Cryptographic Protocol Explication and End-Point Projection

Jay McCarthy[⋆] and Shriram Krishnamurthi

Brown University

Abstract. Cryptographic protocols are useful for engineering trust in transactions. There are several languages for describing these protocols, but these tend to capture the communications from the perspective of an individual role. In contrast, traditional protocol descriptions as found in a state of nature tend to employ a whole-protocol description, resulting in an impedance mismatch.

In this paper we present two results to address this gap between human descriptions and deployable specifications. The first is an end-point projection technique that consumes an explicit whole-protocol description and generates specifications that capture the behavior of each participant role. In practice, however, many whole-protocol descriptions contain idiomatic forms of implicit specification. We therefore present our second result, a transformation that identifies and eliminates these implicit patterns, thereby preparing protocols for end-point projection.

Concretely, our tools consume protocols written in our whole-protocol language, WPPL, and generate role descriptions in the cryptographic protocol programming language, CPPL. We have formalized and established properties of the transformations using the Coq proof assistant. We have validated our transformations by applying them successfully to most of the protocols in the SPORE repository.

1 Problem and Motivation

In recent years, there has been a vast growth of services offered via the Web, such as third-party credit-card handling as offered by several banks. There is growing recognition that these services must offer security guarantees by building on existing protocols and techniques that establish such guarantees.

Fig. 1 shows three examples of actual protocols, as found in a state of nature. Fig. 1 (a) is the specification of the Kerberos protocol [21]; (b) is the specification of the Kao Chow protocol from [17]; and (c) is the specification of the Yahalom protocol [7] for the SPORE repository [22].

These specifications contain a description of what each role of the protocol does at each step of the protocol. They say that at each step, some role a sends a message m to another role b, written $a \rightarrow b : m$. However, it is important to understand that this is not what actually happens. In reality, a emits a message m and b receives a message m' that matches the pattern of m. Recognizing this distinction makes apparent the threat of man-in-the-middle attacks and other message mutilation in the network medium. This

[⋆] Current affiliation: Brigham Young University.

S. Jajodia, and J. Lopez (Eds.): ESORICS 2008, LNCS 5283, pp. 533–547, 2008.

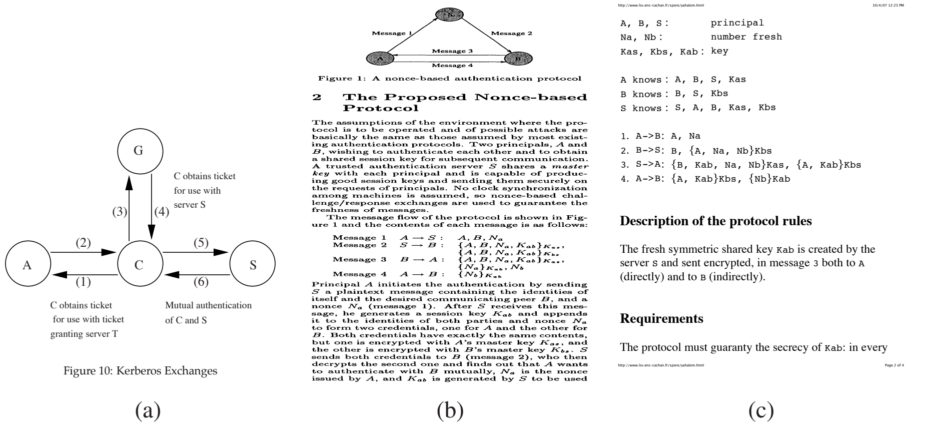
G
A
C
S
(1)
(2)
(3)
(4)
(5)
(6)
C obtains ticket for use with server S
C obtains ticket for use with ticket granting server T
Mutual authentication of C and S
Figure 10: Kerberos Exchanges
(a)
S
A
B
Message 1
Message 2
Message 3
Message 4
(b)
(c)

Figure 1: A nonce-based authentication protocol

2 The Proposed Nonce-based Protocol

The assumptions of the environment where the protocol is to be operated and of possible attacks are basically the same as those assumed by most existing authentication protocols. Two principals, A and B, wishing to authenticate each other and to obtain a shared session key for subsequent communication. A trusted authentication server S shares a *master key* with each principal and is capable of producing good session keys and sending them securely on the requests of principals. No clock synchronization among machines is assumed, so nonce-based challenge/response exchanges are used to guarantee the freshness of messages.

The message flow of the protocol is shown in Figure 1 and the contents of each message is as follows:

Message 1	$A \longrightarrow S$:	A, B, N_a
Message 2	$S \longrightarrow B$:	$\{A, B, N_a, K_{ab}\}_{K_{as}}$, $\{A, B, N_a, K_{ab}\}_{K_{bs}}$
Message 3	$B \longrightarrow A$:	$\{A, B, N_a, K_{ab}\}_{K_{as}}$, $\{N_a\}_{K_{ab}}, N_b$
Message 4	$A \longrightarrow B$:	$\{N_b\}_{K_{ab}}$

Principal A initiates the authentication by sending S a plaintext message containing the identities of itself and the desired communicating peer B, and a nonce N_a (message 1). After S receives this message, he generates a session key K_{ab} and appends it to the identities of both parties and nonce N_a to form two credentials, one for A and the other for B. Both credentials have exactly the same contents, but one is encrypted with A's master key K_{as}, and the other is encrypted with B's master key K_{bs}. S sends both credentials to B (message 2), who then decrypts the second one and finds out that A wants to authenticate with B mutually, N_a is the nonce issued by A, and K_{ab} is generated by S to be used

http://www.lsv.ens-cachan.fr/spore/yahalom.html 10/4/07 12:23 PM

```
A, B, S :        principal
Na, Nb :         number fresh
Kas, Kbs, Kab : key

A knows : A, B, S, Kas
B knows : B, S, Kbs
S knows : S, A, B, Kas, Kbs

1. A->B: A, Na
2. B->S: B, {A, Na, Nb}Kbs
3. S->A: {B, Kab, Na, Nb}Kas, {A, Kab}Kbs
4. A->B: {A, Kab}Kbs, {Nb}Kab
```

Description of the protocol rules

The fresh symmetric shared key `Kab` is created by the server `S` and sent encrypted, in message `3` both to `A` (directly) and to `B` (indirectly).

Requirements

The protocol must guaranty the secrecy of `Kab`: in every

http://www.lsv.ens-cachan.fr/spore/yahalom.html Page 2 of 4

Fig. 1. Protocols in the wild

is called the Dolev-Yao network model [12]. The role of a cryptographic protocol is to describe a sequence of messages that accomplishes some goals—perhaps exchanging data or creating a logical session—even when attacked by a powerful adversary.

All three of the protocols in Fig. 1, and most others like them found in the literature and in repositories, have two characteristics in common:

1. They describe the entire protocol at once, whether diagrammatically (akin to a message sequence chart) or in an equivalent textual format.
2. They have certain idiomatic forms of implicit specification (see below and Sec. 5).

The whole-protocol nature of description is problematic because ultimately, what executes is not the "protocol" per se, but rather various software and hardware devices each implementing the individual roles. Therefore, we need a way to automatically obtain a description of a single role's behavior from this whole-protocol description.

The problem of obtaining individual roles from a composite description is familiar. In the realm of Web Services, is is common to *choreograph* a collection of roles by presenting a similar whole-protocol description. This has led to the definition of *end-point projection* [9], a process of obtaining descriptions of individual role behaviors from the choreography.

Unfortunately, we cannot directly lift the idea of end-point projection to the cryptographic realm because existing descriptions of end-point projection do not handle the complexities introduced by cryptography. If role A sends a message encrypted with key K, then role B can only receive the contents of that message if he has key K. Cryptography introduces information asymmetries like this into communication: it is not always the case that what one role can send, another role can receive. Existing end-point projection systems assume that such asymmetries do not exist. These systems thus focus on communication patterns and neglect communication content. In this paper, we define end-point projection from whole protocol specifications to the CPPL [15] role specification language that builds on past work but considers the question of information asymmetries. We target CPPL because it supports a host of other analyses.

An additional problem is that protocol specifications have a few idiomatic forms. They typically do not explicitly encode: (a) what information is available at the first step of the protocol; (b) where and when various values, such as nonces[1], are generated; (c) when certain messages are *not* deconstructed; and, (d) the order in which messages components are deconstructed. We provide a whole-protocol specification language, WPPL, that allows all these to be expressed explicitly. Furthermore, we provide a transformation that removes idioms (b), (c), and (d) from a protocol.

Formalization. Our work is presented in the context of an adaptation of CPPL, the Cryptographic Protocol Programming Language. We have built an actual tool and applied it to concrete representations of protocols, as we discuss in Sec. 5. All our work is formalized using the Coq proof assistant [24]. Coq provides numerous advantages over paper-and-pencil formalizations. First, we use Coq to mechanically check our proofs, thereby bestowing much greater confidence on our formalization and on the correctness of our theorems. Second, because all proofs in Coq are constructive, our tool is actually a certified implementation that is extracted automatically from our formalization, thereby giving us confidence in the tool also. Finally, being a mechanized representation means others can much more easily adapt this work to related projects and obtain high confidence in the results.

Our Coq formalization[2] includes 2.6k lines of specification and 3.5k lines of proof and was produced in roughly three mythical man-months. The formalization of CPPL and strands spaces consists of 1.1k and 1.5k lines. The definition of WPPL is only 1.3k lines, evenly divided between specification and proof. Finally, the idiom removal transformation is 841 lines of specification and 1.3k lines of proof. The most difficult part of the work was formulating and verifying properties about the transformation; the other components merely required commitment and patience.

Outline. We define the syntax and semantics of our language, WPPL, in Sec. 2. Pursuant to describing our end-point projection, we give the relevant details of CPPL in Sec. 3. We give the end-projection from WPPL to CPPL in Sec. 4. We describe our transformation from idiomatic to explicit protocol specifications in Sec. 5. We follow with related work, and our conclusion.

2 WPPL

WPPL is our domain-specific language for expressing whole cryptographic protocols with trust annotations. It matches the level of abstraction of the Dolev-Yao model [12], i.e., the programmer regards the cryptographic primitives as black boxes and concentrates on the structural aspects of the protocol. In this view of protocol behavior, as each principal executes, it builds up an *environment* that binds identifiers to values encountered and compares these values with subsequent instances of their identifiers.

The Core Language. The syntax of the WPPL core language is presented in Fig. 2. The core language has protocol specifications (s), role declarations (rs), and six types of

[1] Nonces are unique random values intended to be used only once for, e.g., replay protection.

[2] Sources are available at: `http://www.cs.brown.edu/research/plt/dl/esorics2008/`

$$\begin{array}{rll} s & \rightarrow & (\text{spec } rs^* \; a) \\ rs & \rightarrow & [x \; (v^*) \; (u^*)] \\ a & \rightarrow & . \quad | \quad [x \rightarrow y : \Phi \; m \; \Psi] \; a \quad | \quad \text{let } v \; @ \; x = \text{new } nt \; a \\ & | & \text{bind } v \; @ \; x \text{ to } m \; a \quad | \quad \text{match } v \; @ \; x \text{ with } m \; a \quad | \quad \text{derive } \Phi \; @ \; x \; a \\ nt & \rightarrow & \text{nonce} \quad | \quad \text{symkey} \quad | \quad \text{pubkey} \\ m & \rightarrow & \text{nil} \quad | \quad v \quad | \quad k \quad | \quad (m, m') \quad | \quad hash(m) \quad | \quad < v = m > \\ & | & [m]v \quad | \quad [|m|]v \quad | \quad \{m\}v \quad | \quad \{|m|\}v \\ v & \rightarrow & x : t \\ t & \rightarrow & \text{text} \quad | \quad \text{msg} \quad | \quad \text{nonce} \quad | \quad \text{name} \quad | \quad \text{symkey} \quad | \quad \text{pubkey} \end{array}$$

Fig. 2. WPPL Syntax

```
(spec ([a (a b s kas) (kab)]
       [b (b s kbs) (kab)] [s (a b s kas kbs) ()])
 [a -> s : a, b, na:nonce]
 [s -> b : {|a, b, na, kab|} kas, {|a, b, na, kab|} kbs]
 [b -> a : {|a, b, na, kab|} kas, {|na|} kab, nb:nonce]
 [a -> b : {|nb|} kab] .)
```

Fig. 3. Kao Chow in WPPL

actions (a). Programming language identifiers are indicated by x, lists of variables by v^*, and constants (such as 42) by k. The language has syntax for trust management formulas—by convention we write guaranteed formulas as Φ and relied formulas as Ψ.

Examples. Fig. 3 shows the Kao Chow [17] protocol as an idiomatic WPPL specification. We will discuss what exactly is idiomatic about this specification in Sec. 5.

Syntactic Conventions. m refers to both messages and message patterns because of the network model. Consider the m expression `(a, b, na, kab)`. The sender looks up `a`, `b`, etc., in its environment to construct a message that it transmits. From the receiver's perspective, this is a pattern that it matches against a received message (and binds the newly-matched identifiers in its own environment). Because of the intervening network, we cannot assume that the components bound by the receiver are sent by the sender.

A protocol specification declares roles and an action. Role declarations give each role a name x, a list v^* of formal parameters, and a list u^* of identifiers that will be returned by the protocol representing the "goal" of the protocol. Actions are written in continuation-passing style. Although the grammar requires types attached to every identifier, we use a simple type-inference algorithm to alleviate this requirement. (Similarly, we infer the principal executing each action.) However, these technical details are standard, so we do not elaborate them.

Well-formedness. Compilers use BNF context-free grammars to syntactically parse programs. They must also use context-sensitive rules to determine which syntactic expressions are legal programs: e.g., when the syntax refers only to bound identifiers. Similarly, not all WPPL specifications describe realizable protocols. The only surprising

$$\begin{array}{ccccc}
\textsc{Var (Send)} & \textsc{Hash (Send)} & \textsc{SymEnc (Send)} & \textsc{Var (Recv)} & \textsc{Hash (Recv)} \\
\dfrac{v \in \sigma}{\sigma \vdash_s v} & \dfrac{\sigma \vdash_s m}{\sigma \vdash_s hash(m)} & \dfrac{i \in \sigma \qquad \sigma \vdash_s m}{\sigma \vdash_s \{|m|\}(i : symkey)} & \dfrac{}{\sigma \vdash_r v} & \dfrac{\sigma \vdash_s m}{\sigma \vdash_r hash(m)}
\end{array}$$

Fig. 4. Message well-formedness (excerpts)

aspect of the well-formedness condition on WPPL is that, due to cryptographic primitives, the conditions are different on message patterns used for sending and receiving.

Intuitively, to send a message we must be able to construct it, and to construct it, every identifier must be bound. Therefore, a pattern m is well-formed for sending in an environment σ (written $\sigma \vdash_s m$; see Fig. 4 for non-structural rule examples) if all identifiers that appear in it are bound; e.g., the message on line 3 of Kao Chow is not well-formed, because na is not bound.

A similar intuition holds for using a message pattern to receive messages. To check whether a message matches a pattern, the identifiers that confirm its shape—namely, those that are used as keys or under a **hash**—must be known to the principal. Thus, we define that a pattern m is well-formed for receiving in an environment σ (written $\sigma \vdash_r m$; see Fig. 4 for non-structural examples[3]) if all identifiers that appear in key-positions or **hash**es are bound. For example, the message pattern on line 4 of Kao Chow is not well-formed to receive, because b does not know kas.

We write $\sigma \vdash fs$ to mean that the formulas fs are well-formed in the environment σ. This holds exactly when the identifiers mentioned in fs are a subset of σ.

A WPPL specification is well-formed when, for each declared role, the identifiers it uses are bound and the messages it sends or receives are well-formed.

We write $\kappa, \sigma \vdash^r_\rho a$ to mean that an action a is well-formed for role r, that r expects to return ρ, in the environment σ, when it has previously communicated with the roles in κ, where κ and σ are sets of identifiers, r is an identifier, ρ is a list of variables, and a is an action. We write $\vdash^r s$ to mean that the specification s is well-formed for the role r and $\vdash s$ to mean that specification s is well-formed for all roles r that appear therein.

The parameter ρ is necessary because **return**s do not specify what is being returned. The parameter κ is necessary because we do not require explicit communication channels. At first glance, it may seem that we must only ensure that the name of a communication partner is in σ, but this is too conservative. We are able to reply to someone even if we do not know their name. Therefore, κ records past communication partners.

We choose to require well-formed WPPL specifications to be causally connected [9]. This means that the actor in each action is the same as that of the previous action, except in the case of communication. Our presentation does not rely on this property, so we believe we could elide it. However, we believe that to do so would allow confusing specifications that are not commonly found in the literature.

Semantics. The semantics of WPPL contains an implicit end-point projection. Each phrase is interpreted separately for every role as a set of strands that describes one possible local run of that role. These sets are derived from the many possible strands

[3] The HASH (RECV) rule is *not* a typo.

$$\begin{array}{rl} p \rightarrow & \text{proc}\ x\ v^*\ \Psi\ c \\ c \rightarrow & \text{fail} \mid \text{return}\ \Phi\ v^* \mid \text{derive}\ \Phi\ c\ c' \mid \text{let}\ v = lv\ \text{in}\ c \mid \text{send}\ \Phi\ v\ m\ c\ c' \\ & \mid \text{recv}\ v\ m\ \Psi\ c\ c' \mid \text{call}\ \Phi\ x\ v^*\ u^*\ \Psi\ c\ c' \mid \text{match}\ v\ m\ \Psi\ c\ c' \\ lv \rightarrow & \text{new nonce} \mid \text{new symkey} \mid \text{new pubkey} \mid \text{accept} \mid \text{connect}\ v \mid m \\ t \rightarrow & \ldots \mid \text{channel} \end{array}$$

Fig. 5. CPPL Syntax

that may be derived as descriptions for particular phrases. We write $a \rightarrow^r_{\rho,\kappa} s, v$ to mean that an action a, when executed by role r, with return variables ρ, and the current connections κ may produce strand s and requires v to all be unique identifiers.

We use a few strategies in this semantics. First, to ensure that the same identifiers are bound in the strand, we send messages ("internal dialogue") that contain no information, but where the formulas bind identifiers. Second, the interpretation of matching is to emit an identifier, then "over-hear" it, using a message pattern. Third, when binding an identifier, the identifier is replaced by the binding in the rest of the strand.

We write $sp \rightarrow^r s, v$ to mean that spec sp, when executing role r may produce strand s and requires v to all be unique identifiers. We prove that this semantics always results in well-formed strands for well-formed specifications.

Theorem 1. *If* $\vdash^r sp$ *and* $sp \rightarrow^r s, v$ *then* $\vdash s$.

Adversary. Because the semantics of a WPPL specification is a set of strands, the Dolev-Yao adversary of the general strand model is necessarily the adversary for WPPL. A Dolev-Yao adversary has the capability to create, intercept, and redirect messages. He can redirect messages from any party to any party. He can intercept messages to learn new information. He can create messages based on previously learned information. His only constraint is that he cannot break the cryptographic primitives, i.e., he must possess the appropriate key to decrypt a message. Therefore, the only information he knows is derived from an insecure context.

3 CPPL

CPPL [15] is a domain-specific language for expressing cryptographic protocols with trust annotations. The definition we use slightly extends the original work with trust derivations, message binding, empty messages, and explicit failure.

Syntax. The syntax of the CPPL core language is presented in Fig. 5. The CPPL core language has procedure declarations and eight types of code statements. It uses the same syntactic conventions as WPPL.

Well-formedness. CPPL procedures and code statements are well-formed when all identifiers used are bound. This in turn means that messages are well-formed in their appropriate context: sending or receiving. We write $\sigma \vdash c$ to mean that code statement c is well-formed in σ. Similarly, we write $\vdash p$ for well-formedness of procedures.

Semantics. The semantics of a CPPL phrase is given by a set of *strand*, each of which describes one possible local run. We write $c \rightarrow s, v$ to codify that the strand s describes

Comm (Send, Connected)

$$\frac{t \in \kappa \qquad a \Rightarrow^{r}_{\rho,\kappa} c'}{[r \rightarrow t : \Phi\ m\ \Psi]\ a \Rightarrow^{r}_{\rho,\kappa} \text{send } \Phi\ tc\ m\ c' \text{ fail}}$$

Comm (Send, Connect)

$$\frac{t \notin \kappa \qquad a \Rightarrow^{r}_{\rho,\kappa\cup\{t\}} c' \qquad c = \text{send } \Phi\ tc\ m\ c' \text{ fail}}{[r \rightarrow t : \Phi\ m\ \Psi]\ a \Rightarrow^{r}_{\rho,\kappa} \text{let } tc = \text{connect } t \text{ in } c}$$

Fig. 6. End-point projection (excerpts)

the code statement c, under the assumption that the identifiers in v are unique. We write $p \rightarrow s, v$ to mean that a procedure p is described by the strand s with the unique identifiers v.

Theorem 2. *If* $\vdash p$ *then if* $p \rightarrow s, v$, *then* $\emptyset \vdash s$.

4 End-Point Projection

Our end-point projection of wppl into cppl is realized as a compiler. We write $a \Rightarrow^{r}_{\rho,\kappa} c$ to mean that the projection of a for the role r, with return variables ρ, when κ are all roles r has communicated with, is c. An example of this definition is given in Fig. 6.

The compilation is straight-forward, by design of wppl, and is, in principle, no different that previous work on this topic. The only interesting aspect is that cppl uses channels in communication, while wppl uses names. Thus, we must open channels as they are necessary. This is accomplished by the auxiliaries `let_connect` and `let_accept`, which produce a **let** statement that opens the channel through the appropriate means. These introduce new bindings in the cppl procedure that are not in the wppl specifications, and must be specially tracked.

Theorem 3. *If* $\kappa, \sigma \vdash^{r}_{\rho} a$ *and* $a \Rightarrow^{r}_{\rho,\kappa} c$ *then* $\sigma \vdash c$.

Theorem 4. *If* $a \Rightarrow^{r}_{\rho,\kappa} c$, *then if* $a \rightarrow^{r}_{\rho,\kappa} s, v$ *then* $c \rightarrow s, v$.

Theorem 4 expresses preservation of *semantic meaning*: the strand s and the unique set v are identical in both evaluation judgments. This means that the cppl phrase *perfectly* captures the meaning of the wppl phrase.

Another two proofs about the compiler show that there is always a **return** statement and that all **return** statements are the same. This ensures the well-formedness of compiled cppl procedures.

We lift the compiler of actions to specifications. We write $s \Rightarrow^{r} p$ to mean that the projection of the specification s for the role r is the procedure p. For some roles, namely those that are not declared in the specification, we will write $s \Rightarrow^{r} \perp$ to indicate the lack of a projection for the role.

Theorem 5. $\vdash^{r} s$ *implies there is a* p *such that* $s \Rightarrow^{r} p$ *and* $\vdash p$.

Theorem 6. $sp \rightarrow^{r} s, v$ *implies* $sp \Rightarrow^{r} p$ *implies* $p \rightarrow s, v$.

```
a(a:name, b:name, s, kas) (kab)
 let na = new nonce in
 let chans = connect s in
 send _ -> chans (a, b, na) then
 let chanb = accept in
 recv chanb (stoa, btoa, nb:nonce) -> _
 then match stoa {|a, b, na, kab|} kas -> _
 then match btoa {|na|} kab -> _
 then let atob = {|nb|} kab then return
 else fail else fail else fail else fail else fail end
```

Fig. 7. Compilation of Kao Chow role A into CPPL

Example. Fig. 7 is the compilation of one role of the Kao Chow protocol (Fig. 3), after explication. The others are similar.

5 Explicit Transformation

Having shown a correct end-point projection, we turn to the problem of handling the idioms in specification. The Kao Chow specification (Fig. 3) is not well-formed because:

1. a cannot construct the message on line 3 because na is not bound.
2. s cannot construct the message on line 4 because kab is not bound.
3. b cannot match the message on line 4 because kas is not bound.
4. b cannot construct the message on line 5 because nb is not bound.
5. a cannot match the message on line 5 because kab is not bound.

We are specifically interested in allowing protocols to be taken from standard presentations in the literature and used with our compiler. As other researchers have noted [1,4], protocols like this often make use of very loose constructions and leave many essentials implicit. One approach is to reject such protocols outright and force conformance. Our approach is to recognize that there is a de facto idiomatic language in use and support it, rather than throwing out the large number of extant specifications.

The Kao Chow protocol contains all the idioms that we most commonly encounter:

I. Implicitly generating fresh nonces and keys by using a name that does not appear in the rest of the specification (e.g., na, kab, and nb).
II. Allowing roles to serve as carriers for messages that they cannot inspect, without indicating this (e.g., the first part of line 4's message).
III. Leaving the order of pattern-matching unspecified (e.g., line 5).

We remove all these idioms and produce a version of this protocol that is well-formed.

Overview. Our transformation has three stages. The first removes idiom I, by generating new bindings for nonces and keys. The second removes idiom II and is the first step of removing idiom III. It lifts out message components that are possibly unmatchable,

```
  (spec ([a (a b s kas) (kab)]
         [b (b s kbs) (kab)] [s (a b s kas kbs) (kab)])
3’ let na = new nonce
   [a -> s : a, b, na]
4’ let kab = new symkey
   [s -> b : {|a, b, na, kab|} kas, {|a, b, na, kab|} kbs]
5’ let nb = new nonce
   [b -> a : {|a, b, na, kab|} kas, {|na|} kab, nb]
   [a -> b : {|nb|} kab] .)
```

Fig. 8. Kao Chow in WPPL after step one

i.e., encrypted or hashed, and binds them to identifiers. The third stage matches bound identifiers against their construction pattern, when this will succeed. This sequences pattern matching, removing idiom III, and recovers any losses temporarily created by stage two. We conclude with an evaluation of these transformations.

Generation. Our first transformation addresses problems 1, 2 and 4 above by explicitly generating fresh values for all nonces, symmetric keys, and public keys that appear free in the entire protocol. After transformation, Kao Chow is as in Fig. 8.

This transformation is very simple, so we do not present it in detail. Instead, we explain its correctness theorem. We write *gen*(s) as the result of this stage for specification s. Our theorem establishes a condition for when the first idiom is removed:

Theorem 7. *For all s, if its action does not contain an instance of recursive binding, then for all vi, if vi appears free in gen(s), then there exists a type vt, such that vt is* not `nonce`*,* `symkey`*, or* `pubkey` *and vi appears free in s with type vt.*

An action has recursive binding if it contains as a sub-action `bind` r v m a and v appears in m. These actions are not strictly well-formed, because v is not bound in m. However, we cannot assume that input specifications are well-formed—our transformation is to make them so! So, we have parceled out the smallest amount of well-formedness necessary.

This theorem says that any free identifier (a) is not of one of the types for which we can construct a fresh value and (b) is free in the original specification, i.e., was *not* introduced by our transformation. Thus, the first idiom is successfully removed. But we still have problems 3 and 5, so our modified Kao Chow protocol is still not well-formed.

Lifting. The second stage transforms each message construction by introducing a message binding for each message component. It binds those components that can potentially fail to match, namely signing and encryption (which require the key to be matched), and hashing (which requires the hash contents to be matched). This results in Kao Chow further being rewritten to the form shown in Fig. 9. As a result of this transformation, `b` can transmit `msg0` without needing to inspect it on line 4.

This serves to ensure (a) all matching sides of communication are well-formed; (b) messages that are carried without inspection are well-formed for sending; and, (c) sequencing is completely unspecified on the receiving side. We prove a theorem about this stage that establishes criterion (a), but we argue criteria (b) and (c) informally.

```
   (spec ([a (a b s kas) (kab)]
          [b (b s kbs) (kab)] [s (a b s kas kbs) (kab)])
    let na = new nonce
    [a -> s : a, b, na]
    let kab = new symkey
4'' bind msg0 = {|a, b, na, kab|} kas
4'' bind msg1  = {|a, b, na, kab|} kbs
    [s -> b : msg0, msg1]
    let nb = new nonce
5'' bind msg2 = {|na|} kab
    [b -> a : msg0, msg2, nb]
6'' bind msg3 = {|nb|} kab
    [a -> b : msg3] .)
```

Fig. 9. Kao Chow in WPPL after step two

```
    (spec ([a (a b s kas) (kab)]
           [b (b s kbs) (kab)] [s (a b s kas kbs) (kab)])
     let na = new nonce
     [a -> s : a, b, na]
     let kab = new symkey
     bind msg0 = {|a, b, na, kab|} kas
     bind msg1 = {|a, b, na, kab|} kbs
     [s -> b : msg0, msg1]
5''' match msg1 with {|a, b, na, kab|} kbs
     let nb = new nonce
     bind msg2 = {|na|} kab
     [b -> a : msg0, msg2, nb]
6''' match msg0 with {|a, b, na, kab|} kas
6''' match msg2 with {|na|} kab
     bind msg3 = {|nb|} kab
     [a -> b : msg3]
7''' match msg3 with {|nb|} kab .)
```

Fig. 10. Kao Chow in WPPL after step three

One interesting part of our transformation is that it is not structurally recursive in the action. In the cases for communication, binding, and matching, the translation is recursively applied to the result of replacing all instances of the lifted message components in the continuation. Instead, we recur on a natural number bound. We prove that the number of actions is a lower bound for its correctness.

This transformation removes idiom II, but introduces many instances of idiom III.

Opening. The third stage introduces pattern-matching at each point where a message previously bound may be successfully matched against the pattern it was bound to. This results in Kao Chow being rewritten to the final, well-formed, form shown in Fig. 10.

After this stage, every message that can possibly be deconstructed by each role is deconstructed. This removes idiom III by fully specifying the order of pattern matching. In particular, it removes instances of idiom III introduced by the second stage.

Although it follows from this stage's mission statement, it is not necessarily intuitive that this stage will also check that previously unverified message contents are correct. Since this pattern occurs when a message that *could not* be deconstructed becomes transparent, this pattern *is* handled by this step. For example, if a commitment message, $hash(m)$, has been received, but the contents, m, is unknown until a later step; this stage will check the commitment at the appropriate time.

This stage must solve the following problem: find some order whereby the messages may be matched, or "opened." A message may be opened if (a) the identifier it is bound to is bound (which isn't necessarily the case: e.g., `msg0` is not bound on line 7) and (b) it is well-formed for receiving in the environment. At each line of the specification, our transformation will compute the set of messages that may be opened, and the order to open them in. If a message cannot be opened, it will be reconsidered in subsequent lines. Note that this is a recursive set, because opening a message extends the environment, potentially enabling more messages to be opened. Thus, messages that can't be opened may become amenable to opening after some other message is opened.

The core of the transformation is a function that computes this set for each line. This function, `open_bmsgs`, in principle accepts a B, list of identifiers and message patterns, i.e., the bound messages, and σ, an environment. It partitions B into two lists: C, the bound messages that *cannot* be opened; and O, those that can.

In fact, this function cannot not only receive these two values as arguments, because it cannot be defined recursively on either of them. Instead, it is supplied an additional integer bound that must be greater than the number of messages.

Theorem 8. *If* (i, m) *is in* C*, then either* $i \notin \sigma \cup ids(O)$ *or* $\sigma \cup ids(O) \nvdash_r m$.

Theorem 9. *If* $O = p \mathbin{@} (i, m) :: q$*, then* $i \in \sigma \cup ids(p)$ *and* $\sigma \cup ids(p) \vdash_r m$.

Our transformation is trivial, given `open_bmsgs`. In essence, it keeps track of the environment of the role being transformed and the bound messages. Then, it calls `open_bmsgs` and uses the result of a small auxiliary, `build_match`, destructures bound messages, thereby removing idiom III. We prove the following theorem about `build_match`:

Theorem 10. *If* $\kappa, \sigma \cup ids(O) \vdash^r_\rho a$ *and* `build_match` $r\ O\ a = a'$ *then* $\kappa, \sigma \vdash^r_\rho a'$.

Evaluation We now evaluate the effectiveness of these transformations. We did not "evaluate" end-point projection because our theorem established that it succeeds for *all* inputs. Similarly, we have established appropriate correctness conditions for each of the three transformations. However, we still need to determine how well the three transformations actually cover the space of protocols found in practice. (Note that we cannot argue the correctness of the composition of the three transformations, because their input is not well-formed, so there is no foundation for their "correctness".)

We attempted to encode fifty of the protocols in the SPORE repository in WPPL. We successfully encoded forty-three of the protocols. We consider this compelling evidence that WPPL is useful for producing protocols. Of these protocols, *zero* were well-formed in the repository and *all* are well-formed after applying the composed transformation. This demonstrates the transformations can remove idioms from protocol specifications.

Weaknesses and Limitations. WPPL has a few weaknesses that prevent all protocols to be encoded. First, WPPL cannot express unique cryptographic primitives. It can only express asymmetric or symmetric modes of encryption or signing and hashing. Therefore, it cannot express protocols that build these (and other) primitives. For example, the Diffie-Hellman protocol [11] is a method of creating shared symmetric keys. We cannot guarantee from within our framework that these keys are equivalent to our symmetric keys. In essence, this protocol is too low-level for our theory.

Second, there is no way to express conditional execution in WPPL. Thus, there is only one path through a protocol and protocols with built-in failure handling, such as the GJM protocol [14], cannot be written. We could have extended our work to allow the expression of these protocols, but WPPL attempts to capture the spirit of traditional protocol description, and we cannot identify a community consensus on how to write conditional executions.

Third, protocols that rely on parallel execution are not well-formed, as mentioned earlier. For example, a principal cannot transmit two messages in a row, without receiving a message. The transformation does not remove this instance of parallelism.

6 Related Work

End-point Projection. Carbone, Honda, and Yoshida [9] have developed an end-point projection for the Choreography Description Language. Our approach is different in a few fundamental ways. First, they do not specify a well-formedness condition to identify information asymmetries between participants as we do in Sec. 2, because they do not discuss cryptography. Instead, their well-formedness conditions only involve session usage. Second, sessions are an assumed concept in Web Services, whereas in the cryptographic space, session creation is often the goal of a protocol. Therefore, we cannot impose their session usage conditions. Third, they allow conditional and parallel protocol execution, which we do not.

Sabri and Khedri [23] employ the Carbone framework of end-point projection in their development of a framework for capturing the knowledge of protocol participants. They observe that they must verify the well-formedness of protocols given information asymmetries, but do not address this problem. Rather, they assume they are resolved appropriately. Therefore, our work is complementary to theirs.

Corin et al. [10] perform end-point projection in their work on session types. A session type is a graph that expresses a global relation among protocol participants, including the pattern of legal communications. When implementing a role, it is necessary to consider that role's view of the graph. End-point projection is used to derive these session views as types that can be verified at each end-point. Corin's application of end-point projection is different from either Carbone's or ours. While Corin uses EPP at the type-level, we use it at the program-level. However, the technique is basically the same.

Program Slicing. The essence of end-point projection is program slicing [25]. Program slicing is a technique to take a program and remove parts of it that are irrelevant to some particular party. For example, the program slice with respect to some variable is the subset of that program which influences the value of the variable. In our case, we will be slicing a program with respect to some participant and removing parts of the program that do not concern that party. In particular, if the role in question receives a message, it need not be concerned with the generation of that message.

Compiling Traditional Protocol Specifications. CAPSL [20] is a system that transforms a protocol into runnable Java code. CAPSL describe the entire protocol and their compiler performs an end-point projection. Our work is distinct for multiple reasons. First, CAPSL targets the analysis theory of the NRL Protocol Analyzer [19], rather than the stand spaces with rely-guarantee. Second, CAPSL specifications are fully explicit and annotated to resolve idioms, thus they do not resemble the style used in the literature.

Casper [18], "a compiler for the analysis of security protocols", takes programs written in a whole-protocol form and transforms it into CSP processes that can be checked with the FDR model-checker. This system is intended not only to specify the protocol, but also its security goals. Like CAPSL, protocols are required to be fully explicit and without idioms. However, Casper is superior to WPPL in that it deals with properties. This represents a difference in WPPL's focus: protocol deployment rather than development.

CAPSL, Casper, AVISS [3] and CVS [13] use the % operator to annotate when values cannot be understood and must be treated as black boxes. Our transformation essentially generates these annotations are generated where information asymmetries exist.

Jacquemard et al. [16] compile whole-protocol specifications into rewrite rules that model the protocol and can be verified with the daTac theorem-prover. This system specifies the protocol as well as properties about it. The main advantage over Casper, etc. is that daTac can be used to handle infinite state models. In their work, they deal with information asymmetries by tracking the knowledge available to each principal at each point of the protocol. However, they do not revisit earlier message components that were treated as black boxes when the knowledge needed to inspect their contents becomes available as we do. This process of dealing with idioms and information asymmetries is embedded in their semantics, rather than an orthogonal step (as we present it).

Explication. Bodei et al. [5] mention the idioms of traditional protocol specifications. In their work they give some advice for how to manually make such specifications explicit, but they do not automate this process.

Briais and Nestmann [6] investigate the formal semantics of traditional protocol specifications. They address three of four forms of informality mentioned by Abadi [1]. Only two of these correspond to parts of our transformation or specification language. First, to "make explicit what is known before a protocol is run", which we require in the WPPL specification. Second, to "make explicit ... what is to be generated freshly during a protocol run", which we detect as step one of our transformation. In their work, they do not require the usage of the % operator, but they do not revisit old messages, when more information is available, as we do.

Caleiro et al. [8] study the semantics of traditional protocol specifications. In their work, they focus on the internal actions of principals. They give a manual strategy for encoding traditional whole-protocol specifications into a number of single-role specifications in their semantic framework. Their denotational semantics of these specifications makes explicit when and how incoming messages are checked and outgoing messages generated. They also provide a transformation of their specifications into a variant of the spi-calculus [2]. They prove that this transformation is meaningfully related to the denotational semantics. In contrast, our work takes traditional specifications mostly as-is and directly provides a semantics in the strand spaces model. We also provide a formally verified compilation to CPPL for deployment purposes. An important similarity in the two works is that in the their approach messages that cannot be understood are represented as variables in their "incremental symbolic runs", while in our approach, the idiom removal transformation introduces these variables directly into the specification and checked when possible.

CPPL *and Process Calculi.* CPPL is uncommon amongst cryptographic protocol calculi and verifiers. Verifiers typically work with process calculi languages, such as the spi-calculus [2]. In contrast to spi, CPPL is not intended to be verified directly; instead, it is meant to be used in implementations. Verification of CPPL specifications is through its semantic interpretation—the strand spaces model. This model has a rich body of analysis research, as well as formal relations to many other verification methods, which our work can seamlessly leverage for verification purposes.

7 Conclusion

We have presented WPPL, a programming language for whole-protocol specification, along with an end-point projection from WPPL to CPPL. We have shown that this projection is correct through verified proofs. We have also given a transformation that resolves idioms in traditional protocol presentations. We have shown properties of this transformation with verified proofs. We have validated our transformations by applying them successfully to eighty-six percent of the protocols in the SPORE repository. In the future, we would like to extend WPPL to support conditional and parallel execution, and better integrate our work with the existing results from Carbone, et al., for Web Services [9].

Acknowledgments. This work is partially supported by the NSF (CCF-0447509, CNS-0627310, and a Graduate Research Fellowship), Cisco, and Google. We are grateful for the advice and encouragement of Joshua Guttman and John Ramsdell.

References

1. Abadi, M.: Security protocols and their properties. In: Foundations of Secure Computation (2000)
2. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Information and Computation 148(1), 1–70 (1999)
3. Armando, A., Basin, D.A., Bouallagui, M., Chevalier, Y., Compagna, L., Mödersheim, S., Rusinowitch, M., Turuani, M., Viganò, L., Vigneron, L.: The AVISS security protocol analysis tool. In: Computer Aided Verification (2002)
4. Bodei, C., Buchholtz, M., Degano, P., Nielson, F., Nielson, H.R.: Automatic validation of protocol narration. In: Computer Security Foundations Workshop (2003)
5. Bodei, C., Buchholtz, M., Degano, P., Nielson, F., Nielson, H.R.: Static validation of security protocols. Journal of Computer Security 13(3), 347–390 (2005)
6. Briais, S., Nestmann, U.: A formal semantics for protocol narrations. Theoretical Computer Science 389(3), 484–511 (2007)
7. Burrows, M., Abadi, M., Needham, R.: A logic of authentication. Proceedings of the Royal Society Series A 426(1871), 233–271 (1871)

8. Caleiro, C., Viganò, L., Basin, D.: On the semantics of Alice&Bob specifications of security protocols. Theoretical Computer Science 367(1-2), 88–122 (2006)
9. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: European Symposium on Programming (2007)
10. Corin, R., Denielou, P.-M., Fournet, C., Bhargavan, K., Leifer, J.: Secure implementations for typed session abstractions. In: Computer Security Foundations Symposium (2007)
11. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976)
12. Dolev, D., Yao, A.: On the security of public-key protocols. IEEE Transactions on Information Theory 29, 198–208 (1983)
13. Durante, A., Focardi, R., Gorrieri, R.: A compiler for analyzing cryptographic protocols using noninterference. ACM Transactions on Software Engineering and Methodology 9(4), 488–528 (2000)
14. Garay, J.A., Jakobsson, M., MacKenzie, P.: Abuse-free optimistic contract signing. In: International Cryptology Conference (1999)
15. Guttman, J.D., Herzog, J.C., Ramsdell, J.D., Sniffen, B.T.: Programming cryptographic protocols. In: Trust in Global Computing (2005)
16. Jacquemard, F., Rusinowitch, M., Vigneron, L.: Compiling and verifying security protocols. Logic for Programming and Automated Reasoning (2000)
17. Kao, I.L., Chow, R.: An efficient and secure authentication protocol using uncertified keys. Operating Systems Review 29(3), 14–21 (1995)
18. Lowe, G.: Casper: A compiler for the analysis of security protocols. In: Computer Security Foundations Workshop (1997)
19. Meadows, C.: A model of computation for the NRL protocol analyzer. In: Computer Security Foundations Workshop (1994)
20. Millen, J., Muller, F.: Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International (December 2001)
21. Neuman, B.C., Ts'o, T.: Kerberos: An authentication service for computer networks. Technical Report ISI/RS-94-399, USC/ISI (1994)
22. Project EVA. Security protocols open repository (2007), `http://www.lsv.ens-cachan.fr/spore/`
23. Sabri, K.E., Khedri, R.: A mathematical framework to capture agent explicit knowledge in cryptographic protocols. Technical Report CAS-07-04-RK, McMaster University (2007)
24. The Coq development team. The Coq proof assistant reference manual, 8.1 edition (2007)
25. Weiser, M.: Program slicing. In: International Conference on Software Engineering (1981)