End-to-End Security of Information Flow in Web-based Applications

Lenin Singaravelu

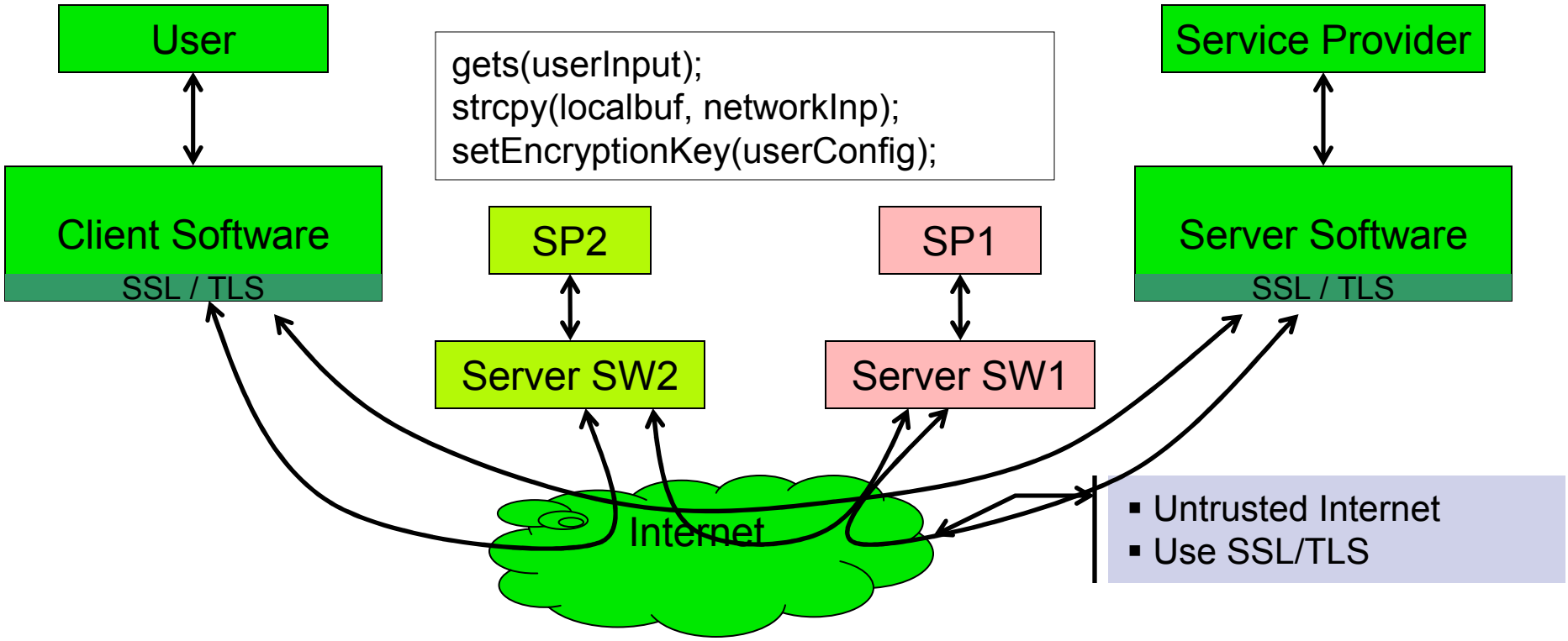
Joint Work with Calton Pu, Hermann Härtig, B. Kauer, A. Boettcher, C. Weinhold and Jinpeng Wei

Mot

Large and Complex end point software with

Web Service Compositions involve multiple service providers operating at different security levels

2 vulnerabilities per month

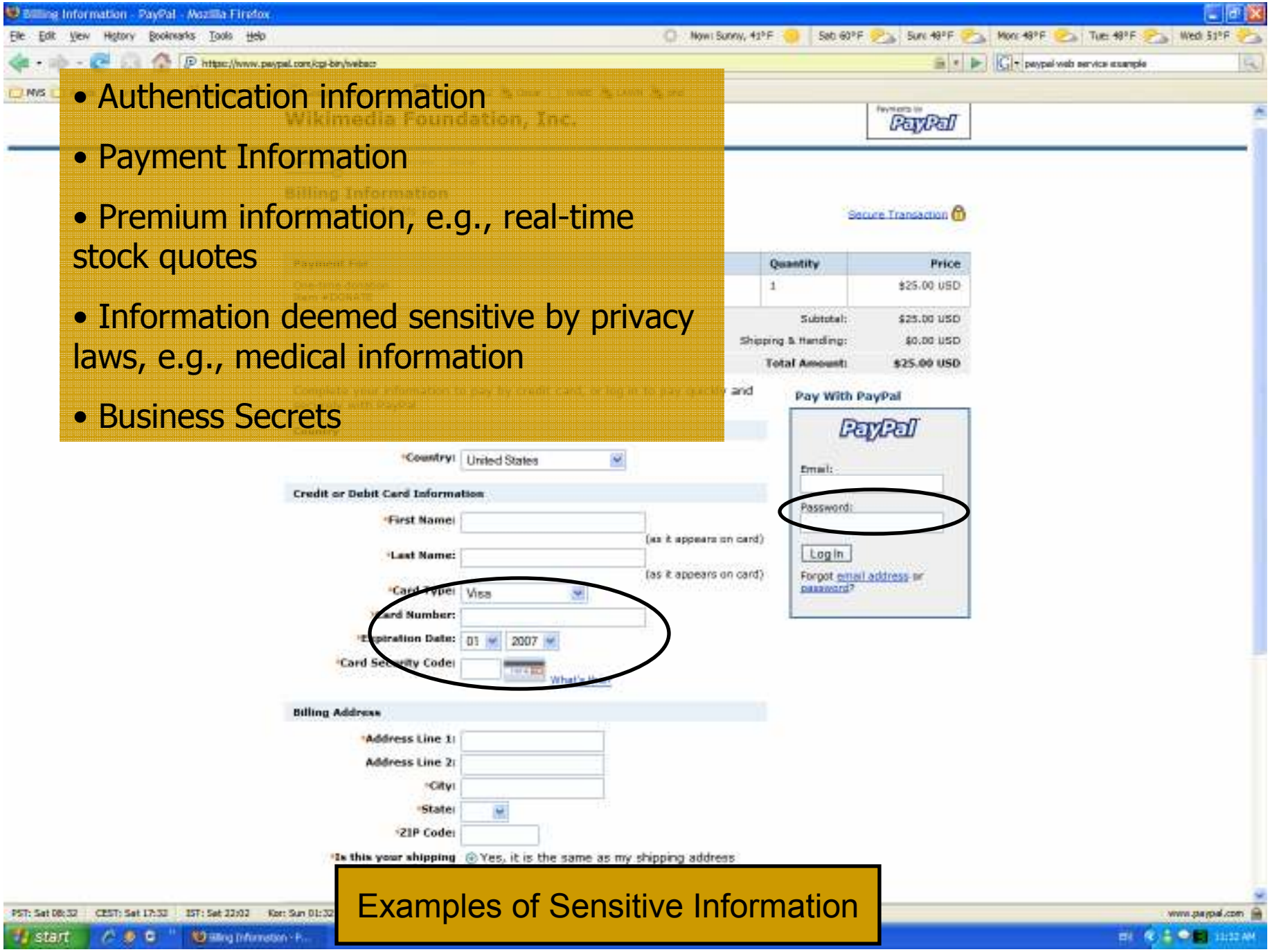


Talk Outline

- Problem Statement
- AppCore Approach
  - Client-side AppCore for https-based Applications
  - Server-side AppCore for Web Service Platforms
- WS-FESec for Web Service Compositions
- Related Work
- Conclusion

Terms and Definitions

- Security-Sensitive Information: Any piece of information that the *Business Logic* or *End-user* imposes confidentiality or integrity requirements
- Trusted Components: Components that operate in trustworthy manner
  - Allowed access to plain-text sensitive data
- Untrusted Components: No constraints on behavior


• Authentication information
• Payment Information
• Premium information, e.g., real-time stock quotes
• Information deemed sensitive by privacy laws, e.g., medical information
• Business Secrets
Billing Information - PayPal - Mozilla Firefox
Secure Transaction
Quantity
Price
Subtotal:
Shipping & Handling:
Total Amount:
Pay With PayPal
Email:
Password:
Log In
Credit or Debit Card Information
Billing Address
Examples of Sensitive Information

Flow of Sensitive Information

- On Network: sensitive information is protected using security protocols such as SSL
  - Protects from snooping or message modification attacks on the Internet
- On End-Points: Large and Complex software used to handle sensitive information in unprotected format
  - Protocols assume end-point software is free from vulnerabilities

Flow of Sensitive Information

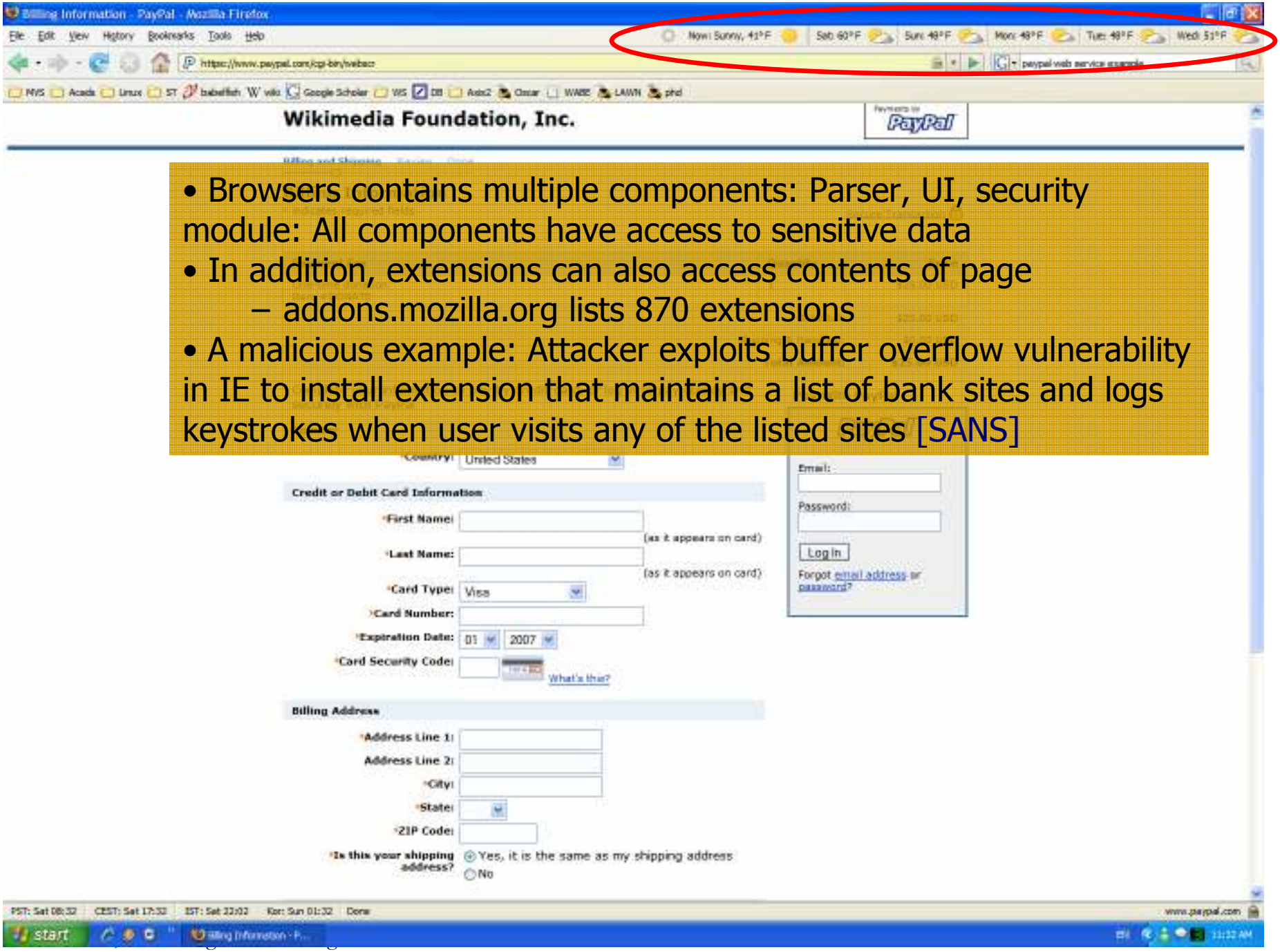
- Via Intermediate Service Nodes that provide value added services in Web Service Compositions
  - e.g., Google maps mashups, Amazon book search for mobile devices
- Intermediate nodes must be allowed to selectively read and modify messages
- SSL, TLS are coarse grained and point to point
- WS-Security currently does not support end-to-end confidentiality in open environment

Challenges in Protecting Sensitive Information

- Security Problems
  - Violation of Principle of Least Privilege
  - Increasing Software Complexity
  - Misbehaving Intermediate Services
- Usability Challenges
  - Popularity of Legacy Software

Security Problems: PoLP

- Security processing co-located with other components
  - Improves performance, as there are no security boundaries to cross
- Components that do not require access to sensitive data have access – Violates Principle of Least Privilege [Saltzer&Schroeder74]



- Browsers contains multiple components: Parser, UI, security module: All components have access to sensitive data
- In addition, extensions can also access contents of page
  - addons.mozilla.org lists 870 extensions
- A malicious example: Attacker exploits buffer overflow vulnerability in IE to install extension that maintains a list of bank sites and logs keystrokes when user visits any of the listed sites [SANS]

Security Problems: Software Complexity (SoCx)

- Large and Complex End-Point software
  - Sensitive and non-sensitive information handled by same software
  - Increased functionality => larger software
- Run-time Extension Mechanisms in some software further increases complexity
- Examples: 1 MLOC for Mozilla v1.0, >2 MLOC for Mozilla Firefox, 110KLOC for Web Service Platforms

SoCx and Vulnerabilities

- Greater software complexity implies more software errors
  - Larger code base is harder to analyze, test and verify
- Software complexity metrics such as LOC, Cyclomatic Complexity, Henry & Kafura's Information Flow metric exhibit positive correlation with software errors [Shepperd93]
- ~2 vulnerabilities per month with IE and Firefox [Secunia]
- Arbitrary code execution, Security bypass vulnerabilities in Web Service Platforms [Secunia]

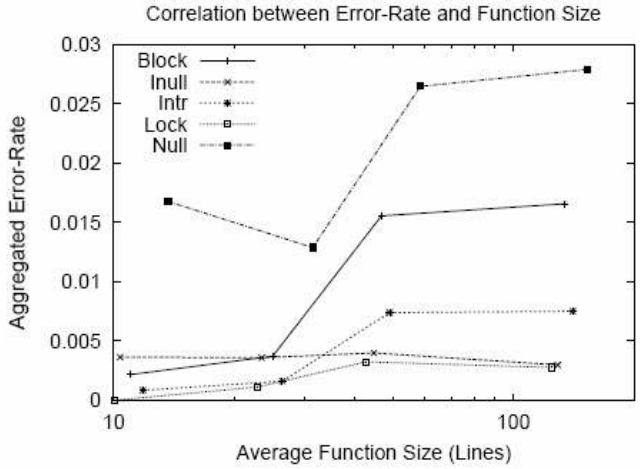


Figure 5: This graph shows the correlation between function sizes and error rates. It is drawn by sorting the functions that have notes by size, dividing them equally into four buckets, and computing the aggregated error rate per bucket for each checker. For all of the checkers except `Inull`, large functions are correlated with higher error rates.

Function size vs. Bugs
Chou et al. SOSP 2001

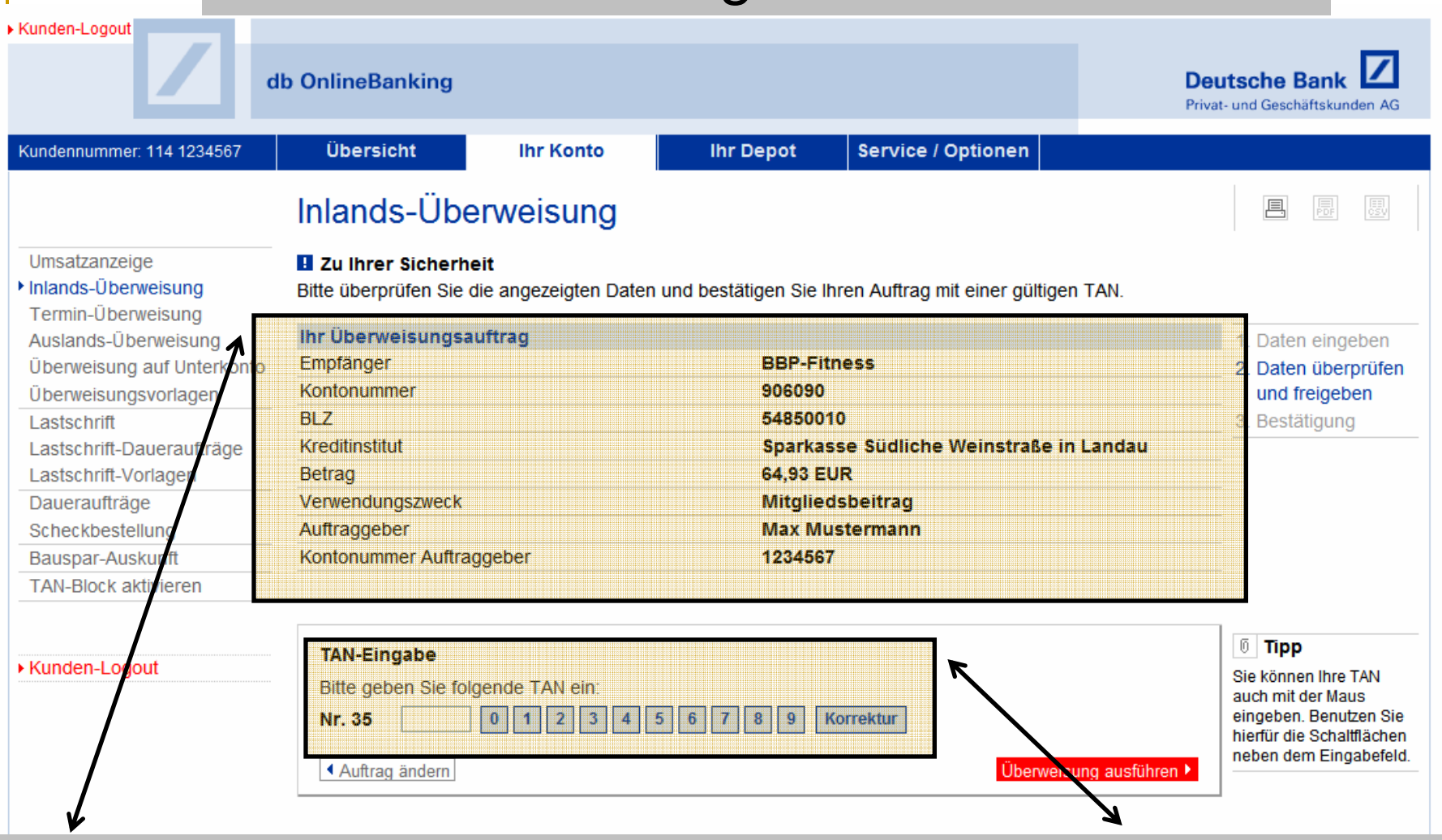
Usability Challenges

- Popularity of Existing Interfaces
  - Users familiar with GUI of browsers
  - Large number of legacy programs dependent on legacy interfaces (e.g., interface between application-level software and middleware)
  - Remote interfaces such as HTTP widely used over the Internet
- Users/Developers used to software with large number of features
  - App. Customization is a desirable feature: 2295 extensions for Firefox
  - Extension architecture of WSPs allows developer to add features such as load balancing, logging, etc…
- Cannot Completely Avoid Reuse of Legacy Code & Interfaces

Opportunities

- Web-based applications exchange information with varying sensitiveness
- Online Banking Session: login+password, Account Status Information, Account Modification information
  - Banks recognize this difference: Transaction Authorization Numbers

Funds Transfer Page of Deutsche Bank



User Sets Transfer Parame during previous Interaction

One Time TAN Number to Authorize Transfer

… Opportunities

- Use different components to handle information with differing sensitivities
  - Potentially reduces functionality of software handling sensitive information

Solution: AppCore Approach

Goal: Restrict the Flow of Security-Sensitive Information to Components that Need Access

Approach

- Split Application into Trusted and Untrusted Parts
  - Trusted Part consists of components that **require** access to sensitive data
- Hide Sensitive data from untrusted part
- Execute Trusted and Untrusted parts in separate protection domains

Design Goals

- Identify Trusted Components and Limit Flow of Sensitive information to such components
- Reduce complexity of Trusted Components
- Reuse Legacy Code as far as possible
- Reuse Legacy Interfaces as far as possible
- Minimize Performance Overheads

Addressing Security Problems

- PoLP: We hide sensitive information from components that do not need access
- SoCx: Trusted Components a subset of complete application
  - Expect diminished complexity

Applications of AppCore Approach

- E-Commerce Transaction Client AppCore
  - Reduced complexity by over order of magnitude
- VPN AppCore and E-Mail signer AppCore reduce complexity by 3X to 5X
- Similar approaches employed to reduce complexity in system services, e.g., SSH [Provos03] and Device Drivers [Ganapathy07]

Timeline

- Problem Statement
- AppCore Approach
  - Client-side AppCore for https-based Applications
  - Server-side AppCore for Web Service Platforms
- WS-FESec for Web Service Compositions
- Related Work
- Conclusion

https-Based Applications

- Online banking, electronic commerce …
  - Use HTTP over SSL to protect information flow over network
- Client application of choice, the Browser, contains multiple security vulnerabilities
- Service Providers recognize this and use one-time keys, dual-factor authentication, mouse input instead of keyboard input etc…
  - Attackers run sophisticated malware to bypass such systems: e.g., Screen Scraper [APWG05]

Client-Side AppCores for https-based Applications

- Leverages fact that sensitiveness of information in https-based applications differs
- e.g., Online banking application exchanges 3 types of information
  - Non-sensitive: bank front page, etc..
  - Low-Sensitivity: Sensitive information but requires lot of functionality: e.g.,Account status with graphs, spreadsheets
  - High-sensitivity: Information that leads to irrevocable account modifications, e.g., TANs leading to funds transfer
- Use AppCore for High-sensitivity information, legacy applications for rest

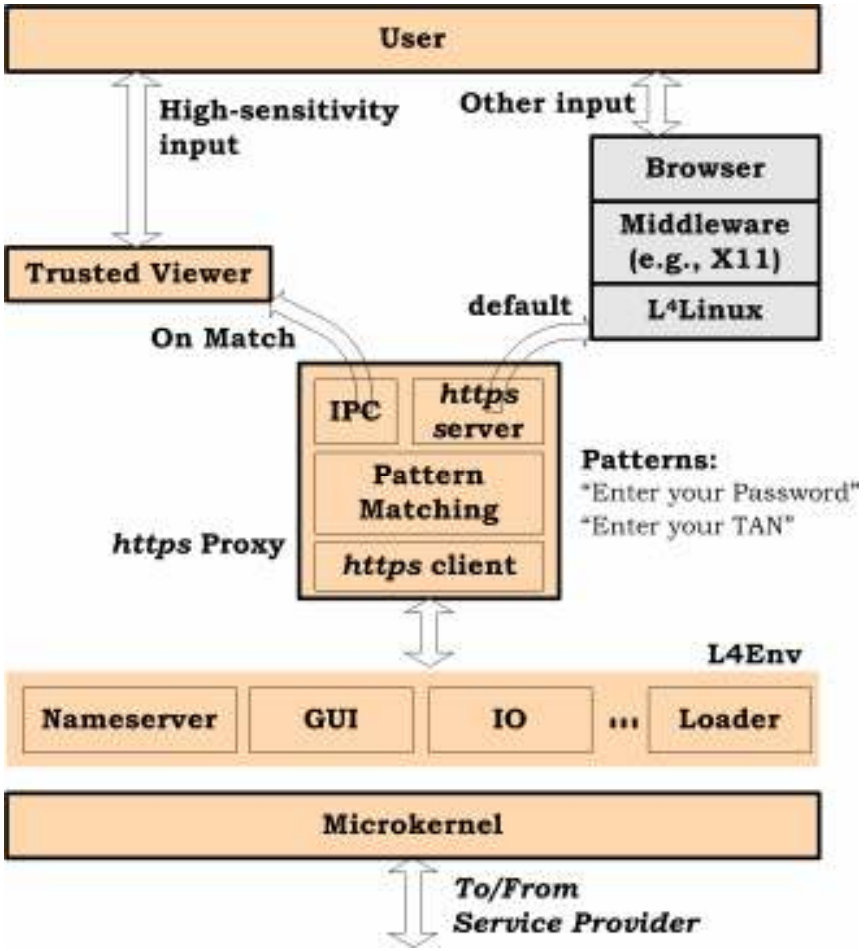
Challenges

- Reusing Legacy Code and Interfaces
  - Minimize modifications to browser
  - Reuse HTTP protocol
- No Explicit labeling of information
- Flexibility in use of AppCore
  - e.g., Use AppCore for high-sensitivity information on home machine, but use AppCore even for low-sensitivity information on public access machines

AppCore

- Use a https proxy to trap all incoming messages
  - Minimal modification to browser (proxy settings)
- Proxy determines sensitiveness of messages
- Sensitive messages forwarded to a small and simple viewer (Trusted Viewer)
- Rest of messages sent to legacy browser
  - Legacy browser runs as untrusted application

System Architecture: BLAC



Inferring Sensitiveness of Information

- Sensitiveness inferred from string patterns
  - Also includes strings that result in generation of sensitive user input
- Patterns can be specified by end-user or web server
  - Flexible use of Trusted Viewer

Implementation

- Implemented on top of the Nizza Security Architecture [Härtig02]
  - L4 microkernel
  - L4Env provides system services such as naming, window manager, device manager
- https proxy and Trusted Viewer execute directly on top of L4 as trusted processes
- Browser executes as untrusted process on top of L^{4}Linux, a paravirtualized legacy OS

Evaluation: Security Properties

- Flow of sensitive information is limited to https Proxy and Trusted Viewer
  - Vulnerable browsers or malicious extensions cannot access sensitive information
- Trusted Viewer and https Proxy are small and simple components
  - Makes exhaustive testing or formal analysis more feasible

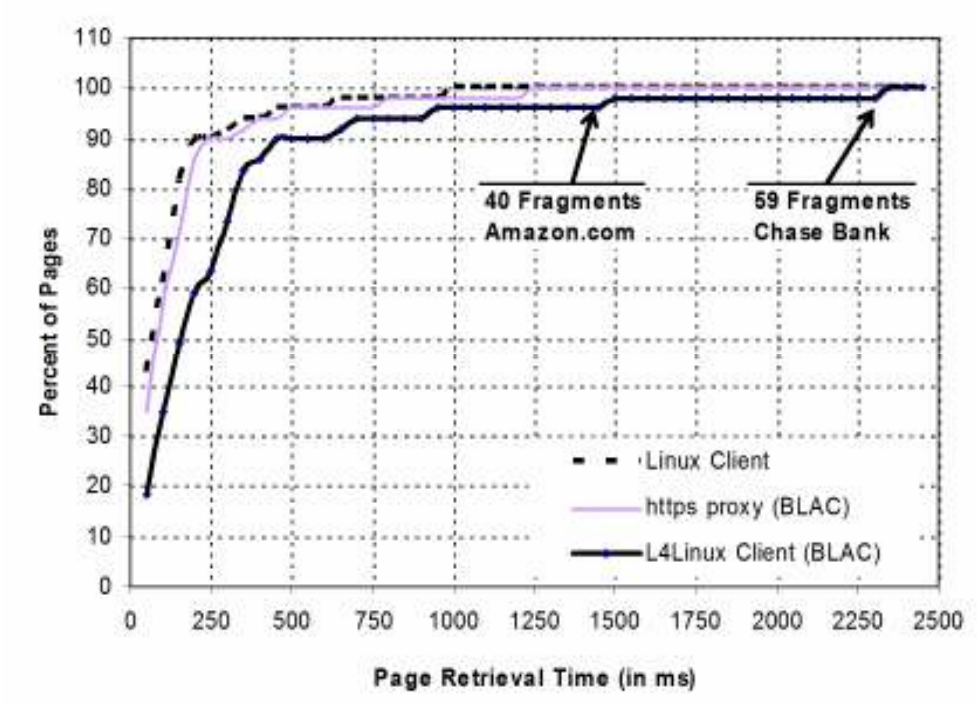
Software Complexity Reductions

Component	BLAC			Linux		
	Composition	LOC	MCC	Composition	LOC	MCC
OS	**L4**	14,000	2,300	**Linux Kernel**	383,000	65,000
Middleware	**L4Env**	86,300	11,300	**X Server**	1,015,000	140,300
https Proxy	**https Proxy**	13,600	1,900	**-**	-	-
Application	**Trusted Viewer**	5,000	290	**Mozilla Firefox**	2,208,000	328,300
Total		**118,900**	**15,790**		**3,606,000**	**533,300**

Performance Evaluation

- Use a trace from 3 banks and Amazon.com
- Sources of Overhead
  - Software executing on virtualized hardware: 5-10% for L4 [Härtig97]
  - https Proxy: Use simple https clients and servers to measure and compare overhead

Page Access Times



- Page access times show 2X slowdown
- Most pages retrieved well within 2 seconds, which satisfies over 75 % of users [Jupiter Research]

Code and Interface Reuse

- BLAC works with unmodified HTTP & SSL protocols
- BLAC reuses browser interface as much as possible
  - User can limit number of pages handled by Trusted Viewer by configuring the https proxy

Discussion

- Real world web servers have multiple complications
  - Non-https login pages, no support for Trusted Computing, convoluted html format
- Server-side support can simplify BLAC
  - e.g., explicit labeling of data sensitivities simplifies proxy

Timeline

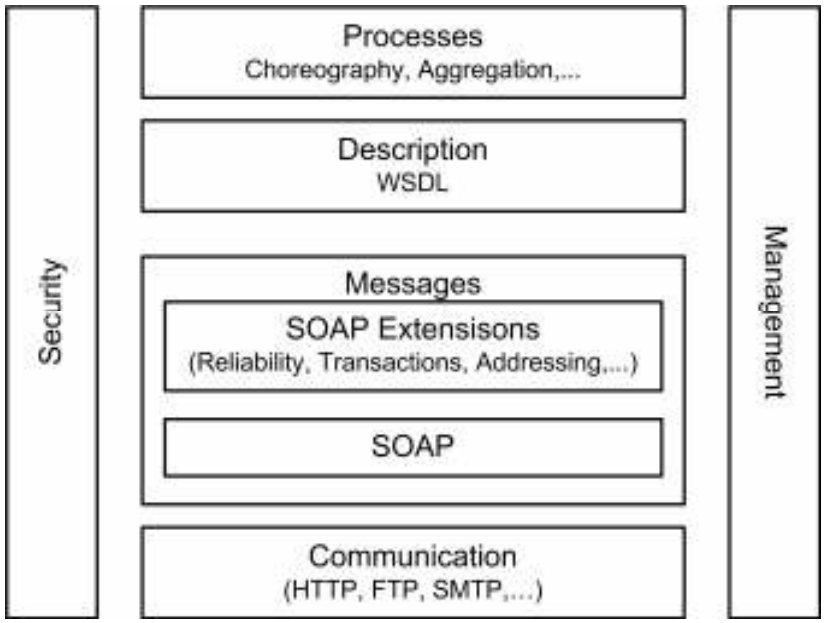
- Problem Statement
- AppCore Approach
  - Client-side AppCore for https-based Applications
  - Server-side AppCore for Web Service Platforms
- WS-FESec for Web Service Compositions
- Related Work
- Conclusion

Web Service Platforms (WSPs)

- Provide Middleware support for Service Oriented Computing
  - e.g., Axis, .NET, WebSphere Application Server
- Increasingly used in security-sensitive services, e.g., PayPal's payment processing web services
- Employ security protocols such as SSL, TLS and WS-Security to protect information flow

W3C's Web Services Architecture

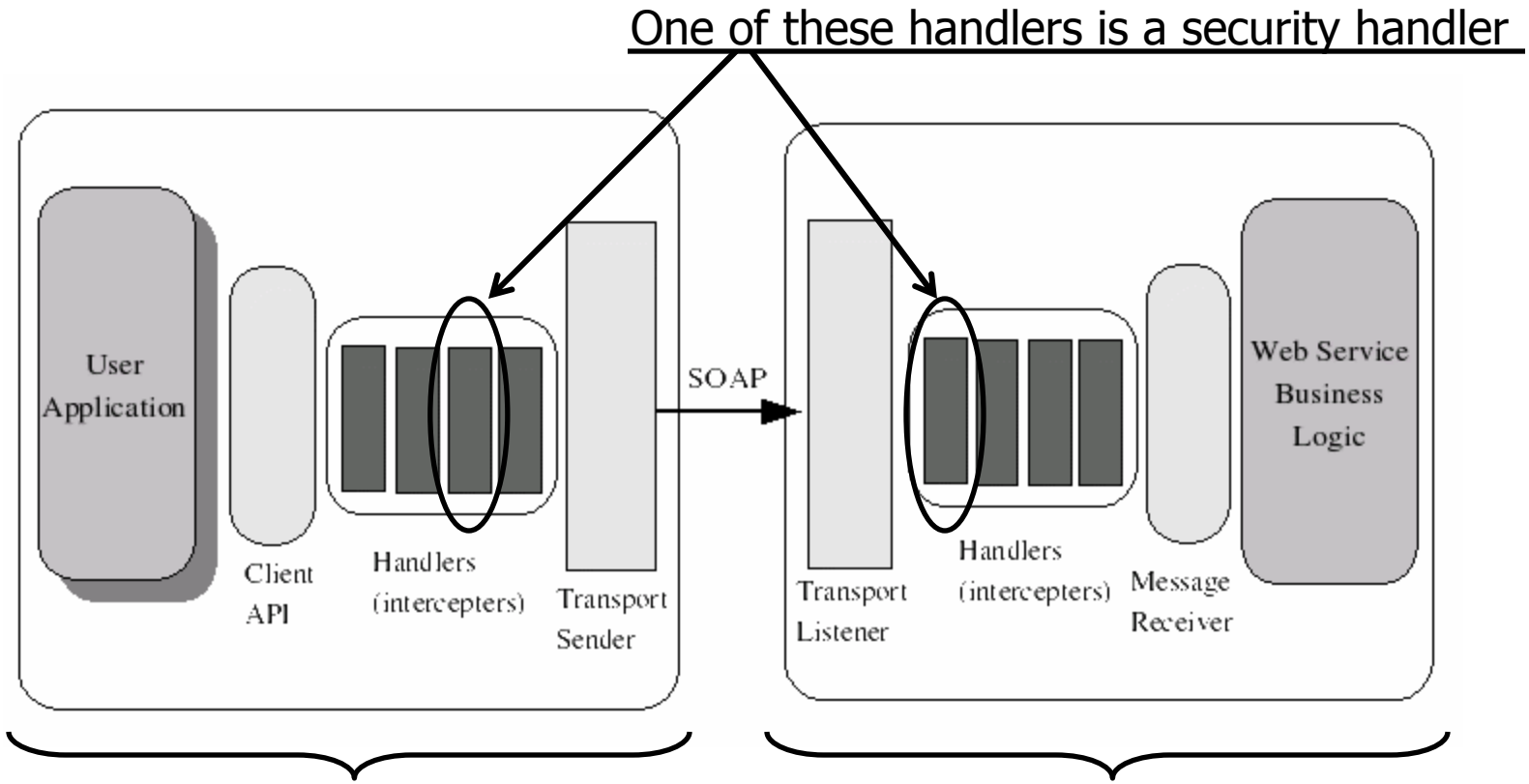
- WSPs implement W3C's web services architecture
- Main components are
  - Communication protocols (HTTP)
  - Message Wrapping (SOAP)
  - WS-* extensions
  - Publish and Discovery mechanisms
- No directions on implementation strategy
  - Security Processing co-located with other types of processing



Axis2 WSP

- Popular Open Source WSP
- Implements a Data-flow model for message processing
- Each Message is assigned to a thread.
- Thread calls appropriate **handlers** in sequence for transport, SOAP, WS-* processing and application code

Information Flow in Axis2



- All Handlers execute in same protection domain and same address space
- Security Handler controlled by configuration file and global variables
  - Both accessible to all handlers

Security Problems in WSPs

- Large and Complex Software, over 110 KLOC
  - Support for extensions and configuration files further complicates analysis and testing
  - Contain multiple security vulnerabilities [Secunia]
- Extensions have access to sensitive data
  - e.g., Indirect access in the Axis2 WSP

PoLP Problem in Web Service Platforms

```
outparam = ctx0.getAxisConfiguration().
                        getParameter("OutflowSecurity");

 if (outparam !=null){
   ome = outparam.getParameterElement();
   itor = ome.getFirstElement().getChildElements();
   while (itor.hasNext()){
     attr = (OMElement) itor.next();
     if("encryptionUser".equals(attr.getLocalName())){
       attr.setText("weak_key");
   }
 }
```

All Extensions have access

Malicious Extensions can disable security processing or specify use of weak keys

Applying AppCore Approach to WSPs

- Identify Trusted Components
- Compose them into AppCore (T-WSP)
  - Modify legacy WSP (U-WSP) to call T-WSP to operate on sensitive data
- Limit flow of sensitive data to T-WSP
- Split application-level code into trusted and untrusted part

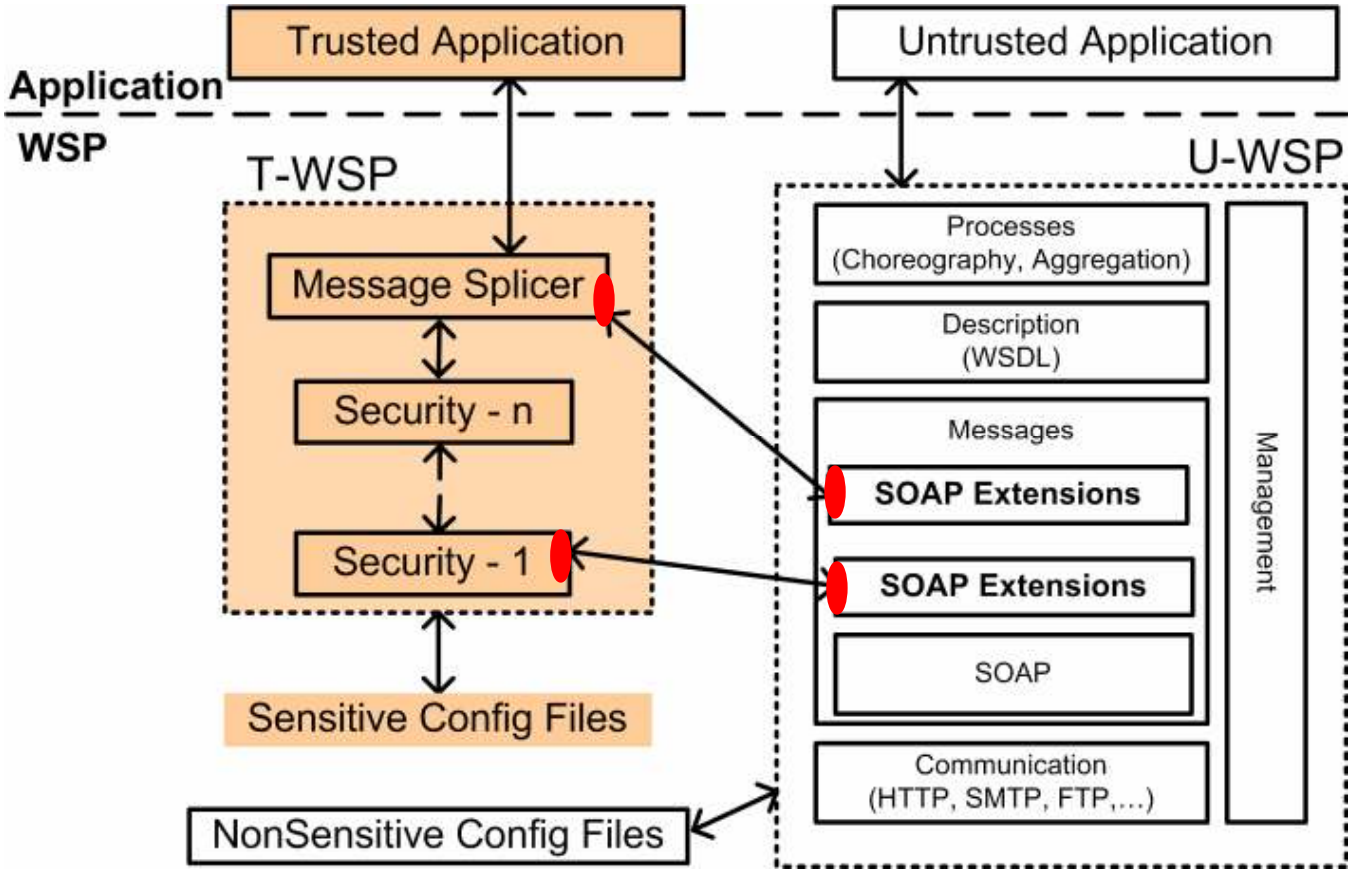
Identifying Trusted Components

- Key Assumption: Sensitive information is protected using WS-Security
  - Note: We do not have to infer sensitiveness of data as in BLAC.
- Rely on W3C specifications and Axis2 source code and documentation to analyze WSP
- Security Related Extensions such as WS-Security, WS-Trust and their config. files are Trusted Components
- Message Splicer for controlling flow of information (explained later)

Untrusted Components

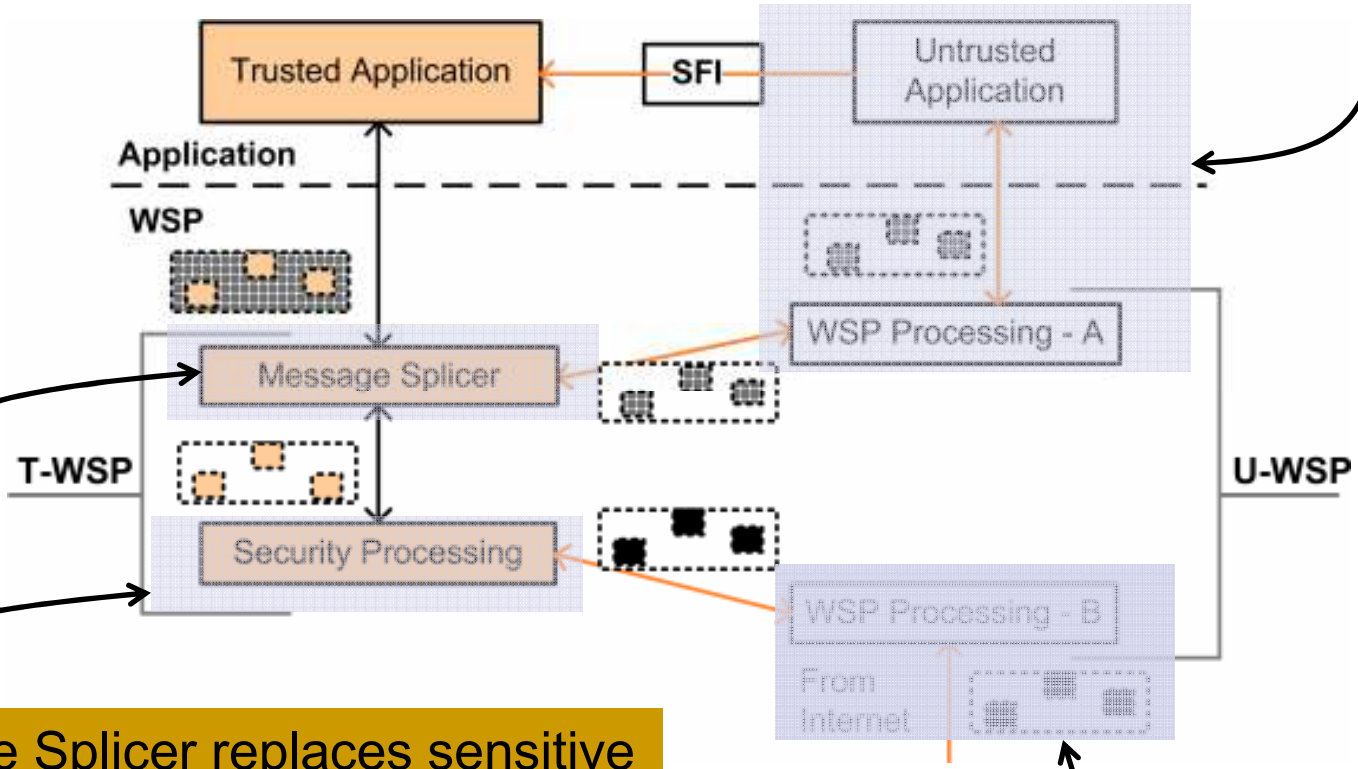
- Components that do not need access to sensitive data
  - We make no assumptions about the properties of these components
- WS-* extensions such as WS-Addressing, WS-ReliableMessaging, WS-ResourceFramework, WS-Coordination, WS-AtomicTransaction etc…
- SOAP processing and transport layer processing
- Untrusted portion of application

ISO-WSP Architecture



Insert RMI Calls. Serialize and Deserialize parameters

Securing Information



Application Support for ISO-WSP

- Information Flow Split into two parts
  - => Application too has to be split
- Example: Payment Processing Web Service

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

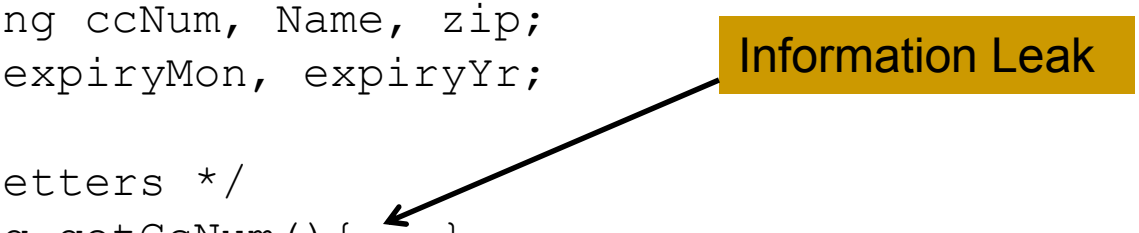
Why Split Applications?

```
public class CreditCard{
   private String ccNum, Name, zip;
   private int expiryMon, expiryYr;

   /* Getters,Setters */
   public String getCcNum(){...}
   public void setCcNum(String num){...}

   /* Validate Card*/
   public boolean validate(){...}

   /*Charge Card and return a Txn ID*/
   public String chargeCard(float amount) {...}
   /* Additional Functions */ ...
}
```



Secure Functional Interface (SFI)

- Interface to Sensitive Objects that is available to untrusted code
  - Provides a restricted view of sensitive objects
- Designed by the developer

Example:

```
/* Classname and Namespace*/
class:=edu.gatech.cc.pp.CreditCard

/* Interface */
interface CCsfi{
   boolean validate();
   String chargeCard(float amount);
}
```

Generate Trusted and Untrusted Code

- Trusted Code handles actual data
- Untrusted Code gets dummy data items + *unique token*
  - Token is a capability to access sensitive data items
- Untrusted Code uses SFI to operate on sensitive data

SFI Example: Untrusted Code

```
public class CreditCardUnt extends CreditCard {
   private String sfiID;
   private CCsfi stub = null;
   public boolean initStub(){…}
   /* override the methods defined in SFI */
   public boolean chargeCard(float amount) {
     if(sfiID != null) {
       initStub();
       stub.validate(sfiID, amount);
     }else{
       super.validate();
     }
   }
}
```

RMI Call. Pass token along with other parameters

Developer Input to Port Apps

- Specify SFIs
  - Generate Trusted and Untrusted Code
- Code to interface Trusted application with T-WSP
- Parameters for Message Splicer
  - Instances of dummy objects, e.g., Credit card with invalid numbers
- Change serializers and deserializers
  - Modify few lines of code
- Input validation code for SFI functions

Implementation Details

- Implement a T-WSP for Apache Axis2
  - Contains a WS-Security Implementation + Message Splicer
- Modify Axis2 to perform RMI for WS-Security processing
  - Serialize and Deserializers for SOAP message
  - ~800 New or Modified LOC
- Implement Payment Processing Service, and port the RUBiS web service
  - Cost of porting discussed later

Payment Processing Service

- ISO-WSP adds 7.6 ms overhead (~19%)
  - 5 ms in the WSP, rest in application
- Application level costs include
  - Deserializing twice – in trusted and untrusted part (~1.5ms)
  - Two RMI calls for charging card and cleaning up state on trusted part (~0.8 ms)

Performance Impact

- Few ms is small compared to hundreds of ms response time of real-world web services [Kim04]
- ISO-WSP only affects flow of sensitive data
- By separation of concerns in interface, impact can be minimized
- E.g., Split interface into authentication interface and Functional Interface
  - Auth interface uses T-WSP, rest use U-WSP
  - T-WSP removed from performance critical path
  - e.g., RUBiS has 6 functions handling sensitive data and 14 handling non-sensitive data

Security Improvements

- Only T-WSP and trusted part of application have access to sensitive data
- Reduces software complexity of WSP by 5X (< 20KLOC to test/verify)
- **Importantly, Most of functionality of legacy WSPs is retained**

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

Cost of Porting

- Port Payment Processor and RUBiS web services
- Majority of porting effort focussed on interface between T-WSP and Application
  - Can be further reduced by using code generators similar to WSDL2Java.sh

Service	SFI	Untrusted Portion	Trusted Portion	Total (% Modified)
Payment Processor	5	3	32	40 (13%)
RUBiS	9	12	28	49 (<1%)

Does not include actual payment processing operations, e.g., charge credit card operations

Code and Interface Reuse

- ISO-WSP reuses legacy WSP for non-sensitive tasks
  - Developers retain access to most functionality
- Message Splicer adds and removes tokens transparently w.r.t remote application
  - No changes to remote interface
- Interface between U-WSP and untrusted application is retained
  - New interface between T-WSP and trusted apps
- Reuse legacy application level code through inheritance

Discussion: Applicability to Other WSPs

- Easy to port to other WSPs
- Naïve mechanism: Source code access not required
  - T-WSP functions as a proxy WSP (similar to https proxy in BLAC)
- Optimized approach: Modify legacy WSP to invoke T-WSP
- Requires:
  - Serialization/Deserialization of SOAP messages
  - Remote Invocation Mechanism

Timeline

- Problem Statement
- AppCore Approach
  - Client-side AppCore for https-based Applications
  - Server-side AppCore for Web Service Platforms
- WS-FESec for Web Service Compositions
- Related Work
- Conclusion

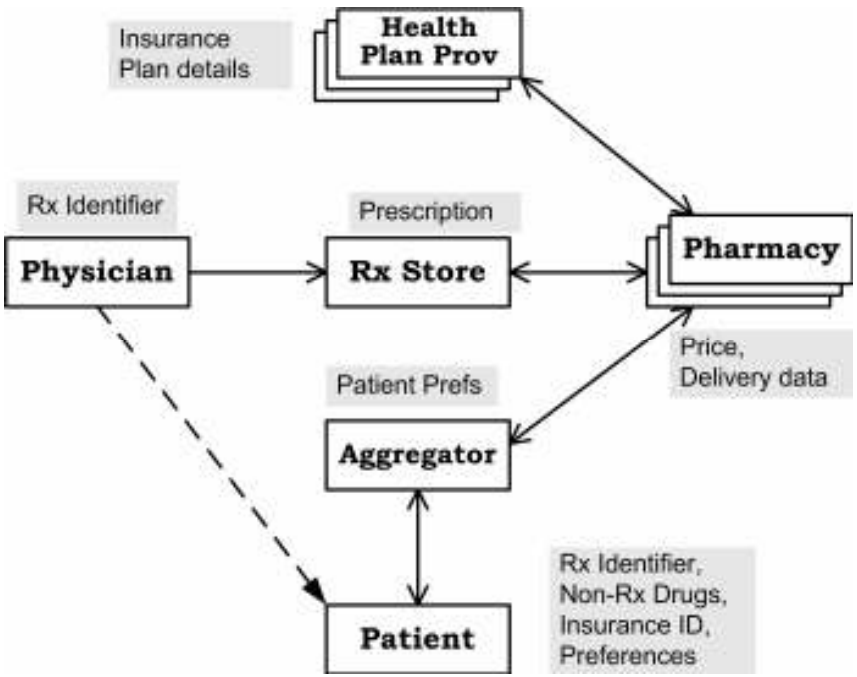
Web Service Compositions

- Terminology: Data producing services, data consumers, and Intermediate services
- Combine multiple services transparently to provide a value added service
  - e.g., overlaying GPS data of bus or apartment listings on top of a Map service, Collecting auction or for sale listings of books from multiple sites, …
- Rising in popularity due to web service interfaces provided by big service providers:
  - e.g., eBay, Google, Yahoo, Amazon, PayPal all provide interesting services
  - Yahoo Pipes: an example of user-driven composition

Security Problems in Compositions

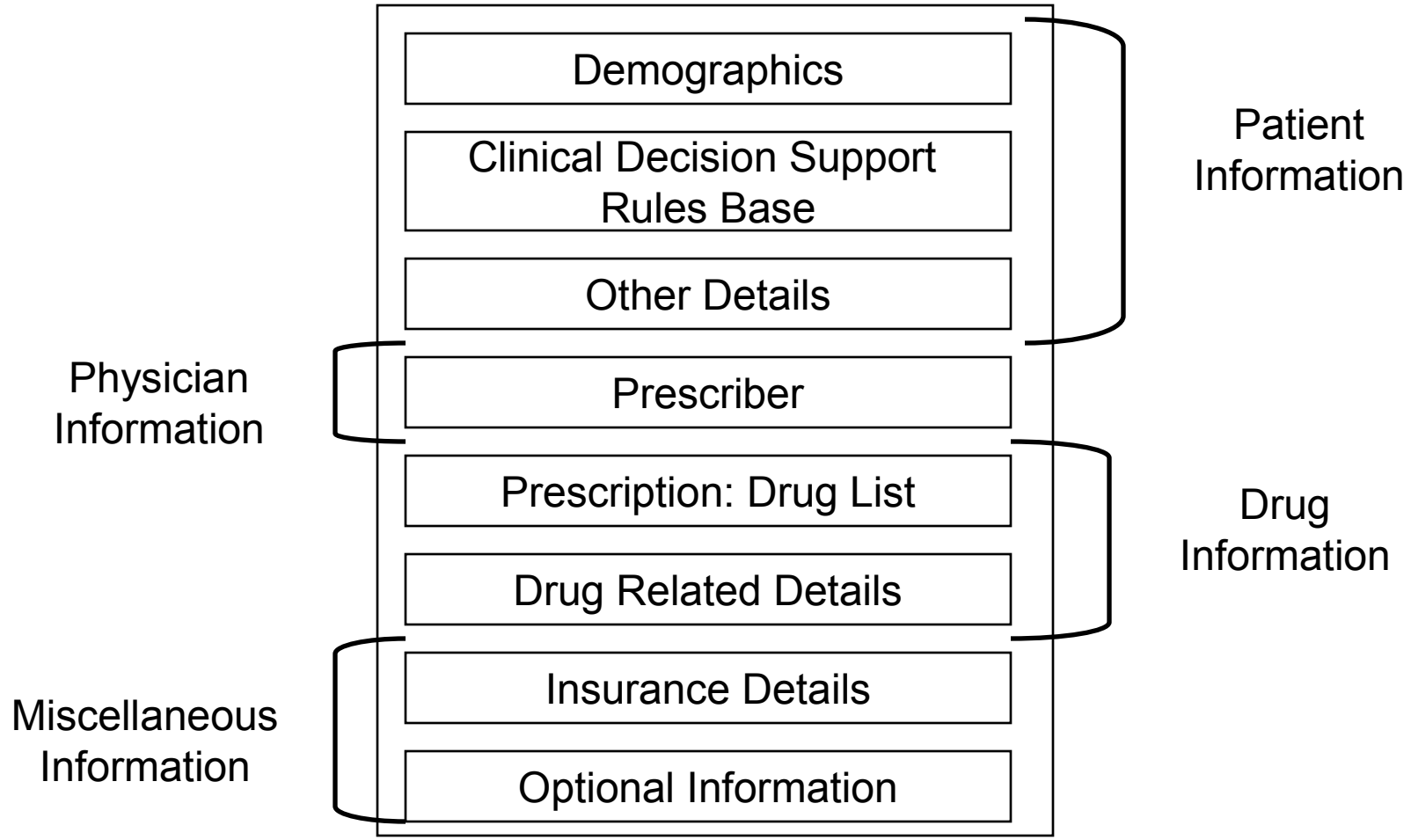
- Traditional Interaction: Consumer talks directly to data producing service
- In Compositions: Consumer talks to intermediate services
  - Intermediate services need read/write access to portions of messages
  - => Consumer now has to trust all intermediate services
- Web services operate in open environment
  - Not possible to trust or even know of all services involved in composition

Example: An Electronic Prescription System (EPS)



- www.pharmacychecker.com, www.nyagrx.com compare price of individual drugs
- EPS prices complete prescription (Rx)
- Adds aggregator service
  - Compares prices, shipping options, etc…

Data Format of Electronic Prescription



Source: eHealth Initiative Executive Summary 2004.

Security Requirements for EPS

- Confidentiality Requirements:
  - Patient and Physician have access to complete Rx
  - Pharmacy that fills the Rx gets access to patient information
- Integrity Requirements:
  - Pharmacies must be able to verify signature on Rx
  - Patient must be able to verify price of Rx as specified by each pharmacy

Usability Requirements

- Pharmacy must be able to look at list of drugs, dosage, etc…
- Aggregator and Pharmacy services must be able to look at coarse-grained patient address information
- Aggregator must be able to look at price of Rx
  - Simple modifications might be allowed: e.g., change number of items desired

Open Environment

- Large number of pharmacies on Internet
  - Patient does not know and might not trust some of them
- Pharmacies do not know of all aggregator services
  - Some aggregator services might be fraudulent, e.g., prefer one pharmacy to another

WS-FESec

- Uses Fine-Grain Signatures and Encryption
  - Requires web service developer input to classify data items
  - Leverages WS-Security for cryptographic operations
- Extends WS-Security specification to better support open environments

Integrity Protection using WS-FESec

- Integrity Groups (IntG): Groups of data items that are relatively independent from rest of message
  - e.g., each item in listing below is independent from rest of items, list of drugs in a Rx
- Developer specifies IntGs for a service
  - Each IntG signed separately
  - => Parts of message can be modified without invalidating the complete message

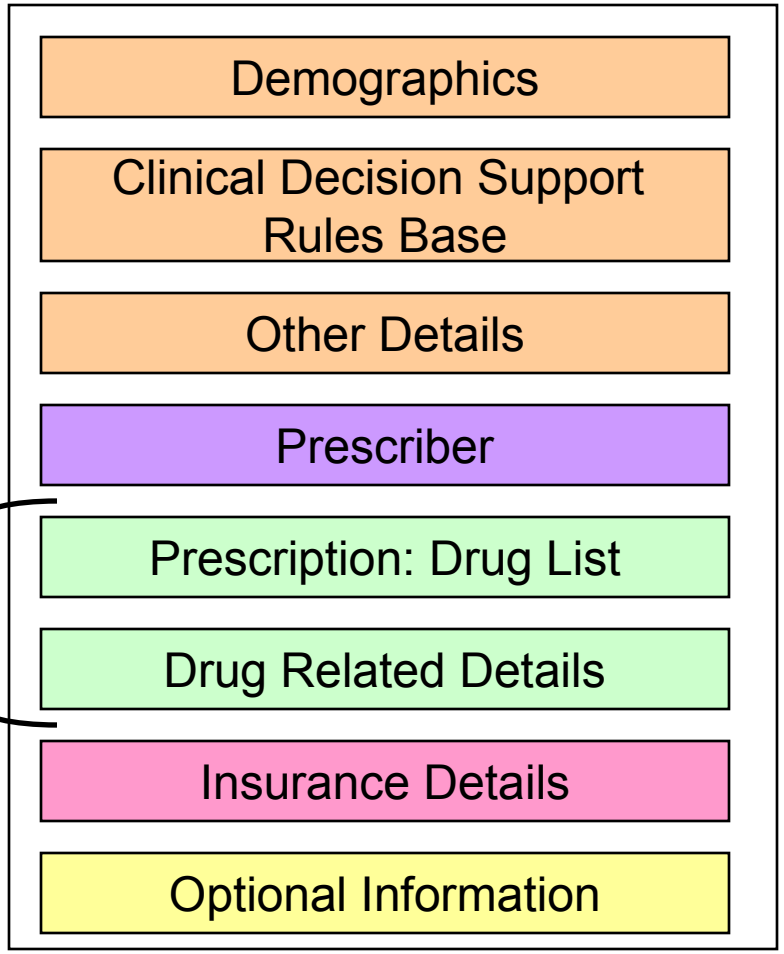
☐		10 Discworld PBs by Terry Pratchett (Eric Soul Music +)		-	**$24.99**	$5.00	1d 04h 08m
☐		NICE LOT OF 7 TERRY PRATCHETT PAPERBACK BOOKS		-	**$18.00**	$4.65	1d 10h 56m
☐		Terry Pratchett DISCWORLD pb's CARPE JUGULUM UK + 4 US		- Buy It Now	**$8.50** $9.90	$4.25	2d 06h 55m
☐		Terry Pratchett COMPLETE(well, almost) DISCWORLD 24 pbs		1	**$39.99**	Not specified	2d 13h 49m
☐		Lot 12 Terry Pratchett Discworld series pb some htf		6	**$16.50**	$6.00	5d 02h 53m
☐		Lords and Ladies NEW Pratchett Terry BOOK		Buy It Now	$2.87	See description	6d 09h 53m

Confidentiality Protection

- Confidentiality Groups (ConfG): Group of items with **same** confidentiality requirements
- i.e., Items that can be seen by same set of service providers
  - e.g., list of drugs can be seen by all pharmacies, however, patient information can be seen only by one pharmacy => 2 separate ConfGs
- Each ConfG encrypted with separate key
  - But key distribution in open environment?

Key Distribution

- Each Color represents a ConfG
  - Drug Information should be available to all pharmacies
- Encrypt each ConfG with a separate key
- WS-Security encode key information in a KeyInfo structure
  - e.g., ConfG key is encrypted with the public key of pharmacies



Key Distribution - II

- Limitations of KeyInfo: Only one KeyInfo per encryption
  - => All recipients of a particular piece of data must share the same secret
- For Drug Information: All Pharmacies must understand the same KeyInfo structure
  - => pharmacies share the private key.
- Not a feasible option in open environment
- Solution: Allow multiple KeyInfo structures

Key Distribution - III

- Insufficient knowledge of Recipients
  - e.g., New online pharmacy unknown at message generation time
- Large number of potential recipients, e.g., thousands on online pharmacies
  - Cannot have KeyInfo for each recipient
- Add CallbackReference to key types
  - Requires recipient to invoke the given URL to get key information and decrypt the message

Callback Reference Example

```
<ds:KeyInfo Id="…" xmlns:ds="…">
   <wsse:SecurrityTokenReference wsu:Id="…"
                      wsse:TokenType=CallbackReference>

        <fesec:CallbackReference
                      URI=http://rxws.com/eps/Auth
                      fesec:AuthMechanism=UsernameToken
                      fesec:MsgID="0xRXID145"/>

  </wsse:SecurityTokenReference>
</ds:KeyInfo>
```

URL to contact to get the key information

Authentication Mechanism to be employed to retrieve the key

Parameters to use to retrieve the key

Evaluation: Security Properties

- Model Web Service compositions as a lattice [Denning76]
- Data items in a message have security classification level (say L_d)
- Services and Clients have classification (say L_{sc})
- Access is allowed only if L_{sc} dominates L_d.
- Challenge: Open environment of web services implies no uniform classification mechanism

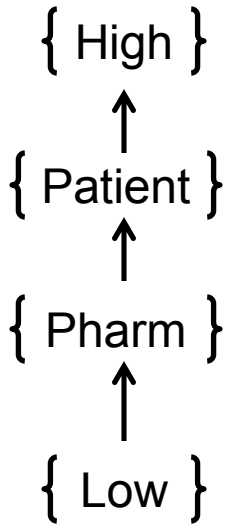
Modeling Compositions as a Lattice

- Introduce two new levels:
  - *Low*: All unprotected data
  - *High*: Imaginary level that dominates all known levels
- Each data producing web service has its own lattice
  - However, all of them share same High and Low levels
- Naïve combination: Attach all Lows and Highs to get lattice
  - Inefficient, but can model web service compositions on the Internet

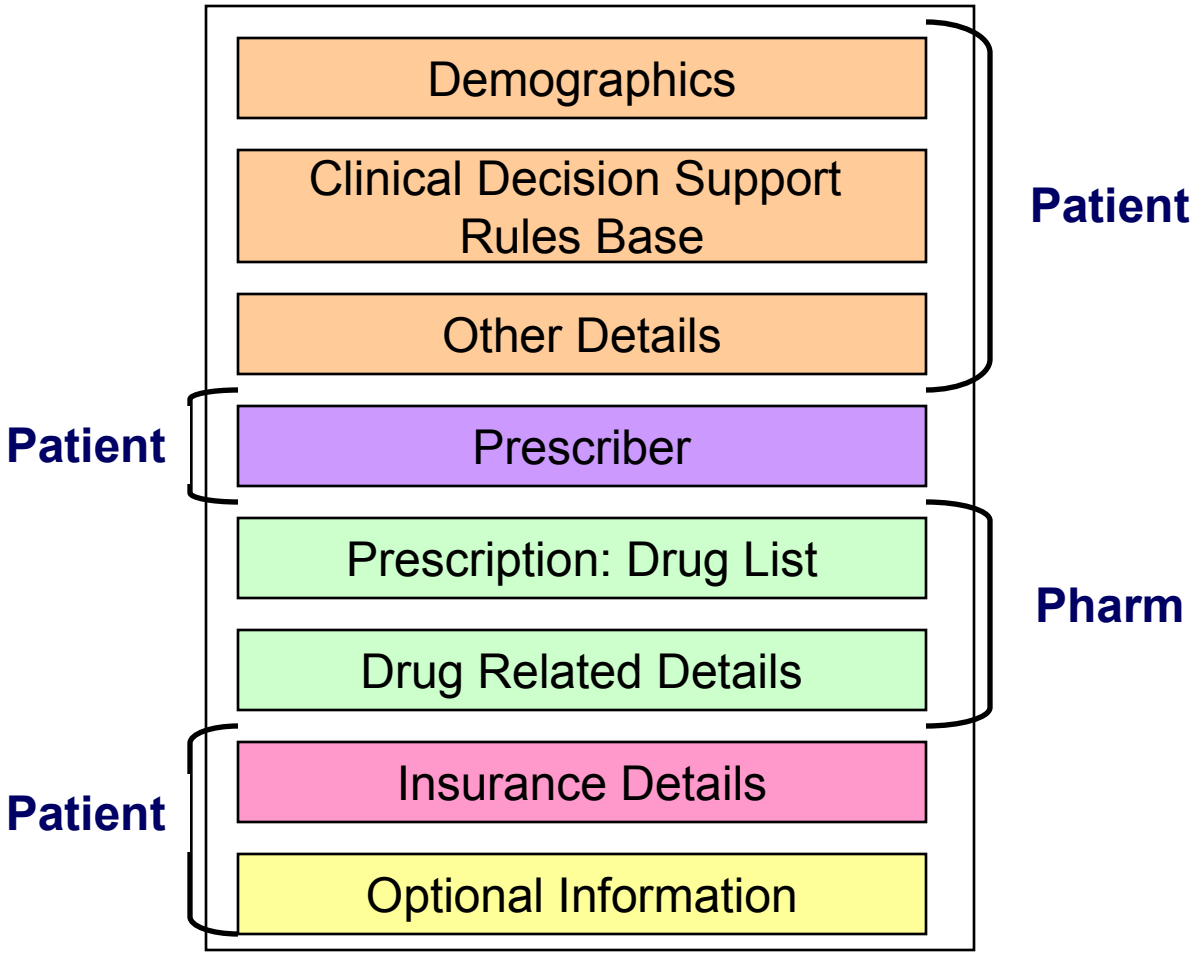
WS-FESec for Lattice

- Each ConfG is a level in a lattice (L_d)
- Recipients possess one or more labels (L_{sc})
- Data producing web service determine labels of recipients
  - e.g., using Authorization mechanisms
- Key distribution in WS-FESec ensures that key is available only if L_{sc} dominates L_d

A Simple Classification System For EPS

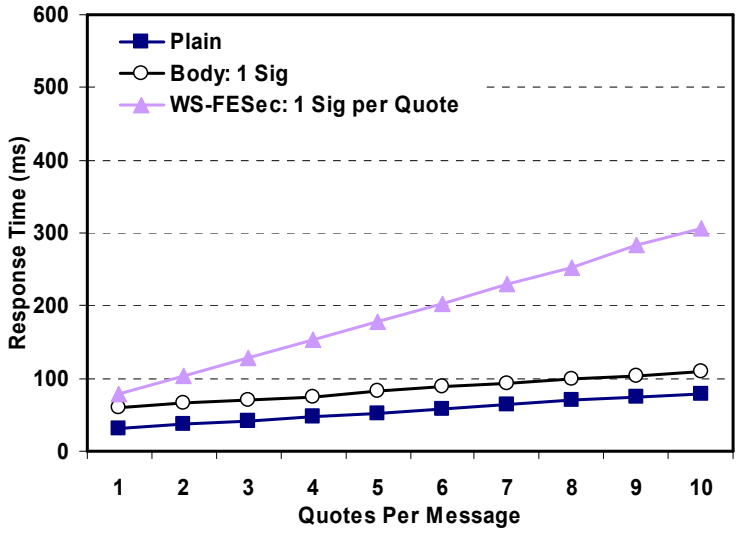


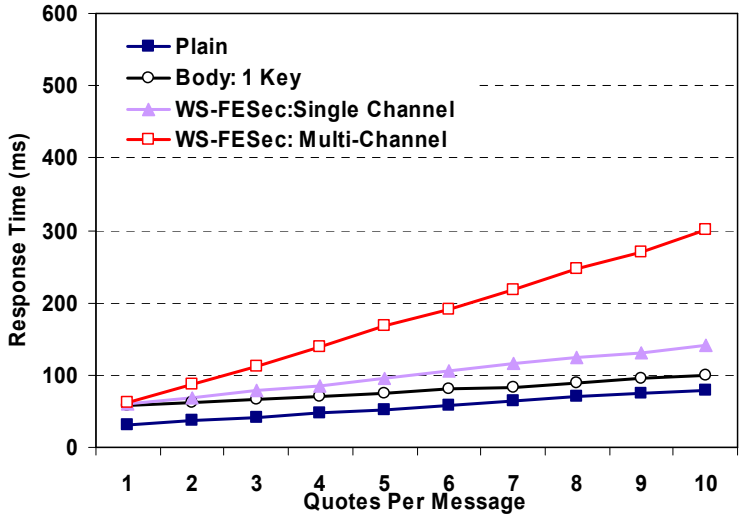
- Initially, Patient & Physician have Patient Label
- Later on, **one** pharmacy gets Patient Label



Performance

- Modified WSS4J library to perform multiple signatures and encryption per message
  - Evaluate using a simple stock quote service
- 20 ms overhead per additional signature, 22 ms per encryption
  - Digital signatures contribute around 15 ms of overhead
- Can be reduced by employing symmetric encryption for KeyInfo, e.g., WS-SecureConversation





Using WS-FESec in Compositions

- Encryption by WS-FESec modifies message format
  - Intermediate services must be capable of working with partially encrypted messages
- WS-FESec can be easily integrated with compositions languages, e.g., BPEL4WS
  - Requires ability to manipulate WS-Security headers

Limitations of WS-FESec

- Fine-grain signatures fail when messages are completely modified
  - e.g., Text to Speech service invalidates all signatures
- WS-FESec does not address confinement, e.g., An authorized web service can release sensitive information
- Trustworthy computing in conjunction with ISO-WSP can provide better guarantees
  - e.g., allowing only a portion of web service access to sensitive information and using Trusted Computing hardware for remote attestation

Timeline

- Problem Statement
- AppCore Approach
  - Client-side AppCore for https-based Applications
  - Server-side AppCore for Web Service Platforms
- WS-FESec for Web Service Compositions
- Related Work
- Conclusion

Related Work

- SSL, TLS, WS-Security provide protection on network
- End-Point Software can be protected in multiple ways
- Protection From Malicious Software:
  - Trusted Computing (TC): Use hardware support to ensure software integrity
  - Integrity Measurement Architecture [Sailer04] extends TC support to include configuration files and runtime extensions
  - WS-Attestation [Yoshihama05] and Trusted Web Service [Song06]
  - Complex software still runs on TC hardware => runtime vulnerabilities compromise system

Related Work - II

- Securing Software from vulnerabilities
  - Defenses against buffer overflow, format string [Lhee03]
  - CCured [Necula02], Incorporating authorization policy enforcement in existing code [Ganapathy07]
  - Address Space Randomization[Bhatkar05], NVariants[Cox06]
  - AppCore reduces code base that needs to be protected: static analysis is more efficient and run time protection is less costly
- Information Flow Restriction: Reduces impact of attacks
  - Capability-based systems – Hydra & Protected Data Paths, MACs - Asbestos, ABAC – Polaris, Language based Information Flow Analysis [Sabelfeld03]
  - Orthogonal to AppCores: Since AppCores have lower functionality requirements, easier to constrain them

Related Work - III

- Refactoring Software
  - Privilege Separation[Provos03], Proxos [TaMin06]
  - Middleware Refactoring for customization [Zhang03]
  - AppCore approach similar, but more emphasis on functional simplification and reusing interfaces
- Browser Defenses
  - VMware's browser appliance, Tahoma [Cox06] use separate VM for browser instances
  - Place functional restrictions on browsers for "complete" sessions: AppCore does not restrict browser behavior

Related Work IV

- Security in Web Service Compositions
  - Reputation-based systems [Yang05], trusted third party-based systems [Carminati05] to gauge trustworthiness
  - Decentralized Orchestration [Chafle05] to control flow of information
  - Security Authorities [Sedukhin03] to enforce end-to-end security
- WS-FESec is designed to handle open environments, specifically to handle uncertainties regarding recipients

Summary

- Presented the AppCore approach to tackle the problem of large and complex software
  - Split software into small trusted (AppCore) and large, legacy untrusted part.
  - AppCore has access to sensitive information and uses legacy part to perform non-sensitive operations
  - Significant reductions in complexity attainable with moderate overheads
- Design and Implementation a client-side AppCore for https-based applications
  - Reuses legacy browser transparently for non-sensitive interactions
  - Works with existing HTTP and SSL protocols

… Summary

- Design and Implementation of a server-side AppCore for web services middleware
  - Security sensitive functionality isolated in T-WSP and trusted part of application
  - Simplify porting of existing applications by proposing SFIs
  - Reuse legacy code for non-sensitive tasks
  - Message Splicer maintains conformity with external interface
- WS-FESec: An end-to-end security framework for web service compositions
  - Specifically designed to work in an open environment
  - WS-FESec supports lattice model of secure information flow
  - Overhead of few tens of milliseconds per signature and encryption

Questions?