A Secure Middleware Architecture for Web Services

Lenin Singaravelu, Jinpeng Wei, Calton Pu
College of Computing, Georgia Institute of Technology
{lenin, weijp, calton}@cc.gatech.edu

Abstract. Current web service platforms (WSPs) often perform all web services-related processing, including security-sensitive information handling, in the same protection domain. Consequently, the entire WSP may have access to security-sensitive information such as credit card numbers, forcing us to trust a large and complex piece of software. To address this problem, we propose ISO-WSP, a new middleware architecture that decomposes current WSPs into two parts executing in separate protection domains: (1) a small trusted T-WSP to handle security-sensitive data, and (2) a large, legacy untrusted U-WSP that provides the normal WSP functionality, but uses the T-WSP for security-sensitive data handling. By restricting security-sensitive data access to T-WSP, ISO-WSP reduces the software complexity of trusted code, thereby improving the testability of ISO-WSP. To achieve end-to-end security, the application code is also decomposed into two parts, isolating a small trusted part from the remaining untrusted code. The trusted part encapsulates all accesses to security-sensitive data through a Secure Functional Interface (SFI). To ease the migration of legacy applications to ISO-WSP, we developed tools to translate direct manipulations of security-sensitive data by the untrusted part into SFI invocations. Using a prototype implementation based on the Apache Axis2 WSP, we show that ISO-WSP reduces software complexity of trusted components by a factor of five, while incurring a modest performance overhead of few milliseconds per request. We also show that existing applications can be migrated to run on ISO-WSP with minimal effort: a few tens of lines of new and modified code.

Keywords. Web Services Middleware, Security, TCBs

1. Introduction

Service-Oriented Computing (more recently also referred to as "service computing") is designed to support rapid creation of new, value-added applications and business processes that can span diverse organizations and computing platforms. Concretely, Paypal's Web Services, eBay Developer Program and Amazon Web Services are illustrative examples of web services being used in mission-critical, security-sensitive, and truly large scale applications. Despite the widespread deployment of web services, however, significant research challenges remain. This paper is concerned with the protection of security-sensitive information in service computing.

Web Service Platforms (WSPs) such as Apache Axis2, Microsoft .NET and IBM WebSphere provide essential functionality such as SOAP messaging and support for pub-

lishing and discovering web services. Additionally, WSPs provide desirable functionality such as support for web service composition, atomicity and message reliability. Support for such large and varied functionality has increased the size and complexity of current WSPs; for example, the open source Axis2 WSP and its extensions account for over 110,000 lines of code (LOC).

Current WSPs often perform all types of processing, including security-sensitive information handling, in the same protection domain. Therefore, all components of the WSP may have direct or indirect access to sensitive data, violating the Principle of Least Privilege [23]. Additionally, the large size and complexity of WSPs hinders their testability, resulting in systems with multiple security vulnerabilities [6,7]. Therefore, even though security-sensitive applications employ protocols such as SSL, TLS and WS-Security, attackers can compromise the flow of sensitive information by exploiting vulnerabilities in the large WSPs.

We propose ISO-WSP, a middleware architecture for web services to address the problem of protecting security-sensitive information flow against potential vulnerabilities inherent in large and complex software packages such as WSPs. Applying the AppCore approach [24], ISO-WSP decomposes current WSPs into two parts: a small, functionally-limited, trusted T-WSP and a large, functionally-rich, untrusted U-WSP. The T-WSP consists of components of a WSP that require access to security-sensitive information and the U-WSP contains the rest of the legacy WSP. The T-WSP and U-WSP are executed in separate protection domains, with the U-WSP invoking the T-WSP when it has to operate on security-sensitive data. By restricting security-sensitive data access to the small T-WSP, ISO-WSP eliminates the need to trust the U-WSP. Since the T-WSP is expected to be considerably smaller than the U-WSP, we improve the testability of the ISO-WSP.

To achieve end-to-end security, the AppCore approach also splits the application code into two parts, isolating a small trusted part from the remaining untrusted code. The trusted part encapsulates all accesses to security-sensitive data through a Secure Functional Interface (SFI). The untrusted part of the application is forced to use this interface if it wants to operate on security-sensitive data. To ease the migration of legacy applications to ISO-WSP, we developed tools to translate direct manipulations of security-sensitive data into SFI invocations, eliminating all access to security-sensitive data by the untrusted part.

We demonstrate the feasibility of our approach by implementing and evaluating an ISO-WSP based on the Axis2 WSP. We show that ISO-WSP results in a five-fold reduction in the size and complexity of the software that has access to sensitive data, while imposing a moderate overhead of few milliseconds per request. We also show that existing applications can be ported to ISO-WSP by adding or modifying few tens of lines of code.

The organization of the rest of the paper is as follows: Section 2 motivates the paper by first presenting the design of WSPs in detail and then discussing the security problems in current WSPs. Section 3 discusses the design of ISO-WSP and Section 4 discusses application-level support for the ISO-WSP architecture. Section 5 describes an implementation of the ISO-WSP architecture based on the Axis2 WSP and evaluates the resulting system. Section 6 discusses the related work and Section 7 concludes the paper.

2. Motivation

We first discuss the framework for web service platforms (WSPs) as specified by W3C. For concreteness, we also present the design of the Axis2 WSP, one specific implementation of the web services framework. Based on this knowledge, we discuss the security problems in current WSPs. For clarity we present definitions for the terms used in this paper:

- *Confidentiality*: Only authorized entities have access to data.
- *Integrity*: Only authorized entities are allowed to modify data and any modification by unauthorized entities can be detected.
- *Security-sensitive* (or) *Sensitive* data item: Any data item that the end-user or the business logic imposes confidentiality or integrity requirements is termed as security-sensitive, e.g., payment information, patient health information. Sensitive data items also include data items that may not be transmitted over the network, e.g., private keys used by the WSP.

2.1 Design of Web Service Platforms

W3C's web services architecture specification [9] specifies the basic framework for WSPs. WSPs such as Apache Axis2 and Microsoft .NET implement this framework. The framework consists of three entities: a service provider, a client and a web service discovery agency. A WSP is used to mediate interactions between the three entities and this requires support for three basic classes of functionality: exchanging messages, describing web services and publishing and discovering web service descriptions. Since this paper addresses the security of information exchanges between the service provider and the client, we focus on the first class of functionality: support for message exchanges.

Fig. 1 presents one view of the web services stack, as illustrated in W3C's specification [9]. All WSPs must implement a transport protocol, typically HTTP, and provide support for a message packaging mechanism, typically SOAP. The WSPs might also choose to support additional message packaging and transport mechanisms such as MIME over SMTP. In addition to the basic features, the WSP must also support functionality such as routing, transaction support, message reliability, security and quality of service. W3C has also standardized many types of additional functionality in the form of WS-* extensions such as WS-Addressing, WS-Security amongst others. Table 1 lists some of the WS-* extensions and briefly describes the functionality of each type of extension. WSPs may also possess an extension architecture to seamlessly integrate new and upcoming WS-* specifications or to provide support for custom extensions for logging or load-balancing.

The Apache Axis2 WSP [20] is one implementation of the web services framework. We use the Axis2 WSP in our analysis as not only is it widely used, it is also available under an open source license, enabling us to gain a clearer understanding of its workings. Axis2 provides support for developing, deploying, managing and invoking web services. Addi-

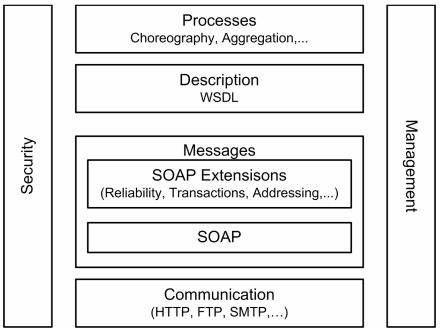

Fig. 1. Web Services Stack. From W3C's Web Services Architecture specification [9].

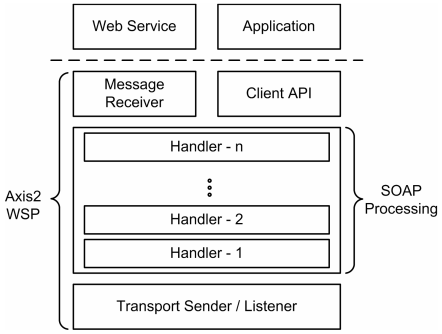

Fig. 2. SOAP Processing Model in Apache Axis2.

Table 1. List of some WS-* extensions and the functionality provided by each of them [4]

Extensions	Functionality
WS-Addressing	Provides a uniform naming scheme to address end-points.
WS-Security, WS-Trust, WS-SecureConversation, ...	Securing messages and establishing trust across different domains.
WS-AtomicTransaction, WS-Coordination, ...	Transaction Support for web services.
WS-ReliableMessaging	Support for reliable delivery of messages over non-reliable communication channels.
WSDL, WS-Policy, ...	Support for description and exchange of metadata between clients and service providers.
WS-Eventing	Event-driven programming support for web services

tionally, Axis2 also provides support for utilizing many WS-* extensions such as WS-Security and WS-ReliableMessaging. As we are primarily interested in the processing of web service requests and responses, we focus on the SOAP processing model of Axis2 [2]. Fig. 2 illustrates the SOAP processing chain for a web service request or response in the Axis2 WSP. Axis2 handles all message interactions using two basic message handling sequences: *In Pipe* and *Out Pipe* for incoming and outgoing messages respectively. As the basic structure of both sequences is similar, we use the *In Pipe* as an illustrative example.

An incoming web service request is first received by the Transport Listener. The Transport Listener creates a message context and starts the In Pipe, which consists of a series of handlers. First, the transport handlers are used to verify transport protocol headers and populate the message context with data from the message. Next, handlers are used to process addressing headers and determine the target service. After this, user specified handlers are executed to perform tasks such as security processing, transaction support or reliable messaging support. Finally, the message is deserialized into language level objects and given to the web service implementation, which executes the business logic.

Axis2 uses handlers to implement and support various types of optional functionalities. For example, handlers are used to support many of the WS-* extensions like WS-Security

and WS-Addressing. Additionally, Axis2 allows for custom handlers to perform web service-specific, non-standard tasks such as admission control, load balancing or logging.

2.2 Security Problems in Web Service Platforms

The WS-Security specification [5] was designed to protect the confidentiality and integrity of information flow in web services. However, WS-Security-based protection can be bypassed by exploiting vulnerabilities in endpoint software. On the server side, attackers can compromise information flow by exploiting vulnerabilities in server software: operating system, web server, WSPs [6,7], or the business logic and its support software (e.g., database software). Similarly, on the client side, attackers can leverage vulnerabilities in client-side WSPs or client applications like the browser. We focus on securing WSPs in this paper. Securing operating systems, support software and applications such as the browser is outside the scope of this paper.

From a security perspective, WSP implementations have two significant issues. First, at the component level, WSPs violate the principle of least privilege (PoLP). PoLP states that components should execute with the least set of privileges necessary to finish the job. WSPs contain many components that do not need access to sensitive data, e.g., transport protocol implementation and WS-* extensions such as WS-Addressing. However, all these components execute in the same address space and same protection domain. Hence they can either directly access security-sensitive data or modify WS-Security processing by modifying security processing parameters and compromise security-sensitive data.

Concretely, in the Axis2 WSP, all handlers execute in the same protection domain as the rest of the WSP and the application-level code. Therefore, handlers that do not require access to sensitive data get access directly (reading message contents) or indirectly. For example, all Axis2 handlers for a given service have access to the service context that contains WS-Security processing parameters. Therefore, misbehaving handlers, even those that operate on encrypted data can compromise information flow by changing WS-Security processing parameters, e.g., by specifying the use of a weak encryption key.

Secondly, a WSP is a complex piece of software. As seen earlier (§ 2.1), WSPs are expected to perform a large number of tasks. Unsurprisingly, they contain large code bases. Additionally, since WSPs have to be configurable and extensible, they also typically possess configuration files, an extension-architecture and multiple extensions. Concretely, the Axis2 WSP alone contains about 23.5 KLOC. Together with the implementations of the multiple WS-* specifications, the WSP contains over 110 KLOC. Additionally, programmers can write custom handlers to carry out other types of processing like load balancing or admission control. These components add to the complexity of the system. Given the large code base and the multitude of ways in which these components can interact with each other, it is not feasible to exhaustively test the components of the WSP, especially as a complete system, and eliminate all vulnerabilities. Moreover, the configurable nature of WSPs means that extensions can be added, enabled and disabled at runtime, further complicating the analysis and testing of WSPs.

3. ISO-WSP

To address the security problems discussed in Section 2.2, we present a new secure middleware architecture for Web Services. The architecture, called ISO-WSP, splits the WSP into two parts: a small trusted T-WSP and a legacy, untrusted U-WSP. The T-WSP consists of components that *require* access to plain-text sensitive data. The U-WSP contains rest of the functionality of legacy WSPs and invokes the T-WSP when it needs to handle security-sensitive data. ISO-WSP manages the flow of information such that sensitive information in plain-text format is only available to the T-WSP. The T-WSP and the U-WSP are then executed in separate protection domains, with the U-WSP being executed as a lower privilege process.

The ISO-WSP architecture has three benefits: First, by limiting access of security-sensitive information to components that need access, ISO-WSP addresses the PoLP problem. Secondly, only the T-WSP has to be tested or verified to ensure security of sensitive data. Since the T-WSP is expected to be smaller than the complete WSP, ISO-WSP reduces the size of the Trusted Components and makes exhaustive testing or formal verification more feasible. Finally, by reusing the U-WSP to operate on non-sensitive or protected data, ISO-WSP also ensures that a majority of functionality expected from typical WSPs is retained.

We first discuss the criteria for determining WSP components that require access to sensitive data. Next, we present the ISO-WSP architecture and discuss the interaction mechanism between the T-WSP and the U-WSP. Finally, we discuss how we limit the flow of sensitive information to the T-WSP, thereby negating the need to trust the U-WSP.

3.1 Categorizing Components of WSPs

Certain components of a WSP require access to plain-text security-sensitive data. Since these components have to be allowed to access or modify sensitive data, they have to be trusted by both the end-user and the business logic. We call such components *Trusted Components* and these components taken together are referred to as the T-WSP. The rest of the components do not have to be trusted and are henceforth referred to as untrusted components (and collectively referred to as the U-WSP).

In our analysis, we assume that all security-sensitive information is protected using WS-Security [5]. While the SSL and TLS protocols can be used to protect information, they are not always suited for web services because of the unique needs of web services, e.g., support for fine-grained encryption or signatures such as signing SOAP headers. Hence, we do not consider them in our analysis.

Identifying the components of the T-WSP requires understanding of the various components of the WSP and their corresponding functions. We relied on W3C's architecture specification [9], specifications for the WS-* extensions such as WS-ReliableMessaging, WS-Addressing [4], and the Axis2 web services middleware architecture guide [2]. Based

on our analysis, we found that the security-related extensions such as WS-Security, WS-Trust and WS-SecureConversation are the components of the WSP that have to be trusted. These components have to be trusted for one of two reasons:

- **They require access to sensitive contents of a message (Direct Access)**. WS-Security implementations fall under this category as they need access to sensitive contents either to protect them (encrypt or sign) or to verify the protection on them (decrypt or verify signatures).
- **Or, they control behavior of components with direct access (Indirect Access).** This includes other security specifications such as WS-Trust and WS-SecureConversation. The implementation of these specifications have to be trusted, even though they do not require access to sensitive data, as these components control WS-Security processing. A malicious implementation can subvert WS-Security processing, e.g., employ a weak encryption key, to gain access to or leak sensitive information.

The rest of the components of the WSP, including the WS-* extensions, are agnostic to message contents and can be treated as untrusted components. For example, the transport layer protocol implementation or SOAP implementation is agnostic to message contents as long as the contents confirm to the SOAP message format. However, there is a special class of untrusted components that requires access to sensitive data such as signature or encryption keys, albeit indirectly. WS-* extensions such as WS-Addressing recommend the use of WS-Security to protect the integrity and in some cases, the confidentiality of SOAP message headers. These extensions require the use of WS-Security and associated signature or encryption keys. Even though they are indirectly dependent on sensitive data, they do not require access to sensitive data, especially the contents of the message body. While these extensions are crucial to maintain the reliability and availability properties of web services, they do not affect the confidentiality and integrity of end-user data. Therefore, we do not treat them as Trusted Components.

3.2 Architecture of ISO-WSP

Fig. 3 provides an overview of the architecture of ISO-WSP. As explained previously (§ 3.1), implementation of security specifications form the T-WSP, and the rest of the components form the U-WSP. In each case, the components are grouped together and executed as independent applications. In addition to the separation of components of the WSP, the configuration files and the application-level code too have to be classified and separated into two categories. Since the behavior of the security specifications can be controlled by configuration files, the corresponding configuration files also have to be trusted and secured from access by untrusted components.

We enforce separation between the T-WSP and the U-WSP by executing them in separate protection domains, with the U-WSP running with lower privileges. This prevents U-WSP from modifying the binaries or configuration files of the T-WSP. This separation also prevents U-WSP from accessing the secret keys used for encryption or decryption. However, separation alone does not prevent the flow of sensitive information to the U-

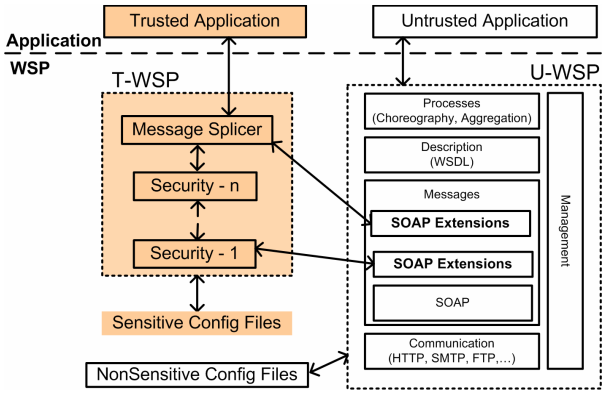

Fig. 3. Architecture of ISO-WSP. Shaded boxes represent Trusted Components. Trusted Components execute in a separate protection domain.

WSP or to the untrusted parts of the application. The Message Splicer, to be discussed in Section 3.3, addresses how we secure the flow of sensitive information in the ISO-WSP. Section 4 discusses our approach of protecting the flow of sensitive information in the application-level code in detail.

A legacy WSP can be converted to an ISO-WSP with a small number of modifications. After constructing a T-WSP, we have to modify the legacy WSP to invoke the T-WSP via remote invocation mechanisms instead of using local calls. This involves identifying parameters that are exchanged between the T-WSP and U-WSP, message and results of security processing, and adding the necessary serializing and deserializing code.

One of the main features of the ISO-WSP architecture is that external entities do not perceive any changes to the functional interface of the WSP. ISO-WSP exports the same external interface as legacy WSPs. The U-WSP provides support for the interface by implementing transport-layer protocols such as HTTP over TCP/IP.

Another key feature of ISO-WSP is that the U-WSP can still execute WS-Security libraries in its protection domain. However, to reiterate, the U-WSP cannot access the secret keys of the T-WSP. Therefore, any data encrypted with these keys is inaccessible to the U-WSP. The U-WSP can use this feature along with weaker keys or keys with weaker security guarantees to provide limited security guarantees. For example, one can envision a service provider maintaining two sets of secret keys: one to protect the confidentiality and integrity of sensitive data items from any attacks on the U-WSP and another to protect all types of data from snooping attacks on the network. Depending on the security requirements of the data items or client preferences, the service provider can choose the set of keys to employ.

3.3 Securing Information Flow with the Message Splicer

Before we discuss information flow security in ISO-WSP, we list the basic assumptions and features of ISO-WSP. First, we assume that all sensitive information is protected using

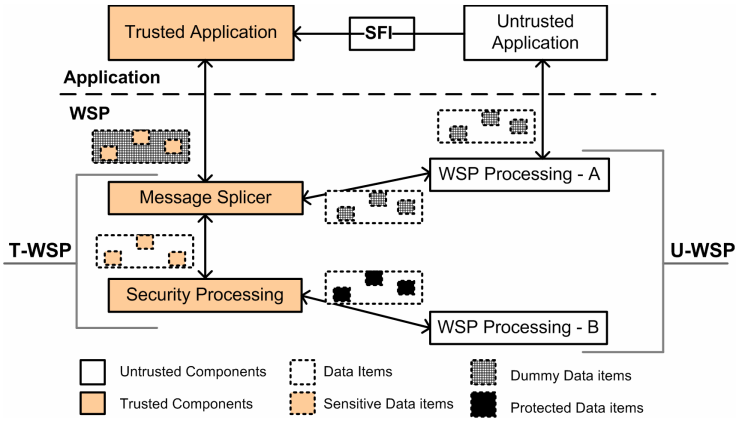

Fig. 4. Securing Information Flow in ISO-WSP. The T-WSP replaces security-sensitive data items with protected data items or dummy data items before passing them on to the U-WSP. SFI is discussed in Section 4.

WS-Security. Furthermore, we assume that sensitive information protected using WS-Security cannot be compromised without first compromising the security extensions. Secondly, WS-Security processing is carried out in a separate protection domain, with the U-WSP being executed as a lower privilege process. Therefore, untrusted components cannot indirectly access sensitive data, e.g., they cannot change the parameters of security processing. We only have to prevent direct access of sensitive information by the U-WSP.

We prevent direct access of sensitive information by the U-WSP by hiding sensitive information from it. ISO-WSP contains a Message Splicer[1] whose function is to replace sensitive data with dummy, non-sensitive data or vice versa. For an incoming message, the Message Splicer replaces sensitive data items with dummy data items and a token that uniquely identifies an instance of a sensitive data item and transfers the message to the U-WSP. The token acts as a capability for the sensitive data item. The Message Splicer passes the sensitive information directly to the trusted part of the application. For an outgoing message, the Message Splicer accepts sensitive data from the trusted part of the application and non-sensitive data from the U-WSP, composes them in a single message, and passes it on for security processing. Sensitive data items are protected (encrypted or signed) before being transferred to the untrusted WSP for further processing and transmission. This process is illustrated in Fig. 4.

To analyze how this process of splitting and merging secures flow of sensitive information in contrast to legacy WSPs, we divide the untrusted WSP into two categories:

- Components that operate below WS-Security (*WSP Processing–B* in Fig. 4): Since these components worked on encrypted or signed data, they did not have *direct* access

[1] Message Splicer is derived from *Gene Splicing*. Gene splicing is the process of creating recombinant DNA by cutting and joining DNA sequences from multiple DNA fragments.

to sensitive data in legacy WSPs. However, they could indirectly compromise flows by modifying security processing parameters, e.g., overwrite configuration files or security processing parameters. In ISO-WSP, they execute with lower privilege and in a separate protection domain and hence, cannot manipulate security processing parameters.

- Components that operate above WS-Security (*WSP Processing–A* in Fig. 4): These components had both *direct* and *indirect* access to sensitive data in legacy WSPs. In ISO-WSP, we replace sensitive data with dummy data items before transferring the message back to these components. Hence, these components are deprived of direct access to sensitive data. Previously discussed arguments against indirect access by untrusted components are equally applicable to these components.

4. Application support for ISO-WSP

From the application developer's point of view, ISO-WSP represents a departure from existing WSPs. The developer has to consciously split the application program into a trusted and an untrusted part. This is a potential drawback for the ISO-WSP architecture as the developer will have to expend additional effort while designing a new application. Additionally, legacy code can no longer be executed on ISO-WSP without appropriate modifications. In this section, we will show that the application developer only needs to carry out some simple steps to design a new application or port a legacy application to ISO-WSP. We will also see that these steps fit in naturally with security-aware design of web service applications.

Modifications to the application level software can be classified into two types: modifications to data structures and modification to code. We capture these modifications by asking the application developer to define an interface, called the Secure Functional Interface, to security-sensitive objects that can be used by untrusted components. We first describe this interface and then show how modifications to data structures and code can be easily effected with the help of this interface. We use Java as the application-level programming language to illustrate the modifications to data and code. However, our approach is equally applicable to other object-oriented languages.

4.1 Secure Functional Interface: A Restricted Interface for Remote Access

To understand the need for defining a restricted interface for remote access of sensitive objects, consider a payment processing service as described in Listing 1-A (see last page). Of the data items, it is reasonable to expect that the customer would like to enforce strong confidentiality properties on the credit card information (Listing 1-B). In particular, the customer does not want the credit card information to be inappropriately stored or transferred. Object-oriented languages such as C++ and Java encapsulate data items, providing limited protection against inappropriate access. Even in such languages, sensitive objects contain functions that allow for retrieval of sensitive information, e.g., *getter* functions that

enable objects to be easily plugged into existing web service platforms (*getCcNum* in Listing 1-B). These methods can be invoked by untrusted objects to gain access to sensitive information. Therefore, we cannot export the complete list of methods supported by the sensitive object to the untrusted object.

We require the application developer to identify sensitive data items and define a restricted interface, called a Secure Functional Interface (SFI), for access by untrusted components. The SFI for a sensitive data item lists all operations on the data item that can be performed by untrusted components. The SFI approach assumes that remote access function implementations are well-designed, i.e., a remote access function does not return sensitive data to the caller or compromise the integrity of sensitive data. Listing 1-C presents the SFI for the CreditCard object, which no longer contains the *getter* functions, thereby denying access to sensitive information to the untrusted part of the application.

Defining SFI is not an undue burden as current language-level features such as the *protected* and *private* keywords in function signatures already make the developers aware of security implications of the function. Defining a SFI is similar to the use of those keywords; however, address-space separation between the trusted data items and untrusted components provides stronger security guarantees about information exchanges between them.

Limitation. Currently, SFI only allows calls from untrusted part of the application to the trusted part of the application. We feel that this is not a major limitation as we can design an analogous interface (say, a Functional Interface for non-sensitive objects) to support calls from trusted part to the untrusted part.

4.2 Modifications to Data Structures

Based on our definition of SFI, one would assume that application-level data structures would be unmodified in ISO-WSP. However, as described in Section 3.3, we replace sensitive data items with dummy items that contain tokens. Therefore we have to modify the corresponding application-level data structures to include a token field. On the untrusted side, we also have to add another field that holds the stub for invoking the remote procedure on the trusted side. Adding these fields can be automated: add the fields to a class inherited from the class mentioned in the SFI file (Listing 1-E). We refer to this class as an *annotated class*.

4.3 Modifications to Code

First, by modifying the local application-level data structures, we are invalidating the schema for the web service interface (WSDL specification). Therefore, we have to modify the local application-level code to use the serializer/deserializer for the annotated class instead of the one for the legacy class. Crucially, the remote application-level code (a client in the case the service provider uses an ISO-WSP or vice versa) does not have to be modified as the Message Splicer in the local ISO-WSP adds these new fields to an incom-

ing XML message and strips the fields from an outgoing message. Once again, we can use the list of sensitive classes from SFI specifications to configure the Message Splicer to strip or add token information to messages.

Secondly, when an untrusted component accesses functions specified in the SFI, it has to be converted into a remote call. This requires two modifications on the untrusted side: (a) parameters to the remote call have to be serializable. Most Java classes can be serialized by adding "*implements serializable*" to the class definition. The default serialization methods along with automatically generated getters and setters are sufficient for serializing and deserializing most classes. Otherwise, the developer will have to write custom serializers and deserializers. And (b), the untrusted classes have to be modified to call the remote interface in case the sensitive data resides in the trusted part of the application. This is easily accomplished by modifying the relevant method to pass the arguments along with the token to the trusted part of the application. On the other hand, if the data resides in the untrusted part of the application, then the local method is invoked.

At this point one should note that the untrusted part can modify itself to make a local call instead of a remote call. However, this would not affect the confidentiality or integrity of sensitive data, as the sensitive data is stored in the trusted part of the application.

On the trusted side, we have to write a remote interface server that handles requests from the untrusted part. The server maintains a collection of security-sensitive objects, indexed by their unique identifiers. Upon receiving a request, the server uses the unique identifier to retrieve the appropriate instance of the sensitive object. It then makes a local call with the arguments provided by untrusted and returns the result.

One potential weakness in the system is that the trusted part now performs operations using data provided by the untrusted part. This may violate the integrity flow in the system, as data is flowing from a lower integrity level (untrusted part) to a higher integrity level (trusted part). We assume that the trusted code validates all inputs from the untrusted part before proceeding with the operation, e.g., by comparing against a local copy.

5. Implementation and Evaluation

5.1 Implementation

We implemented an ISO-WSP prototype based on the Apache Axis2 platform. The T-WSP consisted of a WS-Security implementation (WSS4J [1]) and a Message Splicer. Accordingly, the configuration files for WS-Security also formed a part of the T-WSP. The rest of the Axis2 WSP, along with the other WS-* extensions formed the U-WSP. We modified Axis2 to make remote calls to perform WS-Security related processing. Since we used Java, all remote calls were Java RMI calls. This required around 100 new and modified lines of code (LOC). We also had to modify the WS-Security implementation to serialize and deserialize the parameters. Since the WS-Security configuration files were now a part of the T-WSP, we had to add code to read the configuration files. In all, we had to

modify or add around 700 LOC. Thus, by adding or modifying around 800 LOC, we were able to convert the existing Axis2 WSP into an ISO-WSP prototype.

We execute the T-WSP as a superuser process and the U-WSP as an unprivileged user. The file system permissions of the configuration files of the T-WSP are set such that the U-WSP is unable to read or modify them.

Our implementation of the Message Splicer accepts information about sensitive classes in two ways. First, it allows developers to specify XML files that contain serialized versions of an instance of a sensitive data item with dummy values, e.g., a credit card data item with an invalid card number. The Message Splicer uses these "dummy" values when replacing sensitive objects in incoming messages. It also inserts unique tokens when replacing sensitive objects. Secondly, the Message Splicer accepts sensitive data items in the form of Document Object Model (DOM) fragments from the trusted part of the application. Each fragment possesses unique tokens that are used by the Message Splicer when replacing dummy data items with actual content in outgoing messages.

We implemented a simple payment processing service (described in Listing 1 in the last page) that accepts order information and payment information in the form of a credit card object and returns a confirmation string containing a transaction identifier and the amount charged to the card. We denoted the credit card information as security-sensitive information. To protect this information, the client specifies that the whole request message be encrypted using WS-Security with the public key of the service provider as the encryption key. In our legacy service implementation, the business logic accepts a DOM fragment containing the input parameters and deserializes it into Java objects. Next, it calls the charge card function to generate a confirmation number. The results are converted into a DOM fragment which is then passed on to the WSP for further processing and transmission.

In the implementation on top of ISO-WSP, the developer first specifies an SFI as illustrated in Listing 1-C. We then use our code generator to generate the necessary class files. On the untrusted side of the application, the developer has to modify the existing service in two places: First the developer has to use the serializer/deserializer for the annotated security-sensitive classes. This involves modifying 20 lines of code. Secondly, the developer has to add a call to the cleanup function of the credit card class, so that the trusted object can be freed after request processing. On the trusted side of the application, the developer has to add code to interface with the T-WSP (about 10 LOC). With these modifications, the legacy service can now run on the ISO-WSP implementation.

For an incoming request, both the trusted part and the untrusted part of the application receive DOM fragments, which are deserialized into Java classes. The trusted part registers itself with a Credit Card server that is executing on the T-WSP. The untrusted application calls the charge card function using the SFI. Hence, the call is redirected to the Credit Card server on the trusted side (Listing 1-F), which in turn invokes the charge card function of the appropriate instance of the credit card class. Once the untrusted part gets the confirmation number, it invokes the cleanup method to free the credit card object in the

trusted part of the application. Finally, the results are converted into a DOM fragment and passed on to the U-WSP.

5.2 Performance of ISO-WSP

Our experimental setup consisted of two machines, each with Pentium-4 3.0 GHz processors with 1 GB RAM, running Linux kernel 2.6.15. The machines were connected via a 100 MBps switch. We used Axis2 version 1.1 and WSS4J version 1.5.1 running on top of Apache Tomcat 4.1.31.

There are three sources of overhead in the ISO-WSP. First, since the security-sensitive parts of the WSP and application are separated from the rest of the application, there is the added cost of remote calls. Related to this is the cost of serializing and deserializing data items for remote calls. Finally, there is the cost of performing message splicing operations.

To estimate the RMI overhead, we use a simple echo application with the client and server running on the same machine. Round trip times increased linearly with message sizes, at the rate of about 0.5 ms per kilobyte. For message sizes less than 8 KB, round trip time was less than 1 millisecond which is negligible when compared to typical web service response times which are of the order of hundreds of milliseconds [16].

In the ISO-WSP architecture, we have to transfer SOAP messages between the U-WSP and the T-WSP. This involves converting SOAP messages in DOM format to byte array format and vice versa. To estimate these costs, we used an XML data set from the *XMLBench Document Model Benchmark* [10]. We evaluated the cost of serializing and deserializing the DOM representation of the data set for two XML parsers, the AXIOM pull parser used in the Axis2 WSP (and the U-WSP) and the default Xerces parser configuration used in the T-WSP. At this point, we want to reiterate that we only want an estimate of the costs of serializing and deserializing SOAP messages. We are not attempting to rigorously analyze the performance of the XML parser. For the data set, we found that combined serializing and deserializing costs for small to medium sized XML files (~10 kilobytes) to be less than 4 ms. However, the combined costs for the largest XML file in the dataset (36 KB) was about 14 ms. We omit detailed results due to space limitations.

We do not perform similar microbenchmarks for estimating message splicing costs and application-level serializing and deserializing costs because they are closely dependent on the application-level data structures. Rather, we measure these costs as a part of a concrete web service implementation.

We use the payment processing web service described earlier (§ 5.1) to evaluate the end-to-end overhead imposed by the ISO-WSP architecture. We found that using ISO-WSP increased the average response time from 40.3 ms to 47.9 ms. To accurately characterize the overheads imposed by ISO-WSP, we also measured the time spent in each stage of the ISO-WSP and the results are presented in Fig. 5. We see that the overhead of the additional stages is about 8.8 ms. However, the ISO-WSP implementation as a whole is only 5.2 ms slower than the Axis2 WSP. We found that this is because the Axis2 WSP

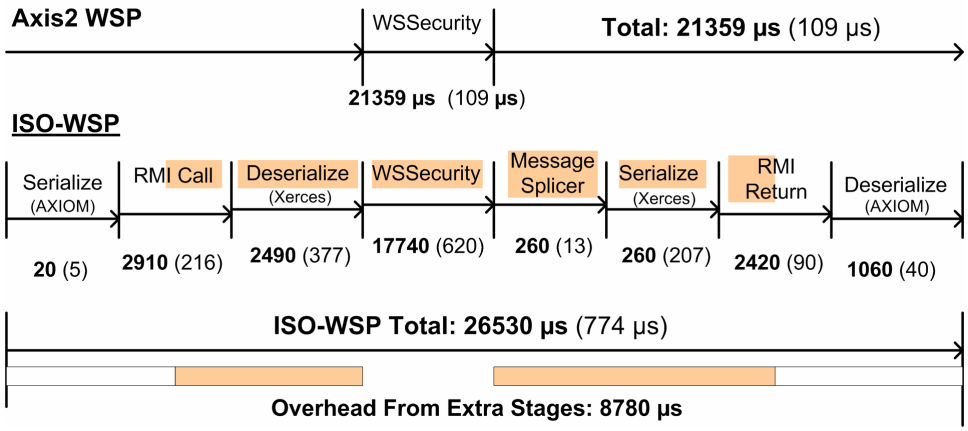

Fig. 5. Comparison of security processing costs for an incoming message. All numbers in microseconds. Numbers in parentheses indicate the 95% confidence interval.

uses the pull-based AXIOM parser whereas the T-WSP in our implementation uses the default Xerces parser configuration. Since WS-Security processing typically involves parsing the whole message (for full message signatures or encryption), the performance of WS-Security with pull-based parsers suffers. However, as expected, the AXIOM parser is faster when serializing and deserializing SOAP messages.

The rest of the costs (~ 2.4 ms) are distributed in the application-level code. These include the cost for two RMI calls from the untrusted part to the trusted part (~0.8 ms): one for charging the credit card and another for cleaning up the state in the trusted part. The second major cost arises because the SOAP message now has to be additionally deserialized on the trusted side (~1.5 ms).

Based on the microbenchmarks and the results for the payment processing web service, we can see that ISO-WSP introduces overheads of the order of a few milliseconds. While this might seem excessive, typical web service invocation time ranges from 0.5 seconds to a few seconds [16]. More importantly, one should note that ISO-WSP is invoked only during the exchange of sensitive information. When exchanging non-sensitive information, ISO-WSP still uses the legacy WSP, thereby maintaining the performance of the legacy WSP.

Moreover, as mentioned in Section 3.2, ISO-WSP is designed such that WS-Security processing can still be performed in the U-WSP. Ideally, the service provider would provide stronger security guarantees when the T-WSP is involved (e.g., protection from potential compromise of large U-WSP). By careful design of web service interfaces, exchange of sensitive information can be separated from performance critical operations. For example, a real-time stock quote service can export two interfaces: an authentication interface that accepts an account password and returns a session identifier and a querying interface that accepts the session identifier and returns stock quotes. And instead of providing a

single public key, the service provider exports two public keys: one for the authentication interface (AI key) and another for the querying interface (QI key). Since the authentication interface involves exchange of account password, this stage is processed using the AI key and the T-WSP. Once account password has been validated, information exchanges for retrieving quotes can be carried out using the QI key and the U-WSP, thereby limiting the performance impact of ISO-WSP.

5.3 Security Properties of ISO-WSP

In Section 3.3, we discussed how ISO-WSP limits flow of sensitive information to components that need access, thereby addressing PoLP issues at the component level. To recap, the T-WSP consists of components that require either direct or indirect access to sensitive data. We augment the T-WSP with a Message Splicer, which replaces sensitive data with dummy data items before releasing the message to the U-WSP. Therefore the T-WSP only releases protected (encrypted or signed) sensitive data to the untrusted part. Moreover, since we execute the T-WSP in a separate protection domain, the U-WSP cannot interfere with the processing in the T-WSP.

Our second motivating factor for constructing an ISO-WSP was the increased complexity of WSPs. Table 2 compares the software complexity of various WSP components and compares them against the T-WSP. We measured two properties: Source Lines of Code and McCabe's Cyclomatic complexity [18]. Empirical studies have shown that both measures of software complexity correlate with number of bugs in code [19]. We see that the T-WSP is a factor of 5 smaller and simpler than the current implementation of the Axis2 WSP, making the T-WSP more amenable to exhaustive testing. The small size of the T-WSP also makes it easier to apply static analysis techniques for monitoring the flow of information, as described in [21].

Extensibility of WSPs is a crucial factor in testing and analysis. Since extensions can change the behavior of WSPs and since they can be added or configured at run time, they complicate the testing process. However, extensions provide useful functionality such as support for addressing, transactions and reliability. By extracting a T-WSP and retaining the functionality of the legacy U-WSP, ISO-WSP attempts to gain the best of both worlds. By limiting extensibility and configurability in the T-WSP, ISO-WSP ensures that the testing process is simplified for T-WSP. And by retaining functionality in the U-WSP, ISO-

Table 2. Comparison of Source Lines of Code (SLOC) and McCabe's Cyclomatic Complexity (MCC) of the T-WSP and the Axis2 WSP along with its extensions. Extensions include implementations of WS-Coordination, WS-ResourceFramework, WS-Addressing, amongst others. All numbers were generated using the JavaNCSS tool [3].

Module	Axis2	Extensions	WS-Security	WSP-Total	T-WSP
SLOC	23,580	70,350	16,900	110,830	19,360
MCC	7,930	24,100	5,180	39,210	6,050

WSP ensures that developers will still be able to use and configure extensions according to their needs.

Limitations. We would like to note that the ISO-WSP is vulnerable to Denial of Service attacks, wherein the U-WSP either corrupts messages or does not invoke the T-WSP for security processing. While this affects the availability of the service, it does not compromise the confidentiality or integrity of security-sensitive information.

ISO-WSP can be enhanced with Trusted Computing hardware [8] and application level support for Trusted Computing, e.g, Integrity Measurement Architecture [22], to provide additional, desirable security properties such as integrity of software stack at load time and remote attestation.

6. Related Work

Previous efforts have addressed refactoring existing systems software [11,15] and application-level software [24] to reduce the size and complexity of software that has access to sensitive data. Our work can be considered as an application of such techniques to web services middleware. In contrast to previous approaches, by replacing sensitive data items with dummy data items in the middleware, we also enable existing application-level software to run with minimal modifications.

There is also considerable research into refactoring and customizing middleware to modularize and simplify them. Zhang et al. [30,31] and Eichberg et al. [13] use Aspect Oriented Programming techniques to modularize middleware and then, customize it according to the needs of applications. OpenCOM [12] and CompOSE/Q [26], amongst others, are examples of reflective middleware that allow for customization of middleware to suit the needs of application. ISO-WSP is an attempt at refactoring middleware with security as the driving factor. Techniques described in [30,31] are applicable to the construction of ISO-WSP.

Trusted Computing initiatives [8] attempt to reassure remote entities about the integrity of the software stack handling sensitive data. The Integrity Measurement architecture [22] aims to extend integrity measurement to include configuration files. WS-Attestation [29] and Trusted Web Services [25] attempt to incorporate Trusted Computing principles into the web services stack. However, an attacker can still compromise the integrity of the software stack by exploiting run time vulnerabilities or by loading malicious extensions at runtime. By reducing the size and complexity of the trusted part of the system, ISO-WSP enables the use of exhaustive testing or static analysis techniques, thereby reducing the number of run-time vulnerabilities.

The use of static analysis techniques [14,27] to identify bugs in code is gaining popularity. For strongly-typed languages such as Java, one can go even further and apply language-based information flow protection techniques [21]. However, the applicability of such techniques to a large code-base is a subject of open research and we expect such techniques to be more amenable to smaller code bases, such as that of the T-WSP.

Lastly, tokens used in ISO-WSP by untrusted applications to operate on security-sensitive data can be viewed as capabilities [28]. The ISO-WSP architecture can be considered as retrofitting a capability-based architecture in to WSPs. Protected Data Paths (PDP) [17] uses a similar approach to hide sensitive information from untrusted application level programs. PDP consists of kernel-level modules that traps I/O calls retrieving sensitive data and replaces sensitive data with tokens, thereby preventing application level programs from directly accessing them. ISO-WSP not only adds tokens, but it also sends back dummy data items with similar structure to the sensitive data item, thereby minimizing the changes to existing applications.

7. Conclusion

In this paper, we presented ISO-WSP, a secure middleware architecture to counter the problem of large and complex WSPs. ISO-WSP consisted of two parts: a small, trusted T-WSP that required access to security-sensitive information and a large and complex U-WSP that retained the features of legacy WSPs. By limiting the flow of sensitive information to the T-WSP, we ensured that the U-WSP does not have to be trusted, thereby improving the testability of the ISO-WSP. We also provided end-to-end support for ISO-WSP by splitting existing applications based on a Secure Functional Interface (SFI) to security-sensitive objects. Using a developer-specified SFI, we generated code that decomposed legacy applications into trusted and untrusted parts, thereby enabling it to run on top of ISO-WSP.

We demonstrated the feasibility of our approach by building an ISO-WSP based on the Apache Axis2 WSP. We showed that ISO-WSP reduced the software complexity of the trusted part by a factor of 5, while imposing a manageable overhead of a few milliseconds per request. We also showed that legacy applications can be ported to run on the ISO-WSP with few tens of lines of new and modified code.

References

1. Apache WSS4J. http://ws.apache.org/wss4j/
2. Axis2 Architecture Guide. http://ws.apache.org/axis2/1_0/ Axis2ArchitectureGuide.html
3. JavaNCSS. http://www.kclee.de/clemens/java/javancss/
4. Microsoft. Web Services Specifications. http://msdn2.microsoft.com/en-us/webservices /aa740689.aspx
5. OASIS Web Services Security (WSS) TC. http://www.oasis-open.org/committees/wss/
6. Secunia. IBM WebSphere Application Server 5.x – Vulnerability Report. http://secunia.com/product/2614/? task=advisories
7. Secunia. Microsoft .NET Framework 1.x – Vulnerability Report. http://secunia.com/product/667/?task=advisories
8. Trusted Computing Group. https://www.trustedcomputinggroup.org/home

9. W3C. Web Services Architecture. http://www.w3.org/TR/ws-arch
10. XMLBench Document Model Benchmark. http://www.sosnoski.com/opensrc/xmlbench/
11. D. Brumley, D. X. Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proc. USENIX Security Symposium*, San Diego, USA. Aug 9-13, 2004.
12. M. Clarke, G.S. Blair, G. Coulson and N. Parlavantzas, An Efficient Component Model for the Construction of Adaptive Middleware, In *Proc. Middleware 2001*, pp. 160-178, 2001.
13. M. Eichberg and M. Mezini, Alice: Modularization of Middleware Using Aspect-Oriented Programming, In *Proc. Software Engineering and Middleware*, pp. 47-63. 2004.
14. D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *18th ACM SOSP*. Banff, Canada, Oct. 2001.
15. Hohmuth, M., M. Peter, H. Härtig, and J. Shapiro. Reducing TCB size by using untrusted components – small kernels versus virtual machine monitors, in *Proc. of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, 2004.
16. Kim, S. M. and Rosu, M. C., A survey of public web services. In *Proc. of 13th WWW Conf. on Alternate Track Papers & Posters*, pp. 312-313, May 2004.
17. J. Kong, K. Schwan and P. Widener, Protected Data Paths: Delivering Sensitive Data via Untrusted Proxies, In *Proc. 2006 Intl. Conf. on Privacy, Security and Trust*, Ontario, Oct. 2006.
18. T.J. McCabe, A Complexity Measure, *IEEE Transactions on Software Engineering*, SE-2 No. 4, pp. 308-320, Dec. 1976.
19. N. Nagappan, T. Ball and A. Zeller, Mining Metrics to Predict Component Failures, In *ICSE 2006*, Shanghai, Nov. 2006.
20. S. Perera et al. Axis2, Middleware for Next Generation Web Services, In *Proc. ICWS 2006*, pp. 833-840, Sept. 2006.
21. A. Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. In *IEEE Journal on Selected Areas in Communications*, 21(1):5-19, January 2003.
22. R. Sailer, X. Zhang, T. Jaeger, and L. V. Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proc. of 13th USENIX Security Symp.*, pp 223--238, 2004.
23. J.H. Saltzer and M.D. Schroeder, The Protection of Information in Computer Systems, In *Proc. of the IEEE*, Vol.63, No.9, Sept. 1975, pp.1278-1308.
24. L. Singaravelu, C. Pu, H. Haertig, C. Helmuth, Reducing TCB Complexity for Security-Sensitive Applications:Three Case Studies, In *Proc. First Eurosys*, Leuven, Belgium, April 2006.
25. Z. Song, S. Lee and R. Masuoka, Trusted Web Service, In *2nd Workshop on Advances in Trusted Computing*, Tokyo, Japan, 2006.
26. N. Venkatasubramanian, et al., Design and Implementation of a Composable Reflective Middleware Framework. In *ICDCS 2001*. April, 2001.
27. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. of ISOC NDSS*, 2000.
28. Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F., HYDRA: the kernel of a multiprocessor operating system. *CACM* 17(6), Jun. 1974, pp. 337-345.
29. S. Yoshihama, T. Ebringer, M. Nakamura, S. Munetoh and H. Maruyama, WS-Attestation: Efficient and Fine-Grained Remote Attestation on Web Services, In *Proc. ICWS'05*, pp. 743-750, 2005.
30. C. Zhang, H.-A. Jacobsen. Refactoring Middleware with Aspects. In *IEEE Trans. on Parallel and Distributed Systems*. 14(11), p. 1058-1073, 2003.
31. C. Zhang, H.-A. Jacobsen. Resolving Feature Convolution with Horizontal Decomposition in Middleware. In *Proc. OOPSLA 2004*. p. 188-205. Vancouver, BC, 2004.

A) Web Service Port

```
PPResults ProcessPayment (OrderInfo ord,
CustomerInfo cinf, CreditCard cc);
```

B) Class Definitions

```
public class CreditCard{
  private String ccNum, Name, zip;
  private int expiryMon, expiryYr;
  /* Getters,Setters */
  public String getCcNum(){...}
  public void setCcNum(String num){.}
  /* Other getters, setters*/...
  /*Charge Card and return a Txn ID*/
  public String chargeCard(float
                amount) {...}
  /* Validate Card*/
  public boolean validate(){...}
  /* Additional Functions */ ...
}
```

Modifications Specified by Developer

C) SFI Definition

```
/* Classname and Namespace*/
class:=edu.gatech.cc.pp.CreditCard
ns:=http://pp.com/CreditCard
/* Interface */
interface CCsfi{
  String chargeCard(float amount);
  boolean validate();
}
```

D) Generated Interface

```
public interface CCsfi{
  public String chargeCard(String
sfiID, float amount);
  public boolean validate(String
                            sfiID);
  public void cleanup(String sfiID);
}
```

E) Generated Code on Untrusted Side

```
public class CreditCardUnt
          extends CreditCard{
  private String sfiID;
  private CCsfi stub = null;
  /* Getters and Setters for sfiID*/

  /* Initialize Stub */
  private void initStub(){ ... }

  /* override the methods defined
     in interface */
  public String chargeCard(float amnt){
    if(sfiID != null){
      initStub();
      return stub.chargeCard(sfiID,
   amnt);
    }else{
      /* Non Sensitive Object
         Call superclass method*/
      return super.chargeCard(amnt);
    }
  }
}
```

F) Generated Code on Trusted Side

```
public class RemoteCCSfiServer
              implments CCsfi{
  /*Store ref to CC*/
  static Hashtable ht;
  /*Add Entry to hashtable*/
  public static void add(String id,
              CreditCardTrust cc){
    ht.add(id, cc);
  }
  /* Init Remote Interface */
  public static void initRI() {...}

  /* Handle remote Calls */
  public String chargeCard(String
          sfiID, float amount){
     CreditCardTrust cc =
    (CreditCardTrust)ht.get(sfiID);
     if(cc != null)
       return cc.chargeCard(amount);
     return "INVALID";
  }
}
public class CreditCardTrust extends
                        CreditCard{
  private String sfiID;

  public void setSfiID(String id){
    RemoteCCSfiServer.add(id, this);
    sfiID = id;
  }
  /* Other functions */...
}
```

Listing 1. Partial Code Listing for a Payment Processing Service. Boxed area indicates modifications specified by developer. Highlighted areas indicate code reuse.