Trusted Multiplexing of Cryptographic Protocols

Jay McCarthy[1] and Shriram Krishnamurthi[2]

[1] Brigham Young University
[2] Brown University

Abstract. We present an analysis that determines when it is possible to multiplex a pair of cryptographic protocols. We present a transformation that improves the coverage of this analysis on common protocol formulations. We discuss the gap between the merely possible and the pragmatic through an optimization that informs a multiplexer. We also address the security ramifications of trusting external parties for this task and evaluate our work on a large repository of cryptographic protocols. We have formally verified this work using the Coq proof assistant.

1 Problem and Motivation

A fundamental aspect of a cryptographic protocol is the set of messages that it may accept. Protocol specifications contain patterns that specify the shape of the messages they accept. These patterns describe an infinite set of messages, because the variables that appear in them may be bound to innumerable values. We call this set a protocol's *message space*.

There is a history of attacks on protocols based on the use of (parts of) messages of one protocol as (parts of) messages of another protocol [2, 11, 14]. These attacks, called type-flaw (or type-confusion) attacks, depend fundamentally on the protocol relation of *message space overlap*. If the message spaces of two protocols overlap, then there is at least one session of each protocol where at least one message could be accepted by both protocols. This property, however, is more general than a "presence of type-flaw attack" property, because not all overlaps are indications of successful attacks. (In fact, it is common for new versions of a protocol to contain many similar messages.)

The message space overlap property not only gives us insight into the protocol and its relation to other protocols it also provides a test for a fundamental deployment property: dispatchability. We define *dispatchability* as the ability for a multiplexer to unambiguously deliver incoming protocol messages to the proper protocol session. (We can compare a protocol session's message space with another session's message space to determine if it is possible to dispatch to the correct session. This basic property is necessary for servers to provide concurrency and support for many protocol clients.)

Servers typically rely on TCP for this property. They assign a different TCP port for each protocol and trust the operating system's TCP implementation to do the dispatching. However, when cryptographic protocols are embedded in other contexts, such as existing Web service protocols (e.g., SOAP), more explicit methods of distinguishing protocol messages must be used. Furthermore, by leaving this essential step implicit, it is not included in the formally verified portion of the protocol specification. This means that the protocol that is actually used is *not* the one that is verified. Finally, the delegated notion of a session (e.g., TCP's or SSL's) may not match the protocol's notion. This is particularly problematic in protocols with more than two participants that are not simply compositions of two-party protocols.

Notice that message space overlap implies that dispatchability is not achievable. If there is a message M that could be accepted by session p and session q of some protocols, then what would a dispatcher do when delivered M? A faulty identification might cause the actual (though unintended) recipient to go into an inconsistent state or even leak information while the intended recipient starves. It cannot unambiguously deliver the message and therefore is not a correct dispatcher. We present a dispatching algorithm that correctly delivers messages if there is no message space overlap. This algorithm provides proof that the lack of message space overlap implies dispatchability.

We present an analysis that determines whether the message spaces of two protocols (sessions of a protocol) overlap. We also present an analysis, phrased as an optimized dispatcher, that determines *why* there is no overlap between two spaces by finding the largest abstractions of two protocols for which there is no overlap. We present our analysis of protocols from the SPORE protocol repository [15] and show how studying them provides insights to improve our analyses.

We present our work in the context of an adaptation of CPPL, the Cryptographic Protocol Programming Language [8]. We have built an actual tool and applied it to concrete representations of protocols. All of our work is formalized using the Coq proof assistant [18], and we make our formalization freely available.[3] 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 in a standard way 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.

Outline. In Section 2, we explain the technical background of our theory. Next, in Section 3, we develop the decision procedure for message space overlap. In Section 4, we show how message space overlap provides a sufficient foundation for a dispatching algorithm. This algorithm is inefficient, so we present an analysis in Section 5 that optimizes it. Finally, we discuss related work and our conclusions.

2 Introduction to CPPL

CPPL [8] is a domain-specific language for expressing cryptographic protocols with trust annotations. CPPL allows the programmer to control protocol actions by using trust constraints so that an action such as transmitting a message will occur only when the indicated trust constraint is satisfied. The CPPL semantics identifies a set of *strands* [17], annotated with trust formulas and the values assumed to be unique, as the meaning of a role in a protocol.

[3] Sources are available at: `http://faculty.cs.byu.edu/~jay/tmp/dispatch09/`.

```
a knows a:name b:name kab:symkey
  learns kabn:symkey
b knows a:name b:name kab:symkey
  learns kabn:symkey

1 a -> b : a, {|na:nonce|} kab
2 b -> a : {|na, nb:nonce|} kab
3 a -> b : {|nb|} kab]
4 b -> a : {|kabn:symkey, nbn:nonce|} kab
```

Fig. 1. Andrew Secure RPC Protocol

```
 1 proc b (a:name b:name kab:symkey) _
 2  let chana = accept in
 3  recv chana (a, {| na:nonce |} kab) -> _ then
 4   let nb = new nonce in
 5   send _ -> chana {| na, nb |} kab then
 6   recv chana {| nb |} kab -> _ then
 7    let nbn = new nonce in
 8    let kabn = new symkey in
 9    send _ -> chana {| kabn, nbn |} kab then
10    return _ (kabn)
```

Fig. 2. Andrew Secure RPC Role `B` in CPPL

We will explain the relevant aspects of CPPL by using the Andrew Secure RPC protocol (Figure 1) and the encoding of its B role in CPPL (Figure 2) as our example.

Message Syntax. The various kinds of messages that may be sent and received are paramount to our investigation. We give their syntax in Figure 3. Messages (m) may be constructed by concatenation (,), hashing ($hash(m)$), variable binding and pattern matching ($< v = m >$), asymmetric signing ($[m]v$), symmetric signing ($[|m|]v$), asymmetric encryption ($\{m\}v$), and symmetric encryption ($\{|m|\}v$). In these last four cases, v is said to be in the *key-position*. For example, in the Andrew role `kab` is in key-position on line 3. Concatenation is right associative. Parentheses are control precedence.

Well-formedness. As a CPPL program executes, it builds a runtime environment of locally known values associated with identifiers. This environment is consulted to determine the values of pattern identifiers in message syntax and is extended during matching when those identifiers are free. Not all syntactically valid messages are well-formed in a CPPL program, because they may refer to free identifiers in positions that cannot be free. We say that such patterns are not *well-formed*.

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 $\sigma \vdash_s m$ herein) if all identifiers that appear in it are bound. For example, the message on line 5 of the Andrew role is well-formed, but if we removed line 4, it would not be because `nb` would not be bound.

$$\begin{array}{rlllllll}
m := & \text{nil} & | & v & | & k & & \\
| & (m, m') & | & hash(m) & | & < v = m > & & \\
| & [m]v & | & [|m|]v & | & \{m\}v & | & \{|m|\}v \\
v := & x : t & & & & & & \\
t := & \text{text} & | & \text{msg} & | & \text{nonce} & & \\
| & \text{name} & | & \text{symkey} & | & \text{pubkey} & | & \text{channel}
\end{array}$$

Fig. 3. CPPL Message Syntax

Surprisingly, the well-formedness condition is different for message patterns used for receiving rather than sending: only some identifiers must be bound due to meaning of the cryptographic primitives.

However, 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, a pattern m is well-formed for receiving in an environment σ (written $\sigma \vdash_r m$) if all identifiers that appear in key-positions or hashes are bound. For example, the message pattern on line 3 of the Andrew role is well-formed because `kab` is bound, but if it were not, then the pattern would not be well-formed.

A CPPL program (p) is well-formed ($\vdash p$), when each message is well-formed in the appropriate context and runtime environment.

Semantics and Adversary. The semantics of a CPPL program is given by a set of *strands* where each strand describes one possible local run. A strand, s, is simply a list of messages that are sent ($+m$) or received ($-m$):

$$s := . \quad | \quad +m \rightarrow s \quad | \quad -m \rightarrow s$$

The adversary in the strand semantics is essentially the Dolev-Yao adversary [5]. Since a strand merely specifies what messages are sent and received rather than how they are constructed, where they are sent, or from whence they come, the adversary has maximal power to manipulate the protocol by modifying, redirecting, and generating messages *ex nihilo*. This ensures that proofs built on the semantics are secure in the face of a powerful adversary.

The basic abilities of adversary behavior that make up the Dolev-Yao model include: transmitting a known value such as a name, a key, or whole message; transmitting an encrypted message after learning its plain text and key; and transmitting a plain text after learning a ciphertext and its decryption key. The adversary can also manipulate the plain-text structure of messages: concatenating and separating message components, adding and removing constants, etc. Since an adversary that encrypts or decrypts must learn the key from the network, any key used by the adversary—compromised keys—have always been transmitted by some participant.

A useful concept when discussing the adversary is a *uniquely originating value*. This is a value that enters the network at a unique location. Locally produced nonces[4] are uniquely originating unpredictable values. By definition, the adversary cannot know these values until they have been sent in an unprotected context.

[4] **N**umbers used **once**.

3 Analysis

In this section we present our analysis that determines when there is a message that could be accepted by two sessions of two protocols. This analysis can then be applied to the case of two sessions of one protocol by comparing a protocol with itself.

The strand space model of protocols is aptly suited for this problem. From the strand, we can read off each message pattern the protocol accepts. For example, the strand $+m_1 \rightarrow -m_2 \rightarrow -m_3 \rightarrow .$ accepts messages with patterns m_2 and m_3. We denote this set of message patterns as $\mathcal{M}\ (s)$ for strand s.

Each message pattern m describes an infinite set of messages (one for each instantiation of the variables in m) that would be accepted at that point of the protocol. If we could compare the sets of two patterns, then we could easily lift this analysis to two protocols s and s' by checking each pattern in $\mathcal{M}\ (s)$ against each pattern in $\mathcal{M}\ (s')$. The essence of our problem is therefore determining when a message pattern m "overlaps" with another message pattern m', i.e., when there is an actual message M that could be matched by both m and m'. We call this analysis match.

3.1 Defining match

We have multiple options when defining match. We could assume that the *structure* of message patterns are potentially ambiguous. That is, we could assume that (m_1, m_2) could possibly overlap with $hash(m_3)$ or $\{m_4\}_k$. We will *not* do this. We assume that messages are encoded unambiguously. Concrete protocol implementations that do not conform to this assumption may have type-flaw attacks [2, 11, 14].

This initial consideration shrinks the design space of match: message patterns must have identical structure for them to possibly overlap. There are two important caveats: variables with type *msg* and bind patterns ($< v = m >$). In the first, we treat such variables as "wildcards" because they will accept any message when used in a pattern. In the second, we ignore the variable binding and use the sub-pattern m in the comparison.

With this structural means of determining when two message patterns potentially overlap, all that remains is to specify when to consider two variables as potentially overlapping. The simplest strategy is to assume that if the types of two variables are the same, then it is possible that each could refer to the same value. We call this strategy type-based and write it match_τ.

Correctness. match_τ is correct if it soundly approximates message space overlap, i.e., if $\neg\ \mathsf{match}_\tau\ m\ m'$ then there is no overlap between the possible messages accepted by pattern m and pattern m'. This implies that $\mathsf{match}_\tau\ m\ m'$ should not be read as "every message accepted by m is accepted by m'" (or vice versa), because there are some environments (and therefore protocol sessions) where there can be no overlap between messages. For example, the pattern x does not overlap with y if x is bound to 2 and y is bound to 3. But there is at least one environment pair that contains at least one message that *is* accepted by both: when x and y are bound to 2 and the message is 2.

Evaluation. The theorem prover can tell us if match_τ is correct, but it cannot tell us if the analysis is useful. We address the utility of the analysis by running it on a large number of protocol role pairs.

We have encoded 121 protocol roles from 43 protocol definitions found in the Security Protocols Open Repository (SPORE) [15] in CPPL. For each role, our analysis generates every possible strand interpretation of the role, then compares each message pattern with those of another role. Analyzing all possible pairs only takes a few seconds and we find that when using match_τ, 15.7% of protocol role pairs are non-overlapping (i.e., for 84.3% of the pairs there is a message that is accepted by both roles in a run.) This is an extravagantly high number.

If we actually look at the source of many protocols in CPPL, we learn why there are such poor results with match_τ. It turns out that many protocols have the following form:

```
1  recv chan (m_1, m:msg) -> _ then
   ...
n  match m m_2 then
```

where m_1 and m_2 are particular patterns, such as (price, p) or $\{m_1\}_k$.

Consider how match_τ would compare this message with another: Because it contains a wildcard message (with type *msg*), it is possible for *any* message to be accepted. This tells us that the specificity of the protocol role deeply impacts the efficacy of our analysis. In the next section, we develop a transformation on protocol roles that increases their specificity. This greatly improves the performance of match_τ.

3.2 Message Specificity

Suppose we have a protocol with the following protocol role:

```
1  recv ch (m1, a) -> _
2  then let nc = new nonce in
3  match m1 {|b, k′|} k -> _
```

If this role were slightly different, then we could execute it with more partners:

```
1′ recv ch (<m1={|b, k′|} k>, a) -> _
2  then let nc = new nonce in
3  match m1 {|b, k′|} k -> _
```

In this modified protocol, the wildcard message `m1` on line 1 is replaced by a *more specific* pattern on line 1′. We say that message pattern m_1 is more specific than message pattern m_2 if for all messages m, $\mathsf{match}_\tau\ m_1\ m$ implies $\mathsf{match}_\tau\ m_2\ m$ (i.e., every message that is accepted by m_1 is accepted by m_2.)

Our transformation, called foldm, increases the specificity of message patterns. It works as follows: for each message reception point where message m is received, foldm records the environment before reception as σ_m, inspects the rest of the role for pattern points where identifier i is compared with pattern p such that $\sigma_m \vdash_r p$, and replaces each occurrence of i in m with $< i = p >$, thereby increasing the specificity of m.

We prove the following theorems about this transformation:

Theorem 1 *If* $\vdash p$ *then* $\vdash \mathsf{foldm}\ p$.

Theorem 2 *Every pattern in p has a corresponding more specific pattern in* foldm *p.*

Preservation. We must also ensure that this transformation preserves the semantics of the protocol meaningfully. However, since we are clearly changing the set of messages accepted by the protocol (requiring them to be more specific), the transformed protocol does not have the same meaning.

The fundamental issue is whether the protocol meaning is different. Recall that the meaning of a protocol is a set of strands that represent potential runs. This is smaller after the transformation. However, if we consider only the runs that end in success—those runs in which a message matching pattern p is provided when expected—then there is no difference in protocol behavior.

Why? Consider the example from above. Suppose that a message M matching the pattern `(m1, a)` is provided at step 1 in the original protocol and that the rest of protocol executes successfully. Then m_1 *must* match the pattern `{|b, k'|} k`, and, the message M must match the pattern `(<m1={|b, k'|} k>, a)`. Therefore, if the same message was sent to the transformed protocol, the protocol would execute successfully. This holds in every case because the transformation *always* results in more specific patterns that have exactly this property.

What happens to runs that fail in the original protocol? They continue to fail in the transformed protocol, but may fail *differently*. Suppose that a message M is delivered to the example protocol at step 1 and the protocol fails. It either fails at step 1 or step 3. If it fails at step 1, then it does not match the pattern `(m1,a)` or the pattern `(<m1={|b, k'|} k>, a)`. Therefore it fails at step 1 in the transformed protocol as well. If it fails at step 3, then the left component of the message M does not match the pattern `{|b,k'|} k`, and, the transformed protocol will fail at step 1 for the very same reason.

In general, then, the transformed protocol's behavior is identical *modulo failure*. If the same sequence of external messages is delivered to a transformed role, then it will either (a) succeed like the untransformed counterpart or (b) fail earlier because some failing pattern matching was moved earlier in the protocol. Semantically, this means that the set of strand bundles that a protocol can be a part of is smaller.

The transformed protocol must actually be used in deployment for the analysis to be sound. If not, a message may be delivered to the wrong recipient. Worse, this misdelivery will only be apparent later when the principal attempts a deeper pattern match. Since the more specific pattern was not matched initially, this deep match will fail and signal an error.

Adversary. This transformation either decreases the amount of harm the adversary can do or does not change it. Since the only difference in behavior is that faulty messages are noticed sooner, whatever action the principal would have taken before performing the lifted pattern matching is not done. Therefore, the principal does *less* before failing, and therefore the "hooks" for the adversary are *decreased*. Of course, for any particular protocol, these hooks may or may not be useful, but in general there are fewer hooks.

Evaluation. When we apply foldm to our test suite of 121 protocol roles and then run the match_τ analysis, we find within seconds that the percentage of non-overlapping role pairs increases from 15.7% to 61%. This means that for 61% of protocol role pairs from our repository, it is always possible to unambiguously deliver a message to a single protocol handler. However, when we look just at the special case of comparing a role with itself (i.e., determining if it is possible to dispatch to sessions correctly) we find that none of the roles have this property according to match_τ.

This is an unsurprising result. Every message pattern p is exactly the same as itself. Therefore, match_τ will resolve that p has the same shape as p and could potentially accept the same messages. The problem is that match_τ looks only at the two patterns. It does not consider the context in which they appear: a cryptographic protocol that may make special assumptions about the values bound to certain variables. In particular, some values are assumed to be unique. For example, in many protocols, nonces are generated randomly and used to prevent replay attacks and conduct authentication tests [7]. In the next section, we incorporate uniqueness into our analysis.

3.3 Relying on Uniqueness

In the Andrew Secure RPC role (Fig. 2), the message received on line 6 must match the pattern $\{|nb|\}_{kab}$, where nb is a nonce that was freshly generated on line 4. This means that *no two sessions* of this role could accept the same message at line 6, because each is waiting for a *different* value for nb.

We call the version of our analysis that incorporates information about uniqueness match_δ. Whenever the analysis compares a variable u from protocol α and a variable v from protocol β, if u is in the set of unique values generated by α or v is in the set of unique values generated by β, then the two are assumed not to match, regardless of anything else about the variables. In all other cases, two variables are assumed to be potentially overlapping. In particular, the types are ignored, unlike match_τ.

Evaluation. When we apply match_δ to our test suite, we find that the percentage of non-overlapping sessions is 0.8%. After applying the foldm transformation, this increases to 14.8%. There is no degradation to the performance of the analysis either: the entire test suite results are available almost instantaneously.

If we look at the other 85.2% of the protocols, is there anything more that can be incorporated into the analysis? There is. The first action of many protocol roles is to receive a particular initiation message. Since this is the *first* thing the role does, it cannot possibly contain a unique value generated by the role. Therefore, the match_δ analysis will not be able to find a unique value that distinguishes the session that the message is meant for. In the next section, we will discuss how to get around this difficulty.

3.4 Handling Initial Messages

The first thing the Andrew Secure RPC role (Fig. 2) does (shown on line 3) is receive a certain message: $(a, \{|na|\}_{kab})$. Since this message does not contain any value uniquely generated for the active session role, it seems that the initial messages of two sessions can be confused. However, a little reflection reveals that initial messages *create* sessions, so by definition they may not be confused across sessions.

Therefore, we can safely ignore the first message of a protocol role, if it is not preceded by any other action, for the purposes of determining the dispatchability of a protocol role's sessions. We must, of course, compare the initial message with all *other*

	match_τ	match_δ	$\mathsf{match}_{\tau+\delta}$
initial	15.7%	10.0%	15.8%
foldm	61.0%	55.2%	62.1%

(a) Non-overlapping Protocol Role Pairs

	match_τ	match_δ	$\mathsf{match}_{\tau+\delta}$
initial	0.0%	00.8%	00.8%
foldm	0.0%	14.8%	14.8%
ι + foldm	31.4%	62.8%	62.8%

(b) Non-overlapping Protocol Role Sessions

Table 1. Analysis Results

messages to ensure that the initial message cannot be confused with, for example, the third message, but we do not need to compare the initial message with itself. When we use this insight with the match_δ analysis, we write it as $\mathsf{match}_{\iota(\delta)}$.

Evaluation. Table 1a presents the results when analyzing each pair of protocol roles. Interestingly, unique values are *not* very useful when comparing roles, although they do increase the coverage slightly. We have inspected the protocols not handled by $\mathsf{match}_{\tau+\delta}$ to determine why the protocol pairs may potentially accept the same message.

1. Protocols with similar goals and similar techniques for achieving those goals typically have the same initial message. Examples include the Neumann Stubblebine, Kao Chow, and Yahalom protocol families.
2. Different versions of the same protocol will often have very similar messages, typically in the initial message, though not always. Often these protocols are modified by making tiny changes so that the other messages remain identical. A good example is the Yahalom family of protocols.
3. Some protocols have messages that cannot be refined by foldm because the key necessary to decrypt certain message components must be received from another message or from a trust management database query. This leaves a message component that will match any other message, so such protocols cannot be paired with a large number of other protocols. One example is the S role of Yahalom.
4. For many protocols, there is dependence among the pattern-matching in the continuation of message reception. (One example is the P role of the Woo Lam Mutual protocol.) As a result, only the independent pattern is substituted into the original message reception pattern. This leaves a variable in the pattern that matches all messages.

Table 1b presents the results when analyzing the sessions of each protocol role. It may seem odd that the $\mathsf{match}_{\iota(\tau)}$ analysis is able to verify any sessions, given our argument against match_τ. Why should removing the initial message make any difference? In 31.4% of the protocols, the protocol receives *only* a single, initial message. We have also inspected the protocols that the most permissive session-based analysis rules out.

1. Some messages simply do not contain a unique value. A prominent example is the A role of many variants of the Andrew Secure RPC protocol.
2. Some roles have the same problems listed above as (3) and (4), except that in these instances the lack of further refinement hides a unique value. One example is the C role of the Splice/AS protocol.

Performance. Computing these tables takes about two minutes.

4 Dispatching

Our analysis determines when there is no message that could be confused during any run of two protocols. We can use this property to build a dispatching algorithm. The algorithm is very simple: forward every incoming message to every protocol handler. (For sessions, we must recognize the initial message and create a new session; otherwise, forward the message to each session.)

This algorithm is correct because every message that is accepted by *some* protocol (session) is only accepted by *one* protocol (session), according to the overlap property. This (absurd) algorithm makes no attempt to determine which protocol an incoming message is actually intended for. This is clearly inefficient. Yet, it shows that distinct message spaces are sufficient for dispatching.

In a network load-balancing setting, where "forwarding a message" actually corresponds to using network bandwidth, this algorithm betrays the intent of load-balancing. On a single machine, where "forwarding a message" corresponds to invoking a handling routine, there are two major costs: (1) a linear search through the various protocol/session handlers; and, (2) the CPU cost associated with each of these handlers. In some scenarios, cost 2 is negligible because most network servers are not CPU-bound. However, since we are dealing with *cryptographic* protocols, the cost of performing decryption only to find an incorrect nonce, etc., is likely to be prohibitive.

A better algorithm would keep a mapping from input message patterns to underlying sessions and efficiently compare new messages with patterns in the mapping prior to delivery. The main problem with this mapping algorithm is that it requires *trust* in the dispatcher: the dispatcher must look inside encrypted components of messages to determine which protocol (session) they belong to. In the next section we discuss how to (a) minimize and (b) characterize the amount of trust that must be given to a dispatcher of this sort to perform correct dispatching.

5 Optimization

Our task in this section is to determine how much trust, in the form of secret data (e.g., keys), must be given to a dispatcher to inspect incoming messages to the point that they can be distinguished. First, we will formalize how deep a dispatcher can inspect any particular message with a certain amount of information. Second, we will describe the process that determines the optimal trust for any pair of protocols (or any pair of sessions of one protocol.) Finally, we formalize the security repercussions of this trust. The end result of this section is a metric of how efficient dispatching can be for a protocol; all protocols should aspire to require no trust in the dispatcher.

Message Redaction. Suppose that a message is described by the pattern $(a, \{|b|\}_k)$. If the inspector of this message does not know key k, then in general[5] this message is not distinguishable from $(a, *)$. We call this the *redaction* of pattern $(a, \{|b|\}_k)$ under an environment that does not contain k. We write $m \downarrow^{\sigma}$ to denote the redaction of message m under σ. This is defined in Figure 4.

[5] There are kinds of encryption that allow parties without knowledge of a key to know that some message is encrypted by *that* key but still not know the contents of the message.

$$\textsc{Nil}\quad \mathrm{nil}\downarrow^{\sigma} = \mathrm{nil} \qquad \textsc{Var}\quad v\downarrow^{\sigma} = v \qquad \textsc{Const}\quad k\downarrow^{\sigma} = k \qquad \textsc{Join}\quad (m,m')\downarrow^{\sigma} = (m\downarrow^{\sigma}, m'\downarrow^{\sigma})$$

$$\textsc{Hash}\quad \frac{\sigma \vdash_s \mathit{hash}(m)}{\mathit{hash}(m)\downarrow^{\sigma} = \mathit{hash}(m)} \qquad \textsc{Hash (Wild)}\quad \frac{\sigma \nvdash_s \mathit{hash}(m)}{\mathit{hash}(m)\downarrow^{\sigma} = *} \qquad \textsc{SymEnc}\quad \frac{k \in \sigma}{\{|m|\}_k\downarrow^{\sigma} = \{|m\downarrow^{\sigma}|\}_k}$$

$$\textsc{SymEnc (Wild)}\quad \frac{k \notin \sigma}{\{|m|\}_k\downarrow^{\sigma} = *} \qquad \dots \qquad \textsc{Bind}\quad < v = m >\downarrow^{\sigma} = < v = m\downarrow^{\sigma} >$$

Fig. 4. Message Redaction

Theorem 3 *A receiver in environment* σ *can interpret* $m\downarrow^{\sigma}$*: for all* σ *and* m, $\sigma \vdash_r m\downarrow^{\sigma}$.

Theorem 4 *Every message that is matched by* m *is matched by* $m\downarrow^{\sigma}$*. (Sec. 3.2)*

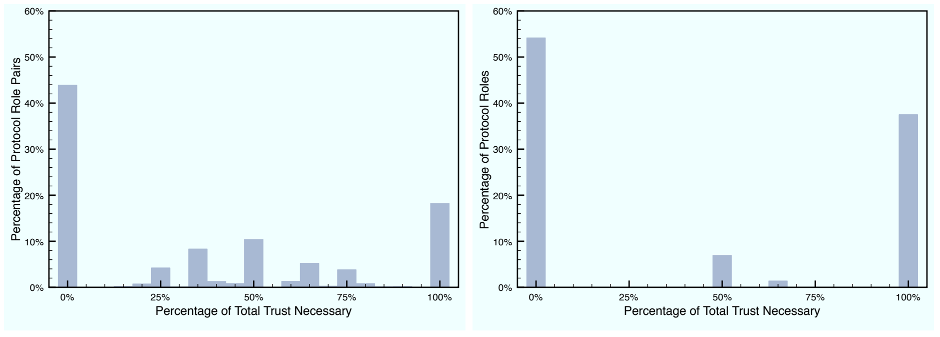
Theorem 5 $\sigma \vdash_r m$ *implies* $m\downarrow^{\sigma} = m$*.*

These theorems establish that $m\downarrow^{\sigma}$ captures the view that a dispatcher, that is trusted with σ only, has of a message m. The next task is to minimize σ while ensuring that match can rule out potential message confusion.

Minimizing σ. Suppose we compare $m = (\{|b|\}_k, \{|c|\}_j)$ with $m' = (\{|b'|\}_{k'}, \{|c'|\}_{j'})$, where b and b' are unique values of their respective protocols, with $\mathsf{match}_{\tau+\delta}$. Because b and b' are unique, the analysis, and therefore the dispatcher, needs to look at b and b' only to ensure that these message patterns cannot describe the same messages. This means that even though the patterns mention the keys k and j (k' and j'), only k (k') is necessary to distinguish the messages. Another way of putting this is that $m\downarrow^{\{k\}} = (\{|b|\}_k, *)$ does not overlap with $m'\downarrow^{\{k'\}} = (\{|b'|\}_{k'}, *)$, according to $\mathsf{match}_{\tau+\delta}$.

We prove that if m and m' cannot be confused according to match, then there is a computable smallest set σ, such that $m\downarrow^{\sigma}$ also cannot be confused with $m'\downarrow^{\sigma}$ according to match. We prove this by first showing that for all m, there is a set $\mathcal{V}_m$, such that for all σ, $m\downarrow^{\mathcal{V}_m\cup\sigma} = m\downarrow^{\mathcal{V}_m}$. In other words, there is a "strongest" set for $\downarrow$ that cannot be improved. This set is the set σ such that $\sigma \vdash_r m$. Our brute-force search construction algorithm then considers each subset of $\mathcal{V}_m$ ($\mathcal{V}_{m'}$) and selects the smallest subset such that the two messages are still distinct after $\downarrow$.

We have run this optimization on our test-suite of 121 protocol roles; it takes about one minute total to complete. Figure 5a breaks down protocol pairs according to the percentage of their keys required to establish trust. This graph shows that 43% of protocol pairs do not require *any* trust to properly dispatch. The other end of the graph shows that only 18% of all protocol pairs require complete trust in the dispatcher. Figure 5b shows the same statistics for protocol sessions. In this situation, 54% of the protocol roles do not require any trust for the dispatcher to distinguish sessions, while 37% require complete trust. These results were calculated in 7.6 minutes and 1.6 seconds respectively.



(a) **Fig. 5.** Trust Optimization Graphs (b)

These experiments indicate that it is very fruitful to pursue optimizing the amount of trust given to a dispatcher. However, we have not yet characterized the security considerations of this trust. We do so in the next section.

Managing Trust. In previous sections, we have discussed how much trust to give to a load-balancer so it can dispatch messages correctly. In this section, we provide a mechanism for determining the security impact of that trust.

Recall that a protocol is specified as a *strand*: a list of messages to send and receive. We have formalized "trust" as a set of keys (and other data) to be shared with a load-balancer. We define a strand transformation $\uparrow^k$ that transforms a strand s such that it shares k by sending a particular message containing k as soon as possible. (It is trivial to lift $\uparrow^k$ to share multiple values.) We define $s\uparrow^k$ as follows:

$$
\begin{aligned}
(sd \rightarrow s)\uparrow^k &= sd \rightarrow +(\mathsf{LB}, v) \rightarrow s \text{ if } k \in \mathit{bound}(sd) \\
(sd \rightarrow s)\uparrow^k &= sd \rightarrow (s\uparrow^k) \text{ if } k \notin \mathit{bound}(sd) \\
.\uparrow^k &= .
\end{aligned}
$$

(This definition clearly preserves well-formedness and performs its task.) In this definition the tag LB indicates that this value is shared with the load-balancer by some means. Depending on the constraints of the environment, this means can be assumed to be perfectly secure or have some specific implementation (e.g., by using a long-term shared key or public-key encryption.)

Since $s\uparrow^k$ is a strand, it can be analyzed using existing tools and techniques [4, 6, 10, 16] to determine the impact of an adversary load-balancer.

6 Insights

The development of the message space overlap analysis and the trust optimization give us insight into *why* and *how* cryptographic protocol message spaces do not overlap.

The effectiveness of match_τ for pairs of protocols demonstrates that it is primarily *shape* that prevents overlap between different protocols. This corresponds with our intuitions, because protocols typically use dissimilar formats.

The disparity between match_τ and match_δ demonstrates that for pairs of protocol sessions, it is uniquely originating values that prevent overlap. Again, this corresponds with our intuitions, because nonces are consciously designed to prevent replay attacks and ensure freshness, which corresponds to the goal of identifying sessions.

The statistical differences between these two analyses in different settings allow us to make these conclusions in a coarse way. But the trust optimization process answers the real question: "Why do two message spaces not overlap?"

When the trust optimization process redacts a message, it is removing the parts of the message that are *not* useful for distinguishing that protocol (session). This means that what remains *is* useful, and therefore the fully redacted message is *only* what is necessary to ensure that there is no message space overlap. Thus, for any two protocols (sessions), it is the trust optimization that explains why there is no overlap.

7 Related Work

Previous Work. In prior work with Guttman and Ramsdell [13], we only addressed the question of when a protocol role supports the use of multiple sessions. In addition, that approach was significantly different from the one presented here. Though we presented a program transformation similar to foldm, we did not formalize the correctness of the transformation. Second, we used only the naïve dispatching algorithm and did not investigate a more useful algorithm. Third, we did not consider pairs of protocols. Therefore, the current presentation is more rigorous, practical, and general.

Our previous problem was only to inspect protocol role message patterns for the presence of distinguishing (unique) values. This is clearly incorrect in the case of protocol role pairs. Consider the role A, which accepts the message M_a, then $(N_a,*)$, and role B, which accepts the message M_b, then $(*,N_b)$, where N_x is a local nonce for x. Each message pattern of each role contains a distinguishing value, so it passes the analysis. But it is not deployable with the other protocol because it is not possible to unambiguously deliver the message (N_a,N_b) after the messages M_a and M_b have been delivered.

It is actually worse than this. We can encode these two protocols as one protocol: accept either M_a or M_b, then depending on the first message, accept $(N_a,*)$ or $(*,N_b)$. Our earlier analysis would ignore the initial messages (which is problematic in itself if M_a and M_b overlap), then check all the patterns in each branch, and report success. This is clearly erroneous because it is possible to confuse an A session with a B session.

This work avoids these problems by directly phrasing the problem in terms of deciding message overlap—the real property of interest rather than a proxy to it as distinguishing values were. It is useful to point out, however, that the earlier work was sound for protocol roles that did not contain branching, which is an very large segment of our test suite. Our use of Coq ensures that our analysis is correct for *all* protocols.

Dispatching. The Guttman and Thayer [9] notion of protocol independence through disjoint encryption and a related work by Cortier et al. [3] study the conditions under which security properties of cryptographic protocols are preserved under composition with one or more other protocols. This is an important problem, since it ensures that it is *safe* to compose protocols. A fundamental result of the Guttman study shows that different protocols must not encrypt similar patterns by the same keys—a similar conclusion to some of our work. However, our work complements theirs by studying whether it is *possible* to compose protocols and, in particular, how efficient such a multiplexer can be. Ideally both of these problems must be addressed before deployment.

Detecting type-flaw attacks [2, 11, 14] is a similar problem to ours. These attacks are based on the inability of a protocol message receiver to unambiguously determine the shape of a message. For example, a nonce may be sent where the receiver expects a key, a composite message may be given in place of a key, etc. These attacks are often effective when they force a regular participant into using known values as if they were keys. Detecting when a particular attack is a type-flaw attack, or when components of a regular protocol execution may be used as such, is similar to our problem. These analyses try to determine when sent message components can be confused with what a regular participant expects. However, in these circumstances a peculiar notion of message matching captures the ambiguity in bit patterns. Some analyses use size-based matching where any message of n-bits can be accepted by a pattern expecting n-bits; for example, an n-bit nonce can be considered an n-bit key. Others assume that message structure is discernible but the leaf-types are not, so a nonce paired with a nonce cannot be interpreted as single nonce, but it may be interpreted as a nonce paired with a key. Our analysis is similar in spirit but differs in the notion of message overlap: we assume that message shapes can be encoded reliably.

Optimization. The problem of optimizing the amount of trust given to a dispatch is very similar in spirit to ordering of pattern-matching clauses [12] and ordering rules in a firewall or router [1], which are both similar to the decision tree reduction problem. However, our domain is much simpler than the general domain of these problems and the constants are much smaller ($|\mathcal{V}_m|$ is rarely greater than 3 for most protocols), so we are not afflicted with many of the motivating concerns in those areas. Even so, these problems really serve only as guidelines for the actual optimization process, not the formulation of the solution (i.e., $\downarrow_\sigma$).

8 Conclusion

We have presented an analysis (match) that determines if there is an overlap in the message space of different protocols (or sessions of the same protocol.) We have shown how it is important to look at real protocols in the development of this analysis (in our case, the SPORE repository [15].) By looking at real protocols, we learned that it was necessary to (1) refine protocol specifications (foldm), (2) incorporate cryptographic assumptions about unique values (match_δ), and (3) take special consideration of the initial messages of a protocol ($\mathsf{match}_{\iota(\delta)}$).

We have shown how this analysis and the message space overlap property can be used to provide the correctness proof of a dispatching algorithm. We have discussed the performance implications of this algorithm and pointed toward the essential features of a better algorithm. We have developed a formalization ($\downarrow_\sigma$) of the "view" that a partially trusted dispatcher has of messages. We have presented an optimization routine that minimizes the amount of trust necessary for match to succeed on a protocol pair. We have presented the results of this analysis for the SPORE repository. We have also formalized the modifications ($\uparrow^k$) that must be made to a protocol in order to enable trust of a load-balancer. Lastly, we have discussed how this optimization explains *why* there is no overlap between two message spaces.

The entire work was formally verified in the Coq theorem proof assistant to increase confidence in our results.

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. A. Begel, S. McCanne, and S. L. Graham. BPF+: exploiting global data-flow optimization in a generalized packet filter architecture. In *Symposium on Communications, Architectures and Protocols*, 1999.
2. C. Bodei, P. Degano, H. Gao, and L. Brodo. Detecting and preventing type flaws: a control flow analysis with tags. *Electronic Notes in Theoretical Computer Science*, 194(1):3–22, 2007.
3. V. Cortier, J. Delaitre, and S. Delaune. Safely Composing Security Protocols. In *Conference on Foundations of Software Technology and Theoretical Computer Science*, 2007.
4. S. F. Doghmi, J. D. Guttman, and F. J. Thayer. Skeletons, homomorphisms, and shapes: Characterizing protocol executions. *Electronic Notes in Theoretical Computer Science*, 173:85–102, 2007.
5. D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.
6. F. J. T. Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *IEEE Symposium on Security and Privacy*, 1998.
7. J. D. Guttman. Authentication tests and disjoint encryption: a design method for security protocols. *Journal of Computer Security*, 12(3/4):409–433, 2004.
8. J. D. Guttman, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Programming cryptographic protocols. In *Trust in Global Computing*, 2005.
9. J. D. Guttman and F. J. Thayer. Protocol independence through disjoint encryption. In *Computer Security Foundations Workshop*, 2000.
10. J. D. Guttman and F. J. Thayer. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, June 2002.
11. J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *Computer Security Foundations Workshop*, 2000.
12. P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Programming Language Design and Implementation*, 1996.
13. J. McCarthy, J. D. Guttman, J. D. Ramsdell, and S. Krishnamurthi. Compiling cryptographic protocols for deployment on the Web. In *World Wide Web*, pages 687–696, 2007.
14. C. Meadows. Identifying potential type confusion in authenticated messages. In *Computer Security Foundations Workshop*, 2002.
15. Project EVA. Security protocols open repository. http://www.lsv.ens-cachan.fr/spore/, 2007.
16. D. X. Song. Athena: a new efficient automated checker for security protocol analysis. In *Computer Security Foundations Workshop*, 1999.
17. F. J. Thayer, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2/3):191–230, 1999.
18. The Coq development team. *The Coq proof assistant reference manual*, 8.1 edition, 2007.