Protocol Dialects as Formal Patterns: A Composable Theory of Lingos
Technical report

Víctor García[1], Santiago Escobar[1], Catherine Meadows[2], and Jose Meseguer[3]

[1]Universitat Politècnica de València, Spain
[2]Retired
[3]University of Illinois at Urbana-Champaign, USA

May 1, 2025

Abstract

Protocol *dialects* are methods for modifying protocols that provide light-weight security, especially against easy attacks that can lead to more serious ones. A *lingo* is a dialect's key security component by making attackers unable to "speak" the lingo. A lingo's "talk" changes all the time, becoming a moving target for attackers. We present several kinds of lingo transformations and compositions to generate stronger lingos from simpler ones, thus making dialects more secure.

1 Introduction

Protocol dialects are methods for modifying protocols in order to provide a light-weight layer of security, especially against relatively easy attacks that could potentially be leveraged into more serious ones. The scenario is usually that of a network of mutually trusting principals, e.g. an enterprise network, that needs to defend itself from outside attackers. The most effective approach is to prevent outside parties from even initiating a communication with group members, e.g. by requiring messages to be modified in some way unpredictable by the attacker.[1]

Protocol dialects are intended to be used as a first line of defense, not as a replacement for traditional authentication protocols. In fact, they are primarily concerned with protecting against weak attackers that are trying to leverage off

---

[1]By "message" we mean a unit of information transferred by the protocol, and can mean anything from a packet to a message in the traditional sense.

protocol vulnerabilities. We can make use of this restricted scope, as well as the assumption that all parties in an enclave trust each other, to use an attacker model based on the on-path attacker model [1, 2]: First, the attacker can read messages, but cannot destroy or replace them. Second, principals trust each other and share a common seed of a pseudo-random number generator that they can use to generate and share common secret parameters. Because of the simplified threat model and trust assumption, dialects can be designed to be simpler than full-scale authentication protocols. Thus they can be expected to be prone to fewer implementation and configuration errors than full-scale protocols. With this in mind, dialects can be used as a separate but very thin layer in the protocol stack, useful as a defense against attacks in case the main authentication protocol is misconfigured or implemented incorrectly. Considering the dialect as a separate layer makes it easier to keep it simple, because it makes it possible to avoid dependencies on the layers it is communicating with. In our work, we consider dialects that sit right above the transport layer in the TCP/IP model, while the protocol it is modifying sits just above the dialect in the application layer. However, other locations are possible for the protocol being modified.

In [2] we introduced a system for specifying protocol dialects as *formal patterns* that specify solutions to frequently occurring distributed system problems that are generic, executable, and come with strong formal guarantees. Moreover, formal patterns can be further composed and instantiated to meet specific needs. The dialect pattern is divided into two components: the *lingo*, which describes the actual message transformation, and the *dialect*, which is responsible for managing lingos and interfacing with the underlying protocol. Methods are also supplied for composing lingos, which allows for the creation of new lingos and dialects from previous ones.

We build on our work in [2] by applying it to the design of dialects. However, unlike [2] we concentrate on the design of lingos instead of the design of dialects as a whole. In order to facilitate this, instead of composing whole dialects as in [2], we develop new formal patterns to create and compose new lingos from simpler ones, which we find to be a more flexible approach that allows better concentration on the problem at hand. These patterns are *lingo transformations*, which map one or more lingos enjoying some properties to a new lingos enjoying some new properties. In particular, we concentrate on lingos that are based on constantly changing secret parameters, which are generated by a pseudorandom number generator shared throughout, itself parametric on a single shared secret, an approach to lingo design introduced by Gogeneni et al. in [1]. Our specifications are written in the Maude rewriting logic language, which allows for executable specifications and model checking rooted in logical theories, including both probabilistic and statistical techniques. Indeed, all examples shown in this paper are executable, and we provide executable Maude specifications in `https://github.com/v1ct0r-byte/Protocol-Dialects-as-Formal-Patterns`.

The paper is organized as follows. Section 2 introduces some preliminaries needed to read the paper. In Section 3 we begin by concentrating on several types of lingo transformations that can be used to guarantee specific properties

of lingos such as non-malleability, authentication, and *f-checkability*, the checkability of correctness by the dialect instead of the protocol. In Section 4 we deal with a more general type of lingo transformation: *compositions*. These include functional composition (composing lingos as functions) and horizontal composition (choosing between different lingos probabilistically). We define lingo composition and introduce several examples to demonstrate its usefulness, including showing how composition can be used to produce a lingo that is stronger than either of its components. In Section 5 we introduce a formal pattern for the dialects that use the lingos, completing the picture. Section 6 provides some open questions and concludes the paper.

1.1 Related Work

Probably, the first work on dialects is that of Sjoholmsierchio et al. in [3]. In this work, the dialect is proposed as a variation of an open-source protocol to introduce new security measures or remove unused code, and is applied to the OpenFlow protocol, introducing, among other things a defense against ciphersuite downgrade attacks on TLS 1.2. This variation was independent of TLS, and was achieved by adding proxies, thus allowing for modification without touching the rest of the system and foreseeing the use of dialects as thin layers. In later work by Lukaszewski and Xie [4] a layer 4.5 in the TCP/IP model was proposed for dialects. In our own model we also make use of proxies for both sending and receiving parties, but stop short of proposing an additional official layer.

The threat model we use in this paper and in [2] is similar to that used by Goginieni et al. in [1] and Ren et. al [5]: an on-path attacker who is not able to corrupt any members of the enclave, as discussed earlier. We have also followed [5]'s suggestion to view dialects as composable protocol transformations. In this paper, we simplify this approach by reducing dialect transformations to more basic lingo transformations, which are generic, compositional methods for producing new lingos (and thus new dialects) from old ones.

In [1, 6, 7] Mei, Gogineni, et al. introduce an approach to dialects in which the message transformation is updated each time using a shared pseudo-random number generator. This means that the security of a transformation depends as much, and possibly more, on its unpredictability as it does on the inability of the attacker to reproduce a particular instance of a transformation. We have adopted and extended this approach. In particular, we model a transformation as a parametrized function, or *lingo*, that produces a new message using the original message and the parameter as input. Both lingo and parameter can be chosen pseudo-randomly, as described in [2].

None of the work cited above (except our own) applies formal design and evaluation techniques. However, in [8] Talcott applies formal techniques to the study of dialects running over unreliable transport, such as UDP. [8] is complementary to our work in a number of ways. First, we have been looking at dialects running over reliable transport, such as TCP. Secondly, we concentrate on a particular attacker model, the on-path attacker with no ability to compro-

mise keys or corrupt participants, and then explore ways of generating dialects and lingos that can be used to defend against this type of attacker. In [8], however, the dialects are relatively simple, but the attacker models are explored in more detail, giving the attacker a set of specific actions that it can perform. Like our model, the work in [8] is formalized in the Maude formal specification language, providing a clear potential synergy between the two models in the future.

1.2 Contributions of this paper

1. We present a theory of lingos that adds important new *properties*, *transformations* and *composition operations* to the notion of lingo introduced in [2]. The main goal of this theory is to improve the security of protocol dialects.

2. We show that these new lingo transformations and composition operations are *formal patterns* [9] that guarantee desired properties and replace and greatly simplify two earlier formal patterns for dialects in [2]. These lingo transformations provide new compositional techniques for creating new lingos and dialects.

3. We provide formal executable specifications in Maude [10] for lingos, their transformations and compositions, and dialects, available in a GitHub repository referenced in this paper.

2 Preliminaries

We summarized background on Maude and Actor systems relevant for this paper.

Maude [10, 11] is a formal specification language and system based on rewriting logic [12, 13], a logic for specifying a wide range of concurrent systems, including, e.g., Petri nets [14], process calculi [15], object-based systems [16], asynchronous hardware [17], mobile ad hoc network protocols [18], cloud-based storage systems [19], web browsers [20], concurrent programming languages [21], distributed control systems [22] and models of mammalian cell pathways [23,24]. A rewite theory is a triple $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory defining the system states as elements of the initial algebra $T_{\Sigma/E}$, and R is a collection of rewrite rules of the form $t \rightarrow t'$ that describe the local concurrent transitions of the concurrent system specified by $\mathcal{R}$.

Maude Functional (resp. System) Modules and Theories. Rewriting Logic has Equational Logic as a sublogic. In Maude this corresponds to the sublanguage of *functional modules and theories*. A Maude *functional module* is an equational theory (Σ, E) delimited with keywords `fmod` and `endfm` and having an *initial algebra semantics*, namely, $T_{\Sigma/E}$. It includes type and operator declarations for Σ (with the `sort`, resp. `op`, keywords), and equations E declared with the `eq` keyword. Users can declare their own operator's syntax with '_'

denoting argument positions, e.g. `_+_` specifies a binary infix symbol. Maude binary operators can be specified with built-in equational attributes such as associativity (`assoc`), commutativity (`comm`) and identity (`id:`). Likewise, a Maude *functional theory* is an equational theory (Σ, E) delimited by keywords `fth` and `endfth` with a *loose semantics* (specifies all algebras satisfying the theory). Similarly, a Maude *system module* (resp. *system theory*) is a rewrite theory (Σ, E, R) delimited with keywords `mod` and `endm` (resp. `th` and `endth`) with an *initial* (resp. *loose*) model semantics. The rules R in a system module (resp, theory) are introduced with the keyword `rl`. See [10, 25] for further details.

Maude Parameterized Modules. Maude supports *parameterized functional modules* having a *free algebra semantics*. They are parameterized by one of more functional theories. For example, `fmod MSET{X :: TRIV} endfm` is a parameterized module with parameter theory `TRIV` having a single parameter type `Elt` and no operations or equations that maps any chosen set X of elements to the free algebra of (finite) multisets on the elements of X. The choice of an actual parameter X is made by a Maude *view* instantiating in this case `TRIV` to the chosen data type X. For example, we can choose as elements the natural numbers by a view `Nat` mapping `TRIV` to the functional module `NAT` of natural numbers and mapping the parameter type (called a sort) `Elt` to the type (sort) `Nat` in `NAT`. The semantics of the thus instantiated module `fmod MSET{Nat}` is the free algebra of multisets with elements in the natural numbers. Likewise, *parameterized system modules* have a *free model semantics*. For example, a parameterized system module `mod CHOICE{X :: TRIV} endm` that imports the parameterized functional module `fmod MSET{X :: TRIV} endfm` can have a rewrite rule for non-deterministically choosing an element in a multiset of elements. Then, the instantiation `mod CHOICE{Nat} endm` provides non-deterministic choice of an element in a multiset of natural numbers.

Actors [26] model distributed systems in which distributed objects communicate through asynchronous message passing. When an actor receives a message, it changes its state, may send new messages, and may create new actors. In [2, 27], actors have been extended to *generalized actor rewrite theories* (GARwThs). These are object-oriented rewrite theories specified in Maude as a special class of system modules that satisfy natural requirements. The distributed states of a GARwTh (terms of sort `Configuration`) are multi-sets of objects (terms of sort `Object`) and messages (terms of sort `Msg`). Multiset union is modelled by an associative and commutative operator '`_ _`' (juxtaposition), with `null` is the empty multi-set. Communication protocols are typical GARwThs.

2.1   Formal Semantics of MQTT

MQTT [28] is a lightweight, publish-subscribe, protocol popular in devices with limited resources or network bandwidth. It requires a transport protocol such as TCP/IP that can provide ordered, lossless, bi-directional connections. MQTT provides no security itself, and, although it can be run over secure transport

5

```
rl [mqtt/C/send-connect] :
    < Me : MqttClient | peer : nothing, cmdList : connect(B); CMDL, Atrs >
    =>
    < Me : MqttClient | peer : nothing, cmdList : CMDL, Atrs >
    (to B from Me : connect(B)) .

rl [mqtt/B/accept-connect] :
    < Me : MqttBroker | peers : Ps, Atrs > (to Me from O : connect(Me))
    =>
    < Me : MqttBroker | peers : insert(O, Ps), Atrs > (to O from Me : connack) .

rl [mqtt/C/recv-connack] :
    < Me : MqttClient | peer : nothing, Atrs > (to Me from O : connack)
    =>
    < Me : MqttClient | peer : O, Atrs > .
```

Figure 1: Subset of actor rules from the MQTT Maude specification, formalizing the semantics of a client connecting to a broker.

protocols such as TLS, it is known to be commonly misconfigured [29]. This makes it an attractive application for protocol dialects [1]. We use MQTT as the base protocol for the examples shown in the paper.

Example 1 *We formalize MQTT in Maude as a protocol example for Dialects. Specifically, we select to specify a subset of MQTT for illustration purposes. We give formal semantics to the following MQTT commands: connect, subscribe, unsubscribe, publish and disconnect. Furthermore, in our MQTT's symbolic model we have two classes of actors, `MqttClient` and `MqttBroker`. Class `MqttClient` defines three attributes: i) the broker it is connected to, ii) a list of commands to execute in order, and iii) a map between topic and value, to store the last value received on the subscribed topic. Class `MqttBrocker` has two attributes: i) a lists of peers and ii) a map between topics and the peers subscribed to it. With this two classes and set of commands we can build and represent environments with multiple clients and brokers, exchanging messages in a concurrent and asynchronous manner.*

As an example of the formal semantics of MQTT, we show the process of a client connecting to a broker with the actor rules of Figure 1. First rule models the behaviour of an `MqttClient` with identifier `Me` sending a `connect` message to an `MqttBroker` with identifier `B`. Second rule represents what the `MqttBroker` will do upon receiving the `connect` message, i.e. add the `MqttClient` to it's set of peers and send back to the sender an acknowledgement message. With the third rule, the `MqttClient` processes any incoming acknowledgement message by setting its `peer` attribute with the senders identifier.

The initial configuration, represented in Figure 2, has three objects: two clients (with identifiers `c1` and `c2` respectively) and one broker (with identifier `b`). Given the commands in the `cmdList` attribute of each client, what will happen is: 1) both clients will connect to the broker, 2) client `c1` will subscribe

```
< c1 : MqttClient | peer : nothing,
                    cmdList : (connect(b) ; subscribe("temp")),
                    lastRecv : empty >
< c2 : MqttClient | peer : nothing,
                    cmdList : (connect(b) ; publish("temp", "34")),
                    lastRecv : empty >
< b : MqttBroker | peers : empty, subscribers : empty >
```

Figure 2: Initial MQTT configuration for experiments.

to topic **temperature** *and, 3) client* **c2** *will publish value* **34** *on that same topic. In the end, client* **c1** *will receive a value published by client* **c2** *regarding the temperature, storing the last received value (34) in attribute* **lastRecv***. For the dialect examples in this work, we use this initial configuration (state).*

3 Lingos

A lingo is a data transformation f between two data types D_1 and D_2 with a one-sided inverse transformation g. The transformation f is *parametric* on a secret parameter value a belonging to a parameter set A. For each parameter a, data from D_1 is transformed into data of D_2, which, using the same parameter a, can be transformed back into the original data from D_1 using the one-sided inverse g. In all our applications lingos will be used to transform the payload of a message in some protocol $\mathcal{P}$ with data type D_1 of payload data. D_2 will then be the data type of transformed payloads. Such transformed payloads can then be sent, either as a single message or as a sequence of messages, to make it hard for malicious attackers to interfere with the communication of honest participants, who are the only ones knowing the current parameter $a \in A$. The point of a lingo is that when such transformed messages are received by an honest participant they can be transformed back using g (if they were broken into several packets they should first be reassembled) to get the original payload in D_1.

Briefly, the set D_1 can be thought of as analogous to the plaintext space, the set D_2 as analogous to the ciphertext space in cryptographic systems, and the parameter set A (also known as the secret parameter set) as analogous to the key space. The elements of A are generally required to be pseudorandomly generated from a key shared by enclave members so it can't be guessed in advance, and it is also generally updated with each use, so information gleaned from seeing one dialected message should not provide any help in breaking another. However, it is not necessarily required that an attacker not be able to guess a after seeing $f(d_1, a)$. The only requirement is that it is not feasible for the attacker to compute $f(d_1, a)$ without having have been told that a is the current input to $f(d_1, a)$. This is for two reasons. First the lingo is intended to provide authentication, not secrecy. Secondly, the lingo is only intended to be secure against an on-path attacker [2] that can eavesdrop on traffic but

not interfere with it. Thus, if an enclave member sends a message $f(d_1, a)$ the attacker, even if it learns a, can't remove $f(d_1, a)$ from the channel and replace it with $f(d'_1, a)$ where d'_1 is a message of the attacker's own choosing. It can send $f(d'_1, a)$ after $f(d_1, a)$ is sent, but $f(d'_1, a)$ will not be accepted if the secret parameter is changed each time the message is sent. The enforcement of the policy on the updating of parameters is the job of the dialect, which will be discussed in Section 5.

Definition 1 (Lingo) *A lingo Λ is a 5-tuple $\Lambda = (D_1, D_2, A, f, g)$, where D_1, D_2 and A are sets called, respectively, the* input, output *and* parameter *sets, f, g are functions $f : D_1 \times A \to D_2$, $g : D_2 \times A \to D_1$ such that[2] $\forall d_1 \in D_1$, $\forall a \in A$, $g(f(d_1, a), a) = d_1$. We call a lingo non-trivial iff D_1, D_2 and A are non-empty sets. In what follows, all lingos considered will be non-trivial.*

Note that a lingo $\Lambda = (D_1, D_2, A, f, g)$ is just a three-sorted (Σ, E)-algebra, with sorts[3] D_1, D_2 and A, function symbols $f : D_1 \times A \to D_2$ and $g : D_2 \times A \to D_1$, and E the single Σ-equation $g(f(d_1, a), a) = d_1$.

This means that there is a natural notion of lingo homomorphism, namely, a Σ-homomorphism. Given lingos $\Lambda = (D_1, D_2, A, f, g)$ and $\Lambda' = (D'_1, D'_2, A', f', g')$, a lingo homomorphism $h : \Lambda \to \Lambda'$ is a triple of functions $h = (h_1, h_2, h_3)$ with $h_1 : D_1 \to D'_1$, $h_2 : D_2 \to D'_2$ and $h_3 : A \to A'$, s.t. $\forall d_1 \in D_1$, $\forall d_2 \in D_2$, $\forall a \in A$,

$$h_2(f(d_1, a)) = f'(h_1(d_1), h_3(a)) \quad and \quad h_1(g(d_2, a)) = g'(h_2(d_2), h_3(a)).$$

In Maude, an equational theory (Σ, E) can be specified as a *functional theory*. Therefore, the above notion of lingo has the following natural specification in Maude:

```
fth LINGO is
    sorts D1 D2 A . op f : D1 A -> D2 . op g : D2 A -> D1 .
    var d1 : D1 . var a : A . eq g(f(d1,a),a) = d1 .
endfth
```

Since lingos are a key component of dialects [2], there is some extra information that a lingo must provide so a dialect can apply it. We easily extend the above theory to one with: i) a function called `param` for computing the corresponding value $a \in A$ from natural numbers, and ii) ingress and egress values denoting how many messages the lingo receives as input, and how many messages does it provide as output.

```
fth PMLINGO is
    protecting NAT .
```

---

[2]The equality $g(f(d_1, a), a) = d_1$ can be generalized to an equivalence $g(f(d, a), a) \equiv d$. The generalization of lingos to allow for such message equivalence is left for future work.

[3]Note the slight abuse of notation: in Σ, D_1, D_2 and A are uninterpreted sort *names*, whereas in a given lingo $\Lambda' = (D'_1, D'_2, A', f', g')$, such sorts are respectively interpreted as sets D'_1, D'_2 and A' and, likewise, the uninterpreted function *symbols* f and g in Σ are interpreted as actual *functions* f' and g' in a given lingo.

```
     including LINGO .
     op param : Nat -> A .
     ops ingressArity egressArity : -> NzNat .
endfth
```

Note the asymmetry between f and g, in the lingo definition, since we do not have an equation of the form $\forall d_2 \in D_2,\ \forall a \in A,\ f(g(d_2, a), a) = d_2$. The reason for this asymmetry is that, given $a \in A$, the set $\{f(d_1, a) \mid d_1 \in D_1\}$ may be a *proper* subset of D_2. However, we show in Corollary 2 below that the equation $f(g(d_2, a), a) = d_2$ does hold for any $d_2 \in \{f(d_1, a) \mid d_1 \in D_1\}$. Other results and proofs are included in Appendix **??**. Of course, for the special case of a lingo such that $\forall a \in A,\ D_2 = \{f(d_1, a) \mid d_1 \in D_1\}$, the equation $\forall d_2 \in D_2$, $\forall a \in A,\ f(g(d_2, a), a) = d_2$ will indeed hold. In particular, the set equality $D_2 = \{f(d_1, a) \mid d_1 \in D_1\}$ holds for all $a \in A$ in the following example.

Example 2 (The XOR$\{n\}$ Lingo) *Let $\Lambda_{xor}\{n\} = (\{0,1\}^n, \{0,1\}^n, \{0,1\}^n, \oplus, \oplus)$, with $\_ \oplus \_ : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ the bitwise exclusive or operation.*

Note that $\Lambda_{xor}\{n\}$ is parametric on $n \in \mathbb{N} \setminus \{0\}$. That is, for each choice of $n \geq 1$ we get a corresponding lingo. This is a common phenomenon: for many lingos, D_1, D_2 and A are not fixed sets, but parametrised data types, so that for each choice of their parameter we get a corresponding instance lingo.

We can specify in Maude the parametrised data type $\{0,1\}^n$ parametric on a non-zero bit-vector length n and its associated xor function. Since in Maude's built-in module NAT of (arbitrary precision) natural numbers there is a function xor that, given numbers n, m, computes the number n xor m whose associated bit-string is the bitwise exclusive or of n and m when represented as bit-strings, we can define such a parametrised module as follows:

```
fth NzNATn is protecting NAT . op n : -> NzNat [pconst] . endfth

fmod BIT-VEC{N :: NzNATn} is protecting NAT .
    sorts BitVec{N} BitStr{N} . subsort BitVec{N} < BitStr{N} .

    op [_] : Nat -> BitStr{N} [ctor] . vars N M : Nat .
    cmb [N] : BitVec{N} if N < (2 ^ N$n) .

    op _xor_ : BitStr{N} BitStr{N} -> BitStr{N} [assoc comm] .
    op _xor_ : BitVec{N} BitVec{N} -> BitVec{N} [assoc comm] .
    eq [N] xor [M] = [N xor M] .
    ...
endfm

view 8 from NzNATn to NAT is op n to term 8 . endv
```

We can easily define its instantiation for bytes and evaluate some expressions.

```
fmod BYTE is
 protecting BIT-VEC{8} * (sort BitVec{8} to Byte) .
endfm
```

9

```
========================================
reduce in BYTE : [3] xor [5] .
result Byte: [6]
========================================
reduce in BYTE : [1000] xor [1] .
result BitStr{8}: [1001]
```

Note that, exploiting the bijective correspondence between the decimal notation and the bit-string notation, the parametric subsort `BitVec{N}` of `BitStr{N}` is defined as the bitvectors of dimension n by the conditional membership axiom `[i] : BitVec\{N\} if i < (2 ^ N$n)`. For convenience, when $n = 8$ we have renamed the sort `BitVec{8}` to `Byte`.

Having defined `BIT-VEC{N :: NzNATn}`, we can now define the parametrised lingo $\Lambda_{xor} = (\{0,1\}^n, \{0,1\}^n, \{0,1\}^n, \oplus, \oplus)$ in Maude as follows:

```
fmod IDLINGO{L :: LINGO} is
    op f' : L$D1 L$A -> L$D2 . op g' : L$D2 L$A -> L$D1 .
    var d1 : L$D1 .  var d2 : L$D2 .  var a : L$A .
    eq f'(d1,a) = f(d1,a) . eq g'(d2,a) = g(d2,a) .
endfm

view xorl{N :: NZNATn} from LINGO to BIT-VEC{N} is
    sort D1 to BitVec{N} . sort D2 to BitVec{N} . sort A to BitVec{N} .
    op f to _xor_ . op g to _xor_ .
endv

fmod XOR-L{N :: NZNATn} is protecting IDLINGO{xorl{N}} . endfm
```

As explained in Section 1, Lingos are transformations used by Dialects to transform the messages from, and to, the underlying object. The parametrized module `IDLINGO` is our approach, for illustration purposes, to show the execution of a lingo's `f` and `g` functions. For more information on Dialects and an example we refer the reader to Section 5.

The meaning of the parametrized module `IDLINGO{L :: LINGO}` is that of the *identical lingo* obtained by instantiating the theory `LINGO` with concrete data types `D1`, `D2` and `A`, and concrete functions `f` and `g`. Recall that such instantiations are achieved in Maude by means of a *view*. The only notational difference is that we rename `f` and `g` by `f'` and `g'`. We then instantiate the theory `LINGO` to `BIT-VEC{N}` by means of a *parametrized view* `xorl{N :: NZNATn}`. Our desired parametrized lingo $\Lambda_{xor} = (\{0,1\}^n, \{0,1\}^n, \{0,1\}^n, \oplus, \oplus)$ is then the parametrized module `XOR-LINGO{N :: NZNATn}`. We can then instantiate it to bytes as follows:

```
fmod BYTE-LINGO is
    protecting XOR-L{8} * (sort BitVec{8} to Byte) .
endfm
========================================
reduce in BYTE-L : f'([3], [5]) .
result Byte: [6]
========================================
```

```
reduce in BYTE-L : g'([3], [5]) .
result Byte: [6]
```

Example 3 (XOR-BSEQ Lingo for Bit-sequences) *Yet another variation the same theme is to define the* $\Lambda_{xor.BSeq}$ *lingo, whose elements are* bit sequences *of arbitrary length. This can be easily done in Maude as follows:*

```
view xor-seq-l from LINGO to NAT is
    sort D1 to Nat . sort D2 to Nat . sort A to Nat .
    op f to _xor_ . op g to _xor_ .
endv
```

And instantiate `IDLINGO` to evaluate some expressions.

```
fmod XOR-BSEQ-LINGO is
    protecting IDLINGO{xorl-Nat} .
endfm
========================================
reduce in XOR-BSEQ-LINGO : f'(3, 5) .
result NzNat: 6
========================================
reduce in XOR-BSEQ-LINGO : g'(6, 5) .
result NzNat: 3
```

Note that `BIT-VEC{N :: NZNATn}` is not the only possible parametrised data type on which a lingo based on the *xor* operation could be based. A different parametrised data type can have as elements the *finite subsets* of a parametrised data type of finite sets, whose elements belong to a parameter set D, and where the *xor* operation is interpreted as the *symmetric difference* of two finite subsets of D. A finite subset, say, $\{a, b, c\} \subseteq D$, with $a, b, d \in D$ different elements, can be represented as the expression *a xor b xor c*. Here is the parametrised module defining such a parametric data type and its instantiation for D the set of quoted identifiers (a built-in module `QID` in Maude with sort `Qid`).

```
fmod XOR-SET{D :: TRIV} is
    sort Set . subsort D$Elt < Set .
    op zero : -> Set . op _xor_ : Set Set -> Set [ctor assoc comm] .
    var S : Set . eq S xor S = zero . eq zero xor S = S .
endfm

view Qid from TRIV to QID is sort Elt to Qid . endv
```

We can use both the functional module and the view to evaluate some expressions.

```
fmod XOR-QID is
    protecting XOR-SET{Qid} .
endfm
========================================
reduce in XOR-QID : ('a xor 'b xor 'c xor 'd) xor
```

```
('c xor 'd xor 'e xor 'f) .
result Set: 'a xor 'b xor 'e xor 'f
========================================
reduce in XOR-QID : 'b xor 'c xor 'b xor 'b xor 'd .
result Set: 'b xor 'c xor 'd
```

Note the isomorphism between vector and the finite set representations: If $|D| = n$, then the power set $\wp(D)$ with symmetric difference is isomorphic to the function set $[D \to \{0,1\}]$ with pointwise *xor* of predicates $p, q \in [D \to \{0,1\}]$, i.e., $p \; xor \; q = \lambda d \in D. \; p(d) \; xor \; q(d)$, which is itself isomorphic to $\{0,1\}^n$ with pointwise *xor*.

To obtain the corresponding parametrised *xor* lingo in its set representation we proceed in a way entirely analogous to our Maude specification of $\Lambda_{xor}\{n\}$:

```
view xorlingo{D :: TRIV} from LINGO to XOR-SET{D} is
    sort D1 to Set . sort D2 to Set . sort A to Set .
    op f to _xor_ . op g to _xor_ .
endv
```

And instantiate `IDLINGO` to evaluate some expressions.

```
fmod LINGO-XOR{D :: TRIV} is
    protecting IDLINGO{xorlingo{D}} .
endfm
fmod QID-LINGO is
    protecting LINGO-XOR{Qid} .
endfm
========================================
reduce in QID-LINGO :
f'('a xor 'b xor 'c xor 'd, 'c xor 'd xor 'e xor 'f) .
result Set: 'a xor 'b xor 'e xor 'f
========================================
reduce in QID-LINGO :
g'('a xor 'b xor 'c xor 'd, 'c xor 'd xor 'e xor 'f) .
result Set: 'a xor 'b xor 'e xor 'f
```

Example 4 (Divide and Check (D&C) Lingo) $\Lambda_{D\&C} = (\mathbb{N}, \mathbb{N} \times \mathbb{N}, \mathbb{N}, f, g)$, *given* $\forall n, a, x, y \in \mathbb{N}$ *then:*

- $f(n, a) = (quot(n + (a + 2), a + 2), rem(n + (a + 2), a + 2))$

- $g((x, y), a) = (x \cdot (a + 2)) + y - (a + 2)$

where quot and rem denote the quotient and remainder functions on naturals.

The idea of the $\Lambda_{D\&C}$ lingo is quite simple. Given a parameter $a \in \mathbb{N}$, an input number n is transformed into a pair of numbers (x, y): the quotient x of $n + a + 2$ by $a + 2$ (this makes sure that $a + 2 \geq 2$), and the remainder y of $n + a + 2$ by $a + 2$. The meaning of "divide" is obvious. The meaning of "check" reflects the fact that a receiver of a pair (x, y) who knows a can check

$x = quot((x \cdot (a+2)) + y, a+2))$ and $y = rem((x \cdot (a+2)) + y, a+2))$, giving some assurance that the pair (x, y) was obtained from $n = (x \cdot (a+2)) + y - (a+2)$ and has not been tampered with. This is an example of an *f-checkable* lingo (see Section 3.2), whereas no such check is possible for any of the isomorphic versions of the Λ_{xor} lingo (see again Section 3.2). The lingo $\Lambda_{D\&C}$ can be formally specified in Maude as follows:

```
fmod NAT-PAIR is protecting NAT .
    sort NatPair .
    op [_,_] : Nat Nat -> NatPair [ctor] . ops p1 p2 : NatPair -> Nat .
    vars n m : Nat . eq p1([n,m]) = n . eq p2([n,m]) = m .
endfm

view D&C-ling from LINGO to NAT-PAIR is
    sort D1 to Nat . sort D2 to NatPair . sort A to Nat .
    var n : D1 . var P : D2 . var a : A .
    op f(n,a) to term [(n+(a+2)) quo (a+2), (n+(a+2)) rem (a+2)] .
    op g(P,a) to term sd(((p1(P) * (a+2)) + p2(P)), (a+2))   .
endv
```

We instantiate `IDLINGO` to evaluate some expressions.

```
fmod D&C-LINGO is
    protecting IDLINGO{D&C-ling} .
endfm
==========================================
reduce in D&C-LINGO : f'(13, 3) .
result NatPair: [3, 3]
==========================================
reduce in D&C-LINGO : g'([3, 3], 3) .
result NzNat: 13
==========================================
reduce in D&C-LINGO : g'(f'(13, 3), 3) .
result NzNat: 13
```

3.1   Basic Properties of Lingos

This section proves several useful properties of lingos.

Lemma 1 Let $\Lambda = (D_1, D_2, A, f, g)$ be a lingo. Then, $\forall d_1, d_1' \in D_1$, $\forall a \in A$ $f(d_1, a) = f(d_1', a) \Rightarrow d_1 = d_1'$.

Proof: Applying g on both sides of the condition, by the definition of lingo we get, $d_1 = g(f(d_1, a), a) = g(f(d_1', a), a) = d_1'$. $\square$

Lemma 2 Let $\Lambda = (D_1, D_2, A, f, g)$ be a lingo. Then, $\forall d_1 \in D_1$, $\forall d_2 \in D_2$, $\forall a \in A$, $d_2 = f(d_1, a) \Rightarrow d_1 = g(d_2, a)$.

Proof: Applying g to both sides of the condition, by the definition of lingo we get, $g(d_2, a) = g(f(d_1, a), a) = d_1$. $\square$.

Corollary 1 *Let* $\Lambda = (D_1, D_2, A, f, g)$ *be a lingo. Then,* $\forall d_1 \in D_1$, $\forall d_2 \in D_2$, $\forall a \in A$, $d_2 = f(d_1, a) \Rightarrow d_2 = f(g(d_2, a), a)$.

Proof: By Lemma 2, $d_1 = g(d_2, a)$. Thus, $d_2 = f(d_1, a) = f(g(d_2, a), a)$. $\square$

Corollary 2 *Let* $\Lambda = (D_1, D_2, A, f, g)$ *be a lingo. Then,* $\forall d_2 \in D_2$, $\forall a \in A$, $((\exists d_1 \in D_1, \ f(d_1, a) = d_2) \Rightarrow d_2 = f(g(d_2, a), a))$.

Proof: By Lemma 2 we have, $\forall d_2 \in D_2, \forall a \in A, (\forall d_1 \in D_1, d_2 = f(d_1, a) \Rightarrow d_2 = f(g(d_2, a), a))$. which means $\forall d_2 \in D_2, \forall a \in A, (\forall d_1 \in D_1, \ \neg(d_2 = f(d_1, a)) \vee d_2 = f(g(d_2, a), a))$ which by d_1 not a free variable in $d_2 = f(g(d_2, a), a)$ is equivalent to $\forall d_2 \in D_2, \forall a \in A, ((\forall d_1 \in D_1, \ \neg(d_2 = f(d_1, a))) \vee d_2 = f(g(d_2, a), a))$ which by duality of quantifiers is equivalent to $\forall d_2 \in D_2, \forall a \in A, (\neg(\exists d_1 \in D_1, \ d_2 = f(d_1, a)) \vee d_2 = f(g(d_2, a), a))$ which means $\forall d_2 \in D_2, \forall a \in A, ((\exists d_1 \in D_1, \ d_2 = f(d_1, a)) \rightarrow d_2 = f(g(d_2, a), a))$ as desired. $\square$

Corollary 3 below shows that users of a lingo can check if a received payload $d_2 \in D_2$ is of the form $d_2 = f(d_1, a)$ for some $d_1 \in D_1$ and $a \in A$ the current parameter, i.e., if it was generated from some actual $d_1 \in D_1$ as $d_2 = f(d_1, a)$. We call a $d_2 \in D_2$ such that $d_2 = f(d_1, a)$ for some $d_1 \in D_1$ *compliant* with parameter $a \in A$. As shown in Corollary 3 compliance with a can be checked by checking the equality $f(g(d_2, a), a) = d_2$ for any given $d_2 \in D_2$ and $a \in A$.

Corollary 3 *Let* $\Lambda = (D_1, D_2, A, f, g)$ *be a lingo. Then,* $\forall d_2 \in D_2$, $\forall a \in A$, $\exists d_1, (f(d_1, a) = d_2) \Leftrightarrow d_2 = f(g(d_2, a), a)$.

Proof: The $(\Rightarrow)$ implication follows from Corollary 2. The $(\Leftarrow)$ implication follows by choosing $d_1 = g(d_2, a)$. $\square$

3.2   f-Checkable Lingos

In this section, we discuss lingos where validity of its generated payloads can be checked by the dialect. We develop a lingo transformation that substantially reduces the probability an attacker successfully forging a message.

Definition 2 (f-Checkable Lingo) *A Lingo* $\Lambda = (D_1, D_2, A, f, g)$ *is called f-checkable iff* $\forall a \in A \ \exists d_2 \in D_2$ *s.t.* $\nexists d_1 \in D_1$ *s.t.* $d_2 = f(d_1, a)$. *The 'f' in f-Checkable Lingo is an uninterpreted function symbol, but once an interpretation is given (i.e., through a view), f becomes interpreted as the function specified by the interpretation.*

As just explained above, Λ_{xor} is not f-checkable. More generally, call a lingo $\Lambda = (D_1, D_2, A, f, g)$ *symmetric* iff (D_2, D_1, A, g, f) is also a lingo. Obviously, Λ_{xor} is symmetric. Any symmetric lingo is *not* f-checkable, since any $d_2 \in D_2$ satisfies the equation $f(g(d_2, a), a) = d_2$.

Example 5 $\Lambda_{D\&C}$ *is f-checkable. Indeed, for each $a \in \mathbb{N}$, any (x,y) with $y \geq a+2$ cannot be of the form $(x,y) = f(n,a)$ for any $n \in \mathbb{N}$. This is a cheap, easy check. The actual check that a d_1 exists, namely, the check $(x,y) = f(g((x,y),a),a)$ provided by Corollary 3, was already explained when making explicit the meaning of "C" in $\Lambda_{D\&C}$.*

Theorem 1 *Let $\Lambda = (D_1, D_2, A, f, g)$ be a lingo, where we assume that D_1, D_2 and A are computable data-types and $|D_1| \geq 2$ Then, $\Lambda^\sharp = (D_1, D_2{}^2, A \otimes A, f^\sharp, g^\sharp)$ is an f-checkable lingo, where:*

- $A \otimes A = A \times A \setminus id_A$, *with* $id_A = \{(a,a) \in A^2 | a \in A\}$

- $f^\sharp(d_1, (a,a')) = (f(d_1, a), f(d_1, a'))$

- $g^\sharp((d_2, d_2'), (a,a')) = g(d_2, a)$

Proof: We first prove that $\Lambda^\sharp = (D_1, D_2{}^2, A \otimes A, f^\sharp, g^\sharp, comp^\sharp)$ satisfies the lingo equation $g(f(d_1, a), a) = d_1$. Indeed,

$$g^\sharp(f^\sharp(d_1, (a,a')), (a,a')) = g^\sharp((f(d_1,a), f(d_1,a')), (a,a')) = g((f(d_1,a),a) = d_1$$

as desired.

To prove that, in addition, $\Lambda^\sharp = (D_1, D_2{}^2, A \otimes A, f^\sharp, g^\sharp)$ is an f-checkable lingo, note that, since $|D_1| \geq 2$, for each $(a,a') \in A \otimes A$ we can choose $d_2^\sharp = (f(d_1,a), f(d_1', a'))$ with $d_1 \neq d_1'$. Then, there is no $d_1'' \in D_1$ such that $d_2^\sharp = f^\sharp(d_1'', (a,a'))$, since this would force $(f(d_1'', a), f(d_1'', a')) = (f(d_1,a), f(d_1', a'))$, which by Lemma 1 would force $d_1'' = d_1 = d_1'$, which is impossible. $\square$

The lingo transformation $\Lambda \mapsto \Lambda^\sharp$ has a simple specification in Maude as the following parametrised module (note that $A \otimes A$ is here denoted by `APair#`):

```
fmod LINGO#{L :: PMLINGO} is protecting INITIAL-EQUALITY-PREDICATE .
    sorts APair# APair D2Pair . subsort APair# < APair .

    op a[_,_] : L$A L$A -> APair [ctor] .
    op d2[_,_] : L$D2 L$D2 -> D2Pair [ctor] .

    vars a a' : L$A .  var d1 : L$D1 . vars d2 d2' : L$D2 .
    cmb a[a,a'] : APair# if a .=. a' = false .

    op f# : L$D1 APair# -> D2Pair .
    eq f#(d1,a[a,a']) = < f(d1,a),f(d1,a') > .
    op g# : D2Pair APair# -> L$D1 .
    eq g#(d2[d2,d2'],a[a,a']) = g(d2,a) .
    ...
endfm
```

Example 6 (f-checkable transformed version of $\Lambda_{xor}\{n\}$) *The lingo transformation $\Lambda \mapsto \Lambda^\sharp$ maps $\Lambda_{xor}\{n\}$ in Example 2 to the f-checkable lingo $\Lambda_{xor}^\sharp\{n\} = (\{0,1\}^n, \{0,1\}^n \times \{0,1\}^n, \{0,1\}^n \otimes \{0,1\}^n, \oplus^\sharp, \oplus^\sharp)$.*

The lingo $\Lambda_{xor}^{\sharp}\{n\}$ and it instance for bytes $\Lambda_{xor}^{\sharp}\{8\}$ are both obtained by instantiating the parametrised module `LINGO#{L :: PMLINGO}` as follows:

```
fmod XOR-LINGO#{N :: NzNATn} is protecting LINGO#{xorpml{N}} . endfm

fmod BYTE-LINGO# is
    protecting XOR-LINGO#{8} * (sort BitVec{8} to Byte) .
endfm
```

And evaluate some expressions.

```
reduce in BYTE-LINGO# : f#([3], [[5], [7]]) .
result D2Pair: d2[[6], [4]]
=========================================
reduce in BYTE-LINGO# :
g#(f#([3], [[5], [7]]), [[5], [7]]) .
result Byte: [3]
```

Suppose the probability of an attacker's successfully forging a message in the original lingo is P_F. The probability of the attacker passing the f-checkability test in the transformed lingo is $1/2^n$. Thus the probability of the attacker's successfully forging a message is the transformed lingo is $P_F/2^n$.

3.3   Malleable Lingos

A cryptographic function is called *malleable* if an intruder, *without knowing the secret key*, can use an existing encrypted message to generate another message also encrypted with the same secret key. In a similar way, if a lingo Λ is what below we call *malleable*, an intruder can disrupt the communication between an honest sender *Alice* and an honest receiver *Bob* in a protocol $\mathcal{P}$ whose messages are modified by means of a lingo Λ^4 by producing a message *compliant* with a secret parameter $a \in A$ of Λ supposedly sent from *Alice* to *Bob* (who share the secret parameter a), but actually sent by the intruder.

To see how malleability can become an issue for lingos, consider a variant of the xor lingo in which the same random bit-string is applied to two messages instead of one. Thus the attacker will see $d_1 \oplus a$, then $d_1' \oplus a$. Although the attacker may be able to find a after seeing both messages, it will not be able to replace either one with a $d_1'' \oplus a$ of its own choosing, since it can't interfere with messages in flight. But suppose that it has a good idea that the first byte of the first message will be y. After it has seen $d_1 \oplus a$, it can then send $y; 0^n \oplus z; 0^n \oplus d_1 \oplus a$, thus sending a message identical to d_1 except that its first byte is z. Note that the attacker can do this *without* knowing a. Thus it is not the fact that a may derivable from the two messages that causes a security vulnerability, but the fact that exclusive-or is malleable. Here is the definition.

---

[4] I.e., honest participants that use a *dialect* $\mathcal{D}_\Lambda(\mathcal{P})$ based on $\mathcal{P}$ and Λ to modify their messages (see §5).

Definition 3 (Malleable Lingo) *Let $\Lambda = (D_1, D_2, A, f, g)$ be a lingo which has been formally specified as a functional module in Maude by an equational theory $(\Sigma_\Lambda, E_\Lambda)$ enjoying the required executability conditions, so that its data types and functions are all computable. Λ is called* malleable *iff there exists a Σ_Λ-term $t(x, y)$ of sort D_2 with free variables x, y of respective sorts D_2 and A and a computable function $r : A \to A$ such that $\forall d_1 \in D_1, \ \forall a, a' \in A,$*

1. *$f(d_1, a) \neq t(f(d_1, a), r(a'))$, where, by convention, $t(f(d_1, a), r(a'))$ abbreviates the substitution instance $t\{x \mapsto f(d_1, a), y \mapsto r(a')\}$*

2. *$\exists d_1' \in D_1$ such that $t(f(d_1, a), r(a')) = f(d_1', a)$.*

Note that, by (2) and the fact that Λ is a lingo, $d_1' = g(t(f(d_1, a), r(a')), a)$.

That is, if $a \in A$ is the current secret parameter that an honest sender *Alice* is using to send d_1 (modified as $f(d_1, a)$) to honest receiver *Bob*, $t(x, y)$ is a *recipe* that an intruder can use to replace $f(d_1, a)$ by $t(f(d_1, a), r(a'))$ to generate a message compliant with the secret parameter a currently used by *Bob* to accept a message modified by *Alice* by means of the secret parameter a. In this way the intruder can spoof *Bob* into believing that $t(f(d_1, a), r(a'))$ came from *Alice*. The function r ensures that the randomly chosen parameter $r(a')$ meets condition (1), since otherwise there would be no spoofing. Let us see two examples of malleable lingos.

Example 7 *The lingo $\Lambda_{xor}\{n\}$ from Example 2 is malleable. The recipe $t(x, y)$ is $x \oplus y$ and $r_\oplus$ is the function $r_\oplus = \lambda y \in \{0, 1\}^n. \ \mathbf{if} \ y \ . = . \ \vec{0} \ \mathbf{then} \ \vec{1} \ \mathbf{else} \ y \ \mathbf{fi} \in \{0, 1\}^n$, where $. = .$ denotes the equality predicate on bitvectors. Therefore, $t(f(d_1, a), r(a')) = d_1 \oplus a \oplus r(a')$, which meets condition (1) because $r(a') \neq \vec{0}$ and obviously meets condition (2), since $t(f(d_1, a), r(a')) = f(d_1 \oplus r(a'), a)$.*

An f-checkable lingo provides a first line of defense against an intruder trying to generate a payload compliant with a secret parameter a. However, the example below shows that an f-checkable lingo can be malleable.

Example 8 *The lingo $\Lambda_{xor}^\sharp\{n\}$ from Example 6 with $n \geq 2$ is comp-malleable with recipe[5] $t(x, y) = (p_1(x) \oplus p_1(y), p_2(x) \oplus p_1(y))$ and function r defined by:*

$$\lambda y \in \{0, 1\}^n \otimes \{0, 1\}^n. \ \mathbf{if} \ \neg(p_1(y). = .\vec{0}) \ \mathbf{then} \ y \ \mathbf{else}$$

$$\mathbf{if} \ \neg(p_2(y) \ . = . \ \vec{1}) \ \mathbf{then} \ (\vec{1}, p_2(y)) \ \mathbf{else} \ (0\,\vec{1}_{n-1}, p_2(y)) \ \mathbf{fi}$$

$$\mathbf{fi} \in \{0, 1\}^n \otimes \{0, 1\}^n,$$

where $0\,\vec{1}_{n-1}$ denotes the n-bitvector starting with 0 and continued with $n-1$ 1's. Note that the key property of $r(y)$ is that $p_1(r(y)) \neq \vec{0}$. Therefore, $t(f^\sharp(d_1, (a, a')), r((a'', a'''))) = (d_1 \oplus a \oplus p_1(r((a'', a'''))), d_1 \oplus a' \oplus p_1(r((a'', a'''))))$ is in fact $f^\sharp(d_1 \oplus p_1(r((a'', a'''))), (a, a'))$, which is different from $f^\sharp(d_1, (a, a')) = (d_1 \oplus a, d_1 \oplus a')$, thus meeting conditions (1) and (2).

---

[5] We assume without loss of generality that $\Sigma_{\Lambda_{xor}^\sharp}$ contains projection functions $p_1, p_2 : D_2 \, D_2 \to D_2$ and $p_1, p_2 : A \, A \to A$ equationally specified by the usual projection equations.

Examples 7–8 suggest the following generalization:

Theorem 2 *Let $\Lambda = (D_1, D_2, A, f, g)$ be a lingo where all data types and all functions are computable, and such that for all $d_1 \in D_1$, $a, a', a'' \in A$, (i) $f(d_1, a) \in D_1$, (ii) there is a computable subset $A_0 \subseteq A$ with at least two different elements a_0, a_1 and such that $a'' \in A_0 \Rightarrow f(d_1, a'') \neq d_1$, and (iii) $f(f(d_1, a), a') = f(f(d_1, a'), a)$. Then both Λ and $\Lambda^\sharp$ are malleable.*

Proof: To see that Λ is malleable, define $r = \lambda a \in A.$ **if** $a \in A_0$ **then** a **else** a_0 **fi**. Then, recipe $f(x, y)$ works, since by (ii), we have $f(f(d_1, a), r(a')) \neq f(d_1, a)$, and both $f(f(d_1, a), r(a'))$ and $f(f(d_1, r(a')), a), a)$ are compliant with a.

To see that $\Lambda^\sharp$ is malleable define r as the function:

$$r = \lambda y \in A \otimes A. \ \textbf{if} \ p_1(y) \in A_0 \ \textbf{then} \ y \ \textbf{else}$$

$$\textbf{if} \ p_2(y) \ .=. \ a_0 \ \textbf{then} \ (a_1, p_2(y)) \ \textbf{else} \ (a_0, p_2(y)) \ \textbf{fi}$$

$$\textbf{fi} \in A \otimes A.$$

r's key property is of course that $\forall y \in A \otimes A$, $p_1(r(y)) \in A_0$. Then, recipe $t(x, y) = [f(p_1(x), p_1(y)), f(p_2(x), p_1(y))]$ works because

$$t(f^\sharp(d_1, (a, a')), r((a'', a'''))) =$$

$$[f(f(d_1, a), p_1(r((a'', a''')))), f(f(d_1, a'), p_1(r((a'', a'''))))] =$$
$$[f(f(d_1, p_1(r((a'', a''')))), a), f(f(d_1, p_1(r((a'', a''')))), a')]$$

so that, by the definition of r and (ii) we have,

$$t(f^\sharp(d_1, (a, a')), r((a'', a'''))) \neq f^\sharp(d_1, (a, a'), r((a'', a''')))$$

and we of course have $(f(f(d_1, p_1(r((a'', a''')))), a),$ $f(f(d_1, p_1(r((a'', a''')))), a'))$ compliant with (a, a'), as desired. $\square$

Being malleable is an undesirable property. One would like to transform a possibly malleable lingo Λ into a non-malleable lingo Λ'. Developing general methods for proving that a lingo is non-malleable is a topic for future research. However, in Section 4.5 we discuss several methods that we conjecture could support lingo transformations $\Lambda \mapsto \Lambda'$ yielding non-malleable lingos Λ'.

3.4 Authenticating Lingos

Here we extend our threat model to one in which it is possible that some enclave members are dishonest and may try to impersonate other enclave members. In this case we assume each pair of principals that communicate with each other shares a unique secret. (When a client-server architecture is used, this means that the number of secrets is only equal to the number of clients.) We show how a lingo whose secret parameter a is updated every time it is used can be turned into a lightweight *authenticating lingo* below.

The key idea about an authenticating lingo is that, if Bob receives a transformed payload $f(d_1, a)$ supposedly from Alice, he has a way to check that it was sent by Alice because he can extract from $f(d_1, a)$ some data that was produced using secret information that only Alice and Bob share. In this case it will be the result of computing a one-way hash function *hash* over their identities and their current shared parameter a. To make this possible, we need to make explicit two data types involved in the communication. First, honest participants in the communication protocol using lingo Λ typically have corresponding *object identifiers* that uniquely identify such participants and belong to a finite data type *Oid*. Second, the result of computing the *hash* belongs to another finite data type H. Here is the definition:

Definition 4 *An* authenticating lingo *is a tuple* $((D_1, D_2, A, f, g), Oid, H, param,$ $hash, code)$*such that:*

1. (D_1, D_2, A, f, g) *is a lingo;*

2. *Oid and H are finite data types of, respectively, object identifiers and outputs of hash.*

3. *param is a function* $param : \mathbb{N} \times Oid \otimes Oid \to A$*;*

4. *hash is a function* $hash : \mathbb{N} \times Oid \otimes Oid \to H$*;*[6] *and*

5. *code is a function* $code : D_2 \times A \to H$

such that $\forall d_1 \in D1$*,* $\forall n \in \mathbb{N}$*,* $\forall (A, B) \in Oid \otimes Oid$*,*

$$code(f(d_1, param(n, (A, B))), param(n, (A, B))) = hash(n, (A, B)).$$

Let us illustrate this notion by means of a transformation $\Lambda \mapsto \Lambda^\alpha$ that generates an authenticating lingo Λ^α from a lingo Λ. For concreteness, assume that Λ is of the form $\Lambda = (\mathbb{N}_{<2^n}, \mathbb{N}_{<2^m}, A_0, f_0, g_0)$ with A_0 a finite set, and that there is a function $param_0 : \mathbb{N}_{\leq 2^k} \to A_0$ assigning to each random number $n \in \mathbb{N}_{<2^k}$ a corresponding parameter in A_0. This reflects the fact that, in practice, two honest participants will share a common parameter $a_0 \in A_0$ by sharing a common secret random number n and using $param_0$ to generate a_0. Assuming a finite data type *Oid* of honest participants, choose a function $hash : \mathbb{N}_{<2^k} \times Oid \otimes Oid \to \mathbb{N}_{<2^j}$, with j and k big enough for *hash* to be computationally infeasible to invert. Choose a function $\iota : \mathbb{N}_{<2^k} \to Invo(m + j)$ mapping each random number n to an *involution*[7] permutation σ on $m + j$ elements. By abuse of notation, if $\vec{b} = b_1, \dots, b_{m+j}$ is a bit sequence of length $m + j$, we let $\sigma(b_1, \dots, b_t) = b_{\sigma(1)}, \dots, b_{\sigma(m+j)}$.

Then define Λ^α as follows: $\Lambda^\alpha = ((\mathbb{N}_{<2^n}, \mathbb{N}_{<2^{m+j}}, A_0 \times Invo(m + j) \times \mathbb{N}_{<2^j}, f, g),$
$Oid, \mathbb{N}_{<2^j}, param, hash, code)$ where:

---

[6]*hash* should have good properties as a hash function. This seems impossible, since H is finite but $\mathbb{N}$ is not. But this problem is only apparent. In practice the values $n \in \mathbb{N}$ used as first arguments of *hash* (and also as first arguments of *param*) will be random numbers n such that $n \in \mathbb{N}_{<2^k}$ for some fixed k.

[7]An involution is a permutation σ such that σ is its own inverse permutation, i.e., $\sigma = \sigma^{-1}$.

1. $f(d_1, (a_0, \sigma, d)) = \sigma(f_0(d_1, a_0); d)$, where we view $f_0(d_1, a_0)$ and d as bit vectors of respective lengths m and j, $f_0(d_1, a_0); d$ denotes their concatenation as bit sequences.

2. $g(\vec{b}, (a_0, \sigma, d)) = g_0(p_1(\sigma(\vec{b})), a_0)$, where by abuse of notation identify $\vec{b}$ with a number smaller than 2^{m+j}, and where p_1 (resp. p_2) projects an $m + j$ bitvector to its first m bits (resp. its last j bits).

3. $param : \mathbb{N}_{<2^k} \times Oid \otimes Oid \to A_0 \times Invo(m + j) \times \mathbb{N}_{<2^j}$ is the function $\lambda(n, (A, B)). (param_0(n), \iota(n), hash(n, (A, B)))$.

4. $code(\vec{b}, (a_0, \sigma, d)) = p_2(\sigma(\vec{b}))$.

The reader can check that Λ^α is an authenticating lingo.

4 Lingo Compositions

Lingos can be composed in various ways to obtain new lingos. Lingo compositions provide modular, automated methods of obtaining new lingos from existing ones. Such compositions are often stronger, i.e., harder to compromise by an attacker, than the lingos so composed. We define four such lingo composition operations, namely, *horizontal* and *functional* composition of lingos, *products* of lingos, and *tuplings* of lingos. We also explain how a lingo's input and output data types can be modified by means of *data-adaptors*.

4.1 Horizontal Composition of Lingos

Given a finite family of lingos $\vec{\Lambda} = \{\Lambda_i\}_{1 \leq i \leq k}$, $k \geq 2$, all sharing the same input data type D_1, their *horizontal composition*, denoted $\bigoplus \vec{\Lambda}$ is intuitively their union, i.e., in $\bigoplus \vec{\Lambda}$ the input data type D_1 remains the same, D_2 is the union of the $D_{2.i}$, $1 \leq i \leq k$, and A is the disjoint union of the A_i, $1 \leq i \leq k$. $\bigoplus \vec{\Lambda}$ is a more complex lingo that each of its component lingos Λ_i, making it harder to compromise by an attacker, because it becomes a hydra with k heads: quite unpredictably, sometimes behaves like some Λ_i and sometimes like another Λ_j. Furthermore, it has a bigger parameter space, since $|A| = \Sigma_{1 \leq i \leq k} |A_i|$.

Definition 5 (Horizontal Composition) *Let $\vec{\Lambda}$ be a finite family of lingos $\vec{\Lambda} = \{\Lambda_i\}_{1 \leq i \leq k}$, $k \geq 2$, all having the same input data type D_1, i.e., $\Lambda_i = (D_1, D_{2.i}, A_i, f_i, g_i)$, $1 \leq i \leq k$. Let $\vec{d}_0 = (d_{0.1}, \ldots, d_{0.k})$ be a choice of default $D_{2.i}$-values, $d_{0.i} \in D_{2.i}$, $1 \leq i \leq k$. Then, the horizontal composition of the lingos $\vec{\Lambda}$ with default $D_{2.i}$-values $\vec{d}_0$ is the lingo:*

$$\bigoplus_{\vec{d}_0} \vec{\Lambda} = (D_1, \bigcup_{1 \leq i \leq k} D_{2.i}, \bigcup_{1 \leq i \leq k} A_i \times \{i\}, \oplus\vec{f}, \oplus\vec{g})$$

where, for each $d_1 \in D_1$, $d_2 \in \bigcup_{1 \leq i \leq k} D_{2.i}$, and $a_i \in A_i$, $1 \leq i \leq k$,

1. $\oplus \vec{f} = \lambda(d_1, (a_i, i)).f_i(d_1, a_i)$,

2. $\oplus \vec{g} = \lambda(d_2, (a_i, i)).$ **if** $d_2 \in D_{2.i}$ **then** $g_i(d_i, a_i)$ **else**
   $g_i(d_{0.i}, a_i)$.

$\bigoplus_{\vec{d_0}} \vec{\Lambda}$ *is indeed a lingo:* $\oplus \vec{g}(\oplus \vec{f}(d_1, (a_i, i)), (a_i, i)) = g_i(f_i(d_1, a_i), a_i) = d_1$.

We have already pointed out that, in practice, two protocol participants using a lingo to first modify and then decode a payload d_1 that, say, Alice sends to Bob, agree on a secret parameter $a \in A$ by agreeing on a secret random number n, because both use a function param $: \mathbb{N} \to A$ to get $a = param(n)$. This then poses the practical problem of synthesizing a function $\oplus par\vec{a}m : \mathbb{N} \to \bigcup_{1 \le i \le k} A_i \times \{i\}$ for $\bigoplus_{\vec{d_0}} \vec{\Lambda}$ out of the family a functions $\{param_i : \mathbb{N} \to A_i\}_{1 \le i \le k}$ used in each of the lingos Λ_i. Furthermore, different lingos in the family $\vec{\Lambda}$ may have different degrees of strength against an adversary. This suggest favouring the choice of stronger lingos over that of weaker lingos in the family $\vec{\Lambda}$ to achieve a function $\oplus par\vec{a}m$ that improves the overall strength of $\bigoplus_{\vec{d_0}} \vec{\Lambda}$. This can be achieved by choosing a bias vector *$\vec{\beta} = (\beta_1, \dots, \beta_k) \in \mathbb{N}^k_{>0}$, so that, say, if lingo Λ_i is deemed to be stronger than lingo Λ_j, then the user chooses $\beta_i > \beta_j$. That is, $\vec{\beta}$ specifies a* biased dice *with $k \ge 2$ faces, so that the dice will show face j with probability $\frac{\beta_j}{\Sigma_{1 \le i \le k} \beta_i}$. Therefore, we can use a pseudo-random function $throw_{\vec{\beta}} : \mathbb{N} \to \{1, \dots, k\}$ simulating a sequence of throws of a k-face dice with bias $\vec{\beta}$ to get our desired function $\oplus par\vec{a}m$ as the function:*

$$\oplus par\vec{a}m(n) = (param_{throw_{\vec{\beta}}(n)}(n), throw_{\vec{\beta}}(n)).$$

Example 9 *To see how horizontal composition can strengthen lingos, consider the D&C lingo described in Example 4, in which $f(n, a) = (x, y)$ where $x = quot(n + (a + 2), a + 2)$ and $y = rem(n + (a + 2), a + 2)$. We note that any choice of $0 \le y < a + 2$ will pass the f-check, so the choice of $y = 0$ or 1 will always pass the f-check. To counter this, let* reverse D&C *be the lingo with $f(n, a) = (y, x)$, where y and x are computed as in D&C. The attacker's best strategy in reverse D&C is the opposite of that in D&C. Thus composing D&C and reverse D&C horizontally with a bias vector $(.5, .5)$ means that the attacker strategy of choosing the first (respectively, second) element of the output to be 0 succeeds with probability 0.5 in any particular instance, as opposed to probability 1 for D&C by itself.*

The following result is immediate.

Lemma 3 *If each Λ_i in $\vec{\Lambda} = \{\Lambda_i\}_{1 \le i \le k}$, $k \ge 2$, is f-checkable, then $\bigoplus_{\vec{d_0}} \vec{\Lambda}$ is also f-checkable.*

The horizontal composition of k lingos sharing a common input data type can be specified in Maude as a functional module parametrised by the k-`LINGO` parameter theory. We illustrate here the construction for $k = 2$, where 2-`LINGO` is abbreviated to `DILINGO`. Likewise for $k = 3$, where 3-`LINGO` is abbreviated to `TRILINGO`.

```
fth DILINGO is
    sorts D1 D21 D22 A1 A2 .

    op f1 : D1 A1 -> D21 . op f2 : D1 A2 -> D22 .
    op g1 : D21 A1 -> D1 . op g2 : D22 A2 -> D1 .
    op d0.1 : -> D21 . op d0.2 : -> D22 .

    var a1 : A1 . var a2 : A2 . var d1 : D1 .
    eq g1(f1(d1, a1), a1) = d1 .
    eq g2(f2(d1, a2), a2) = d1 .
endfth

fmod HOR-COMP{DL :: DILINGO} is
    sorts A D2 . subsorts DL$D21 DL$D22 < D2 .

    op [_]1 : DL$A1 -> A [ctor] . op [_]2 : DL$A2 -> A [ctor] .

    op +f : DL$D1 A -> D2 . op +g : D2 A -> DL$D1 .

    var d1 : DL$D1 .  var a1 : DL$A1 . var a2 : DL$A2 .
    var d21 : DL$D21 . var d22 : DL$D22 .  var d2 : D2 .

    eq +f(d1,[a1]1) = f1(d1,a1) . eq +f(d1,[a2]2) = f2(d1,a2) .
    eq +g(d21,[a1]1) = g1(d21,a1) . eq +g(d22,[a2]2) = g2(d22,a2) .
    ceq +g(d2,[a1]1) = g1(d0.1,a1) if d2 :: DL$D21 = false .
    ceq +g(d2,[a2]2) = g2(d0.2,a2) if d2 :: DL$D22 = false .
endfm
```

Example 10 (Horizontal composition of XOR-BSeq and D&C Lingos)
The lingos $\Lambda_{xor.BSeq}$ of Example 3 and $\Lambda_{D\&C}$ of Example 4 share the same D_1, namely, $\mathbb{N}$. They do therefore have a horizontal composition $\Lambda_{xor.BSeq} \oplus_{\vec{d}_0} \Lambda_{D\&C}$ for any choice of $\vec{d}_0$. We leave as an exercise for the reader to define an appropriate view $\boldsymbol{v}$ from the theory DLINGO *to* NAT-PAIR *in Maude, so that the horizontal composition $\Lambda_{xor.BSeq} \oplus_{\vec{d}_0} \Lambda_{D\&C}$ is just the instantiation* HOR-COMP{v}.

4.2   Functional Composition of Lingos

Given two lingos Λ and Λ' such that the output data type D_2 of Λ coincides with the input data type of Λ', it is possible to define a new lingo $\Lambda \odot \Lambda'$ whose f and g functions are naturally the compositions of f and f' (resp. g and g') in a suitable way. $\Lambda \odot \Lambda'$ is clearly more complex and harder to compromise than either Λ or Λ'. In particular, its parameter space is $A \times A$ with cardinality $|A| \times |A'|$; a much bigger set whose secret parameters (a, a') are considerably harder to guess than either a or a'.

Definition 6 (Functional Composition) *Given lingos $\Lambda = (D_1, D_2, A, f, g)$ and $\Lambda' = (D_2, D_3, A', f', g')$, their functional composition is the lingo $\Lambda \odot \Lambda' = (D_1, D_3, A \times A', f \cdot f', g * g')$, where for each $d_1 \in D_1$, $d_3 \in D_3$, and $(a, a') \in A \times A'$,*

- $f \cdot f'(d_1, (a, a')) =_{def} f'(f(d_1, a), a')$,

- $g * g'(d_3, (a, a')) =_{def} g(g'(d_3, a'), a)$.

$(D_1, D_3, A \times A', f \cdot f', g * g')$ is indeed a lingo, since we have:

$$g * g'(f \cdot f'(d_1, (a, a')), (a, a')) = g(g'(f'(f(d_1, a), a'), a'), a)$$
$$= g(f(d_1, a), a) = d_1.$$

The functional composition transformation $(\Lambda, \Lambda') \mapsto \Lambda \odot \Lambda'$ can be formally specified in Maude with a parametrised module, parametric on the theory COMP-LINGOS of (functionally) composable lingos.

```
fth COMP-LINGOS is
    sorts D1 D2 D3 A A'.
    op f : D1 A -> D2 . op f' : D2 A' -> D3 .
    op g : D2 A -> D1 . op g' : D3 A' -> D2 .

    var d1 : D1 . var d2 : D2 . var a : A . var a' : A' .
    eq g(f(d1,a),a) = d1 . eq g'(f'(d2,a')) = d2 .

    op param : Nat -> A . op param' : Nat -> A' .
endfth

fmod FUN-COMP{CL :: COMP-LINGOS} is
    sort pairA . op a[_,_] : CL$A CL$A' -> pairA [ctor] .

    var d1 : D1 . var d3 : D3 . var a : A . var a' : A' . var n : Nat .
    op f.f' : D1 pairA -> D3 . eq f.f'(d1,a[a,a']) = f'(f(d1,a),a') .
    op g*g' : D3 pairA -> D1 . eq g*g'(d3,a[a,a']) = g(g'(d3,a'),a) .

    op param.param' : Nat -> pairA .
    eq param.param'(n) = a[param(n),param'(n)] .
endfm
```

Example 11 (Functional composition of XOR-BSeq and D&C Lingos)
The lingos $\Lambda_{xor.BSeq}$ of Example 3 and $\Lambda_{D\&C}$ of Example 4 are functionally composable as $\Lambda_{xor.BSeq} \odot \Lambda_{D\&C}$, because $\mathbb{N}$ is both the output data type of $\Lambda_{xor.BSeq}$ and the input data type of $\Lambda_{D\&C}$. It is possible to define an appropriate view w from the theory COMP-LINGOS to NAT-PAIR in Maude, so that the functional composition $\Lambda_{xor.BSeq} \odot \Lambda_{D\&C}$ is just the instantiation FUN-COMP{w}. Note that, by Lemma 4 below, and Theorem 1, $\Lambda_{xor.BSeq} \odot \Lambda_{D\&C}$ is f checkable, in spite of $\Lambda_{xor.BSeq}$ not being so.

Lemma 4 *Given lingos $\Lambda = (D_1, D_2, A, f, g)$ and $\Lambda' = (D_2, D_3, A', f', g')$, if Λ' if f-checkable, then the functional composition $\Lambda \odot \Lambda'$ is also f-checkable.*

Proof: We need to show that for each $(a, a') \in A \times A'$ there exists $d_3 \in D_3$ such that $\nexists d_1 \in D1$ such that $d_3 = f'(f(d_1, a), a')$. But, by assumption, there exists a $d_3 \in D_3$ such that $\nexists d_2 \in D2$ such that $d_3 = f'(d_2, a')$. This fact applies, in particular, to $d_2 = f(d_1, a)$. $\square$

4.3 Modifying and Composing Lingos by means of Data Adaptors

Two lingos $\Lambda = (D_1, D_2, A, f, g)$ and $\Lambda' = (D_2', D_3, A', f', g')$ will not be composable via functional composition if $D_2 \neq D_2'$. However, they may be functionally composable by using a *data adaptor* that "connects" the data types D_2 and D_2' (i.e makes them equal). The most obvious case is when D_2 and D_2' are in bijective correspondence, i.e., when, as data types, they are identical up to a change of data representation. For example, D_2 may be a binary representation of naturals and D_2' a decimal representation of naturals. This can be generalized to the case when D_2 is in bijective correspondence with a subset of D_2' by means of a *section-retract pair* (j, r) of functions $j : D_2 \to D_2'$ and $r : D_2' \to D_2$ such that $\forall d_2 \in D_2$, $r(j(d_2)) = d_2$. A section-retract pair (j, r) will be denoted $(j, r) : D_2 \rightleftarrows D_2'$. Note that this forces j to be injective and r surjective, and therefore defines a bijective correspondence between $j(D_2)$ and D_2'. Note also that (j, r) and (r, j) are both section-retract pairs iff $j : D_2 \to D_2'$ is bijective with inverse bijection $j^{-1} = r$. Therefore, section-retract pairs generalize bijective correspondences. We will treat the general section-retract pair case, leaving the bijective special case to the reader. Furthermore, in what follows we will call a section-retract pair $(j, r) : D_2 \rightleftarrows D_2'$ a *data adaptor*.

The first key observation is that, given a lingo $\Lambda = (D_1, D_2, A, f, g)$ we can obtain another lingo by either adapting D_1 by means of a data adaptor $(j, r) : D_1' \rightleftarrows D_1$, or adapting D_2 by means of a data adaptor $(j', r') : D_2 \rightleftarrows D_2'$.

Definition 7 *Given a lingo* $\Lambda = (D_1, D_2, A, f, g)$ *and data adaptors* $(j, r) : D_1' \rightleftarrows D_1$ *and* $(j', r') : D_2 \rightleftarrows D_2'$, *we define* $(j, r); \Lambda$ *and* $\Lambda; (j', r')$ *as follows:*

- $(j, r); \Lambda = (D_1', D_2, A, \widehat{f}, \widehat{g})$, *s.t.* $\forall d_1' \in D_1'$, $\forall d_2 \in D_2$, $\forall a \in A$, $\widehat{f}(d_1', a) = f(j(d_1'), a)$ *and* $\widehat{g}(d_2, a) = r(g(d_2, a))$

- $\Lambda; (j', r') = (D_1, D_2', A, \widetilde{f}, \widetilde{g})$, *s.t.* $\forall d_1 \in D_1'$, $\forall d_2' \in D_2'$, $\forall a \in A$, $\widetilde{f}(d_1, a) = j'(f(d_1, a))$ *and* $\widetilde{g}(d_2', a) = g(r'(d_2'), a)$

Theorem 3 $(j, r); \Lambda$ *and* $\Lambda; (j', r')$ *from Definition 7 are both lingos.*

Proof: $(j, r); \Lambda = (D_1', D_2, A, \widehat{f}, \widehat{g})$ is a lingo because,

$$\widehat{g}(\widehat{f}(d_1', a), a) = r(g(f(j(d_1'), a), a)) = r(j(d_1')) = d_1'$$

Likewise, $\Lambda; (j', r') = (D_1, D_2', A, \widetilde{f}, \widetilde{g})$ is a lingo because,

$$\widetilde{g}(\widetilde{f}(d_1, a), a) = g(r'(j'(f(d_1, a)), a)) = g(f(d_1, a), a) = d_1$$

$\square$

Theorem 4 *For* $(j, r); \Lambda$ *and* $\Lambda; (j', r')$ *from Definition 7 the following equality of lingos holds:*
$$((j, r); \Lambda); (j', r') = (j, r); (\Lambda; (j', r'))$$

Proof: Let use denote $((j,r);\Lambda);(j',r')=(D_1',D_2',A,f_1,g_1)$ and $(j,r);(\Lambda;(j',r'))=(D_1',D_2',A,f_2,g_2)$. We need to show that $f_1=f_2$ and $g_1=g_2$. Indeed, for all $d_1'\in D_1'$, $d_2\in D_2'$ and $a\in A$ we have:

$$f_1(d_1',a)=j'(\widehat{f}(d_1',a))=j'(f(j(d_1'),a))=\widetilde{f}(j(d_1'),a)=f_2(d_1',a)$$

$$g_1(d_2',a)=\widehat{g}(r'(d_2',a))=r(g(r'(d_2',a)))=r(\widetilde{g}(d_2',a))=g_2(d_2',a)$$

$\square$

The following result shows how data adaptors can enable functional compositions of lingos that could not be so composed otherwise.

Theorem 5 *Let $\Lambda=(D_1,D_2,A,f,g)$ and $\Lambda'=(D_2',D_3,A',f',g')$ be lingos, $(j,r):D_2\rightleftarrows D_2'$ a data adaptor, and all the above-mentioned data types and functions computable. Then, the following two functional compositions of lingos are identical:*

$$(\Lambda;(j,r))\odot\Lambda'=\Lambda\odot((j,r);\Lambda')$$

Proof: Let $(\Lambda;(j,r))\odot\Lambda'=(D_1,D_3,A\times A',f_1,g_1)$, and $\Lambda\odot((j,r);\Lambda')=(D_1,D_3,A\times A',f_2,g_1)$. We need to show that $f_1=f_2$ and $g_1=g_2$. Indeed, for all $d_1\in D_1$, $d_3\in D_3$ and $(a,a')\in A\times A'$ we have:

$$\begin{aligned}f_1(d_1,(a,a'))&=f'(\widetilde{f}(d_1,a),a')=f'(j(f(d_1,a)),a')\\&=\widehat{f'}(f(d_1,a),a')=f_2(d_1,(a,a')),\end{aligned}$$

$$\begin{aligned}g_1(d_3,(a,a'))&=\widetilde{g}(g'(d_3,a'),a)=g(r(g'(d_3,a')),a)\\&=g(\widehat{g'}(d_3,a'),a)=g_2(d_3,(a,a')).\end{aligned}$$

$\square$

4.4 Cartesian Products and Tupling of Lingos

Recall that a lingo is just a (non-trivial) 3-sorted (Σ,E)-algebra. By Birkhoff's Variety Theorem, (Σ,E)-algebras are closed under Cartesian products. Since products of non-trivial (Σ,E)-algebras are non-trivial, this means that lingos are closed under Cartesian products. The simplest case is that of a binary Cartesian product. Given any two lingos $\Lambda=(D_1,D_2,A,f,g)$ and $\Lambda'=(D_1',D_2',A',f',g')$, its *Cartesian product* $\Lambda\times\Lambda'$ is the lingo, $\Lambda\times\Lambda'=(D_1\times D_1',D_2\times D_2',A\times A',f\times f',g\times g')$ where, by definition,

$$f\times f'=\lambda((d_1,d_1'),(a.a')).\,(f(d_1,a),f'(d_1',a')),\text{ and}$$

$$g\times g'=\lambda((d_2,d_2'),(a.a')).\,(g(d_2,a),g'(d_2',a')).$$

This generalizes in a straightforward way to the product of k lingos, $k\geq 2$, i.e., Given a family of k lingos $\{\Lambda_i=(D_{1.i},D_{2.i},A_i,f_i,g_i)\}_{1\leq i\leq k}$, their *Cartesian*

product $\Lambda_1 \times \ldots \times \Lambda_k$ is the lingo: $\Lambda_1 \times \ldots \times \Lambda_k = (D_{1.1} \times \ldots \times D_{1.k}, D_{2.1} \times \ldots \times D_{2.k}, A_1 \times \ldots \times A_k, f_1 \times \ldots \times f_k, g_1 \times \ldots \times g_k)$.

A related lingo composition operation is the tupling of lingos which share the same input data type. Given lingos $\Lambda = (D_1, D_2, A, f, g)$ and $\Lambda' = (D_1, D_2', A', f', g')$, their *tuping* $[\Lambda, \Lambda']$ is the 5-tuple, $[\Lambda, \Lambda'] = (D_1, D_2 \times D_2', A \times A', [f, f'], g \lhd g')$ where, by definition, $[f, f'] = \lambda(d_1, (a, a')). \ (f(d_1, a), f'(d_1, a'))$ and $g \lhd g' = \lambda((d_2, d_2'), (a, a')). \ g(d_2, a)$. $[\Lambda, \Lambda']$ is indeed a lingo, since we have:

$$(g \lhd g')([f, f'](d_1, (a, a'))) = (g \lhd g')(f(d_1, a), f'(d_1, a')) = g(f(d_1, a), a) = d_1.$$

Tupling generalizes in a straightforward way to the product of k lingos, $k \geq 2$ sharing a common input data type, i.e., given a family of k lingos $\{\Lambda_i = (D_1, D_{2.i}, A_i, f_i, g_i)\}_{1 \leq i \leq k}$, their *tupling* $[\Lambda_1 \times \ldots \times \Lambda_k]$ is the lingo:

$$[\Lambda_1, \ldots, \Lambda_k] = (D_1, D_{2.1} \times \ldots \times D_{2.k}, A_1 \times \ldots \times A_k, [f_1, \ldots, f_k], g_1 \lhd g_2 \ldots, g_k)$$

with the obvious definition of $[f_1, \ldots, f_k]$ and $g_1 \lhd g_2 \ldots, g_k$ generalizing the case $k = 2$.

To illustrate the usefulness of the tupling operation we need yet another simple remark: by Birkhoff's Variety Theorem, (Σ, E)-algebras are closed under *subalgebras*. In particular, if $\Lambda = (D_1, D_2, A, f, g)$ is a lingo, a *sublingo* is a Σ-algebra $\Lambda' = (D_1', D_2', A', f', g')$ such that $D_1' \subseteq D_1$, $D_2' \subseteq D_2$, $A' \subseteq A$, $f' \subseteq f$ and $g' \subseteq g$. Since we wish Λ' to be non-trivial, we also require that D_1', D_2' and A' are all non-empty. Λ' is then a lingo because Λ is so, as Birkhoff pointed out. We then write $\Lambda' \subseteq \Lambda$ to denote the sub-lingo relation. Here is an interesting example:

Example 12 *Let $\Lambda = (D_1, D_2, A, f, g)$ be a lingo. Recall its f-checkable transformed lingo $\Lambda^\sharp$ from Theorem 1. Note that we have a sub-lingo inclusion, $\Lambda^\sharp \subseteq [\Lambda, \Lambda]$.*

4.5 Towards Non-malleable Lingos

In hindsight, lingo transformations and lingo compositions may provide useful methods to obtain non-malleable lingos. We discuss below some methods that we conjecture will be useful for this purpose. A first method is applying a transformation of the form $\Lambda \mapsto (j, r); \Lambda$, where $(j, r) : D_0 \rightleftarrows D_1$ is such that $j(D_0) \subset D_1$ is a "sparse" subset of D_1. For example, Λ can be Λ_{xor} or $\Lambda_{xor}^\sharp$ and D_0 can be the set of all the words in the English language, with n large enough so that j can represent words as bit-vectors in $\{0, 1\}^n$ by mapping each English word to a (possibly padded) concatenation of the ASCII bit-vectors for its letters. The intruder will see an obfuscated message of the form $j(d_0) \oplus a$ with d_0 a word in English, and using the recipe $x \oplus y$ will replace it by the obfuscated message $j(d_0) \oplus a \oplus r_{xor}(a')$. But recipe $x \oplus y$ is unlikely to pass muster for $(j, r); \Lambda_{xor}$, since, due to the sparsity of $j(D_0)$ in D_1, it is highly unlikely that the f-checking equality test $j(d_0) \oplus r_{xor}(a') = j(r(j(d_0) \oplus r_{xor}(a')))$ will hold,
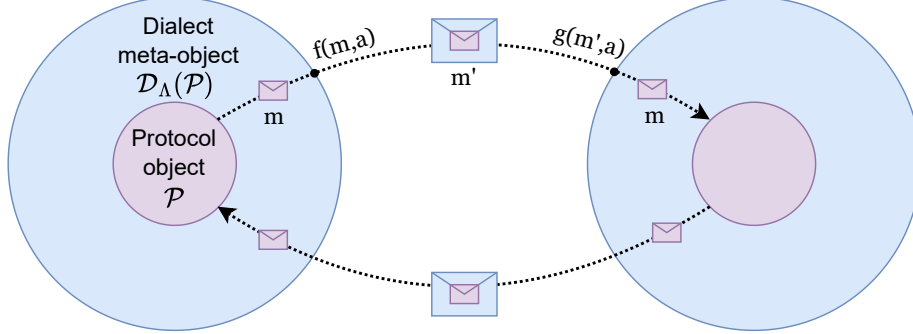
Figure 3: Dialect meta-object enhanced from [2].

since this would exactly mean that $j(d_0) \oplus r_{xor}(a')$ is (the ASCII bit-vector representation of) a word in English. This is not a proof that $(j, r); \Lambda_{xor}$ is non-malleable: it is just a hint that it seems unlikely to be malleable. A similar hint could be given for $(j, r); \Lambda_{xor}^\sharp$.

Given a malleable lingo Λ, we conjecture that a second method to transform it into a non-malleable one may be by means of a functional composition $\Lambda \odot \Lambda'$ with a suitably chosen lingo Λ'. For example, Λ may satisfy the equation $f(f(d_1, a_1), a_2) = f(f(d_1, a_2), a_1)$. But Λ' may be chosen so that the corresponding equation for $\Lambda \odot \Lambda'$, namely,

$$f'(f(f'(f(d_1, a_1), a_1'), a_2), a_2') = f'(f(f'(f(d_1, a_2), a_2'), a_1), a_1'),$$

where $a_1, a_2 \in A$ and $a_1', a_2' \in A'$, fails to hold, thwarting the success of some recipes. Again, we are just giving some hints about methods that need to be further developed. This is analogous to how malleable cryptographic functions are made non-malleable by composing them with other suitable functions.

5 Dialects

The *dialect* construction is a *formal pattern* of the form :

$$\mathcal{D} : (\Lambda(D), \mathcal{P}(D)) \mapsto \mathcal{D}(\Lambda(D), \mathcal{P}(D))$$

where $\mathcal{P}(D)$ is a protocol, i.e., an actor-based system, with D its data type of payloads, $\Lambda(D)$ is a lingo with input data type D, and $\mathcal{D}(\Lambda(D), \mathcal{P}(D))$ is a transformed protocol that endows $\mathcal{P}(D)$ with lingo $\Lambda(D)$. In $\mathcal{D}(\Lambda(D), \mathcal{P}(D))$, each protocol participant of $\mathcal{P}(D)$ is *wrapped* inside a *dialect meta-object* that uses lingo $\Lambda(D)$ to obfuscate the communication between the honest protocol actors of $\mathcal{P}(D)$. We call $\mathcal{D}(\Lambda(D), \mathcal{P}(D))$ *a dialect* for $\mathcal{P}(D)$. The $\mathcal{D}$ construction is both *protocol-generic* and *lingo-generic*: it applies to any protocol $\mathcal{P}(D)$ and to any lingo $\Lambda(D)$, provided D is shared. The essential behavior of a dialect is summarized in Figure 3, which shows how a meta-object's sending action uses

f to transform outgoing messages and a meta-object's receiving action uses g (both with the same *shared* secret parameter a) to decode the original payload and pass it to the receiving object. In $\mathcal{D}(\Lambda(D), \mathcal{P}(D))$ the code of protocol $\mathcal{P}(D)$ is not changed in any manner: the meta-object wrappers are added in an entirely modular way.

In Maude, the generic $\mathcal{D}$ construction is specified as parametrised object modules (using keywords `omod` and `endom`), with two parameter theories: `PMLINGO` and `PROTOCOL`. The former is the theory for Lingos presented in Section 3. The `PROTOCOL` theory is specified as

```
oth PROTOCOL is
    sort Payload . msg to_from_:_ : Oid Oid Payload -> Msg .
endoth
```

requiring a sort `Payload`, which has to match with sort `D1` from `PMLINGO`, and a message format. The object module also declares a *Dialect* class with the following attributes.

1. `conf`: the underlying protocol (or dialect)

2. `in-buffer`: collection of incoming messages before applying the lingo's decoding function (i.e. g)

3. `peer-counters`: a map between object identifiers and natural numbers that serves as a counter for the number of messages exchanged. It is also used in the computation of the parameters used by the lingo's transformation functions.

Furthermore, dialects add three new rules available in Figure 4. These rules are protocol generic, meaning they can be applied independently of the protocol. The rule labels and their semantics are as follows.

1. Rule labelled out processes messages produced by the wrapped protocol, sending the transformed messages into the network, by applying the lingo's f function with the corresponding parameter.

2. Rule labelled deliver collects incoming messages to the underlying protocol in the `in.buffer` attribute.

3. Rule labelled in processes messages stored in the buffer, by applying the lingo's g function with the proper parameter, delivering the original version of the message to the protocol.

Notice how in Figure 4, both rules `out` and `in` apply the f and g respectively and supply the argument by calling the lingo's `param` function. Recall from Section 3, that this function computes a value $a \in A$ from $\mathbb{N}$. The dialect meta-object is responsible of making the parameter change. In a nutshell, the dialect supplies to the `param` function the current message number from the map `peer-counter`. Nevertheless, this is one of the approaches that dialects can use to make lingo's more unpredictable.

$$in \; : \; < O_1 : DC \,|\, in.buffer : M_{in}, atts' > \; (to \; O_1 \; from \; O_2 \; : \; P)$$
$$\rightarrow \; < O_1 : DC \,|\, in.buffer : (M_{in} \cup (to \; O_1 \; from \; O_2 \; : \; P), atts' >$$

$$out \; : \; < O_1 : DC \,|\, conf : (< O_1 : C \,|\, atts > \; (to \; O_2 \; from \; O_1 \; : \; P) \cup \; M),$$
$$peer.counters : R, \; atts' >$$
$$\rightarrow \; < O_1 : DC \,|\, conf : (< O_1 : C \,|\, atts > \; M \setminus (to \; O_2 \; from \; O_1 \; : \; P)),$$
$$\text{update}(peer.counters : R, atts') >$$
$$(to \; O_2 \; from \; O_1 \; : \; f(P, param(R[O_2])))$$

$$deliver \; : \; < O_1 : DC \,|\, conf : (< O_1 : C \,|\, atts > M), \; in.buffer : M_{in},$$
$$peer.counters : R, \; atts' >$$
$$\rightarrow \; < O_1 : DC \,|\, conf : (< O_1 : C \,|\, atts > M \cup \{g(M_{selected}, \text{param}(R[O_2]))\},$$
$$in.buffer : (M_{in} \setminus M_{selected}),$$
$$\text{update}(peer.counters : R, atts') >$$
$$if \; size(M_{in}) \geq egressArity \; \wedge M_{selected} := take(egressArity, M_{in}) \; \wedge$$
$$fromOidTag(M_{selected}, R) = true \; .$$

Figure 4: Meta-Actor Rewrite Rules.

Example 13 (Dialects over MQTT) *Given the lingo from Example 3 we can transform the MQTT protocol of Example 1 by means of a dialect. To do so we need to instantiate the* **DIALECT** *module with a valid lingo view and a protocol view such as*

```
protecting DIALECT{xor-seq-l, MqttProtocol{Nat} .
```

Let us use the the **BitVec**$\{N\}$ *type described in Section 3 as the payload of messages from MQTT. We can have obtain a dialect where the lingo uses the exclusive or operation over* **BitVec**$\{N\}$*, but adapted to comply with the* **PMLINGO** *theory. Our formal semantics of MQTT is parametric enough to provide a representation of its payload as bit vectors just by changing the view given to the parameter. The instantiation of such a scenario is as follows.*

```
protecting DIALECT{xor-lingo{BitVec{4}}, MqttProtocol{BitVec{4}}} .
```

In the case we need to increase the size, for example to 8, we can do so in a very simple way. Just by changing the view given as parameter, for both lingo and protocol, we obtain a more secure dialect since the range of parameters A has increased exponentially.

```
protecting DIALECT{xor-lingo{BitVec{8}}, MqttProtocol{BitVec{8}}} .
```

Another dialect instantiation with stronger guaranties would be the lingo from Example 10. We provide in Appendix **??** *an adapted example view* `xor&dc-lingo-BitVec{8}` *for Example 10 and a view* `dhor-comp` *for DHOR-COMP. Thanks to the modularity of lingos we can obtain the new dialect just with the following line.*

```
protecting DIALECT{DHOR-COMP{xor&dc-lingo-BitVec{8}},
                            MqttProtocol{BitVec{8}}} .
```

Furthermore, if we want a dialect in which messages are first encoded with XOR, and then made stronger through the use of Divide and Check, we can automatically obtain a dialect that uses the lingo of Example 11 as

```
protecting DIALECT{FUN-COMP{xor+dc-lingos-BitVec{8}},
                            MqttProtocol{BitVec{8}}} .
```

All the examples shown in the paper are executable in Maude, and we made them available in a GitHub repository at `https://github.com/v1ct0r-byte/Protocol-Dialects-as-Formal-Patterns` *.*

6   Concluding Remarks

Dialects are a resource-efficient approach as a first line of defense against outside attackers. Lingos are invertible message transformations used by dialects to thwart attackers from maliciously disrupting communication. In this paper we propose new kinds of lingos and make them stronger through lingo transformations, including composition operations, all of them new. These lingo transformations and composition operations are *formal patterns* with desirable security properties. We also propose a refined and simpler definition for dialects by replacing two former dialect composition operations in [2] by considerably simpler and more efficient lingo composition operations. All these concepts and transformations have been formally specified and made executable in Maude.

Our next step is to explore lingos in action as they would be used by dialects in the face of an on-path attacker. We have been developing a formal intruder model that, when combined with the dialect specifications provided in this paper, can be used for statistical and probabilistic model checking of dialects. This model provides the capacity to specify the probability that the intruder correctly guesses secret parameters. For this we are using QMaude [30], which supports probabilistic and statistical model checking analysis of Maude specifications.

References

[1] K. Gogineni, Y. Mei, G. Venkataramani, and T. Lan, "Verify-pro: A framework for server authentication using communication protocol dialects," in *MILCOM 2022-2022 IEEE Military Communications Conference (MILCOM)*, pp. 450–457, IEEE, 2022.

[2] D. Galán, V. García, S. Escobar, C. Meadows, and J. Meseguer, "Protocol dialects as formal patterns," in *European Symposium on Research in Computer Security*, pp. 42–61, Springer, 2023.

[3] M. Sjoholmsierchio, "Software-Defined Networks: Protocol Dialects," Master's thesis, Naval Postgraduate School, Monterey, California, USA, Dec. 2019.

[4] D. Lukaszewski and G. G. Xie, "Towards software defined layer 4.5 customization," in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, pp. 330–338, IEEE, 2022.

[5] T. Ren, R. Williams, S. Ganguly, L. De Carli, and L. Lu, "Breaking embedded software homogeneity with protocol mutations," in *Security and Privacy in Communication Networks: 18th EAI International Conference, SecureComm 2022, Virtual Event, October 2022, Proceedings*, pp. 770–790, Springer, 2023.

[6] Y. Mei, K. Gogineni, T. Lan, and G. Venkataramani, "MPD: moving target defense through communication protocol dialects," in *Security and Privacy in Communication Networks - 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6-9, 2021, Proceedings, Part I* (J. García-Alfaro, S. Li, R. Poovendran, H. Debar, and M. Yung, eds.), vol. 398 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 100–119, Springer, 2021.

[7] K. Gogineni, Y. Mei, G. Venkataramani, and T. Lan, "Can you speak my dialect?: A Framework for Server Authentication using Communication Protocol Dialects," *arXiv preprint arXiv:2202.00500*, 2022. Publisher: arXiv Version Number: 1.

[8] C. Talcott, "Dialects for coap-like messaging protocols," *arXiv preprint arXiv:2405.13295*, 2024.

[9] J. Meseguer, "Building correct-by-construction systems with formal patterns," in *Recent Trends in Algebraic Development Techniques - 26th IFIP WG 1.3 International Workshop, WADT 2022, Aveiro, Portugal, June 28-30, 2022, Revised Selected Papers* (A. Madeira and M. A. Martins, eds.), vol. 13710 of *Lecture Notes in Computer Science*, pp. 3–24, Springer, 2022.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. M. Oliet, J. Meseguer, and C. Talcott, *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science, Springer, July 2007.

[11] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott, "Programming and symbolic computation in Maude," *JLAMP*, vol. 110, p. 100497, Jan. 2020.

[12] J. Meseguer, "Conditional rewriting logic as a united model of concurrency," *Theor. Comput. Sci.*, vol. 96, no. 1, pp. 73–155, 1992.

[13] J. Meseguer, "Twenty years of rewriting logic," *J. Algebraic and Logic Programming*, vol. 81, pp. 721–781, 2012.

[14] M.-O. Stehr, J. Meseguer, and P. C. Ölveczky, "Rewriting logic as a unifying framework for Petri nets," in *Unifying Petri Nets*, pp. 250–303, Springer, 2001.

[15] N. Martí-Oliet and J. A. Verdejo-López, "Implementing CCS in Maude," in *Actas de las VIII Jornadas de Concurrencia: Cuenca, 14 a 16 de junio de 2000*, pp. 81–96, Universidad de Castilla-La Mancha, 2000.

[16] J. Meseguer, "A Logical Theory of Concurrent Objects and its realization in the Maude Language," in *Research Directions in Concurrent Object-Oriented Programming* (G. Agha, P. Wegner, and A. Yonezawa, eds.), pp. 314–390, MIT Press, 1993.

[17] M. Katelman, S. Keller, and J. Meseguer, "Rewriting semantics of production rule sets," *Journal of Logic and Algebraic Programming*, vol. 81, no. 7-8, pp. 929–956, 2012.

[18] S. Liu, P. C. Ölveczky, and J. Meseguer, "Modeling and analyzing mobile ad hoc networks in Real-Time Maude," *Journal of Logical and Algebraic Methods in Programming*, 2015.

[19] R. Bobba, J. Grov, I. Gupta, S. Liu, J. Meseguer, P. C. Ölveczky, and S. Skeirik, "Survivability: Design, formal modeling, and validation of cloud storage systems using Maude," in *Assured Cloud Computing* (R. H. Campbell, C. A. Kamhoua, and K. A. Kwiat, eds.), ch. 2, pp. 10–48, Wiley-IEEE Computer Society Press, 2018.

[20] S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y.-M. Wang, "A systematic approach to uncover security flaws in GUI logic," in *2007 IEEE Symposium on Security and Privacy (SP'07)*, pp. 71–85, IEEE, 2007.

[21] J. Meseguer and G. Roșu, "The Rewriting Logic Semantics Project," *Theoretical Computer Science*, vol. 373, pp. 213–237, 2007.

[22] K. Bae, J. Meseguer, and P. C. Ölveczky, "Formal patterns for multirate distributed real-time systems," *Science of Computer Programming*, vol. 91, pp. 3–44, 2014.

[23] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, and K. Sonmez, "Pathway logic: Symbolic analysis of biological signaling," in *Biocomputing 2002*, pp. 400–412, World Scientific, 2001.

[24] C. Talcott, S. Eker, M. Knapp, P. Lincoln, and K. Laderoute, "Pathway logic modeling of protein functional domains in signal transduction," in *Biocomputing 2004*, pp. 568–580, World Scientific, 2003.

[25] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott, "Maude manual (version 3.3.1)," tech. rep., SRI International, 2023.

[26] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[27] S. Liu, J. Meseguer, P. C. Ölveczky, M. Zhang, and D. A. Basin, "Bridging the semantic gap between qualitative and quantitative models of distributed systems," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, pp. 315–344, 2022.

[28] U. Hunkeler, H. L. Truong, and A. J. Stanford-Clark, "Mqtt-s - a publish/subscribe protocol for wireless sensor networks.," in *COMSWARE* (S. Choi, J. Kurose, and K. Ramamritham, eds.), pp. 791–798, IEEE, 2008.

[29] M. Hron, "Are smart homes vulnerable to hacking," August 16, 2018.

[30] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo, "Qmaude: quantitative specification and verification in rewriting logic," in *International Symposium on Formal Methods*, pp. 240–259, Springer, 2023.