, we first define the *ground sub-message patterns (GSMP)* of a set of terms M as $GSMP(M) \equiv \{t \in SMP(M) \mid fv(t) = \emptyset\}$. This definition is extended to constraints $\mathcal{A}$ as the set $GSMP(\mathcal{A}) \equiv GSMP(trms(\mathcal{A}) \cup setops(\mathcal{A}))$ and similarly for protocols.

Example 5.4. We will typically study the ground subterms of each individual protocol in parallel with the abstraction of the other. For the example, the set $GSMP(\mathcal{P}_{ks,1} \parallel \mathcal{P}^{\star}_{ks,2})$ is the closure under subterms of the following set:

$$\{\mathsf{attack}_1, \mathsf{sign}(\mathsf{inv}(pk), \mathsf{pair}(a, npk)), (pk, \mathsf{ring}(a)), (pk, f(a, s)) \mid pk, npk, a, s \in C,\\ \Gamma(\{pk, npk\}) = \{\mathsf{pkey}\}, \Gamma(\{a, s\}) = \{\mathsf{enum}\}, f \in \{\mathsf{valid}, \mathsf{revoked}, \mathsf{begin}_i, \mathsf{end}_i\}, i \in \{1, 2\}\}$$

and $GSMP(\mathcal{P}^{\star}_{ks,1} \parallel \mathcal{P}_{ks,2})$ is the closure under subterms of the following set:

$$\{\mathsf{attack}_2, \mathsf{inv}(pk), \mathsf{crypt}(pk, \mathsf{upd}(a, s, npk, \mathsf{pw}(a, s))), (pk, \mathsf{pubkeys}(s)), (pk, f(a, s)) \mid pk, npk \in C,\\ a, s \in C, \Gamma(\{pk, npk\}) = \{\mathsf{pkey}\}, \Gamma(\{a, s\}) = \{\mathsf{enum}\}, f \in \{\mathsf{valid}, \mathsf{seen}, \mathsf{begin}_i, \mathsf{end}_i\}, i \in \{1, 2\}\}$$

□

Let $\mathcal{S}$ be a set of ground terms disjoint from the set of basic public terms. For composition we will require that component protocols are disjoint in their ground sub-message patterns except for basic public terms and shared terms:

Definition 5.5 (GSMP disjointness). Given two sets of terms M_1 and M_2, and a ground set of terms $\mathcal{S}$, we say that M_1 and M_2 are *$\mathcal{S}$-GSMP disjoint* iff $GSMP(M_1) \cap GSMP(M_2) \subseteq \mathcal{S} \cup \{t \mid \emptyset \vdash t\}$. Furthermore, given two constraints $\mathcal{A}_1$ and $\mathcal{A}_2$ we say that they are $\mathcal{S}$-GSMP disjoint iff the sets $trms(\mathcal{A}_1) \cup setops(\mathcal{A}_1)$ and $trms(\mathcal{A}_2) \cup setops(\mathcal{A}_2)$ are $\mathcal{S}$-GSMP disjoint. This is similarly extended to protocols as expected. □

5.3 Declassification and Leaking

As the public and private keys in the keyserver example illustrate, we want in general that some of the shared terms $\mathcal{S}$ are secret from the intruder, while others may be known to the intruder, and in fact it can be crucial for the interplay of the components which ones are which. Also one may want to allow in some cases that the intruder may learn old keys, e.g., the private keys to a revoked key in the keyserver example. To allow for such a dynamic secrecy status, we define the following policy about the secrecy of terms in $\mathcal{S}$: initially, all elements of $\mathcal{S}$ are considered secret, and a protocol can explicitly *declassify* elements of $\mathcal{S}$. It is considered a *leak* if the intruder can find out an element of $\mathcal{S}$ that has not been declassified, and we will require as part of the composition that no component protocol (in isolation) leaks. In the keyserver example, all public keys are immediately declassified in the transition where they are created. If the creator is dishonest, then also the corresponding private key is declassified. Finally, when keys are revoked, then also the private key to the revoked key is declassified.

The notation for such declassification of a message M is simply $\star$ send M: the message is sent out, so the intruder can see it, and the $\star$ label ensures that all declassifications that occur in a protocol (or in a constraint) are visible in its abstraction. This means that all declassifications are part of the interface that the protocol provides to other protocols, and the proof of security thus ensures that each component is secure in the presence of all declassifications that the other protocols may perform. In the keyserver example, $\mathcal{P}^{\star}_{ks,1}$ guarantees that it will only declassify private keys (via $\mathsf{intruderKeygen}_1$) where they are freshly created and not member of any set and (via $\mathsf{keyUpdateUser}_1$) in a transition where they have been in valid for an honest U before and are just removed. Together with the abstraction of the other transactions we get an interface that allows to verify in $\mathcal{P}^{\star}_{ks,1} \parallel \mathcal{P}_{ks,2}$ that no private key can ever be leaked while the corresponding public key is in the valid set of an honest agent.

Observe that this leakage policy can play a crucial role in the goals of a protocol composition, but we may have secrecy goals that are "local" to a protocol, i.e., that affect data that is not in $\mathcal{S}$. This would in fact be the case for the secrecy goal of classical parallel protocol composition where all messages are disjoint and no shared sets are used.

For a constraint $\mathcal{A}$ with model $\mathcal{I}$ we formally define the set of terms that has been declassified in $\mathcal{A}$ under $\mathcal{I}$:

Definition 5.6 (Declassification). Let $\mathcal{A}$ be a labeled constraint and $\mathcal{I}$ a model of $\mathcal{A}$. Then the set of *declassified secrets of $\mathcal{A}$ under $\mathcal{I}$* is defined as follows:

$$\mathit{declassified}(\mathcal{A},\mathcal{I}) \equiv \{t \mid \{s_1,\ldots,s_n \mid \star\colon \text{receive } s_1,\ldots,s_n \text{ occurs in } \mathcal{I}(\mathcal{A})\} \vdash t\} \qquad \square$$

Given a protocol $\mathcal{P}$, a reachable constraint $\mathcal{A}$ (i.e., $0 \Rightarrow^*_{\mathcal{P}} \mathcal{A}$), and a model $\mathcal{I}$ of $\mathcal{A}$, then $\mathcal{I}(\mathcal{A})$ represents a concrete protocol run and the set $\mathit{declassified}(\mathcal{A},\mathcal{I})$ represents those terms that are derivable from the messages that have been declassified by honest agents during the protocol run. By definition the basic public terms are therefore also declassified. Note also that in this definition we have reversed the direction of the declassification transmission, because the send and receive steps of reachable constraints are duals of the transaction rules they originated from.

Declassification also allows us to share terms that have secrets from $\mathcal{S}$ as subterms but which are not themselves meant to be secret. For instance, public key certificates have as subterm the private key of the signing authority, and such certificates can be shared between protocols by modeling them as shared secrets that are declassified when first published.

Finally, if the intruder learns a shared term that has not been declassified then it counts as an attack. We say that protocol $\mathcal{P}$ *leaks* a secret s if there is a reachable satisfiable constraint $\mathcal{A}$ where the intruder learns s before it is declassified:

Definition 5.7 (Leakage). Let $\mathcal{A}$ be a labeled constraint with model $\mathcal{I}$ and let $\mathcal{S}$ be a set of shared terms. $\mathcal{A}$ *leaks a secret from* $\mathcal{S}$ *under* $\mathcal{I}$ iff there exists $s \in \mathcal{S} \setminus \mathit{declassified}(\mathcal{A}, \mathcal{I})$ and a protocol-specific label $\ell \in \mathcal{L}$ such that $\mathcal{I} \models \mathcal{A}|_\ell \cdot \mathsf{send}\, s$. □

Our notion of leakage requires that one of the components in isolation leaks a secret. This may seem like an undue restriction (that it counts only as a leakage if one protocol alone can leak), but we will make this as one of the prerequisites of composition, i.e., a quite weak requirement that can be checked for each protocol in isolation. Then the compositionality result ensures that the composition does not leak the shared terms. Note also that the set $\mathit{declassified}(\mathcal{A}, \mathcal{I})$ is unchanged during projection of $\mathcal{A}$, and so it suffices to pick the leaked s from the set $\mathcal{S} \setminus \mathit{declassified}(\mathcal{A}, \mathcal{I})$ instead of $\mathcal{S} \setminus \mathit{declassified}(\mathcal{A}|_i, \mathcal{I})$.

Example 5.8. The terms occurring in the GSMP intersection of the two keyserver protocols are (a) public keys pk, (b) private keys of the form inv(pk), (c) agent names, and (d) operations on the shared set families valid, begin_i, and end_i. Agent names are basic public terms in our example, i.e., $\emptyset \vdash \mathsf{a}$ for all constants a of type agent. The public keys are initially secret, but we immediately declassify them whenever they are generated. To satisfy GSMP disjointness of $\mathcal{P}_{ks,1} \parallel \mathcal{P}^\star_{ks,2}$ and $\mathcal{P}^\star_{ks,1} \parallel \mathcal{P}_{ks,2}$ it thus suffices to choose the following set as the set of shared terms:

$$\mathcal{S} = \{pk, \mathsf{inv}(pk), (pk, f(a,s)), f(a,s) \mid \Gamma(\{a,s\}) = \{\mathsf{enum}\}, \Gamma(pk) = \mathsf{pkey}, f \in \{\mathsf{valid}, \mathsf{begin}_1, \mathsf{end}_1, \mathsf{begin}_2, \mathsf{end}_2\}, pk, a, s \in C\}$$

Note that we want the set symbols like valid to be private. This is because terms like (pk, valid(a, s)) occurs as a GSMP term in both component protocols, and so we have to prevent the intruder from constructing such terms even after declassification of keys pk. Hence, we model the set expressions like valid(a, s) as secrets to prevent the intruder from constructing (pk, valid(a, s)) when the intruder knows the constants pk, a, and s. □

5.4 Parallel Compositionality for Constraints

With these concepts defined we can list the requirements on constraints that are necessary to apply our result on the constraint level:

Definition 5.9 (Parallel composability). Let $\mathcal{A}$ be a constraint and let $\mathcal{S}$ be a ground set of terms disjoint from the set of basic public terms. Then $(\mathcal{A}, \mathcal{S})$ is *parallel composable* iff

(1) for all protocol-specific labels $\ell, \ell' \in \mathcal{L}$, if $\ell \neq \ell'$ then $\mathcal{A}|_\ell$ and $\mathcal{A}|_{\ell'}$ are $\mathcal{S}$-GSMP disjoint,
(2) for all $\ell\colon (t,s), \ell'\colon (t',s') \in \mathit{labeledsetops}(\mathcal{A})$, if (t,s) and (t',s') are unifiable then $\ell = \ell'$,
(3) $\mathcal{A}$ is type-flaw resistant and well-formed,

where $\mathit{labeledsetops}(\mathcal{A}) \equiv \{\ell\colon (t,s) \mid \ell\colon \mathsf{insert}\ t\ s \text{ or } \ell\colon \mathsf{delete}\ t\ s \text{ or } \ell\colon t\ \mathsf{in}\ s \text{ occurs in } \mathcal{A} \text{ or there exists } \phi_{\mathrm{neg}} \text{ and } \bar{x} \text{ such that } t\ \mathsf{notin}\ s \text{ occurs in } \phi_{\mathrm{neg}} \text{ and } \ell\colon \forall \bar{x}.\ \phi_{\mathrm{neg}} \text{ occurs in } \mathcal{A}\}$. □

The first requirement is at the core of our compositionality result and states that the protocols can only share basic public terms and shared terms. The second condition is our requirement on stateful protocols; it implies that shared sets must be labeled with a $\star$. Finally, the third and last condition is needed to apply the typing result and it is orthogonal to the other conditions; it is indeed only necessary so that we can apply Theorem 3.4 and restrict ourselves to well-typed attacks. Typing results with different requirements could potentially be used instead.

With the composability requirements defined we can state our main result on the constraint-level:

Theorem 5.10 (Parallel Compositionality on the Constraint-Level). *If $(\mathcal{A}, \mathcal{S})$ is parallel composable and $\mathcal{I} \models \mathcal{A}$ then there exists a well-typed interpretation $\mathcal{I}_\tau$ such that either $\mathcal{I}_\tau \models \mathcal{A}|_\ell$ for all protocol-specific labels $\ell \in \mathcal{L}$ or some prefix $\mathcal{A}'$ of $\mathcal{A}$, that ends in a* receive *action, leaks a secret from $\mathcal{S}$ under $\mathcal{I}_\tau$.*

That is, we can obtain a well-typed model of all projections $\mathcal{A}|_\ell$, $\ell \in \mathcal{L}$, for satisfiable parallel composable constraints $\mathcal{A}$—or one of the projections has leaked a secret. In other words, if we can verify that a parallel composable constraint $\mathcal{A}$ does not have any well-typed model of all protocol-specific projections, and no prefix of $\mathcal{A}$ leaks a secret under any well-typed model, then it is unsatisfiable, i.e., there is no "attack".

5.5 Proving Parallel Compositionality on the Constraint-Level (Theorem 5.10)

In the following, we present a high-level proof of Theorem 5.10 (for the detailed proof in Isabelle, see [28]). Readers not interested in the proof details might want to continue directly with Sec. 5.6.

As an intermediate result we first prove the theorem for "stateless" constraints, i.e., for constraints that do not contain any inserts, deletes, and positive and negative set-membership actions. Then the theorem is established by lifting the intermediate result to stateful constraints using a variant of a constraint reduction technique from [27].

5.5.1 Proving Parallel Compositionality on the Constraint-Level for "Stateless" Constraints. For Theorem 5.10 we need to show that for satisfiable parallel composable constraints $\mathcal{A}$ with shared terms $\mathcal{S}$ we can obtain a well-typed model of all projections $\mathcal{A}|_\ell$, $\ell \in \mathcal{L}$, or $\mathcal{A}$ has leaked a secret in one of the projections. In a nutshell we show that any term t_i occurring in a $\ell\colon \mathsf{send}\, t_1, \ldots, t_n$ action of $\mathcal{A}$ needs only to be constructed from terms of protocol ℓ, unless leakage has occurred previously. Given a constraint $\mathcal{A}$ and a set of shared terms $\mathcal{S}$ we now define a useful variant $\vdash^{\mathcal{A}}_{\mathrm{GSMP}}$ of the intruder deduction relation $\vdash$ as the restriction of $\vdash$ to the GSMP terms of $\mathcal{A}$ only. This relation has a useful property:

Lemma 5.11. *Let $t \in GSMP(\mathcal{A})$ and let $M \subseteq GSMP(\mathcal{A})$. Then $M \vdash t$ iff $M \vdash^{\mathcal{A}}_{\mathrm{GSMP}} t$.*

Note that for well-typed $\mathcal{I}$ the intruder knowledge $\mathcal{I}(ik(\mathcal{A}))$ is a subset of $GSMP(\mathcal{A})$, and for all send $t_1, \ldots, t_n$ actions occurring in $\mathcal{A}$ the ground messages $\mathcal{I}(t_i)$ are also in $GSMP(\mathcal{A})$. Lemma 5.11 is therefore useful because we can prove that, under well-typed models, all terms occurring in send actions can be derived purely through derivation of other GSMP terms, without ever deriving, as an intermediate step, a term outside of the GSMP set. In other words, for parallel composable constraints under well-typed models we can reduce the intruder derivation problem to $\vdash^{\mathcal{A}}_{\mathrm{GSMP}}$.

One of the most difficult parts of the parallel compositionality proof is to show that the messages that the intruder must produce in a constraint can be derived under all projections or are leaked, and this part we can—thanks to the previous lemma—state as a property of $\vdash^{\mathcal{A}}_{\mathrm{GSMP}}$:

Lemma 5.12. *Let $(\mathcal{A}, \mathcal{S})$ be parallel composable. Furthermore, let $\mathcal{I}$ be a well-typed model of $\mathcal{A}$. If $\mathcal{I}(ik(\mathcal{A})) \vdash^{\mathcal{A}}_{\mathrm{GSMP}} t$, then either*

(C_1) $t \notin \mathcal{S} \setminus declassified(\mathcal{A}, \mathcal{I})$, and

(C_2) for all $i \in \mathcal{L}$, if $t \in GSMP(\mathcal{A}|_i)$ then $\mathcal{I}(ik(\mathcal{A}|_i)) \vdash^{\mathcal{A}}_{\mathrm{GSMP}} t$,

or there exists $s \in \mathcal{S} \setminus declassified(\mathcal{A}, \mathcal{I})$ and $j \in \mathcal{L}$ such that $\mathcal{I}(ik(\mathcal{A}|_j)) \vdash^{\mathcal{A}}_{\mathrm{GSMP}} s$.

With Lemma 5.11 we can then prove the following easily-established but important consequence of Lemma 5.12:

Lemma 5.13. *Let $(\mathcal{A}, \mathcal{S})$ be parallel composable, $\mathcal{I}$ be a well-typed model of $\mathcal{A}$, $i \in \mathcal{L}$ be a label, and t a term such that $t \in GSMP(\mathcal{A}|_i)$. If $\mathcal{I}(ik(\mathcal{A})) \vdash t$ then either $\mathcal{I}(ik(\mathcal{A}|_i)) \vdash t$ or there exists $s \in \mathcal{S} \setminus declassified(\mathcal{A}, \mathcal{I})$ and $j \in \mathcal{L}$ such that $\mathcal{I}(ik(\mathcal{A}|_j)) \vdash s$.*

Now we can use Lemma 5.13 to show that the models $\mathcal{I}$ of parallel composable constraints $\mathcal{A}$ are also models of the projections $\mathcal{A}|_i$, or some secret is leaked:

Lemma 5.14. *Let $(\mathcal{A}, \mathcal{S})$ be parallel composable and let $\mathcal{I}$ be a well-typed model of the stateless constraint $\mathcal{A}$. Then either $\mathcal{I} \models \mathcal{A}|_\ell$, for all $\ell \in \mathcal{L}$, or some prefix $\mathcal{A}' \cdot i\colon \mathsf{receive}\ t_1, \ldots, t_n$ of $\mathcal{A}$ leaks a secret from $\mathcal{S}$ under $\mathcal{I}$.*

Finally, we can use Theorem 3.4 (the typing result) to relax the well-typedness assumption of Lemma 5.14 and prove the result on the level of "stateless" constraints:

Lemma 5.15. *Let $(\mathcal{A}, \mathcal{S})$ be parallel composable and let $\mathcal{I}$ be a model of the stateless constraint $\mathcal{A}$. Then there exists a well-typed interpretation $\mathcal{I}_\tau$ of $\mathcal{A}$ such that either $\mathcal{I}_\tau \models \mathcal{A}|_\ell$ for all $\ell \in \mathcal{L}$ or some prefix $\mathcal{A}' \cdot i\colon \mathsf{receive}\ t_1, \ldots, t_n$ of $\mathcal{A}$ leaks a secret from $\mathcal{S}$ under $\mathcal{I}_\tau$.*

5.5.2 Proving Parallel Compositionality on the Constraint-Level for Stateful Constraints. For stateful constraints the proof idea is to use a variant of a reduction technique introduced in [27] to reduce the compositionality problem for stateful constraints to the compositionality problem for "stateless" constraints, i.e. constraints without set-operations:

Definition 5.16 (Translation of symbolic constraints). Given a set $D = \{\ell_1\colon (t_1, s_1), \ldots, \ell_n\colon (t_n, s_n)\}$, where D is finite, each t_i and s_i are terms, and $\ell_i \in \mathcal{L} \cup \{\star\}$ are labels, we define the *projection of D to ℓ*, written $|D|_\ell$, as follows: $|D|_\ell = \{\ell'\colon d \in D \mid \ell = \ell'\}$. For a constraint $\mathcal{A}$ its translation into (finitely many) stateless constraints is denoted by $tr(\mathcal{A}) = tr_\emptyset(\mathcal{A})$ where:

$$
\begin{aligned}
&tr_D(0) = \{0\}\\
&tr_D(\ell\colon \mathsf{insert}\ t\ s \cdot \mathcal{A}) = tr_{D\cup\{\ell\colon (t,s)\}}(\mathcal{A})\\
&tr_D(\ell\colon \mathsf{delete}\ t\ s \cdot \mathcal{A}) = \{\\
&\quad \ell\colon (t,s) \doteq d_1 \cdot \ldots \cdot \ell\colon (t,s) \doteq d_i \cdot \ell\colon (t,s) \not\doteq d_{i+1} \cdot \ldots \cdot \ell\colon (t,s) \not\doteq d_n \cdot \mathcal{A}' \mid\\
&\quad |D|_\ell = \{\ell\colon d_1, \ldots, \ell\colon d_i, \ldots, \ell\colon d_n\}, 0 \le i \le n, \mathcal{A}' \in tr_{D\setminus\{\ell\colon d_1,\ldots,\ell\colon d_i\}}(\mathcal{A})\}\\
&tr_D(\ell\colon t\ \mathsf{in}\ s \cdot \mathcal{A}) = \{\ell\colon (t,s) \doteq d \cdot \mathcal{A}' \mid \ell\colon d \in |D|_\ell, \mathcal{A}' \in tr_D(\mathcal{A})\}\\
&tr_D(\ell\colon (\forall \bar{x}.\ \phi \vee t_1\ \mathsf{notin}\ s_1 \vee \cdots \vee t_n\ \mathsf{notin}\ s_n) \cdot \mathcal{A}) = \{\\
&\quad \ell\colon (\forall \bar{x}.\ \phi \vee \psi_1) \cdot \ldots \cdot \ell\colon (\forall \bar{x}.\ \phi \vee \psi_k) \cdot \mathcal{A}' \mid\\
&\quad \psi_1 \wedge \cdots \wedge \psi_k \text{ is the conjunctive normal form of } \textstyle\bigvee_{i\in\{1,\ldots,n\}} \bigwedge_{\ell\colon d\in|D|_\ell} (t_i, s_i) \not\doteq d,\\
&\quad \phi = t_1' \not\doteq s_1' \vee \cdots \vee t_m' \not\doteq s_m', \mathcal{A}' \in tr_D(\mathcal{A})\}\\
&tr_D(\ell\colon \mathfrak{a} \cdot \mathcal{A}) = \{\ell\colon \mathfrak{a} \cdot \mathcal{A}' \mid \mathcal{A}' \in tr_D(\mathcal{A})\} \text{ otherwise}
\end{aligned}
$$

□

$tr(\mathcal{A})$ reduces $\mathcal{A}$ into a finite set of constraints without set-operations, and these constraints have together exactly the same models as $\mathcal{A}$. The idea is, in a nutshell, that for a given $\mathcal{A}$, only finitely many set updates have occurred, and so one can convert each set-operation that occur in $\mathcal{A}$ into finitely many $\doteq$ and $\not\doteq$ actions that together preserve the meaning of the original set-operation.

Note that we apply projections $|D|_\ell$ when translating set operations with label ℓ. Hence, we never "mix" two set operations with different labels in the reduction. A crucial point here is that the parallel compositionality conditions make such mixing unnecessary, and this enables us to prove a strong relationship between translated constraints and projections:

Lemma 5.17. *Let $i \in \mathcal{L}$ be a label. If $\mathcal{B} \in tr_D(\mathcal{A})$ then $\mathcal{B}|_i \in tr_{|D|_i \cup |D|_\star}(\mathcal{A}|_i)$.*

By a straightforward induction proof over the structure of constraints we can also prove that *tr* preserves the properties we need for our compositionality result:

Lemma 5.18 (Preservation of well-formedness and compositionality). *If $\mathcal{A}$ is well-formed and parallel composable, and if $\mathcal{B} \in tr(\mathcal{A})$, then $\mathcal{B}$ is well-formed and parallel composable.*

Now the core idea is to reduce the compositionality problem for stateful constraints to stateless constraints using the translation *tr*. For that reason we need to show that the translation is correct, i.e., that the set of models of the input constraint is exactly the set of models of the translation:

Lemma 5.19 (Semantic equivalence of constraint reduction). *Let $\mathcal{A}$ be a constraint and let $D = \{\ell_1 : (t_1, s_1), \ldots, \ell_n : (t_n, s_n)\}$. Assume that all unifiable set operations occurring in $\mathcal{A}$ and D carry the same label, i.e., if $\ell : (t, s), \ell' : (t', s') \in \mathit{labeledsetops}(\mathcal{A}) \cup D$ and $\exists \delta.\ \delta((t, s)) = \delta((t', s'))$ then $\ell = \ell'$. Assume also that the set of variables occurring in D is disjoint from the bound variables of $\mathcal{A}$. Then the models of $\mathcal{A}$ are the same as the models of $tr(\mathcal{A})$, i.e., $[\![M, \mathcal{I}(D); \mathcal{A}]\!]\ \mathcal{I}$ iff there exists $\mathcal{B} \in tr_D(\mathcal{A})$ such that $[\![M, \emptyset; \mathcal{B}]\!]\ \mathcal{I}$.*

5.5.3 Putting Everything Together. For proving Theorem 5.10 we now only need to lift Lemma 5.15 to stateful constraints. That is, given $\mathcal{I} \models \mathcal{A}$ we obtain $\mathcal{B} \in tr(\mathcal{A})$ such that $\mathcal{I} \models \mathcal{B}$. For $\mathcal{B}$ we can apply Lemma 5.15; either $\mathcal{I}_\tau \models \mathcal{B}|_i$ for all $i \in \mathcal{L}$ or $\mathcal{B}$ leaks, under some well-typed model $\mathcal{I}_\tau$. Finally, with Lemma 5.19 and 5.17 we can show that either $\mathcal{I}_\tau \models \mathcal{A}|_i$ for all $i \in \mathcal{L}$ or $\mathcal{A}$ leaks. Thus, the compositionality result on the constraint-level is established.

5.6 The Main Result and Proof: Parallel Compositionality for Protocols

Until now our parallel compositionality result has been stated on the level of constraints. As a final step we now explain how we can use Theorem 5.10 to prove a parallel compositionality result for protocols. We first define our compositionality requirement on protocols that ensures that all reachable constraints are parallel composable:

Definition 5.20 (Parallel composability). Let $\mathcal{P} = \|_{i \in \mathcal{L}} \mathcal{P}_i$ be a composed protocol and let $\mathcal{S}$ be a ground set of terms disjoint from the basic public terms. Then $(\mathcal{P}, \mathcal{S})$ is *parallel composable* iff

(1) for all protocol-specific labels $\ell, \ell' \in \mathcal{L}$, if $\ell \neq \ell'$ then $\mathcal{P}|_\ell$ and $\mathcal{P}|_{\ell'}$ are $\mathcal{S}$-GSMP disjoint,
(2) for all $\ell : (t, s), \ell' : (t', s') \in \mathit{labeledsetops}(\mathcal{P})$, if (t, s) and (t', s'), after having their variables renamed apart, are unifiable then $\ell = \ell'$,
(3) $\mathcal{P}$ is type-flaw resistant and well-formed,
(4) for each transaction T of $\mathcal{P}$, only the first send action occurring in T may carry the $\star$ label,
(5) for each transaction T of $\mathcal{P}$ and for each action $\ell : a$ of T, if an attack constant of the form attack_i occurs in a then $\ell = i$. □

Note that, for a protocol $\mathcal{P}$, the terms occurring in $\mathcal{P}^\star$ is a subset of the terms occurring in $\mathcal{P}$. So for composed protocols $\|_{i \in \mathcal{L}} \mathcal{P}_i$ and any protocol-specific label $\ell \in \mathcal{L}$ the terms occurring in the projected protocol $\|_{i \in \mathcal{L}} \mathcal{P}_i|_\ell$ is equal to the terms occurring in the protocol $\mathcal{P}_\ell \parallel \mathcal{P}_1^\star \parallel \cdots \parallel \mathcal{P}_{\ell-1}^\star \parallel \mathcal{P}_{\ell+1}^\star \parallel \mathcal{P}_{\ell+2}^\star \parallel \cdots$. When $\mathcal{L} = \{1, 2\}$ the first condition of Definition 5.20 becomes the requirement that $\mathcal{P}_1 \parallel \mathcal{P}_2^\star$ and $\mathcal{P}_1^\star \parallel \mathcal{P}_2$ are $\mathcal{S}$-GSMP disjoint.

Note also that we in condition two need to ensure that the set-operations we are checking have disjoint sets of variables. The reason is that set-operations occur alpha-renamed in the reachable constraints, and so we have to consider that here as well.

For protocols we also need to require that their composition is type-flaw resistant. It is not sufficient to simply require it for the components in isolation; unifiable messages from different components might break type-flaw resistance otherwise. Note also that type-flaw resistance of a protocol $\mathcal{P}$ implies that the constraints reachable in $\mathcal{P}$ are type-flaw resistant, because $\mathit{SMP}(\mathcal{A}) \subseteq \mathit{SMP}(\mathcal{P})$ for any constraint $\mathcal{A}$ reachable in $\mathcal{P}$ and because these constraints consist of instances of the duals of transactions occurring in $\mathcal{P}$; likewise for GSMP disjointness. Thus if $(\|_{i \in \mathcal{L}} \mathcal{P}_i, \mathcal{S})$ is parallel composable then $(\mathcal{A}, \mathcal{S})$ is parallel composable for any constraint $\mathcal{A}$ reachable in $\|_{i \in \mathcal{L}} \mathcal{P}_i$.

Condition four is technically not required to establish the compositionality result, but it is beneficial as it allows for a simpler requirement on leakage than what is given in Theorem 5.10, namely one for which we do not need to involve prefixes of constraints, as the following illustrates:

Example 5.21. Let $\mathcal{S}$ be the set $\{s\}$ for a ground term s that is not a basic public term. Consider two protocols $\mathcal{P}$ and $\mathcal{P}'$, both consisting of a single transaction: $\mathcal{P}$ contains the transaction with

the transaction strand $1\colon \mathsf{send}\ s \cdot \star\colon \mathsf{send}\ s$ while $\mathcal{P}'$ contains the reversed transaction $\star\colon \mathsf{send}\ s \cdot 1\colon \mathsf{send}\ s$. The first protocol $\mathcal{P}$ does not satisfy condition four of Definition 5.20 because the second send action is labeled with $\star$, but the second protocol $\mathcal{P}'$ does. Since transactions are executed atomically the ordering of the send actions in transactions makes no difference. However, the constraint $1\colon \mathsf{receive}\ s \cdot \star\colon \mathsf{receive}\ s$ is reachable in $\mathcal{P}$, and has a proper prefix that leaks s, namely $1\colon \mathsf{receive}\ s$. This leakage can never occur in an actual run of $\mathcal{P}$ since the prefix is not reachable in $\mathcal{P}$. The prefix is also not reachable in $\mathcal{P}'$—in fact, no prefix of a reachable constraint in $\mathcal{P}'$ leaks s. Thus it is sufficient—and beneficial—to only require that reachable constraints of parallel composable protocols do not leak, rather than requiring it for all prefixes of the reachable constraints. □

Note that condition four is not a restriction; since transactions are executed atomically the ordering of the send actions has no effect on the result of taking a transaction.

Finally, condition five states that each attack constant is supposed to carry within it the label of the component protocol it occurs in. This is so that we can later express, by using the appropriate attack constant, that a particular component protocol has an attack. We therefore require that attack constants share the labels of the actions they occur in. Note that, by well-formedness, attack constants can only occur in actions of the form $\mathsf{send}\ \mathsf{attack}_i$, and condition five then implies that for composed protocols attack constants can only occur in actions of the form $i\colon \mathsf{send}\ \mathsf{attack}_i$. While this condition restricts where attack constants can occur, note that those constants exist solely to express when an attack has happened and condition five still allows for that.

Example 5.22. Continuing Example 5.8 we now show that $\mathcal{P}_{ks,1} \parallel \mathcal{P}_{ks,2}$ is parallel composable, i.e., that it satisfies the conditions of Definition 5.20. We have previously shown type-flaw resistance and well-formedness for a similar keyserver protocol in Example 3.3 and so we focus on the remaining conditions here. GSMP disjointness of the composed keyserver protocol was explained in Example 5.8. Hence the first condition of Definition 5.20 is satisfied. Note that $\mathit{labeledsetops}(\mathcal{P}_{ks,1} \parallel \mathcal{P}_{ks,2})$ consists of instances of labeled terms from the following set:

$$\{1\colon (PK_0, \mathsf{ring}(A_0)), 1\colon (PK_1, \mathsf{revoked}(A_1, S_1)), 2\colon (PK_2, \mathsf{seen}(A_2, S_2)),\\ \star\colon (PK_3, \mathsf{valid}(A_3, S_3)), \star\colon (PK_4^i, \mathsf{begin}_i(A_4^i, S_4^i)), \star\colon (PK_5^i, \mathsf{end}_i(A_5^i, S_5^i)) \mid i \in \{1, 2\}\}$$

For all pairs $\ell\colon (t, s)$, $\ell'\colon (t', s')$ in this set we have that $\ell = \ell'$ if (t, s) and (t', s') are unifiable (note that we do not need to apply variable-renaming because each pair of distinct elements in the set already have disjoint variables). Hence condition two is satisfied. Finally, the fourth and fifth conditions are satisfied since for each $i \in \{1, 2\}$ the constant attack_i only occurs in $\mathcal{P}_{ks,i}$ and each transaction with send actions only has the first of those labeled with $\star$. □

Two useful properties of parallel composable protocols are that their reachable constraints are also parallel composable and that attack constants actually denote that attacks have happened in those component protocols that share their labels:

Lemma 5.23. *If $(\mathcal{P}, \mathcal{S})$ is parallel composable then all constraints reachable in $\mathcal{P}$ are parallel composable.*

Lemma 5.24. *If $(\mathcal{P}, \mathcal{S})$ is parallel composable, $\mathcal{A}$ is reachable in $\mathcal{P}$, and if $\mathcal{I}(ik(\mathcal{A})) \vdash \mathsf{attack}_\ell$, then the action $\ell\colon \mathsf{receive}\ \mathsf{attack}_\ell$ occurs in $\mathcal{A}$ (and therefore also occurs in $\mathcal{A}|_\ell$).*

Coming back to the problem of leakage on the protocol-level we now define a notion of leakage for protocols that does not involve prefixes of constraints, but only those constraints that are reachable in the given protocol:

Definition 5.25 (Well-typed leakage-free protocol). Let $\mathcal{S}$ be a set of ground terms and let $\mathcal{P}$ be a protocol. Then $\mathcal{P}$ is *well-typed leakage-free* iff for all reachable constraints $\mathcal{A}$ in $\mathcal{P}$ of the

form $\mathcal{A} = \mathcal{A}' \cdot i\colon \text{receive } t_1, \ldots, t_n$, there does not exist a well-typed interpretation $\mathcal{I}_\tau$ and a $s \in \mathcal{S} \setminus \mathit{declassified}(\mathcal{A}, \mathcal{I}_\tau)$ such that $\mathcal{I}_\tau \models \mathcal{A} \cdot \mathsf{send}\, s$.

This is compatible with the notion of leakage on the constraint-level, as the following two lemmata show:

Lemma 5.26. *Let $(\|_{i\in\mathcal{L}}\mathcal{P}_i, \mathcal{S})$ be parallel composable, and let $\mathcal{P}$ be either the composed protocol $\|_{i\in\mathcal{L}}\mathcal{P}_i$ or a projection $\|_{i\in\mathcal{L}}\mathcal{P}_i|_\ell$ for a $\ell \in \mathcal{L}$. Let furthermore $\mathcal{A}$ be reachable in $\mathcal{P}$, $\mathcal{A}'$ be a prefix of $\mathcal{A}$, $\mathcal{I}$ an interpretation, and let $s \in \mathcal{S} \setminus \mathit{declassified}(\mathcal{A}', \mathcal{I})$, such that $\mathcal{I} \models \mathcal{A}' \cdot \mathsf{send}\, s$. Then there exists a prefix $\mathcal{B}$ of $\mathcal{A}$ with the following properties:*

- *$\mathcal{B}$ is reachable in $\mathcal{P}$.*
- *$\mathcal{B}$ ends in a* receive *action.*
- *$\mathit{declassified}(\mathcal{B}, \mathcal{I}) = \mathit{declassified}(\mathcal{A}', \mathcal{I})$.*
- *$\mathcal{I} \models \mathcal{B} \cdot \mathsf{send}\, s$.*

Lemma 5.27. *Let $\mathcal{P} = \|_{i\in\mathcal{L}}\mathcal{P}_i$ be a composed protocol, $\mathcal{S}$ be a set of shared terms, and let $(\mathcal{P}, \mathcal{S})$ be parallel composable. If all projections $\mathcal{P}|_\ell$, $\ell \in \mathcal{L}$, are well-typed leakage free then no prefix of a constraint reachable in $\mathcal{P}$ leaks a secret from $\mathcal{S}$ under any well-typed model.*

As a consequence of Theorem 5.10 and Lemma 5.27 we have that any protocol $\mathcal{P}_k$, where $k \in \mathcal{L}$, can be safely composed with any number of other protocols $\|_{i\in\mathcal{L}\setminus\{k\}}\mathcal{P}_i$ provided that $\mathcal{P}_k \parallel (\|_{i\in\mathcal{L}\setminus\{k\}}\mathcal{P}_i^\star) = \|_{i\in\mathcal{L}}\mathcal{P}_i|_k$ is secure and that none of the projected protocols $\|_{i\in\mathcal{L}}\mathcal{P}_i|_\ell$, where $\ell \in \mathcal{L}$, leak a secret:

Theorem 5.28. *Let $\mathcal{P} = \|_{i\in\mathcal{L}}\mathcal{P}_i$ be a composed protocol and let $k \in \mathcal{L}$. If $(\mathcal{P}, \mathcal{S})$ is parallel composable, $\mathcal{P}|_k$ is well-typed secure in isolation, and if for all $\ell \in \mathcal{L}$ the protocol $\mathcal{P}|_\ell$ is well-typed leakage-free, then all goals of $\mathcal{P}_k$ hold in $\mathcal{P}$ (even in the untyped model).*

Note that the only requirement on the projected protocols $\|_{i\in\mathcal{L}}\mathcal{P}_i|_\ell$ is that they do not leak any shared terms (before declassifying), but we do not require that they are completely secure. This means that, if we have a secure protocol $\mathcal{P}_1$, then the goals of $\mathcal{P}_1$ continue to hold in any composition with another protocol $\mathcal{P}_2$ that satisfies the composability conditions and does not leak shared terms, even if $\mathcal{P}_2$ has some attacks. This is in particular interesting if we run a protocol $\mathcal{P}_1$ in composition with numerous other protocols that are too complex to verify in all detail.

As a corollary we have that the composition of composable and secure protocols is secure:

Corollary 5.29. *If $(\mathcal{P}_1 \parallel \cdots \parallel \mathcal{P}_N, \mathcal{S})$ is parallel composable and if for all $i \in \{1, \ldots, N\}$ the protocol $\mathcal{P}_1^\star \parallel \cdots \parallel \mathcal{P}_{i-1}^\star \parallel \mathcal{P}_i \parallel \mathcal{P}_{i+1}^\star \parallel \cdots \parallel \mathcal{P}_N^\star$ is well-typed secure in isolation and well-typed leakage-free then the composition $\mathcal{P}_1 \parallel \cdots \parallel \mathcal{P}_N$ is also secure (even in the untyped model).*

5.7 Discussion & Limitations

Our result requires that the messages shared between protocols are either public or initially secret (but can be declassified at a later point). Our result allows both keys and even more complex messages to be shared between protocols, and these messages can even be declassified dynamically.

Additionally, we allow for sharing of sets. This is already much more general than the existing works, but of course there are requirements on the shared sets that are necessary for the composition to work, e.g., the protocol designer has to provide the right interface to the sets and the component protocols must adhere to this interface.

One limitation of our result lies in the supported primitives. With Ana we can model the standard decryption operations, which allows us to model and analyze a wide range of protocols. Still, there are also relevant and important protocols that we cannot model: since we are working in the free

algebra, terms are equal if and only if they are syntactically equal, and so we cannot directly model, for instance, that some operators are commutative, which is needed for Diffie-Hellman or blind signatures. We are currently working on a result that supports algebraic properties; this requires generalizations at every level of the Isabelle formalization, e.g., from terms to *equivalence classes* of terms.

6 CHECKING THE CONDITIONS AUTOMATICALLY

While the results established in the previous section comprise our main compositionality theorems, actually applying them in practice requires a bit more work: GSMP disjointness and type-flaw resistance, in particular, are non-trivial to prove. In this section we show how these syntactic checks can be fully automated. As with all the other theory in this paper, everything has been formalized in Isabelle/HOL and integrated with PSPSP [25]. But note that the following is not limited by the restrictions of the automated verification approach of PSPSP: we can still model protocols and check the compositionality conditions even if we cannot automatically verify their security goals.

6.1 Finite SMP Representations

We first tackle the issue of automatically checking type-flaw resistance. The only difficulty with checking type-flaw resistance of a protocol $\mathcal{P}$ (and of transactions) is that the set of its sub-message patterns, $SMP(trms(\mathcal{P}) \cup setops(\mathcal{P}))$, is usually infinite. One solution [27] is to instead compute a finite set that represents, as well-typed instances, all the sub-message patterns of $\mathcal{P}$, and then substitute this set for the set of sub-message patterns when checking type-flaw resistance. To that end we first define what it means for a term to be a well-typed instance:

Definition 6.1 (Well-typed term instance). A term t is a *well-typed instance of* s iff s matches t and $\Gamma(t) = \Gamma(s)$ (i.e., there exists a well-typed substitution θ such that $t = \theta(s)$). Moreover, if N and M are sets of terms then N is a *well-typed-instance subset of* M iff for all $t \in N$ there exists an $s \in M$ such that t is a well-typed instance of s. □

One issue with finding a finite set to represent the sub-message patterns of a protocol $\mathcal{P}$ is that the set of sub-message patterns is closed under both subterms and well-typed instances. Since there may be variables with composed type in $\mathcal{P}$, all proper subterms of well-typed instances of those variables occur in $SMP(\mathcal{P})$ as well. To actually cover these terms we need to ensure that there exist instances of the composed-typed variables that are "as general as possible", and for that reason we define the following notion of a set being closed under such instances:

Definition 6.2 (Composed-type-instance closed). A set of terms M is composed-type-instance closed iff for all $x \in fv(M)$ with a composed type of the form $\Gamma(x) = f(\tau_1, \ldots, \tau_n)$ there exists a term $f(y_1, \ldots, y_n) \in M$ where the variables $y_1 \ldots, y_n$ are distinct and where each y_i has type τ_i. □

Now we can define what it means for a finite set M to represent the sub-message patterns of a protocol. Since the *SMP* set is closed under both subterms and analysis keys we need to, in particular, require that those terms are covered by the set M. In addition, the set needs to be composed-type-instance closed. We call such sets *finite SMP representations*:

Definition 6.3 (Finite SMP representations). A finite set of terms M is called an *SMP representation* iff the following conditions are satisfied:

(1) $subterms(M)$ and $\{k \in K \mid \mathsf{Ana}(t) = (K, R), t \in M\}$ are well-typed-instance subsets of M.
(2) M is composed-type-instance closed.

Furthermore, M is called an *SMP representation for the protocol* $\mathcal{P}$ iff M is an SMP representation and $trms(\mathcal{P}) \cup setops(\mathcal{P})$ is a well-typed-instance subset of M. □

The following lemma states that such finite SMP representations are indeed sufficient for checking type-flaw resistance:

Lemma 6.4 (Computable type-flaw resistance of sets). *Let M be a finite set of terms and let δ be a well-typed variable-renaming such that $fv(M) \cap fv(\delta(M)) = \emptyset$. Then M is type-flaw resistant if the following conditions hold:*

(1) M is a finite SMP representation.
(2) For all $t, s \in M \setminus \mathcal{V}$, if t and $\delta(s)$ are unifiable then $\Gamma(t) = \Gamma(s)$.

A simple algorithm for computing *SMP* representation sets is to compute the following inductively defined set whose rules mimic those for the *SMP* set but restricts instantiation to only those terms that are needed to ensure that composed-typed variables are sufficiently instantiated:

Definition 6.5 (Computing SMP representation sets). Let M be a finite set of terms and let $\mathfrak{v}_i^\tau$, for any i, be a distinct variable of type τ that does not occur in M (i.e., $\mathfrak{v}_i^\tau \neq \mathfrak{v}_j^\tau$ whenever $i \neq j$). Then $SMP_0(M)$ is defined as the least set closed under the following rules:

(1) $M \subseteq SMP_0(M)$.
(2) If $t \in SMP_0(M)$ and $s \sqsubseteq t$ then $s \in SMP_0(M)$.
(3) If $t \in SMP_0(M)$ and $\mathsf{Ana}(t) = (K, R)$ then $K \subseteq SMP_0(M)$.
(4) If $x \in SMP_0(M) \cap \mathcal{V}$ and $\Gamma(x) = f(\tau_1, \ldots, \tau_n)$, $n > 0$, then $f(\mathfrak{v}_1^{\tau_1}, \ldots, \mathfrak{v}_n^{\tau_n}) \in SMP_0(M)$. □

Note that SMP_0 is closed under analysis keys and so $SMP_0(M)$ is not guaranteed to be finite. For instance, the rule $\mathsf{Ana}(f(x)) = (\{f(f(x))\}, R)$ would technically lead to an infinite $SMP_0(M)$ set if a term of the form $f(x)$ occurs in M. Besides such artificial examples, however, the set will be finite.

6.2 Automating the Parallel Composability Conditions

With type-flaw resistance solved we can now consider the remaining conditions that are needed for parallel compositionality. Here there are two issues that need to be solved. First, the set of shared terms is—like the set of sub-message patterns—an infinite set in general, and we again want to instead consider a symbolic finite set that represents the set of shared terms. For such a finite set S a reasonable choice is to consider it a representation of the set of shared terms consisting of all the well-typed ground instances of terms in S that are not basic public terms:

Definition 6.6 (Finite shared terms representation). Let S be a finite set of symbolic terms and let f be the function defined as $f(t) \equiv \{\delta(t) \mid fv(\delta(t)) = \emptyset, \Gamma(\delta(t)) = \Gamma(t)\}$ for all terms t. Then S is a *finite shared terms representation* of the set $\mathcal{S}_S$ defined as $\mathcal{S}_S \equiv \{t \in f(s) \mid s \in S\} \setminus \{t \mid \emptyset \vdash t\}$. □

Second, we need to finitely represent GSMP sets as well. Fortunately, we can here directly reuse the notion of finite SMP representations since GSMP sets consist, after all, of ground SMP terms. To automate the GSMP disjointness condition we then need to define a variant of GSMP disjointness that works on finite SMP representations:

Definition 6.7 (Computable GSMP disjointness). Let S be a finite shared terms representation of $\mathcal{S}_S$, let M_1 and M_2 be two sets of terms, and let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two protocols. Then $\mathcal{P}_1$ and $\mathcal{P}_2$ satisfy the computable GSMP disjointness conditions w.r.t. S, M_1, and M_2 iff the following conditions hold:

(1) $fv(M_1) \cap fv(M_2) = \emptyset$, and
(2) M_1 is a finite SMP representation for $\mathcal{P}_1$ and M_2 is a finite SMP representation for $\mathcal{P}_2$, and
(3) for all $t_1 \in M_1$ and $t_2 \in M_2$, if t_1 and t_2 are of the same type and unifiable then either $\emptyset \vdash t_1$ and $\emptyset \vdash t_2$ or there exists a $s \in S$ such that t_1 and t_2 are well-typed term instances of s. □

Note that determining if a term is a basic public term, i.e. if $\emptyset \vdash t$, is easy: since the intruder knowledge is here empty, and since nothing new can be derived by decomposing something that has

just been composed, deriving t can be accomplished solely by repeatedly applying the (*Compose*) rule of Definition 2.1. In other words, $\emptyset \vdash f(t_1, \ldots, t_n)$ iff $f \in \Sigma^n_{pub}$ and $\emptyset \vdash t_i$ for all $i \in \{1, \ldots, n\}$.

Finally, we can now prove the following lemma that gives sufficient conditions for checking GSMP disjointness:

Lemma 6.8 (Automated GSMP disjointness). *Let S be a finite shared terms representation of $\mathcal{S}_S$, let M_1 and M_2 be two sets of terms, and let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two protocols. If $\mathcal{P}_1$ and $\mathcal{P}_2$ satisfy the computable GSMP disjointness conditions w.r.t. S, M_1, and M_2, then $\mathcal{P}_1$ and $\mathcal{P}_2$ are $\mathcal{S}_S$-GSMP disjoint.*

The remaining parallel compositionality conditions are trivially automated, and so we have the following lemma that gives sufficient computable conditions for parallel compositionality to hold:

Lemma 6.9 (Sufficient parallel compositionality conditions). *Let S be a finite shared terms representation of $\mathcal{S}_S$. Let $\mathcal{P}$ be a composed protocol and let $L \subseteq \mathcal{L}$, $|L| \geq 2$, be the set of protocol-specific labels occurring in $\mathcal{P}$. For each $\ell \in L$ let M_ℓ be a finite set of terms. If*

(1) for all $\ell, \ell' \in L$, if $\ell \neq \ell'$ then $\mathcal{P}|_\ell$ and $\mathcal{P}|_{\ell'}$ satisfy the sufficient GSMP disjointness conditions w.r.t. S, M_ℓ, and $M_{\ell'}$, and

(2) conditions (2), (3), (4), and (5) of Definition 5.20 hold for $\mathcal{P}$,

then $(\mathcal{P}, \mathcal{S}_S)$ is parallel composable.

7 OTHER FORMS OF COMPOSITION

There have been several works that consider other forms of composition, e.g., [11] for sequential composition, i.e., where one protocol, say, exchanges a session key, and another protocol uses that key. The generality of our result—that we can compose protocols that share sets—allows to simply express this as a special case of parallel composition (with shared sets): one protocol performs a key exchange and puts the key into a set (on the sender and receiver side), the other protocol obtains the key from there. The particular advantage is that we can support, in fact, arbitrary interactions of the protocols, not necessarily strictly sequential ones, without having to prove a new result for that form of composition.

As an example, we have modeled a simplified version of TLS 1.2 for an unauthenticated user negotiating shared keys with a web server. (TLS 1.3 requires algebraic properties, support for which we are currently working on.) Moreover, we have modeled the SAML Single-Sign On protocol, that assumes two unilaterally authenticated channels like TLS provides them: 1. between a user and an identity provider on which the user authenticates themselves using a password, 2. between the user and a relying party on which the user forwards a credential from the identity provider. As an interface between the two protocols we use shared sets where TLS delivers negotiated keys, and SSO picks them up. We have checked in Isabelle that they satisfy the requirements of our compositionality result.

The specifications are too long to present here in full; it is found in the additional material [29]. Instead, we give a simpler example of a key exchange with unilateral authentication (a trivialized TLS if you will) and compose it with a simple login. This still allows for illustrating the essentials of sequential-style composition.

7.1 A Sequential Composition Example

The first protocol is shown in Fig. 3. Let the set of shared terms $\mathcal{S}$ consist of all private constants of type skey (since they will represent shared keys) and the private keys $\mathsf{inv}(\mathsf{pk}(A))$ of every agent A. We also consider pk as a public function here, so that all public keys are public terms. We will also consider passwords, pw, but they are not part of $\mathcal{S}$ since they only occur in the second protocol. The initialKnowledge rule is releasing the private key of every dishonest agent B and,

initialKnowledge(B: dis)
- ⋆ send inv(pk(B)).

clientHello(A: hon, B: hon)
- new K: skey
- ⋆ insert K clientSessionKeys(A, B)
- send crypt(pk(B), newSession(A, B, K)).

serverRcv(A: agent, B: hon, K: skey)
- receive crypt(pk(B), newSession(A, B, K))
- ⋆ insert K serverSessionKeysUnauth(B).

authGoal(A, B: hon, B': agent, K: skey)
- ⋆ K in clientSessionKeys(A, B)
- ⋆ K in serverSessionKeysUnauth(B')
- ⋆ $B \neq B'$
- attack.

intruderGen()
- new K: skey
- ⋆ insert K dishonest
- ⋆ send K.

clientHelloD(A: hon, B: dis)
- new K: skey
- ⋆ insert K clientSessionKeys(A, B)
- ⋆ send K
- send crypt(pk(B), newSession(A, B, K)).

secrecyGoal(A: hon, B: hon, K: skey)
- receive K
- ⋆ K in clientSessionKeys(A, B)
- attack.

Fig. 3. A key-exchange protocol without client authentication

in the login protocol, every password that B has with an honest agent A. The rule intruderGen allows the intruder to create fresh keys and insert them into a special set dishonest that contains all intruder-generated keys.

Next, the rule clientHello defines how an honest sender A can start the protocol with an honest B: A generates a fresh key K and inserts it into its clientSessionKeys set, and this step is labeled ⋆ as we use this set as an interface to the subsequent protocol. Finally, A sends a message encrypted with the public key of B and containing K under a format newSession. The rule clientHelloD is the same for the case that B is dishonest. The difference to the previous rule is only that K is released here. Note that one could have unified this rule with the previous clientHello by simply setting B: agent; the intruder can obtain the K from the encrypted message since they have inv(pk(B)) for every dishonest B. However, the ⋆-labeled step send K means K is explicitly declassified; without this, the protocol would actually be leaking a secret (since all keys are in $\mathcal{S}$).

Note that these rules do not authenticate A, but intuitively in some sense B is authenticated, because only the intended recipient can read the key generated by A. This is best observed in the following rule serverRcv: here an honest server B receives a message from any client A (honest or dishonest) and stores it in the set serverSessionKeysUnauth(B). Note that this set is parameterized only over the name B, in contrast to its counter-part clientSessionKeys(A, B) that has both names as parameters. This is because the server cannot be sure about the identity of A, and the server does not even note the name that the sender claims to be. (We could, in fact, omit the sender-name in the encrypted message without introducing a vulnerability.)

This models an almost trivial key-exchange protocol without client authentication; note it does not even prevent replay. It is somewhat similar to what old TLS up to version 1.2 did, i.e., the client generating a (basis for) the symmetric key, and encrypting it with the public key of the server.

We define two goals here: secrecyGoal simply states that the intruder must not obtain the session key that an honest A has generated for an honest B; note this goal expresses basically A's point of view. That this corresponds also to B's view then follows from the authGoal, namely that if an honest A has generated a key for an honest B, then no other honest agent $B' \neq B$ can receive this key. This is a variant of standard non-injective (i.e., not counting replay as an attack) authentication, adapted to unilateral authentication.

As protocol to sequentially compose, we consider now the login protocol shown in Fig. 4. It has the same intruder rules as the key-exchange protocol. The first step of this protocol is sendPW, where an honest client A who wants to log in to the server B (who could be honest or dishonest)

initialKnowledge′(A: hon, B: dis)
- ⋆ send inv(pk(B))
- send pw(B, A).

sendPW(A: hon, B: agent, K: skey)
- ⋆ K in clientSessionKeys(A, B)
- send scrypt(K, login(A, B, pw(A, B))).

authenticateKey(A: hon, B: hon, K: skey)
- receive scrypt(K, login(A, B, pw(A, B)))
- ⋆ K in serverSessionKeysUnauth(B)
- ⋆ K in clientSessionKeys(A, B)
- ⋆ delete K serverSessionKeysUnauth(B)
- ⋆ insert K serverSessionKeysAuth(A, B).

agreementLogin1(A: hon, B: hon, K: skey)
- receive scrypt(K, login(A, B, pw(A, B)))
- ⋆ K in serverSessionKeysUnauth(B)
- ⋆ K notin clientSessionKeys(A, B)
- attack.

intruderGen′()
- new K: skey
- ⋆ insert K dishonest
- ⋆ send K.

authenticateKeyD(A: dis, B: hon, K: skey)
- receive scrypt(K, login(A, B, pw(A, B)))
- ⋆ K in serverSessionKeysUnauth(B)
- ⋆ delete K serverSessionKeysUnauth(B)
- ⋆ insert K serverSessionKeysAuth(A, B).

secrecyLogin(A: hon, B: hon, K: skey)
- receive K
- ⋆ K in serverSessionKeysAuth(A, B)
- attack.

agreementLogin2(A: hon, B: hon, K: skey)
- ⋆ K in serverSessionKeysAuth(A, B)
- ⋆ K notin clientSessionKeys(A, B)
- attack.

Fig. 4. A login protocol for sequential composition with the key-exchange from Fig. 3.

takes one of the clientSessionKeys(A, B) (i.e., that were negotiated by the key-exchange protocol) and encrypts its password pw(A, B) with it (and structure the message with the format login).

There are several rules that describe how an honest server B can receive such a message. First, authenticateKey checks that there is a key K in serverSessionKeysUnauth(B) (i.e., that has been negotiated previously with *somebody*). Before this transaction, the key K is unauthenticated, i.e., B cannot be sure who created K (the person claiming to be A). Decisive for the authentication is that this message contains the password pw(A, B) (and, if the protocol works correctly, it is not known to the intruder unless A or B is dishonest). The rule authenticateKey is for a particular case only: that A is honest and K in clientSessionKeys(A, B). Recall that clientSessionKeys is actually a set maintained by A (containing the keys A has created for use with B), so the server B would not be able to check this condition. However, observe that in the case K notin clientSessionKeys(A, B) the rule agreementLogin1 is applicable, i.e., then we would have a violation of authentication, because B would be accepting at this point a key K as authenticated by A while A has actually not created this key for use with B. Since the case K notin clientSessionKeys(A, B) triggers an attack, it is without loss of attacks to restrict the authenticateKey rule to the case K in clientSessionKeys(A, B), i.e., that A has indeed meant K for communication with B. In this case, B moves the key K from serverSessionKeysUnauth(B) to serverSessionKeysAuth(A, B), i.e., registering the key as authenticated.

The case of B receiving the login from a dishonest A is modeled by the rule authenticateKeyD. Here, the server directly registers K as authenticated by A. Finally, we have the following additional goals about keys that end up in the set of authenticated keys between honest A and B: secrecyLogin says they must not be known by an intruder, and agreementLogin2 means that the client A indeed created them for use with B, i.e., the standard non-injective agreement on the key.

Call P_1 the key-exchange from Fig. 3 and call P_2 the login from Fig. 4. We have verified in Isabelle with PSPSP [25] for two honest agents and one dishonest agent that $P_1 \parallel P_2^\star$ and $P_1^\star \parallel P_2$ are secure, i.e., each protocol in parallel with the abstraction of the other is secure. With the composition framework we have further proved automatically—without any bounds on the number of agents—that P_1 and P_2 are parallel composable with respect to the set of shared terms consisting of the private keys, the passwords, and the symmetric keys. Further, since P_1 and P_2 do not leak we have

that $P_1 \parallel P_2$ is secure, which models in fact a sequential composition where P_1 establishes shared keys and P_2 uses them.

Observe that the key-exchange protocol P_1 can in fact be replaced by any key-exchange protocol P_1' that induces the same abstraction $P_1^\star$, and similar for the login P_2 (and of course the parallel composability conditions have to hold, but that is essentially that the message formats of the protocols do not interfere). Let us thus review these interface $P_1^\star$ and $P_2^\star$ again more closely, i.e., restricting the protocols to $\star$-labeled steps.

$P_1^\star$ says that the private key and passwords of the intruder are declassified and that the intruder can generate new keys that are inserted into the set dishonest (rules initialKnowledge and intruderGen). Moreover, any agent can create fresh keys and insert them into clientSessionKeys(A, B), and if B is dishonest, these keys are declassified and given to the intruder (rules clientHello and clientHelloD). Since the protocol must not leak shared terms, it is an implicit consequence of this specification that all other keys must thus be kept secret. Moreover, an honest B will accept arbitrary keys into serverSessionKeysUnauth (rule serverRcv). (The attack rules secrecyGoal and authGoal do not contribute to $P_1^\star$, since their consequence attack is not labeled $\star$.)

In $P_2^\star$, sendPW and the attack rules are not contributing for the same reason. The two rules authenticateKey and authenticateKeyD in the abstracted versions now say that any unauthenticated key K at server B can be moved to the authenticated keys for an agent A, if A is honest and indeed considers K as a session key with B, or if A is dishonest.

Both protocols' interfaces thus completely abstract from the way this is implemented: they only talk about asymmetric keys and passwords only as far as what the intruder initially knows.[3]

In the additional material [29] you can find an example that goes beyond the simple sequential composition by adding another protocol that updates keys for the above composition.

7.2 Vertical Composition

In [19] we have considered vertical compositionality, where vertical means that we have one "high-level" application protocol that uses a "low-level" channel protocol for tasks like secure transport, e.g., a banking service running over TLS. The vertical composition paper, in fact, is building on the compositionality result of this paper: it uses the parallel composition as one step in its construction, where sets act as an interface between high- and low-level protocols, similar to what we have done in this section. The particular difficulty is that, due to the vertical nature of this composition, the low-level protocol *embeds* the messages of the high-level protocol, and thus the messages of the high-level protocol need to be part of $\mathcal{S}$ to satisfy our composition requirements. [19] therefore extends the typing system to allow for an abstract *payload* data type (so that one can use an arbitrary high-level protocol), and then shows in several transformation steps that it is sufficient to verify the low-level protocol with nonces as payloads, though one has to allow for both fresh and repeated nonces, as well as for both secret and public nonces to cover all cases of payloads in an abstract way.

8 CONCLUSION AND RELATED WORK

Our composition theorem for parallel composition is the latest in a sequence of parallel composition results, each of them pushing the boundaries of the class of protocols that can be composed [2–4, 12, 14, 15, 21–23]. The first results simply require completely disjoint encryptions; subsequent results allowed the sharing of long-term keys, provided that wherever the common keys are used, the content messages of the different protocols are distinguished, for instance by tagging. Other

[3]One can easily improve the key-exchange so that it satisfies injective agreement as well (i.e., old keys are not accepted by a server a second time). The keyserver example from the previous section achieves this.

aspects are which primitives are supported as well as what forms of negative conditions, e.g., to support as goals the full geometric fragment [2], and under which conditions privacy properties can be preserved under protocol composition [4].

Our result lifts the common requirement that the component protocols only share a fixed set of long-term public and private constants. Our result allows for stateful protocols that maintain databases (such as a key server) and the databases may even be shared between these protocols. This includes the possibility to declassify long-term secrets, e.g., to verify that a protocol is even secure if the intruder learns all old private keys. Both databases, shared databases, and declassification are considerable generalizations over the existing results.

Like [2] our result links the parallel compositionality result with a typing result such as the result of [27], i.e., essentially requiring that all messages of different meaning have a distinguishable form. Under this requirement it is sound to restrict the intruder model to using only well-typed messages which greatly simplifies many related problems. While one may argue that such a typing result is not strictly necessary for composition, we believe it is good practice and also fits well with disjointness requirements of parallel composition. Moreover, many existing protocols already satisfy our typing requirement, since, unlike tagging schemes, this does not require a modification of a protocol as long as there is some way to distinguish messages of different meaning.

There are other types of compositionality results for sequential and vertical composition, where the protocols under composition do build upon each other, e.g., one protocol establishes a key that is then subsequently used by another protocol [3, 11, 15, 18–20, 38]. This requires that one protocol satisfies certain properties (e.g. that the key exchange is authenticated and secret) for the other protocol to rely on.

As the example of sequential composition in Sec. 7 shows the support of shared databases in a compositionality result opens a whole alley of applications: we can use the databases as an interface between components of a system. Here, the system thus work in parallel but not independent of each other. In all examples in this paper, each database has been modeled as belonging to one particular agent. Thus, when using databases as an interface between components, they are in fact *not* communications between different agents (via shared memory), but communications through an interface between components belonging to a single agent. Each of these components, however, can communicate over the normal network with components at other agents. Hence, this allows for integrating into the security question more traditional compositionality aspects from software, that are not concerned with a notion of attackers [7, 8, 16, 17, 36].

So far, compositionality for security protocols results are almost exclusively "paper-and-pencil" proofs. The proof arguments are often quite subtle, e.g., given an attack where the intruder learned a nonce from one protocol and uses it in another protocol, one has to prove that the attack does not rely on this, but would similarly work for distinct nonces. It is not uncommon that parts of such proofs are a bit sketchy with the danger of overlooking some subtle problems as for instance described in [26]. For this reason, we have formalized the compositionality result in the proof assistant Isabelle/HOL [39], extending the formalization of [24, 26, 27], giving the extremely high correctness guarantee of machine-checked proofs. To our knowledge, this work is the first such formalization within the symbolic model of a compositionality result in a proof assistant, with the notable exception of a study in Isabelle/HOL of compositional reasoning on concrete protocols [10].

Besides the symbolic model, there are several works about compositionality in the cryptographic research [6, 9, 33, 34]. Here, one describes components both in terms of a real system and an ideal functionality and in the composition one basically proves that the real system is indeed indistinguishable for an attacker from the ideal system. There are first works [1] towards formalizing this step in a theorem prover. Then in proving the high-level system one can instead rely on the ideal functionality of the low-level components. In this way one can work in a security proof

upwards in several layers from a low cryptographic layer to the high application layer. There is a similarity in the real vs. ideal paradigm and our paradigm of a protocol $\mathcal{P}$ and its abstraction $\mathcal{P}^\star$ and the fact that in a compositionality proof we can rely on $\mathcal{P}^\star$. However, the setup is substantially different: $\mathcal{P}^\star$ represents *the declassifications and changes* that the protocol $\mathcal{P}$ can make to databases shared with another protocol $\mathcal{P}'$ and the compositionality proof then shows that $\mathcal{P}'$ is secure when the environment only makes declassifications and changes contained in $\mathcal{P}^\star$ (and vice-versa for $\mathcal{P}$ and $\mathcal{P}'^\star$). While an ideal system may abstract communicated messages, hiding the cryptographic implementation in the abstract version, our approach rather has to give verbatim the interaction that happens. This means also that we have the concept of shared messages and declassification when a message is available to the intruder. Thus, the cryptographic compositionality frameworks are closer to vertical composition in the sense of [19]. While the cryptographic compositionality frameworks perform proofs on the cryptographic level, our work is on a symbolic (term-algebra) level. This makes many arguments easier for us and allows us to argue on a transition system level where the interaction with shared databases can be expressed at least much easier.

This has indeed another advantage of the present approach: our result is closely linked to automated methods for protocol verification, in particular our PSPSP tool [25] that allows for automated verification of a large class of stateful protocols. This means that not only can a complex system—composed of several component protocols—be proved correct entirely in Isabelle, but also that this proof can be obtained automatically.

ACKNOWLEDGMENTS

This work was supported by the Sapere-Aude project "Composec: Secure Composition of Distributed Systems", grant 4184-00334B of the Danish Council for Independent Research and by the "CyberSec4Europe" European Union's Horizon 2020 research and innovation programme under grant agreement No 830929. We thank Luca Viganò, Anders Schlichtkrull, and the reviewers for helpful comments.

REFERENCES

[1] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Catalin Hritcu, Kenji Maillard, and Bas Spitters. 2021. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. In *CSF*. IEEE, 1–15.

[2] Omar Almousa, Sebastian Mödersheim, Paolo Modesti, and Luca Viganò. 2015. Typing and Compositionality for Security Protocols. In *ESORICS (LNCS, Vol. 9327)*. Springer, Heidelberg, Germany, 209–229.

[3] Suzana Andova, Cas J. F. Cremers, Kristian Gjøsteen, Sjouke Mauw, Stig Fr. Mjølsnes, and Saša Radomirović. 2008. A framework for compositional verification of security protocols. *Inf. Comput.* 206, 2-4 (2008), 425–459.

[4] Myrto Arapinis, Vincent Cheval, and Stéphanie Delaune. 2015. Composing Security Protocols: From Confidentiality to Privacy. In *POST*, Riccardo Focardi and Andrew Myers (Eds.). Springer, Heidelberg, Germany, 324–343.

[5] Myrto Arapinis and Marie Duflot. 2014. Bounding messages for free in security protocols - extension to various security properties. *Inf. Comput.* 239 (2014), 182–215.

[6] Michael Backes, Birgit Pfitzmann, and Michael Waidner. 2007. The reactive simulatability (RSIM) framework for asynchronous systems. *Inf. Comput.* 205, 12 (2007), 1685–1720.

[7] Manfred Broy. 1997. Interactive and Reactive Systems: States, Observations, Experiments, Input, Output, Nondeterminism, Compositionality and all That. In *Foundations of Computer Science (LNCS, Vol. 1337)*, Christian Freksa, Matthias Jantzen, and Rüdiger Valk (Eds.). Springer, Heidelberg, Germany, 279–286.

[8] Achim D. Brucker and Burkhart Wolff. 2008. An Extensible Encoding of Object-oriented Data Models in HOL. *Journal of Automated Reasoning* 41 (2008), 219–249. Issue 3.

[9] Chris Brzuska, Antoine Delignat-Lavaud, Konrad Kohbrok, and Markulf Kohlweiss. 2018. State-Separating Proofs: A Reduction Methodology for Real-World Protocols. *IACR Cryptol. ePrint Arch.* 306 (2018).

[10] Denis Frédéric Butin. 2012. *Inductive analysis of security protocols in Isabelle/HOL with applications to electronic voting*. Ph.D. Dissertation. Dublin City University.

[11] V. Cheval, V. Cortier, and B. Warinschi. 2017. Secure Composition of PKIs with Public Key Protocols. In *CSF*. IEEE, 144–158.

[12] Céline Chevalier, Stéphanie Delaune, Steve Kremer, and Mark D. Ryan. 2013. Composition of password-based protocols. *Formal Methods in System Design* 43, 3 (2013), 369–413.
[13] Rémy Chrétien, Véronique Cortier, Antoine Dallon, and Stéphanie Delaune. 2020. Typing Messages for Free in Security Protocols. *ACM Trans. Comput. Log.* 21, 1 (2020), 1:1–1:52. https://doi.org/10.1145/3343507
[14] Véronique Cortier and Stéphanie Delaune. 2009. Safely composing security protocols. *Formal Methods in System Design* 34, 1 (2009), 1–36.
[15] Ştefan Ciobâcă and Véronique Cortier. 2010. Protocol Composition for Arbitrary Primitives. In *CSF*. IEEE, 322–336.
[16] Willem-Paul de Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. 2012. *Concurrency Verification: Introduction to Compositional and Non-Compositional Methods*. Cambridge University Press, USA.
[17] Hartmut Ehrig, Bernd Mahr, Ingo Claßen, and Fernando Orejas. 1992. Introduction to Algebraic Specification. Part 1: Formal Methods for Software Development. *Comput. J.* 35, 5 (1992), 460–467.
[18] Santiago Escobar, Catherine A. Meadows, José Meseguer, and Sonia Santiago. 2010. Sequential Protocol Composition in Maude-NPA. In *ESORICS*. Springer, Heidelberg, Germany, 303–318.
[19] Sébastien Gondron and Sebastian Mödersheim. 2021. Vertical Composition and Sound Payload Abstraction for Stateful Protocols. In *CSF*. IEEE, 1–16. https://doi.org/10.1109/CSF51468.2021.00038
[20] T. Groß and S. Mödersheim. 2011. Vertical Protocol Composition. In *CSF*. IEEE, 235–250.
[21] Joshua D. Guttman. 2009. Cryptographic Protocol Composition via the Authentication Tests. In *FOSSACS*. Springer, Heidelberg, Germany, 303–317.
[22] Joshua D. Guttman and F. Javier Thayer. 2000. Protocol Independence through Disjoint Encryption. In *CSFW*. IEEE Computer Society, 24–34. https://ieeexplore.ieee.org/xpl/conhome/6924/proceeding
[23] N. Heintze and J. D. Tygart. 1994. A model for secure protocols and their compositions. In *Security and Privacy*. IEEE, 2–13.
[24] Andreas Viktor Hess. 2019. *Typing and Compositionality for Stateful Security Protocols*. Ph.D. Dissertation. Technical University Denmark.
[25] Andreas Viktor Hess, Sebastian Alexander Mödersheim, Achim D. Brucker, and Anders Schlichtkrull. 2021. Performing Security Proofs of Stateful Protocols. In *CSF*. IEEE, United States, 1–16.
[26] Andreas Viktor Hess and Sebastian Mödersheim. 2017. Formalizing and Proving a Typing Result for Security Protocols in Isabelle/HOL. In *CSF*. IEEE, 451–463.
[27] Andreas Viktor Hess and Sebastian Mödersheim. 2018. A Typing Result for Stateful Protocols. In *CSF*. IEEE, 374–388.
[28] Andreas V. Hess, Sebastian Mödersheim, and Achim D. Brucker. 2020. Stateful Protocol Composition and Typing. *Archive of Formal Proofs* (April 2020). https://isa-afp.org/entries/Stateful_Protocol_Composition_and_Typing.html.
[29] Andreas V. Hess, Sebastian Mödersheim, and Achim D. Brucker. 2022. Stateful Protocol Composition in Isabelle/HOL - Supplementary Material. https://people.compute.dtu.dk/samo/StateParCompAdditionalMaterial.tgz.
[30] Andreas V. Hess, Sebastian Mödersheim, Achim D. Brucker, and Anders Schlichtkrull. 2020. Automated Stateful Protocol Verification. *Archive of Formal Proofs* (April 2020). https://isa-afp.org/entries/Automated_Stateful_Protocol_Verification.html.
[31] Andreas Viktor Hess, Sebastian Alexander Mödersheim, and Achim D. Brucker. 2018. Stateful Protocol Composition. In *ESORICS 2018 (LNCS, Vol. 11098)*. Springer, Heidelberg, Germany, 427–446. Extended version [32].
[32] A. V. Hess, S. A. Mödersheim, and A. D. Brucker. 2018. *Stateful Protocol Composition (Extended Version)*. Technical Report. DTU Compute. Technical Report-2018-03. https://people.compute.dtu.dk/samo/.
[33] Ralf Küsters and Max Tuengerthal. 2011. Composition Theorems Without Pre-established Session Identifiers. In *CCS*. ACM, New York, NY, USA, 41–50.
[34] Ralf Küsters, Max Tuengerthal, and Daniel Rausch. 2020. The IITM Model: A Simple and Expressive Model for Universal Composability. *J. Cryptol.* 33, 4 (2020), 1461–1584.
[35] Gavin Lowe. 1997. A Hierarchy of Authentication Specifications. In *CSFW*. IEEE, 31–44.
[36] Robin Milner. 1989. *Communication and concurrency*. Prentice Hall, Saddle River, NJ, USA.
[37] Sebastian Mödersheim and Georgios Katsoris. 2014. A Sound Abstraction of the Parsing Problem. In *CSF*. IEEE, 259–273.
[38] S. Mödersheim and L. Viganò. 2009. Secure pseudonymous channels. In *ESORICS (LNCS, Vol. 5789)*. Springer, Heidelberg, Germany, 337–354.
[39] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. LNCS, Vol. 2283. Springer, Heidelberg, Germany.
[40] David J. Otway and Owen Rees. 1987. Efficient and Timely Mutual Authentication. *ACM SIGOPS Oper. Syst. Rev.* 21, 1 (1987), 8–10.