

# Privacy, Abstract Encryption and Protocols: an ASM Model – Part I

Dean Rosenzweig, Davor Runje and Neva Slani

University of Zagreb

**Abstract.** We introduce an executable formal model of abstract encryption using the specification language AsmL, based on Abstract State Machines of Gurevich, providing a simple executable models for cryptographic protocols. We show strong universality properties of our descriptions of patterns, protocol roles and environment behaviors—no ASM program can do better, given the same information.

## 1 Introduction

We provide an executable formal model of abstract encryption using the specification language Asml [GSCG01], based on the Abstract State Machines model of Gurevich. The features of the ASM methodology enable us to avoid some difficulties of the more usual term–model of abstract encryption.

This seems to be the first formal model of abstract encryption that captures the nondeterministic character of encryption, and that can distinguish a situation where a known encryption is matched by identity, from the more usual situation where it is matched by its contents, through decryption. It also seems to be the first such model to capture, in an abstract setting, the notions of one–way functions, indistinguishability and nonmalleability, which are central to computational, probabilistic polytime cryptography, as local properties, native to the underlying formal framework—that of ASMs.

For a hierarchy of pattern classes we show rather strong universal properties: our patterns can do as well as any ASM program, given the same initial information, in both the job of extracting information from messages input, and in the job of creating messages to output. The composition and the semantics of pattern classes is given abstractly here, by an abstract syntax and deduction rules; the reader can examine and download their AsmL code from [web].

Out of patterns protocol roles can be composed. They are automatically executable. We show that any environment behavior with a multiset of such roles can also be described by a configuration of patterns. This means that a formal proof of attack impossibility, showing that a *description* of such behavior with patterns is impossible, also proves that no ASM program can mount an attack.

Implicit in these results is a model–checker for attacks: our patterns, protocol roles and environment behavior descriptions are running objects which can print and read themselves. We expect to present such a model checker explicitly soon.

A way towards employment of theorem-proving technology of simple first-order logic with equality for security protocol analysis, suggested by our work, is hinted at in the concluding remarks, introducing Part II of this paper.

We build on the entire body of the ASM literature, as well as on formal studies of cryptographic protocols. In particular, G. Bella and E. Riccobene have used ASMs to analyze cryptographic protocols before [BR97, BR98]. They have however, unlike our model, reflected the usual term-models of abstract encryption. Our logical analysis in Part II, which drove Part I in a way, has been directly influenced by [DMP01], from where we borrow the usage of predicate source. Unlike their logic of actions, we are able to get by with simple first-order logic with equality.

Due to space limitations, we cannot introduce the reader very gently either to AsmL, ASMs or cryptographic protocols; we have to assume of the reader some familiarity with both bodies of literature.

We are indebted to the volume editors for being extremely forthcoming, and to anonymous referees for valuable comments.

## 2 Abstract Encryption

The notion of privacy of object-oriented programming languages of the C++ family, given some provisos, provides enough machinery for a faithful model of abstract encryption (on the abstraction level of the programming language). The provisos which should suffice are

1. the language has no facilities for explicit manipulation of addresses and (mis)usage of known memory layout;
2. the equality of class-objects is intensional, i.e. two newly created class-objects are never equal, unless an explicit redefinition of equality says so;
3. the language has a precise mathematical semantics.

The specification language AsmL, developed recently at Microsoft Research [GSCG01], satisfies all the above requirements. To get the idea, consider the small class in figure AsmL 1.

---

```
sealed public class Hash
  private subject as Message
  public accept(m as Message) as Boolean
    return subject=m
  public override Equals(obj as Object) as Boolean
    match obj
      h as Hash: return h.accept(subject)
      _: return false
```

**AsmL 1.** Hash

---

Whatever type `Message` might be, `hash(m) = new Hash(m)` defines a one-way function! It is a 1–1 function in view of redefined equality, it is easy to compute, but, within the abstraction level of the language, impossible to invert in general: unless an agent (program, process, ...) already has an access to the subject, it cannot obtain it through its hash.

It is well known that, as soon as we have one way functions, we also have strong cryptography [DK01].

The basic classes defined in figure AsmL 2 should suffice to convey the idea. To conserve some space, we do not treat hashes, timestamps and signatures in this paper, but they can (and will) be added in a smooth way.

---

```
type Message = Key or Nonce or Encryption or Byte or Boolean or MessageSeq
sealed public class Nonce

sealed public class Encryption
  private k as Key
  private m as Message
  beDecrypted(d as Key) as Message?
    if k.accept(d) then return m else return null

abstract public class Key
  abstract function accept(k as Key) as Boolean
  function encrypt(m as Message) as Encryption
    return new Encryption(me, m)
  function decrypt(e as Encryption) as Message?
    return e.beDecrypted(me)
```

---

## AsmL 2. Basic Classes

---

Intuitively, if a program has access to an object  $e$  of class `Encryption` without having access to a key which would get accepted by  $e.k$ , it has no way of obtaining  $e.m$ . This behavior is ensured if the concrete subclasses of `Key` are sealed without any added functionality, cf. figure AsmL 3. Even more, the program has no way of distinguishing  $e$  from another `Encryption` with the same property—as far as no appropriate key is accessible, they are equally amorphous, and interchangeable.

This is, to the best knowledge of the authors, also the first formal model of abstract encryption that captures the nondeterministic character of encryption, and that can distinguish a situation where a known encryption is matched by identity, from the more usual situation where it is matched by its contents, through decryption. While not often seen in protocols, matching of encryptions by identity is not entirely inconceivable, say in order to thwart a replay attack. If the deterministic character of some encryption algorithms needs to be regained, an appropriate class of encryptions can have its equality overruled.

Given a precise mathematical semantics for the language, the above remarks can be made completely precise.

---

```

sealed public class PrivateKey extends Key
  override function accept(k as Key) as Boolean
    match k
      pk as PublicKey: return pk.accept(me)
      _: return false
  public opposite() as Key
    return new PublicKey(me)

sealed public class PublicKey extends Key
  private priv as PrivateKey
  override function accept(k as Key) as Boolean
    return priv=k
  public override Equals(pub as Object) as Boolean
    match pub
      pk as PublicKey: return pk.accept(priv)
      _: return false

sealed public class SharedKey extends Key
  override function accept(k as Key) as Boolean
    return me=k
  public opposite() as Key
    return me

```

---

### AsmL 3. Key Classes

---

The mathematical semantics for AsmL, in particular the treatment of its object-oriented aspects, is as yet somewhat implicit, scattered around the body of ASM literature [BG01,GGV02,GSV01,GSCG01], although no one doubts that we shall very soon see it spelled out in full detail. Thus our discussion here necessarily remains somewhat sketchy, and will have to be redone. Yet we believe that the semantics is understood well enough to be certain about our claims.

The semantics is based on a notion of state as a first-order structure with a background of (hereditarily finite) collections: sequences, sets and maps [BG00]. It is also often useful to view a state as a mapping from *locations* [GSCG01] to *values*, elements of the underlying universe.

The semantics supports notions of *reachability* through a value, and of *accessibility* to a class-object, both relative to a state, as follows:

*Reachability* of a value through another value is the smallest reflexive and transitive relation which is

- closed under support of collections (in case of maps, both the keys and the values);
- closed under components of structures;
- closed under public members of class-objects.

A value is *accessible* to a class-object  $o$  if it is reachable through

- values accessible as global in the corresponding declaration scope;

- data members of  $o$ , public or private.

We skip the complications due to `friend` and other access control of AsmL, since we don't use it in our models, we assume members to be either `public` or `private`. Since the `decrypt` function of class `Key` is a proper function, i.e. its evaluation has no side effects and its value depends only on its explicit arguments, we may safely add a closure condition including its values to the notion of accessibility:

- if  $k$  and  $e$  are accessible to  $o$  in  $\mathcal{S}$ , then the value of  $k.\text{decrypt}(e)$  is reachable through  $e$ .
- if  $k$  is a decryption key reachable through  $x$ , then so is the opposite key  $\bar{k}$ .

The last statement reflects the fact that public key can in most schemes be extracted from (the usual representation of) the private one. This is directly represented by the function `opposite()`; the correspondence is 1–1 in view of the redefined equality of `PublicKey`.

Values accessible to  $o$  in a state  $\mathcal{S}$  suffice to interpret all terms (in general excluding procedure and function calls) occurring in  $o$ 's program (though not in general to calculate the updates), and thus constitute the (partial) *local state* of  $o$  in  $\mathcal{S}$ ,  $\mathcal{S}_o$ . If a value is accessible to  $o$  in  $\mathcal{S}$ , we shall also say that it is accessible in  $\mathcal{S}_o$ .

The above notions are closely related to the notions of relevant or critical object and local state of [BGS99,BG01].

Sketchy as the above might be, we are convinced that the fully spelled out semantics of AsmL will support the following

*Claim.* Let  $k$  be a key,  $m$  a message,  $\mathcal{S}_o$  the local state of a class–object  $o$  in state  $\mathcal{S}$ .

1. Let, in  $\mathcal{S}$ ,  $e$  be an encryption such that  $k.\text{decrypt}(e) = m$ . If neither  $k$  nor  $m$  is accessible in  $\mathcal{S}_o$ , then they are also inaccessible in the extended partial state  $(\mathcal{S}_o, e)$  (a state  $o$ 's program started with input  $e$  would start in).
2. Let  $k$  be a decryption key (shared or public) and  $m$  a message. If either  $\bar{k}$  or  $m$  is inaccessible in  $\mathcal{S}_o$ , and remains inaccessible through possible submachine states created by  $o$ 's program fired at  $\mathcal{S}$ , the program of  $o$  fired at  $\mathcal{S}_o$  cannot create an encryption  $e$  such that  $k.\text{decrypt}(e) = m$ .
3. Let, in  $\mathcal{S}$ ,  $e_1, e_2$  be two encryptions inaccessible in  $\mathcal{S}_o$ , so that no key  $k$  such that  $k.\text{decrypt}(e_i) \neq \text{null}$  is accessible in  $\mathcal{S}_o$ . Then there is an isomorphism of extended local states  $(\mathcal{S}_o, e_i)$ ,  $i \in \{1, 2\}$  which maps  $e_1$  to  $e_2$ , and is an identity on accessible objects of  $\mathcal{S}_o$ .

The above properties are the faithful abstract counterparts of the classical probabilistic–polytime properties of encryption: 1. corresponds to *secrecy*, 2. corresponds to *nonmalleability*, and 3. corresponds to *indistinguishability*, [BDPR98]. We shall in the sequel often call the last property *indistinguishability*, not necessarily restricted to encryptions.

This is, to the best of the authors’ knowledge, the first formal model of abstract encryption which can reflect the notions of secrecy, nonmalleability and indistinguishability, central to computational cryptography, as immediate local properties, native to the underlying formal machinery—ASMs.

This correspondence frees us from several difficulties that the usual term-models of abstract encryption have with ‘messages seen but not understood’, [AR02,Syv00], and paves the way for future work on a smooth relation between abstract and computational encryption models. We shall discuss it in more detail elsewhere.

For full code of our basic model of abstract encryption, the reader might care to look at [web].

### 3 Patterns

*Example 1.* The notorious example of the Needham–Schroeder protocol is usually informally presented as follows:

$$\begin{array}{ccc}
 a & \xrightarrow{\{a, n_a\}_b} & b \\
 a & \xleftarrow{\{n_a, n_b\}_a} & b \\
 a & \xrightarrow{\{n_b\}_b} & b
 \end{array}$$

where agents are identified with their public keys, and encryption of message  $m$  under key  $k$  is depicted by  $\{m\}_k$ .

Such informal presentations, as well as most formal studies of abstract encryption [DMP01,FHG98,FHG99,FA01,CDL+00,DLMS02] and others, are, one way or another, about message patterns. Listening to the subject, we take the patterns seriously. In the first approximation, patterns are something like messages but can also contain variables. They are usually, in formal studies, represented by terms of some vocabulary. Here they also play some roles usually played by terms. However, since messages are not just ground terms, formal representation of patterns, faithful to their informal use, can hardly be construed just as terms. In the above informal diagram, the patterns convey messages both created for output and analyzed from input, and these two roles are not entirely symmetric.

In our executable model we have constructed a hierarchy of intelligent pattern classes, appropriate for the encryption model, which can both match and create messages (and print and read themselves etc). In this section we provide a succinct formal representation of their construction and semantics, and prove the essential features of what they can do. The idea is that *patterns suffice*: whatever information an (ASM) agent can extract from a message, a pattern can do as well, given the same information; whatever message an (ASM) agent can create, a pattern can do as well, given the same information.

When we compare capabilities of patterns to those of ASM programs, we assume that the programs are *isolated*: they start, possibly with an input, and

in finitely many steps they terminate, possibly yielding an output, without communicating with the external world in the meantime. This means also not calling any functions and procedures (except for `decrypt` and creation of objects, but all forms of iteration are allowed to them). The rationale is that, in a cryptographic protocol situation, all communication with the external world should be explicitly included in the protocol specification; what we have in mind is the internal steps between two communication points.

The interested reader can examine and download the full AsmL code for pattern classes from [\[web\]](#).

### 3.1 Abstract Syntax

The composition of pattern-classes can be succinctly represented by the following abstract syntax:

$$\begin{aligned}
 p &\rightarrow \text{createPat} \mid \text{bytePat} \mid \text{boolPat} \mid [p_1, \dots, p_n] \mid \text{letPat} \\
 \text{createPat} &\rightarrow \text{keyPat} \mid \text{noncePat} \mid \text{encryptPat} \\
 \text{keyPat} &\rightarrow \text{eKeyPat} \mid \text{dKeyPat} \\
 \text{dKeyPat} &\rightarrow \text{prv} \mid \text{sv} \\
 \text{eKeyPat} &\rightarrow \text{op}(\text{dKeyPat}) \mid \text{pbv} \\
 \text{noncePat} &\rightarrow \text{nv} \\
 \text{bytePat} &\rightarrow \text{Byte} \mid \text{bv} \\
 \text{boolPat} &\rightarrow \text{true} \mid \text{false} \mid \text{tv} \\
 \text{encryptPat} &\rightarrow \text{EP}(\text{ev}, \text{eKeyPat}, p) \mid \text{ev} \\
 \text{letPat} &\rightarrow \text{let } v = p_1 \text{ in } p_2
 \end{aligned}$$

where the patterns with names ending in ‘v’ represent the variables of appropriate types. In the `let` pattern the variable  $v$  is assumed to be fresh, occurring only in  $p_2$ . In the encryption pattern, the variable  $ev$  is a (somewhat clumsy) technical device only needed in order to retain in an assignment every object created and received; it involves a global syntactical constraint: this variable must occur uniquely, as the first argument of an `EP` construct, in a pattern. Of course, the encryption variable  $ev$  can not occur in pattern  $p$ . Whenever it is of no importance, we drop the variable  $ev$  in an encryption pattern.

### 3.2 Assignments and Matching

An *assignment* is a mapping of some pattern variables to messages of appropriate types. The idea is that an assignment represents the ‘knowledge’ of a (local) state, the collection of creatable messages (nonces, keys and encryptions) accessible to it. We shall sometimes use the AsmL notation  $v \in \sigma$ , where  $\sigma$  is an assignment and  $v$  a variable, with the meaning that  $v$  is in the *domain* of  $\sigma$ , and the *ad hoc* notation  $\sigma.(v \rightarrow m)$  for the extension of  $\sigma$  mapping also  $v$  to  $m$  (given  $v \notin \sigma$ ). Let us fix some terminology.

A creatable message is *known* to  $\sigma$  if it is in the range of  $\sigma$  (we shall also consider a public key  $\bar{k}$  to be known whenever its private counterpart  $k$  is); it

is *learned* by  $\sigma$  through a message  $m$  if it is not reachable through  $\sigma$ , but is reachable through  $\sigma, m$ .

An assignment  $\sigma$  is *complete* if all creatable messages reachable through  $\sigma$  are also known to  $\sigma$ ; an assignment is complete for a state  $S$  if all creatable messages accessible to  $S$  are known to  $\sigma$ .

Two messages  $m$  and  $m'$  are *indistinguishable* for a complete assignment  $\sigma$  if they are indistinguishable for any (local) state  $\sigma$  is complete for.

A term-like aspect of patterns is that appropriate assignments map them to messages.

**Definition 1.** *Let  $\tau$  be an assignment and  $p$  a pattern. By simultaneous induction we define what it means for  $\tau$  to be appropriate for  $p$ , and if it is, what is the message  $p^\tau$ :*

- if  $p$  is a constant  $c$ , then  $\tau$  is appropriate for it and  $c^\tau = c$ ;
- if  $p$  is a variable  $v$  then  $\tau$  is appropriate for it if  $v$  is in its domain and  $v^\tau = \tau(v)$ ;
- if  $p$  is  $\text{op}(v)^\tau$  then  $\tau$  is appropriate for it if it is appropriate for  $v$ , and  $\text{op}(v)^\tau = \overline{v^\tau}$ ;
- if  $p$  is concatenation  $[p_1, \dots, p_n]^\tau$  then  $\tau$  is appropriate for  $p$  if it is appropriate for all  $p_i$ , and  $[p_1, \dots, p_n]^\tau = [p_1^\tau, \dots, p_n^\tau]$
- if  $p$  is an encryption pattern  $\text{EP}(ev, kp, p)$  then  $\tau$  is appropriate for it if both
  - $\tau$  is appropriate for  $ev, kp$  and  $p$ , and
  - for some key  $k$  we have  $kp^\tau = k \wedge k.\text{decrypt}(ev^\tau) = p^\tau$ ;
then  $\text{EP}(ev, kp, p)^\tau = ev^\tau$ ;
- if  $p$  is  $\text{let } v = p_1 \text{ in } p_2$ ,  $\tau$  is appropriate for it if both
  - $\tau$  is appropriate for  $p_1$ , and
  - $\tau.(v \rightarrow p_1^\tau)$  is appropriate for  $p_2$ ;
then  $(\text{let } v = p_1 \text{ in } p_2)^\tau = p_2^{\tau.(v \rightarrow p_1^\tau)}$

Notice that it is an encryption variable in the encryption pattern which enforces the above consistency constraint.

If  $\sigma$  is an assignment, a pattern  $p$  is *consistent* with it if there is a  $\tau \supseteq \sigma$  appropriate for  $p$ ; a message  $m$  matches such a consistent pattern  $p$  over  $\sigma$  if, for appropriate  $\tau \supseteq \sigma$ , we have  $p^\tau = m$ . We shall call such  $\tau$  a *matching witness* for  $\sigma, p, m$ .

There will be in general many patterns that a message  $m$  matches over  $\sigma$ . Is there a ‘best’ one in some clear sense? The answer depends on what we want to do with the message. Consider the following two scenarios:

1. An isolated program  $\pi$  in local state  $S$  with input  $m$  needs to extract some information learned through  $m$ . A pattern  $p$  would be ‘the best’ in this sense if no isolated program  $\pi$  could do better than  $p$ , given an assignment  $\sigma$  complete for  $S$ .
2. An isolated program  $\pi$  in local state  $S$  wants to output  $m$ . A pattern  $p$  would be ‘the best’ in this sense if it enforces creation of all objects that  $\pi$  starting from  $S$  needs to create.

We pursue both scenarios.

### 3.3 Patterns Suffice for Analysis

Let  $\sigma$  be an assignment,  $m$  a message. We say that  $m$  *opens boxes* in  $\sigma$  if  $\sigma$  knows some encryption  $e$  without knowing its decryption key, but learns such a key through  $m$ .

To factor out some complications, we first prove the theorem about analysis for the case of a message  $m$  that doesn't open boxes in  $\sigma$ .

**Theorem 1.** *Let  $\sigma$  be a complete assignment,  $m$  a message that doesn't open boxes in  $\sigma$ . Then there is a pattern  $p$  such that any message  $m'$ , which is indistinguishable from  $m$  for  $\sigma$ , matches  $p$  over  $\sigma$ . Respective minimal matching witnesses  $\tau, \tau'$  are complete and also indistinguishable for  $\sigma$ .*

This means that no isolated program with input can analyze a message better than a pattern could. Since both  $\sigma$  and  $\tau$  are complete here, knowing and learning obtain very sharp meanings: a message is known if it is in the range of  $\sigma$ ; it is learned by the match if it is in the range of  $\tau - \sigma$ . One could also say that a submessage of  $m$  was *matched* if it is known.

Variables occurring in  $p$  can also be split into two categories: those *learned* from  $m$  by the matching, i.e. not in  $\sigma$ ; and *matched* in  $m$  otherwise.

*Proof.* For  $\sigma, m$  we first construct the pattern  $p$ .

**Pattern Construction.** Associate to each creatable submessage  $c$  of  $m$ , a variable  $v_c$  of appropriate type, under the following provisos:

1. if  $c$  is known in  $\sigma$ ,  $c = \sigma(v)$ , then  $v_c$  is  $v$ ;
2. if  $c$  is a public key  $\bar{k}$  not in the range of  $\sigma$  but is known via the known private key  $k = \sigma(v)$ , then  $v_k$  is  $v$ ;
3. unknown public keys get a fresh *public* key variable;
4. other unknown creatable messages get a fresh variable;
5. repeated occurrences of unknown  $c$  get the same  $v_c$ .

Now construct a pattern  $p'$  by induction over  $m$  (doing some auxiliary jobs as well) as follows:

6. if  $m$  is a constant,  $p'$  is  $m$ ;
7. if  $m$  is a public key  $\bar{k}$  with an associated private key variable  $v_k$ , then  $p'$  is  $op(v_k)$ ;
8. if  $m$  is other creatable message  $c$ , with associated variable  $v_c$ , then  $p'$  is  $v_c$ .
9. if  $m$  is  $[m_1, \dots, m_n]$ , then  $p'$  is  $[p_1, \dots, p_n]$ , where  $p_i$ 's are given by the induction hypothesis;
10. in case of unknown encryption learned from  $m$ , with the decryption key reachable through  $\sigma, m$ , an additional pattern  $p_e$  has to be created and associated to  $e$ :  $p_e$  is  $EP(v'_e, op(v_k), p'_1)$ , where  $v'_e$  is a fresh encryption variable and  $p'_1$  exists by induction hypothesis. Notice that  $p'_1$  does not contain any EP or **let** constructs. It is built from variables and possibly concatenations only.

This finishes the construction of  $p'$ .

11. Let  $e_1, \dots, e_n$  be the selected encryptions in  $m$ , in an ordering respecting the syntactical constraint on **let**, then

$$p = \mathbf{let} \ v_{e_1} = p_{e_1} \ \mathbf{in} \ \dots \mathbf{let} \ v_{e_n} = p_{e_n} \ \mathbf{in} \ p',$$

$$p_{e_i} = \mathbf{EP}(v'_{e_i}, \mathbf{op}(k_{e_i}), p'_i).$$

This finishes the construction of  $p$ .

Notice that all variables occurring in  $p$ , apart from those bound by **let**, are exactly the variables  $v_c$  associated to  $c$  which are either known to  $\sigma$ , or learned through  $m$ , or some  $v'_e$  for a selected encryption  $e$ .

Now let us construct  $\tau - \sigma$ . For all creatable submessages of  $m$  which are either known to  $\sigma$  or learned through  $m$ , and which are not the encryptions selected in proviso 10., set  $\tau(v_c) = c$ . For the selected encryptions, set  $\tau(v'_e) = e$ , where  $v'_e$  is given by proviso 10. By construction,  $\tau$  is a complete extension of  $\sigma$ , and if it is a matching witness at all, then it is a minimal one.

To see that  $\tau$  is a matching witness, see that all patterns  $p'_i$  and  $p'$  consist only of constants, variables either known to  $\tau$  or shown in **let**, or concatenations. See further that each encryption variable  $v'_{e_i}$  occurs literally once in the pattern  $p$ , and is bound by  $\tau$  to  $e_i$ . By induction we have:  $\tau_i = \tau.(v_{e_1} \rightarrow e_1) \dots (v_{e_{i-1}} \rightarrow e_{i-1})$  is appropriate for  $p_{e_i}$  for  $i = 1, \dots, n$  and  $p_{e_i}^{\tau_i} = e_i$ . Finally  $\tau_{n+1} = \tau.(v_{e_1} \rightarrow e_1) \dots (v_{e_n} \rightarrow e_n)$  is appropriate for  $p'$ ,  $p'^{\tau_{n+1}} = m$  and  $p^\tau = m$ .

For some message  $m'$  indistinguishable from  $m$ , the same construction would work with the same pattern  $p$  and with the same variables, yielding  $\tau'$  which is indistinguishable from  $\tau$  for  $\sigma$ : each  $\tau(v)$  is, by straightforward induction over its construction, indistinguishable from  $\tau'(v)$  from  $\sigma$ .  $\square$

The above described patterns will suffice for most ‘normal’ conditions. We have however to consider also some scenarios which are not inconceivable, though rarely, if ever, seen in protocols, such as obtaining an encrypted message, and, some time later, the decryption key.

Since we want to make strong completeness claims, we need to add an ability to express expectations on the form of encryptions learned but not yet understood (to be possibly understood in the future). To reduce such situations to the pattern–matching paradigm we need a notion of an extended pattern. An *extended pattern* is a pair  $(p, \vec{v}_e)$ , where  $p$  is a pattern and  $\vec{v}_e$  is a sequence of encryption variables.

An assignment  $\tau$  is *appropriate* for an extended pattern  $(p, \vec{v}_e)$  if  $\tau$  is appropriate for  $p$  and  $p^\tau = [m, \tau(v_{e_1}), \dots, \tau(v_{e_n})]$ . Then we define the message  $(p, \vec{v}_e)^\tau$  as  $m$ .

**Theorem 2.** *Let  $\sigma$  be a complete assignment and  $m$  any message. Then there is a (possibly extended) pattern  $(p, \vec{v}_e)$  such that any message  $m'$ , which is indistinguishable from  $m$  for  $\sigma$ , matches  $p$  over  $\sigma$ . Respective minimal matching witnesses  $\tau, \tau'$  are complete and also indistinguishable for  $\sigma$ .*

*Proof.* We only have to cover the case when  $m$  opens some boxes in  $\sigma$ ; denote the opened boxes by  $e_1, \dots, e_n$ , named in  $\sigma$  by  $v_{e_1}, \dots, v_{e_n}$ . A message

$m' = [m, e_1, \dots, e_n]$  does not open any boxes in  $\sigma'$ , which is  $\sigma$  with  $v_{e_1}, \dots, v_{e_n}$  removed. A message is reachable through  $\sigma, m$  if and only if it is reachable through  $\sigma', m'$ . By theorem 1, there is a pattern  $p$  and an complete assignment  $\tau$  extending  $\sigma'$  such that  $p^\tau = m'$ . Notice that  $\tau$  also extends  $\sigma$  since for each encryption  $e_i$  reachable through  $m'$  in  $\sigma'$ , an appropriate variable  $v_{e_i}$  occurs in  $p$ . But then  $\tau$  is appropriate for  $(p, \vec{v}_e)$  and  $(p, \vec{v}_e)^\tau = m$ .  $\square$

**Corollary 1.** *If  $\sigma$  can distinguish  $m$  from  $m'$ , it can do so by a (possibly extended) pattern.*

### 3.4 Patterns Suffice for Synthesis

Let us turn to the scenario 2, that of creation.

We shall say that a creatable submessage  $c$  of message  $m$  gets *created* in  $m$  from  $\sigma$  if it doesn't occur in  $m$  only as a submessage of some message known to  $\sigma$ .

**Theorem 3.** *Let  $\sigma$  be a complete assignment and  $m$  any message. Then there is a pattern  $p$  matching  $m$  over  $\sigma$  such that a matching witness  $\tau \supseteq \sigma$  is complete, and a minimal extension of  $\sigma$  which knows all messages created in  $m$  from  $\sigma$ .*

This means that no isolated ASM program with output can create a message that some pattern could not have created given the same knowledge. Since both  $\sigma$  and  $\tau$  are complete here, matching and learning obtain duals, *forwarding* and *creating*: a submessage of  $m'$  if forwarded if it is in range of  $\sigma$ , and created if it is in the range of  $\tau - \sigma$ . Variables occurring in  $p$  can be classified in the same way. The essence of the crucial *source*-axiom of [DMP01] can in these terms be succinctly said: if you match a message you have created before, then someone has learned it.

*Proof.* We construct the pattern  $p$  by the same clauses as for analysis, in the proof of theorem 2, replacing provisos 3 and 10 by provisos 3' and 10':

- 3'. unknown public keys get fresh *private* key variables—we are imitating creation now, and it is impossible to create just a public key.
- 10'. like 10, but take *all* encryptions created in  $m$  from  $\sigma$ : if a key is missing, we shall create one;

If there are no encryptions created in  $m$  from  $\sigma$ , a straightforward induction proves the theorem. To see that it works in general, note that, by induction, when  $p_e^\tau$  gets evaluated, it is with an assignment which already knows all encryptions properly contained in  $e$ .  $\square$

### 3.5 Patterns Can Do by Themselves

In our executable model we have constructed a hierarchy of intelligent pattern classes, appropriate for the encryption model, which can both match and create messages (and print and read themselves etc). They have methods, `doCreate()`

and `doMatch()` which execute theorems 1 and 2, that is compute the matching witnesses by themselves.

The interested reader can find the full code of patterns classes on the [web](#). We represent these methods here abstractly, by deduction rules, and prove the relations of rules to theorems 1 and 3. The deduction rules can be seen as an abstraction the code, while the code can be seen as an implementation of the rules; in fact, they are a joint fixed point of an iterative process.

**Matching Rules.** An operational semantics of matching can be succinctly represented by the following deduction rules, with

$$\sigma, p, m \searrow \tau$$

meaning: with assignment  $\sigma$ , a message  $m$  matches the pattern  $p$ , yielding new assignment  $\tau$ . If no rule is applicable, the match fails. The output assignment extends the input one with the objects found on the way.

$$\sigma, c, c \searrow \sigma \tag{1}$$

$$\sigma, v, \sigma(v) \searrow \sigma \quad v \in \{sv, prv, pbv, nv, bv, tv, ev\} \tag{2}$$

$$\sigma, \text{op}(dv), \overline{\sigma(dv)} \searrow \sigma \quad (dv : dKeyPat) \tag{3}$$

$$\frac{v \notin \sigma}{\sigma, v, c \searrow \sigma.(v \rightarrow c)} \tag{4}$$

$$\frac{\sigma, p_i, m_i \searrow \tau \quad \tau, [p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n], [m_1, \dots, m_{i-1}, m_{i+1}, \dots, m_n] \searrow \omega}{\sigma, [p_1, \dots, p_n], [m_1, \dots, m_n] \searrow \omega} \tag{5}$$

$$\frac{ev \notin \sigma \quad \sigma.(ev \rightarrow m), p, \sigma(dv).\text{decrypt}(m) \searrow \tau}{\sigma, \text{EP}(ev, \text{op}(dv), p), m \searrow \tau} \tag{6}$$

$$\frac{ev \in \sigma \quad \sigma, p, \sigma(dv).\text{decrypt}(\sigma(ev)) \searrow \sigma}{\sigma, \text{EP}(ev, \text{op}(dv), p), \sigma(ev) \searrow \sigma} \tag{7}$$

$$\frac{\sigma^-, p_2, m \searrow \tau \quad v \notin \tau}{\sigma, \text{let } v = p_1 \text{ in } p_2, m \searrow \tau.(v \rightarrow \sigma(v))} \tag{8}$$

$$\frac{\sigma^-, p_2, m \searrow \tau \quad \tau, p_1, \tau(v) \searrow \omega}{\sigma, \text{let } v = p_1 \text{ in } p_2, m \searrow \omega^-(v \rightarrow \sigma(v))} \tag{9}$$

$$\frac{\sigma - \vec{v}_e, p, [m, \sigma(v_{e_1}), \dots, \sigma(v_{e_n})] \searrow \tau}{\sigma, (p, \vec{v}_e), m \searrow \tau} \tag{10}$$

The assignments  $\sigma^-, \omega^-$  above are respectively  $\sigma, \omega$  with the variable  $v$  removed, to ensure the local scope of `let`. Matching rule (10) handles the extended pattern case. We do not use, in our proofs, the power of the nondeterministic sequence rule, which allows backtracking in search for a successful match. We have included it nevertheless in order not to restrain the expressive power of the pattern language—without it, say if left-to-right matching were imposed, the pattern  $[\text{EP}(ev, \text{op}(dv), p), dv]$  with unknown  $dv$  would not match any message, while

the seemingly equivalent one  $\mathbf{let} \ v = \mathbf{EP}(ev, \mathbf{op}(dv), p) \ \mathbf{in} \ [v, dv]$  could match; in our code, see [web], the nondeterminism is implemented by suspending an encryption without a known decryption key, and retrying it later.

**Theorem 4.** *Let  $\sigma$  be a complete assignment and  $m$  a message. Then there is (possibly extended) pattern  $p$  such that  $\sigma, p, m \searrow \tau$ , where  $\tau$  is an analyzing assignment of theorem 1.*

*Proof.* In the view of theorem 2, we can limit the proof to the case of  $m$  not opening boxes in  $\sigma$  and a simple pattern  $p$ .

We construct a pattern  $p'$  in exactly the same way as in the theorem 1, except for one minor modification. Replace 11 by 11':

- 11'. like 11 but with an additional constraint on the ordering of encryptions  $e_i$ :  
 if the matching  $k_{e_i}$  is unknown to  $\sigma$ , then it occurs in a pattern  $p'$  or  $p'_j$  for some  $j > i$ .

Notice that, because of syntactic constraint on  $\mathbf{let}$  and by our construction, the following must also hold: if  $v_{e_i}$  does not occur in  $p'$  then  $v_{e_i}$  occurs in  $p'_j$  for some  $j > i$ . Such an ordering exists since all encryptions and decryption keys are reachable through  $\sigma, m$ .

We now prove the theorem by induction over the construction of  $m$ . The only interesting case is when  $m$  contains encryptions unknown to  $\sigma$ , with decryption keys either known to  $\sigma$  or learned by  $\sigma$  through  $m$ . In this case  $p$  is of form

$$\mathbf{let} \ v_{e_1} = p_{e_1} \ \mathbf{in} \ \dots \ \mathbf{let} \ v_{e_n} = p_{e_n} \ \mathbf{in} \ p'.$$

To match  $p$  against  $m$ , according to  $\mathbf{let}$  rule, we have first to match  $\sigma, p'$  against  $m$ , yielding  $\tau_n$ ; then match  $\tau_i, p_{e_i}$  against  $e_i$  yielding  $\tau_{i-1}$ ,  $i = n, \dots, 1$ .

See that, in the first step, matching  $p'$  against  $m$ , is the same as if all the  $e_i$  were black boxes: encryptions whose decryption keys are neither known to  $\sigma$ , nor learned by it through  $m$ . By induction over the uninteresting cases, this match succeeds, and  $\tau_n$  is a complete assignment knowing about all creatable submessages of  $m$  not enclosed in encryptions there.

By the ordering of  $e_i$ 's, both  $e_n$  and its decryption key is known to  $\tau_n$ . Now iterate the induction hypothesis (since each  $e_i$  and its decryption key is in  $\tau_i$ ) yielding:  $\tau_{i-1}$  is a complete assignment extending  $\tau_i$ , knowing all the creatable submessages of  $e_i$  which are not enclosed in nested encryptions there; further, if  $i > 1$ , the decryption key for  $e_{i-1}$  is known to  $\tau_{i-1}$ ,  $i = n, \dots, 1$ .

But then  $\tau_0$  is the  $\tau$  of the theorem.  $\square$

**Creating Rules.** An operational semantics of creating can be succinctly represented by the following deduction rules, with

$$\sigma, p \nearrow m, \tau$$

meaning: with assignment  $\sigma$  the pattern  $p$  creates a message  $m$ , yielding new assignment  $\tau$ . Creation is not supposed to fail. The output assignment extends

the input one with the objects created on the way. Of course, the output message is unique only up to inessential nondeterminism of **new**.

$$\sigma, c \nearrow c, \sigma \quad (11)$$

$$\sigma, v \nearrow \sigma(v), \sigma \quad (12)$$

$$\frac{cv \notin \sigma \quad c = \mathbf{new} \ T}{\sigma, cv \nearrow c, \sigma.(cv \rightarrow c)} \quad c \in \{nv, sv, prv\}, T \text{ appropriate} \quad (13)$$

$$\frac{\sigma, dv \nearrow \tau, k}{\sigma, \mathbf{op}(dv) \nearrow \bar{k}, \tau} \quad (14)$$

$$\frac{\sigma, p_1 \nearrow \tau, m_1 \quad \tau, [p_2, \dots, p_n] \nearrow [m_2, \dots, m_n], \omega}{\sigma, [p_1, \dots, p_n] \nearrow [m_1, \dots, m_n], \omega} \quad (15)$$

$$\frac{ev \notin \sigma \quad \sigma, kp \nearrow \tau, k \quad \tau, p \nearrow m, \omega \quad e = \mathbf{new} \ \mathbf{Encryption}(k, m_1)}{\sigma, \mathbf{EP}(ev, kp, p) \nearrow e, \omega.(ev \rightarrow e)} \quad (16)$$

$$\sigma, \mathbf{EP}(ev, kp, p) \nearrow \sigma(ev), \sigma \quad (17)$$

$$\frac{\sigma^-, p_1 \nearrow m_1, \tau \quad \tau.(v \rightarrow m_1), p_2 \nearrow m_2, \omega}{\sigma, \mathbf{let} \ v = p_1 \ \mathbf{in} \ p_2 \nearrow m_2, \omega^-.(v \rightarrow \sigma(v))} \quad (18)$$

**Theorem 5.** *Let  $\sigma$  be a complete assignment and  $m$  a message. Then there is a pattern  $p$  such that  $\sigma, p \nearrow m', \tau'$ . For any complete assignment  $\omega$ , to which no submessage, created from  $\sigma$  either in  $m$  or in  $m'$ , is known,  $m$  is indistinguishable from  $m'$ .*

*Proof.* Take a pattern  $p$  as in the proof of theorem 3. The creation rules traverse  $m$  in a bottom-up manner, creating in fact, by invoking **new**, new versions of submessages created in  $m$  from  $\sigma$ . By rules and the construction of  $p$  every submessage needed is created only once and put in the right places. Since the respective submessages created in  $m$  and  $m'$  are fresh to  $\omega$  satisfying the conditions of the theorem, there are also indistinguishable to it.  $\square$

## 4 Modelling Protocols

Protocol roles are usually informally depicted as a sequence of input–output patterns, which is often directly reflected in the formal models. We follow the venerable practice, but in our terms. A nice side effect is that we can immediately execute the roles; the theorems of the preceding section delineate precisely the result of this execution.

We show that any environment interaction with a multiset of protocol roles can also be faithfully represented by a configuration of (extended) patterns. By *any* we mean any, under some extremely general constraints, which outrule only true magic, jumping out of the abstraction level of the language. This means that impossibility of *configurations* expressing attacks of this or that kind, in fact implies impossibility of such *attacks* by any ASM programs. Since ASM programs are universal in several rather strong senses [Gur00,BG01], this universality property is as strong as one could hope for.

## 4.1 Protocol Roles

**Definition 2.** A role action is a pair of patterns; a role is a sequence of role actions of fixed length  $n$ ,  $[p_1 \Rightarrow q_1, \dots, p_n \Rightarrow q_n]$ . Current configuration of a role is a pair of assignment and action index  $(\sigma, i)$ , denoted (for brevity) by  $\sigma_i$ .

A role can be seen as an interactive machine with input and output of messages, with a program given by a sequence of actions. A configuration consists of state of memory (assignment) and program counter (action index). To specify a transition relation for such a machine, one specifies the relation of current configuration and input to next configuration and output,  $\sigma_{i-1} \xrightarrow[\text{out}_i]{\text{in}_i} \sigma_i$ :

$$\sigma_{i-1} \xrightarrow[\text{out}_i]{\text{in}_i} \sigma_i \equiv \exists \tau. \sigma_{i-1}, p_i, \text{in}_i \searrow \tau \wedge \tau, q_i \nearrow \text{out}_i, \sigma_i \quad (19)$$

*Example 2.* Roles of Needham-Schroeder protocol are as follows. Initially, variables  $pr_a$  and  $pr_b$  hold respective private encryption keys.

Initiator =  $[b \Rightarrow \text{EP}(b, [\text{op}(pr_a), n_a]), \text{EP}(\text{op}(pr_a), [n_a, n_b]) \Rightarrow \text{EP}(b, n_b)]$

Responder =  $[\text{EP}(\text{op}(pr_b), [a, n_a]) \Rightarrow \text{EP}(a, [n_a, n_b]), \text{EP}(\text{op}(pr_b), n_b) \Rightarrow \text{true}]$

**Definition 3.** A run of a role in response to inputs  $in_1, \dots, in_k$ ,  $k \leq n$  is defined as:

$$\sigma_0 \xrightarrow[\text{out}_1]{\text{in}_1} \sigma_1 \dots \xrightarrow[\text{out}_k]{\text{in}_k} \sigma_k.$$

The trace of that run is  $\sigma_0, (in_1, out_1), \dots, (in_k, out_k)$ . A run or trace is full if  $k = n$ .

## 4.2 Environment Behaviors

A protocol is a set of roles, together with some wiring connecting output of one role with input of another. The idea of the wiring is to describe the desired message-transporting behavior of the environment.

**Definition 4.** An environment behavior consists of a set of role-instances (multiset of roles)  $r_1, \dots, r_k$ , together with their initial states  $\sigma_0^1, \dots, \sigma_0^k$ , and a behavior trace: an initial state  $\sigma_0$ , and a sequence of role-tagged message pairs  $((out_i, in_i), r_j)$  such that:

- (1) For every role  $r_j$ , its initial state  $\sigma_0^j$ , together with subsequence of  $r_j$ -tagged message pairs  $((out_i, in_i), r_j)$ , is a trace of  $r_j$ .
- (2) Every message  $out_i$  can be created by an ASM program which has access to creatable messages only through  $\sigma_0$  and messages  $in_1, \dots, in_{i-1}$  (except for the ones created by itself).

Since we view an environment as a black box, we should say that two environment behaviors are *indistinguishable* if for every role instance  $r_j$  involved the resulting role traces are indistinguishable to the initial state  $\sigma_0^j$  of  $r_j$ .

Security goals for protocols are often expressed as properties of sets of role configurations. For instance, one of Lowe’s guarantees [Low97] can be expressed as follows: whenever the set contains a responder’s role  $B$  in its final state, it also contains an initiator’s role  $A$  in its final state, and they agree on some data in their assignments. Whatever a protocol goal might be, an *attack* can be defined as an environment behavior leading its roles to configurations forming a bad set, one not having the desired property. It is certainly reasonable to expect that protocol goal is invariant under isomorphism, in particular with respect to inessential nondeterminism of **new**.

An *environment behavior description* is just like an environment behavior, but with a trace description instead of a trace: replace messages  $(out_i, in_i)$  with patterns  $(outPat_i, inPat_i)$ , where  $inPat_i$  can be in general also extended.

Environment behavior descriptions can be directly executed to compute environment behaviors. The algorithm maintains an assignment of pattern variables to messages, denoted by  $\sigma_i$  after step  $i$ . The step  $i + 1$  is:

- let  $r_j$  be the role with which  $(outPat_{i+1}, inPat_{i+1})$  are tagged,
- compute  $\sigma_i, outPat_{i+1} \nearrow out_{i+1}, \tau$ ,
- then write  $out_{i+1}$  to the input of  $r_j$ ,
- then wait for  $in_{i+1}$  to appear on its output,
- then match  $\tau, inPat_{i+1}, in_{i+1}$  to get  $\sigma_{i+1}$ ,

see [web] for code.

*Example 3.* The desired postman–like environment behavior for Needham-Schroeder protocol, and Lowe’s attack on it, can be described with the following environment behavior descriptions, respectively. In both cases, take an instance  $A$  of the initiator with the initial state  $\sigma_0^A$  binding only a private key variable  $pr_a$  to its private key, and an instance  $B$  of the responder with initial state binding its private key variable  $pr_b$ . The initial state  $\sigma_0$  binds variables  $a$  and  $b$  to public keys so that  $\sigma_0(a) = \sigma_0^A(pr_a)$  and  $\sigma_0(b) = \sigma_0^B(pr_b)$ . The respective trace descriptions are then

$$\begin{aligned} \text{Postman} &= [((b, ev_1), A), ((ev_1, ev_2), B), ((ev_2, ev_3), A), ((ev_3, tv), B)] \\ \text{Lowe} &= [((\text{op}(pr_e), \text{EP}(\text{op}(pr_e), [a, n_a])), A), ((\text{EP}(b, [a, n_a]), ev_1), B), \\ &\quad ((ev_1, \text{EP}(\text{op}(pr_e), n_b)), A), ((\text{EP}(b, n_b), tv), B)] \end{aligned}$$

where  $n_a, n_b$  are fresh nonce variables,  $ev_i$  are fresh encryption variables,  $tv$  is a fresh boolean variable, and  $pr_e$  is a fresh private key variable.

Since the notion of an attack is rather general, while the notion of attack description is quite specific, one might think that a restriction to environment behavior descriptions would limit the class of attacks considered. If for a given protocol security goal we prove that there is no attack description, what have we proved? In view of the universality properties of patterns, it is not surprising that we have proved that there is no attack whatsoever, at least not in the sense of the abstract encryption model.

**Theorem 6.** *For every environment behavior  $\mathcal{B}$  there is a behavior description  $D$  with the same role instances and the same initial states such that any behavior  $\mathcal{B}'$  computed by  $D$  is indistinguishable from  $\mathcal{B}$  for the initial states.*

*Proof.* For each  $(out_i, in_i)$  and  $\sigma_i$  in  $\mathcal{B}$  we proceed as follows: from  $\sigma_{i-1}$  and  $out_i$  by theorem 3 we create a pattern  $outPat_i$  and a complete assignment  $\tau_i$ ; from  $\tau_i$  and  $in_i$  we create an extended pattern  $inPat_i$  and a complete assignment  $\sigma_i$  by theorem 2.

Let  $S_{i,j}, S'_{i,j}$  be the local states of role  $r_j$  after the  $i$ -th action of the environment  $\mathcal{B}, \mathcal{B}'$  respectively, represented by assignments  $\sigma_{i,j}, \sigma'_{i,j}$ . By induction over  $i$ , for all  $i$  and  $j$ ,  $\sigma_{i,j}$  is, for  $\sigma_{0,j}$ , indistinguishable from  $\sigma'_{i,j}$ .  $\square$

Since patterns, protocol roles and environment behavior descriptions are implemented executable objects, a model checker for protocol behaviors is implicit in the above construction. We hope to present it explicitly very soon.

## Concluding Remarks

Our work indicates that ASMs and AsmL can be very useful tools for modelling and investigating cryptographic protocols.

Since ASMs are implemented, ASM models of protocols are immediately executable; since patterns are universal, pattern-based ASM models are universal.

Known security proofs for simple protocols, such as Needham-Schroeder-Lowe, Perrig-Song, . . . are easily reproduced by hand in our current framework. It would be more exciting if we could use the models for machine-assisted, if not outright automatic, proving of security properties for larger industrial-scale protocols.

This is precisely what we intend to do, and a part of the work will soon appear in Part II of this paper.

A small prerequisite is extending our treatment to hashes, timestamps and signatures, which is straightforward. A larger prerequisite for is plugging in logic, in a useful way. The predicates `learned`, `matched`, `forwarded`, `created` have the obvious interpretation, when their subjects are the individual actions, and the actions can automatically generate first-order formulae completely determining their (fixed-point) life. The individual actions are in fact the *proclats* of [BG01], and the whole environment behavior can be seen, in view of theorem 6, as a *ken* of [BG01]. It can automatically generate a first-order formula completely describing the behavior, up to indistinguishability from the initial states. This formula is a close cousin of the Turing-tableaux description often used to prove the theorems of Church and Trakhtenbrot: if it is satisfiable at all, it is also satisfiable in an environment behavior. Together with a few simple facts about information flow, easily proved about any such ken, this sentence can be used to prove security properties with surprising simplicity, at least for the little examples we tried by hand—the key seems to be that the subjects of the basic predicates are the actions, and not, like usual, the agents, not even the roles. In Part II we explore the possibility of automatic generation of such formulae, and hooking them to theorem-proving technology.

## Future Work

There are several further avenues to explore from here, which are not Part II but real future work. Two of them seem to us to be particularly exciting:

1. In [AR02] the bridge from abstract to computational cryptography has already been built, on the level of individual messages. It is not at all difficult to carry it over to our framework. It seems however that our framework is very convenient for an attempt to extend the bridge along the dynamics of protocol execution, represented here as a simple static structure. The goal this might achieve is a neat separation of concerns: to see that a protocol is computationally safe, in sense of probabilistic polytime, it should suffice to prove a) that it is abstractly safe, and b) that the cryptographic algorithms used are computationally safe.
2. We have analyzed matching of messages to patterns, in two ways: on input and on output. This might be extended to matching patterns to patterns, in a kind of directed unification, read-to-write and write-to-read. The results should be something similar to symbolic trace analysis of [FA01], but with an important addition: adding a new role into a behavior would also be a natural step, in a systematic search for any attack whatsoever. We know by [DLMS02] that this cannot possibly converge in general, since it would make security properties decidable; but finding sufficient conditions for convergence would extract decidable classes.

## References

- AR02. Martin Abadi and Phillip Rogaway. Reconciling two views of cryptography. *Journal of Cryptology*, 15(2):103–127, 2002. [6](#), [18](#)
- BDPR98. M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *CRYPTO '98*, volume 1462 of *LNCS*, 1998. [5](#)
- BG00. Andreas Blass and Yuri Gurevich. Background, reserve, and Gandy machines. In *Proceedings of CSL'2000*, volume 1862 of *LNCS*, 2000. [4](#)
- BG01. Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms. Technical Report MSR-TR-2001-117, Microsoft Research, 2001. [4](#), [5](#), [14](#), [17](#)
- BGS99. A. Blass, Y. Gurevich, and S. Shelah. Choiceless polynomial time. *Annals of Pure and Applied Logic*, 100(1-3), 1999. [5](#)
- BR97. Giampaolo Bella and Elvinia Riccobene. Formal analysis of the Kerberos authentication system. *Journal of UCS*, 3(12):1337–1381, 1997. [2](#)
- BR98. Giampaolo Bella and Elvinia Riccobene. A realistic environment for crypto-protocol analyses by ASMs. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*, 1998. [2](#)
- CDL<sup>+</sup>00. Illiano Cervesato, Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *Proceeding of 13th IEEE CSFW*, 2000. [6](#)
- DK01. Hans Delfs and Helmut Knebl. *Introduction to Cryptography*. Springer, 2001. [3](#)

- DLMS02. N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Multiset rewriting and the complexity of bounded security protocols. Technical report, 2002. [6](#), [18](#)
- DMP01. Nancy Durgin, John Mitchell, and Dusko Pavlovic. A compositional logic for protocol correctness. In *Proceeding of 14th IEEE CSFW*, 2001. [2](#), [6](#), [11](#)
- FA01. M.P. Fiore and M. Abadi. Computing symbolic models for verifying cryptographic protocols. In *Proceeding of 14th IEEE CSFW*, pages 160–173, 2001. [6](#), [18](#)
- FHG98. F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Honest ideals on strand spaces. In *Proceeding of 11th IEEE CSFW*, 1998. [6](#)
- FHG99. F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2/3), 1999. [6](#)
- GGV02. Uwe Glaesser, Yuri Gurevich, and Margus Veanes. An abstract communication model. Technical Report MSR-TR-2002-55, 2002. [4](#)
- GSCG01. Yuri Gurevich, Wolfram Schulte, Colin Campbell, and Wolfgang Grieskamp. *AsmL: The Abstract State Machine Language*, 2001. Version 1.5. [1](#), [2](#), [4](#)
- GSV01. Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Toward industrial strength abstract state machines. Technical Report MSR-TR-2001-98, 2001. [4](#)
- Gur00. Yuri Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000. [14](#)
- Low97. Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of 10th IEEE CSFW*, 1997. [16](#)
- Syv00. Paul Syverson. Towards a strand semantics for authentication logic. In *Electronic Notes in TCS*, volume 20, 2000. [6](#)
- web. Web. <http://www.fsb.hr/~drosenzw/protocols>. [1](#), [6](#), [7](#), [12](#), [13](#), [16](#)