Compiling Cryptographic Protocols for Deployment on the Web

Jay McCarthy
Brown University

Joshua D. Guttman
MITRE Corporation

John D. Ramsdell
MITRE Corporation

Shriram Krishnamurthi
Brown University

ABSTRACT

Cryptographic protocols are useful for trust engineering in Web transactions. The Cryptographic Protocol Programming Language (CPPL) provides a model wherein trust management annotations are attached to protocol actions, and are used to constrain the behavior of a protocol participant to be compatible with its own trust policy.

The first implementation of CPPL generated stand-alone, single-session servers, making it unsuitable for deploying protocols on the Web. We describe a new compiler that uses a constraint-based analysis to produce multi-session server programs. The resulting programs run without persistent TCP connections for deployment on traditional Web servers. Most importantly, the compiler preserves existing proofs about the protocols. We present an enhanced version of the CPPL language, discuss the generation and use of constraints, show their use in the compiler, formalize the preservation of properties, present subtleties, and outline implementation details.

Categories and Subject Descriptors

C.2 [**Network Protocols**]: Protocol Verification

General Terms

Performance, Security, Languages, Theory, Verification

Keywords

CPPL, Cryptographic Protocols, HTTP, Sessions

1. PROBLEM AND MOTIVATION

A growing array of services, such as third-party credit-card handling as offered by several banks, is now offered via Web-based protocols. These services need to be reliable in several ways: (a) they must be cryptographically trustworthy, (b) their implementations must be sound, and (c) their implementations must scale to handle high customer loads.

As the number of custom protocols increases, there is increasing interest in domain-specific programming languages for defining cryptographic protocols. A good representative example is CPPL [4], which makes protocol specification relatively easy (compared to writing the definition in a regular programming language), and which automatically compiles to trustworthy cryptographic libraries to avoid programmer errors in their selection and use. By also supporting effective rely-guarantee proof techniques for reasoning about security properties [5], CPPL discharges reliability requirements (a) and (b). For instance, CPPL has successfully described almost all of the protocols in the SPORE [10] repository.

The compiler for CPPL, unfortunately, does not produce scalable implementations. It generates executables that open a specified TCP port and keep it open while handling one protocol run (or "session"). While this is sufficient for demonstration purposes, this reliance on open ports and on single sessions makes the current CPPL implementation untenable for scalable deployment over the stateless HTTP Web protocol, which has been a cornerstone of the Web.

Compiling to handle multiple sessions atop HTTP is not trivial. It requires being able to *unambiguously* determine which session should receive an incoming message; faulty identification means the actual recipient may get into an inconsistent state or even leak information, while the intended session will starve. As the paper explains, precise definition of the "right" session, and of progress of a session, is quite subtle, especially in the presence of cryptography (Sec. 5.3).

This work presents a static analysis that generates constraints from a protocol description and compiles them into an extended description of the protocol (Sec. 4). A generic dispatcher, deployable on a stock Web server, employs the embedded constraints to route each incoming protocol message to the correct session (Sec. 5). We also briefly describe the compiler that implements this process (Sec. 6).

Not all protocols have sufficient constraints to make them dispatchable (Sec. 3). It can be tempting to modify protocols to circumvent this problem, but this is not always acceptable. Obviously, it can be difficult to modify deployed Web service clients. More importantly, the protocols have already been subject to proof, and changes to them can potentially invalidate these proofs—and as experience has shown, protocols are already quite subtle and are sensitive to small changes. Our analysis nevertheless points to the source of weakness in the protocol to suggest places where it could be modified (Sec. 6).

Though this is a practical paper, it must perforce present a theoretical account. This theory is to qualify and prove that (1) the compilation process preserves the existing proofs about CPPL programs and (2) the dispatcher never delivers a message to the wrong session (Sec. 4.5 and Sec. 5.3). These proofs are necessary to demonstrate that we have preserved criteria (a) and (b) while enabling (c).

Reciprocating the benefits that Web technology confers on CPPL, there are benefits that CPPL offers the Web. As increasingly important applications reside on the Web and as security concerns correspondingly grow, programmers would benefit from employing secure protocols to establish trust between client and server. Systems like CPPL provide a way of describing and analyzing such protocols. This work takes a step toward making the fruits of such analysis accessible to Web programmers. Web programmers will eventually be able to analyze the protocols they currently employ atop SSL and AJAX communication, as they must do.

SSL by itself only authenticates the server to the client, not the other way around. Client SSL does authenticate clients to servers. Unfortunately client SSL is often not enabled. Even if it is, however, it does not help in the case where a client wants to instantiate multiple distinct sessions of a service, as when running distinct sessions in different tabs of a Web browser. Thus, even client SSL is insufficient to provide the key criterion that we call *distinguishability*, which we define and show how to implement in Section 5.

2. INTRODUCTION TO CPPL

CPPL is a domain specific language for expressing cryptographic protocols with trust annotations. It matches the level of abstraction of the Dolev-Yao model [2], in the sense that the programmer regards the cryptographic primitives as black boxes, and concentrates on the structural aspects of the protocol. CPPL allows the programmer to control protocol actions using trust constraints [5], so that an action such as transmitting a message will occur only when the indicated trust constraint is satisfied.

Semantic Interpretation. The CPPL semantics identifies a set of *strands* [11] annotated with trust formulas as the meaning of a role in a protocol. A strand merely specifies what messages are sent and received. The representation does not specify to whom message are sent or from whence they are received. This corresponds to a model that allows an adversary to have 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 face of a powerful adversary.

A strand as the meaning of a protocol is *local* in the sense that it describes what one principal P does. It says nothing about how messages are routed on a network; nothing about what another principal P' does with messages received from P; nothing about how another principal P' created the messages that P receives; etc. In essence, it describes only a single principal executing a single run.

We must specify how to combine strands to fully reason about CPPL programs. This *global* semantics has been provided by earlier work [4]. It defines a regular strand—a strand that conforms to the semantic interpretation of some protocol. It then explains how we may reason that multiple regular strands must be communicating, given the values they share. We may then reason about whether secrecy and other protocol goals are reached. However, the details of all this are not important for our purposes, because this work focuses on preserving the local semantics under a different implementation, rather than developing a new global semantics.

Trust Management. The sender of a message must guarantee the formula associated with sending the message, by showing the formula is a consequence of its local theory. The receiver of a message relies on a formula associated with receiving the message by adding the formula to its local theory for use in later deductions. Relying on a formula is justified when the protocol is sound.

A protocol is sound if in every execution, whenever a message is received and its formula is relied upon, there were corresponding message transmissions with guaranteed formulas that allow it to be deduced. A protocol designer is responsible for demonstrating a protocol is sound. A sound protocol is easily implemented using CPPL. The language is designed around a few simple concepts: branching on incoming and outgoing messages, and consulting a trust management engine [5] during messaging. These simple concepts are natural to the designer of cryptographic protocols and proof authors. This is beneficial because there is no impedance mismatch between the description a designer uses to show soundness and the CPPL program that implements the protocol.

$$\begin{array}{rcl}
p & \rightarrow & \text{proc } f\ y^*\ \Psi\ c \\
c & \rightarrow & \text{return } \Phi\ x^* \\
 & | & \text{let } x = \text{new in } c \\
 & | & \text{let } x = \text{accept in } c \\
 & | & \text{let } x = \text{channel } y \text{ in } c \\
 & | & (sb^*) \mid (x\ rb^*) \mid (cb^*) \\
 & | & (x\ mb^*) \\
sb & \rightarrow & \text{send } \Phi\ x\ m\ c \\
rb & \rightarrow & \text{recv } m\ \Psi\ c \\
mb & \rightarrow & \text{match } m\ c \\
cb & \rightarrow & \text{call } \Phi\ f\ x^*\ y^*\ \Psi\ c \\
m & \rightarrow & x \quad | \quad m, m' \quad | \quad r\,m \\
 & | & (m) \quad | \quad [\,m\,]\,x \quad | \quad \{\,m\,\}\,x
\end{array}$$

Figure 1: CPPL_m Language

The Core Language. The syntax of the CPPL_m core language is presented in Figure 1. The CPPL core language has procedure declarations and seven types of code statements. A minor extension of the language, CPPL_m, introduces one more code statement. Therefore, every CPPL program is syntactically a CPPL_m program, although we often distinguish between the two languages in the body.

The previous work did not describe match statements, because they do not induce message transmission or require message reception. They were therefore left out of the earlier semantics, although they were included in the earlier implementation. We describe them because they play an important role in our work. In addition, they are essential to building useful protocols.

Programming language identifiers are indicated by x and y, lists of identifiers by x^*, message tags (such as `High`) by r, and procedure names by f. When used to concatenate message patterns, the comma operator is right associative, and tagging binds less tightly than comma. The language has syntax for guarantees and relies—by convention we write guarantees as Φ and relies as Ψ—which are finite lists of trust management formulas. We use finite lists, which we interpret conjunctively.

A procedure declaration specifies the name f of the procedure, a list (y^*) of formal parameters, and a list of preconditions Ψ involving the formal parameters. The body of the procedure is a code statement c. A code statement may be: a return instruction, which specifies a list of postconditions Φ and return parameters (x^*); a let-statement; or a list of send branches, receive branches, call branches, or match branches. An identifier x is either a lowercase identifier `id`, or else an identifier with typing information `id:type`. We write $\mathtt{ide}(X)$ for the set of identifiers used in the phrase X. We write $\mathit{Free}(\Sigma, X)$ for the set of identifiers free in X, where Σ represents bound identifiers.

A well-formed code statement c with two return statements at different locations must have the same postconditions Φ and return parameters x^*.

The Runtime Environment. The language is organized around a specific view of protocol behavior. In this view, as a principal executes a single local run of a protocol, it builds up an *environment* that binds identifiers to values encountered. These bindings are commitments, never to be updated; once a value has been bound to an identifier, future occurrences of that identifier must match the value or else execution of this run aborts. In particular, when a known value is expected in an incoming message, any other value will prevent execution of this run from continuing.

Informal Execution Semantics. To explain how procedures execute, we first introduce an auxiliary notion: *guaranteeing* formulas Φ in a runtime environment. This means asking the runtime trust management system to attempt to ascertain the formulas Φ. Identifiers in Φ already bound in the runtime environment are instantiated to the associated values. Identifiers not yet bound in the runtime environment are instantiated by the trust management system, if possible, to values that make the formulas Φ true. The runtime environment extended with these new bindings is the result of successfully guaranteeing Φ. If the runtime trust management system fails to establish an instance of Φ the guarantee fails.

To execute a **return** statement, we attempt to guarantee the formulas Φ. If successful, we select from the resulting environment the values of each of the return parameters x^*; these values are returned to the caller. If the attempt to guarantee Φ fails, execution terminates abnormally, and the caller is informed of the failure. The caller receives no parameter values in case of failure.

To execute a list of **send branches**, the runtime trust management system selects a branch within which it can successfully guarantee the formulas Φ. The message pattern m specified on this branch, instantiated using the values in the resulting extended runtime environment, is then transmitted. Execution proceeds with the continuation[1] c embedded within this send branch in the extended environment. If the runtime trust management system fails to guarantee the formulas Φ on any send branch, then execution terminates abnormally, and the caller is informed of the failure.

To execute a list of **receive branches** with identifier x, the runtime environment is consulted for the value bound to x. This value should be a channel. When a message is received over this channel, the message is matched against the patterns m within the receive branches. In a successful match, the message must agree with the runtime environment for identifiers in m that are already associated with a value. Other identifiers in m will be bound to the values observed in the incoming message, yielding an extended runtime environment. If at least one receive branch has a successful match, one such branch is selected. The formulas Ψ are instantiated using the extended runtime environment, and supplied to the runtime trust management system as additional premises. Execution proceeds with the continuation c embedded within this receive branch in the extended environment. If no receive branch has a successful match, then execution terminates abnormally, and the caller is informed of the failure.

To execute a list of **call branches**, the system treats the call branches as sends followed by receives. The system acts as if the principal `sends` the message x^* after guaranteeing the formulas Φ, then `recvs` the message y^* and relies on the formulas Ψ. That is, the the runtime trust management system selects a branch, within which it can successfully guarantee the formulas Φ. It calls the associated subprotocol procedure f with the parameters x^* instantiated using the values in the resulting extended runtime environment. This procedure may return normally, in which case it supplies values for the parameters y^*; execution continues with the embedded continuation c, using the extended runtime environment. The instances of the formulas Ψ are supplied to the runtime trust management system as additional premises during execution of c. If the f does not return normally, then execution may continue with a different call branch; execution proceeds in the original environment, without any extension from the abnormally terminated call branch.

To execute a list of **match branches** with identifier x, the runtime environment is consulted for the value bound to x. This value is matched against the patterns m within the match branches. In a successful match, the value must agree with the runtime environment for identifiers in m that are already associated with a value. Other identifiers in m will be bound to the values contained in the value of x, yielding an extended runtime environment. If at least one match branch has a successful match, one such branch is selected. Execution proceeds with the continuation c embedded within this match branch in the extended environment. If no match branch has a successful match, then execution terminates abnormally, and the caller is informed of the failure.

3. INFORMAL SOLUTION

Having summarized CPPL_m, we return to the problem of the paper: deploying protocols on the Web. On the Web, a single server usually runs several sessions of a protocol. The essence of the deployment problem is thus determining which (if any) of many sessions of a protocol should receive an incoming message. What information does a message contain that can help a dispatcher, and how can we use it?

Our strategy is to inspect the protocol encoded in the CPPL_m program and determine whether, at each message reception, the message's content is sufficient for uniquely identifying a session. Besides message structure, this largely boils down to finding distinguishing values in the message that are visible at that point.

The set of distinguishing values is not fixed. Many applications may manifest such values specific to that application, such as the user's identity. In all applications, however, nonces are globally unique, and therefore distinguish the session that generated them. In this work, we will therefore provide the most conservative analysis by treating only locally-originating nonces as distinguishing values. Our solution, however, easily generalizes to other, including application-specific, notions of distinction. We note where such values can be used as we describe our analysis.

These distinguishing values are assumed to be unique and nonforgeable, like nonces in the Dolev-Yao [2] model, and therefore not vulnerable to duplication by attackers. This is essential to the correctness of our analysis.

We will refer to a message that contains a distinguishing value as *dispatchable*, because the destination session is identifiable. We will also call a code phrase dispatchable if all messages received within it are dispatchable.

We may be tempted to simply inspect each branch of each `recv` statement and ensure that the message pattern contains an identifier bound to a distinguishing value, but this will reject programs that should be accepted, as shown below. `match` statements contain additional information that will allow more `recv` statements to be considered dispatchable. This fact will complicate our analysis, but will produce a more useful answer.[2]

For example, the following procedure clearly contains a `recv` statement that contains a distinguishing value and therefore is dispatchable:

```
1 proc example1 (chnl, v) _
2  let n = new in
```

[1]Some readers may be unfamiliar with or scared of this term. A continuation is simply "what remains to be done" in a computation. Notice that CPPL_m contains no sequencing operation; instead, what computation occurs after any given statement is specified directly as part of that statement, by the c at the end. Each c is that statement's continuation.

[2]Still more information could be extracted from the trust management logic database. We could pursue this angle, but do not to avoid creating dependencies on the particular trust management logic, thereby enabling users to employ whichever one is most appropriate for their setting.

```
3   (send _ chnl n
4    (chnl recv n _
5          return _ v))
```

The message received on line 4 is exactly the nonce n.

However, the following procedure *appears* to contain a recv statement that does not contain a distinguishing value and therefore is not dispatchable:

```
1 proc example2 (chnl, v) _
2  let n = new in
3   (send _ chnl n
4    (chnl recv m _
5          (m match n
6             return _ v)))
```

The message received on line 4 is a completely new binding, m, and therefore not identifiable directly as the nonce n. However, on line 5 of the program, that value is checked against n, so the program will succeed *only* if $\mathtt{m} = \mathtt{n}$. Thus, we should consider this program as being dispatchable.

This last example demonstrates a general aspect of match statements: they impose constraints on messages that are received *earlier* in the program. In example2, the match on line 5 imposes the constraint that the message received on line 4, m, must match the pattern n, i.e., must equal n.

The following pattern, common to *commitment* protocols [3], demonstrates a more complicated constraint imposition:

```
1 proc commit (chnl) _
2  (chnl recv payload _
3        ...
4        (chnl recv key _
5              (payload match {objective} key
6                       return _ objective)))
```

In this protocol, one participants receives an encrypted payload (line 2), performs some further work and communication (elided in line 3), then receives a commitment from the other participant in the form of a key (line 4), to the data sent earlier (line 5), which contains the objective (line 6). In this example, the match on line 5 imposes the constraint on the message received on line 2 that "There must exist some value key which decrypts payload".

Another kind of constraint is introduced by branching in the language. Consider the following contrived example:

```
1 proc example3 (chnl, left, right) _
2  (chnl recv prod _
3        (prod match left
4               return _ left
5              match right
6               return _ right))
```

In this example, the match beginning on line 3 imposes the constraint on the message received on line 2 that "prod must match either left or right."

Neither of these examples is dispatchable, as neither the message patterns nor the constraints imply that the message contains a distinguishing value. However, more complicated protocols with similar constraints are dispatchable. In Sec. 5.1 we develop how to make this judgment.

Workflow. A protocol engineer uses our tool to turn a CPPL$_\mathrm{m}$ source program into an executable that can be used to deploy their protocol. If the protocol passes our analysis, the engineer will have a deployable binary. If not, they see an error indicating the message of the protocol that does not contain a distinguishing value.

$$
\begin{array}{rcll}
\mu & \rightarrow & \top & [\textsc{Top}] \\
 & \mid & x \sim m \; ; \; \mu & [\textsc{Match}] \\
 & \mid & \mu + \mu' & [\textsc{Or}] \\
 & \mid & \exists x^{*} . \mu & [\textsc{Exists}]
\end{array}
$$

Figure 2: Constraint Language Syntax

Solution Overview. Our system consumes a CPPL$_\mathrm{m}$ file, generates all the constraints on incoming messages imposed by the rest of the program, checks that the incoming messages all contain some distinguishing value, then produces an executable. This executable uses a runtime library that provides cryptographic communication and checks for the satisfaction of the computed constraints. This executable is combined with a server that consumes messages and dispatches them to sessions.

The server can use a very naïve dispatching algorithm. Whenever it receives a message, it can attempt to deliver it to all sessions, stopping when one session has accepted the message. If no session accepts the message, then it can attempt to create a new session for the message. Only if this final step fails is the message rejected.

The analysis of this paper makes such a dispatching algorithm sufficient. In the absence of such an analysis, the algorithm would be doubly erroneous: it might deliver messages to the wrong session, i.e., one intended for session *A* might resume session *B*, causing both *B* to incorrectly resume and *A* to incorrectly starve. Sec. 5 describes the algorithm formally and justifies our claim that it is not susceptible to such errors.

4. CONSTRAINTS

The heart of this section show how constraints are generated from CPPL$_\mathrm{m}$ phrases (Sec. 4.2). To support this, we describe the language of constraints (Sec. 4.1) and give their semantics. Once we have generated constraints, we show how to annotate CPPL$_\mathrm{m}$ programs with them by translation to an intermediate language, CPPL$_\mathrm{m+c}$, in Sec. 4.3. As a final bookkeeping step, we present the semantics of this annotated language in Sec. 4.4. With semantics in hand, we then show that properties proved about translated CPPL programs are maintained, satisfying one of our primary goals.

4.1 Constraint Language

Fig. 2 specifies the syntax of the constraint language. In this syntax, the ; and $+$ operators are right-associative, and $+$ binds less tightly than ;. The language is best explained through a description of the informal execution semantics. This semantics depends on a runtime environment binding identifiers to values, corresponding to the runtime environment of the CPPL$_\mathrm{m}$ program at the point where constraint satisfaction occurs.

To check a TOP constraint of the form $\top$, nothing need be done, as this constraint is always satisfied.

To check a MATCH constraint of the form $x \sim m; \mu$, the runtime environment is consulted for the value bound to *x*. This value is matched against the pattern *m*. For the match to be successful match, the value must agree with the runtime environment for identifiers in *m* that are already associated with a value. Other values in *m* will be bound to the values contained in the value of *x*, yielding an extended runtime environment. If the match is successful, constraint checking continues with the constraint μ in the extended runtime environment. If μ is satisfied in this environment, then this constraint is satisfied.

An example of a MATCH constraint is the constraint on the recv statement on line 4 in example2 in Sec. 3: $\mathtt{m} \sim \mathtt{n}; \top$. In this con-

$$\textsc{Match}\quad \frac{\sigma_1 = \sigma \oplus \sigma' \qquad \begin{array}{c}\sigma[x] \text{ matches } m' \\ \mathsf{dom}(\sigma') \subseteq \mathsf{ide}(m')\end{array} \qquad \sigma_1 \models \mu}{\sigma \models (x \sim m' \,;\, \mu)}$$

$$\textsc{Or, Left}\quad \frac{\sigma \models \mu}{\sigma \models \mu + \mu'} \qquad \textsc{Or, Right}\quad \frac{\sigma \models \mu'}{\sigma \models \mu + \mu'}$$

$$\textsc{Exists}\quad \frac{\sigma[x \mapsto v]^* \models \mu \text{ for some } v^*}{\sigma \models \exists x^*.\mu} \qquad \textsc{Top}\quad \frac{}{\sigma \models \top}$$

Figure 3: Constraint Satisfaction

straint m is the message received on line 4, n is the pattern used in the match statement on line 5, and the $\top$ constraint comes from the continuation of this match, i.e., the return on line 6.

To check an OR constraint of the form $\mu + \mu'$, the two sub-constraints are checked independently, i.e., the runtime environment is copied in each branch. If one of the sub-constraints is satisfied, this constraint is satisfied and the other need not be consulted.

To check an EXISTS constraint of the form $\exists x^*.\mu$, the runtime environment is extended with values for x^*, if possible, that make the constraint μ satisfied. If there is no assignment of the identifiers x^* to values for which μ may be satisfied, then this constraint is not satisfied.

The formal semantics of constraint satisfaction is given in Fig. 3. The judgments have the form:

$$\sigma \models \mu$$

where the runtime environment is represented by σ and the constraint is represented by μ.

Runtime Implementation. The semantics does not explicitly state how to generate values for the identifiers bound in EXISTS constraints during satisfaction checking. An implementation must solve this problem. The obvious solution is to use unification of identifiers with CPPL_m values and identifiers in match patterns. Identifiers introduced by these constraints are initially unbound. Identifiers are bound by their first comparison in a MATCH constraint. After an identifier has been bound, a failure to unify with a value in a MATCH constraint is a failure of the EXISTS constraint that introduced the identifier.

There are some subtleties, however, due to the use of cryptography. For example, the following constraint can be satisfied with unification:

$$[x \mapsto (0,0)] \models \exists y.x \sim (y,y); \top$$

The following, however, cannot:

$$\sigma \models \exists x, k.M \sim \{x\}_k; x \sim V; \top$$

because it refers to the contents of an encryption for which the key, k, is unknown. We can, however, still perform some checking of this constraint. Specifically, M can be checked to ensure that it is long enough (in bits) to be decrypted to an x that is as long (in bits) as the value of V.

Our compiler also optimizes some constraints at compile-time. Informally, if the identifier introduced in an EXISTS constraint does not appear in any of the message patterns in the sub-constraint, then the EXISTS constraint can be removed. This allows the constraint-solver to short-circuit the unification mechanism in many cases. This optimization could be incorporated into the constraint generation phase of our analysis, but to do so would complicate our analysis and shift attention from the essence of the problem.

4.2 Constraint Generation

This section discusses how to generate constraints from CPPL_m phrases.

For each recv branch, we must find the constraints imposed on the message by the continuation. Each code statement in the continuation of the receive statement can refer to identifiers bound by the message. For example, in the `commit` protocol (Sec. 3) the continuation starting on line 3 imposes constraints on the message received in line 2 through the use of the identifier `payload`. Although in that example this is the only identifier bound by the pattern on line 2, in general a receive pattern may bind any number of identifiers.

Therefore, our constraint generation analysis is defined with respect to (a) the identifiers bound by the message, (b) all bound identifiers, to decide what identifiers are introduced by the message, and (c) a code statement, initially the continuation of the recv.

The phrase "bound by the message" is quite subtle. The message binds identifiers in the recv branch message pattern *and* in any match pattern where the variable being matched was bound by the message. For example, in the `commit` protocol, the message received on line 2 introduces `payload` on line 2 and `objective` on line 5.

We will now present the analysis. The judgments are of the form:

$$\Sigma, \Pi \models c : \mu$$

where Σ represents the set of bound identifiers and Π represents a set of identifiers bound by the message. These judgments can be roughly categorized into a few cases based on the code statements:

- *Statements without continuations,* such as return, that generate a TOP constraint.
- *Statements that introduce bindings,* such as let, recv, send, and call statements that generate an EXISTS constraint for each of the identifiers they bind.
- *Statements that represent branching,* such as recv branches, send branches, and match branches, that introduce OR constraints for each branch.

Examples of each of these categories are presented in Fig. 4. However, match statements do not fit into any of these categories. The judgments for them are shown in Fig. 5.

If the identifier being matched, x, is in Π, then this match represents further use of the original message. Therefore, any identifiers bound by this match must be incorporated into the Π used to analyze the continuation. Furthermore, the pattern used to match the identifier must be incorporated into the constraints via a MATCH constraint. If the identifier being matched is *not* in Π, then this match is treated like any other statement that introduces bindings.

In Fig. 6 we present an example of constraint generation for lines 4 through 6 of the commit protocol example. In this derivation, we use $< i >$ to represent the continuation beginning on line i of the source.

4.3 Translation

Having generated these constraints, we still need to show how to incorporate them into an analysis that checks for the presence of distinguishing values. We provide these constraints to the analysis by translating CPPL_m phrases into another language that contains these constraints as annotations on recv statements.

$$\cfrac{\cfrac{\{\texttt{payload},\texttt{key},\texttt{objective}\},\{\texttt{payload},\texttt{objective}\} \models \ <6> \ : \top \qquad \texttt{payload} \in \{\texttt{payload}\}}{\{\texttt{payload},\texttt{key}\},\{\texttt{payload}\} \models \ <5> \ : \texttt{payload} \sim \{\texttt{objective}\}\ \texttt{key}; \top}}{\{\texttt{payload}\},\{\texttt{payload}\} \models \ <4> \ : \exists \texttt{key}.\texttt{payload} \sim \{\texttt{objective}\}\ \texttt{key}; \top}$$

Figure 6: CPPL$_\mathrm{m}$ Example Constraint Generation

RETURN
$$\Sigma,\Pi \models \mathsf{return}\ \Phi\ x^* \ : \ \top$$

SEND
$$\frac{\{y^*\} = \mathit{Free}(\Sigma,\Phi) \qquad \Sigma \cup \{y^*\},\Pi \models c \ : \ \mu}{\Pi \models \mathsf{send}\ \Phi\ x\ m\ c \ : \ \exists y^*.\mu}$$

SEND BRANCH
$$\frac{\Sigma,\Pi \models sb_0 \ : \ \mu_0 \qquad \ldots \qquad \Sigma,\Pi \models sb_n \ : \ \mu_n}{\Sigma,\Pi \models (sb_0 \ldots sb_n) \ : \ (\mu_0 + \cdots + \mu_n)}$$

Figure 4: CPPL$_\mathrm{m}$ Constraint Generation Examples

MATCH (x IN Π)
$$\frac{\begin{array}{c}\{y^*\} = \mathit{Free}(\Sigma,m)\\ \Sigma \cup \{y^*\},\Pi \cup \{y^*\} \models c \ : \ \mu \qquad x \in \Pi\end{array}}{\Pi \models x\ \mathsf{match}\ m\ c \ : \ x \sim m \ ; \ \mu}$$

MATCH (x NOT IN Π)
$$\frac{\{y^*\} = \mathit{Free}(\Sigma,m) \qquad \Sigma \cup \{y^*\},\Pi \models c \ : \ \mu \qquad x \notin \Pi}{\Pi \models x\ \mathsf{match}\ m\ c \ : \ \exists y^*.\mu}$$

Figure 5: CPPL$_\mathrm{m}$ Constraint Generation: Matching

We will refer to this new language as CPPL$_\mathrm{m+c}$. We specify the syntax below as a modification to CPPL$_\mathrm{m}$. The syntax is identical, except that an annotation has been added to the recv branch case: the constraint, μ, imposed on this statement by its continuation.

$$rb \quad \rightarrow \quad \mathrm{recv}\ m\ \mu\ \Psi\ c$$

The form of translation judgments is:

$$\Sigma \vdash c \rightsquigarrow c'$$

where Σ is a set of bound identifiers, c is a CPPL$_\mathrm{m}$ program statement, and c' is the CPPL$_\mathrm{m+c}$ program statement produced by the translation and eventually run in the server.

Except for the judgment dealing with recv statements, all the judgments are obvious and only carry along information about the introduction of identifiers. Therefore, they are not included in this short presentation. The judgment for recv statements is:

RECEIVE
$$\frac{\begin{array}{c}\Pi_0 = \mathit{Free}(\Sigma,m)\\ \Sigma \cup \Pi_0 \vdash c \rightsquigarrow c' \qquad \Sigma \cup \Pi_0,\Pi_0 \models c \ : \mu\end{array}}{\Sigma \vdash \mathsf{recv}\ m\ \Psi\ c \ \rightsquigarrow\ \mathsf{recv}\ m\ \mu\ \Psi\ c'}$$

The identifiers initially bound by the message pattern are used as the initial set of identifiers bound by the message (Π_0) in the constraint generation process (Sec. 4.2) that analyzes the continuation of the recv statement. The generated constraint, μ, is used in the resulting translated program statement.

MATCH AND RELY
$$\frac{\begin{array}{c}\sigma[x]\ \text{matches}\ m\\ \sigma_1 = \sigma \oplus \sigma' \qquad \mathsf{dom}(\sigma') \subseteq \mathsf{ide}(m) \qquad \sigma_1;\Gamma \vdash \ c : s,\upsilon\end{array}}{\sigma;\Gamma \vdash \ (x\ \texttt{match}\ m\ c\ rb^*) : s,\upsilon}$$

MATCH ALTERNATIVE
$$\frac{\sigma;\Gamma \vdash \ (x\ \ mb^*) : s,\upsilon}{\sigma;\Gamma \vdash \ (x\ \texttt{match}\ m\ c\ mb^*) : s,\upsilon}$$

Figure 8: Semantics of match

4.4 CPPL$_{m+c}$ Local Semantics

In keeping with previous CPPL work [4], we give the semantics of CPPL$_\mathrm{m+c}$ procedures and code statements by describing the *strands* that specify their possible behavior. Each strand is a sequence of transmissions and receptions that is possible for a principal executing this CPPL$_\mathrm{m+c}$ phrase faithfully. As mentioned in Sec. 2, this describes the local nature of protocol execution.

Our local semantics is identical to previous work [4], except in the RECEIVE AND RELY, RECEIVE ALTERNATIVE MATCH AND RELY, and MATCH ALTERNATIVE cases.

The semantics of a receive statement (Fig. 7) is identical to the earlier semantics, except that the extended runtime environment, σ_1, is checked against the μ constraint according to the constraint satisfaction semantics in Fig. 3. (And, for technical reasons the semantics of RECEIVE ALTERNATIVE must be modified for the new syntax.)

The semantics of a match statement (Fig. 8) is very close to the semantics of a receive statement, except: (a) there is no message reception or transmission, and (b) we must explicitly state that the value bound to x in the runtime environment, σ, matches the message pattern, m. This is normally implicit in the description of the strand unleashed by a receive statement.

The semantics described in this section is a replacement for the local semantics of earlier work [4]. The original work also describes a global semantics relating bundles of strands. In this work, there is no reason to modify the global semantics of that earlier work. We elide it for brevity.

4.5 Preservation of Properties

We would like to show that properties proved about CPPL programs remain true about the translated CPPL$_\mathrm{m+c}$ programs.

THEOREM 1. *If a* CPPL *phrase, c, unleashes a strand s by the earlier semantics [4], then the translation of c unleashes strand s by the* CPPL$_{m+c}$ *semantics.*

PROOF. Consider the changes made between CPPL and CPPL$_\mathrm{m+c}$. The two differences are (1) the introduction of match statements and (2) the introduction of constraint satisfaction of recv statements. A CPPL program, which by definition does not include match statements, unleashes the same strand by the CPPL$_\mathrm{m+c}$ semantics, after translation, because (1) no match statements are in-

RECEIVE AND RELY
$$\frac{\sigma_1 = \sigma \oplus \sigma' \qquad \sigma_1 \models \mu \qquad \mathsf{dom}(\sigma') \subseteq \mathsf{ide}(m) \qquad \sigma_1;\Gamma,\Psi\sigma_1 \vdash\ c : s,\upsilon}{\sigma;\Gamma \vdash\ (x\ \mathtt{recv}\ m\ \mu\ \Psi\ c\ rb^*) : (-\mathsf{msg}(x,m)\sigma_1\,,\ \Psi\sigma_1) \Rightarrow s,\upsilon}$$

RECEIVE ALTERNATIVE
$$\frac{\sigma;\Gamma \vdash\ (x\ rb^*) : s,\upsilon}{\sigma;\Gamma \vdash\ (x\ \mathtt{recv}\ m\ \mu\ \Psi\ c\ rb^*) : s,\upsilon}$$

Figure 7: Semantics of recv

troduced by translation and (2) only match statements generate constraints that may not be satisfied. Therefore, the strand unleashed by a CPPL phrase is identical to the strand unleashed the translation of this phrase. □

This is not an *if and only if* statement: because match statements do not cause communication and only reject messages, it is possible for some CPPL$_{\text{m+c}}$ program, p, to unleash a strand s which is also unleashed by some CPPL program, p', such that p is *not* the translation of p'.

Notice that we have not yet discussed whether our compilation of CPPL$_{\text{m+c}}$ programs preserves properties, or, more formally, correctly executes the strand. We have simply shown that the translation of CPPL to CPPL$_{\text{m+c}}$ does not change the strands unleashed by CPPL programs.

5. DISPATCHING

So far we have calculated the constraints imposed on incoming messages, recorded the constraints as annotations in an extended language, and taken a first step towards proving correctness. We still must describe how these constraints can be used to guarantee that all messages received by a protocol contain distinguishing values. We must also describe the algorithm used for dispatching.

Sec. 5.1 describes the analysis that is run on CPPL$_{\text{m+c}}$ programs to ensure that all received messages contain distinguishing values. Sec. 5.2 presents the algorithm used by the dispatching server that relies on this analysis. Sec. 5.3 proves correctness properties on both of these.

5.1 Analyzing CPPL$_{m+c}$ Programs

Given a CPPL$_{\text{m+c}}$ program, or more generally a CPPL$_{\text{m+c}}$ phrase, we must check that all received messages contain a distinguishing value. A message may contain such a distinguishing value directly, in a message pattern, or indirectly, in the constraints imposed on the message by the rest of the program. The analysis to check these conditions is fairly straight-forward, once we understand exactly what we are checking. In particular, we are checking for *visible* distinguishing values.

The notion of *visibility* is somewhat subtle: if a message component contains a distinguishing value but is encrypted by an unknown key, then the distinguishing value is *not* visible. Similarly, if our language modeled hashing, then a distinguishing value would be visible if it was hashed together with any number of visible data. Our definition of visibility is given in Fig. 9.

The majority of the analysis is obvious, merely carrying along information across the structure of the code phrase, and is therefore not included in this short presentation. The judgments are of the form:

$$\nu \vdash c$$

signifying the fact that given the set of distinguishing values ν, all messages received in c contain a visible distinguishing value.

The only interesting judgments are on one variant of let statements, recv statements, and match statements.

$$\begin{aligned}
&\mathtt{vis} : \{x^*\} \times \mu \to \mathtt{bool}\\
&\qquad \mathtt{vis}(\nu,\mu) = \mathtt{vis}(\nu,\emptyset,\mu)\\[1em]
&\mathtt{vis} : \{x^*\} \times \{x^*\} \times \mu \to \mathtt{bool}\\
&\qquad \mathtt{vis}(\nu,\sigma,\mu+\mu') = \mathtt{vis}(\nu,\sigma,\mu) \wedge \mathtt{vis}(\nu,\sigma,\mu')\\
&\qquad \mathtt{vis}(\nu,\sigma,\exists x^*.\mu) = \mathtt{vis}(\nu,\sigma \cup \{x^*\},\mu)\\
&\qquad \mathtt{vis}(\nu,\sigma,\top) = \mathtt{false}\\
&\qquad \mathtt{vis}(\nu,\sigma,x \sim m;\mu) = \mathtt{vis}(\nu,\sigma,m) \vee \mathtt{vis}(\nu,\sigma,\mu)\\[1em]
&\mathtt{vis} : \{x^*\} \times \{x^*\} \times m \to \mathtt{bool}\\
&\qquad \mathtt{vis}(\nu,\sigma,x) = x \in (\nu - \sigma)\\
&\qquad \mathtt{vis}(\nu,\sigma,(m,m')) = \mathtt{vis}(\nu,\sigma,m) \vee \mathtt{vis}(\nu,\sigma,m')\\
&\qquad \mathtt{vis}(\nu,\sigma,(r\ m)) = \mathtt{vis}(\nu,\sigma,m)\\
&\qquad \mathtt{vis}(\nu,\sigma,(m)) = \mathtt{vis}(\nu,\sigma,m)\\
&\qquad \mathtt{vis}(\nu,\sigma,[m]x) = \mathtt{vis}(\nu,\sigma,m) \wedge x \notin \sigma\\
&\qquad \mathtt{vis}(\nu,\sigma,\{m\}x) = \mathtt{vis}(\nu,\sigma,m) \wedge x \notin \sigma
\end{aligned}$$

Figure 9: Visibility under constraints

LET (NEW)
$$\frac{\nu \cup \{x\} \vdash c}{\nu \vdash \mathsf{let}\ x = \mathit{new\ in}\ c}$$

This rule simply introduces a new distinguishing value into the set ν. Recall that the *new* case of the let form generates a globally-unique, i.e., locally-originating, nonce.[3]

MATCH (ALIASING)
$$\frac{x \in \nu \qquad \nu \cup \{y\} \vdash c}{\nu \vdash x\ \mathsf{match}\ y\ c}$$

MATCH
$$\frac{x \notin \nu \qquad \neg\exists y.m = y \qquad \nu \vdash c}{\nu \vdash x\ \mathsf{match}\ m\ c}$$

These rules handle match statements. The interesting subtlety is that pattern matching can introduce aliases for existing distinguishing values. When such an alias is detected, i.e., when the pattern is simply an identifier, y, then that identifier must be added to the ν set. (Recall that only identifiers may appear on the left side of a match, so detecting aliasing is very simple.)

RECEIVE
$$\frac{\nu \vdash c \qquad \nu \subseteq \mathsf{ide}(m) \vee \mathtt{vis}(\nu,\mu)}{\nu \vdash \mathsf{recv}\ m\ \mu\ \Psi\ c}$$

This is the pivotal rule. It checks that either the message pattern or the constraint contains a visible distinguishing value, by employing the definition of visibility given in Fig. 9.

[3]If a protocol author has more information about what values are distinguishing, then custom rules similar to this one can be added that incorporate identifiers bound to these values into the ν set.

There are a few subtleties we do not formalize here. First, subprotocols invoked by a protocol must themselves be dispatchable. Second, these protocols must not have messages that overlap, in the sense discussed in the conclusion. Third, the analysis as presented will fail on protocols that start with message reception, because no distinguishing values have been transmitted. However, we can relax this restriction, but it requires some uninteresting technical modifications to the analysis.

5.2 The Dispatching Algorithm

Having shown an analysis that ensures every expected incoming message contains a distinguishing value, we must show how to build a dispatching server that can correctly dispatch incoming messages. The fundamental implication of our analysis is that only one session will accept any given message, because each message contains a distinguishing value. Given this, defining our server is very simple. The server simply attempts to deliver an incoming message to all existing sessions. By the above, at most one session will accept the message.

Before we formally present the dispatching algorithm, we must clarify the notion of a "session". A *session*, s, captures the entire state of a protocol run: the runtime environment, the trust management database, and the current continuation. A session may be *initialized* by a procedure p by creating the runtime environment containing bindings for p's arguments, initializing a trust management database, and using the continuation embedded in p. A session is *waiting* if its continuation is a `recv` statement, i.e., it is waiting for an incoming message. We say that a session s *evolves* into a session s' if executing the continuation of s with the given runtime environment and trust management database reduces after some number of steps to the session s'. If a session s evolves to a session s' where the continuation is a `return` statement, then the session s is said to have *returned*. A session s that cannot evolve to any session s' is said to be *stuck*. We say that we *deliver* a message M to a waiting session s when we allow s to evolve and receive message M at the `recv` statement at the top of the continuation. Notice that if we deliver a message M to a waiting session s where M does not match the message pattern or constraints in any of the branches of the `recv` statement at the top of s's continuation, then s is stuck.

The server obeys the following algorithm, with P representing the CPPL$_{\text{m+c}}$ protocol being served.

1. Initialize *sessions* to $\{\}$.
2. Wait for a message, M.
3. Replicate *sessions* to create tmp.
4. If tmp is the empty set, then go to step 7. If not, continue.
5. Remove a session, s, from tmp.
6. Deliver message M to s.
   (a) If s returns, remove s from *sessions*, and go to step 2.
   (b) If s evolves into a waiting session s', remove s from *sessions*, add s' to *sessions* and go to step 2.
   (c) If s is stuck, go to step 4.
7. Initialize the protocol P to create a new session s_{new}. Deliver message M to s_{new}.
   (a) If s_{new} returns, go to step 2.
   (b) If s_{new} evolves into a waiting session s', add s' to *sessions* and go to step 2.
   (c) If s_{new} is stuck, reject the message and go to step 2.

5.3 Correctness of Analysis and Dispatching

Our goal is not only to deploy protocols on the Web but also to show that, in the process of doing so, we have preserved any properties proven about them. We must therefore show that our dispatching algorithm is "correct" in some way. Correctness is, however, difficult to define in this context.

One definition of correctness would be that if a message is delivered to a session, then that session can run to completion. One way of stating this formally is: If a message M_0 is delivered to a session s, then if some sequence of messages $M_1, \ldots, M_n$ is subsequently delivered to s, then session s will return.

This condition is, however, not easy to prove, and it is not always possible to even check in an actual implementation. For example, consider the `commit` protocol given in Sec. 3. As we mentioned, it is not possible to check that some `key` will decrypt `payload` (indeed, cryptographic security relies on this fact). Given this, it is difficult to imagine a means of realistically implementing a checker for the above condition for a message received on line 2.

In other words, it is difficult to establish that the dispatching algorithm is "right", i.e., that it ensures *progress*. Absent this stronger form of correctness, we should at least try to demonstrate the algorithm does not do anything "wrong", i.e., that it exhibits *preservation*. In particular, it seems clearly wrong to deliver a message meant for one session to some other session.

Our work adopts this approach, but with a practical restriction. To understand this restriction, consider two sessions, A and B, of a protocol that employs the empty message, ε, as a keep-alive. Under the strand-space network model, when an ε message is received, there is no reason why it should be delivered to A rather than to B. Informally, there is no harm in sending an ε message "meant" for A to B, provided A eventually receives an ε message. Pragmatically, however, it is clear that conditions outside the pure strand space model—such as timeouts—can cause session abortion.

Our analysis therefore forces every every message to contain a distinguishing value. This is a property that the ε message, in particular, fails to exhibit. So long as messages contain distinguishing values that some session accepts but no other sessions can, we can easily demonstrate that no message will be delivered incorrectly. This final analysis is therefore implemented by our compiler. It is satisfied by a wide range of existing protocols, and also provides a useful guideline for the creation of new ones.

To prove this form of preservation, we first show that we have correctly formulated a definition of visibility and that our analysis is correct.

LEMMA 1. *If* $\texttt{vis}(\nu,\sigma,m)$ *for some message pattern* m *and set of unknown identifiers* σ*, then some identifier in* ν *is visible.*

PROOF. We proceed by structural induction over the cases of m.
Case (1) Assume that $m = x$. If $x \in \nu$, then x is visible and is in ν. If $x \notin n$ or x is unknown, then x is not visible or in ν. This our base case.
Case (2) Assume that $m = (m', m'')$. By induction.
Case (3) Assume that $m = (r\ m')$. By induction.
Case (4) Assume that $m = (m')$. By induction.
Case (5) Assume that $m = [m']x$. We may proceed by induction provided the signing key x is known.
Case (6) Assume that $m = \{m'\}x$. We may proceed by induction provided the encrypting key x is known. □

LEMMA 2. *If* $\texttt{vis}(\nu,\sigma,\mu)$ *for some constraint* μ *and set of unknown identifiers* σ*, then some identifier in* ν *is visible.*

PROOF. We proceed by structural induction over the cases of μ.

Case (1) Assume that $\mu = \top$ and $\mathtt{vis}(\nu, \sigma, \mu)$. This is clearly a contradiction of our definition.

Case (2) Assume that $\mu = \mu' + \mu''$. This OR constraint represent two branches of the program. We must ensure that all paths through the program contain distinguishing values. Thus, each subconstraint must contain some identifier of ν.

Case (3) Assume that $\mu = \exists x^*.\mu'$. By induction.

Case (4) Assume that $\mu = x \sim m; \mu'$. By induction and Lemma 1. □

THEOREM 2. *If a* CPPL$_{m+c}$ *phrase passes the analysis, then each message it receives contains a distinguishing value.*

PROOF. We proceed by induction and cases. Clearly the only interesting case is recv statements.

A recv statement is satisfied only if some distinguishing value, i.e., a member of the set ν, n, is in the set of identifiers of the message pattern, or if it is visible under the constraints, μ, of the statement. In the first case, the distinguishing value is clearly in the message. In the second case, we know by Lemma 2 that the message contains the distinguishing value. □

Now that we have shown our analysis is correct, we can prove our algorithm is correct by relying on these proofs. We do so in two steps.

LEMMA 3. *If the dispatching algorithm rejects a message M, then no session could accept message M.*

PROOF. Suppose that some session s could have accepted message M. If the message has been rejected, then it must have been rejected in step 7c. This implies that s_{new} rejected the message and that the set tmp was empty (step 4).

Suppose that tmp was initially $\{s_0, \ldots, s_n\}$. For tmp to become empty, step 5 must be executed for each session s_0 through s_n. At step 6, the algorithm must have taken branch 6c, to return to step 4 and then 5. The algorithm only takes branch 6c when the session (s_i) rejects the message.

Thus all sessions have rejected M, resulting in a contradiction with the assumption that some session s exists. □

LEMMA 4. *If a session s accepts a message M, then no other session s' could have accepted message M.*

PROOF. Assume that some other session s' exists that also accepts message M.

If s accepts M, then it is waiting on some pattern p for a message that satisfies constraints μ and M matches p and satisfies the constraints μ.

If s' accepts M, then it is waiting on some pattern p' for a message that satisfies constraints μ' and M matches p' and satisfies the constraints μ'.

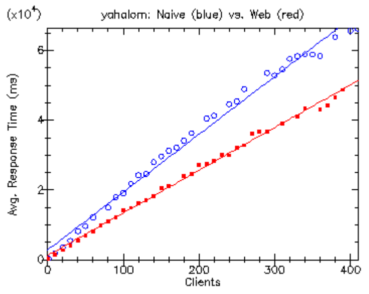
Recall that the code associated with s and s' have passed the analysis. Therefore p or μ contains some visible distinguishing value n, and p' or μ' contains some visible distinguishing value n'.

If M satisfies both patterns and constraints, then it contains both n and n'. This is a contradiction to our assumptions about distinguishing values, and in particular, our Dolev-Yao [2] assumptions about nonces, which are globally unique and non-forgeable.

Thus, s' must not exist. □

THEOREM 3. *The dispatching algorithm delivers all messages that can be delivered and does not deliver a message to session s that should be delivered to session s'.*

PROOF. By Lemmas 3 and 4. □



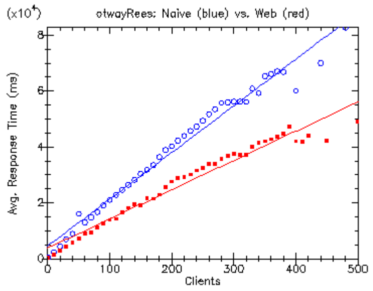


Figure 10: Typical Benchmark Results

6. IMPLEMENTATION

We have modified the existing CPPL compiler to support the analysis described above. This compiler is implemented using OCaml [7] and translates a CPPL$_m$ source file into OCaml. Each CPPL$_m$ procedure is translated into an OCaml procedure that takes a number of CPPL$_m$ values as arguments and returns a tuple of results.

Our implementation performs the analysis described in this paper; if the analysis fails, the compiler displays an error. The error message advises the protocol author about which line contains a recv statement that does not contain a distinguishing value. For example, an error message would be displayed for line 2 of the `commit` protocol example from Sec. 3. This helps a protocol author identify where hand soundness proofs must occur or where the protocol must be changed for Web deployment.

In our previous implementation, translated CPPL source files were combined with OCaml source files that set up the environment correctly and then invoked the CPPL procedures with the proper arguments. Our implementation provides a dispatching server, like a Web server, written in OCaml that runs the CPPL$_m$ procedures on incoming messages as described by our algorithm. Our implementation also records the values returned by each successful protocol run, whereas the earlier implementation prints this information when the single protocol run completes.

Performance. We have built a benchmarking suite based on a sample of protocols from the SPORE [10] repository. Fig. 10 shows results for two protocols which captures the trend across the suite. The horizontal axis shows the number of concurrent clients and the vertical axis shows the average time to session completion. The upper lines show the outcome for the previous compiler, and the lower lines show the outcome for our approach.

The reason for this gain is clear. Enabling deployment of CPPL protocol specifications on stock Web servers engenders both usability (since both deployers and clients can leverage the Web's infrastructure) and scalability (owing to the Web's statelessness). In particular, deployments can now leverage the frequent improvements being made to Web server performance to handle more simultaneous sessions.

7. RELATED WORK

In the introduction, we alluded to a strawman solution involving uniform modification of protocols. That approach corresponds to past work [8] on compiling sequential interactive programs into programs that run in Web servers. Their compiler consumes console programs that present a prompt and react to what the user types into Web applications that present a form and resume when the user submits a response. The compiler functions by computing and storing the continuation at each interaction point, and encoding the computation's closure in a combination of hidden fields, database records, and cookies. On response, this continuation is invoked with the client's response and the program continues, oblivious to the modification. A nonce records which continuation to resume at each form submission.

This solution is inappropriate for CPPL programs for several reasons. First, CPPL programs need not follow every send with a receive. Second, CPPL programs may involve more than two participants. A third, essential, reason why this solution is not appropriate is that it changes the format of the messages by adding the encoded continuations. In our environment this corresponds to changing the protocol itself, which makes existing proofs invalid.

There is another reason why that work is inappropriate here. A goal of that research is to ensure that continuations may be invoked safely multiple times, e.g., that the Web browser's Back button may be used safely. We expressly do not wish to support this because it would correspond to replay attacks and violate freshness assumptions.

One aspect of our work extends CPPL to include match statements that perform pattern matching. Other languages for programming cryptographic protocols also contain this functionality. Haack and Jeffrey [6] discuss their pattern-matching system in the context of the Spi-calculus. They also discuss the subtlety we mention in Sec. 4.1 regarding whether identifiers introduced by EXISTS forms can be used in an encrypting position in a pattern. Our work is complementary to theirs in this aspect, as our system is based on a strand-space semantics and this feature is not our sole contribution.

8. CONCLUSION

We have shown how a CPPL_m program can be analyzed and compiled into a form that allows it to be used in the context of a Web server. We have argued that the properties proved on the original protocol are preserved, and we have presented a dispatching strategy that enables multiple sessions to not interfere with one another. We have built an implementation that can be used by protocol engineers to produce efficient implementations.

The approach used in this work may be applied to other problems as well. For example, there is work that produces protocols that guarantee that participants implement session types [1, 9]. We could analyze the constraints of existing protocols and ensure that they imply that messages contain the appropriate cryptographic forms that enforce the session type property, thereby allowing this work to be applied to existing protocols.

We would like to extend our language with regular expressions types (to represent XML) and extend our analyses to handle them, then apply our work to XML Web service protocols that exist in the wild. We would also like to optimize the dispatcher design presented here to eliminate many tests when many protocol runs share similar constraints and message patterns, by associating the shape of incoming messages with existing sessions.

Finally, we would like to generalize our analysis to accommodate multiple protocols simultaneously. This would require an analysis of all participating protocols to ensure that there are no messages that match receive branches in multiple protocols, i.e., to ensure that the messages do not overlap.

Acknowledgments. This material is based upon work supported by NSF grant CNS-0627310 and under a National Science Foundation Graduate Research Fellowship.

9. REFERENCES

[1] K. Bhargavan, R. Corin, C. Fournet, and A. D. Gordon. Secure sessions for web services. In *Workshop on Secure Web Services*, pages 56–66, 2004.

[2] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.

[3] O. Goldreich. *The Foundations of Cryptography*. Cambridge University Press, Cambridge, UK, 2004.

[4] J. D. Guttman, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Programming cryptographic protocols. In *Trust in Global Computing*, pages 116–145, 2005.

[5] J. D. Guttman, F. J. Thayer, J. A. Carlson, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Trust management in strand spaces: A rely-guarantee method. In *European Symposium on Programming*, pages 325–339, 2004.

[6] C. Haack and A. Jeffrey. Pattern-Matching Spi-Calculus. In *Formal Aspects in Security and Trust*, pages 55–70, 2004.

[7] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml System*. INRIA, `http://caml.inria.fr/`, 2000. Version 3.00.

[8] J. Matthews, R. B. Findler, P. T. Graunke, S. Krishnamurthi, and M. Felleisen. Automatically restructuring programs for the Web. *Automated Software Engineering*, 11(4):337–364, 2004.

[9] K. Onda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming*, pages 122–138, 1998.

[10] Project EVA. Security protocols open repository. http://www.lsv.ens-cachan.fr/spore/, 2007.

[11] 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.