

The Cryptographic Abstract Machine

Dean Rosenzweig and Davor Runje

University of Zagreb

Abstract. The Cryptographic Abstract Machine is an executional model of cryptographic actions, independent of the concrete cryptographic procedures employed, even of the abstraction level of the underlying model of cryptography. This is motivated both by a theoretical purpose of relating the dynamics of protocol executions at different levels of abstraction, and by a practical purpose of enabling automatic generation of provably correct code implementing protocol roles from high level specifications. Here we define the CrAM and show how slightly refurbished message patterns of [RRS03] can be compiled to CrAM code both for analysis and for creation of messages, and prove the correctness and completeness of that compilation.

Introduction

The Cryptographic Abstract Machine (CrAM in the sequel) is an executional model of cryptographic actions, independent of the concrete cryptographic procedures employed, even of the abstraction level of the underlying model of cryptography. Three such levels can be discerned:

- *abstract* or formal model, where only the abstract structure of cryptographic messages is represented, typically by terms of a vocabulary, abstracting away from their concrete representation and concrete cryptographic algorithms, i.e. [Low96,FHG99,CDL⁺00,DMP03,BR97,BR98]. . .
- *computational* model, admitting that messages are bitstrings, operating under complexity-theoretic assumptions on cryptographic algorithms, see for instance [BN00,BDPR98] for splendid examples of the definitional effort;
- *concrete* programming APIs, relying on concrete message-encoding schemes and services of a concrete cryptographic library and/or device.

The last two levels are usually related by complexity-theoretic conjectures on combinatorial problems underlying cryptographic algorithms, such as factoring or discrete logarithm — for work towards relating the first two levels see [AR02,MW03].

We propose the CrAM in the context of a broader program, with the goal of deriving provably secure code implementing cryptographic protocols. We see the problem of achieving that goal decomposed as follows:

1. develop a language of abstract message patterns, akin to the usual messages-and-arrows representation, supporting formal proofs under the assumptions of abstract cryptography;

2. decouple the message patterns from the assumptions of abstract cryptography, allowing their direct interpretation also in computational models and/or concrete implementations of cryptographic algorithms;
3. provide a framework for translating formal proofs into computational settings, under the common assumptions on cryptographic algorithms;
4. generate provably correct implementations of pattern-based protocol descriptions for a given programming language and a cryptographic programming library.

Aim 1 was largely accomplished in [RRS03]: the message pattern language is proven to be universal for analysis and synthesis of cryptographic messages under a model of abstract cryptography. A formal proof establishing impossibility of an attack with message patterns also proves impossibility of an attack in the context of abstract cryptography.

Aim 3 is left for future work. Our intent is to develop a framework for systematic translation of proofs for the abstract cryptography model into proofs for different computational models, with target theorems of form: abstractly safe protocol + computationally safe algorithms = computationally safe protocol, for different notions of ‘computationally safe’. It seems that the properties of abstract cryptography used in a formal proof, together with the computational security criterion desired, will largely dictate the computational security assumptions on algorithms needed. We see this future work as direct continuation and application of [AR02,MW03] for analysis of static messages and [MW03,War03] for dynamic protocol execution. The CrAM seems to be the right setting, since it provides a simple and precise way of saying that cryptographic agents at different abstraction levels ‘do the same’: they execute the same CrAM program in different environments.

This paper intends to realize most (though not all) of aims 2 and 4.

We start, in section 1, by laying down the vocabulary and the assumptions common to all abstraction levels we rely on throughout the paper.

In section 2 we revisit the message pattern language of [RRS03] in order to allow different models of cryptography, at different levels of abstraction, to be plugged into patterns. For the purpose of this paper, we intentionally disregard interpretations of cryptographic objects, and place a minimal set of assumptions on a model of cryptography needed to prove equivalence of message patterns and CrAM programs. The modified language of message patterns recaptures the universality properties of patterns wrt the model of abstract cryptography of [RRS03].

The main result of this paper is generation of provably correct implementation of a pattern-based protocol description for a given programming language and a cryptographic programming library. This is accomplished by compiling patterns to CrAM code, in a provably correct and complete way. In view of the representation of protocol roles of [RRS03] as sequences of match-create pairs of patterns, we obtain compilation of protocol roles to CrAM code automatically, by pasting and glueing with instructions for input-output.

The intuition of a protocol role as an interactive machine with input and output of messages gets a concrete form in the CrAM, which is now also decoupled from any specific model of encryption, and works with any reasonable model supported by the vocabulary and assumptions.

We show that both compilation algorithms are both correct and complete, which means that CrAM programs are at least as strong as patterns, in any specific incarnation of both.

In section 3 we define the CrAM, and in section 4 we define the compilation. Proofs of security for pattern-based protocol descriptions, to be done in the completion of aim 2 indicated above, will compose with our correctness and completeness proofs presented here so as to prove security of CrAM-implementations of protocols. A CrAM-implementation becomes a real implementation either by interpreting the CrAM directly (one way of doing that is AsmL code to be found at [Web]), or by expanding CrAM programs into programs of the target implementation language in a provably correct way. The latter possibility is the only really practical one, but conceptually it is *deja vu*, and is skipped here to conserve some space. In view of our technical framework, that of ASMs as implemented in AsmL, it is rather straightforward to expand CrAM code to provably correct code in any language with a well defined ASM semantics, such as C, C++, Java [GH93,HS00,Wal95,SSB01].

The results presented here are in a sense independent of the ASM framework. The justification for relying on ASMs and AsmL (if one is still needed) is threefold:

- The framework supports our model of abstract cryptography [RRS03] well.
- Main results of this paper are essentially on compiler correctness/completeness, and ASMs are a proven time-tested tool [BR94,BD96,GZ00,SSB01] for that.
- The thrust of our program is about relating different levels of abstraction, and this is the core business of the ASM methodology.

For ASM methodology, our results then just open another application area, in a way somewhat different from that of [BR97,BR98] — we intend not only to use ASMs in order to prove protocols correct, but also to generate their implementations in a uniform way. Formal ASM models for patterns, CrAM and the compilation, in form of executable AsmL code, can be found at [Web].

1 Vocabulary and Assumptions

Any instance of the CrAM is defined wrt the following basic universes: universe M of encoded messages, and universe O of abstract cryptographic objects.

The distinction between M and O is real in concrete cryptographic APIs: every API the authors know about needs to represent a key internally before accessing it for encryption, it is accessible for encryption only in form of an abstract object, say a handle in old-fashioned APIs, an O , distinct from the byte-array representation of the key, an M used for communicating it externally. Even

plain byte-objects, such as nonces and hashes, are typically stored internally in an internal form, as an O , distinct in general from the format used for external communication, that of M , say PKCS #7 [PKC93] or PGP messages.

We assume the universe O to be split into a finite set of pairwise disjoint types T_1, \dots, T_n . For each type T we will use, both in patterns and in the CrAM, typed variables v^T to range over T . We shall also use *raw variables*, which will range over raw messages from M , with metavariable r , as well as metavariables o , m for objects from O , M respectively, and metavariable v for arbitrary object variables.

Each type is associated with one or more of *kinds* such as `Nonce`, `Hash`, `EncryptionKey`, etc. The table below lists the kinds, together with a metavariable associated with each kind, to denote variables to range over types of that kind.

K	<code>PrivateKey</code>	h	<code>Hash</code>	e	<code>Encryption</code>
k	<code>PublicKey</code>	c	<code>PrimitiveValue</code>	sk	<code>SigningKey</code>
sh	<code>SharedKey</code>	cr	<code>CreatableObject</code>	s	<code>Signature</code>
n	<code>Nonce</code>	ek	<code>EncryptionKey</code>	vk	<code>VerifyingKey</code>
a	<code>AgentName</code>	dk	<code>DecryptionKey</code>	t	<code>Tuple</code>

The names of the kinds listed above are at this level only a heuristic indication of our intentions; every incarnation of the CrAM must provide its meanings for them by providing the associated concrete types.

The type `RSAES-OAEP-DecryptionKey-1024` is an example of such a concrete type of the kind `PrivateKey`, embodying a concrete cryptographic decryption primitive RSADP (using default hash function and mask generation algorithm: SHA-1), with a concrete encoding method EME-OAEP, according to PKCS #1 standard [PKC02], and specific key material — 1024 bit long RSA private key. Universe M of a CrAM instance with this type must also include encodings of all messages encrypted with RSAES-OAEP using 1024 bit long private RSA keys, together with encodings of all 1024 bit long private and public RSA keys.

We assume the following relationships between kinds:

1. Each type of kind `PrivateKey` or `SharedKey` is, nonexclusively, a `DecryptionKey` type or a `SigningKey` type; on the other hand each `DecryptionKey` or `SigningKey` type is either a `PrivateKey` type or a `SharedKey` type;
2. Each `PublicKey` or `SharedKey` type is, nonexclusively, an `EncryptionKey` type or a `VerifyingKey` type; on the other hand each `EncryptionKey` type or `VerifyingKey` type is either a `PublicKey` or a `SharedKey` type;
3. the `CreatableObject` types are exactly all `Nonce` types, `PrivateKey` types and `SharedKey` types.

The `PrimitiveValue` types are understood to consist of booleans, bytes and other primitive values needed. The type `RSAES-OAEP-DecryptionKey-1024` from the example above has the following kinds associated with it: `PrivateKey`, `DecryptionKey` and `CreatableObject`.

The vocabulary contains the procedure signatures listed in the following table, associated to each type T of appropriate kind (with signatures in terse, but

understandable programming style):

kind of T	"shared interface"	"object interface"
any	$T \text{ decode}_T(M)^\dagger$	$M \text{ encode}_T(T)$
Tuple	$T \text{ createTuple}_T(\vec{M})^\dagger$	$\vec{M} \text{ analyzeTuple}_T(T)$
CreatableObject	$T \text{ create}_T()^*$	
Hash	$M \text{ hash}_T(M)$	
PrivateKey		$T' \text{ op}_T(T)$
EncryptionKey		$M \text{ encrypt}_T(T, M)^{* \dagger}$
DecryptionKey		$M \text{ decrypt}_T(T, M)^\dagger$
SigningKey		$M \text{ sign}_T(T, M)^{* \dagger}$
VerifyingKey		Boolean $\text{verify}_T(T, M, M)^\dagger$

where the procedures marked with $*$ can be nondeterministic, and those marked with \dagger might fail, say in case of unexpected format of an argument; \vec{M} is a shorthand for $\text{Seq of } M$ of AsmL.

The intended interpretation of $\text{encode}_T/\text{decode}_T$ is encoding/decoding between objects of type T and encoded messages, according to a formatting standard carried by T . Decoding might fail, since the message may not be a valid encoding of some T . Thus, given a type T and an encoded message $m \in M$, we either have $\text{encode}_T(\text{decode}_T(m)) = m$, or decode_T fails at m .

Having the same encoding is the only criterion of object identity used in the CrAM.

It would be a nice property of encodings that the domains of decode_T of different types be disjoint; this would thwart type-flaw attacks on protocols, and it is usually achieved by type-tagging the encodings. For the purpose of this paper, we do not make any such assumptions.

Types of kind **Tuple** represent different methods of encoding/decoding of message tuples, of fixed or bounded or arbitrary arity. Encoding of a tuple with createTuple_T can fail, say if an input sequence is of wrong length. Given a tuple object t of type T , we assume $\text{encode}_T(\text{createTuple}_T(\text{analyzeTuple}_T(t))) = \text{encode}_T(t)$. Notice the use of encode_T for establishing equality of abstract objects of type T .

Types of kind **CreatableObject**, those of nonces, private and shared keys, also have a $\text{create}_T()$ method attached, returning a presumably fresh object.

An object of type T of kind **Hash** carries a concrete hashing algorithm used in hash_T . Different types may carry different algorithms, say types **SHA-1** and **MD5** carry SHA-1 [FIP95] and MD5 [Riv92] cryptographic hash algorithms respectively.

The same holds for objects of key-types, but they, in addition to carrying an algorithm, also carry a concrete key to be used in encrypt_T , decrypt_T , sign_T and verify_T . For example, an object of type **RSAES-OAEP-Decrypt-1024** carries a 1024 bit long RSA private key, while $\text{decrypt}_{\text{RSAES-OAEP-Decrypt-1024}}$ is exactly the procedure **RSAES-OAEP-Decrypt** from [PKC02].

For each type T of kind **PrivateKey** we also have a function op_T , returning an object of attached type T' , which has to be of kind **PublicKey**. Extending the

previous example, the type `RSAES-OAEP-EncryptionKey-1024` of kinds `PublicKey` and `EncryptionKey`, is the codomain of `opRSAES-OAEP-DecryptionKey-1024`.

We also assume that decodings of the codomain of `encryptT` belong to an `Encryption` type, and that decodings of the codomain of a `signT` belong to a `Signature` type.

This is all that we have to assume of key and hash types for the purposes of this paper. It is our understanding that also

- either `encryptT'(opT(K), m)` fails,
- or `decryptT(K, encryptT'(opT(K), m)) = m`,

and likewise for `verifyT/signT` and for shared keys, but we do not use such assumptions in this paper.

It is also common to assume that created objects are "fresh", encryptions are "hard to break", signatures are "hard to forge", and hashes are "hard to invert". But we cannot even *say* on this level of abstraction what exactly "fresh", "hard", "break", "forge" or "invert" mean, since CrAM instances at different levels of abstraction, and different instances at the same level of abstraction, will in general interpret these notions differently. In [RRS03] we provide one such interpretation in the context of abstract encryption, where i.e. "hard" means impossible and "invert" means invert. In more concrete instances of the CrAM these notions will obtain different probabilistic polytime interpretations. It is a primary purpose of the CrAM to facilitate relating protocol executions under different interpretations.

Notation and Mapping to OOP

In the sequel we use an OOP notation for the above vocabulary, in order to minimize the distance between notation and the AsmL model. In the signatures of the third column of the above table, the "object interface", the first argument is always of type T . It is common in OOP to suppress such an argument, and see it as the "method subject" or "message receiver". The type-subscript is also superfluous, since the type is implicit in the subject. Thus we shall write $x.\text{encrypt}(m)$ for `encryptT(x, m)` whenever x is of type T of kind `EncryptionKey`, and likewise for all signatures in the third column.

Thus our kinds can be seen as *interfaces*, and the concrete types as concrete *classes* implementing some of these interfaces.

The signatures of the second column, the "shared interface", do not have such designated subjects, they can be seen as declaring "shared" or "static" methods of the attached types. Since a shared method cannot technically be declared in an AsmL interface, we use the following device.

In the AsmL model [Web] the typed variables will also be represented by objects. The type of an object representing v^T knows about T , and the shared signatures can become the interface this type has to implement. Thus, if v^T is a variable of type T , we shall write $v^T.\text{decode}(m)$ instead of `decodeT(m)` whenever we have a variable v^T at hand, what we always will. We shall use all signatures

$$\begin{aligned}
varPat & ::= c \mid K \mid k \mid sh \mid n \mid a \mid e \mid s \mid h \mid r \\
constPat & ::= c : C \\
opPat & ::= k : \text{op}(K) \\
keyOfPat & ::= k : \text{keyOf}(a) \\
encPat & ::= e : E(\text{keyPat}, p) \\
sigPat & ::= s : S(\text{keyPat}, r) \\
hashPat & ::= h : H(r) \\
tuplePat & ::= t : [p_1, \dots, p_n] \\
letPat & ::= \text{let } r = p_1 \text{ in } p_2 \\
p & ::= varPat \mid constPat \mid opPat \mid keyOfPat \mid encPat \mid sigPat \\
& \quad \mid hashPat \mid tuplePat \mid letPat \\
keyPat & ::= k : \text{keyOf}(a) \mid k : \text{op}(K) \mid K \mid k \mid sh
\end{aligned}$$

Fig. 1. Message pattern syntax

from the second column in this way only. Thus both patterns and the CrAM can remain blissfully ignorant of the concrete types: accessing them only through interfaces, they have no need to mention the concrete types at all.

2 Message Patterns

In this section we revisit the message patterns of [RRS03]. We have related them there to the abstract model of encryption, defined how they can match a message, extracting information from it, and create a message, and shown that appropriate patterns, for any message, can do as well as any AsmL program.

The kinds, interfaces of (types of) cryptographic objects, if used in the patterns, allow us to decouple the patterns from the particular model of abstract encryption, so patterns can apply to any cryptographic model supported by the vocabulary. The patterns can thus be seen as representing the abstract structure of a broad class of cryptographic messages.

This requires slight adaptation of syntax, essentially adopting kinds of cryptographic objects (in addition to extending the syntax with signatures, hashes and agent names). The rules for matching and creation need to be adapted so as to rely only on the functionality of interfaces provided by the kinds, instead of that of the abstract encryption model.

The universality results for patterns wrt abstract encryption can be easily regained by defining the appropriate types.

2.1 Syntax and Semantics

The syntax of message patterns is given in figure 1. It deviates from that of [RRS03] in the following respects:

$\sigma, \rho, v, m \searrow \sigma, \rho$	$\sigma(v).\text{encode}() = m$
$\sigma, \rho, r, m \searrow \sigma, \rho$	$\rho(r) = m$
$\sigma, \rho, v, m \searrow \sigma + \{v \mapsto v.\text{decode}(m)\}, \rho$	$v \notin \sigma$
$\sigma, \rho, r, m \searrow \sigma, \rho + \{r \mapsto m\}$	$r \notin \rho$
$\sigma, \rho, c : C, m \searrow \sigma + \{c \mapsto C\}, \rho$	$c \notin \sigma, C.\text{encode}() = m$
$\sigma, \rho, k : \text{op}(K), m \searrow \sigma + \{k \mapsto o_k\}, \rho$	$k \notin \sigma, o_k = \sigma(K).\text{op}(),$ $o_k.\text{encode}() = m$
$\sigma, \rho, k : \text{keyOf}(a), m \searrow \sigma + \{k \mapsto k.\text{decode}(m)\}, \rho$	$k \notin \sigma, m = \nu(\sigma(a).\text{encode}())$
$\frac{kp, \sigma, \nu \searrow_K o_k, \sigma'}{\sigma, \rho, s : S(kp, r), m \searrow \sigma' + \{s \mapsto s.\text{decode}(m)\}, \rho}$	$s \notin \sigma, o_k.\text{verify}(\rho(r), m)$
$\frac{\dots \sigma_{i-1}, \rho_{i-1}, p_i, m_i \searrow \sigma_i, \rho_i \dots}{\sigma, \rho, h : H(r), m \searrow \sigma + \{h \mapsto h.\text{decode}(m)\}, \rho}$	$h \notin \sigma, h.\text{hash}(\rho(r)) = m$
$\frac{\dots \sigma_{i-1}, \rho_{i-1}, p_i, m_i \searrow \sigma_i, \rho_i \dots}{\sigma, \rho, t : [p_1, \dots, p_n], m \searrow \sigma_n, \rho_n}$	$t \notin \sigma, o_t = t.\text{decode}(m)$ $\vec{m} = o_t.\text{analyzeTuple}(),$ $\sigma_0 = \sigma + \{t \mapsto o_t\},$ $\rho_0 = \rho, i = 1, \dots, n$
$\frac{\sigma, \rho, p_2, m \searrow \sigma', \rho' \quad \sigma', \rho', p_1, \rho'(r) \searrow \sigma'', \rho''}{\sigma, \rho, \text{let } r = p_1 \text{ in } p_2, m \searrow \sigma'', \rho''}$	$r \notin \rho$
$\frac{kp, \sigma, \nu \searrow_K o_k, \sigma' \quad \sigma' + \{e \mapsto e.\text{decode}(m)\}, \rho, p, o_k.\text{decrypt}(m) \searrow \sigma'', \rho''}{\sigma, \rho, e : E(kp, p), m \searrow \sigma'', \rho''}$	$e \notin \sigma$
$v_k, \sigma, \nu \searrow_K \sigma(v_k), \sigma$	$v_k \in \sigma, v_k = K, k, sh$
$k : \text{op}(K), \sigma, \nu \searrow_K o_k, \sigma + \{k \mapsto o_k\}$	$k \notin \sigma, K \in \sigma, o_k = \sigma(K).\text{op}()$
$k : \text{keyOf}(a), \sigma, \nu \searrow_K o_k, \sigma + \{k \mapsto o_k\}$	$k \notin \sigma, a \in \sigma, o_k = k.\text{decode}(\nu(\sigma(a).\text{encode}()))$

Fig. 2. Matching patterns to messages

1. the variables are now typed according to the vocabulary, denoted in the syntax by corresponding metavariables of appropriate kind;
2. the encryption variable of [RRS03] has moved up front, and obtained another role, additional to that of recording an encryption seen/created: its type now determines how to decode an encryption from an external to an internal representation, from an M to an O ;
3. each compound pattern has been decorated with an appropriate variable for the same purpose;
4. patterns $k : \text{keyOf}(a), h : H(r), s : S(\text{keyPat}, r)$ have been added to deal with agent names, hashes and signatures respectively.

Patterns can now match messages from M , and come with a storage split into the following stores:

- *object store* σ , a finite map of object variables (variable–objects) to O ;
- *message store* ρ , a finite map of raw variables to M ;
- *certificate store* ν , a finite map of encodings of agent names (objects of `AgentName` types) to encodings of public keys (objects of `PublicKey` types).

Pattern matching is defined with a relation $\sigma, \rho, \nu, p, m \searrow \sigma', \rho'$, where m is a message and p is a pattern. When ν is understood from context, we write $\sigma, \rho, p, m \searrow \sigma', \rho'$. The relation is defined by deduction rules, very similar to those of [RRS03], with the following modifications:

1. new rules have been added for the new constructs;
2. `encode` and `decode` must mediate between internal and external forms of cryptographic objects;
3. equality of objects is determined only by their external representations;
4. rules are adapted to rely only on the functionality provided by types of cryptographic objects and variables used; and
5. the nondeterministic rule for matching tuples has been replaced by its leftmost–deterministic instance for simplicity, since this doesn’t affect the universality results of [RRS03].

The rules are given in figure 2, together with side conditions on their applicability.

Creating messages by patterns is defined by deduction rules as well, proving statements of form

$$p, \sigma, \rho, \nu \nearrow m, \sigma'$$

When ρ, ν are understood from context, we write $p, \sigma, \rho \nearrow m, \sigma'$ or $p, \sigma \nearrow m, \sigma'$.

The rules, given in figure 3, are adapted from the corresponding rules of [RRS03] in very much the same way as the matching rules above.

2.2 Recapturing the Relation to Abstract Encryption

In order to recapture the universality properties that the patterns of [RRS03] have wrt the model of abstract encryption explained there, we only have to describe the types used. To keep things simple, we shall not extend the types of abstract messages of [RRS03] here (which is straightforward to do); we shall, for sake of this argument, suppress the new constructs (signatures, hashes and agent names) from patterns instead.

Both universes O and M will coincide with the type `Message` of [RRS03]. The types of kinds `PublicKey`, `PrivateKey`, `SharedKey`, `Nonce`, `Encryption`, `Byte` or `Boolean`, `Tuple` (we do not need any others) will be `PublicKey`, `PrivateKey`, `SharedKey`, `Encryption`, `Nonce`, `Byte` or `Boolean`, `MessageSeq` respectively. `PrivateKey` and `SharedKey` types will be of `DecryptionKey` kind, while `PublicKey` and `SharedKey` types will be of `EncryptionKey` kind.

Interpretation of `encodeT` will be identity of T , while `decodeT(m)` will be either m for m of type T (since $M = O$ now), or failing otherwise. Thus `decodeT` will act as a type–checker. In case of type `MessageSeq`, `createTuple` is the `MessageSeq` constructor while `analyzeTuple` extracts the sequence from a `MessageSeq` object.

$v, \sigma \nearrow \sigma(v).\text{encode}(), \sigma$	
$cr, \sigma \nearrow o.\text{encode}(), \sigma + \{cr \mapsto o\}$	$cr \notin \sigma, o = cr.\text{create}()$
$r, \sigma \nearrow \rho(r), \sigma$	
$c : C, \sigma \nearrow C.\text{encode}(), \sigma + \{c \mapsto C\}$	$c \notin \sigma$
$k : \text{op}(K), \sigma \nearrow o_k.\text{encode}(), \sigma + \{k \mapsto o_k\}$	$k \notin \sigma, o_k = \sigma(K).\text{op}()$
$\frac{k : \text{op}(K), \sigma + \{K \mapsto K.\text{create}()\} \nearrow m, \sigma'}{k : \text{op}(\sigma(K)), \sigma \nearrow m, \sigma'}$	$k \notin \sigma, K \notin \sigma$
$k : \text{keyOf}(a), \sigma \nearrow m_k, \sigma + \{k \mapsto k.\text{decode}(m_k)\}$	$k \notin \sigma, m_k = \nu(\sigma(a).\text{encode}())$
$\frac{kp, \sigma, \nu \nearrow_K o_k, \sigma' \quad p, \sigma' \nearrow m, \sigma''}{e : \text{E}(kp, p), \sigma \nearrow m_e, \sigma'' + \{e \mapsto e.\text{decode}(m_e)\}}$	$e \notin \sigma, m_e = o_k.\text{encrypt}(m)$
$\frac{kp, \sigma, \nu \nearrow_K o_k, \sigma'}{s : \text{S}(kp, r), \sigma \nearrow m_s, \sigma' + \{s \mapsto s.\text{decode}(m_s)\}}$	$s \notin \sigma, m_s = o_k.\text{sign}(\rho(r))$
$h : \text{H}(r), \sigma \nearrow m_h, \sigma + \{h \mapsto h.\text{decode}(m_h)\}$	$h \notin \sigma, m_h = h.\text{hash}(\rho(r))$
$\frac{\dots p_i, \sigma_{i-1} \nearrow m_i, \sigma_i \dots}{t : [p_1, \dots, p_n], \sigma \nearrow o_t.\text{encode}(), \sigma_n + \{t \mapsto o_t\}}$	$t \notin \sigma, \sigma_0 = \sigma, \rho_0 = \rho,$ $o_t = t.\text{createTuple}(\vec{m})$
$\frac{p_1, \sigma, \nearrow m', \sigma', \quad p_2, \sigma', \rho + \{r \mapsto m'\} \nearrow m'', \sigma''}{\text{let } r = p_1 \text{ in } p_2, \sigma, \nearrow m'', \sigma''}$	$r \notin \rho$
$v_k, \sigma, \nu \nearrow_K \sigma(v_k), \sigma$	$v_k = K, k, sh$
$v_k, \sigma, \nu \nearrow_K o, \sigma + \{v_k \mapsto o\}$	$o = v_k.\text{create}(), v_k = K, sh$
$\frac{K, \sigma, \nu \nearrow_K o_K, \sigma'}{k : \text{op}(K), \sigma, \nu \nearrow_K o_k, \sigma' + \{k \mapsto o_k\}}$	$k \notin \sigma, o_k = o_K.\text{op}()$
$k : \text{keyOf}(a), \sigma, \nu \nearrow_K o_k, \sigma + \{k \mapsto o_k\}$	$k \notin \sigma, o_k = \nu(\sigma(a).\text{encode}()).\text{decode}()$

Fig. 3. Creating messages through patterns

The create methods, where they are needed (in cases of types `PrivateKey`, `SharedKey` and `Nonce`) are implemented by appropriate default constructors.

Under these provisos, as the reader can easily check, the present rules work exactly like the rules of [RRS03], if we disregard the nondeterminism of the tuple-matching rule there, which is not needed for the results of [RRS03].

3 The Cryptographic Abstract Machine

The Cryptographic Abstract Machine is a simple machine with a program, storage, accumulator, input and output. Its storage is decomposed into stores σ , ρ and ν .

More formally, a Cryptographic Abstract Machine is a tuple

$$\mathcal{M} = (\text{prog}, \text{pc}, \sigma, \rho, \nu, \text{in}, \text{out}, \text{acc})$$

instruction	update set	condition
INPUT	$\{\text{in} := \text{null}, \text{acc} := \text{in}\}$	$\text{in} \neq \text{null}$
OUTPUT	$\{\text{out} := \text{acc}\}$	$\text{out} = \text{null}$
LOAD r	$\{\text{acc} := \rho(r)\}$	
STORE r	$\{\rho(r) := \text{acc}\}$	
STORE CONST $c C$	$\{\sigma(c) := C\}$	
DECODE v	$\{\sigma(v) := v.\text{decode}(\text{acc})\}$	
ENCODE v	$\{\text{acc} := \sigma(v).\text{encode}()\}$	
CREATE cr	$\{\sigma(cr) := cr.\text{create}()\}$	
ANALYZE TUPLE $t \vec{r}$	$\{\rho(r_i) := \sigma(t).\text{analyzeTuple}() \mid r_i \in \vec{r}\}$	
CREATE TUPLE $t \vec{r}$	$\{\sigma(t) := t.\text{createTuple}([\rho(r_i) \mid r_i \in \vec{r}])\}$	
OPPOSITE $K k$	$\{\sigma(k) := \sigma(K).\text{op}()\}$	
KEY OF	$\{\text{acc} := \nu(\text{acc})\}$	
MATCH r	\emptyset	$\text{acc} = \rho(r)$
ENCRYPT ek	$\{\text{acc} := \sigma(ek).\text{encrypt}(\text{acc})\}$	
DECRYPT dk	$\{\text{acc} := \sigma(dk).\text{decrypt}(\text{acc})\}$	
SIGN sk	$\{\text{acc} := \sigma(sk).\text{sign}(\text{acc})\}$	
VERIFY $vk r$	\emptyset	$\sigma(vk).\text{verify}(\rho(r), \text{acc})$
HASH h	$\{\text{acc} := h.\text{hash}(\text{acc})\}$	

Fig. 4. CrAM instructions

where the stores σ, ρ, ν are as in Section 2.1, pc is a program counter holding a nonnegative integer, acc holds a message, and in, out hold either messages or the null object null . The program prog is a sequence of instructions defined in figure 4. Most instructions address variables of some concrete type, what provides them with a specific semantics. It is through the choice of concrete types that the domains of cryptographic objects and messages processed by the machine, as well as cryptographic algorithms used, are determined: we can (and do) run the same CrAM programs both with the entirely abstract model of encryption of [RRS03], and with concrete cryptographic APIs, see [Web].

Executing an instruction, if possible, transforms

$$\mathcal{M} = (\text{prog}, \text{pc}, \sigma, \rho, \nu, \text{in}, \text{out}, \text{acc})$$

to some

$$\mathcal{M}' = (\text{prog}, \text{pc}+1, \sigma', \rho', \nu, \text{in}', \text{out}', \text{acc}')$$

by executing the updates associated to each instruction below. There are no branches or jumps in the CrAM, since it is concerned just with analysis and

synthesis of messages. If needed, they can be easily added to the machine, or the machine as defined here can be used as a submachine of a machine with some control flow mechanisms. Note that there is no ν' : in our present model the certificate store is constant.

Executing an instruction can be impossible for one of the following reasons:

1. An explicit side condition of the instruction does not hold.
2. An expression occurring in the updates or in the side conditions cannot be executed because a method is not defined at an argument.

In any of these cases we imagine the machine has failed, aborted (in our implementation we raise an exception).

The fact that one step of execution successfully transforms \mathcal{M} to \mathcal{M}' as above will be denoted by $\mathcal{M} \rightarrow \mathcal{M}'$; the reflexive transitive closure of \rightarrow will be denoted by $\xrightarrow{*}$.

Since some of the operations used in the semantics of CrAM instructions can be nondeterministic, \rightarrow and $\xrightarrow{*}$ are not functions, they are in general just relations.

In the *initial* state of a CrAM we have $\text{pc} = 0$; it is *terminated* if $\text{pc} = |\text{prog}|$, i.e. it has executed its entire program without failure. A CrAM in an initial state *terminates* if there is a terminated CrAM \mathcal{M}' such that $\mathcal{M} \xrightarrow{*} \mathcal{M}'$; we shall say that it terminates with m', σ', ρ' if the value stored in the accumulator is m' , and the object store and the message store of \mathcal{M}' are respectively σ', ρ' . When m' is not of interest, it will be skipped.

The syntax and the semantics of individual instructions is displayed in figure 4, where the first column provides the syntax, the second column the semantics of instruction execution, given by a set of ASM updates, in terms of $\sigma, \rho, \nu, \text{in}, \text{out}, \text{acc}$, and the third column the explicit side condition, if any, under which the instruction can execute.

4 Compiling Message Patterns to CrAM

In this section we show how to compile message patterns to CrAM programs, both for matching and for creation. In view of the representation of protocol roles of [RRS03] as sequences of match–create pairs of patterns, we obtain compilation of protocol roles to CrAM code automatically, by pasting and glueing with instructions for input–output.

The intuition, expressed in [RRS03] by saying

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).

gets a concrete form in the CrAM, which is now also decoupled from any specific model of encryption, and works with any reasonable model supported by the vocabulary and assumptions.

pattern	instructions
$v \ (v \in \sigma)$	[STORE r^s , ENCODE v , MATCH r^s]
$v \ (v \notin \sigma)$	[DECODE v]
$r \ (r \in \rho)$	[MATCH r]
$r \ (r \notin \rho)$	[STORE r]
$c : C$	[STORE r^s , STORE CONST $c \ C$, ENCODE c , MATCH r^s]
$k : \text{op}(K)$	[STORE r_s , OPPOSITE $K \ k$, ENCODE k , MATCH r_s]
$k : \text{keyOf}(a)$	[DECODE k , STORE r^s , ENCODE a , KEY OF, MATCH r^s]
$e : E(dk, p)$	[DECODE e , DECRYPT dk] + $\text{compile}\downarrow(p, \sigma + \{e\}, \rho)$
$s : S(vk, r)$	[DECODE s , VERIFY $vk \ r$]
$s : S(k : \text{op}(K), r)$	[DECODE s , OPPOSITE $K \ k$, VERIFY $k \ r$]
$s : S(k : \text{keyOf}(a), r)$	[DECODE s , STORE r^s , ENCODE a , KEY OF, DECODE k , LOAD r^s , VERIFY $k \ r$]
$h : H(r)$	[DECODE h , STORE r^s , LOAD r , HASH h , MATCH r^s]
$t : [p_1, \dots, p_n]$	[DECODE t , ANALYZE TUPLE $t \ \vec{r^s}$] + $\sum_{i=1}^n ([\text{LOAD } r_i^s] + \text{compile}\downarrow(p_i, \sigma_i, \rho_i))$
let $r = p_1$ in p_2	$\text{compile}\downarrow(p_2, \sigma, \rho) + [\text{LOAD } r] + \text{compile}\downarrow(p_1, \sigma + \Delta\sigma_2, \rho + \Delta\rho_2)$

Fig. 5. Compilation of a matching pattern to CrAM instructions

We show that both compilation algorithms are both correct and complete, which means that CrAM programs are at least as strong as patterns, in any specific incarnation of both.

Compilation for Matching The compilation algorithm operates in the context of (domains of) current stores σ, ρ , and can be seen as a mapping of form

$$\text{compile}\downarrow(p, \sigma, \rho) \mapsto (\text{prog}, \sigma', \rho')$$

If executing the compiled program would overwrite some variable already in σ or ρ , compilation is assumed to fail. Eg. if one of the variables designated in the patterns, ones shown to the left of ‘:’, is in σ , then compilation fails.

The inductive clauses defining the compilation algorithm are given in the figure 5. Domains σ', ρ' are domains of a storage resulting from successfully executing a program given by the table, using initial storage with domains σ, ρ .

Often the result of compilation involves some *scratch* raw variables, denoted in the code by r^s . By convention, every occurrence of a scratch variable is fresh, occurring nowhere in the pattern, and nowhere else in the compiled code except for the places indicated. This ensures that scratch variables invoked for a part of a pattern cannot interfere with compilation or interpretation of other parts.

pattern	instructions
v ($v \in \sigma$)	[ENCODE v]
cr ($cr \notin \sigma$)	[CREATE cr , ENCODE cr]
r ($r \in \sigma$)	[LOAD r]
$c : C$	[STORE CONST c C , ENCODE c]
$k : \text{op}(K)$	$\text{compile}\uparrow_K(k : \text{op}(K), \sigma) + [\text{ENCODE } k]$
$k : \text{keyOf}(a)$	$\text{compile}\uparrow_K(k : \text{keyOf}(a), \sigma)$
$e : E(kp, p)$	$\text{compile}\uparrow_K(kp, \sigma) + \text{compile}\uparrow(p, \sigma + \Delta\sigma_{kp}) +$ [ENCRYPT $\text{keyVar}(kp)$, DECODE e]
$s : S(kp, r)$	$\text{compile}\uparrow_K(kp, \sigma) + [\text{LOAD } r, \text{SIGN } \text{keyVar}(kp), \text{DECODE } s]$
$h : H(r)$	[LOAD r , HASH h , DECODE h]
$t : [p_1, \dots, p_n]$	$\sum_{i=1}^n (\text{compile}\uparrow(p_i, \sigma_i) + [\text{STORE } r_i^s]) + [\text{CREATE TUPLE } t \vec{r}^s + \text{ENCODE } t]$
let $r = p_1$ in p_2	$\text{compile}\uparrow(p_1, \sigma) + [\text{STORE } r] + \text{compile}\uparrow(p_2, \sigma + \Delta\sigma_1)$

$$\text{keyVar}(kp) = \begin{cases} kp & kp = k, K, sh \\ k & kp = k : \text{op}(K), k : \text{keyOf}(a) \end{cases}$$

$$\text{compile}\uparrow_K(kp, \sigma) \mapsto (prog, \sigma')$$

pattern	instructions	condition
k_v	[]	$(k_v \in \sigma),$ $k_v = K, k, sh$
cr_k	[CREATE cr]	$cr_k = K, sh$
$k : \text{op}(K)$	$\text{compile}\uparrow_K(K, \sigma) + [\text{OPPOSITE } K \ k]$	
$k : \text{keyOf}(a)$	[ENCODE a , KEY OF, DECODE k]	

Fig. 6. Compilation of creation pattern into CrAM instructions

The reader might easily note that the compiled code has the write–once property: if the compiled code is executed with stores whose domains it is compiled with, every variable occurring in the compiled code gets written at most once.

It is also easy to check that neither the rules nor the compiled code for matching invoke any possibly nondeterministic operation; the result of matching, if any, is uniquely determined.

Correctness and completeness of the compilation is established by

Theorem 1. *Let p be a pattern, σ, ρ, ν respective assignments, m a message. Then we have $\sigma, \rho, \nu, p, m \searrow \sigma', \rho'$ if and only if the corresponding CrAM*

$$M = (\text{compile}\downarrow(p, \sigma, \rho, \nu), 0, \sigma, \rho, \nu, \text{null}, \text{null}, m)$$

terminates in a state with σ', ρ'' , where ρ'' is an extension of ρ' with some scratch variables not occurring in p .

The proof proceeds by induction over definition of $\text{compile}\downarrow()$, and consists in tedious, though straightforward examination of cases. The reader can try a few cases herself, or find a reasonably expanded proof in the full version of this paper [Web]. We intend to present a fully machine-verified proof elsewhere.

Compilation for Creation The compilation algorithm can be seen as a mapping of form

$$\text{compile}\uparrow(p, \sigma) \mapsto (\text{prog}, \sigma')$$

under the same conventions as the compilation algorithm for matching. It is presented in figure 6, which should be read in the same way as figure 5. Again, if executing the compiled program would overwrite some variable already in σ , compilation is assumed to fail.

The execution of the compiled program can be nondeterministic, since it could use potentially nondeterministic procedures `create`, `encrypt` and `sign`.

The correctness and completeness of the algorithm is established by

Theorem 2. *Let p be a pattern, σ, ρ, ν appropriate assignments, and m a message. Then we have $p, \sigma, \rho, \nu \nearrow m', \sigma'$ if and only if the corresponding CrAM*

$$M = (\text{compile}\uparrow(p, \sigma, \rho, \nu), 0, \sigma, \rho, \nu, \text{null}, \text{null}, m)$$

can terminate in a state with $\text{acc} = m', \sigma'$ and ρ' , with ρ' extension of ρ with some scratch variables not occurring in p .

The proof also proceeds by induction over definition of $\text{compile}\uparrow()$, and a reasonably expanded proof can be found in the full version of the paper, see [Web].

References

- AR02. M. Abadi and P. Rogaway. Reconciling two views of cryptography. *Journal of Cryptology*, 15(2):103–127, 2002.
- BD96. E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.
- 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.

- BN00. M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of generic composition paradigm. In *ASIACRYPT '00*, volume 1976 of *LNCS*, 2000.
- BR94. E. Börger and D. Rosenzweig. The WAM – Definition and Compiler Correctness. In *Logic Programming: Formal Methods and Practical Applications*, pages 20–90. North-Holland, 1994.
- BR97. Giampaolo Bella and Elvinia Riccobene. Formal analysis of the Kerberos authentication system. *Journal of UCS*, 3(12):1337–1381, 1997.
- 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.
- CDL⁺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.
- DMP03. Nancy Durgin, John Mitchell, and Dusko Pavlovic. A compositional logic for protocol correctness. *Journal of Computer Security*, 11, 2003.
- 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.
- FIP95. NIST. Secure Hash Standard. *FIPS PUB 180-1*, 1995.
- GH93. Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In *CSL 92*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.
- GZ00. G. Goos and W. Zimmermann. Verifying Compilers and ASMs. In *ASM 2000*, volume 1912 of *LNCS*. Springer-Verlag, 2000.
- HS00. J. Huggins and W. Shen. The Static and Dynamic Semantics of C. Technical Report CPSC-2000-4, Kettering University, 2000.
- Low96. Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055 of *LNCS*, 1996.
- MW03. D. Micciancio and B. Warinschi. Completeness theorems for the Abadi-Rogaway logic of encrypted expressions. *Journal of Computer Security*, 2003.
- PKC93. RSA Laboratories. *PKCS #7, Version 2.10*, November, 1993.
- PKC02. RSA Laboratories. *PKCS #1, Version 2.1*, June, 2002.
- Riv92. R. Rivest. The MD5 message-digest algorithm. *RFC 1321*, 1992.
- RRS03. D. Rosenzweig, D. Runje, and N. Slani. Privacy, abstract encryption and protocols: an ASM model – Part I. In *ASM 2003*, volume 2589 of *LNCS*. Springer-Verlag, 2003.
- SSB01. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
- Wal95. C. Wallace. The Semantics of the C++ Programming Language. In *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.
- War03. B. Warinschi. A computational analysis of the Needham-Schroeder(-Lowe) protocol. In *Proceedings of 16th IEEE CSFW*. ACM Press, 2003.
- Web. <http://www.fsb.hr/matematika/index.php?ulaz=protocols>.

A Appendix: The Proofs

A.1 Compilation for Matching

Theorem 1. *Let p be a pattern, σ, ρ, ν respective assignments, m a message. Then we have $\sigma, \rho, \nu, p, m \searrow \sigma', \rho'$ if and only if the corresponding CrAM*

$$M = (\text{compile}\downarrow(p, \sigma, \rho, \nu), 0, \sigma, \rho, \nu, \text{null}, \text{null}, m)$$

terminates in a state with σ', ρ' , where ρ' is an extension of ρ with some scratch variables not occurring in p .

Proof. By simple induction, over the recursive structure of $\text{compile}\downarrow()$. The induction step splits into several cases, those of $\text{compile}\downarrow()$ definition. In each case we shall show both directions simultaneously.

Case $v \in \sigma$. We have $\sigma, \rho, v, m \searrow \sigma, \rho$ under the condition $\sigma(v).\text{encode}() = m$. But then $\text{compile}\downarrow(v, \sigma, \rho, \nu) = [\text{STORE } r^s, \text{ENCODE } v, \text{MATCH } r^s]$ which executes as follows (we shall only show σ, ρ, acc)

$$\begin{array}{llll} \sigma & \rho & m & \text{initially} \\ \sigma \ \rho + \{r^s \mapsto m\} & & m & \text{by STORE } r^s \\ \sigma \ \rho + \{r^s \mapsto m\} \ \sigma(v).\text{encode}() & & & \text{by ENCODE } v \end{array}$$

and the subsequent $\text{MATCH } (r^s)$ can execute under the condition that $(\rho + \{r^s \mapsto m\})(r^s) = m = \sigma(v).\text{encode}()$.

Case $v \notin \sigma$. We have $\sigma, \rho, v, m \searrow \sigma + \{v \mapsto v.\text{decode}(m)\}, \rho$. But then $\text{compile}\downarrow(v, \sigma, \rho, \nu) = [\text{DECODE } v]$ which executes as follows

$$\begin{array}{llll} \sigma & \rho & m & \text{initially} \\ \sigma + \{v \mapsto v.\text{decode}(m)\} \ \rho & & m & \text{by DECODE } v \end{array}$$

Case $r \in \rho$. We have $\sigma, \rho, r, m \searrow \sigma, \rho$ under the condition $\rho(r) = m$. But then $\text{compile}\downarrow(r, \sigma, \rho, \nu) = [\text{MATCH } r]$, which can execute (changing nothing but the pc) under the condition that $\rho(r) = m$.

Case $r \notin \rho$. We have $\sigma, \rho, v, m \searrow \sigma, \rho + \{r \mapsto m\}$. But then $\text{compile}\downarrow(r, \sigma, \rho, \nu) = [\text{STORE } r]$ which executes as follows

$$\begin{array}{llll} \sigma & \rho & m & \text{initially} \\ \sigma \ \rho + \{r \mapsto m\} & & m & \text{by STORE } r \end{array}$$

Case $c : C$. We have $\sigma, \rho, c : C, m \searrow \sigma + \{c \mapsto C\}, \rho$ under the condition that $c \notin \sigma$ and $c.\text{decode}(m) = C$. But then $\text{compile}\downarrow(c : C, \sigma, \rho, \nu)$ succeeds only if $c \notin \sigma$, and the analysis is entirely parallel to that of case $v \notin \sigma$.

Case $k : \text{op}(K)$. We have $\sigma, \rho, k : \text{op}(K), m \searrow \sigma + \{k \mapsto k.\text{decode}(m)\}, \rho$ under the condition that $k \notin \sigma, K \in \sigma$ and $k.\text{decode}(m) = \sigma(K).\text{op}()$. But then $\text{compile}\downarrow(k : \text{op}(K), \sigma, \rho, \nu)$ succeeds only if $k \notin \sigma$, and the analysis is entirely

parallel to that of case $c : C$, noting that the resulting code can only execute if also $K \in \sigma$.

Case $k : \text{keyOf}(a)$. We have $\sigma, \rho, k : \text{keyOf}(a), m \searrow \sigma + \{k \mapsto k.\text{decode}(m)\}, \rho$ under the condition that $k \notin \sigma, a \in \sigma$ and $m = \nu(\sigma(a).\text{encode}())$. But then $\text{compile}\downarrow(k : \text{keyOf}, \sigma, \rho, \nu)$ succeeds only if $k \notin \sigma$, and the analysis is entirely parallel to that of case $k : \text{op}(K)$, noting that the resulting code can only execute if also $a \in \sigma$.

Case $e : E(dk, p)$. Note that it can match only if $dk \in \sigma, e \notin \sigma$. Then we have $\sigma, \rho, e : E(dk, p), m \searrow \sigma + \{e \mapsto e.\text{decode}(m)\}, \rho, p, \sigma(dk).\text{decrypt}(M) \searrow \sigma', \rho'$. By inductive assumption the latter condition is satisfied if and only if the machine

$$M_p = (\text{prog}_p, 0, \sigma', \rho, \nu, \text{null}, \text{null}, \sigma(dk).\text{decrypt}(m))$$

where $\sigma' = \sigma + \{e \mapsto e.\text{decode}(m)\}$ and $\text{prog}_p = \text{compile}\downarrow(p, \sigma', \rho, \nu)$, terminates in a state with σ'', ρ'' .

But then $\text{compile}\downarrow(e : E(dk, p), \sigma, \rho, \nu)$ succeeds only if $e \notin \sigma$, yielding code which starts to execute as follows

$$\begin{array}{llll} \sigma & \rho & m & \text{initially} \\ \sigma' & \rho & m & \text{by DECODE } e \\ \sigma' & \rho & \sigma(dk).\text{decrypt}(m) & \text{by DECRYPT } dk. \end{array}$$

The last instruction can only execute if $dk \in \sigma$. Thus prog_p takes over precisely in the state needed by the inductive assumption.

Case $s : S(vk, r)$. Note that it can match only if $vk \in \sigma, r \in \rho, s \notin \sigma$. Then $\sigma, \rho, s : S(vk, r), m \searrow \sigma + \{s \mapsto s.\text{decode}(m)\}, \rho$ under the condition that $\sigma(vk).\text{verify}(\rho(r), m)$ is true. But then $\text{compile}\downarrow(e : S(vk, r), \sigma, \rho, \nu)$ succeeds only if $s \notin \sigma$, yielding code which starts to execute as follows

$$\begin{array}{llll} \sigma & \rho & m & \text{initially} \\ \sigma + \{s \mapsto s.\text{decode}(m)\} & \rho & m & \text{by DECODE } e. \end{array}$$

The subsequent **VERIFY** $vk \ r$ can execute (changing nothing but the pc) only if $vk \in \sigma, r \in \rho$, and then $\sigma(vk).\text{verify}(\rho(r), m)$ is true.

Case $s : S(k : \text{op}(K), r)$. The key-fetching rules can work for this case only if $k \notin \sigma, K \in \sigma$, and in that case we have $\sigma, \rho, s : S(k : \text{op}(K), r), m \searrow \sigma + \{k \mapsto \sigma(K).\text{op}(), s \mapsto s.\text{decode}(m)\}, \rho$ under the condition that $s \notin \sigma, r \in \rho$ and further $\sigma(k).\text{verify}(\rho(r), m)$ is true. But then $\text{compile}\downarrow(e : S(k : \text{op}(K), r), \sigma, \rho, \nu)$ succeeds only if $s, k \notin \sigma$, yielding code which starts to execute as follows.

$$\begin{array}{llll} \sigma & \rho & m & \text{initially} \\ \sigma + \{s \mapsto s.\text{decode}(m)\} & \rho & m & \text{by DECODE } e \\ \sigma + \{s \mapsto s.\text{decode}(m), k \mapsto \sigma(K).\text{op}()\} & \rho & m & \text{by OPPOSITE } K \ k \end{array}$$

The rest of the analysis is the same as for the previous case, noting that the last instruction above can only execute if $K \in \sigma$.

Case $s : S(k : \text{keyOf}(a), r)$. The key-fetching rules can work for this case only if $k \notin \sigma, a \in \sigma$, and in that case we have $\sigma, \rho, s : S(k : \text{keyOf}(a), r), m \searrow \sigma', \rho$, where $\sigma' = \sigma + \{k \mapsto k.\text{decode}(\nu(\sigma(a).\text{encode()})), s \mapsto s.\text{decode}(m)\}$, under the condition that $s \notin \sigma, r \in \rho$ and further $\sigma(k).\text{verify}(\rho(r), m)$ is true. But then $\text{compile}\downarrow(e : S(k : \text{keyOf}(a), r), \sigma, \rho, \nu)$ succeeds only if $s, k \notin \sigma$, yielding code which starts to execute as follows.

σ	ρ	m	initially
$\sigma + \{s \mapsto s.\text{decode}(m)\}$	ρ	m	by DECODE e
$\sigma + \{s \mapsto s.\text{decode}(m)\}$	$\rho + \{r^s \mapsto m\}$	m	by STORE r^s
$\sigma + \{s \mapsto s.\text{decode}(m)\}$	$\rho + \{r^s \mapsto m\}$	$\sigma(a).\text{encode}()$	by ENCODE a
$\sigma + \{s \mapsto s.\text{decode}(m)\}$	$\rho + \{r^s \mapsto m\}$	$\nu(\sigma(a).\text{encode}())$	by KEY OF
σ'	$\rho + \{r^s \mapsto m\}$	$\nu(\sigma(a).\text{encode}())$	by DECODE k
σ'	$\rho + \{r^s \mapsto m\}$	m	by LOAD r^s

The rest of the analysis is entirely parallel to the two previous cases, noting that the rest of the compiled code can only execute if $a \in \sigma, r \in \rho$.

Case $h : H(r)$. We have $\sigma, \rho, h : H(r), m \searrow \sigma + \{h \mapsto h.\text{decode}(m)\}, \rho$, under the condition that $h \notin \sigma$ and $h.\text{hash}(\rho(r)) = m$. But then $\text{compile}\downarrow(h : H(r), \sigma, \rho, \nu)$ succeeds only if $h \notin \sigma$, yielding code which starts to execute as follows.

σ	ρ	m	initially
σ'	ρ	m	by DECODE h
σ'	$\rho + \{r^s \mapsto m\}$	m	by STORE r^s
σ'	$\rho + \{r^s \mapsto m\}$	$\rho(r)$	by LOAD r
σ'	$\rho + \{r^s \mapsto m\}$	$h.\text{hash}(\rho(r))$	by HASH h

where $\sigma' = \sigma + \{h \mapsto h.\text{decode}(m)\}$. The LOAD r instruction shown can only execute if $r \in \rho$, and the ultimate MATCH r^s instruction can execute only if $h.\text{hash}(\rho(r)) = m$.

Case $t : [p_1, \dots, p_n]$. We have $\sigma, \rho, t : [p_1, \dots, p_n], m \searrow \sigma_n, \rho_n$, given that $t \notin \sigma$, $\sigma_0 = \sigma + \{t \mapsto t.\text{decode}(m)\}, \rho_0 = \rho$, and $\sigma_{i-1}, \rho_{i-1}, p_i, m_i \searrow \sigma_i, \rho_i$, for $i = 1, \dots, n$, where each m_i is the i -th component of $t.\text{decode}(m).\text{analyzeTuple}()$. But then, by induction hypothesis, the machines

$$M_i = (\text{prog}_i, 0, \sigma_{i-1}, \rho_{i-1}^+, \nu, \text{null}, \text{null}, m_i)$$

terminate with σ_i, ρ_i^+ if and only if the premisses of the rule are satisfied (where $\text{prog}_i = \text{compile}\downarrow(p_i, \sigma_{i-1}, \rho_{i-1}^+$, and ρ_i^+ is ρ_i extended by some scratch variables). But then $\text{compile}\downarrow(t : [p_1, \dots, p_n], \sigma, \rho)$ succeeds only if $t \notin \sigma$, and the resulting

code executes as follows.

σ	ρ	m	initially
σ_0	ρ	m	by DECODE t
σ_0	$\rho + \{r_i^s \mapsto m_i\}$	m	by ANALYZE TUPLE $t \ \vec{r}$
σ_0	$\rho + \{r_i^s \mapsto m_i\}$	m_1	by LOAD r_1^s
\dots			
σ_1	ρ_1^+	m'_1	by i.h.forprog ₁
\dots			
σ_{n-1}	ρ_{n-1}^+	m_n	by LOAD r_n^s
\dots			
σ_n	ρ_n^+	m'_n	by i.h.forprog _n

Due to our discipline of using always fresh scratch variables, they don't interfere, in particular the scratch variables of ρ_i^+ do not interfere with execution of `progj` for $j > i$.

Case let $r = p_1$ in p_2 . We have σ, ρ , let $r = p_1$ in $p_2, m \searrow \sigma'', \rho''$ given that $\sigma, \rho, p_2, m \searrow \sigma', \rho'$ and then $\sigma', \rho', p_1, \rho'(p_1) \searrow \sigma'', \rho''$. By induction hypothesis the premisses are true if and only if the machines

$$M_2 = (\text{prog}_2, 0, \sigma, \rho, \nu, \text{null}, \text{null}, m)$$

$$M_1 = (\text{prog}_1, 0, \sigma', \rho', \nu, \text{null}, \text{null}, m)$$

terminate with σ', ρ'^+ , and σ'', ρ''^+ respectively, where `progi` is the compilation of p_i in appropriate environments. But then `compile↓(let $r = p_1$ in p_2, σ, ρ)` executes as follows:

σ	ρ	m	initially
\dots			
σ'	ρ'^+	m'	by i.h.forprog ₂
σ'	ρ'^+	$\rho'(r)$	by LOAD r
\dots			
σ''	ρ''^+	m''	by i.h.forprog ₁

A.2 Compilation for Creation

Lemma 1. *Let kp be a key pattern and σ, ν appropriate assignments. Then we have $kp, \sigma, \nu \nearrow_K o_k, \sigma'$ if and only if the corresponding CrAM*

$$M = (\text{compile}\uparrow_K(kp, \sigma), 0, \sigma, \rho, \nu, \text{null}, \text{null}, m)$$

terminates in a state with $\text{acc} = m', \sigma'$ and ρ , with $\sigma'(\text{keyVar}(kp)) = o_k$.

Proof. Proof is by induction on the definition of `compile` \uparrow_K .

Case $k_v \in \sigma$. We have `compile` $\uparrow_K(k_v, \sigma) = []$. The CrAM always terminates in the initial state, and $\sigma(\text{keyVar}(kp)) = \sigma(k_v)$ while $k_v, \sigma \nearrow_K \sigma(k_v), \sigma$.

Case $cr_k \notin \sigma$. We have `compile` $\uparrow_K(cr_k, \sigma) = [\text{CREATE } cr_k]$ and $\Delta\sigma = \{cr_k\}$. Hence $cr_k \notin \sigma$.

The CrAM always terminates with $\sigma' = \sigma + \{cr_k \mapsto o_k\}$ and $\sigma'(\text{keyVar}(kp)) = \sigma'(cr_k) = o_k$, where $o_k = cr_k.\text{create}()$. We also have $cr_k, \sigma \nearrow_K o_k, \sigma + \{cr_k \mapsto o_k\}$ since $cr_k \notin \sigma$.

Case $k : \text{op}(K)$. We have

$$\text{compile}\uparrow_K(k : \text{op}(K), \sigma) = \text{compile}\uparrow_K(K) + [\text{OPPOSITE } K \ k].$$

From $k \in \Delta\sigma$ we have $k \notin \sigma$.

The CrAM always terminates, since $K \in \sigma'$ by induction hypothesis, with $\sigma'' = \sigma' + \{k \mapsto o_k\}$ and $\sigma''(\text{keyVar}(kp)) = \sigma''(k) = o_k$, where $o_k = o_K.\text{op}()$, after the following execution:

m	σ	ρ	initially
m	σ'	ρ	by $\text{compile}\uparrow_K(K, \sigma)$
m	$\sigma' + \{k \mapsto o_k\}$	ρ	by $\text{OPPOSITE } K \ k$

By the induction hypothesis, $K, \sigma \nearrow_K o_K, \sigma'$ and $o_K = \sigma'(K)$. From $k \notin \sigma$ we have $k \notin \sigma'$ and, finally, $k : \text{op}(K), \sigma \nearrow_K o_k, \sigma' + \{k \mapsto o_k\}$.

Case $k : \text{keyOf}(a)$. We have

$$\text{compile}\uparrow_K(k : \text{keyOf}(a), \sigma) = [\text{ENCODE } a, \text{KEY OF}, \text{DECODE } k].$$

From $k \in \Delta\sigma$ we have $k \notin \sigma$.

The CrAM terminates, if and only if o_k is defined, with $\sigma' = \sigma + \{k \mapsto o_k\}$ and $\sigma'(\text{keyVar}(kp)) = \sigma'(k) = o_k$, where $o_k = k.\text{decode}(\nu(\sigma(a).\text{encode}()))$, after the following execution:

m	σ	ρ	initially
$\sigma(a).\text{encode}()$	σ	ρ	by $\text{ENCODE } a$
$\nu(\sigma(a).\text{encode}())$	σ	ρ	by KEY OF
$\nu(\sigma(a).\text{encode}())$	$\sigma + \{k \mapsto o_k\}$	ρ	by $\text{DECODE } k$

By definition, $k : \text{keyOf}(a), \sigma \nearrow_K o_k, \sigma + \{k \mapsto o_k\}$ if and only if o_k is defined.

Theorem 2. *Let p be a pattern, σ, ρ, ν appropriate assignments, and m a message. Then we have $p, \sigma, \rho, \nu \nearrow m', \sigma'$ if and only if the corresponding CrAM*

$$M = (\text{compile}\uparrow(p, \sigma, \rho, \nu), 0, \sigma, \rho, \nu, \text{null}, \text{null}, m)$$

terminates with m', σ' and ρ' , with ρ' extension of ρ with some scratch variables not occurring in p .

Proof. The proof is by induction over the definition of $\text{compile}\uparrow$.

Case $v \in \sigma$. We have $\text{compile}\uparrow(v, \sigma) = [\text{ENCODE } v]$. The CrAM always terminates, with $\sigma(v).\text{encode}(), \sigma, \rho$, after executing the single instruction, while $v, \sigma, \rho, \nu \nearrow \sigma(v).\text{encode}(), \sigma$.

Case cr . We have $\text{compile}\uparrow(cr, \sigma) = [\text{CREATE } , \text{ENCODE } cr]$, with $\Delta\sigma = \{cr\}$. Hence $cr \notin \sigma$.

The CrAM always terminates, with $o_{cr}.\text{encode}(), \sigma = \{cr \mapsto o_{cr}\}, \rho$, where $o_{cr} = cr.\text{create}()$, after the following execution:

m	σ	ρ	initially
m	$\sigma + \{cr \mapsto o_{cr}\}$	ρ	by CREATE cr
$o_{cr}.\text{encode}()$	$\sigma + \{cr \mapsto o_{cr}\}$	ρ	by ENCODE cr

From $cr \notin \sigma$, we have $cr, \sigma, \rho, \nu \nearrow o_{cr}, \sigma + \{cr \mapsto o_{cr}\}$.

Case $r \in \rho$. We have $\text{compile}\uparrow(r, \sigma) = [\text{LOAD } r]$. The CrAM always terminates, with $\rho(r), \sigma, \rho$, while $r, \sigma, \rho, \nu \nearrow \rho(r), \sigma$.

Case $c : C$. We have $\text{compile}\uparrow(c : C, \sigma) = [\text{STORE CONST } c C, \text{ENCODE } c]$, with $\Delta\sigma = \{c\}$. Hence $c \notin \sigma$. The CrAM always terminates, with $C.\text{encode}(), \sigma + \{c \mapsto C\}, \rho$, while $c : C, \sigma, \rho, \nu \nearrow C.\text{encode}(), \sigma + \{c \mapsto C\}$.

Case $k : \text{op}(K)$, $K \in \sigma$. We have

$$\text{compile}\uparrow(k : \text{op}(K), \sigma) = [\text{OPPOSITE } k K, \text{ENCODE } k],$$

with $\Delta\sigma = \{k\}$. Hence $k \notin \sigma$.

The CrAM always terminates, with $o_k.\text{encode}(), \sigma + \{k \mapsto o_k\}, \rho$, where $o_k = K.\text{op}()$, after the following execution:

m	σ	ρ	initially
m	$\sigma + \{k \mapsto o_k\}$	ρ	by OPPOSITE $k K$
$o_k.\text{encode}()$	$\sigma + \{k \mapsto o_k\}$	ρ	by ENCODE k

From $k \notin \sigma$ and $K \in \sigma$, we have $k : \text{op}(K), \sigma, \rho, \nu \nearrow o_k.\text{encode}(), \sigma + \{k \mapsto o_k\}$.

Case $k : \text{op}(K)$, $K \notin \sigma$. We have

$$\text{compile}\uparrow(k : \text{op}(K), \sigma) = [\text{CREATE } K, \text{OPPOSITE } k K, \text{ENCODE } k],$$

with $\Delta\sigma = \{k, K\}$. Hence $k, K \notin \sigma$.

The CrAM always terminates, with $o_k.\text{encode}(), \sigma + \{k \mapsto o_k\} + \{K \mapsto o_K\}, \rho$, where $o_k = o_K.\text{op}()$, $o_K = K.\text{create}()$ after the following execution:

m	σ	ρ	initially
m	$\sigma + \{K \mapsto o_K\}$	ρ	by CREATE K
m	$\sigma + \{K \mapsto o_K\} + \{k \mapsto o_k\}$	ρ	by OPPOSITE $k K$
$o_k.\text{encode}()$	$\sigma + \{K \mapsto o_K\} + \{k \mapsto o_k\}$	ρ	by ENCODE k

From $k, K \notin \sigma$, we have

$$k : \text{op}(K), \sigma + \{K \mapsto o_K\}, \rho, \nu \nearrow o_k.\text{encode}(), \sigma + \{K \mapsto o_K\} + \{k \mapsto o_k\},$$

and finally $k : \text{op}(K), \sigma, \rho, \nu \nearrow o_k.\text{encode}(), \sigma + \{K \mapsto o_K\} + \{k \mapsto o_k\}$.

Case $k : \text{keyOf}(a)$. We have

$$\text{compile}\uparrow(k : \text{keyOf}(a), \sigma) = [\text{ENCODE } a, \text{KEY OF}, \text{DECODE } k],$$

with $\Delta\sigma = \{k\}$. Hence $k \notin \sigma$.

The CrAM terminates if and only if $k.\text{decode}(m_k)$ is defined, with $m_k, \sigma + \{k \mapsto k.\text{decode}(m_k)\}, \rho$, where $m_k = \nu(\sigma(a).\text{encode}())$, after the following execution:

m	σ	ρ	initially
$\sigma(a).\text{encode}()$	σ	ρ	by ENCODE a
m_k	σ	ρ	by KEY OF
m_k	$\sigma + \{k \mapsto k.\text{decode}(m_k)\}$	ρ	by DECODE k

From $k \notin \sigma$, we have $k : \text{keyOf}(a), \sigma \nearrow m_k, \sigma + \{k \mapsto k.\text{decode}(m_k)\}$ if and only if $k.\text{decode}(m_k)$ is defined.

Case $e : \text{E}(kp, p)$. We have

$$\text{compile}\uparrow(e : \text{E}(kp, p), \sigma) = \text{compile}\uparrow_K(kp, \sigma) + \text{compile}\uparrow(p, \sigma') + [\text{ENCRYPT keyVar}(kp), \text{DECODE } e],$$

with $\Delta\sigma = \Delta\sigma_{kp} + \Delta\sigma_p + \{e\}$. Hence $e \notin \sigma$.

The CrAM terminates with $m_e, \sigma'' + \{e \mapsto e.\text{decode}(m_e)\}, \rho'$, where $m_e = o_k.\text{encrypt}(m_{sub})$, after the following execution:

m	σ	ρ	initially
m'	σ'	ρ	by $\text{compile}\uparrow_K(kp)$
m_{sub}	σ''	ρ'	by $\text{compile}\uparrow(p)$
m_e	σ''	ρ'	by ENCRYPT keyVar(kp)
m_e	$\sigma'' + \{e \mapsto e.\text{decode}(m_e)\}$	ρ'	by DECODE e

By lemma 1, $(\text{compile}\uparrow_K(kp), \sigma, \rho, \nu, \text{in}, \text{out}, m)$ terminates with m', σ' if only if $kp, \sigma, \nu \nearrow_K o_k, \sigma'$, where $\sigma'(\text{keyVar}(kp)) = o_k$.

By the induction hypothesis, $(\text{compile}\uparrow(p), \sigma', \rho, \nu, \text{in}, \text{out}, m')$ terminates with m_{sub}, σ'', ρ' if only if $p, \sigma', \rho, \nu \nearrow m_{sub}, \sigma''$.

Finally, since $e \notin \sigma$, $(\text{compile}\uparrow(e : \text{E}(kp, p), \sigma, \rho, \nu, \text{in}, \text{out}, m))$ terminates with $m_e, \sigma'' + \{e \mapsto e.\text{decode}(m_e)\}, \rho'$ if only if $e : \text{E}(kp, p), \sigma \nearrow m_e, \sigma'' + \{e \mapsto \text{decode}(m_e)\}$.

Case $s : \text{S}(kp, r)$. We have

$$\text{compile}\uparrow(s : \text{S}(kp, r), \sigma) = \text{compile}\uparrow_K(kp) + [\text{LOAD } r, \text{ENCRYPT keyVar}(kp), \text{DECODE } s],$$

with $\Delta\sigma = \Delta\sigma_{kp} + \{s\}$. Hence $s \notin \sigma$.

The CrAM terminates with $m_s, \sigma + \{\text{keyVar}(kp) \mapsto o_k\} + \{s \mapsto s.\text{decode}(m_s)\}, \rho$, where $m_s = o_k.\text{sign}(\rho(r))$, after the following execution:

m	σ	ρ	initially
-----	----------	--------	-----------

m'	σ'	ρ	by $\text{compile}\uparrow_K(kp)$
$\rho(r)$	σ'	ρ	by LOAD r
m_s	σ'	ρ	by SIGN $\text{keyVar}(kp)$
m_s	$\sigma' + \{s \mapsto s.\text{decode}(m_s)\}$	ρ	by DECODE s

Again, by lemma 1, $(\text{compile}\uparrow_K(kp), \sigma, \rho, \nu, \text{in}, \text{out}, m)$ terminates with m', σ' if only if $kp, \sigma, \nu \nearrow_K o_k, \sigma'$, where $\sigma'(\text{keyVar}(kp)) = o_k$.

Since $s \notin \sigma$, $(\text{compile}\uparrow(s : S(kp, r), \sigma, \rho, \nu, \text{in}, \text{out}, m))$ terminates with $m_s, \sigma + \{\text{keyVar}(kp) \mapsto o_k\} + \{s \mapsto s.\text{decode}(m_s)\}, \rho$ if and only if

$$s : S(kp, r), \sigma \nearrow m_s, \sigma' + \{s \mapsto s.\text{decode}(m_s)\}.$$

Case $h : H(r)$. We have $\text{compile}\uparrow(h : H(r), \sigma) = [\text{LOAD } r, \text{HASH } h, \text{DECODE } h]$, with $\Delta\sigma = \{h\}$. Hence $h \notin \sigma$.

The CrAM always terminates, with $m_h, \sigma + \{h \mapsto h.\text{decode}(m_h)\}, \rho$, where $m_h = h.\text{hash}(\rho(r))$, after the following execution:

m	σ	ρ	initially
$\rho(r)$	σ	ρ	by LOAD r
m_h	σ	ρ	by HASH h
m_h	$\sigma + \{h \mapsto h.\text{decode}(m_h)\}$	ρ	by DECODE h

By the definition of \nearrow we have $h : H(r), \sigma \nearrow m_h, \sigma + \{h \mapsto h.\text{decode}(m_h)\}$.

Case $t : [p_1, \dots, p_n]$. We have

$$\begin{aligned} \text{compile}\uparrow(t : [p_1, \dots, p_n], \sigma) &= (\Sigma_{i=1}^n \text{compile}\uparrow(p_i) + [\text{STORE } r_i^s]) + \\ &[\text{CREATE TUPLE } t \overrightarrow{r^s}, \text{ENCODE } t], \end{aligned}$$

with $\Delta\sigma = \Sigma_{i=1}^n \Delta\sigma_i + \{t\}$. Hence $t \notin \sigma$.

The CrAM terminates with $o_t.\text{encode}(), \sigma_n + \{t \mapsto o_t\}, \rho_n + \{r_n^s \mapsto m_n\}$, where $o_t = t.\text{createTuple}(\overrightarrow{r^s})$, $\sigma_0 = \sigma$ and $\rho_0 = \rho$, after the following execution:

m	σ	ρ	initially
m_1	σ_1	ρ_1	by $\text{compile}\uparrow(p_1)$
m_1	σ_1	$\rho_1 + \{r_1^s \mapsto m_1\}$	by STORE r_1^s
\vdots			
m_n	σ_n	ρ_n	by $\text{compile}\uparrow(p_n)$
m_n	σ_n	$\rho_n + \{r_n^s \mapsto m_n\}$	by STORE r_n^s
m_n	$\sigma_n + \{t \mapsto o_t\}$	$\rho_n + \{r_n^s \mapsto m_n\}$	by CREATE TUPLE $t \overrightarrow{r^s}$
$o_t.\text{encode}()$	$\sigma_n + \{t \mapsto o_t\}$	$\rho_n + \{r_n^s \mapsto m_n\}$	by ENCODE t

By the induction hypothesis, $(\text{compile}\uparrow(p_1), \sigma, \rho, \nu, \text{in}, \text{out}, m)$ terminates with m_1, σ_1, ρ_1 if only if $p_1, \sigma_0, \rho_0, \nu \nearrow m_1, \sigma_1$ and $(\text{compile}\uparrow(p_i), \sigma_{i-1}, \rho_{i-1} + \{r_i^s \mapsto m_i\}, \nu, \text{in}, \text{out}, m_{i-1})$

terminates with m_i, σ_i, ρ_i if only if $p_i, \sigma_{i-1}, \rho_{i-1} + \{r_i^s \mapsto m_i\}, \nu \nearrow m_i, \sigma_i$ for $i = 2, \dots, n$.

Furthermore, each ρ_i is an extension of ρ with some fresh scratch variables not occurring in a pattern p_i . Hence $p_i, \sigma_{i-1}, \rho + \{r_i^s \mapsto m_i\} \nearrow m_i, \sigma_i$ if and only if $p_i, \sigma_{i-1}, \rho_{i-1} + \{r_i^s \mapsto m_i\} \nearrow m_i, \sigma_i$. Finally, since $t \notin \sigma$, we have $(\text{compile}\uparrow(t : [p_1, \dots, p_n]), \sigma, \rho, \nu, \text{in}, \text{out}, m)$ terminates, with $o_t.\text{encode}(), \sigma_n + \{t \mapsto o_t\}, \rho_n + \{r_n^s \mapsto m_n\}$, if and only if $t : [p_1, \dots, p_n], \sigma \nearrow o_t.\text{encode}(), \sigma_n + \{t \mapsto o_t\}$.

Case let $r = p_1$ in p_2 . We have

$$\text{compile}\uparrow(\text{let } r = p_1 \text{ in } p_2) = \text{compile}\uparrow(p_1) + [\text{STORE } r^s] + \text{compile}\uparrow(p_2).$$

The CrAM terminates with m_2, σ_2, ρ_2 , after the following execution:

m	σ	ρ	initially
m_1	σ_1	ρ_1	by $\text{compile}\uparrow(p_1)$
m_1	σ_1	$\rho_1 + \{r \mapsto m_1\}$	by STORE r
m_2	σ_2	ρ_2	by $\text{compile}\uparrow(p_1)$

By the induction hypothesis, $(\text{compile}\uparrow(p_1), \sigma, \rho, \nu, \text{in}, \text{out}, m)$ terminates with m_1, σ_1, ρ_1 if only if $p_1, \sigma, \rho \nearrow m_1, \sigma_1$ and $(\text{compile}\uparrow(p_2), \sigma_1, \rho_1 + \{r \mapsto m_1\}, \nu, \text{in}, \text{out}, m_1)$ terminates with m_2, σ_2, ρ_2 if only if $p_2, \sigma_1, \rho_1 + \{r \mapsto m_1\} \nearrow m_2, \sigma_2$.

Due to our variable discipline, we have $p_2, \sigma_1, \rho_1 + \{r \mapsto m_1\} \nearrow m_2, \sigma_2$ if and only if $p_2, \sigma_1, \rho + \{r \mapsto m_1\} \nearrow m_2, \sigma_2$, and, finally, $(\text{compile}\uparrow(p_1), \sigma, \rho, \nu, \text{in}, \text{out}, m)$ terminates, with m_2, σ_2, ρ_2 , if and only if let $r = p_1$ in $p_2, \sigma, \rho \nearrow m_2, \sigma_2$.