Secure Execution of Distributed Session Programs

Nuno Alves
Freelance Consultant

Raymond Hu
Imperial College London
rhu@doc.ic.ac.uk

Nobuko Yoshida
Imperial College London
yoshida@doc.ic.ac.uk

Pierre-Malo Deniélou
Imperial College London
malo@doc.ic.ac.uk

The development of the SJ Framework for session-based distributed programming is part of recent and ongoing research into integrating *session types* and practical, real-world programming languages. SJ programs featuring session types (protocols) are statically checked by the SJ compiler to verify the key property of *communication safety*, meaning that parties engaged in a session only communicate messages, including higher-order communications via *session delegation*, that are compatible with the message types expected by the recipient.

This paper presents current work on security aspects of the SJ Framework. Firstly, we discuss our implementation experience from improving the SJ Runtime platform with security measures to protect and augment communication safety at runtime. We implement a transport component for secure session execution that uses a modified TLS connection with authentication based on the Secure Remote Password (SRP) protocol. The key technical point is the delicate treatment of *secure session delegation* to counter a previous vulnerability. We find that the modular design of the SJ Runtime, based on the notion of an Abstract Transport for session communication, supports rapid extension to utilise additional transports whilst separating this concern from the application-level session programming task. In the second part of this abstract, we formally prove the target security properties by modelling the extended SJ delegation protocols in the π-calculus.

1 Session Programming in SJ

It has become increasingly important to understand and specify the behaviour of applications across many domains as sequences of communications and interaction between concurrently executing components, as opposed to independent black boxes that simply return a final output from a given input. Unfortunately, the most common technologies and programming techniques for communication-based programming in use today do not provide the level of support and safety guarantees enjoyed for traditional single-threaded, "localised" programming. For example, low-level network socket APIs provide only the minimal mechanisms for exchanging untyped data in an unstructured manner, and higher-level RPC/RMI libraries are typically coupled to the synchronous call-return pattern and lack the facility to encapsulate a series of such exchanges as one complete unit of interaction.

SJ [10, 12] is an extension of Java that uses *session types* [8] to address the above issues. Programmers use session types, which can be thought of as communication protocols, to specify the abstract communication behaviour of a program. Concrete communication behaviour is implemented using special session primitives; in particular, SJ supports higher-order session types, implemented by *session delegation* actions, that express the migration of ongoing sessions to new parties. The SJ compiler performs static session type checking, guaranteeing that session implementations conform to their declared types, i.e. a SJ session between compatible peers will never reduce to a communication error other than premature termination due to e.g. network failure. Other works have presented SJ implementations of communication-based applications from widely different domains, such as Internet Web services [10] and parallel algorithms for cluster computing [3], demonstrating competitive performance in both cases.

K. Honda and A. Mycroft (Eds.): Programming Language Approaches to Concurrency and communication-cEntric Software 2010 (PLACES'10)
EPTCS 69, 2011, pp. 1–11, doi:10.4204/EPTCS.69.1

© Alves, Hu, Yoshida and Deniélou
This work is licensed under the Creative Commons Attribution License.

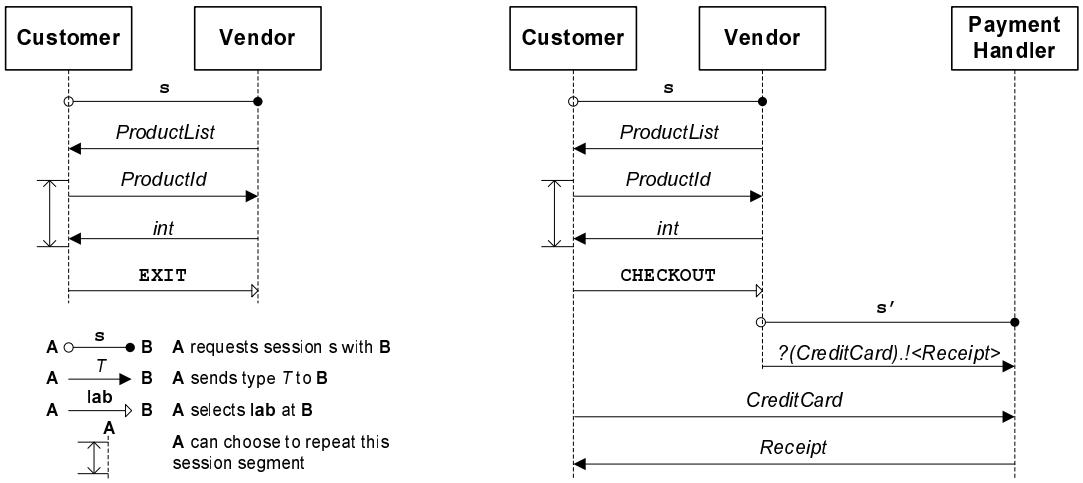


Figure 1: The interaction between a Customer, the Vendor and Payment Handler service in an online purchase session.

Securing session execution. To enforce the statically verified communication safety property at runtime, the SJ Runtime (SJR) validates peer compatibility at session initiation and provides a session monitoring service that dynamically tracks session progress against the expected type. Nevertheless, the development of the SJ Framework as a research project, until now, had yet to focus on *session security* as a primary objective. Indeed, the security of session delegation has been a frequently posed question. In this paper, we identify a potential vulnerability in the existing SJ session delegation protocols; we then implement and formalise SJR extensions for secure session execution, solving the above problem and strengthening runtime communication safety. § 2 starts with an example SJ application (a modified version of the main example in [10]) that illustrates both the usual security concerns and issues specific to session delegation. § 3 then briefly discusses the design and implementation of a new *Transport Module* that enables secure session execution over a modified TLS connection with authentication based on the Secure Remote Password (SRP) protocol. In § 4, we formalise the extended delegation protocols encapsulated by the new Transport Module, and prove the target security properties. Finally, § 5 concludes by describing our ongoing and future work.

2 An Example SJ Application

To illustrate the session security issues, in particular regarding session delegation, we examine the interaction in a modified version of the main Web services example from [10], an implementation of Use Case C-UC-001 from the W3C Web Services Choreography Requirements [5].

The basic scenario is an online purchase session involving three parties, a Customer (**C**), the Vendor (**V**) and a Payment Handler service (**H**). The interaction between these parties constituting one session is depicted in Figure 1, where each side represents one of the two ways to complete the session. Both start by **C** connecting to **V**, and **V** sending a list of the products for sale. In the next part, **C** adds a product to the basket and **V** returns the updated total cost of the basket; this segment can be repeated an arbitrary number of times by **C**. After this, **C** has two choices. On the left-hand side, **C** cancels the purchase and ends the session by selecting the `EXIT` branch. On the right-hand side, **C** proceeds by selecting the `CHECKOUT` branch. At this point, **V** enters a session with the third party, **H**. The single action between

```
protocol customerToVendor {
  cbegin. // Client session request.
  ?(ProductList). // Get product list.
  ![ // Can repeat this segment.
    !<ProductId>. // Add product to basket.
    ?(int) // Get updated basket total.
  ]*.
  !{ // Two branch options.
    CHECKOUT: // Proceed to checkout.
      !<CreditCard>.
      ?(Receipt),
    EXIT: // Cancel purchase.
  }
}
```

```
protocol vendorToCustomer {
  sbegin.
  !<ProductList>.
  ?[
    ?(ProductId).
    !<int>
  ]*.
  ?{
    CHECKOUT:
      ?(CreditCard).
      !<Receipt>,
    EXIT:
  }
}
```

Figure 2: Session types (declared as SJ protocols) for the interaction depicted in Figure 1.

V and **H** is an example of *session delegation*: the message type itself is a session type that specifies the remaining session actions to be completed by **H** on behalf of **V**. After **V** delegates its side of the session to **H**, the interaction proceeds between **C** and **H**: **C** sends her credit card details, and **H** issues a receipt.

Session types for this application. Figure 2 lists the session types, declared as SJ protocols, for the interaction between **C** and **V** from the perspective of each party. The `customerToVendor` protocol starts with the `cbegin` element to denote the client side of the session. Receiving and sending messages have the syntax, e.g. `?(ProductList)` and `!<ProductId>` respectively. The repeated segment of the session is specified as a session iteration type, `![...]`. Here, the `!` signifies that **C** controls the termination condition of the loop. Finally, the two session branches are collected within the `!{...}` constructor and labelled, e.g. `CHECKOUT`; the `!` again signifies that **C** makes the branch decision. The `vendorToCustomer` protocol is the *dual* session type that describes the reciprocal behaviour, in this case given by simply inverting the output (`!`) and input (`?`) symbols. Note that the `?[...]` (resp. `?{...}`) type specifies that **V** should follow the iteration (resp. branch) decision made by **C**.

Recall that **V** will not actually perform the final `?(CreditCard).!<Receipt>` exchange itself, but will delegate these actions to **H**. These actions must still be specified in `vendorToCustomer` to achieve duality between the behaviours of **C** and **H** (in SJ, session initiation between non-dual parties raises an exception). However, the delegation action from **V** to **H**, specified by the protocols

```
protocol vendorToHandler {
  cbegin.
  !<?(CreditCard).!<Receipt>>
}
```

```
protocol handlerToVendor {
  sbegin.
  ?(?(CreditCard).!<Receipt>)
}
```

ensures that **V** indeed fulfils the `vendorToCustomer` protocol contract. This Use Case demonstrates how session types provide a type-safe discipline for communications programming, including higher-order communications that evolve the shape of the session network.

Due to space limitations, we omit the application-level implementations of these protocols in order to focus on the runtime security of delegation. The full source code for this example can be found in the `tests/src` directory (see the `places.purchase` package) of the SJ Google Code repository (linked from [12]); see [10] for further explanation of the SJ session primitives. The types and implementation of this application can be readily extended (e.g. to pass additional information from **V** to **H** before the delegation, and for **H** to return an acknowledgment after completing the session with **C**) by adding the

required session interactions.

A vulnerability in session delegation. Needless to say, conducting the above session over an insecure transport connection jeopardises message confidentiality and integrity, an especial concern for highly sensitive messages like `CreditCard`. For this purpose, the SJR includes SSL and HTTPS variants of the basic TCP and HTTP Transport Modules, implemented using the standard Java APIs for these features. However, the current version of SJ does not have dedicated support for mutual peer authentication outside of TLS certificate-based authentication (which is used to accomplish only unilateral authentication in most typical TLS authentication scenarios).

In addition to the general attacks mentioned above, we identify a specific vulnerability in the SJ session delegation protocols. The first work on implementing session delegation [10] presented three alternative SJR protocols with varying tradeoffs for coordinating the three (or four) parties involved in the delegation of an ongoing session. Two of these protocols, the Resending Protocol and Bounded Forwarding Protocol, are termed as *reconnection-based*. As the name suggests, these protocols involve closing the original transport-level connection underlying the application-level session being delegated, and establishing a new connection to reflect the session migration. A detailed recap of the delegation protocols is beyond the scope of this paper, but Figure 3 lists an instance of the Resending Protocol for the above application: V (resp. H, C) denotes the SJR supporting **V** (resp. **H**, **C**), $DS_C^V(H)$ denotes the Delegation Signal for the delegation from **V** to **C** with passive party **H**, $ST_{\mathbf{C}}^{\mathbf{V}}$ the remaining type of the session between **V** and **C** from the former's side, ACK_{CV} the Delegation Acknowledgement from C to V, and $LM(ST_{\mathbf{V}}^{\mathbf{C}} - ST_{\mathbf{C}}^{\mathbf{V}})$ the "lost messages" corresponding to the difference between the two session types. The reconnection itself is performed by the C SJR over steps 6 and 7. The crucial point is that the lack of mutual authentication between all *three* peers, and hence the inability to confirm that the party accepted by H (step 2) is the same as the original C, allows an attacker to infiltrate the session, masquerading as C, at this weak point. This is called Reconnection Vulnerability later in this paper and also applies to the Bounded Forwarding delegation protocol [10] for the same reasons. This vulnerability occurs because a normal session is designed to be binary (only two peers) and upon delegation, a third (or fourth) party is involved.

For secure session delegation, we extend the reconnection-based protocols with additional authentication message exchanges: Figure 4 lists the new secure version of the original protocol instance from Figure 3. Our extended protocol is different from the original in step 1, the creation of a fresh credential by V, and steps 2 and 5, which send the credential to H and C respectively. H then stores the credential and waits for a connection on p_H. The key action in the extended protocol lies on step 9, where C sends the credential to H after connecting: if H can validate that the credentials match, the connection is successfully established, otherwise the delegation has failed and the session is aborted.

The above protocol listings correspond to Case 1 of the Resending Protocol [10]. A complete analysis of all four Delegation Cases for our Secure Resending and Forwarding Protocols is specified in [2].

3 Implementation of a Secure Transport Module

We first summarise the SJ Framework and the structure of the SJ Runtime (SJR). We then briefly explain the design and implementation of a secure Transport Module plugin for the SJR that solves the security issues described in § 2.

The SJ Framework comprises the compiler and the SJ Runtime (SJR). The compiler transforms SJ programs into a *transport-independent* form in standard Java, translating the application-level SJ session

1.	$V \to H$:	`START_DELEGATION`			
2.	H:	Open server socket on free port p_H; accept connection on p_H			
3.	$H \to V$:	p_H			
4.	$V \to C$:	$DS^V_C(H) = \langle ST^{\mathbf{v}}_{\mathbf{C}}, IP_H, p_H \rangle$			
5.	$C \to V$:	ACK_{CV}			
6.	C:	Close s	6'.	V:	Close s
7.	C:	Connect to $IP_H : p_H$			
8.	$C \to H$:	$LM(ST^{\mathbf{C}}_{\mathbf{v}} - ST^{\mathbf{v}}_{\mathbf{C}})$			

Figure 3: Operation of the original Resending Protocol between the Vendor, Handler and Customer.

1.	V:	Credential creation			
2.	$V \to H$:	`START_DELEGATION::CRED`			
3.	H:	Open server socket on free port p_H; accept connection on p_H			
4.	$H \to V$:	p_H			
5.	$V \to C$:	$DS^V_C(H) = \langle ST^{\mathbf{v}}_{\mathbf{C}}, IP_H, p_H, \texttt{CRED} \rangle$			
6.	$C \to V$:	ACK_{CV}			
7.	C:	Close s	7'.	V:	Close s
8.	C:	Connect to $IP_H : p_H$			
9.	$C \to H$:	`CRED`	9'.	H:	`CRED` checking
9a.	-pass:	Connection successful	9a'.	-pass:	Connection established
9b.	-fail:	Credential rejected, close s	9b'.	-fail:	Authentication error, close p_H
10.	$C \to H$:	$LM(ST^{\mathbf{C}}_{\mathbf{v}} - ST^{\mathbf{v}}_{\mathbf{C}})$			

Figure 4: Operation of the *Secure* Resending Protocol between the Vendor, Handler and Customer.

primitives in terms of Java control flow and calls to *Interaction Services* hosted by the SJR, also implemented in Java. Thus, SJ programs can be executed on any standard JVM, where the purpose of the SJR is to perform the requested Interaction Services as actions on an underlying transport connection. The key element in the SJR is the *Abstract Transport Interface*, which represents an idealised asynchronous, reliable and order-preserving message-oriented transport for session communication. Interaction Service components are implemented as actions on the Abstract Transport, which are in turn implemented by *Transport Modules* that encapsulate the communication mechanisms of specific "concrete" transports, such as TCP and shared memory. The Abstract Transport thus serves to decouple the realisation of session interaction semantics in the SJR from the provision of the underlying communication mechanisms.

Design. Our primary goal was to provide a SJR Transport Module that incorporates a means of peer authentication in addition to message confidentiality and integrity, and thereby solve the security concern exposed by the reconnection-based delegation protocols. After evaluating the available options (see [2] for the omitted details), our chosen design adapts TLS to use the Secure Remote Password (SRP) [13] authentication method [7]. Although standard usage of TLS provides confidentiality and integrity, standard authentication relies on certificates produced by external authorities. We choose to combine TLS and SRP in order to provide session security without requiring trusted third-parties, whilst still being able to support the certificate-based authentication in our implementation. Retaining session independence from external third-parties is also in line with the established session theory (which our formalism in § 4 follows), where scope restriction of each session to just the two parties involved is an important invariant in proving communication safety [11].

Implementation and the extended delegation protocols. TLS consists of three basic phases: negotiation of the algorithms supported, key exchange and authentication, and symmetric cipher encryption/message authentication. Unfortunately, none of the publically available Java implementations of SSL/TLS fulfilled all of our requirements: pure Java implementations either lacked support for SRP integration or were otherwise incomplete, and using JNI to interface e.g. C libraries would break SJ portability [2]. As a consequence, we decided to implement SRP as an initial mutual authentication phase before the standard TLS phases. At the client side, we use a modified SJ transport negotiation protocol to activate the SRP; at the server side, we override the behaviour of the `accept` method of the standard Java `SSLServerSocket` API.

With the SRP authentication in place, the resending-based delegation protocols are strengthened by using the (successfully authenticated) session-sender (i.e. the party delegating the session — in our example, **V**) to generate and distribute fresh, secure credentials on behalf of the party performing the reconnection (**C**), as illustrated in § 2. By presenting these credentials to the session-receiver (**H**) over the *new* TLS connection (ruling out replay attacks), the latter can confirm the authenticity of the connecting party, which was a gap in the authentication of the final two peers after the session delegation. The detailed protocol specifications for the extended delegation protocols can be found in [2].

4 A Process Model for Secure Sessions

To prove the design of the new Transport Module satisfies the intended security properties, we model the extended delegation protocols in the π-calculus and formalise each property. The three original properties for correct session delegation are Linearity, Liveness and Session Consistency, which were proved by case analysis for each of the four valid delegation scenarios (referred to as *Delegation Cases 1–4*) [10]. To these, we add a new *Session Security* property to rule out attacks on the delegation protocols: there are three aspects to this property, corresponding to the key security mechanisms that the delegation protocols depend on. In the following, we focus on the three aspects of the latter; the other properties (plus the cases omitted from below) are fully documented in [2]. See a formalisation of the Resending Protocol in the appendix. As a convention, we use *A*, *B* and *C* to respectively denote the passive party, session-sender and session-receiver in the three-party delegation scenarios (Delegation Cases 1–3 in [2]). (In the previous example of § 2, the passive party, session-sender and session-receiver are respectively the Customer, the Vendor and the Payment Handler; we now use a general notation for the delegation roles rather than the parties from that specific example.) Our approach has been influenced by [8, 11]. Below, we use "∀protocols" to mean all of the protocols formalised in [2].

Freshness *(The credentials are fresh for only the current session.)*

$$\forall \text{protocols}.\exists P \equiv \mathsf{make}(\mathit{Cred}); Q \text{ s.t. } P \text{ is a session-sender}$$

where $\mathsf{make}(\mathit{Cred}); Q$ is a binder for *Cred* in Q whose reduction instantiates *Cred* to a fresh value in Q. Freshness means that every session-sender P (defined as $P \equiv s!\langle \mathit{Cred}\rangle; P_1; s'!\langle ..., \mathit{Cred}, ...\rangle; P_2)$) in the protocols creates a new credential for that specific session. Freshness in the Resending Protocol is verified for Delegation Case 1 as follows (we omit the parallel composed processes):

$$B = \mathsf{make}(\mathit{CredA}); s'_{\mathrm{BC}}!\langle \mathit{CredA}\rangle; B_1 \longrightarrow s'_{\mathrm{BC}}!\langle \mathit{CredA}\rangle; B_1 \longrightarrow$$
$$B_1 \longrightarrow^* \overline{s_{\mathrm{AB}}}!\langle S', IP_{\mathrm{C}}, x_{p_C}, \mathit{CredA}\rangle; B_2 \longrightarrow B_2$$

B creates a new credential specific to the current session before the actual delegation action, which it sends to both C and A (for A, it is sent together with other important session information). So, we can conclude that Freshness holds in this case since the credential is fresh by the semantics of make.

Ternary Authentication for Delegation *(Session delegation only succeeds if the credentials of the passive party matches that of the session-receiver. Otherwise, a delegation error has occurred and the session is terminated.)*

$$\begin{array}{ll} & \forall \text{protocols}.\exists P_C \equiv s?(x_{Cred});\, P;\, port\,(x\colon \overline{S}).x?(y_{Cred});\\ & \qquad\qquad \texttt{if}\ x_{Cred} == y_{Cred}\ \texttt{then}\ x \triangleleft \mathbf{Success};\, Q\ \texttt{else}\ x \triangleleft \mathbf{Fail};\, \mathsf{close}(x)\\ \text{and} & \exists P_A \equiv \overline{port}\,(x\colon S).x!\langle y_{Cred}\rangle;\, x \triangleright \{\mathbf{Success} : Q, \mathbf{Fail} : \mathsf{close}(x)\}\\ \text{s.t.} & P_C \text{ is the session-receiver and } P_A \text{ is the passive party} \end{array}$$

where P is defined to be a session-receiver if $P \equiv s?(Cred);\, P_1;\, s'?(Cred);\, P_2$, and a passive party if $P \equiv s'?(\ldots,Cred,\ldots);\, P_1;\, s!\langle Cred\rangle;\, P_2$. This property states that in all protocols, the session-receiver must start by receiving a set of credentials from the session-sender. The session-receiver then accepts a new connection on the open *port*, and receives another set of credentials. If the credentials match, then the session is continued by selecting **Success** branch; otherwise the new connection is closed, aborting the session. The passive party starts by requesting a connection on *port* followed by sending the credential it received from the session-sender. If the credential is accepted it similarly follows the **Success** branch; otherwise it goes to the **Fail** branch, which closes the new connection from the other end.

Below we verify Credential Checking for the Resending Protocol Delegation Case 2 in [2]. We omit B's specification and interactions, and focus on the interactions between A and C.

$$\begin{array}{l} \qquad\qquad\qquad\qquad\qquad\qquad \text{ohh}\\ \text{Let}\ \ P \ = \ x \triangleright \{\mathbf{Success} : x!\langle LM(S-S')\rangle, \mathbf{Fail} : \mathsf{close}(x)\}\\ \text{and}\ \ Q \ = \ \texttt{if}\ x_{CredA} == y_{CredA}\ \texttt{then}\ x \triangleleft \mathbf{Success};\, x?(x_{\mathrm{LM}})\ \texttt{else}\ x \triangleleft \mathbf{Fail};\, \mathsf{close}(x);\, \mathsf{close}(p_C) \end{array}$$

$$\begin{array}{rcl} A \mid B \mid \overline{s'_{BC}}?(x_{CredA});\, C_1 & \longrightarrow^* & \overline{p_C}\,(x\colon S).x!\langle z_{CredA}\rangle;\, P \mid B' \mid p_C\,(x\colon \overline{S}).x?(y_{CredA});\, Q\\ & \longrightarrow^* & x!\langle z_{CredA}\rangle;\, P \mid B' \mid x?(y_{CredA});\, Q\\ & \longrightarrow^* & (P \mid B' \mid Q) \longrightarrow^* \mathbf{0} \end{array}$$

The delegation protocol starts with $C = \overline{s'_{BC}}?(x_{CredA});\, C_1$ receiving a credential from B. Then A connects to C via p_C and establishes a new session (bound to x). To finalise the delegation, A sends its credentials to C who compares this value with the one received from B: if they match, the delegation is successful; otherwise the session is closed along with the port p_C.

Reconnection Vulnerability *(Protection against attacks from the Network, ensuring message authentication, confidentiality and integrity.)*

The formalism assures protection of sessions from Network attacks due to the firm restriction that each session channel can only be accessed by the processes involved in that session, i.e. each channel is private to those parties and cannot be interfered by others. To fully prove this property, an additional layer is needed to model the lower level protocols underneath the session calculus; one suitable approach would be the Applied π-Calculus [1], which we leave as future work. As a simple demonstration, however, we can model a Network Attacker E that attempts to interfere with the genuine session parties.

$$\begin{array}{rl} \mathrm{E} = & s_{ABC}?(S', x_{IP_C}, y_{p_C}, z_{CredA});\, s_{ABC}!\langle ACK\rangle;\, \overline{y_{P_C}}\,(z\colon S).z!\langle z_{CredA}\rangle;\\ & \qquad z \triangleright \{\mathbf{Success} : z!\langle LM(S-S')\rangle, \mathbf{Fail} : \mathbf{0}\} \end{array}$$

If E is able to intercept the delegation information, then she will be able to masquerade as A without C knowing. Note that E does not close the session after the *ACK* as the attack may benefit from remaining connected to all parties. However, since there is no interaction with any external s_{ABC} in any of the delegation protocols, we can infer that E will remain blocked forever. This is in line with the design choice to keep SJ close to the original session types theory by restricting the scope of all sessions from foreign parties.

5 Conclusion

The aim of this work was to examine and improve session security in SJ. In addition to standard security concerns, we identified a vulnerability in the resending-based Resending and Bounded Forwarding protocols by modelling and analysing the delegation protocols in the π-calculus. To overcome this problem, we implemented a new Transport Module for secure session execution that combines TSL with SRP authentication. We found that the structure of the SJ Framework cleanly decouples application-level logic related to implementing specific sessions from the provision of general, lower-level communication mechanisms. Hence, we were able to readily extend the SJR so that all existing SJ programs can immediately utilise this new Transport Module without any modifications to the application source code or other SJR components. To verify the correctness of our approach, we incorporated our new extensions into the formal model to formalise and prove the intended security properties.

Future Work. We are currently looking at modifying a TLS implementation to include the SRP protocol in the cipher suite list. This way, the additional key exchange could be avoided and SRP would be included in the key exchange phase of the TLS protocol. We are also investigating several other ways of securing the delegation protocols instead of using the TLS-SRP. Timestamping and revocation of credentials can also be explored.

Another direction for our work is to handle the securisation of the extension of SJ to multiparty session types [9]. Previous work has been done [6, 4] that secures multiparty session execution in F# despite compromised participants. We plan to adapt it to the SJ framework and extend it to support delegation.

We finally want to model the delegation protocols at a lower level in order to prove the absence of the Reconnection Vulnerability (protection against network attacks, ensuring message integrity, authentication and confidentiality). This is not possible by using the standard π-calculus since it already assumes sessions to be isolated from each others. We plan to use the applied π-calculus [1], in which we can model primitives for exchanging keys during the cryptographic protocol and detail all other steps of the protocols.

References

[1] Martín Abadi & Cédric Fournet (2001): *Mobile values, new names, and secure communication*. In: *POPL*, pp. 104–115, doi:10.1145/360204.360213.

[2] Nuno Alves (2009): *A Secure Session-Based Distributed Programming Language*. *Imperial College London - MSc Thesis* http://www.doc.ic.ac.uk/teaching/distinguished-projects/2010/n.alves.pdf.

[3] Andi Bejleri, Raymond Hu & Nobuko Yoshida: *Session-based Programming for Parallel Algorithms*. *Proceedings of the ETAPS workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software PLACES'09* doi:10.4204/EPTCS.17.2.

[4] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet & James J. Leifer (2009): *Cryptographic Protocol Synthesis and Verification for Multiparty Sessions*. In: *CSF*, pp. 124–140, doi:10.1109/CSF.2009.26.

[5] *Web Services Choreography Requirements*. http://www.w3.org/TR/ws-chor-reqs/.

[6] Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan & James J. Leifer (2007): *Secure Implementations for Typed Session Abstractions*. In: *CSF*, pp. 170–186, doi:10.1109/CSF.2007.29.

[7] D. Taylor, T. Wu, N. Mavrogiannopoulos, and T. Perrin (2007): *Using the Secure Remote Password (SRP) Protocol for TLS Authentication*. *RFC 5054* Available at http://tools.ietf.org/html/rfc5054.

[8] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *ESOP '98: Proceedings of the 7th European Symposium on Programming*, Springer-Verlag, doi:10.1016/j.entcs.2007.02.056.

[9] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In: *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, New York, NY, USA, pp. 273–284, doi:10.1145/1328438.1328472.

[10] Raymond Hu, Nobuko Yoshida & Kohei Honda (2008): *Session-Based Distributed Programming in Java*. In: *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pp. 516–541, doi:10.1007/978-3-540-70592-5_22.

[11] Dimitris Mostrous & Nobuko Yoshida (2009): *Session-Based Communication Optimisation for Higher-Order Mobile Processes*. In: *TLCA '09*, Springer, pp. 203–218, doi:10.1007/978-3-642-02273-9_16.

[12] *SJ homepage*. http://www.doc.ic.ac.uk/~rhu/sessionj.html.

[13] Thomas Wu (1998): *The secure remote password protocol*. In: *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pp. 97–111. Available at http://www.isoc.org/isoc/conferences/ndss/98/wu.pdf.

A Appendix

A.1 Resending Protocol - Case 1

This is the most used case in which there is a simple delegation between three parties and the passive party is waiting for a receiving instruction.

A - $[[P \mid s_{\mathrm{AB}} : \langle S \rangle \vec{h}]]$

$$\begin{array}{l} \mathrm{A} = s_{\mathrm{AB}}?(S', x_{\mathrm{IPc}}, y_{\mathrm{pc}}, z_{\mathrm{CredA}}); s_{\mathrm{AB}}!\langle ACK \rangle; \mathsf{close}(s_{\mathrm{AB}}); \overline{y_{\mathrm{pc}}}\,(x\colon S).x!\langle z_{\mathrm{CredA}} \rangle; \\ x \triangleright \{\mathbf{Success} : x!\langle LM(S - S') \rangle, \mathbf{Fail} : \mathsf{close}(x)\} \end{array}$$

A.1.1 Typing

$\Gamma \vdash \mathbf{0} \triangleright s_{\mathrm{AB}} : \mathtt{end} \cdot x : \mathtt{end}$

$\Gamma \vdash x \triangleright \{\mathbf{Success} : x!\langle LM(S - S') \rangle, \mathbf{Fail} : \mathsf{close}(x)\}$
$\quad \triangleright s_{\mathrm{AB}} : \mathtt{end} \cdot x : \&\{\mathbf{Success} : !\langle String \rangle, \mathbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash x!\langle z_{\mathrm{CredA}} \rangle; x \triangleright \{\mathbf{Success} : x!\langle LM(S - S') \rangle, \mathbf{Fail} : \mathsf{close}(x)\}$
$\quad \triangleright s_{\mathrm{AB}} : \mathtt{end} \cdot x : !\langle CredT \rangle; \&\{\mathbf{Success} : !\langle String \rangle, \mathbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash \overline{y_{\mathrm{pc}}}\,(x\colon S).x!\langle z_{\mathrm{CredA}} \rangle; x \triangleright \{\mathbf{Success} : x!\langle LM(S - S') \rangle, \mathbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\mathrm{AB}} : \mathtt{end}$

$\Gamma \vdash s_{\mathrm{AB}}!\langle ACK \rangle; \mathsf{close}(s_{\mathrm{AB}}); \overline{y_{\mathrm{pc}}}\,(x\colon S).x!\langle z_{\mathrm{CredA}} \rangle;$
$\quad x \triangleright \{\mathbf{Success} : x!\langle LM(S - S') \rangle, \mathbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\mathrm{AB}} : !\langle ACK \rangle; \mathtt{end}$

$\Gamma \vdash s_{\mathrm{AB}}?(S', x_{\mathrm{IPc}}, y_{\mathrm{pc}}, z_{\mathrm{CredA}}); s_{\mathrm{AB}}!\langle ACK \rangle; \mathsf{close}(s_{\mathrm{AB}}); \overline{y_{\mathrm{pc}}}\,(x\colon S).x!\langle z_{\mathrm{CredA}} \rangle;$
$\quad x \triangleright \{\mathbf{Success} : x!\langle LM(S - S') \rangle, \mathbf{Fail} : \mathsf{close}(x)\} \triangleright s_{\mathrm{AB}} : ?(S, nat, nat, CredT); !\langle ACK \rangle; \mathtt{end}$

This process starts by receiving the information regarding the session receiver, along with the credential, followed by sending an ACK confirming the reception. Then, it closes the session it has with B, followed

a connection to the receiver and sending the credential. If the connection is accepted, it sends information about the lost messages; otherwise the new connection is closed.

B - $[[Q \mid \overline{s_{\mathrm{AB}}} : \langle S' \rangle \vec{h'} \mid s'_{\mathrm{BC}} : \langle S'' \rangle \vec{h''}]]$

$$\mathrm{B} = \mathsf{make}(\mathrm{CredA});\ s'_{\mathrm{BC}}!\langle CredA \rangle;\ b\,(x_{\mathrm{pc}} : S).\overline{s_{\mathrm{AB}}}!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle;\ \overline{s_{\mathrm{AB}}}?(x_{\mathrm{ACK}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}})$$

A.1.2 Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s_{\mathrm{AB}}} : \mathtt{end} \cdot s'_{\mathrm{BC}} : \mathtt{end} \cdot x_{\mathrm{pc}} : \mathtt{end}$

$\Gamma \vdash \overline{s_{\mathrm{AB}}}?(x_{\mathrm{ACK}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}}) \triangleright \overline{s_{\mathrm{AB}}} :?(ACK);\mathtt{end} \cdot s'_{\mathrm{BC}} : \mathtt{end} \cdot x_{\mathrm{pc}} : \mathtt{end}$

$\Gamma \vdash \overline{s_{\mathrm{AB}}}!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle;\ \overline{s_{\mathrm{AB}}}?(x_{\mathrm{ACK}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}})$
$\quad \triangleright \overline{s_{\mathrm{AB}}} :!\langle S, nat, nat, CredT \rangle;?(ACK);\mathtt{end} \cdot s'_{\mathrm{BC}} : \mathtt{end} \cdot x_{\mathrm{pc}} : \mathtt{end}$

$\Gamma \vdash b\,(x_{\mathrm{pc}} : S).\overline{s_{\mathrm{AB}}}!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle;\ \overline{s_{\mathrm{AB}}}?(x_{\mathrm{ACK}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}})$
$\quad \triangleright \overline{s_{\mathrm{AB}}} :!\langle S, nat, nat, CredT \rangle;?(ACK);\mathtt{end} \cdot s'_{\mathrm{BC}} : \mathtt{end}$

$\Gamma \vdash s'_{\mathrm{BC}}!\langle CredA \rangle;\ b\,(x_{\mathrm{pc}} : S).\overline{s_{\mathrm{AB}}}!\langle S', IPc, x_{\mathrm{pc}}, CredA \rangle;\ \overline{s_{\mathrm{AB}}}?(x_{\mathrm{ACK}});\ \mathsf{close}(\overline{s_{\mathrm{AB}}})$
$\quad \triangleright \overline{s_{\mathrm{AB}}} :!\langle S, nat, nat, CredT \rangle;?(ACK);\mathtt{end} \cdot s'_{\mathrm{BC}} :!\langle CredT \rangle;\mathtt{end}$

B starts by creating a fresh credential for that delegation, which is sent to C and later to A along with other important information on how to connect to C. In the end it receives an ACK from A indicating that everything went well, so it can close the session it has with A.

C - $[[R \mid \overline{s'_{\mathrm{BC}}} : \langle S^3 \rangle \vec{h^3}]]$

$$\mathrm{C} = \overline{s'_{\mathrm{BC}}}?(x_{\mathrm{CredA}});\ (\nu\ pc)\ (\overline{b}\,(pc : S).pc\,(x : S).x?(y_{\mathrm{CredA}});$$
$$\mathtt{if}\ x_{\mathrm{CredA}} == y_{\mathrm{CredA}}\ \mathtt{then}\ x \triangleleft \mathbf{Success};\ x?(x_{\mathrm{LM}})\ \mathtt{else}\ x \triangleleft \mathbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc))$$

A.1.3 Typing

$\Gamma \vdash \mathbf{0} \triangleright \overline{s'_{\mathrm{BC}}} : \mathtt{end} \cdot pc : \mathtt{end} \cdot x : \mathtt{end}$

$\Gamma \vdash x?(x_{\mathrm{LM}}) \triangleright \overline{s'_{\mathrm{BC}}} : \mathtt{end} \cdot pc : \mathtt{end} \cdot x :?(String);\mathtt{end}$

$\Gamma \vdash x \triangleleft \mathbf{Success};\ x?(x_{\mathrm{LM}}) \triangleright \overline{s'_{\mathrm{BC}}} : \mathtt{end} \cdot pc : \mathtt{end} \cdot x : \oplus\{\mathbf{Success} :?(String);\mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

Let $Q = \mathtt{if}\ x_{\mathrm{CredA}} == y_{\mathrm{CredA}}\ \mathtt{then}\ x \triangleleft \mathbf{Success};\ x?(x_{\mathrm{LM}})\ \mathtt{else}\ x \triangleleft \mathbf{Fail};\ \mathsf{close}(x);\ \mathsf{close}(pc)$

$\Gamma \vdash Q \triangleright \overline{s'_{\mathrm{BC}}} : \mathtt{end} \cdot pc : \mathtt{end} \cdot x : \oplus\{\mathbf{Success} :?(String);\mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash x?(y_{\mathrm{CredA}});\ Q \triangleright \overline{s'_{\mathrm{BC}}} : \mathtt{end} \cdot pc : \mathtt{end} \cdot x :?(CredT);\oplus\{\mathbf{Success} :?(String);\mathtt{end}, \mathbf{Fail} : \mathtt{end}\}$

$\Gamma \vdash pc\,(x : S).x?(y_{\mathrm{CredA}});\ Q \triangleright \overline{s'_{\mathrm{BC}}} : \mathtt{end} \cdot pc : \mathtt{end}$

$\Gamma \vdash \overline{b}\,(pc : S).pc\,(x : S).x?(y_{\mathrm{CredA}});\ Q \triangleright \overline{s'_{\mathrm{BC}}} : \mathtt{end}$

$\Gamma \vdash \overline{s'_{\mathrm{BC}}}?(x_{\mathrm{CredA}});\ (\nu\ pc)\ (\overline{b}\,(pc : S).pc\,(x : S).x?(y_{\mathrm{CredA}});\ Q) \triangleright \overline{s'_{\mathrm{BC}}} :?(CredT);\mathtt{end}$

The session receiver starts by receiving the credential from the session sender. Then, it creates a new port *pc* to accept a new connection from the passive party. It then sends that port to B on the shared channel *b*, followed by the waiting of the connection from A. This process terminates by receiving the credential from A and checking if this credential matches with the one received from B: if it does, it accepts the information regarding the lost messages; otherwise the new session is closed, together with the opened port *pc*.