An Evidence Management Model for Web Services Behavior

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Murat Gunestas
Master of Science
Atilim University, 2005
Bachelor of Science
The Faculty of Security Science, 1998

Director: Duminda Wijesekera, Assoc. Professor
Department of Computer Science

Fall Semester 2009
George Mason University
Fairfax, VA

DEDICATION

I dedicate this to my greatest loss ever.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to Professor Duminda Wijesekera for his constant guidance and encouragement. He is the one, who is always available and open to new ideas.

My sincere appreciations go to Turkish National Police and Turkish Government for their scholarship that supported me during the research period in the US.

I would like to thank my committee members Professor Edgar Sibley for his well-timed suggestions, Professor David Schum for his deep work on evidence science that render my dissertation more expressive, and Professor Jeremy Allnutt for his comments and cooperative approach.

My sincere thanks go to my whole family for their patience and my aunt for her constant faith and support.

Finally I would like to thank my friends Ahmet Eker, Halim Iltas, and Tugrul Turhal for their hospitality and company since the very first moment I arrived at the town.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

B2B	Business to Business
BI	Business Intelligence
BPEL	Business Process Execution Language
CEGWS	Comprehensive Evidence Generation Web Service
CEP	Complex Event Processing
CSV	Comma Separated Values
CPU	Central Processing Unit
CR	Completeness Rate
DoS	Denial of Service
EDWS	Evidence Derivation Web Service
EGF	Evidence Generation Framework
EOA	Evidence of Availability
EOD	Evidence of Delivery
EOF	Evidence of Failure
EOO	Evidence of Origin
EOV	Evidence of Agreement Violation
FWS	Forensic Web Services
IDS	Intrusion Detection System
IoC	Inversion of Control
JVM	Java Virtual Machine
LFP	Least Fixed Point
LR	Log Record
LRI	Log Record Index
MEI	Message Evidence Index
MEP	Message Exchange Patterns
MTT	Message Type Table
OELP	Optimistic Evidence Layer Protocol
OWASP	Open Web Application Security Project
PKI	Public Key Infrastructure
RPC	Remote Procedure Call
RR	Recruit Rate
SARESA	Sense & Response Service Architecture

SCT	Security Context Token
SEC	Securities Exchange Commission
SELP	Simple Evidence Layer Protocol
SLA	Service Level Agreement
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
ST	Signature Table
STS	Security Token Service
TTP	Trusted Third Party
UML	Unified Modeling Language
WS	Web Services
WS-CDL	Web Services Choreography Description Language
WSCI	Web Service Choreography Interface
WSDL	Web Service Definition Language
WSLA	Web Service Level Agreement
WSRT	Web Service Role Table
WSTT	Web Service Threshold Table
XML	eXtensible Markup Language
XSS	Cross-site Scripting

ABSTRACT

AN EVIDENCE MANAGEMENT MODEL FOR WEB SERVICES BEHAVIOR

Murat Gunestas, PhD

George Mason University, 2009

Dissertation Director: Duminda Wijesekera

Web service choreographies, orchestrations and dynamically invoking web services are three kinds of sample compositions. These compositions create service inter-dependencies that can be misused for monetary or other gains. When a misuse is reported, investigators have to navigate through a collection of web-service or network logs to recreate suspected misuses. In order to facilitate this task, I propose creating forensic web services (FWS), specialized web services that, when used, would securely maintain transactional records between other web services. An independent agency can re-link these secure records residing in distributed FWS stations to reproduce the transactional history, and thereby substantiate or refute claims of misuse by providing supporting or refuting evidence.

As multi-participant transactions migrate to web services, there is a potential for some of these parties to not fulfill their specified obligations or to work to achieve objectives

contrary to those specified objectives. Preserving evidence of service behavior of all participating actors in complex web-based transactions can resolve such shortcomings. In order to achieve this, I propose a three-layered framework to preserve evidence of service behaviors in a non-refutable way. The lowest layer of my framework preserves transactional evidence of pair-wise participation using cryptographically secured FWS. The second layer uses this pair-wise evidence to derive evidence of complex interactions. The highest layer generates evidence of complex transactional behavior.

Web service choreographies can be misused at multiple levels: namely exploiting their technical capabilities that I refer to as *Service Misuses* and using them to design complex illegal business schemes that I refer to as *Business Misuses*, such as Ponzi, pyramid, or money laundering schemes. One of the main problems with the latter kind of misuses is that they appear similar to a legal multi-stage business scheme to an external observer with a microscopic view; but in truth are macroscopically illegal. I define some of these schemes precisely and show how to produce evidence of them using cryptographically secure local message repositories. Such evidence would be helpful to financial fraud investigators, business arbiters, potential investors, and judicial actors.

Detecting service or business misuses, in particular, over a set of evidence of observed web service interactions through a post-mortem investigation might disclose an extremely dramatic level of damage as is in the case of Ponzi schemes. Live detection of business misuses can assist a collection of services by alerting them to a spreading misuse that

may target them or help in preventing service misuses. I abstract post-mortem detection queries for business and service misuses.

CHAPTER 1

INTRODUCTION

1.1. Problem Statement

Interdependencies among web services are arising ubiquitously and consequently any misbehaving service affects other services. In such an environment, unless precautions are taken, the opportunity to provide accountability is reduced. Consequently, any service level log records stored at a disadvantaged and grieving service would provide little or no value in identifying the cause of grief. Holding the whole collection of interacting services accountable would be one method to address this inadequacy, but this would not address global business misuses that can be built from pairs of legal interactions. As will be shown during the course of my dissertation, Pyramidal and Ponzi schemes [52, 53, and 54] are two cases in point.

1.2. Thesis Statement

My thesis statement is that *it is possible to construct a forensically sound evidence management framework to accurately account for the global behaviors of composed web services in a secure, participant-neutral, and non-refutable way as a web service itself.* I

propose a web-service framework to do so, and to decompose the demonstration of its viability into four sub parts:

1. The first is to design a prototype implementation of a suitable framework. My framework constructs a web service that I refer to as Forensic Web Services (FWS) that generate evidence at service invocation times from composed web services; and, using the distributed approach; they can collect those evidences to provide a comprehensive evidence of the externally observable behavior of the complete compositions.
2. The second is to upgrade the FWS into a three-layered framework, called the Evidence Generation Framework (EGF), which includes FWS at the bottom layer for pair-wise evidence generation, Evidence Derivation Web Service (EDWS) at the middle, and Comprehensive Evidence Generation Web Service (CEGWS) at the top; and to design a modular agent for endpoints capable of communicating with other components of the proposed framework. In EGF, using runtime interception, the bottom layer can record messages exchanged among parties, check session invariants, and verify the existence of signatures on the fly without polluting the business logic.
3. The third is, based on evidences generated at the bottom layer, to design the CEGWS at the top along with web service choreography pattern and business misuse mining. Unlike the distributed approach followed in EGF, I designed a central approach for collecting evidences to recreate composed activities. That is, FWS stations as

proposed earlier are forced to push externally observed message evidences to CEGWSs at service invocation times.

4. Finally, to detect, prevent and alert ongoing choreography misuses, I enhanced EDWS and CEGWS to generate online messages indicating the misuse. In this respect, I designed more abstract patterns for business and service misuses along with pattern directed queries based on those patterns.

1.3. Significance of Contributions

Digital forensics seeks legal evidence on computer/information systems. Digital forensic examinations are performed using specific methodologies in accordance with the digital environment and type of evidence under search. For example, a volatile medium (e.g. memory) examination would have different priorities from a forensic examination on databases. Forensics may address many needs, including but not limited to gathering evidence for legal cases, data recovery, debugging and performance. In summary, relevant information should be collected answering three questions; "What is the nature of incident?", "Who is/are involved?", and "When did it happen?" To answer these questions, digital forensics attempts to discover the current state of the digital artifact, which can be a database, a log file, a floppy disk or a mobile device [1].

Unlike traditional forensics implementations, applying forensics to web service infrastructures introduces novel problems such as need for *neutrality* and

comprehensiveness. The *reliability* issue, conversely, has always been a requirement for all forensics examinations.

Web services are owned by organizations; that is, they have equal rights in the court of law when any dispute between parties turns into a law suit. Any log records residing at one party's site would have no forensic value under these circumstances because records could have been altered to favor the owner. Redundancy of evidences would also raise as an issue for such cases, thus diminishing the credibility of evidences [2]. Many forensics investigations conducted through traditional systems have been held based on one site's records. For traditional systems, these actions may be thought of as reasonable because investigators take advantage of querying users and establishing human factors to corroborate digital evidence. In service oriented architectures (SOA), both sites in a dispute would be automated and retain their own records. Both records would be under question by the opponent party, thus showing the need to have a *neutral* third party capturing and preserving evidence between interacting parties.

As described earlier, web service compositions may span over many web services owned by many organizations. Such interdependent services create long, interdependent information flows. Thus, malicious data may stream over many web services. From the forensics perspective, besides neutrality, the evidence gathered should be *comprehensive* enough so that investigation can reach every related end point web service in order to reveal the actions performed by every party to the transaction. If not, incomplete

evidence may point to non-malicious web service nodes as the source of malice, thus misleading the investigators through the examination. This comprehensive approach also helps in converging the evidences as Schum [2] introduces as a force of evidence.

Yet another important principle that any evidence should possess is *reliability*. In a court of law, judicial fellows want to be convinced of the evidence, especially when it comes from a digital source. Because impersonation and replay attacks do occur in web services, cryptographic mechanisms would help in protecting the creator of information passed around in messages by signing them digitally. Such a requirement would entail web services relying on a state-of-the-art cryptography platform such as Public Key Infrastructure (PKI), which, to my opinion, meets the credibility property of evidence introduced by Schum [2].

1.4. Summary of Contributions

I extended Herzberg's [7] evidences layer concept that addresses neutrality and reliability principles on evidences to service oriented architectures, developing the concept to three-layered evidence generation framework.

I added a new layer to the existing web services stack that can operate non-repudiation protocols and endpoint agents running this layer as a proof of concept.

I extended the framework to detect/generate comprehensive evidences for web service choreography use cases and business misuses in the case of Ponzi/Pyramidal illegal business schemes.

I extended the framework to detect misuses online at service level and business level of composed web services. This extension can prevent service misuses and alert relevant web services against business misuses.

1.5. Limitations of the Dissertation

I have designed and described essential parts of the framework through the dissertation. My framework, however, has yet to be fully implemented. Three major factors can impact such an implementation. The first is that, although I have designed those critical parts ready for a prospective scalable design which can distribute over diverse systems, such as, implementing appropriate trust delegation specifications, I have not addressed their *scalability*. The second is because I propose signatures and encryption based on PKI, *cryptographic overhead* may add unacceptably large computation time. Although I have pointed out some practical solutions, such as implementing secure conversations which can alleviate delays and computation overheads, I have not considered the performance degradation arising out of my design decisions. The final concern is potential *storage overhead*. Once again, this is considered out of scope of my dissertation.

1.6. Organization of the Dissertation

Chapter 2 provides a brief background regarding service-oriented architectures particularly in the case of web services. Instead of including a separate literature review chapter, the "Related Work" sections of each chapter discuss and compare works relevant to my dissertation in the context of each self-contained chapter. Because chapters 3, 4, 5, and 6 are self-contained they detail their own problem statement as well. As an introductory work, Chapter 3 describes a single-layered distributed approach for generating and collecting evidences of web services behavior and shows its promise through a case study. Conversely, Chapter 4 proposes a three-layered centralized approach to detecting web service misuses and describes how business logic at endpoint services can integrate with the bottom layer without altering in existing code. Categorizing web service misuses into business and service level, Chapter 5 describes how business misuses can be mined at the top level out of evidences previously generated at the bottom level and stored at a central repository. Extending the work in previous chapters, Chapter 6 describes an online detection for service that can detect service misuses and alert about ongoing and business misuses. Chapter 7 reports my experiment results of accuracy and performance tests. Finally, Chapter 8 concludes the dissertation summarizing my contributions and discussing possible future research areas.

CHAPTER 2

BACKGROUND

2.1. Introduction

Two conceptual elements lie at the basis of the current web services: (1) Use of XML (eXtensible Markup Language), SOAP (Simple Object Access Protocol) [3], and WSDL (Web Service Definition Language) [4] as basic building material; and (2) Complex applications built upon long-running, sometimes transactional executions created from basic elements using choreography, orchestration and compositional methods.

2.2. Basic Paradigm

XML format underlies the entire web service architecture and its artifacts. All schemas, definition files, and messages transmitted are formed with XML. WSDL, a XML based definition file, defines the interface of a web service in order for the service to be invoked by other services in accordance with the specifications of internal executions. SOAP, a XML based protocol, defines the metadata of the messages to be exchanged between services. Operations are defined in WSDL documents and they are the only mechanisms that can be employed for web services to communicate with each other. SOAP messages

are defined and exchanged as incoming and outgoing messages through the operations. WSDL proposes four types of operations:

Notification: One message is sent to many receivers, such as broadcasting.

One-Way: The message is sent and no response is expected, such as Fire-and-Forget.

Request-Response: A typical RPC (Remote Procedure Call) structure: The message is sent from sender to receiver and response is pushed back to the sender.

Solicit-Response: Request is sent without any data and the response is expected.

Although there are four proposed operation types, the message exchanges can be defined in two ways, in summary, One-Way and Request-Response—this is so because notification and response-solicitation can both be represented by one-way and request-response messages, respectively.

2.3. Composition Paradigm

The message exchange patterns (MEP) described above form the basis for the entire web service paradigm. These simple MEPs construct collaboration scenarios using the appropriate composition models. Two issues matter in defining a composition: (1) The specification of the individual services (2), and the pattern of collaboration.

2.3.1. Design Types

Selecting the target provider services can be accomplished either statically or dynamically, that is, at design-time or run-time. Design-time selections entail *a priori* determination while run-time selections can introduce the opportunity to switch between web services among those that provide the same service.

Static Composition: Static compositions propose web services to be selected and determined through the business applications at design-time. Currently, most web service implementations are static. A designer makes the selection manually based on description files (WSDL) published on the web. The designed application logic is deployed into either a business process engine as a process file or into any web service container in hard code. Unless any changes are applied to the logic, the web services specified in the application never change.

Dynamic Composition: Unlike static composition, a designer specifies a class of web services using their exposed properties rather than selecting a particular collection of web services. The logic itself selects specific web services at run-time by asking any filter database residing at the site of the consumer or global Quality of Services provider residing on the Internet.

Static web service composition introduces less anonymity than the dynamic counterpart, thus taking less effort in forensic examination. Because dynamic composition imposes more burden in terms of revealing service activities and its actual performance at run-time, it increases the need to have a comprehensive platform that preserves evidence of activities that occurs through an orchestration.

2.3.2. Patterns

Some authors [5] categorize web service composition from another perspective, that is, its patterns. According to these patterns, web services can be composed either of their typical pattern (hierarchical) or of a little more complex one (conversational).

Hierarchical Composition: Through this pattern, the consumer web service calls another composite web service, passing the input parameter and receiving the result. Other than this request-response activity no other call is employed to the same instance at the target. The complexity of the composition is hidden in this pattern because the target system never allows changing its internal state other than using atomic calls.

Conversational Composition: This pattern is mostly used when web services need to interact with each other more than once in order to execute the same complex transaction. In these scenarios, the target system, unavoidably, makes its internal state mutable, thereby causing overlapping instances to be created within parties to the composition.

From the forensics point of view, representing and recreating the activities in the latter pattern is more difficult than the former. Figure 2.1 illustrates the comparison between the two patterns. In the hierarchical pattern, the nested instance of an external web service completely finishes before returning the result while many interactions between instances can survive in the conversational pattern. Although describing what happened exactly during execution in the hierarchical pattern is reasonable, this may not be the case with conversational patterns.

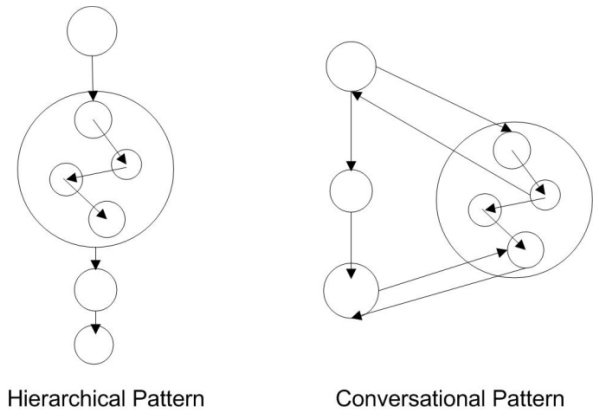


Figure 2.1. Hierarchical and Conversational Patterns (Adapted from [5]).

2.3.3. Composition Standards and Languages

Although there are many standards and specifications for web services, here, state-of-the-art orchestration and choreography specifications are discussed specifically. BPEL (Business Process Execution Language) is a language for business process modeling. WS-BPEL and BPEL4WS are its two popular implementations for web service architecture. They can define both abstract and executable processes. They are two tools

to realize orchestration of composite web services from a centralized service. Conversely, WSCI (Web Service Choreography Interface) and WS-CDL (Web Services Choreography Description Language) create a global view of multi-party choreographies of web services from their individual description files. These languages enable collaborative processes that are recruiting multiple web services, and facilitate interactions between them from a global, high-level perspective rather than an individual service's request-respond perspective.

CHAPTER 3

FORENSIC WEB SERVICES: DISTRIBUTED APPROACH

3.1. Introduction

Web services are being used for many financial, government and military purposes. Their application is performed through seamlessly integrating web services of different organizations over the Internet using choreographies, orchestrations, dynamic invocations, brokering etc. These service-level compositional techniques create complex dependencies between web services belonging to different organizations and can be exploited. When exploited, they can affect multiple servers and organizations, resulting in financial loss or infrastructural damage. Investigating such incidents would require that dependencies between service invocations be retained in a neutral and secure way so that the alleged activity can be recreated in an undeniable way while preserving evidence that could lead to and support appropriate prosecutorial activity. Material evidence currently extractable from web servers such as log records, XML firewall alerts from end point services, and the like, do not have forensic value because defendants can rightfully claim that they did not send that message, and plaintiffs can fabricate or alter the log record to deceive the court. In order to facilitate and base such investigations on reliable

infrastructure that can convince judicial systems, I propose designing *Forensic Web Services (FWS)* that preserve appropriate evidence to recreate the composed web service invocations independent of the parties with a vested interest. This would have a greater chance of being accepted in a court of law. A non-repudiation argument with log entries collected from many web servers has no forensic value. Forensics on web services could never be treated as a bilateral problem between two web services while there are so many standards and architectures composing multiple services and generating global activities.

Consequently, FWS provide on-line forensic capabilities to other web services as a web service itself. To utilize them, FWS need to be integrated with web services that require them – referred to as customer web services of FWS. In order to do so, FWS provide a centralized service access point to its customer services. This information retained by FWS acting as a trusted third party can be directly provided to forensic examiners. Previous proposals to monitor web services [6] and generating evidence [7, 8, 9] have been for business purposes, and to the best of my knowledge I am unaware of their usage in forensic examinations.

Organizations that are tightly integrated with each other through web transactions and processes can benefit from FWS in many ways. Firstly, organizations need to hold their partner services accountable when their vulnerabilities affect transactional confidentiality, availability, etc. Secondly, details of malicious activity may impact the severity of punishments or collectible monetary compensation. I show that undeniable

logging of critical information exchanges are an effective way to meet these two needs. Although not for forensics purposes, some logging and processing approaches exist for web services [10, 11], such as WSLogA [6]. Work reported in [12] offers an approach for online investigations for traditional digital forensic processes. However, none of them employs a trusted third party to generate and preserve evidence and a framework, as well as generate conclusive evidence as provided by the FWS framework.

The rest of the chapter is written as follows. Section 3.2 describes some web-service exploits [13, 14, and 15], of which I use one as a case study. Section 3.3 describes the structure and functionality of the Forensic Web Service Framework. Section 3.4 describes the FWS logging that occurs during service invocations. Section 3.5 describes how an alleged transaction can be recreated in order to determine the guilty party. Section 3.6 illustrates the work described in Section 3.5 through a case study. Section 3.7 describes related work and Section 3.8 concludes the chapter.

3.2. Overview of Web Service Attacks

There are many attacks on web services, such as WSDL/UDDI scanning, parameter tampering, replays, XML rewriting, man-in-the-middle, eavesdropping, routing detours [15, 16, 17, 18, 19, 20, 21], and so on. In addition to web service attacks classified in [13, 14], dynamic service selection, choreography, orchestration, and composition increase the ways of exploiting web services, such as application and dataflow attacks [22,23,24].

Now I show the details of a sample cross-site scripting (XSS) attack used to illustrate the capabilities of FWS. A typical XSS [25] attack may inject a malicious script to harm a web service that dynamically builds some of its information.

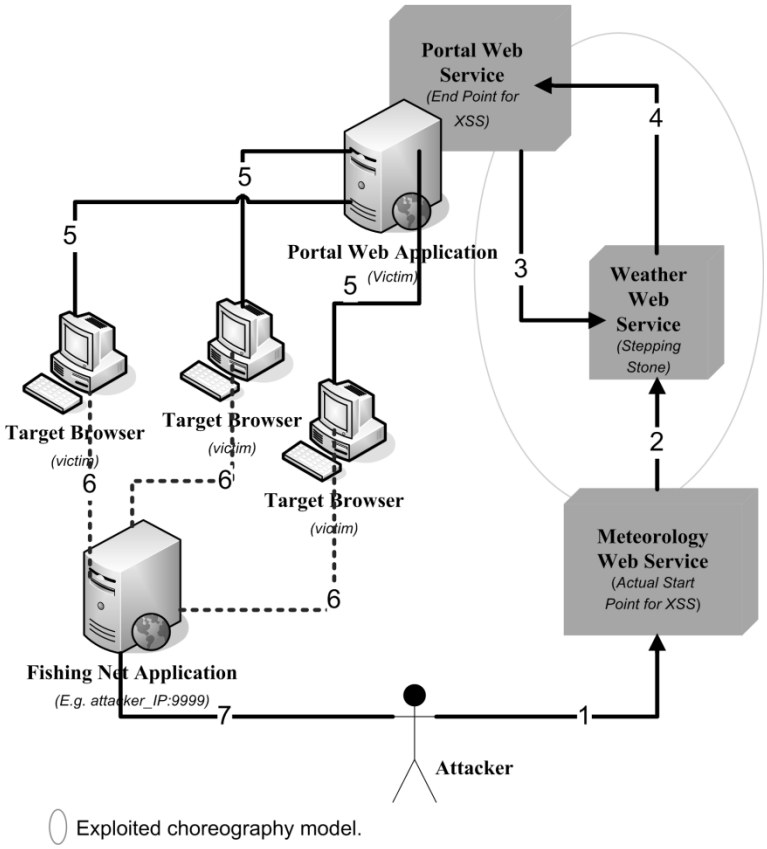


Figure 3.1. A Cross-Site Scripting (XSS) Attack Using Web Services

1. Attacker updates Meteorology Web Service (MET_WS) database with "regionID=234;description="Heavy Rain **<script>document.write('<img src=http://attacker_IP:9999?c='+ document.cookie + ' >'); </script>**""

2. According to the Choreography model, MET_WS fires regional messages to update Weather Web Service (WEA_WS) updateRegion (..ID=”234";Description=”..+mal-script+..”)
3. Portal Web Service (POR_WS) sends weatherRequest(ID=”234")
4. WEA_WS sends weatherRespond(ID=”234";Description=”..+mal-script+..”)
5. Portal Web Application emits the mal-script in html form to requesting browsers.
6. Vulnerable browsers run the mal-script and send cookie information to Attacker’s Fish Net Application.
7. Attacker retrieves sensitive information from cookies.

Figure 3.1 shows an attacker with stolen credentials injecting some malicious data and invoking an update operation on a meteorology service that stores this script (including instructions to steal cookies from web browsers). MET_WS gets infected with malicious data and delivers the data ignorantly to the WEA_WS, firing the updateRegion message. WEA_WS, accordingly to their choreography, passes malicious data to POR_WS, among other legal information. Then a web application, say *Portal Web Application*, invoking a weatherRequest operation at WEA_WS retrieves this malicious data and publishes the weather information to its subscribers in an html form, thereby making the subscribers download the mal-script and send their personal information stored in cookies to the attacker’s *Fishing Net Application*. Consequently, a *Fishing Net Application* managed by the *Attacker* can obtain sensitive user information as shown in Figure 3.1. An attacker, aware of choreography among web services, exploits this model and has Portal Web

Application delivered malicious data to its members using web services in this choreography model.

The stated XSS attack shows how the business logic of a web service can be used to attack a server that depends upon other web services. In this scenario, *Portal Web Service* can claim that *Weather Web Service* sent the malicious content, whereas the actual source was *Meteorology Web Service*. This illustrates the need to have a mechanism that irrefutably points to the source of malice.

3.3. The Forensic Web Service Framework

The Forensic Web Service Framework provides two essential services:

1. **Pair-wise evidence generation:** Collect transactional evidence of transactions that occur between pairs of services at service invocation times.
2. **Comprehensive evidence generation:** On demand, compose pairs of evidences collected at services invocation times, and produce complex transactional scenarios that occurred during specified periods, and provide them for forensic examiners.

In order to do so, FWS use *Trusted Third Parties (TTP)* that sits in between any two transactions. To obtain the services of a FWS system, all web servers sign-up with a forensic web service, as shown in Figure 3.2. In order to create comprehensive evidence of an attack scenario, all relevant FWS agents must cooperate by providing relevant pair-wise transactional evidences that are stored with them. To locate registered FWS servers,

there is a FWS registry of all FWS servers. Figure 3.2 illustrates typical message flows earning forensics capabilities to web services. Ellipse boxes refer to the member domain of any FWS. Every web service registered to any FWS utilizes its evidence modules to route its messages over FWS stations to reach their ultimate goals (dashed lines); every FWS can call each other's services through some investigation algorithms such as "collectDependents" (solid bold lines). Some central services for registry and security purposes, for example, would inevitably be called through the framework at any time (solid lines).

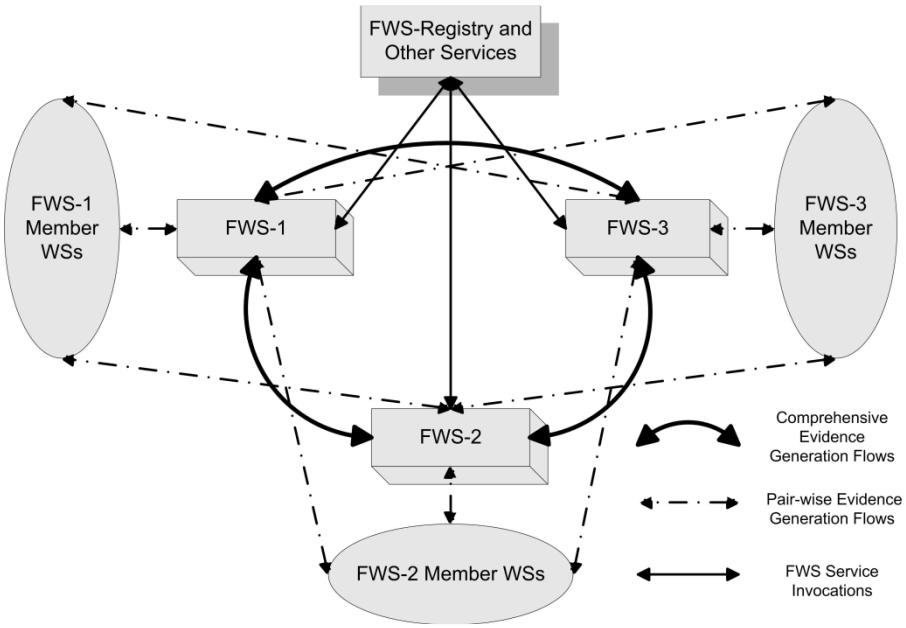


Figure 3.2. The FWS Framework and Message Flows

The following are necessary for FWS systems to function as required:

1. The web-service call stack must be enriched with a WS-Evidence layer.

2. A message format is needed for communicating WS-Evidence layer messages and storing them in the FWS servers.
3. All web services must use a client agent (Evidence module) that re-routes their transactional messages through FWS servers.
4. The underlying system must provide a trust base and cryptographic services.

3.3.1. Enhanced Web-Services Call Stack

The existing WS stack consists of a three layers, where the bottom layer is consisting of SOAP messages, the middle layer of WS-Secure Conversations and the top layer of WSDL specifications. I propose to add an evidences layer in between the middle layer and the top layer to reroute transactions through the FWS servers, thereby allowing *Sender WS* and *Receiver WS* communication using their WSDLs to remain independent of the underlying WS-Evidence layer. Figure 3.3 shows how WS-Evidence is applied to a message that flows through web services and their existing stacks. Flows 1 and 6 show the activity performed by the agents; flows 2 and 5 show the communications occurring at the SOAP level; and 3 and 4 represent inputs and outputs from FWS-TTPs.

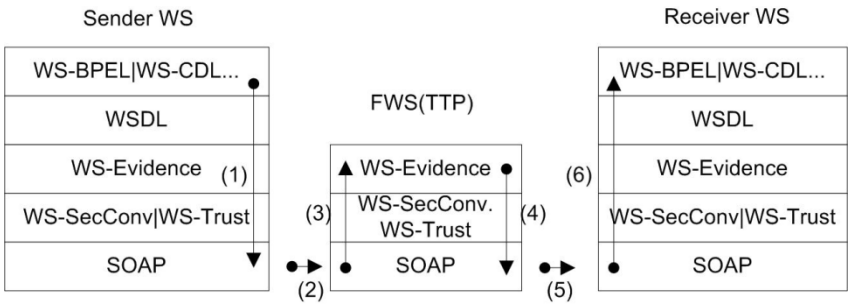


Figure 3.3. WS-Evidence Stack

3.3.2. WS-Evidence Message Format

WS-Evidence uses the message format of <#session|#message|#signature$_K$(#session|#message/sequence|#message/envelope))>, where # refers to the points in XML format, | refers to concatenation of elements, and / points to the sub parts of elements, to exchange between sending customer, FWS and receiving customer. Here the session element identifies a WS-Evidence conversation, and message corresponds to an element carrying the actual upper layer message along with its sequence number (message/sequence) in the conversation, such as, for example, sequence number 2 corresponds to a response message if message exchange pattern (MEP) type is two-way and the protocol is SELP (soon to be described). Each endpoint, either sender or receiver, signs session, message/sequence, and message/envelope parts of the message in the ds:Signature element [26] of the message. Figure 3.4 illustrates a sample WS-Evidence message instance along with significant parts.

```
<wse:deliverRequest ...>
 <wse:Session id="session">
  < wse:ttp>http://www.forensicsoa.org/DeliverService</fws:ttp>
  < wse:Sender>http://www.source.org</fws:Sender>
  < wse:Receiver>http://www.destination.org/services/EchoService</fws:Receiver>
  < wse:Identifier>uuid:88754b00-161a-11da-b6d6-198de3c47c5</fws:Identifier>
  < wse:Operation>echo
    < wse:MEP>Two-Way</fws:MEP>
  </ wse:Operation>
 </ wse:Session>
 < wse:Message>
  <wsu:Timestamp>
   <wsu:Created …>2005-09-08T10:05:27Z</wsu:Created>
  </wsu:Timestamp>
  < wse:MessageID>uuid:c3671020-15e0-11da-9b3a-f0439d4867bd</fws:MessageID>
  < wse:Sequence Id="sequence">1|2|4</fws:Sequence>
  < wse:Envelope Id="envelope">
       PD94bWwgdmVyc2lvbj0nMS4wJyBlbmNvZGlc29hcGVudjpF…….. ……
       …….      L3NvYXBlbnY6Qm9keT48L3NvYXBlbnY6RW52ZWxvcGU+
  </ wse:Envelope>
 </ wse:Message>
 < wse:Evidence>
   <ds:Signature>
    <ds:SignedInfo>
    <ds:Reference URI="#session" >
    <ds:Reference URI="#sequence" >
    <ds:Reference URI="#envelope" >
   </ds:SignedInfo>
   <ds:SignatureValue/>
   <ds:KeyInfo/>
  </ds:Signature>
 </ wse:Evidence>
</ wse :deliverRequest>
```

Figure 3.4. A Sample WS-Evidence Message

FWS store the messages in two formats; `LogRecordIndex` (LRI) and `LogRecord` (LR). A LRI refers to the record of a single message within a WS-Evidence conversation. LR stores entire WS-Evidence sessions including all messages delivered to and/or generated by the FWS. LRI records are used for two reasons: the first for quick searches and the second for pointing to the entire LR. Each LRI is stored at both FWSs (*operator* and *non-operator FWS* -- soon to be described). LR, on the other hand, is stored only at the *operator FWS* and can be reached using LRI's that refer to it. As shown in Figure 3.5, a FWS storing a LRI sets the value of its `status` field to that of the

`message/sequence` part of the message. The FWS also sets the `timestamp` with the value of `message/timestamp` part of the message and the `recordinfo` with the value of `session` part of the message. The `envelope` and `ds:signature` parts are not represented in LRIs but in LRs. LR contains the `recordIndex` part that has final `timestamp` and `status` values of the conversation referring to `timestamp` and `sequence` values of the last message in the conversation respectively.

LRI for wse:Message Seq.1

```
<logRecordIndex ..>
    <timestamp>05:00</timestamp>
    <status>1</status>
    <recordInfo protocol="URI"..>
        <sessionID algorithm="URI"/>
        <MEPType>string</MEPType>
        <agreement/>
        <partners>
            <sender/>
            <fwsttp/>
            <receiver/>
        </partners>
    </recordInfo>
</logRecordIndex>
```

LRI for wse:Message Seq.2

```
<logRecordIndex ..>
    <timestamp>05:01</timestamp>
    <status>2</status>
    <recordInfo protocol="URI"/>
</logRecordIndex>
```

LRI for wse:Message Seq.3

```
<logRecordIndex ..>
    <timestamp>05:02</timestamp>
    <status>3</status>
    <recordInfo protocol="URI"/>
</logRecordIndex>
```

```
<logRecord …>
  <recordIndex>
    <timestamp>05:02</timestamp>
    <status>3</status>
    <recordInfo protocol="URI"..>
      ...
    </recordInfo>
  <recordIndex>
  <wse:Message>
    <session/>
    <message/>
      <timestamp>05:00</timestamp>
      <sequence>1</sequence>
      <envelope>...</envelope>
    <message>
    <ds:signature/>
  </wse:Message>
  <wse:Message>
    <session/>
    <message/>
      <timestamp>05:01</timestamp>
      <sequence>2</sequence>
      <envelope>...</envelope>
    <message>
    <ds:signature/>
  </wse:Message>
  <wse:Message>
    <session/>
    <message/>
      <timestamp>05:02</timestamp>
      <sequence>3</sequence>
      <envelope>...</envelope>
    <message>
    <ds:signature/>
  </wse:Message>
 </logRecord>
```

Figure 3.5. LRI and LR Formats

3.3.3. Evidence Module

Routing transactional information through FWS servers require that all transactions be reliably intercepted and routed, as shown in Figure 3.6.

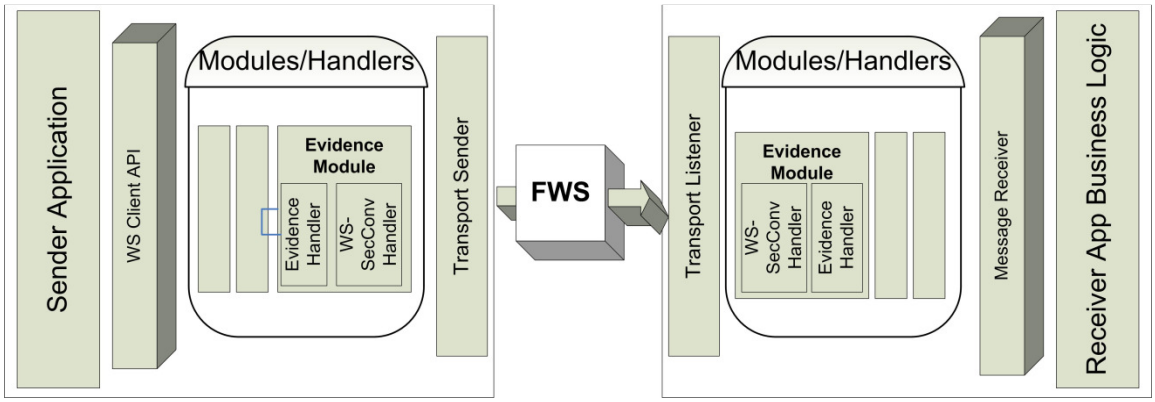


Figure 3.6. Evidence Module Brief Architecture (adapted from [30])

Although the next chapter describes the Evidence Module architecture in detail as a proof of concept, here I briefly describe a sender process and a receiver process sitting in front of each web service end point:

1. **The Sender Process***:* The Evidence Module captures the SOAP message from the upper layer (either from an upper handler in the handler chain or directly from sender API) as shown in the first pillar of Figure 3.7; and encapsulates the message in WS-Forensics message format (see the second pillar in Figure 3.7) by adding signatures, routing the message to the operator FWS, etc., and submitting it to the lower layer—that is, WS-SecureConversation/WS-Trust handler (soon to be described in Section 3.4).

2. **The Receiver Process:** The Evidence Module manages the WS-Evidence message from the lower layer. After validating signature according to the WS-Evidence session context it extracts the original SOAP message and either passes it to another handler (if it exists) in the chain or dispatches it to the intended service|portype|operation entity, provided the message is missing an upper handler in the chain.

Many vendors [27, 28, 29, 30] support handler chains in front of their web application interface. For example, Axis2 [31] allows dynamic module engagement in their web services. My proposal attaches handler modules at both sides of the communication, similar to that of [9].

3.3.4. Underlying Layer

WS-Evidence is designed to run over a secure layer with the following services:

Authentication: senders, receivers and FWS nodes.

Delegated Authentication: As a trusted third party, FWS nodes authenticate themselves to the receiver on behalf of the sender.

Confidentiality and Integrity of the Channels: between senders or receiver and FWS nodes must provide these.

Reliability*:* Messages in channels between FWS nodes and customer nodes must be reliable.

Upper Layer	WS-Evidence (Evidence Module) Receiver<->Sender	WS-SecConv/WS-Trust (e.g. Rampart) Sender/Receiver<->FWS
<soap:Envelope> <soap:Body> <GetCustomerInfo> <CustID>1</CustID> </GetCustomerInfo> </soap:Body> </soap:Envelope>	<soap:Envelope> <soap:Body> <deliverRequest ...> <session> <message > <timestamp/> <sequence>1</sequence> <envelope>$Envelope$</envelope> </message> <ds:Signature> <ds:SignedInfo> <ds:Reference URI="#session" > <ds:Reference URI="#sequence" > <ds:Reference URI="#envelope" > </ds:SignedInfo> <ds:SignatureValue> <ds:KeyInfo> </ds:Signature> </deliverRequest > </soap:Body> <soap:Envelope>	<soap:Envelope ……"> <soap:Header> <wss:Security > <wsc:SecurityContextToken Id='FWS-SCT'/> <wsc:DerivedKeyToken Id='DKS'/> <ds:Signature> <ds:SignedInfo> **<ds:Reference URI="#bodySignature"/>** </ds:SignedInfo> <ds:KeyInfo> <wss:SecurityTokenReference> <wss:Reference URI='#DKS' /> </wss:SecurityTokenReference> </ds:KeyInfo> </ds:Signature> <wsc:DerivedKeyToken Id='DKE'/> <xenc:ReferenceList> **<xenc:DataReference URI='#bodyEncryption'/>** </xenc:ReferenceList> </wsse:Security> </soap:Header> <soap:Body Id="**#bodySignature**"> <EncryptedData Id="**#bodyEncryption**" <EncryptionMethod/> <wss:SecurityTokenReference> <wss:Reference URI='#DKE' /> </wss:SecurityTokenReference> <CipherData> <CipherValue>**7qH0……iA==**</CipherValue> </CipherData> </EncryptedData> </soap:Body> </soap:Envelope>

Figure 3.7. SOAP Message Snapshots in Consecutive Layers

Two properly implemented standards, WS-Trust [32] and WS-SecureConversation [33] satisfy these requirements. WS-Trust issues, renews and verifies tokens for security, and WS-SecureConversation builds secure sessions using XML encryption and signature. The processes described in Section 3.3 require secure channels between end-point web services and FWS nodes. Following briefly show how WS-Evidence message flows between a sender and a receiver through to a FWS using underlying security layer.

1- WS-SecureConversation/WS-Trust handler of the sender grabs the WS-Evidence message (see the second pillar in Figure 3.7) and builds a secure conversation by the means of Security Context Token (SCT) obtained from the Security Token Service (STS). FWS nodes also may have this role. The WS-Evidence message is encrypted by WS-SecureConversation (see the third pillar in Figure 3.7) and pushed into the transport layer to be sent to the FWS node through the conversation

2- WS-SecureConversation/WS-Trust handler of the FWS node receives the encrypted SOAP message, decrypts it, extracts the actual WS-Evidence message, and pushes into the WS-Evidence layer to be processed as described in the next section.

3- After processing WS-Evidence message, the FWS node pushes the message to its WS-SecureConversation/WS-Trust handler to build another secure conversation with the receiver as described in the first step. Then, the message is encrypted by the security handler, to be sent to the receiver through the conversation.

4- WS-SecureConversation/WS-Trust handler of the receiver receives the encrypted SOAP message, decrypts it, extracts the actual WS-Evidence message, and pushes into the WS-Evidence layer to be dispatched.

The reason for implementing two security contexts is to make message content transparent to the FWS so that they can scan the contents for further investigation, thereby allowing the possibility of isolating the ones used for forensic purposes. Having one context would eliminate the initial phases, thus alleviating the performance problem. However, I address such policy-specific decisions in my ongoing work.

3.4. Gathering Evidence at Service Invocation Time

As stated earlier, FWS servers gather pair-wise transactional evidence that flows between sender and receiver web services, using the *Simple Evidence Layer Protocol (SELP)* [7]. There are four entities involved in the process: *sender*, *receiver*, *operator FWS*, and *non-operator FWS*. *Operator FWS* refers to a FWS selected by either party to manage the steps listed below, and the *Non-operator FWS* belongs to the other party. I omit a detailed algorithm to select the operator FWS. Steps followed by the operator FWS are as follows, visualized in Figures 3.8 and 3.9.

Assuming MsgSeq.1 is a request message coded as "1" through WS-Evidence specification , MsgSeq.2 is a response coded as "2", MsgSeq.-1 is a failure coded as "-1", and MsgSeq.3 is an acknowledgment coded as "3", typical FWS TTP acts as below;

1- FWS receives MsgSeq.1 (<#session|#message|#signature$_{Sender\text{-}K}$(#session|"1"|# env)>).

2- Validates, stores the message, creates an LR and LRI for MsgSeq.1 and notifies non-operator FWS.

3- MsgSeq.1 is forwarded to the Receiver and starts a timer.

4- If the response MsgSeq.2 cannot reach the FWS before timing out then, MsgSeq.-1 (<#session|#message|#signature$_{FWS\text{-}K}$ (#session|"-1"|# env)>) is signed by the FWS; it is stored and sent back to the Sender and an LRI is created and sent to the non-operator FWS. If MsgSeq.2 (<#session|#message|#signature$_{Receiver\text{-}}$

$_K$(#sessionl”2”|# env)>) arrives on time then, it is forwarded to the sender and stored in FWS along with notifying the non-operator FWS with its LRI.

5- FWS creates, signs and sends MsgSeq.3 (<#session|#message|#signature$_{FWS\text{-}K}$ (#sessionl”3” |#env)>) to the receiver. It also stores the message in the LR and sends the LRI to the non-operator FWS.

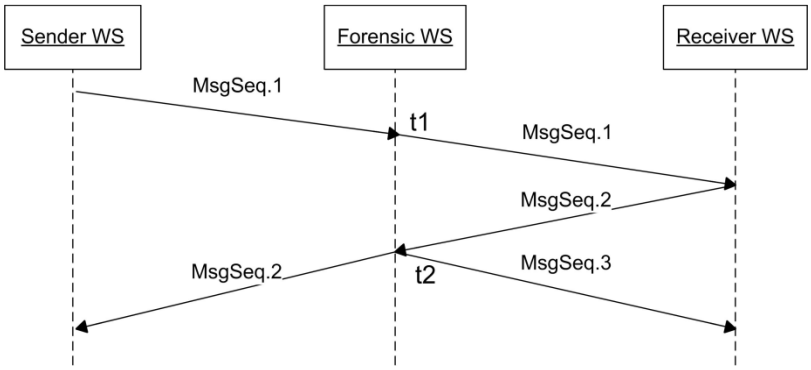


Figure 3.8. An Operator FWS Managing the SELP Protocol (Adapted from [7])

The dependencies between stored data are maintained using LRIs sent from operator FWS to non-operator FWS, thus allowing any further investigator-process to hop up between FWS stations that store dependent records.

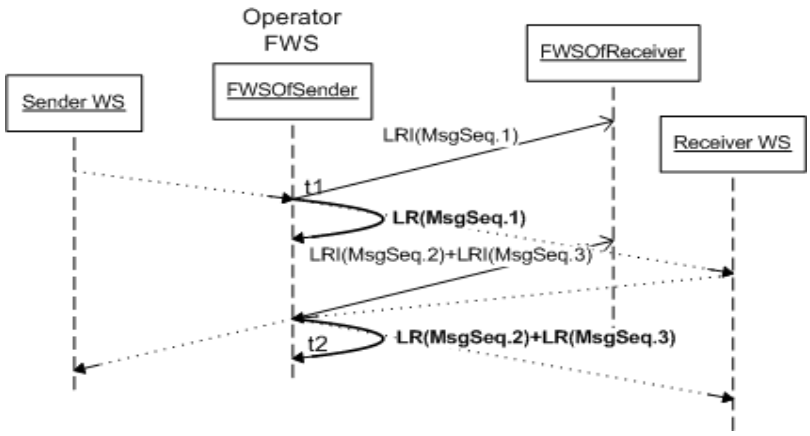


Figure 3.9. An Operator FWS Storing Messages

3.4.1. Pair-wise Evidences

The SELP protocol and FWS event logs retain the evidence to verify the following claims:

Evidence of Origin (EOO): Sender's claims of timely transmitting.

Evidence of Delivery (EOD): Receiver's claims of timely delivery.

Evidence of Failure (EOF): Sender's claims of receiver's failures of timely receipt.

Evidence of Availability (EOA): Either party's claim of non-availability of the other.

Evidence of Agreement Violation (EOV): Either party's claim of contractual violation.

The next chapter shows their building blocks within WS-Evidence messages.

3.5. Creating Evidence for Scenarios

As stated earlier, the main objective of the FWS Framework is post-mortem investigations on inter-dependent scenarios containing more than one party in a

comprehensive manner. In order to do so, as shown in Figure 3.2, the FWS Framework utilizes FWS, FWS-Registry, and other services to generate comprehensive evidence by retrieving and processing evidence of pair-wise transactions stored in FWS nodes. I first describe data types used to represent the comprehensive evidence in the FWS framework, and the scenario generation algorithm that uses a graph to model service inter-dependencies that should exist between the involved parties. Two parameters used in the process, *node thresholds* and *time thresholds*, demarcate the boundary of scenario generation. The nodes of the graph are web services and log records create their inter-dependencies.

3.5.1. Data Types Representing Scenarios

FWS store sender-receiver information between web services in LRI tables that I use to generate the dependency graph. Nodes of the dependency graph are of the complex type `WebServiceNode`, where each `WebServiceNode` has a unique ID and the field `nodeLevel` represents the degree of adjacency of a web service node to the root web service node of the dependency graph. For example, `nodeLevel=1` means the web service is directly adjacent to the root web service node. The field `nodeThreshold` is the boundary of the dependency graph. For example, a graph of a web service node with `nodeLevel=3` and `nodeThreshold=3` will not expand further over this specific web service node but only over other nodes with a lesser `nodeLevel` than `nodeThreshold`. The edges of the graph are represented using the complex data type `LogRecordEdge` with

components `SenderID` and `ReceiverID` attributes. Figure 3.10 and 3.11 shows a sample node and a sample edge respectively.

```
<webServiceNode id="www.geocoding.com" nodeLevel="2" nodeThreshold="3"
    <webService>
        <Location>http://www.geocoding.com/service</Location>
        <Port>LongitudePT</Port>
        <Service>#geosrv</Service>
        <Operation>getLongitude</Operation>
        <IPAddress>66.234.12.231</IPAddress>
        <InstanceID>
    </webService>
</webServiceNode>
```

Figure 3.10. An Instance of WebServiceNode

```
<logRecordEdge id="uuid:#21323232323-11.12.2007:12:23:04"
senderID="www.geocoding.com" receiverID="www.weatherservices.com"
dependencyDirection="Two-Way">
    <logRecord>
        <recordIndex>
            <timestamp>2002-10-10T12:00:00-05:00</timestamp>
            <status>1</status>
            <recordInfo protocol="SELP">
                <sessionID>
                <MEP>Two-Way|One-Way</MEP>
                <agreement>
                <partners>
            </recordInfo>
        </recordIndex>
        <wse:Message> $MsgSeq.1 </wse:Message>
        <wse:Message> $MsgSeq.2 </wse:Message>
        <wse:Message> $MsgSeq.3 </wse:Message>
    </logRecord>
</logRecordEdge>
```

Figure 3.11. An Instance of LogRecordEdge

3.5.2. Building Digital Evidence Bag

All dependency decisions use the following:

```
WS-A sends WS-B in “one way”=> WS-B depends on WS-A

WS-A sends WS-B in “two way”=> WS-B depends on WS-A and WS-A depends on WS-B
```

FWS build a Digital Evidence Bag complying with the requirements in [34] by using the algorithm written using pseudo BPEL in Figure 3.12. The algorithm works as follows: Any `Requestor`, such as a plaintiff or a prosecutor participating in the FWS framework can start building Digital Evidence Bags for an alleged attack by invoking the `generateEvidenceBag` process—which is done by including the `webServiceNode` (pointing to the suspected WS as root), `startTime` (time when suspected activity first detected), `timeThreshold` (defines the scope of investigation in terms of time), and `nodeThreshold` (defines the scope of investigation in terms of node depth) in the `EvidenceBagIn` message in line 4. Between lines 5 and 7 the FWS first checks which FWS (`rootFWS`) controls the web service node in question (`rootWS`), assigns the address of the `rootFWS` partner link, and starts processing by invoking the `collectDependents` process of that `rootFWS` with the `DependentsBagIn` message, which in turn runs in a distributed-recursive manner. It is replied to with the `DependentsBag` message and employs a set of refinement tasks on the `logRecordEdges` part of the message, such as sorting, and grouping the records by the `fwsttp` field of LRI information in each `logRecordEdge` in the `logRecordEdges` array as referred in line 8. Because the `collectDependents` process only stores LRI information in `LogRecordEdges` there are no actual log records contained at this step. In order to turn lightweight `LogRecordEdges` (with LRI) into actual `LogRecordEdges` containing LRs, the `generateEvidenceBag` process first extracts distinct `fwsttp`'s from `logRecordEdges` into an array as pointed in line 9. Between lines 10 and 21, utilizing the *flowN* structure in BPEL [35], the algorithm creates dynamic parallel execution scopes for each distinct `fwsttp`. For each distinct

`fwsttp`, it also creates dynamic partner links named `OwnerFWSOfLogRecords` and builds `LogRecordEdgesInput` arrays. Then, `getLogRecordsByValue` operations are invoked for each parallel scope and the results are combined in `logRecordEdgesForEvidenceGraph`. `evidenceBagOut` is assigned with `logRecordEdgesForEvidenceGraph` and constitutes the actual `EvidenceGraph` document. Between lines 23 and 25, some other necessary procedures may be applied to the document, such as scanning, signature matching, encrypting and signing, according to the policy in an appropriate order. Finally the requester is replied to in line 26.

```
1. partnerLinks: SecurityService; VirusScannerService;
SignatureDetectionSrv; RootFWS; Requestor; FWSRegistry
2. variables: EvidenceBagIn; EvidenceBagOut; LogRecordEdges;
DependentsBagIn; DependentsBag;
LogRecordEdgesForEvidenceGraph;
3. begin
4.      receive EvidenceBagIn from Requestor
5.      invoke getFWSs(RootWS) in FWSRegistry
6.      assign RootFWS partnerLink
7.      assign EvidenceBagIn to DependentsBagIn
8.      invoke collectDependents(DependentsBagIn) in RootFWS
9.      assign DependentsBag to LogRecordEdges
10.     assign distinct ArrayOfFWSTTP from LogRecordEdges
        ←!– Invokes a set of FWSTTPs to get actual LREs by their LRIs →
        ←!– using flowN loop structure →
11.     flowN N='countNodes(' ArrayOfFWSTTP '...)' indexVariable='index'
12.             partnerLink: OwnerFWSOfLogRecords
13.             variables: LogRecordEdgesOutput
14.             assign OwnerFWSOfLogRecords partnerLink
15.             invoke getLogRecordsByValue in OwnerFWSOfLogRecords
16.             receive LogRecordEdgesOutput as getLogRecordsByValue callback
                ←!————————– Stores the result ————————————→
17.             append LogRecordEdgesForEvidenceGraph
                  from LogRecordEdgesOutput
18.     end of flowN
19.     assign LogRecordEdgesForEvidenceGraph to EvidenceBagOut
20.     invoke scan(EvidenceBagOut) in VirusScannerService
21.     invoke detect(EvidenceBagOut) in SignatureDetectionSrv
22.     invoke signAndEncrypt(EvidenceBagOut) in SecurityService
23.     reply EvidenceBagOut to Requestor
24. end
```

Figure 3.12. Pseudo BPEL for Generating Evidence Bags

`generateEvidenceBag` is a wrapper process of the `collectDependents` process (specified as a BPEL in my original design) that contains an algorithm that is inspired by King's dependency graph generation algorithm [36] and Wang's evidence graph generation study [37]. I use `LogRecordIndex`'s (LRI) and not `LogRecord`'s (LR) because the latter reside only on one FWS, making the algorithm unusable. The process first creates instances of `WebServiceNodes` and `LogRecordEdges` arrays using the schema mentioned earlier. Then, it loads the `dependentsBagIn` message into these objects setting the `webServiceNode` part as a root level node; all other values in the input message are loaded into the corresponding variable. After the initialization phase, the algorithm listed in Figure 3.13 is used. Created objects, `webServiceNodes` and `logRecordEdges` are the nodes and edges of the dependency graph. The algorithm traverses the LRIs starting from the decreasing order of time in search of dependent web service nodes among the sender/receiver fields of the log records and inserts them into the `logRecordEdges` setting `senderID`, `receiverID`, and `dependencyDirection` attributes (if their timestamp is within the time threshold). When a new partner web service is found in the LRIs, it adds this partner web service node into the `webServiceNodes` object only if the current web service node's `nodeLevel` is equal or less than the `nodeThreshold`.

When a partner web service node that does not belong to the operator FWS is found, the neighbor FWS hosting this partner web service node is found by querying the FWS-Registry. The same algorithm is executed in the chosen neighbor FWS by invoking the `collectDependents` process of that FWS. This time, current web service nodes and

the log record edges added into the graph so far, are sent to initiate the same process in neighbor FWS, along with a smaller start time and larger node level values in the web service nodes. Because the start time and node level information are kept and transferred to external FWSs, the gathering of unrelated log records and infinite loops is prevented. The return message from the neighbor FWS is in the `DependentsBag` schema; therefore, web service nodes and log record edges are added to the current `DependentsBag`.

```
←!– Starts extracting values (timeThreshold, nodeThreshold, etc.) →
←!– from DependentsBagIn and initializes creating the →
←!– WebServiceNodes and LogRecordEdges instances of GRAPH →
1. baseTime = startTime - timeThreshold
2. for each logRecordIndex LRI ∈ FWS {
3.    timeThreshold=timeTreshold - (startTime - LRI.timeStamp)
4.    startTime=LRI.timestamp
5.    for each webServiceNode WS ∈ GRAPH {
6.        if (SenderWS | ReceiverWS ∈ LRI & LRI ∉ GRAPH &
          R.timestamp ≥ baseTime & WS.nodeLevel ≤ WS.nodeThreshold) {
7.            Add LRI as edge into GRAPH
8.            if (LRI's partner web service PWS ∈ GRAPH) {
9.                PWS.nodeLevel=WS.nodeLevel+1
10.               PWS.nodeThreshold=nodeThreshold
11.               Add the PWS into GRAPH }
12.           if (LRI's PWS ∈ this.FWS & LRI's PWS ∈ GRAPH) {
13.               NeighbourFWS = getFWS(PWS)
14.               NeighbourFWS.collectDependents(DependentsBagIn)
15.               Merge DependentsBagOut into GRAPH}}}}
16. return GRAPH in DependentsBagOut format
```

Figure 3.13. Comprehensive Evidence Generation (*collectDependents* Algorithm)

The `collectDependents` process returns a `dependentsBag` output to the wrapper process, `generateEvidenceBag`. Because `logRecordEdges` in `dependentsBag` only contains LRI information, it is refined by the wrapper process, and building the `evidenceBag`. In order to meet the requirements [38] on chain of custody for digital forensics, the document is signed with the private key of the host FWS.

3.6. A Case Study: The XSS Attack

Now, I show how any agent can use FWS to create comprehensive evidence for the XSS attack described earlier. Through the Case Study, I assume that FWS-1 owns POR_WS (Portal Web Service), GEO_WS (Geocoding Web Service), and so many others, while FWS-2 owns WEA_WS (Weather Web Service) and MET_WS (Meteorology Web Service), along with other many services. Table 3.1 lists sample log records available at FWS-1 and FWS-2 in LRI (Log Record Index) format. Arrows illustrate how the *collectDependents* algorithm reveals activities dependent to each other spanning over many web services and FWS stations. The bold records refer to log records linked to each other and used to build the dependency graph as edges between web service nodes. Each record applies the LRI format ({Timestamp| SessionID| status| fwsttp| Sender| Receiver}).

Assume that an official decides to generate a digital evidence bag of this incident using the above parameters--please remember the "collectDependents" algorithm in Figure 3.13 and the parameters to draw the dependency scope in the examination: *rootWS=POR_WS*,

startTime=17, *nodeThreshold=3,* and *timeThreshold=16*. The examiner first defines the *rootFWS* by querying from FWS-Registry and sets FWS-1 as the root, thereby invoking the *collectDependents* process at FWS-1. I now apply the algorithm to this example through the following steps.

Table 3.1. Log Record Indexes in FWS-1 and FWS-2

LRIs in FWS-1	LRIs in FWS-2
…	…
1|234|Response|FWS-1|TRE_WS|VRE_WS.vresPT.ReserveVehicle	1|234|Response|FWS-1|TRE_WS|VRE_WS.vresPT.ReserveVehicle
2|2134|Request|FWS-2|WEA_WS|GEO_WS..getLongtitute	**2|2134|Request|FWS-2|WEA_WS|GEO_WS..getLongtitute**
3|2134|Response|FWS-2|WEA_WS|GEO_WS..getLongtitute (7)	**3|2134|Response|FWS-2|WEA_WS|GEO_WS..getLongtitute** (5)
4|2164|Response|FWS-1|TRE_WS|HRE_WS.hresPT.ReserveHotel	4|2164|Response|FWS-1|TRE_WS|HRE_WS.hresPT.ReserveHotel
6|21572|Request|FWS-1|TRE_WS|ARE_WS.aresPT.ReserveAirline (6)	**5|2196|Request|FWS-2|MET_WS|WEA_WS..updateRegion**
9|21572|Response|FWS-1|TRE_WS|ARE_WS.aresPT.ReserveAirline	7|34563|Request|FWS-2|VRE_WS|HRE_WS.hresPT.HotelLocation (4)
10|34534|Request|FWS-2|POR_WS|WEA_WS..weatherRequest	8|34563|Response|FWS-2|VRE_WS|HRE_WS.hresPT.HotelLocation
12|22534|Request|FWS-1|TRE_WS|VRE_WS.vresPT.ReserveVehicle	**10|34534|Request|FWS-2|POR_WS|WEA_WS..weatherRequest**
13|34534|Response|FWS-2|POR_WS|WEA_WS..weatherRespond	11|34567|Response|FWS-2|VRE_WS|HRE_WS.hresPT.HotelLocation (2)
14|34523|Request|FWS-1|WEA_WS|GEO_WS..getLongtitute (1)	12|22534|Request|FWS-1|TRE_WS|VRE_WS.vresPT.ReserveVehicle (3)
17|34523|Response|FWS-1|WEA_WS|GEO_WS..getLongtitute	13|34534|Response|FWS-2|POR_WS|WEA_WS..weatherRespond
…	14|34523|Request|FWS-1|WEA_WS|GEO_WS..getLongtitute
…	...
…	17|34523|Response|FWS-1|WEA_WS|GEO_WS..getLongtitute

Step 1: FWS-1 first retrieves LRIs in Table 3.1, and starts traversing on LRIs in decreasing time order. In this example, this order spans LRIs from time 17 to time 1

because of *timeThreshold* value 16. The algorithm skips LRIs in times 17 and 14 because they do not contain POR_WS as any other partner in their records. It finds an LRI with time 13 related to POR_WS, adds the LRI as *LogRecordEdge*, and adds the partner web service, WEA_WS, as the dependent *WebServiceNode* because it has yet to be included in the graph created so far.

Step 2: Because WEA_WS is registered to another FWS, FWS-2, this process assigns the FWS-2 as neighbor FWS and invokes the *collectDependents* process with *rootWS=WEA_WS, startTime=13, timeThreshold=12, logRecordEdges,* and *webServiceNodes* already in the graph.

Step 3: FWS-2 retrieves the LRIs and starts from the LRI with time 13. It ignores LRI 13 because it is already in the graph and LRIs with time 12 and 11 because of irrelevancy.

Step 4: FWS-2 adds the LRI in time 10 because the partners are already in the graph. The records in 8 and 7 are ignored because of their irrelevancy to the graph. FWS-2 adds the LRI in time 5 because one of its partners is included in the graph. The other partner MET_WS is added to the graph with a higher node level 3. Because MET_WS is registered to FWS-2 there is no need to call another FWS to collect its dependents.

Step 5: After ignoring 4, the records in time 3 and 2 are added because of their relevancy to WEA-WS.

Step 6: FWS-2 returns to FWS-1 since there remains no record to traverse.

Step 7: FWS-1 continues to process from LRI with time 12. It ignores LRIs with time 10, 3, and 2 although they are relevant, but they are already in the graph. It also ignores other records because they are unrelated to the graph.

```
<EvidenceBag xmlns:p1="http://ite.gmu.edu/schema/FWSXmlSchema"
xmlns:p2="http://www.w3.org/2000/09/xmldsig#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<evidenceBagSynopsis>
    <OperatorFWS FWS-1>
    <RootWS POR_WS >
    <processStartTime>2002-10-10T12:00:00-05:00</processStartTime>
    <processEndTime>2002-10-10T12:00:00-05:00</processEndTime>
    <startTime>2002-10-10T12:00:00-05:00</startTime>
    <timeThreshold>16</timeThreshold>
    <nodeThreshold>3</nodeThreshold>
    <numberOfWebServiceNodes>4</numberOfWebServiceNodes>
    <numberOfLogRecordEdges>3</numberOfLogRecordEdges>
    <requestorPrincipal>www.justice.gov/#33332324242</requestorPrincipal>
</evidenceBagSynopsis>
<dependencyGraph>
    <webServiceNodes length="4">
        <webServiceNode id="POR_WS" nodeLevel="0"..>
        <webServiceNode id="WEA_WS" nodeLevel="1"..>//added at step 1
        <webServiceNode id="MET_WS" nodeLevel="2"..>//added at step 4
        <webServiceNode id="GEO_WS" nodeLevel="2"..>//added at step 5
    </webServiceNodes>
<logRecordEdges length="3">
    <logRecordEdge id="34534" senderID="POR-WS" receiverID="WEA_WS"
dependencyDirection="Two-Way"> //added at step 1
        ...
        <Message>MsgSeq.1</Message>//added at step 4
        <Message>MsgSeq.2</Message>//added at step 1
        ...
    </logRecordEdge>
    <logRecordEdge id="2196" senderID="MET_WS" receiverID="WEA_WS"
dependencyDirection="One-Way"> //added at step 4
        ...
        <Message>MsgSeq.1</Message>//added at step 4
        ...
    </logRecordEdge>
    <logRecordEdge id="2134" senderID="WEA_WS" receiverID="GEO_WS"
dependencyDirection="One-Way"> //added at step 4
        ...
        <Message>MsgSeq.1</Message>//added at step 5
        <Message>MsgSeq.2</Message>//added at step 5
        ...
    </logRecordEdge>
</logRecordEdges>
</dependencyGraph>
<ds:Signature>
<ds:SignedInfo>
<ds:Reference URI="#evidenceBagSynopsis" >
<ds:Reference URI="#dependencyGraph" >
</ds:SignedInfo>
<ds:SignatureValue>
<ds:KeyInfo>
</ds:Signature>
</EvidenceBag>
```

Figure 3.14. The Instance of EvidenceBag for The Case Study

Figure 3.14 illustrates the GRAPH produced through the above steps, where `LogRecordsEdges` is an array of `LogRecordEdge`'s and `WebServiceNodes` is of `WebServiceNode`'s collected through the run. The output is signed (notice ds:Signature) by the executer FWS of the `generateEvidenceBag` process.

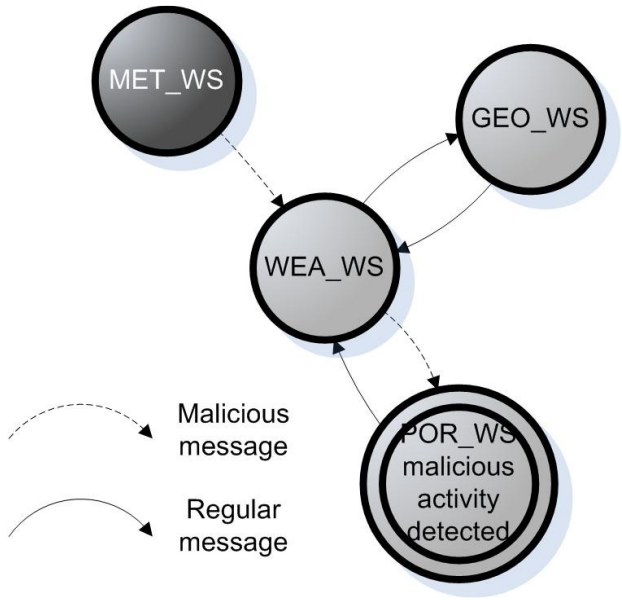


Figure 3.15. The Dependency Graph for the Case Study

The case study described above shows how the FWS framework could be helpful for revealing dependencies between web services through composition models and scenarios as illustrated in Figure 3.2. Arrows refer to Log Records as edges in a graph; and circles refer to Web Services as nodes. The figure depicts how the web service choreography instance through the Case Study could be represented. This figure is also a result of a typical digital evidence bag document that constitutes a graph which points to dependencies among the source (MET_WS) and the victim (POR_WS) of malicious

activity/path, as well as a possible stepping stone (WEA_WS) through the incident. The scenarios can be improved; and FWS could be applied to more complex attacks. For the sake of clarity, a simple scenario has been implemented through this chapter.

3.7. Related Work

To the best of my knowledge, there is no distributed forensic framework for investigating inter-related web services designed so far. However, the work cited in the rest of this section shares some common features with my objectives or methods.

WS-NRExchange [9] influenced the model I employ for pair-wise evidence generation with some differences. [9] provides a framework to support fair non-repudiable B2B (Business to Business) communications on the basis of a trusted deliver agent notion. It implements the coeffey-saidha [39] protocol to provide non-repudiation in their work. However, the framework is designed to run with other protocols as well. Reference [9] only proposes delivering evidences to the related parties, but not preserving them in trusted agents. Furthermore, choreographed, composed services are ignored. Although [9] was not designed for forensics, I use relevant parts in my pair-wise evidence generation.

Herzberg et al. [7] introduces the notion of having an Evidences Layer for e-commerce transactions. They propose this layer to be at the bottom of the e-commerce stack and on top of a transport layer (such as TLS/SSL, or TCP/IP). They introduce two protocols to

generate and deliver evidences to involved parties in message exchange; the first is the *Simple Evidence Layer Protocol* and the second is the *optimistic* one. They employ notaries in the first protocol while generating and delivering evidences. I use the layering approach of Herzberg [7] in the web service stack with minor changes, such as adding the time stamping point, and the use of their SELP as my pair-wise evidence generation protocol. Like others, Herzberg et al [7] was not designed for forensics.

I use trusted third parties for pair-wise evidence generation as did Coffey-Saidha et al [39]. Certified email protocols [44] have also used them, although there are mail certification protocols without TTP [8]. Although inline TTPs are immature for business transactions, they add value to forensics evidence. Onieva [40] gives the intermediary usage perspective in the implementation of inline TTPs for e-commerce transactions. Onieva also supports multi-recipient cases through these intermediaries, but not for forensics. Bilal [41] uses BPEL for non-repudiation protocol implementation in web services, but does not use TTP, his method thereby lacking the capability to handle message content.

WSLogA [11] tracks web service invocations by logging service invocations using SOAP intermediaries. Therefore, it captures the external behavior of service invocations. The main purpose of WSLogA is to provide feedback to business organizations by comprehensively logging services usage records. However, it does not address any

distributed collection mechanism necessary to gather comprehensive forensic evidence over services sharing multiple servers.

My work has been influenced by many papers on network forensics, of which I describe two. Wang uses IDS (Intrusion Detection System) alerts [37] to generate an evidence graph for network forensic analysis. *Local reasoning* and *global reasoning* helps them in defining malicious activity in individual hosts and networks respectively. Unlike Web Server Nodes in my study, they use hosts as nodes in their graphs.

ForNet [43] is another distributed forensic framework that uses logs from routers in a network to run agents that provide their log records to ForNet servers. Unlike Wang [37], ForNet use succinct information of every regular network packets adequate to trace the actual source of packets even when they are spoofed. Although ForNet was not designed for Web Services, my work has been inspired by the design of it.

[45] introduces web services collecting log records. Instead of neutral observation of interactions, it, rather, focuses on providing PKI-based secure audit trails that can be stored any un-trusted hosts so that any modification, alteration on audit trail cannot go undetected. Similar mechanism is designed partially for my digital evidence bag documents, since the evidences should have integrity.

3.8. Conclusions

Composed, choreographed or stand-alone web services span many applications and legal domains. Consequently, any vulnerability in one service can be exploited to affect more than one service. Once a complaint of an alleged attack is launched, it is necessary to investigate the nature and source of the attack and assign blame for it. I proposed a framework referred to as Forensic Web Services that provides this capability as a service to other web services by logging service invocations. I have shown my preliminary design and stated how collected logs can provide the capability to produce a bag of digital evidence to re-create the attack from its logs.

CHAPTER 4

EVIDENCE GENERATION MODEL FOR WEB SERVICES

4.1. Introduction

Many multi-party businesses are in the process of being transferred to web services. In addition, due to the open nature of the Internet, despite best efforts, imposters and even legitimate actors may not adhere to their obligations within a complex business transaction. In such scenarios, it may be necessary to examine exchanged information, and re-create the business process in order to find the incorrect or inappropriate components, and possibly correct them. The success of such an effort depends upon being able to preserve the data obtained from the transaction, a.k.a. *evidence,* in their original form so that the parties to a dispute or an external entity - such as a jurist - could be convinced of the true nature of the transaction that took place. Because most complex web transactions are constructed by a synchronized collection of two-party data exchanges, if exchanges can be preserved in their original form and provide convincing arguments as to their synchronized exchange, or lack thereof, I can convincingly construct the instance of the transaction that took place. The objective of this chapter is to provide a systematic methodology for this proposed process. In this chapter I concentrate on the breaches that violate security related properties, and I concentrate on a

methodology that addresses preserving evidence of three main security properties: namely, confidentiality, integrity and availability. With respect to these issues, I describe a methodology that promotes some software engineering issues that arise in the process: namely, separation of concerns, reusability, and interoperability. Integrating an evidence generation mechanism with existing systems is a burden while using current approaches [9, 6]. It requires some level of control over communicating parties, such as retaining complete records of exchanged messages, etc. It may also force senders/receivers to engage in additional activities such as signing and joining sessions. The question is: how can the evidence layer achieve this in a way that is transparent to the business logic?

As a solution, I propose enhancing the notion of Evidences Layer proposed by Herzberg [7] for networks into a three-layered framework, and show a modular design of my framework. I achieve this modularity by using the principle referred to as Inversion of Control (IoC) for web services, where IoC [46] is a style of software construction whereby generic code controls the execution of problem-specific code. The generic code is developed independently and reused on demand in different contexts. IoC provides sufficient control over communicating parties while maintaining separation of evidence generation logic. Using runtime interception, the proposed evidence layer can record messages exchanged among parties, check session invariants, and verify the existence of signatures on the fly without polluting the business logic.

The rest of the chapter is written as follows. Section 4.2 describes my overall design approach. Section 4.3 describes the three layers of the framework. Section 4.4 presents my prototype architecture. Section 4.5 describes the process of creating evidence of global behavior from local behaviors. Section 4.6 describes related work and Section 4.7 concludes the chapter.

4.2. Approach

Figure 4.1 shows a high-level view of my three-layered evidence generation model, with three parties transacting with each other. If all of them subscribe to proposed service, the evidence layer establishes communication channels transparent to the three parties to collect and preserve communications at the lowest layer of the framework. This is done by storing them in cryptographically secure repositories. I use IoC to weave these evidence generation modules that intercept invocation from/to the services and forward them to the evidence framework. I show a prototype implementation of the pair-wise model using Axis2, and present many protocols that can be used at this layer.

The second layer can use a rule engine or a mining system to derive additional facts from them, thereby being able to reveal violations that are not directly evident in pair-wise message records. Although I do not provide details of rules and their utility at this layer, I briefly demonstrate how evidences of complex scenarios can be derived from stored instances of pair-wise communications. The next chapter shows the details.

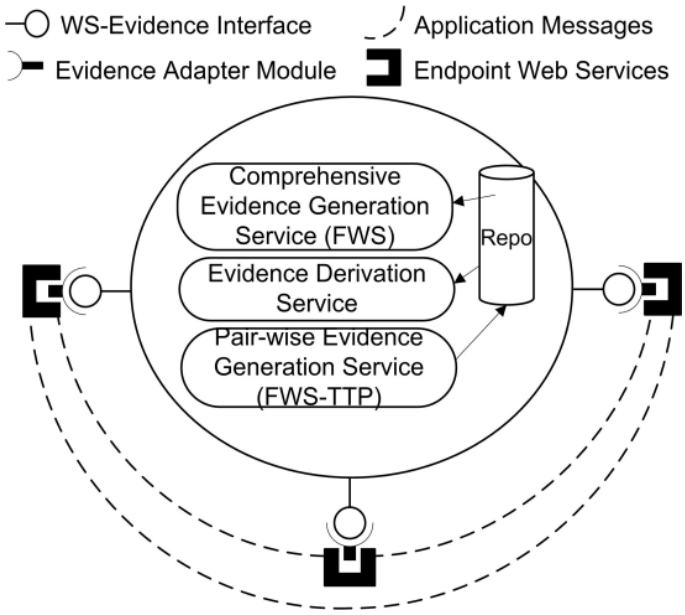


Figure 4.1. The Evidence Generation Framework

4.3. Evidence Generation and Retrieval

4.3.1. Pair-wise Evidence Generation

I provide *non-repudiation*, *fairness*, and *timeliness* in my pair-wise evidence generation. I use digital signatures to provide proof of receipt and delivery, link a message to its creator/sender and provide message integrity. In web services architecture, URLs may define identities and some organizational information as may appear in a digital certificate. Fair message exchange and non-repudiation evidences may be problematic because a sender may always prefer to get a proof or receipt.

For accountability, I use fair non-repudiation mechanisms that utilize Trusted Third Parties (TTP). I do so because, although there are fair exchange protocols for two participants (for e.g. Markowitch [8]), these assume that the participants have a-priori knowledge of the message contents; I do not use them because web services may not always know the contents.

I require *timeliness* because of the time sensitive nature of most business transactions. Due to communication delays and the possibility of the endpoint's malicious intent, many previous studies suggest using time-stamp authorities—but these take additional messages. Although the framework has endpoints signed the time-stamps at their sites it bases evidence records on time observed at TTPs.

Evidence servers gather pair-wise transactional evidence that flows between sender and receiver web services, employing inline TTPs using the *Simple Evidence Layer Protocol (SELP)* or offline TTPs using *Optimistic Evidence Layer Protocol* (*OELP).* Herzberg's [7] SELP and OELP are two protocols used by end-points to obtain non-repudiable evidence by using a specific message format and digital signatures. Because the messages are XML and SOAP based I use the message format of `<#session|#message|#signature`$_K$`(#session|#message/sequence|#message/envelope))>` of which I described the parts in previous chapter.

4.3.2. Evidence Strata

The evidence repositories created using any of the stated protocols can be used to retrieving (1) pair-wise evidence, (2) derived evidence, and (3) comprehensive evidence.

Pair-wise evidence refers to evidences that are part of a particular interaction that may be one-way, two-way, complete, or incomplete. Pair-wise evidences are mostly of interest to one party and exchanged at service invocation time. They are also used to construct derived and comprehensive evidence. *Evidence of origin* would help the receiver to hold the sender accountable of the incoming message and *evidence of delivery* would help a sender in the same way. Conversely, an *evidence of failure* of either of these reveals the *non-availability* of the other party for service.

Derived evidence helps revealing violations against one web service's security property or violations regarding a service level agreement, such as *evidence of availability* or *evidence of violation,* respectively.

Comprehensive evidence refers to evidences that help in revealing multi-party executions of global compositions.

4.4. Prototype Architecture

4.4.1. Reference Architecture

It is essential to know little about Axis2 [30, 31] architecture in order to comprehend how evidence layer model is realized through a selected endpoint framework.

Message Exchanges in Axis2: Four main types of message exchange occur in Axis2 architecture; In-Only, Out-Only, In-Out, and Out-In. *In-Only* and *Out-Only* messages represent incoming or outgoing messages that are one way and not responded. Through the chapter, I, uniquely, call these messages *OneWay* for that In-Only and Out-Only messages refer to the same thing in the wire. In server side, typically, for one way operations an *operation context* that features in-only message exchange pattern is created. The client side, on the other hand creates an operation context featuring out-only pattern. Client side calls fireAndForget() method to push out-only messages into out-flow, thus sending it to the ultimate goal as a one way message. In-Out and Out-In messages represent incoming or outgoing messages that are two way and are to be responded. This pattern introduces two messages at least; and I call the first one, *TwoWay1st* and *TwoWay2nd* for the second. In server side, typically, for two way operations an operation context that features in-out message exchange pattern is created. The client side, on the other hand creates an operation context featuring out-in pattern. Client side calls sendReceive() method to push first message into out-flow, thus sending it to the ultimate

goal as a request message. After application logic prepares the response message the message is pushed into out-flow and sent back to the sender client.

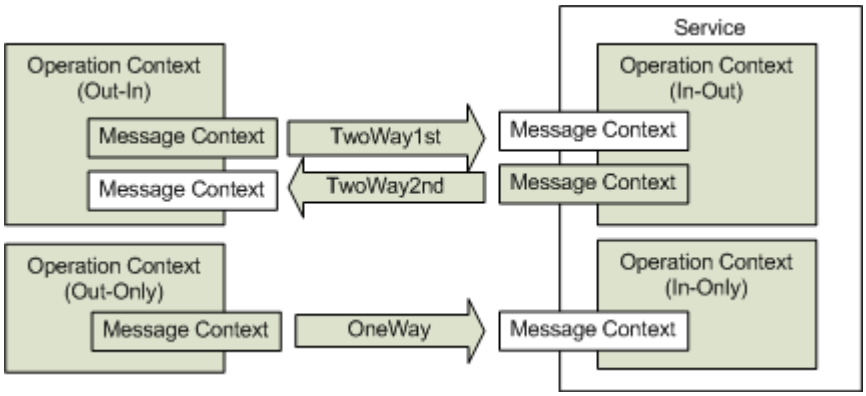


Figure 4.2. Context-based Message Exchange in Axis2

Context-based management: A typical message to be sent or received has a lifecycle in Axis2 architecture. Business logic, for instance, creates a request message with appropriate parameters, such as Endpoint reference, target service name, operation or action name for the service. A *Message Context* through the out flow has been created for that request. The system preserves all necessary properties of the message and builds the basic request as a SOAP message upon the configuration of the system. Through the out flow, the system, as the case may access and alter Message Context.

Axis2 manages WSDL message exchange patterns by the means of mutually created *Operation Contexts*. In accordance with the role (client or server) it features, any endpoint send and receive messages through an Operation Context which contain message contexts, as illustrated in Figure 4.2.

Extensible Message Handling: Axis2 architecture provides a phased handling mechanism in both inflow and outflow pipes. The message context mentioned above drops by each handler registered for related service. A handler chain mechanism regulates the order of handler execution by the means of phases defined in configuration context, that is, from the very beginning of Axis2 web service framework. Axis2 allows user defined phases, thus leading handlers to run in a layered behavior.

4.4.2. Evidence Module Architecture

The prototype implementation is simply designed over the above architecture, where Axis2 allows modules to place their own *handlers* to retain their own control over messages. A module can utilize the extensible message handling mechanism of Axis2 to craft and process messages through *In-Flow* and *Out-Flow* pipes towards Axis2 channels – all business layer messages on their way to the transport layer and all transport layer messages in the opposite direction. As shown in Figure 4.3, I employ EvidenceOutHandler and EvidenceInHandler to handle application messages that are originating from the services engaged to the module and WS-Evidence messages targeting evidence-mindful services. To successfully realize pair-wise evidence generation, there is need for additional messages shuttling between parties (endpoints and TTPs), such as control and acknowledgment messages which entail the module to have *evidence context* that will last longer than regular Axis2 operation contexts. Therefore, my evidence module proposes evidence sessions that squeeze in existing context-based

sessions that are inevitably retarded because of the delays employed by additional messages and evidence processing. I however map evidence context to operation context using *Internal Message Handling* to successfully abstract pair-wise evidence generation from existing contexts. Evidence context includes information such as protocol type (e.g. SELP or OELP etc), time-out value, TTP address, etc. My module employs *control messages* in order for participants to negotiate on evidence context information and to start an evidence session. Through the first design I introduce three control messages; *CreateSession, CreateSessionResponse,* and *TerminateSession.*

Once the session is established, the module performs internal message handling that refers to a set of message transformation activities (building and processing respectively outgoing and incoming messages) between transport, WS-Evidence, and upper layers. *Internal component calls* facilitate the transformation process leading to the use of custom components, such as MessageBuilder, MessageSender, MessageValidator, and MessageProcessor. As shown in Figure 4.3, a typical pair-wise evidence generation session is maintained by means of the software artifacts described below.

EvidenceInHandler captures incoming messages before releasing them to the use of the upper phases' handlers and the targeted message receiver. This handler looks up the related evidence session using session ID value in the incoming message. According to the state of the session and the type of the message, it calls the required component to process the message.

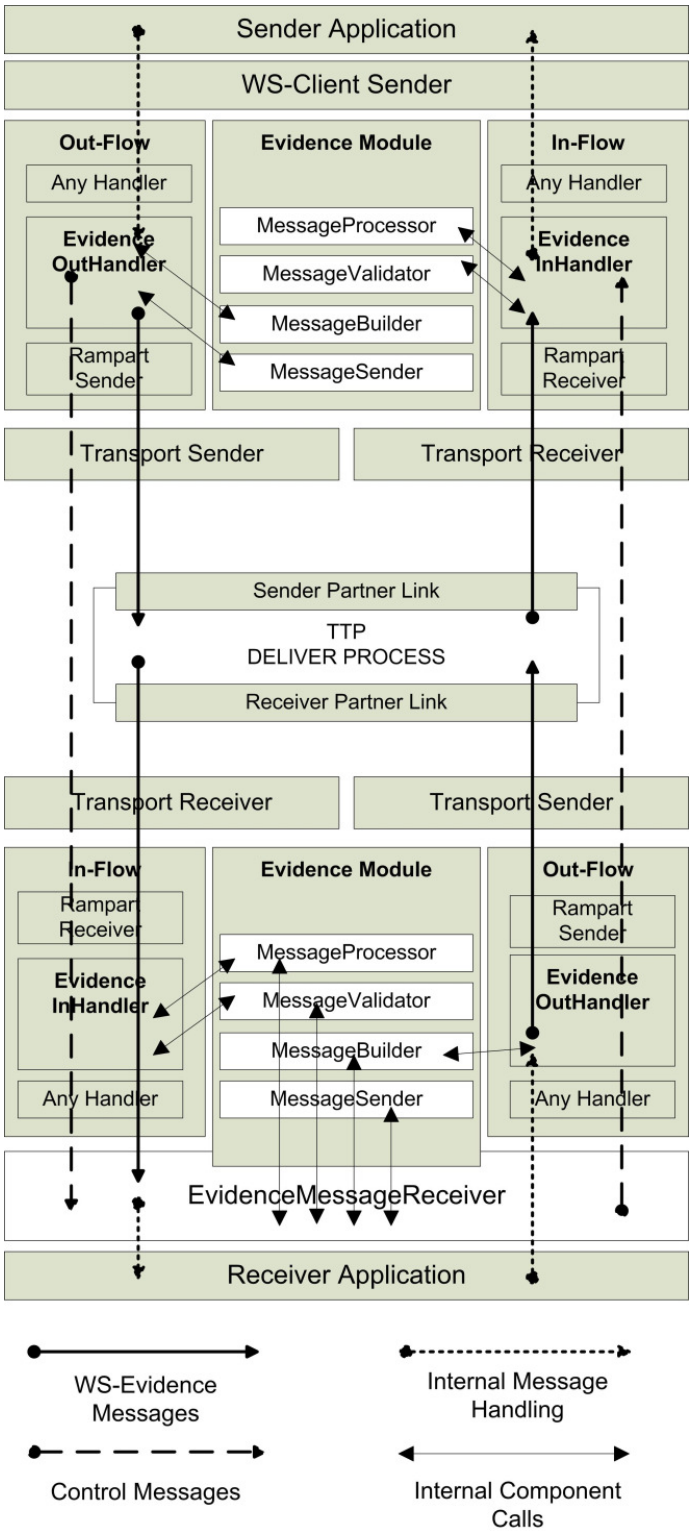


Figure 4.3. Evidence Module Architecture

EvidenceOutHandler captures outgoing messages before being sent by transport sender. It may either create a new evidence session related to the application message if there is none yet, or call the relevant component to craft the message.

EvidenceMessageReceiver obtains control and acknowledgement messages and responds with the appropriate message using components in charge. However, applications messages are processed, extracted here, and released to business logic in a form in which they are expected from the actual endpoint.

MessageSender pushes additional messages into out flow. Message processors call it when an outgoing message is to be sent.

MessageBuilder builds messages either from scratch (e.g. additional messages) or extends existing ones with new parts. EvidenceOutHandler and MessageSender mostly call this component through a builder factory to instantiate correct message builders according to message type to be sent out. Builders utilize a set of Axiom API to create/alter SOAP parts and messages. They also employ signing mechanism using xml-dsig specification.

MessageProcesssor processes the incoming messages, such as extracting the inner application envelope, to invoke the actual receiver application operation**.** EvidenceInHandler and MessageReceiver mostly utilize this component. Like message

builders, every message type has its own processor instance obtained from a factory object. Very similar to message builders, processors also use Axiom API to parse and modify the SOAP messages

MessageValidator is in use for verifying digital signatures that are underpinning the evidence mechanism proposed by WS-Evidence. Like message builders, it implements this mechanism by pursuing the xml-dsig specification [26].

Having taken control, my module sends and receives *WS-Evidence messages* via TTP or directly from/to other endpoints in accordance with the protocol selected for the session. Here, I describe how this architecture takes part in generating evidences, as a layer mapping critical activities in the architecture to protocol runs by implementing In-line and Off-line TTPs.

4.4.2.1. Inline TTP

Three entities involved in inline TTPs are *sender*, *receiver*, *and a TTP*. Sender and receiver sides have evidence modules that act as agents that generate evidence messages at endpoints. Figures 4.4 and 4.5 show the relevant steps in a UML (Unified Modeling Language) sequence diagram [47].

Two Way Implementation:

1. The evidence module in the Sender side intercepts (in *EvidenceOutHandler*) the request of an envelope and pauses the message context.
2. Creates (using *MessageBuilder*) and sends (using *MessageSender*) a CreateSessionRequest to the receiver web service for the target operation.
3. The evidence module in the receiver side receives (in *EvidenceMessageReceiver*) the message and creates and sends a response message back to the sender. It also creates a session.
4. Sender's evidence module builds TwoWay1st (<#session|#message|#signature$_{Sender-K}$(#session|"1"|# env)>) from the message context paused and sends it to TTP.
5. The TTP recieves TwoWay1st, stores the message, forwards it to the receiver, and starts a timer.
6. Receiver's evidence module intercepts (in *EvidenceInHandler*) the message, processes (e.g. validates using *MessageProcessor* and *MessageValidator*) and extracts the actual envelope to release it to the expected receiver operation.
7. Receiver application prepares a response message and sends it back to the sender.
8. Receiver's evidence module intercepts the response envelope, builds a TwoWay2nd (<#session |#message |#signature$_{Receiver-K}$(#session|"2"|# env)>) message and sends it back to the TTP. If the response TwoWay2nd cannot reach the TTP before timing out, then, Failure (<#session|#message |#signature$_{TTP-K}$ (#session|"-1"|# env)>) is signed by the TTP; it is stored and sent back to the Sender.

9. The TTP forwards TwoWay2nd to the sender; it also creates, signs, stores, and sends TwoWayAck (<#session|#message|#signature$_{TTP-K}$ (#session|"3" |#env)>) to the receiver.

10. Sender's evidence module intercepts the message, processes (e.g. validates) and extracts the actual envelope releasing it to the expected application. It also creates and sends a TerminateSession to the receiver web service for related session, thus, terminating the session.

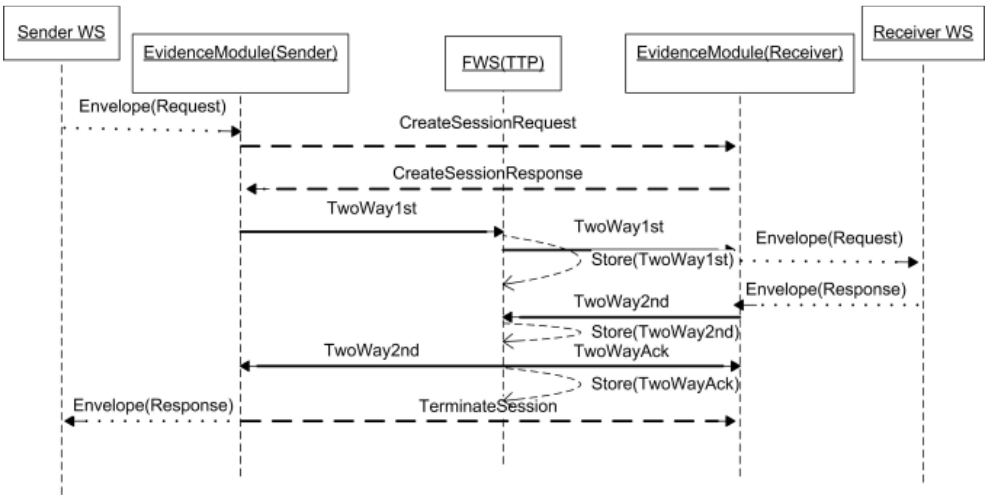


Figure 4.4. Inline TTP – Two Way

One Way Implementation:

The protocol run is similar with two way implementation in the first three steps. Starting at the fourth step, the protocol follows the steps below:

4. Sender sends a OneWay (<#session| #message |#signature$_{Sender-K}$ (#session|"4"|# env)>) message to the TTP.

5. The TTP receives the OneWay message, stores the message, and forwards it to the Receiver. It also creates, signs, stores, and sends OneWayAck (<#session|#message|#signature$_{TTP\text{-}K}$ (#session|"5" |#env)>) to the sender.
6. Receiver's evidence module intercepts the message, processes (e.g. validates) and extracts the actual envelope to release it to the expected receiver operation.
7. Sender's evidence module creates and sends a TerminateSession to the receiver web service for the related session, thus, terminating the session.

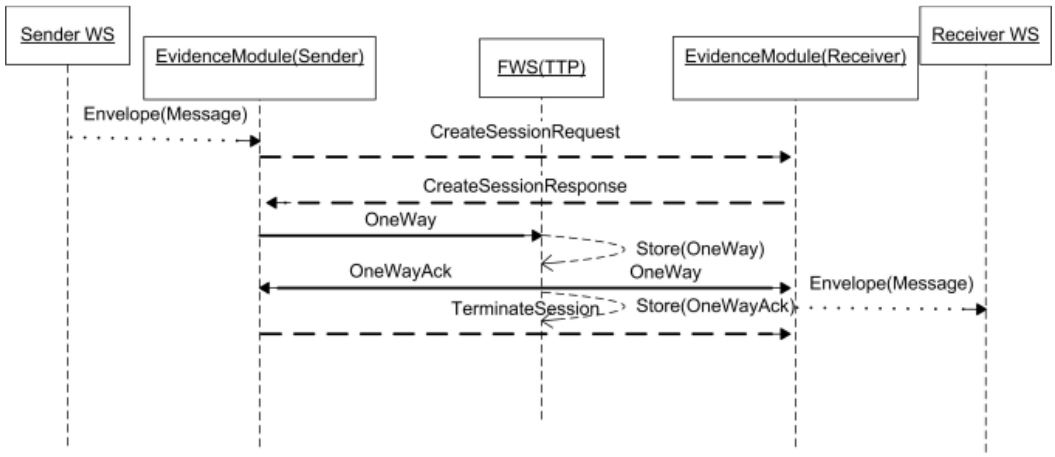


Figure 4.5. Inline TTP - One Way

4.4.2.2. Offline TTP

There are normally two entities involved in protocol run: *sender*, and *receiver*. [7] Proposes TTP involvement when any acknowledgments are not generated. This protocol may be chosen when there is a risk of performance bottleneck or man-in-the-middle attack on TTPs. In this case TTPs do not monitor the activity online, but knows the

session ID to collect evidence after the exchange. The corresponding one way and two way protocols are as follows. Figures 4.6 and 4.7 show the relevant steps in a UML sequence diagram [47].

Two Way Implementation:

1. Sender's evidence module intercepts the request envelope and pauses the message context.
2. Creates and sends a CreateSessionRequest to the receiver web service for the related operation.
3. Receiver's evidence module intercepts the message and creates and sends a response message back to the sender. It also creates a session and informs TTP with session information (CreateSession message).
4. Sender's evidence module builds TwoWay1st (<#session|#message|#signature$_{Sender-K}$(#session|"1" |# env)>) from the message context paused and sends it to the receiver directly.
5. Receiver's evidence module intercepts the message, processes (e.g. validates) and extracts the actual envelope to release it to the expected receiver operation.
6. Receiver application prepares a response message and sends it back to the sender.
7. Receiver's evidence module intercepts the response envelope, builds a TwoWay2nd (<#session |#message |#signature$_{Receiver-K}$(#session|"2"|# env)>) message and sends it back to the sender.

8. Sender's evidence module creates, signs, stores, and sends TwoWayAck (<#session |#message| #signature$_{Sender\text{-}K}$ (#session|"3" |#env)>) to the receiver.
9. Both endpoints' evidence modules send evidence messages (TwoWay2nd for receiver and TwoWay1st|TwoWayAck for sender) they collected during the session run to the TTP.
10. Sender's evidence module creates and sends a TerminateSession to the receiver web service for the related session, thus, terminating the session.

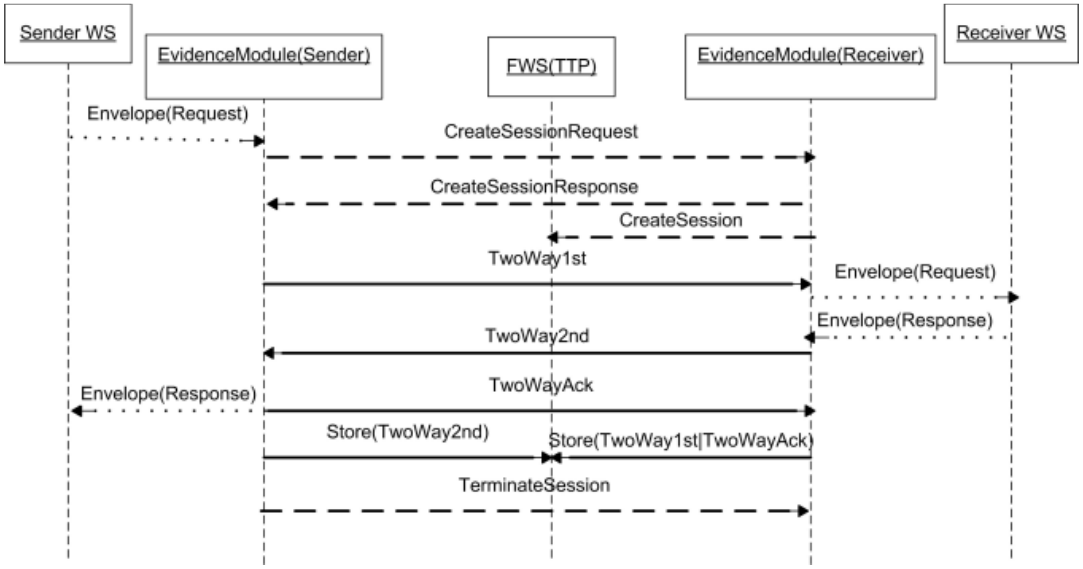


Figure 4.6. Offline TTP - Two Way

One Way Implementation:

The protocol run is similar to the two way implementation in the first three steps. Starting at the fourth step, the protocol follows the steps below:

4. Sender sends OneWay (<#session|#message| #signature$_{Sender\text{-}K}$(#session|"4"|# env)>) message to Receiver.
5. Receiver's evidence module intercepts the message, processes (e.g. validates) and extracts the actual envelope to release it to the expected receiver operation.
6. Receiver creates, signs, stores, and sends OneWayAck (<#session|#message|#signature$_{Receiver\text{-}K}$ (#session|"5" |#env)>) to the sender.
7. Both sender and receiver send the evidences (OneWayAck by Sender and OneWay by Receiver) they gathered during the invocation time to TTP.
8. The evidence module in the Sender side creates and sends a TerminateSession to the receiver web service for the related session, thus, terminating the session.

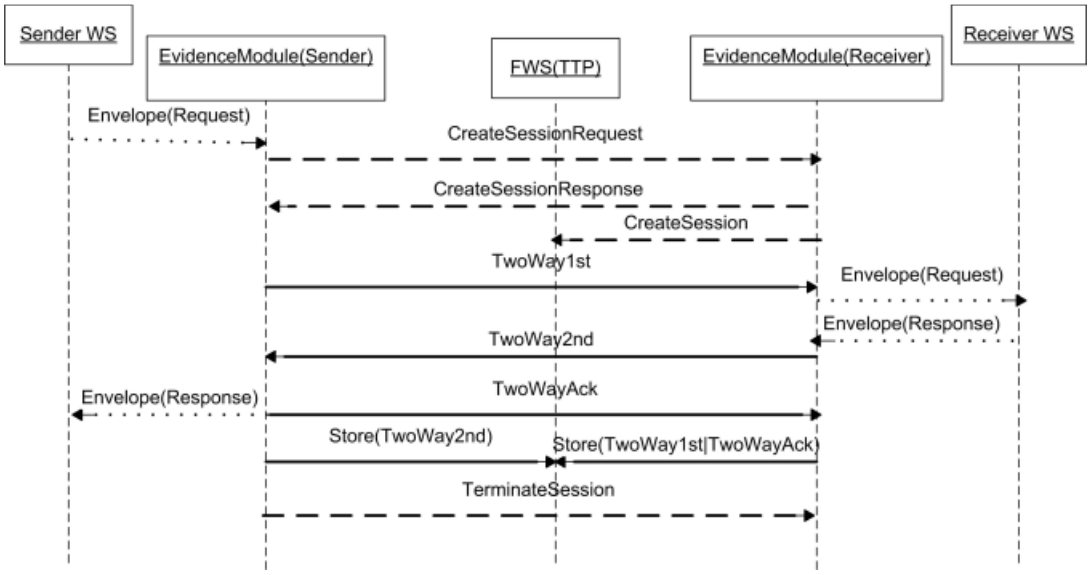


Figure 4.7. Offline TTP - One Way

4.5. Building Evidences

4.5.1. Pair-wise Evidence

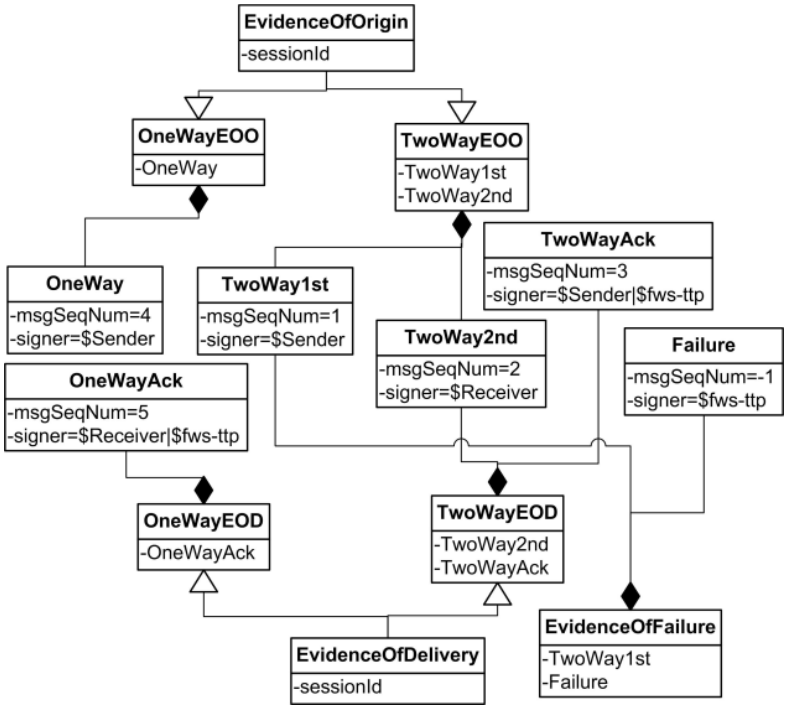


Figure 4.8. Building Blocks of Pair-wise Evidence

As detailed through the protocol runs above, WS-Evidence messages contain signatures as evidence, thereby constituting various pair-wise evidences that may be of interest to endpoints. Figure 4.8 illustrates how WS-Evidence messages constitute pair-wise evidences through a UML class diagram [47].

4.5.2. Derived Evidence

In this case, endpoints can gather evidences from TTPs at any time rather than service invocation time. In order to generate evidences from TTPs for specific time intervals I rely on the evidences stored at TTPs. Evidences gathered this way can be used by a web service to exculpate from accusations. Depending upon the service level agreements, the number of evidences would increase. I here give two examples assuming endpoints have a time-out agreement in the first case and scheduled invocations in the second.

Evidence of Availability (EOA) [7]: Availability of a web service for a certain time interval refers to the fact that TTP has not produced any *Failure* (refer to step 8 in the Two Way implementation) message in that period. Either against counterfeit Failure evidence or for no specific reason, any web service may request availability evidence in order to exculpate itself at various stages. To do so, it prepares an EOARequest containing a start time and an end time and sends it to the FWS. FWS checks the records for Failure evidences which prove that the web service did not respond to some requests where it meets the time criteria in EOARequest. If FWS encounters no evidence then it produces new evidence of availability proving that the service was available at that time interval, signs it and sends to the requester service in EOAResponse message.

Evidence of Violation (EOV): Web services make agreements at the service level. For example, some services may need to be updated at certain times with one way messages.

They define this requirement in an agreement. Services that are not invoked at the scheduled time interval may request an evidence of violation in order to exculpate themselves if any incident occurs that stems from the absence of this invocation. The web service in question prepares an EOVRequest message including the start time and end time values defining time interval and also sender web service's identity. FWS checks its records for submissions originating from that sender and targeted at the web service in question where the time criteria are met in the request. If FWS encounters no records of submission while meeting the criteria then it produces an evidence of violation message, signs it and sends to the requester service.

4.6. Related Work

Rather than applying inline or offline TTP notion, [48] propose a novel language, BP-Mon, for observing business processes in BPEL. One can translate BP-Mon queries into BPEL processes so that they can run those monitor processes on the same execution environment. While this provides capability of observing the details of internal runs, this however, lacks non-repudiation, thereby introducing less sound evidences from forensic perspective.

Ardissono [49] proposed a framework to support monitoring choreographed services and detection of faults along with notification of affected parties. The framework is based on WS-Coordination. There are similarities between the framework and WS-

BusinessActivity: (1) The framework notifies the monitor about the choreography paths traversed during the execution of overall choreography model. (2) The monitor uses choreography specs to inference the possibility of how successful the execution of choreography model through the states of web services involved in the choreography. It measures the portion of the choreography that has been completed at a given time. While neutrality can be reached at some level the framework lacks non-repudiation; and endpoints can always act unlike their notifications which cannot cryptographically be detected.

As mentioned earlier WS-NRExchange [9] influenced the model I employ for pair-wise evidence generation with some differences. Their framework is designed to run with many non-repudiation protocols [8]. Their work, however, reveals that they had little success in separating WS-NRExchange from lower layers; they are bound to Java RMI. Axis2-based services, however, successfully separates the lower layer (HTTP, SMTP, TCP, etc) from the SOAP layer. Reference [9] only proposes delivering evidences to the related parties, but not using stored evidences in trusted agents for further evidence derivation as described in Section 4.5.2 Furthermore, they do not address choreographed, composed services.

Although Herzberg et al. [7] introduced the notion of an Evidences Layer for e-commerce transactions; they never mention how endpoints run such protocols. Their work seems sound in separating evidences layer from business logic, however, it lacks explaining

how such layer could be attractive for e-commerce systems in terms of integration costs, such as alteration in their existing code.

I use trusted third parties for pair-wise evidence generation as did others [40, 41] through their studies. Onieva [40] gives the intermediary usage perspective in the implementation of inline TTPs for e-commerce transactions, and supports multi-recipient cases through these intermediaries, but not as a complete separate layer. Bilal [41] uses BPEL for non-repudiation protocol implementation in web services, but does not use TTP; his method thereby lacks the capability of producing derived evidences based on log records.

I employed handlers over existing web service architecture in order to design Evidence Module. Although there are many vendors or platforms [27,28,29,30] that provide infrastructure for chaining handlers in their web service stack, with Axis2, Apache et al. [31] have implemented some ws-* standards, such as Rampart for WS-SecureConversation, Rahas for WS-Trust, Sandesha2 for WS-RM, and Kandula for WS-Coordination [42].

As mentioned earlier WSLogA [6] tracks web service invocations by logging service invocations using SOAP intermediaries. Therefore, it captures the external behavior of service invocations. The main purpose of WSLogA is to provide feedback to business organizations by comprehensively logging services usage records. However, they never produce non-refutable evidences.

4.7. Conclusions

Previous Chapter showed that neutral, tamper resistant evidences can be used for dispute resolution and mal-actor participation among web services. In this chapter, I showed how web services can use such an evidence resolution framework. This can be introduced to web services in a systematic way that would eliminate custom re-engineering web services.

CHAPTER 5

DETECTING ILLEGAL BUSINESS SCHEMES IN CHOREOGRAPHED WEB SERVICES: THE PONZI/PYRAMIDAL CASE

5.1. Introduction

Dynamic service invocations and generating content specific operations among choreographed web services are being deployed across many industries, creating service inter-dependencies between web services. These dynamic service inter-dependencies can be exploited to create a new class of misuses. Some of them exploit the infrastructural dependencies of the services themselves and others use them to create illegal business schemes. This chapter focuses on detecting a special type of the second kind: namely Ponzi/Pyramidal investment schemes created to defraud unsuspecting investors.

Implemented illegal business schemas are difficult to detect because most of them are similar to legal business schemas for a microscopic observer, and become apparent only through a macroscopic view. Thus, they can elude local monitoring of web transactions. Among the plethora of possible illegal business schemes, I choose to study two popular ones: namely, pyramidal and Ponzi schemes. These are difficult to differentiate from

Multi Level Marketing schemes that either run their own business or invest in others. However, the basic dynamics of Ponzi/pyramidal schemes is *rob Paul to pay George* [50]. Charles Ponzi [51] was the first to run such a scheme. For many years, he collected money that promised return on investment within 90 days for customers who enrolled others in the scheme. He returned the money to the early investors using the money invested by late joiners. Other than running this scheme, he never ran any business nor invested in other businesses and, consequently, did not produce any profit nor incurred any loss. Charles Ponzi was the first, but not the last, to run such a scheme. Even today, many incidents, including those that conduct Internet based Ponzi schemes, are being investigated [52, 53]. In 2006, 25,000 web sites suspected of promoting a pyramid scheme were investigated to shut down by the Securities Exchange Commission (SEC) [54].

Table 5.1. Pyramid Scheme (Adapted from [55])

	# Payment of $400		
Level 1 $100 x 3 = $300	#	#	#
Level 2 $30 x 9 = $270	# # #	# # #	# # #
Level 3 $30 x 27 = $810	# # # # # # # # #	# # # # # # # # #	# # # # # # # # #
Level 4 $30 x 81 = $2430	# # # # # # # # # ...	# # # # # # # # # ...	# # # # # # # # # ...
.....			
..... Level 21	10460353203 #s Even Larger than World Population		

Classic pyramidal schemes also use the same principle, and are shown in Table 5.1. A smart con-artist, or orchestrator, that originates the scheme convinces the top-level

investors by promising them large returns on investment, as do subsequently recruited investors to their potential customers. Table 5.1 shows the activities of an investor and his recruits. My example is organized as a 4-level payment scheme with a span of 3: that is, only up to 4 levels of ancestral recruiters will gain a profit from investments, and every recruit at any level recruits 3 others. The first level investor recruits 3 others and gets paid $100 per recruit. These recruits are in-turn expected to recruits 3 others ad infinitum, thereby building a recruit tree. The original investor will be paid $30 (not $100 this time) per sub-recruits at levels 2, 3 and 4. The original investor does not get involved with sub-recruits at levels 2, 3, and 4 and does not profit from investors beyond level 4. The orchestrator considers recruit activity to be complete when he receives an investment of $400 from an investor. And, therefore, the orchestrator pays his recruiters $100 for 1st parent or $30 for the 3 immediate ancestral recruiters, thus paying only $190 at most in return on one investment of $400. Consequently, any investor makes $3810 (2430+810+270+300) as return on investment (ROI) to a promoter if s/he successfully recruits 3 other promoters and they subsequently become successful in recruiting a minimum specified number of other investors (3 in this scheme) all the way down to the 4th level. However, because of active promotions, the scheme disperses fast; but, it becomes unsustainable at some later time. Then, many promoters are unpaid due to the lack of new investors [55].

As financial institutions and their business partners are moving to service oriented architectures such as dynamic brokering over investment firms or stock markets that are

using semantic web services are holding more promise. Nevertheless, these techniques provide a way to orchestrate illegal and unfair business practices. Therefore, in order to detect such illegal schemes, one needs to have a comprehensive non-repudiable perspective of complex multi party transaction models. A multi party communication can arise in two possible ways among dynamic web services. In the first, a static communication pattern is specified, and all participants follow this prior-known pattern. In the second, web services discover and transact with other services, thereby dynamically creating choreographies that were unknown a-priori. Consequently, in order to discover illegal activity, one has to seek illegal business transactions using both kinds of choreographies.

The rest of the chapter is written as follows. Section 5.2 shows how web choreographies can be misused. Section 5.3 explains the framework to collect evidences of pair-wise web communications in order to detect web choreographies. Section 5.4 defines the objectives satisfiable from collected evidence. Section 5.5 describes how choreographed global behavior can be derived from local observations. Section 5.6 formally defines Ponzi schemas. Section 5.7 describes how global misbehavior can be mined from a collection of local evidence of business transactions. Section 5.8 describes how to estimate damages resulting from a Ponzi Schema. Section 5.9 describes related work and Section 5.10 has my concluding comments.

5.2. Misusing Choreographies

5.2.1. Business Misuses

In business misuses, an orchestrator creates a large business scheme that abuses legal constraints in producing profits, without abusing the underlying choreography or attacking the infrastructure. Sometimes, a business level mal-actor is a partner in a choreography that deviates from the originally specified choreography. A choreography is said to *deviate from its specification* if one of the participants of the transaction does not behave as specified. Such deviations may provide an undue advantage to one partner over others. For example, a travel agency may favor recommending certain hotels or car-rentals over others that provide comparable or better value to tourists.

5.2.2. Service Misuses

These attacks exploit design flaws on static choreography models, where mal-actors abuse visible syntactic inter-dependencies of choreographed services.

A *dataflow attack* can be used to leak mal-code into partner services among regular data. Unless the recipient service checks for content, malicious data can pass between the systems. Chapter 3 describes a cross site scripting attack scenario that may leave evidence that can be used to identify a stepping stone as the attacker.

Instantiation flooding [56] generates a typical DoS attack on composed web service when a mal-actor repeatedly invokes a *receiver* process at a target web service. Such attacks affect mostly hierarchically composed web services where the flood of requests can deviate the state of the choreography engine (for e.g. BPEL runtime). Alternatively, an attacker can target a specific web service by using a partner in choreography as a steppingstone; where, on request, a process of the steppingstone may invoke a process at the target service. Flooding may crash the steppingstone and the target for which the attacker can blame the steppingstone.

5.3. The Evidence Generation Framework

This section briefly reviews an evidence generation framework proposed in previous chapter, and will be used to track business level choreographic misuses. This *Evidence Generation Framework (EGF)* shown in Figure 4.1, consist of three layers. The bottom layer (Pair-wise Evidence Generation Service – FWS-TTP) generates evidence for pair-wise interactions between web services. The middle layer (Evidence Derivation Service - EDWS) derives facts from available pair-wise evidence in order to refute or justify claims of agreement violations between communicating partner services. The highest layer (Comprehensive Evidence Generation Service - CEGWS) generates instances of requested choreographies from layer 2 and layer 3 data. The EGF provides on-line evidence generation and management capabilities to other web services as a web service itself. In order to use the services of EGF, other web services (referred to as member

services of the EGF) should integrate EGF with themselves using a centralized service access point. Thereafter, EGF acts as a trusted third party.

The EGF as a service receives and retains service requests and responds in a cryptographically secure manner, retains these correspondences in secure repositories, and provides them for dispute resolution and forensic investigations. EGF provides so called *evidence adapters* for all requests.

The previous chapter shows a prototype implementation of EGF layer 1, and presents many protocols based on One-Way and Request-Response message exchange patterns (MEP). Currently, EGFs provide evidence for *non-repudiation*, *fairness*, and *timeliness* using digital signatures to provide proof of receipt and delivery, link a message to its creator/sender, and provide message integrity. For accountability, EGF uses fair non-repudiation mechanisms that utilize Trusted Third Parties (TTP). Because, although there are fair exchange protocols for two participants that do not use TTPs (for e.g. Markowitch [8]), these protocols assume that the participants have prior knowledge of the message contents. I do not use them because web services may not always know expected message content. I require timeliness because of the time sensitive nature of most business transactions. I base evidence records on time observed at TTPs. EGF servers gather pair-wise transactional evidence that flows between sender and receiver web services, employing inline TTPs that use the *Simple Evidence Layer Protocol (SELP)* or offline TTPs using *Optimistic Evidence Layer Protocol* (*OELP)* of Herzberg's

[7]. SELP and OELP are two protocols used by end-points to obtain non-repudiable evidence by using a specific message format and digital signatures.

5.4. Evidence of Observed Interactions

Web services use many kinds of messages, such as one-way or request-response, in order to choreograph business processes among themselves that correspond to the four proposed WSDL operation types (in-only, out-only, in-out, out-in). Nevertheless, an external observer that is ignorant of the business processes can observe only One-Way and Request-Response MEPs, formalized in Definition 5.1.

Definition 5.1 (messages): A web-services message consists of the following components:

1. *Mandatory Fields:* of Sender and Receiver Time, where the first two are chosen from URLs and the last chosen from T.
2. *Optional Fields:* a finite set of attributes from a set A.
3. *Message content:* consists of strings from an alphanumeric set C. I use | to denote string concatenation and $sig_A(r)$ to denote the string obtained from signing r with A's key.
4. *Notation:* If m is a message, I denote "m.a" to be the value of the attribute "a" in message "m". For example, m.time is the value of the timestamp on m.

Definition 5.1 establishes notation to be used in describing the messages used to extract knowledge about externally observable facts about choreographies. Because different choreography specifications may select different labels for their identifier fields in order to bypass naming convention problems, I use XPATH expressions to specify ID values. Secondly, because any fabricator can produce messages, I rely on cryptographically secure messages to ascertain reliable evidence. The messages are collected to derive so called *evidence objectives* - claims that are to be substantiated or refuted using the collected evidence, such as the origin of generation, properties of messages, or intended recipient. Such evidence is generated from cryptographically secure messages. Sometimes objectives such as *evidence of delivery* or *evidence of non-availability* may require messages to be signed by a TTP.

Definition 5.2 (Primitive Evidence Objectives):

1. *Evidence of Origin:* A message m with origin A and content $r|sig_A(r)$ from A to B is said to provide evidence of origin.
2. *Evidence of Delivery:* A message m with content $ack|sig_{TTP}(ack|m)$, where TTP is a trusted third party or content $ack|sig_B(ack|m)$, and where B is m.recipient is said to provide evidence of delivery.
3. *Message Evidence:* An evidence of a message m is said to be a pair (m_1, m_2), where m_1 is an evidence of origin and m_2 is an evidence of delivery of m.

As definition 5.2 says, I require cryptographic evidences from a web service of a trusted third party for claims of origin and delivery. Previous chapter shows how Evidence of

Origin and Evidence of Delivery can be collected at TTPs using the WS-Evidence messages that are generated using non-repudiation protocols. These message evidences (MEs) are stored as Log Records (LRs) throughout the Evidence Generation Framework as described in Chapter 4. Because LRs may contain large volumes of data, I use Message Evidence Indexes (MEI) that refers to messages. Table 5.2 shows a sample index table, where the first column is an index for stored packets that have attributes of time, sender, message string, and content in subsequent columns. As shown, index 1 refers to a message with content <..invID..> sent by A to B.

Table 5.2. MEI Table

ID	Time	Sender	Receiver	Msg	Content
63..	21	A	B	r	<..invID..>
67..	22	B	C	m	”..”
68..	23	C	B	k	<..payID..>

5.5. Evidence of Choreography

Although there are many choreography specification languages, my objective is to recognize choreographies from externally observed messages. Hence, I need to develop a method to abstract the relevant properties of these messages. This section develops the basic notation used to specify potential relationships between messages.

Authors [57, 58] describe three different artifacts to correlate messages among web services, keys, properties, and time intervals, listed from the strongest to the weakest with respect to relating messages. Time intervals use message time stamps and attribute values

to relate messages. A key is a pre-chosen attribute of a message that can uniquely identify a scenario in the given application domain. I use all three forms of message correlation. Consequently, I define three kinds of choreography patterns. I use the following notations in order to define my notion of choreography patterns.

Definition 5.3 (Message Attribute Equations): Suppose that $(a_1,\ldots,a_n)$ is a vector of message attributes and m, m' are messages. Then an equation of the form $m.a_i=m'.a_j$ is said to be a message attribute equation. A finite collection of such equations where all equations are chosen from $(a_1,\ldots,a_n)$ is said to be a set of attribute equations. If comparing only one attribute value between any two chosen messages can uniquely relate them, then the set of equations is said to be *set of key equations*.

For example, m_1.sender = m_2.reciver and m_1.domain = m_2.domain are two message attribute equations. But m_1 and m_2 are compared using two attributes from every message. Thus, they do not form a set of key equations. Assume that an investor can be identified from a field, say *id* issued by an investment company instead of using a global identifier. For example, it can use 3 as an investment id for two different investments sent to different investment companies, such as company A and B, yet cannot use 3 for other investments sent to the same company, say A. To define a set of key equations for this case, I assume there is an invest message sent to a company that has only the *id* attribute and the authorization message sent to the bank has attributes invComp and invID. Hence

invest.receiver == authorization and invComp and invest.id ==authorization.invID are key equations.

Definition 5.4 (Time-based Choreography Patterns): I recursively define the set of time-based choreography patterns $\mathbb{P}_{time}$ as follows:

1. If m is a message, then $\{m\} \in \mathbb{P}_{time}$. That is, a set consisting of a message is considered a time-based message pattern.
2. Suppose that $x \in \mathbb{P}_{time}$ and $y \in \mathbb{P}_{time}$ are time-based patterns. Then, $x \otimes_t y \in \mathbb{P}$ is said to be a time based concurrency pattern whose members are $\{(a,b) | a \in x \text{ and } b \in y\}$.
3. Suppose that $x \in \mathbb{P}_{time}$ and $y \in \mathbb{P}_{time}$ are time-based patterns. Then, $x \cup_t y \in \mathbb{P}_{time}$ is said to be a time-based choice pattern whose members are $x \cup_t y$. That is, the pattern $x \cup_t y$ has members of x or members of y.
4. Suppose that $x \in \mathbb{P}_{time}$ and $y \in \mathbb{P}_{time}$ are time-based patterns. Then, $x;_t y \in \mathbb{P}$ is said to be a time-based sequencing pattern whose members are $\{(a,b) | a \in x, b \in y, a.time \leq b.time\}$. That is, the pattern x must appear before the pattern y.
5. Suppose that $x \in \mathbb{P}_{time}$ is a time-based pattern. Then recursively define the time-based recursive patterns $x^{1,t}=x$, $x^{(n+1),t}=f(x^{n,t},x)$, where the function f defines a choreography

for $x^{(n+1),t}$ from $x^{n,t}$ and x using the operators $\otimes_t$, $\cup_t$ and $;_t$. Then define $x^{*,t}$ to be LFP(f,x). That is, $x^{*,t}$ is the least fixed point of the inductive definition.

Definition 5.5 (Property-based Choreography Patterns): I recursively define the set of property-based choreography patterns $\mathbb{P}_{prop}$ as follows:

1. If m is a message, then $\{m\} \in \mathbb{P}_{prop}$. That is, a set consisting of a message is considered a property-based pattern.
2. Suppose that $x \in \mathbb{P}_{prop}$ and $y \in \mathbb{P}_{prop}$ are property-based patterns, and $(a_1,\ldots,a_n)$ is a vector of message attributes. Then, one can say that $x \otimes_p y \in \mathbb{P}_{prop}$ is a property based concurrency pattern whose members are $\{(a,b) | a \in x, b \in y, a.a_{i1}=b.a_{j1}, \ldots a.a_{im}=b.a_{jm}\}$. That is, the pair of messages (a,b) are chosen from patterns x and y satisfying the condition that satisfy a set of attribute equations chose from a vector $(a_1,\ldots a_n)$ of attributes.

   Suppose that $x \in \mathbb{P}_{prop}$ and $y \in \mathbb{P}_{prop}$ are property-based patterns. Then, $x \cup_p y \in \mathbb{P}_{prop}$ is a property-based choice pattern whose members are chosen from the set $x \cup_p y$. That is, a property-based pattern $x \cup_p y$ has members of x or members of y.

3. Suppose that $x \in \mathbb{P}_{prop}$ and $y \in \mathbb{P}_{prop}$ are patterns and $(a_1,\ldots a_n)$ is a vector of message attributes. Then, $x;_p y \in \mathbb{P}_{prop}$ is a property-based sequencing pattern whose members are $\{(a,b) |\ a \in x,\ b \in y,\ a.time \leq b.time, a.a_{i1}=b.a_{j1}, \ldots\ a.a_{im}=b.a_{jm}$. That is, the pattern x must appear before the pattern y and the patterns must satisfy the specified set of attribute equations $a.a_{i1}=b.a_{j1}, \ldots\ a.a_{im}=b.a_{jm}$.
4. Suppose that $x \in \mathbb{P}_{prop}$ is a pattern and $(a_1,\ldots,a_n)$ is a vector of message attributes. Then, recursively define the time-based recursive patterns using the equations $x^{1,t}=x$, $x^{(n+1),p}=f(x^{n,p},x)$ where the function f defines a choreography for $x^{(n+1),p}$ from $x^{n,p}$ and x and a set of message equations, say E using the operators $\otimes_p$, $\cup_p$ and $;_p$. Then, define $x^{*,p}$ to be LFP(f,x,E). That is, $x^{*,p}$ is the least fixed point of the inductive definition.

Definition 5.6 (Key-based Patterns): Replacing "property-based equations" with "key-based equations" in definition 5.5 gives key-based choreography patterns.

Definition 5.7 (State of a Pattern): If $S=S_1;_x S_2$, where x could refer to t (i.e. time based) or prop (i.e. property based), then S_1 is said to be a state of a choreography.

For example, $(m_1 \otimes_p m_2)$ is s state of the choreography patterns $((m_1 \otimes_p m_2);_p m_3);_p m_4$ and it represents a partial computation of the complete choreography pattern.

Figure 5.1 illustrates the choreographies of *invest* and *pay* transactions. In state 1, the *Invest* message (1) has been sent to the Investment Company and the Authorize message (2) has been sent to the Bank. When the Bank sends the *Confirmation* message (3) to the Investment Company, then the state of the choreography becomes 2. The *Deliver* message (4) is the last observed message of this choreography, and its receipt represents state 3. The messages of choreographies are defined as follows:

invest.sender = Investor, invest.recivever=InvComp, invest.content =

$invest|sig_{Investor}(Invest)$

authorize.sender=Investor, authorize.reciever=Bank,

$authorize.content=authorize|sig_{Investor}(authorize)$

confirm.sender=Bank, confirm.reciever=InvComp,

$confirm.content=confirm|sig_{Bank}(confirm)$

deliver.sender=InvCompany, deliver.reciever=Investor,

$deliver.content=deliver|sig_{InvCompany}(deliver)$

Then define choreographies for investing and paying (shown in Figure 5.1) as:

1. Investing = $(invest \otimes_p authorize);_p$ $confirm;_p$ deliver where the message attribute equations are given as:

   invest.sender=authorize.sender and

   authorize.reciever=confirm.sender and

   confirm.reciever=invest.sender

2. Paying = (pay$\otimes_p$authorize);$_p$ confirm;$_p$ acknowledge where the message attribute equations are given as:

   pay.sender=authorize.sender and

   authorize.reciever=confirm.sender and

   confirm.reciever=invest.sender

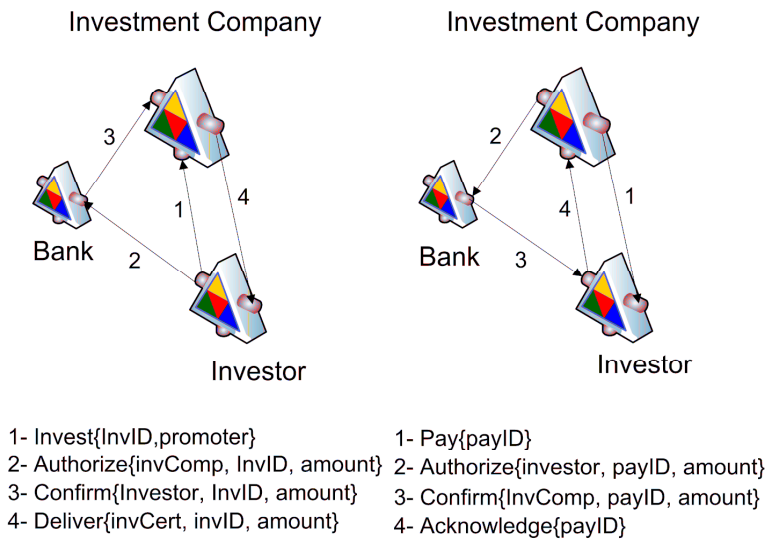


Figure 5.1. Sample Invest and Pay Choreographies

Deriving such choreographies from external observations can reveal some illegal business transactions or reveal some illegal parts of large financial businesses. Additionally, members of a transaction can derive the actual instance of the transaction that they participate in and observer the behavior of their partners. For example, a party B authorizes the Bank to release some amount of money to party A. The Bank, however, releases the money with some additional fees each time, thereby reducing the money

deposited in A's account. In order to avoid being a party to such deceptions, any service provider should be able to obtain all instances of participating choreographies.

5.5.1. Pattern Directed Choreography Mining

In this section I show how to mine choreography instances of given patterns from log records of all observed web transactions using streamSQL [59] and a StreamBase platform [60]. StreamSQL is an event pattern language that can be used to define queries over streams of data and StreamBase is an event processing platform that can run those queries over input source from a file or a database and produce outputs. StreamSQL has several commands, of which I describe a few that I used. **CREATE INPUT STREAM** creates data streams from a named file pre-configured in a known schema. **CREATE OUTPUT STREAM** creates an output stream pre-configured according to a schema. The **PATTERN** phrase is used to define the search criteria from multiple input streams. A **WITHIN** phrase is used to create the maximum size of a window that moves along a collection of *aligned* streams searching for a pattern.

```
GenereateCHOR-Investing
Description: Given the MEI table and the pattern, emits the appropriate messages pertaining
to instances of the pattern.
Input: MEIs are processed as multiple (four for this pattern) inputs
Output: CHOR Investing
1  CREATE INPUT STREAM MEI ($MEI schema);
2  CREATE OUTPUT STREAM InvestsOut;
3  CREATE STREAM InvestOut ;
4  CREATE STREAM AuthorizeOut ;
5  CREATE STREAM ConfirmOut ;
6  CREATE STREAM DeliverOut ;
7  SELECT * FROM MEI
8      WHERE msg=="invest" AND  receiver=="B"
9      INTO InvestOut
10     WHERE msg=="authorize"AND receiver=="Bank"
11     INTO AuthorizeOut
12     WHERE msg=="confirm" AND sender=="Bank" AND receiver=="B"
13     INTO ConfirmOut
14     WHERE msg=="deliver" AND sender=="B"
15     INTO DeliverOut;
16 SELECT AinvestB.*, AauthorizeBank.*, BankconfirmB.*, BdeliverA.*
17     FROM PATTERN (((InvestOut AS AinvestB AND AuthorizeOut AS AauthorizeBank)
               THEN   ConfirmOut AS BankconfirmB) THEN DeliverOut AS BdeliverA)
18     WITHIN 8 ON time
19     WHERE AinvestB.sender=BdeliverA.receiver AND AauthorizeBank.sender=AinvestB.sender
                AND AauthorizeBank.sender=BdeliverA.receiver
20     INTO InvestsOut;
```

Figure 5.2. Generate Evidence of Invest Choreography

The query in Figure 5.2 accepts MEI records in ascending order of their timestamp fields. In order to successfully process the pattern query, it produces four streams to process different message patterns such as *invest*, *deliver*. Appropriate predicates in each **WHERE** clause provide message pattern features, such as sender and receiver information. The **WITHIN** phrase uses a window of size 8 in the time filed of records in search of the specified pattern. The **SELECT** part gathers the required information about the detected pattern and emits the result to InvestsOut table specified as the output form

as a parameter to the **INTO** phrase. Figure 5.3 shows how an *Investing* choreography is mined from the 12 MEI records of Table 5.3.

Table 5.3. Sample MEI Records

ID	Time	Sender	Receiver	Msg	Content
23	1	D	A	purchase	productID=3
34	2	A	Bank	authorize	..invID=23...
45	3	G	F	book	Room=56
47	4	A	B	invest	invID=23
78	5	Q	O	invest	L1Promoter
79	6	P	O	pay	150
83	7	Bank	B	confirm	invID=23
91	8	H	L	invest	L2Promoter
93	9	B	A	deliver	..invID=23
96	10	K	L	pay	30
97	11	G	F	book	Room=57
98	12	V	X	pay	30

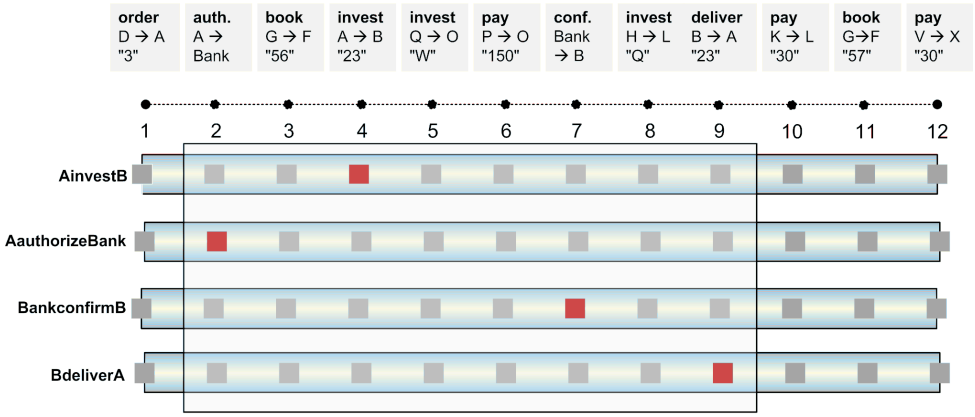


Figure 5.3. Generating Evidence for *Investing*

The query collects evidences for a specified choreograph pattern. Although the sliding window helps prevent the same record being counted to create scenario instances for more than one episode, I can use other message attributes to prevent this double counting. For example, I can do so by using relations definable using XPATH based functions as shown below right after line 20 of the query.

```
WHERE      GetXPATHValue(AinvestB.content,"../invID/") ==
           GetXPATHValue(AauthorizeBank.content,"../invID/") &&
           GetXPATHValue(AinvestB.content,"../invID/") ==
           GetXPATHValue(BankconfirmB.content,"../invID/") &&
           GetXPATHValue(AinvestB.content,"../invID/") ==
           GetXPATHValue(BdeliverA.content,"../invID/")
```

In addition, if I knew a key set of attributes (that is a set of attributes that uniquely identify a choreography pattern), those can be passed in order to identify the instances easily. Investor ID=23 would be an example of such a key attribute. Then, the query finds all evidence related to investor 23, as opposed to finding invest instances for all investors.

5.6. Evidence of Global Misuse

Mining choreographies that are created due to message contents from external observations requires linkage parameters, which can be derived from some externally invisible message content making them not very helpful for external monitors and auditors. One opportunity to obtain them arises when one of the unhappy participants, say, a victim, makes a complaint mostly about a financial loss. In that case, I use the following method to detect a spreading Ponzi like scheme.

1. Accept the complaints from victims.
2. Examine the content of specimen records such as a promoting or invest message provided by the potential victim.
3. Determine the parameters in evidence that can be linked together.
4. Detect choreographies and design them as (e.g. or pattern CHOR) dynamics of the algorithms.
5. Create the algorithms/queries.

6. Run the algorithms/queries in appropriate order and more than one when required.
7. Collect a set of comprehensive evidences and determine if the scheme is illegal and its effect over the network.
8. Broadcast an alert to current and potential future victims.

The method described above works for hierarchical schemes. Different methods could be applied for other types of schemes. I apply my heuristic method to generate the algorithms for a Ponzi-like scheme.

5.6.1. Ponzi Schemes over Web Services

Typical Ponzi-like schemes have three types of actors: a malicious investment service acting as the *orchestrator* (a.k.a. con-artist) and many investor services acting as *promoters* or *victims* (depending upon their investment and return rates). Figure 5 illustrates how such actors collaborate with each other to spread the financial scheme over a network of web services. The Investment Company, InvComp (a.k.a Orchestrator), promotes A (Promoter) to recruit B into its scheme (by promising a quick return on investment) and encourages it to promote other potential investors. This promoting activity (using Promote messages) *may not necessarily be observed through the records* because promoters may choose other means to convince investors. After being promoted, if A invests (Invest messages) in InvComp, then one can say that A has been recruited. Then A starts promoting InvComp to other investors in order to get a quick return on

his/her investment, and in the process recruits B. I recognize that B has been promoted by A because of the reference value in the content field of the invest message sent by B to InvComp. In accordance with the return policy of the scheme, InvComp awards A with a payment (evidenced in the Pay message from InvComp to A). The choreographies between Investment Company, Recruiters, and Recruitees spread in an investor web service network. When a recruitee cannot recruit enough investors, then it loses the money invested, thus being treated as victim (Victim). Figure 5.4 illustrates those complete and incomplete recruit choreographies.

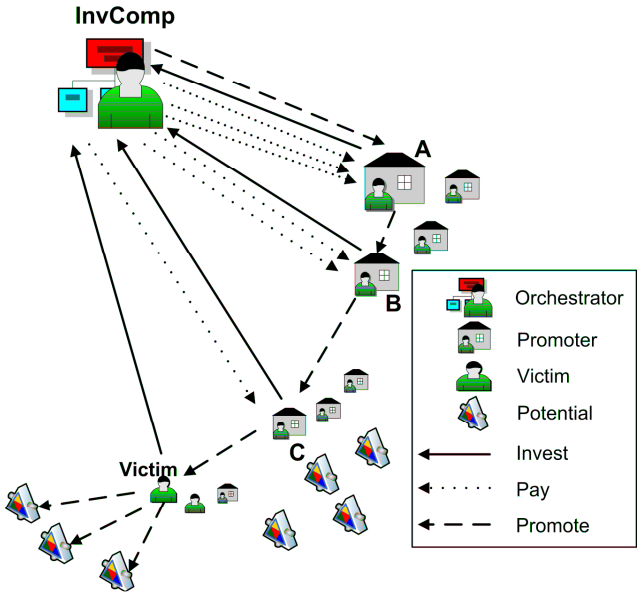


Figure 5.4. Ponzi-like Recruits over Web Services

However, I do not assume that I know the global scheme when mining, and instead assume that either a promoter web service identity or an invest message is submitted to a law enforcement agency by a victim. Following the heuristic algorithm presented above,

I can find invest, pay, and promote messages that contain attributes that refer to each other, thus collaborating with each other in a pervasive manner.

5.6.2. Pattern Discovery

Here I describe how to discover the patterns that will help in creating comprehensive evidence of illegal business schemes. Following the heuristic algorithm above, let us assume that a victim brings an *invest* message that contains a *promoter* web service. A query on evidence repository could show that it receives pay messages from other web services and also have sent an invest message to the same alleged investment company's web service. A set of sample records of invest and pay method are shown in Table 5.4.

Table 5.4. MEI Tuples Featuring a Misuse Scheme

ID	Time	Sender	Receiver	Msg	Content
...					
	45	B	InvComp	invest	Promoter=A
	55	InvComp	A	pay	150
...					
	67	C	InvComp	invest	Promoter=B
	76	InvComp	B	pay	150
	78	InvComp	A	pay	30
...					
	87	Victim	InvComp	invest	Promoter=C
	89	InvComp	C	pay	150
	92	InvComp	B	pay	30
	104	InvComp	A	pay	30

Given the records in Table 5.4, anyone can observe the pattern that keeps the fraudulent activity alive, where the invest messages are linked by *sender* header fields and *promoter* content fields. That is, every promoter web service in an invest message is being paid

right after this invest message. Now, it is time to correlate MEs to conclude that the promoter, victim, and the orchestrator may be involved in a hidden *recruit* choreography. In cases where the promoting activity does not involve a web service message, I can include those message or choreography patterns as part of the recruit activity. I name this content-based choreography as *recruit*. The pattern **I** as the signature of "rob Paul to pay George" activity and pattern **II** as the link between the recruiter and recruitee, thereby enabling the mining of recruit paths to create recruit trees from MEI records. For simplicity, I define patterns succinctly; but, more complex patterns may include pay messages, adding complexity to queries.

Table 5.5. Ponzi Scheme of Fan 1 and Depth 1

I	invest;$_p$ pay where invest.sender=A and invest.reciever=B and pay.sender=B and pay.reciever=C and invest.prometer=pay.reciever
II	invest1;$_p$ invest2 where invest1.sender=C and invest1. receiver=B and invest2.sender=A and invest2.reciever=B and invest2.promoter= invest1.sender

Table 5.4 shows that investors subsequently play a promoter's role within a subsequent recruit choreography pattern making a recursive investment scheme. For example, notice that investor B sends an invest message to the InvComp at time 45. At time 67 another investor, C, makes a reference to B through its investment message and B gets a payment (see the pay message at time 76) from InvComp afterwards. The same choreography can be observed between C and other subsequent investors, revealing that they recruited other

investors as shown in later records. Thus, in order to detect such a scheme, I need to recognize these recursive investment and payback schemes. The recursive scheme creates a *recruit-tree* that joins a recruiter to all of its recruitees. By traversing a path in a recruit tree from a chosen (victim) node to the root of the tree (say *recruit-paths*), one could find the orchestrator. The path Victim-C-B-A in Figure 5.4 is such a recruit path.

I use the following notation to specify recruit trees formally. Choose any integer k (to be used as the fan out of the recruit tree). Then, all finite sequences of $\{0,1,\ldots,k\text{-}1\}$ are used as identifiers for web service nodes. I use the notation $k^{<\omega}$ to denote the set of finite subsequences of $\{0,1,\ldots,k\text{-}1\}$. For example, all binary sequences can be used to index trees with fan out 2, where the left child of node x_p is x_{po} and the right child is x_{p1}, where p is a finite sequence of integers $\{0,1\}$, i.e. $2^{<\omega}$. I also use the notation $p<q$ to denote that p is a subsequence of q where $p,q\in k^{<\omega}$. I denote the length of any $p\in k^{<\omega}$ by $|p|$. Now suppose $p\in k^{<\omega}$ where $|p|=m$ and $p=<p_0,..,p_{m-1}>$. Then, define the ith ancestors of p for $i\geq 1$ as ancestor(i) $=<p_0,..,p_{m-i}>$. Ø represents the empty string in $k^{<\omega}$.

Definition 5.8 (Recruit Trees of fan out *k* and depth *m*): Suppose I is an investment company web service. Inductively define active(n) for every integer *n* as follows:

1. active(0)={m} where m is defined as a message where m.sender=$P_{\emptyset}$, m.receiver=I (say Eq 1)

2. Suppose active(n) has been defined and $p \in k^{<\omega}$ with |p|=n. Then, for each $i \in \{0,1,\ldots k-1\}$ define active(p^i)=msg;$_p$PayBack(p^i) (say Eq 2) where msg satisfies the property msg.sender=P_{p^i} msg.reciever=I and msg.content=invest (say Eq 3) and PayBack(p^i) is of the form

   PayBack(p^i)=$msg_1 \cap_p msg_2,\ldots,\cap_p msg_m$ where every msg_i is of the form msg_i.content=pay, msg_i.sender=I, msg_i.receiver=$P_{ancestor(p,i)}$ (say Eq 4) for $i \leq l$.
3. Let active(n+1)=active(p^0)$\cup_p$,..,$\cup_p$active(p^(n-1))
4. Define a recruit tree to be active* =LFP(f,m,E) where the function f is defined in (1) and (2), the set of message equations E are defined in (1) and (2), and the message m is defined in (1).

Notation: I denote the class of Ponzi schemes of fan *k* and depth *l* and attribute equations *E* as *Ponzi(k,l,E),* where E is the collection of equations Eq 1, Eq 2 Eq 2 and Eq4.

Definition 5.8 provides a generic definition for Ponzi-like schemes where the number of recruits employed by any recruiter is limited to an integer *k* and the number of ancestors deriving a payback from the recruitment is at most an integer value *l*. As an explanation, the web service nodes are numbered by strings chosen from $\{0,1,\ldots,k-1\}$, resulting in trees where every node has at most n. Thus, the parameter p in Definition 8 is used to denote a path with |p| elements in such a tree. Thus, step 0, with the empty string Ø, represents the recruiter in item (1) of the definition. Item (2) of the definition assumes that the tree has been defined up to a path *p* of length *n* and finds its next level. This step consists of a sequential composition of two steps. In the first part, P_p sends messages to

each of its children to invest. Then, the second part of the choreography shows each of these children investing in I, followed by the investor I paying ancestors of these children. The ancestors that are being paid back are limited to at most *l* generations. Item (3) of definition 8 collects all possible paths that extend the tree to the next level n+1; and, Item (4) collects all sub trees with depth n+1. Thus, the pattern I in Table 5.5 is a Ponzi scheme where *n*=2 and *l*=1. That is, every recruiter recruits only two investors, and the only person that benefits from these recruits' investments are their recruiter.

5.7. Detecting Global Misuses

Here I introduce a special query based on StreamSQL that discovers the pattern **I** defined in Table 5.5.

	DetectRecruits **Description**: Glides over MEIs using window size 3 to detect *pattern I* along with the predicates specified in WHERE clause. **Input:**MEI tuples **Output:**Ponzi-like recruit MEI pairs
1	**CREATE INPUT STREAM** MEI ($MEI schema);
2	**CREATE OUTPUT STREAM** PonziDetectOut ;
3	**CREATE STREAM** InvestFilterOut ;
4	**CREATE STREAM** PayFilterOut ;
5	**SELECT** * **FROM** MEI
6	**WHERE** msg=="invest" **INTO** InvestFilterOut
7	**WHERE** msg=="pay" **INTO** PayFilterOut;
8	**SELECT** "Ponzi-like recruit" **AS** detected, invest.time **AS** investTime, pay.time **AS** payTime,
	pay.receiver **AS** recruiter, invest.sender **AS** recruitee
9	**FROM PATTERN** (InvestFilterOut **AS** invest **THEN** PayFilterOut **AS** pay)
10	**WITHIN** 3 (days) **ON** time
11	**WHERE** invest.receiver==pay.sender **AND**
	regexmatch(".*"+"promoter="+pay.receiver+".*", invest.content)
12	**INTO** PonziDetectOut;

Figure 5.5. Detecting Recruits of Ponzi Schemes

DetectRecruits query in Figure 5.5 accepts MEI records in ascending order of timestamps. In order to successfully process the pattern query, it filters the records into two, invest and pay employing the predicates defined in line 6 and 7. This will allow the pattern to employ the appropriate template (see **THEN** phrase) in line 9. That is, invest messages are expected before pay messages. Predicates defined in line 11 say that the receiver of the invest message should be equal to the sender of the following pay message and the promoter value in the content of the invest message should be the receiver of the following pay message. The window size is set to 3 in line 10. The **SELECT** part gathers the required information about the detected pattern and emits the result to the **PonziDetectOut** table. Figure 5.6 shows how two Ponzi-like recruits are detected using a window of size 3 from a collection of 9 MEIs. Figure 5.6 shows how messages in a MEI table come into the query and are processed as two separate streams shown as pipes, where transparent rectangles represent two different snapshots of the query window, one that arrived at time 5 and the other that arrived at time 8.

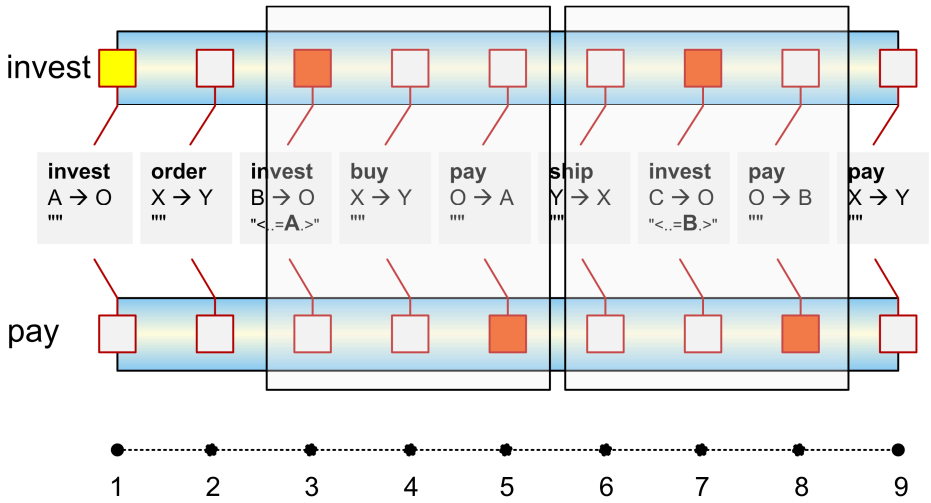


Figure 5.6. Detecting Ponzi-like Patterns

Detecting few Ponzi-like recruits may give little confidence in declaring that a Ponzi scheme was detected. In order to increase the confidence, I can define some minimum support value as threshold, thus alerting only when the threshold is met. The query below can be added to strengthen the previous detecting query along with a predefined minimum support value. **AlertRecruits** query alerts each time at least 6 Ponzi-like recruits are detected over the output of the previous query. **AlertRecruits** can be added to **DetectRecruits** query to decrease the number of false positives.

	AlertRecruits **Description**: Counts over detected Ponzi-like recruits using window size 6 as minimum support. Emits Ponzi alerts when minimum support is reached **Input:**PonziDetectOut from DetectRecruits **Output:**Ponzi alerts
1	**SELECT** "Ponzi Alerts", count() **AS** minSup
2	**FROM** PonziDetectOut [**SIZE** 6 **TUPLES**]
3	**INTO** PonziAlerts;

Figure 5.7. Enhancing Ponzi Detection

5.8. Generating Comprehensive Evidence

This section shows how to detect the orchestrator, or the earliest known recruiter, of a Ponzi schema by climbing any recruit path towards its beginning. Then one can follow all possible paths that originate at the detected recruiter and discover all others that invested in the Ponzi scheme. This can be found by creating a choreography defined as follows:

Suppose the complaint brings the invest message *msg* and it is known that the investment company used by all participants is I, then define ancestorChain(n) as follows:

ancestorChain(0)=*msg*,

ancestorChain(n+1)=$m_l;_p(m_1;_pk_1\cup_p,.., \cup_p m_{l-1};_pk_{l-1});_p$ ancestor(ancestorChain(n),n)

satisfying the equations E :

m_1.content=pay and k1.content=invest,…,m_{l-1}.content=pay and k_{l-1}.content=invest and m_l.content=invest and m_l.time < m_1.time < k_1.time and ,…, m_l.time < m_{l-1}.time > k_{l-1}.time and m_l.reciever=I.

Define Earliest(msg) as lfp(f,*msg*,E).

As a special case, I show how to compute the ancestor chain of the pattern II in Table 5.5 using StreamSQL below in Figure 5.8.

.	**ClimbRecruitPath** **Description**: Given the promoter traces back the MEI records and finds the path and the distance to/from Orchestrator using *pattern II*. **Input:** Promoter $P, MEI tuples **Output:** Ancestor Chain of Promoters as RecruitPathOut
1	**CREATE INPUT STREAM** MEI ($MEI schema);
2	**CREATE OUTPUT STREAM** RecruitPathOut($MEI schema , newPromoter string);
3	**CREATE STREAM** LocalStream ;
4	**DECLARE** pointerPromoter string **DEFAULT**$P**UPDATE FROM** (**SELECT** newPromoter **AS**
5	pointerPromoter **FROM** RecruitPathOut);
6	**SELECT** * **FROM** MEI
7	**WHERE** msg=="invest" **AND** receiver=="0" **AND** sender==pointerPromoter
8	**INTO** LocalStream;
9	**SELECT** time, sender, receiver, msg, content, GetXPATHValue(content,"../promoter/") **AS** newPromoter
10	**FROM** LocalStream
11	**INTO** RecruitPathOut;

Figure 5.8. Computing the Orchestrator

ClimbRecruitPath is a trace-back query. Given a recruiter, it traces back recruiters upwards to find the orchestrator. Therefore, it can derive the start time of the scheme and traverses every record in descending order of timestamps looking for the sender of invest

messages. In order to do so, my query declares a dynamic variable (see **DECLARE**) called **pointerPromoter** in line 4 and the suspected promoter is passed as a default value (see **DEFAULT**) as the orchestrator. Each time the output emits a hop meeting the criteria in line 7, the promoter value is picked out from the content of the invest message by XPATH function (**SELECT** clause in line 9). It is then written to the output stream (line 2), and assigned to the dynamic variable **pointerPromoter** in line 5. The newly assigned value is used as the predicate in line 7 for locating the next message if it matches the sender value.

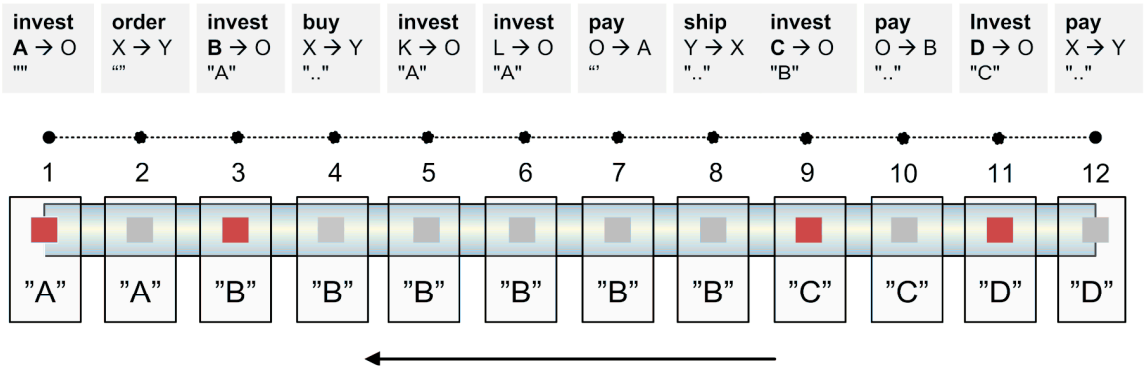


Figure 5.9. Climbing the Recruit Path

Figure 5.9 illustrates how the query traverses back through sample MEI tuples. The query emits tuples at times 11, 9, 3 and 1. Such a query reveals the distance between a specified recruiter and the orchestrator. Because long paths imply the un-sustainability of the scheme, one could get the likelihood of a recruiter becoming a victim of the scheme (that is not being able to recover the investment and the predicted profits). One other benefit of this algorithm is that it locates one of the oldest message records to start my trace-forward algorithms and to create the maximal comprehensive evidence of the scheme.

Having discovered the earliest recruits or the orchestrator of the schema, one can start tracing forward and generating the evidence. The next query compiles the evidence, using pattern II. The output is a table partially shown in Figure 5.11.

	GenerateRecruitTree **Description**: Tracing forward the MEIs, outputs an appropriate table to create tree-view of the scheme using the *pattern II*. **Input:**MEI tuples **Output:** RecruitsOut table leading to recruiter->recruitee tree structure
1	**CREATE INPUT STREAM** MEI ($MEI schema);
2	**CREATE OUTPUT STREAM** RecruitsOut ;
3	**CREATE STREAM** InvestFilterOut ;
4	**SELECT** * **FROM** MEI
5	**WHERE** msg=="invest" **AND** receiver=="O" **INTO** InvestFilterOut;
6	**SELECT** recruitee.time **AS** recruitTime,
	recruiter.sender **AS** recruiter, recruitee.sender **AS** recruitee
7	**FROM PATTERN** (InvestFilterOut **AS** recruiter **THEN** InvestFilterOut **AS** recruitee)
	WITHIN 6 (days) **ON** time
8	**WHERE** recruiter.sender== GetXPATHValue(recruitee .content,"../promoter/")
9	**INTO** RecruitsOut;

Figure 5.10. Generate Recruit Tree

The query in Figure 5.10 accepts MEI tuples. The first **SELECT** clause in line 4 is a typical filter with predicates addressing invest messages that are sent to the suspected orchestrator "O". Pattern II is defined through the query after **PATTERN** clause. The **PATTERN** clause duplicates the invest MEIs so that it can apply the appropriate template (see **THEN** clause) and predicates (see **WHERE** clause in line 8) between messages. The query uses a window of size 6, limiting the query in finding the patterns only within the specified period. Small size windows may lead the query to miss more correlations than do big size windowed queries. Figure 5.11 shows how the query progresses and when it emits its findings.

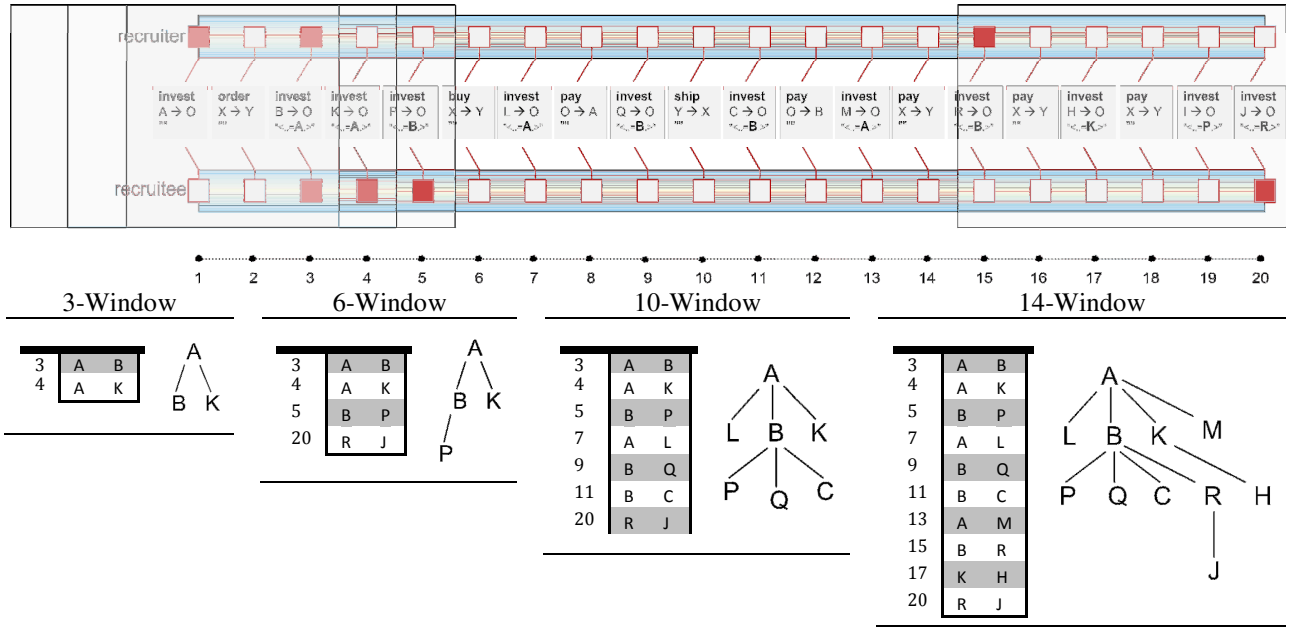


Figure 5.11. Generating Recruit Tree

Figure 5.11 illustrates how the query advances through the records. At time 3, the query emits the first recruit; and, at time 4, the second. When the query processing reaches time 5, another recruit is discovered. By time 20, a window of size 6 does not find 6 consecutive messages that create a specified pattern. Tables in Figure 5.11 show how the window size impacts the outputs of the query. Right next to the tables, corresponding trees reveal the promoting architecture of the scheme at various levels.

5.9. Damage Estimation

Finally, I show how to estimate the damage caused by such schemes and the profits made by the promoters. Based on the findings above, collecting the messages exchanged

between endpoints can be used to measure the damage. **RenderWinLossTable** query in Figure 5.12 creates and renders a **WinLossTable** that contains promoter web services found by previous query and their earnings and losses through the scheme run.

The query in Figure 5.12 accepts MEIs in ascending order of time and promoter web services (see line 2) found by the **GenerateRecruitTree**. After creating the table in line 4, it loads the table with the incoming promoter web service list (see **INSERT** in line 5). Because I assume that there are homogenous invest and pay choreographies as described earlier—not an unrealistic assumption compared to real scenarios, one can access amount values from deliver and acknowledgment messages respectively using invID and payID values in first invest and pay messages of both choreography instances (remember Figure 3). This is done by using the patterns in lines 13 and 17 that locate correct amounts for each invest and pay choreography instances. Collecting invest and pay amounts would lead us to computing the amount of money flows between web services involved in the scheme, thus revealing who has what role through the incident, such as victim, promoter, or orchestrator. Notice the **UPDATE** clauses in lines 19 and 20, previously collected amounts of gain and paid are set correctly in the table when they meet the criteria in **WHERE** clause. That is, each correspondent promoter web service is updated in gain and paid fields when the query encounters invest and pay message related to them.

```
RenderWinLossTable
Description: Tracing forward the MEIs, renders an appropriate table showing how web
services took part in the scheme.
Input:MEI tuples, promoters from GenerateRecruitTree query as RecruitTreeResultsIn
1   CREATE INPUT STREAM MEI ($MEI schema);
2   CREATE INPUT STREAM RecruitTreeResultsIn (
      ID int, promoter string);
4   CREATE MEMORY TABLE WinLossTable (
      ID int, promoter string, paid double, gain double
    )   PRIMARY KEY(ID) USING btree;
5   INSERT INTO WinLossTable (ID, promoter, paid, gain)
      SELECT ID,promoter,0.0 AS paid ,0.0 AS gain
      FROM RecruitTreeResultsIn;
6   CREATE STREAM InvestDeliverOut ;
7   CREATE STREAM PayAckOut ;
8   SELECT * FROM MEI
9       WHERE (msg=="invest" AND receiver=="O") OR (msg=="deliver" AND sender=="O")
        INTO InvestDeliverOut
10      WHERE (msg=="pay" AND sender=="O") OR (msg=="acknowledge" AND receiver=="O")
        INTO PayAckOut;
11  CREATE STREAM InvestOut ;
12  SELECT
        double(GetXPATHValue(deliver.content,"./amount/"))AS amount,
        invest.sender AS sender,
        invest.receiver AS receiver
13      FROM PATTERN (InvestDeliverOut AS invest THEN InvestDeliverOut AS deliver)
        WITHIN 3 ON time
14      WHERE invest.msg=="invest" AND deliver.msg=="deliver" AND
         GetXPATHValue(invest.content,"../invID/") ==  GetXPATHValue(deliver.content,"../invID/")
        INTO InvestOut;
15  CREATE STREAM PayOut ;
16  SELECT
        double(GetXPATHValue(ack.content,"./amount/")) AS amount,
        pay.sender AS sender,
        pay.receiver AS receiver
17      FROM PATTERN (PayAckOut AS pay THEN PayAckOut AS ack)
        WITHIN 3 ON time
18      WHERE pay.msg=="pay" AND ack.msg=="acknowledgment" AND
        GetXPATHValue(pay.content,"./payID/")== GetXPATHValue(ack.content,"./payID/")
        INTO PayOut;
19  UPDATE WinLossTable USING InvestOut AS i
        SET paid = paid + i.amount
        WHERE promoter == i.sender;
20  UPDATE WinLossTable USING PayOut AS p
        SET gain = gain + i.amount
        WHERE promoter == p.receiver;
```

Figure 5.12. Rendering a Damage Table for Recruit Tree

5.10. Related Work

Luckham [61] proposes Rapide, an event pattern language that defines complex patterns, that has been implemented in some service oriented architectures. To my knowledge, none of them provide non-repudiable messages. Luckham [61] also provides rules to specify business collaborations compliant with the ISO 15022 standard. Although complex event processing (CEP) is a wide application area, most of the efforts do not derive global behavior from external observations.

Widder et.al. [62] propose a new approach based on discriminant analysis of events, grouping them if they represent an unknown pattern. They envision using their method in recognizing new patterns of credit card use and fraud related to them. Their approach, however, strongly depended on having complete knowledge of events to accurately derive behavior. Thus, maliciously created events may raise difficulties in to-be-developed detection algorithms. They implement the experiment environment based on a CEP [61] engine.

Semantic correlation of message exchanges allows recreating the exact instances of the choreographies. DePauw et al. [63] present a heuristic algorithm to find the correlations between messages. They employed a set of refinement efforts on tables to achieve the correlations. Barros et al. [57] list other possible opportunities for correlation as described earlier. However, they offer no algorithm to employ them.

5.11. Conclusions

I have precisely specified choreographies that could be used to detect Ponzi and other illegal schemas occurring among web services. I have shown how to specify these choreographies using StreamSQL, a language and a run-time that can process queries over streams of data. Although my choreographies only specify some Ponzi schemas, the method holds promise in specifying and detecting other illegal business schemes [64] that can be mined from repositories of financial transactions. The next chapter addresses extending my method in developing an online warning system that detects business schemas that appear legal from a microscopic view, but are macroscopically illegal.

CHAPTER 6

ONLINE DETECTION AND ALERT MODEL AGAINST MISUSES OVER WEB SERVICES

6.1. Introduction

Financial institutions and their business partners are moving to service oriented architecture; and semantic web services are building much more promise such as dynamic brokerage over investment firms or the stock market. The previous chapter introduced queries that generate evidence of web services behavior such as legal choreographies or misusing choreographies in the case of *Ponzi/Pyramidal schemes*. Those pattern queries generate evidence out of messages stored at repositories. Especially, illegal business schemes may keep running and dispersing over new web services as time passes. Rather than post mortem or late detection there is need to have an online detection and alert mechanism for immediate responses, such as informing potential victim services regarding the spreading of illegal business activity.

In addition to business misuses, mentioned above, there are exploits at service level as well. Mal-actors in those cases abuse the inter-dependency spanning over the web services employed by static choreography models. Dataflow attack, for example, is a

special type of service level exploits that is most difficult to detect since the malicious code leaks into services among regular data. Chapter 3 described a XSS (cross site scripting) attack scenario in detail; web services, would definitely consider online detection, prevention, or alert mechanism on demand against those. Instantiation flooding [56] is another service misuse of denial of service (DoS) type on web service compositions. Briefly, the attacker repeatedly invokes the receiver operation of the process at the target web service. The target engine tries to instantiate every request, thus reaching DoS. The web services that are confident on maximum throughput values would consider a detection and prevention model against such kind of misuses.

Through the chapter, I give brief explanation of my framework which I enhance for it can help online detection of web choreography misuses in section 6.2. Section 6.3 describes my online detection model and sample queries than can detect different types of misuses in real time. Section 6.4 describes an alert model that can warn potential intended endpoints, such as potential dependent web services. Section 6.5 describes the online detection architecture that is designed for the EGF framework. Section 6.6 discusses related work; and Section 6.7 concludes the chapter.

6.2. The EGF in Online Mode

In order to facilitate and base evidence generation on a reliable infrastructure that can convince the services who wants accountability on their transactions and fast detection

when misused, I proposed designing an *Evidence Generation Framework (EGF)* that preserve appropriate evidence to recreate the composed web service invocations independently of the partners of the transaction in Chapter 4. While evidence derivation and comprehensive evidence generation is done by storing/retrieving evidences in cryptographically secure repositories in Chapter 5, I here propose upgrading those layers to function in online mode by caching evidence streams and querying from those. To do so, bottom layer passes evidence indexes into upper layers at service invocation time, thus feeding these services with live evidences for mining complex ones out of them. Having employed cache based live queries onto those layers I propose a twofold response model; alert and prevent as illustrated in Figure 6.1.

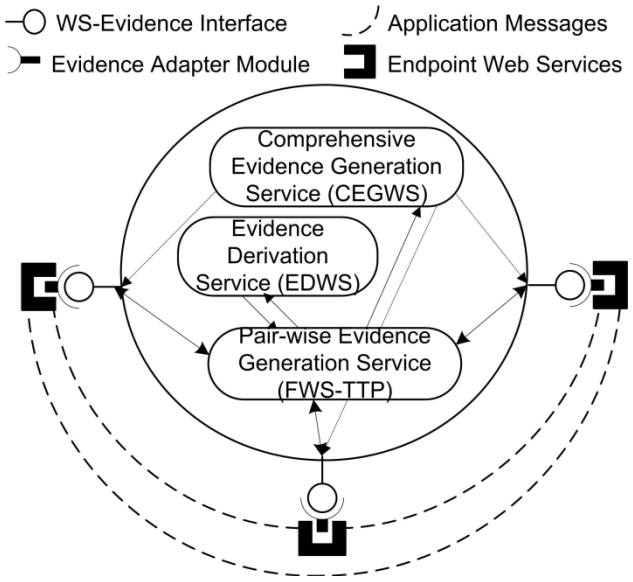


Figure 6.1. The EGF in Online Mode

6.2.1. Enhanced Pair-wise Evidence Generation

The EGF provides online detection by means of two services: CEGWS (Comprehensive Evidence Generation Web Service) and EDWS (Evidence Derivation Web Service) as illustrated in Figure 6.1. The former generates evidence against global and complex misuses and the latter generates evidence against service misuses.

As I explained pair-wise evidence generation process in detail through previous chapters, WS-Evidence messages flow between endpoint web services (notice double sided arrows in Figure 6.1) through FWS-TTPs in a specific message structure as below:

<#session| #message |#signature$_{\text{Sender-K}}$ (#session|”4”|# env)>

In order to enrich the EGF with online capability, I propose enhancing the pair-wise evidence generation process at TTPs. Below, the pseudo BPEL diagram shows a typical delivery process connecting to online services at service invocation time, thus leading to online detection, prevention, or alert mechanisms on demand.

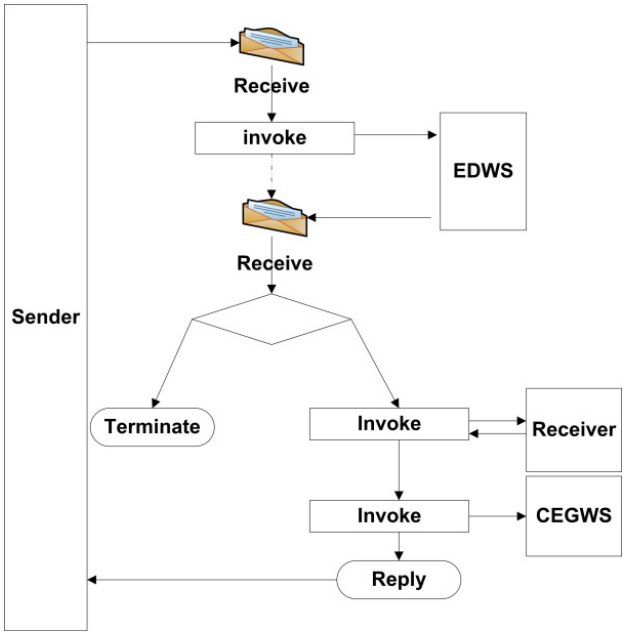


Figure 6.2. Deliver Process for Online EGF

The pseudo process above shows sender-receiver interactions and live detection invocations rather than detailed activities through the process. The deliver process at TTPs extracts each message received (notice the first *Receive* in Figure 6.2) in MEI format defined in previous chapter where *sender* and *receiver* fields extracted from #session—*msg* and *content* fields from the #message parts of a WS-Evidence application message (e.g. OneWay in Figure 2); and *ID* and *time* fields are assigned by the process itself—and stores them at stations. In order to glue the process to online services for detection, prevention, and alerting I enrich the process as described above. That is, the process forwards MEIs into EDWS (notice the *Invoke* after first *Receive*) and CEGWS (notice the *Invoke* before *Reply*) thus earning the EGF framework live detection capabilities.

6.2.2. Evidence Derivation

In the second layer, endpoints can gather evidences from TTPs at any time rather than service invocation time. In order to generate evidences from TTPs for specific time intervals I rely on the evidences stored at TTPs. Evidences gathered this way can be used by a web service to exculpate from accusations. Depending upon the service level agreements, the number of evidences would increase. Chapter 4 explains samples for evidence of violations against time-out agreements and scheduled invocations between two endpoints. I here demonstrate how Evidence Derivation services (EDWS) can derive evidences regarding service misuses online, thus leading to immediate feedbacks to the bottom layer for probable prevention.

6.2.3. Comprehensive Evidence Generation

The top layer can use a rule engine or a mining system to generate global (multi-party) facts, thereby being able to reveal misuses that are not directly evident in pair-wise message records (first layer) and cannot be revealed deriving the evidences at the second layer. Through the chapter, I demonstrate how live evidences of complex scenarios can be mined from evidence of observed interactions of pair-wise communications. The framework generates alerts for potential victims, investigators, or arbiters of such global misuses so that they can take immediate actions.

6.3. Online Detection Model

Any intrusion detection approach today mentions two types of detection model; one is anomaly detection and the other is misuse based. For my work, I follow the misuse based approach.

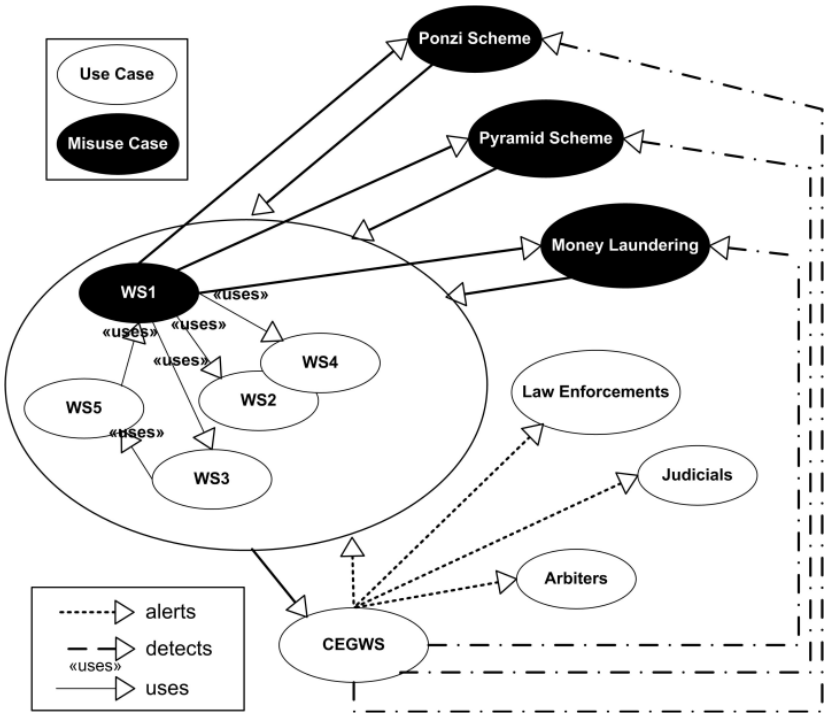


Figure 6.3. Business Misuse Case

Through the misuse based approach, I categorize the misuses over web services. Low level security implementations (e.g. WS-Security, WS-Trust) eliminate most vulnerability and prevent exploits over them such as identity impersonation, confidentiality violations. Through the layer, WS-Evidence, that I proposed earlier, I claim that more complex, misuses over web services can be detected. I classify them as

service misuses that are targeting services directly. Denial of service is the most common technique to misuse any services, thus very likely to target web services. Dataflow attack is another way of employing a misuse mostly exploiting vulnerability at endpoint web services. The most complex misuses over web services would be *business misuses* as illustrated in a misuse case diagram in Figure 6.3.

6.3.1. Online Detection of Business Misuses

Web services build choreographies and perform illegal business activities such as Ponzi schemes, pyramid schemes, or money-laundering global models. My detection model looks for specific misuse pattern featuring this activity. Rule generators of the EGF framework either heuristically figure out what the pattern exactly is or they may employ more abstract misuse patterns which more likely tends to produce false positives. I proposed a heuristic algorithm for discovering Ponzi-like misuse of choreographies in Chapter 5. However, I need more abstract patterns to detect business misuses without being dependent on any specific scenario case.

6.3.1.1. Abstracting Misuse Patterns

The query proposed in previous chapter is concrete and can be strictly applied to a specific domain of misuse emergences. For example, my heuristically discovered patterns would apply to only one specific orchestrator of the scheme. Here, I propose abstracting those concrete patterns to address more misuses that may have occurred than those based on one orchestrator. To achieve this, I use *role based* abstraction to generalize the

endpoint web services' involvement in the service choreography. I also use *type based* abstraction to generalize messages flowing between endpoints. In order to determine to what roles and types the web services and messages pertain, I propose using two methods; the first is to mine past data and learn their classes and the second is to get their roles at registration. Using these techniques I believe I can create type and role tables for messages and web services respectively. Through this chapter, I assume I already have those classification tables and PEG services having access those tables.

6.3.1.2. Mapping Messages to Types

Web services carry message names through the body parts of SOAP envelopes corresponding to the content of the messages. Different XML schemas may use different names for the same entity. For example, a pay activity may be represented by either "payInput" or "sendPay" through their SOAP envelopes. Hence, a typical type table should be able to address one-to-many activity-to-messages relations as illustrated in Table 6.1.

Table 6.1. Message Type Table (MTT)

ID	Message	Type
1	PayInput	Pay
2	SendPay	Pay
3	PayRequest	Pay
4	Investment	Invest
5	InvInput	Invest

Given the above table and a MEI record, instead of MEI.msg==”invest” or MEI.msg==”pay”, I achieve abstracting as below for invest and pay types of messages through my pattern queries:

```
MTT.message==MEI.msg AND MTT.type=="Invest"
MTT.message==MEI.msg AND MTT.type=="Pay"
```

6.3.1.3. Mapping Web Services to Roles

As a reminder of the restriction to external observations of messages, I only have web service endpoints’ identities as sender and receiver. Because I base my signature verification on PKI, that is, only the public keys registered in the system; I, however, now need role information. For example, a web service would act as an investment company accepting investing messages from investors and would act as an investor investing on other companies. A typical role table should be able to address many-to-many web services to roles relations as illustrated in Table 6.2.

Table 6.2. Web Service Role Table (WSRT)

ID	Service	Role
1	A	Investee
2	A	Investor
3	C	Investor
4	D	Bank
5	A	BankCustomer

Given the above table and a MEI record, for both receiver and sender of a message, I achieve abstracting as below for invest type of message through my pattern queries:

```
WSRT.service==MEI.receiver AND WSRT.role=="Investee"
WSRT.service==MEI.sender AND WSRT.role=="Investor"
```

```
DetectRecruits
Description: Glides over MEIs using window size 3 to detect pattern I in previous chapter
along with the predicates specified in WHERE clause.
Input:MEI tuples
Output:Ponzi-like recruit MEI pairs
1   CREATE INPUT STREAM MEI ($MEI schema);
2   CREATE INPUT STREAM MessageTypesIn (
    ID int, message string, type string);
3   CREATE INPUT STREAM WSRolesIn (
    ID int, service string, role string);
4   CREATE OUTPUT STREAM PonziDetectOut ;
5   CREATE MEMORY TABLE MessageTypesTable (
      ID int, message string,  type string)  PRIMARY KEY(ID) USING btree;
6   CREATE MEMORY TABLE WSRolesTable (
      ID int, service string,  role string)    PRIMARY KEY(ID) USING btree;
7    INSERT INTO MessageTypesTable (ID, message, type)
8      SELECT ID, message, type FROM MessageTypesIn;
9   INSERT INTO WSRolesTable (ID, service, role)
10     SELECT ID, service, role FROM WSRolesIn;
11  CREATE STREAM InvestFilterOut ;
12  CREATE STREAM PayFilterOut ;
13  SELECT * FROM MEI, MessageTypesTable AS MTT,
      WSRolesTable AS WSRTInvestee, WSRolesTable AS WSRTInvestor
14    WHERE MTT.message==MEI.msg AND MTT.type=="invest" AND
        WSRTInvestee.service == MEI.receiver AND
        WSRTInvestee.role==”Investee”
15    INTO InvestFilterOut
16    WHERE MTT.message==MEI.msg AND MTT.type=="pay" AND
        WSRTInvestor.service == MEI.receiver AND
        WSRTInvestor.role==”Investor”
17     INTO PayFilterOut;
16  SELECT "Ponzi-like recruit" AS detected, invest.time AS investTime,
      pay.time AS payTime, pay.receiver AS recruiter,
      invest.sender AS recruitee
17    FROM PATTERN (InvestFilterOut AS invest THEN
        PayFilterOut AS pay)
18    WITHIN 3 (days)  ON time
19    WHERE invest.receiver==pay.sender AND
        regexmatch(".*"+pay.receiver +".*", invest.content)
20    INTO PonziDetectOut;
```

Figure 6.4. Live Detection of Ponzi-like Recruits

6.3.1.4. Using More Abstract Content Linkage

Previous chapter used exact XPaths of linkage parameters through the message. Here I generalize this as well. Therefore, for my abstract detection queries I propose regular

expression matching through the entire content of messages. Below is an example of looking for a promoter reference through an invest message without binding it to a specific path of a specific schema.

```
regexmatch(".*"+payMEI.receiver +".*", investMEI.content)
```

Hereafter I show how to mine business misuse instances of given patterns over live MEIs of all observed web transactions using streamSQL [59] and a StreamBase platform [60] that I explained in previous chapter.

The query in Figure 6.4 is in streamSQL and can detect an abstract misuse pattern, say, of what happens in real time. **DetectRecruits** accepts live MEI records sorted in ascending order of timestamp by a sort operator. It also accepts message type table in line 2 and web service role table in line 3 from the local source and loads them into memory tables in lines 7-10. In order to successfully process the abstracted pattern query, it filters the records into two: invest and pay employing the predicates defined in lines 14 and 16. Notice the **WHERE** clauses in these predicates employ abstractions by looking up type and role tables as described earlier. Having invest and pay streams separate, the query, now, employs the appropriate template (see **THEN** phrase) in line 17. That is, invest messages are expected before pay messages. Predicates defined in line 19 say that the receiver of the invest message should be equal to the sender of the following pay message and the promoter value in the content of the invest message should be the receiver of the

following pay message. The window size is arbitrarily set to 3 in line 18. The **SELECT** part gathers the required information about the detected pattern and emits the result to the **PonziDetectOut** table. Figure 6.5 shows how two Ponzi-like recruits are detected using the window of size 3 from a collection of 9 MEIs. Figure 6.5 shows how messages in a MEI table come into the query and are processed as two separate streams shown as pipes, where transparent rectangles represent two different snapshots of the query window; one that arrived at time 5 and the other that arrived at time 8.

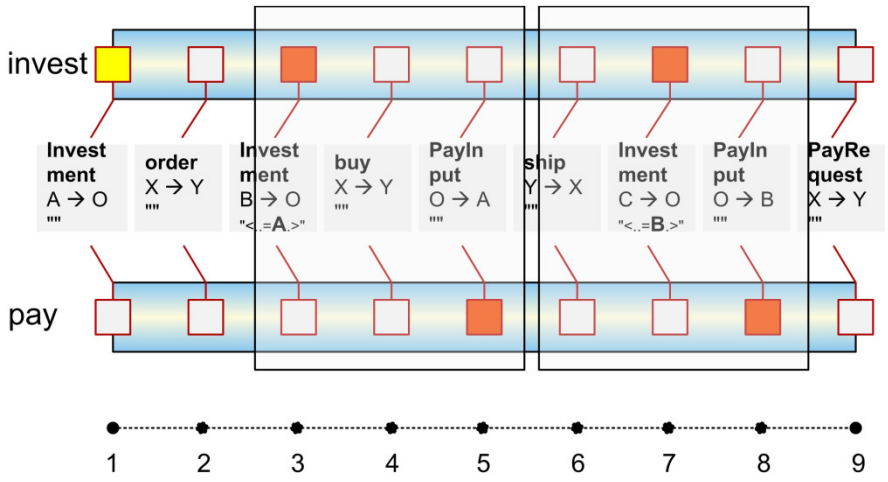


Figure 6.5. Detected Ponzi-like Patterns

Detecting few Ponzi-like recruits may give little confidence in declaring that a Ponzi scheme was detected. In order to increase the confidence one can define some minimum support value as threshold thus alerting only when it meets. The **AlertRecruits** query presented in previous chapter (see Figure 5.7) can be appended to above query as it is to achieve this.

6.3.2. Online Detection of Service Misuses

Unlike business misuses, service misuses do not need a global perspective to indicate illegal intent. Even only one malicious message can launch an attack or exploit some vulnerability at a target web service. Or a specifically designed set of messages can employ exploits at a target web service. For both cases below I briefly describe *malicious content* and *instantiation of flooding* techniques and propose detection queries for them.

6.3.2.1. Malicious Content

Here the signature of the misuse would be malicious content carried inside the messages. A typical example would be XSS attack described in Chapter 3. Detecting such attacks needs less complicated queries that require scanning the content and a well built library of malicious scripts. Assuming already having such a library as illustrated in Table 6.3, the query in Figure 6.6 detects messages that contain those signatures of malicious scripts. Those script signatures might be derived some prevention cheat sheets such as OWASP's (Open Web Application Security Project) [65].

Table 6.3. Signature Table (ST)

ID	Misuse	Signature
1	XSS	<script>
2	BufferOverFlow	/sh
3	BufferOverFlow	/bash

```
   DetectMaliciousContent
   Description: Checks every message content if there is a malicious content.
   Input: MEI tuples
   Output: Matched attacks
1  CREATE INPUT STREAM MEI ($MEI schema);
2  CREATE INPUT STREAM AttackSignaturesIn (
     ID int, name string, signature string);
3  CREATE OUTPUT STREAM AttacksOut;
4  CREATE MEMORY TABLE SignatureTable (
       ID int, name string,  signature string
     )   PRIMARY KEY(ID) USING btree;
5  INSERT INTO SignatureTable (ID, name, signature)
6          SELECT ID, name, signature FROM AttackSignaturesIn;
7  SELECT SIG.name AS misuse FROM SignatureTable AS SIG, MEI
8    WHERE regexmatch(".*"+SIG.signature+".*", MEI.content)
9    INTO AttacksOut;
```

Figure 6.6. Detecting Malicious Content

DetectMaliciousContent accepts live messages in MEI tuples and loads attack signatures prior to their run in lines 5-6. Attack signatures may reside in a database table or a regular expression file. In either case, there are readers and database clients to pass tuples into the input adapter, thus allowing them to be used in expression matches at line 8.

6.3.2.2. Instantiation of Flooding

Another service misuse type would be denial of service that is very common for every type of service application. Below the SOAP layer the problem is the same with typical DoS over HTTP services; however, in the web services case, the transport layer may vary, thus a SOAP layer solution would helpful. Assuming that TTP processes run over hardware with high computation power, I address instantiation floods targeting receiver services. Each receiver web service may declare different thresholds for its processes

depending upon their business logic or memory usage. Therefore, as shown in Table 6.4, there is need to have a table at TTP stations storing web services, and relevant threshold values of maximum throughput, say, per second, against probable instantiation flooding attacks.

Table 6.4. Web Service Threshold Table (WSTT)

ID	Service	Threshold
1	A	4
2	B	19
3	C	99

```
DetectInstantiationFlooding
Description: Using time based window checks if there is a set of messages targeting at same
receiver exceeding its threshold in number.
Input:MEI tuples
Output:"Instantiation Flooding" alerts containing attacker, victim, count and time interval
information
1   CREATE INPUT STREAM MEI ($MEI schema);
2   CREATE INPUT STREAM ThresholdsIn (
      ID int, service string, threshold int);
3   CREATE OUTPUT STREAM DoSsOut ;
4   CREATE MEMORY TABLE ThresholdTable (
      ID int, receiver string, threshold int
    )   PRIMARY KEY(ID) USING btree;
5   INSERT INTO ThresholdTable (ID, receiver, threshold)
6     SELECT ID, service, threshold FROM ThresholdsIn;
7   CREATE STREAM AggregateByTimeOut ;
8   SELECT sender, receiver, count() AS count,
      firstval(time) AS startTime, lastval(time) AS endTime
9     FROM MEI [SIZE 8 ADVANCE 1 TIME OFFSET 0]
10    GROUP BY receiver, sender INTO AggregateByTimeOut;
11  SELECT "Instantiation Flooding" AS misuse, a.sender AS attacker,
      a.receiver AS victim, count,  a.startTime, a.endTime
12    FROM AggregateByTimeOut a, ThresholdTable t
13    WHERE count>t.threshold AND a.receiver==t.receiver
14    INTO DoSsOut;
```

Figure 6.7. Detecting Instantiation Flooding

The query in Figure 6.7 accepts live messages in MEI tuples and loads service thresholds prior to its run in lines 5-6. Service thresholds may reside in a database table or a regular expression file. In either case there exist readers and database clients to pass tuples into the input adapter, thus allowing them to be used in threshold matches. Since the actual frequency of the messages determines whether a set of messages is malicious or not, the above query uses the time window rather than using timestamp values inside records. That is, messages incoming every 8 seconds are processed by the query. And the window shifts every second as coded in line 9. For every 8-second sets of messages the query groups the messages in sender and receiver fields in line 10 and selects the count values for each group in line 8. The **WHERE** clause in line 13 detects if there is an "Instantiation Flooding" attempt from a certain "sender" (called attacker) based on "receiver" (called victim) services' threshold criteria in a 8-second window of live MEI records. Finally the matching result is emitted in line 14.

6.4. Alert Model

Having generated enough evidences regarding a misuse, my framework sends targeted alerts. As described above I categorize misuses and define their types. In order to introduce a stable and robust alerting framework I have to address two issues: First, I need to define an effective domain of web services to alert. Based on types of alerts the scope of the web service network to be alerted is determined. Second, I need to tune the queries to produce unique alerts per misuse detection, thus denying false positives. Here I describe how to scale down the alert domain.

6.4.1. Scaling Down the Alert Domain

The alert model described above has to scope the alerting domain. One important reason for that is to deny the framework producing so many unnecessary alert messages, thus causing network congestions, waste of computation at endpoints, and triggering endpoints to take actions mistakenly. I describe three ways to narrow the alert recipient domain. First, depending upon misuses I consider alerting only dependent web services given a message, thus creating a *dependency tree* starting from the message as root. Second, for some misuses one may need to know the *types of web services* that have the potential to take part in the misuse as victim. As described earlier, Ponzi schemes target web services that are the type of investor, that is, web services that are likely to make investments through web services architectures. As third, not only for predefined misuses but also for use cases there is need to warn *potential members*. Mostly for anomaly based scenarios the framework alerts the potential domain of a choreography model regarding a suspicious unknown misuse of the model. Hereafter, I describe how I can draw the intended domain regarding those methods.

6.4.1.1. Dependency Tree Generation

When a suspected activity occurs and is detected relating to a message then there is need to alert other services upon this activity. One of the essential issues in an alert model is to scale down the alert domain for fast access to intended endpoints and not spending time trying to access/disturb irrelevant services. Learning dependencies over past message

interactions becomes vital at this point and is twofold: First, one can trace back to learn possible web service invocations causing the root activity; and second, tracing forward could allow one to learn downstream invocations possibly caused by the root. However, the observed two messages may not necessarily be linked to each other. For example, when A sends a message to B and B sends a message to C, the first message may not have to be the cause of the second. Authors [57, 58, 63] work mostly on exact correlation of messages. Basu et. al. [66] proposes a probabilistic work that is trying to understand the message correlations with various probabilities. Here I, however, do not need exact causal correlations to obtain the downstream dependents of a message. The query below learns downstream dependencies for a web service as root. The query expects to start from a record in the past and traverses the records forward generating a dependents table.

```
GenerateDependencyTree
Description: Tracing forward the MEIs, outputs a table to create tree-view of dependents.
Input: MEI tuples
Output: DependentsOut table leading to invoker->dependent tree structure
1  CREATE INPUT STREAM MEI ($MEI schema);
2  CREATE INPUT STREAM DependentsIn ( service string);
3  CREATE OUTPUT STREAM DependentsOut (
     time timestamp, invoker string, dependent string );
4  CREATE MEMORY TABLE DependencyTable (service string)
     PRIMARY KEY(service) USING btree;
5  INSERT INTO DependencyTable (service)
6    SELECT service FROM DependentsIn;
7  INSERT INTO DependencyTable (service)
8    SELECT dependent FROM DependentsOut;
9  CREATE STREAM NotInTreeOut;
10 SELECT MEI.*, dt_out.service AS inTree
11   FROM MEI OUTER JOIN DependencyTable AS dt_out
12   WHERE MEI.receiver == dt_out.service INTO NotInTreeOut;
13 SELECT NITO.time AS time, NITO.sender AS invoker,
     NITO.receiver AS     dependent
14   FROM NotInTreeOut AS NITO, DependencyTable AS dt
15   WHERE NITO.sender==dt.service AND isnull(NITO.inTree)
16   INTO DependentsOut;
```

Figure 6.8. Generating Dependency Tree (Forward)

The query in Figure 6.8 accepts MEI records and a root service as shown in line 2. The query loads (lines 5-6) the root service into the **DependencyTable** which is created on memory and is appended each time the new dependent services are found. The **SELECT** in lines 10-12 retrieves the MEI even if the receiver of the MEI is not in the table and adds a new field, **inTree** , as null. The next **SELECT** checks if the **inTree** value is null and the sender is in the **DependencyTable**. If the criteria meet in line 15 the output stream (notice "**FROM DependentsOut**") inserts a new dependent service into the table in lines 7-8, thus building a downstream dependency table.

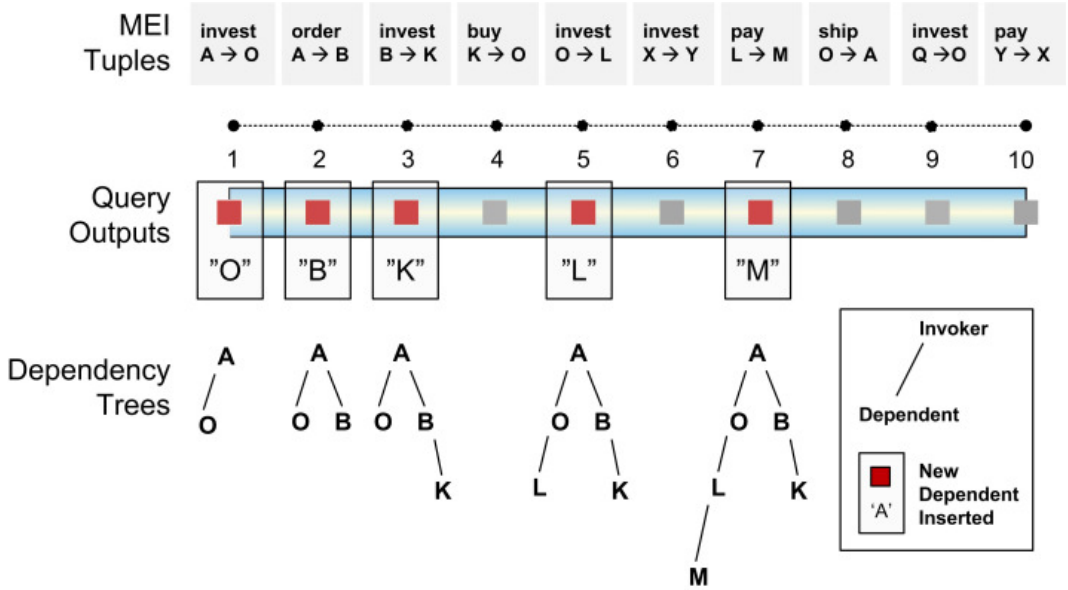


Figure 6.9. Generated Dependency Trees

Figure 6.9 illustrates the actions that **GenerateDependencyTree** takes traversing over a set of MEI tuples listed at the top in ascending order of time. Tuples at times 1,2,3,5, and 7 meet the criteria that the query looks for. It emits outputs including invoker and dependent fields. These two fields lead to building dependencies in tree form.

6.4.1.2. Web Service Types

The EGF requires web services to register under specific role types, such as “investor”, “investee” or “bank”. This could be associated with related misuse types. That is, an investor web service might be a victim, promoter, or a prospective victim for a Ponzi-like misuse as defined earlier. This could help the EGF to alert the exact web services when a specific misuse is detected by looking up the potential web services of the associated type. For example, only investor web services would be alerted for a detected Ponzi-like misuse.

6.4.1.3. Potential Members

When a suspicious activity is observed regarding a global model all possible web services that can get involved in the model are to be alerted. This might be achieved in two ways: First, early association with global models (use patterns) and second extracting the global models along with all possible branches that it may climb up at any probability. The first can be performed during the first registration of use pattern. The latter requires that global patterns be examined in detail along with all logical splits, choices, loops, etc.

6.5. Online EGF Architecture

The EGF introduces two different online architectures for detection, prevention, and alerting. The first is for business misuses, thereby employed remotely on a central system that gathers all messages. The second is for service misuses, thus employed at TTP

stations mostly for prevention purposes or marking the malicious activity for detection purposes.

6.5.1. Business Level Design

A central online web service called CEGWS generates business level comprehensive evidences. As shown in Figure 6.10, collecting the Message Evidence Index records from TTPs at their service invocation times when it produces and stores alerts. CEG Alert clients send alerts to relevant web services that might be threatened by the misuse.

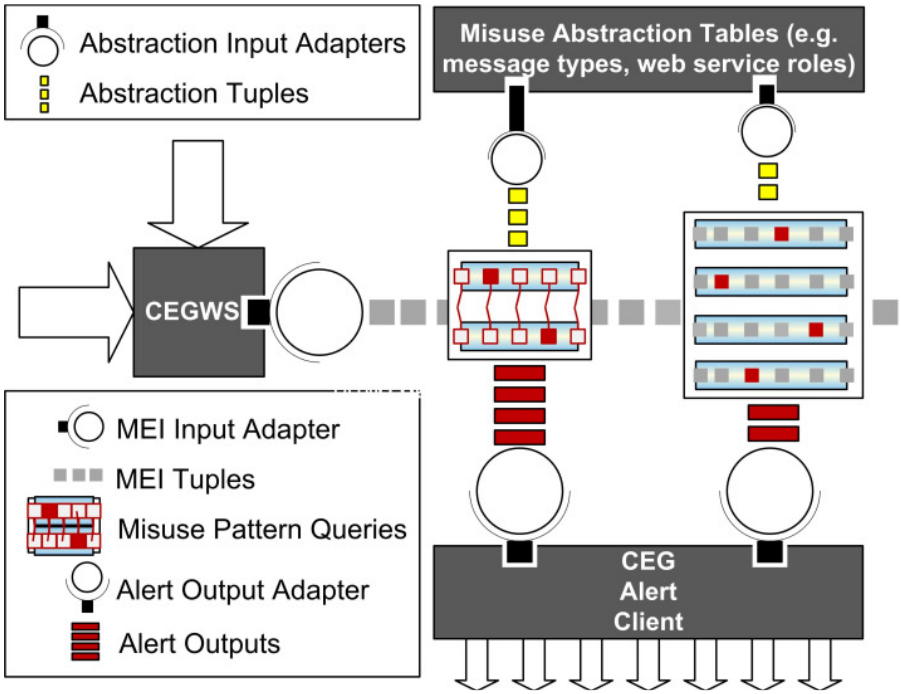


Figure 6.10. Online CEGWS Architecture

The **CEG (Comprehensive Evidence Generation) Web Service** collects MEI records from other FWS-TTP stations. The records are stored on one hand and directed to the

StreamBase MEI input adapters to be processed for evidence of business misuses or other comprehensive evidences. The web service accepts one way WS-Evidence messages including MEIs from TTP stations and lower level security mechanisms are performed for confidentiality and authentication purposes.

The **MEI Input Adapter** is a live input adapter accepting SOAP messages and outputting MEI tuples in real time into online detection event applications. As the StreamBase input adapter API empowers such live adapters (e.g. JMS, IBM MQ) I propose designing a real time adapter which is extracting index records from WS-Evidence SOAP messages and passing MEI records into StreamSQL process engines for query runs.

MEI Tuples are produced by MEI input adapters. They all enter StreamBase processors and related StreamBase event application. They, at this early phase, are inevitably unsorted on the *time* field.

The **Sort Operator,** using an on-demand window size, sorts incoming records because MEI records are assumed unordered when they first arrived from several TTP stations. Therefore, patterns that need time based dependencies among the messages coming from a variety of stations could even be detected.

Misuse Pattern Queries are based on a particular pattern query language, streamSQL, and as shown earlier, discovered misuse patterns are queried using those streamSQLs. StreamBase's modular event application structure would introduce the capability of mining many patterns consecutively. In accordance with the pattern streaming MEIs might be ordered in ascending or descending timestamps. Each query is supposed to emit different outputs as alerts, thus requiring calling output adapter simultaneously for several times.

Alert Outputs are mostly designed in SELECT parts of streamSQLs of misuse patterns. The type of detected misuse is the essential part for any alert output so that the alert output adapter can take appropriate action and the Alert Client can invoke web services that might get affected. The schema below shows the essential fields for a typical alert output.

The **Alert Output Adapter** processes over the alert outputs and based on the type of misuse it calls Alert WS-Client to send alert messages.

The **Alert Client** is a typical WS client called by the Alert Output Adapter when misuse is detected. It creates SOAP communications with endpoint web services through WS-Evidence alert message specification. It fires one way alert messages as including the details described below.

Alert Messages contain information related to a business misuse incident. Business misuses may be in several types (#misuse_type) and each types of misuse may feature various schemes (#misuse_code). I, therefore, propose coding each scheme distinctly. Each misuse scheme consists of one or more malicious actors (#mal_actors{actor1, actor2…actorN}), thereby alert messages having a series of mal-actors.

{#misuse_type, #misuse_code, #mal_actors{actor1, actor2…actorN}}

Evidence modules at endpoints or specifically designed XML firewalls can absorb the alert messages in above format. They can take immediate actions such as creating a rule ignoring messages coming from suspected malicious actors or creating a black list for not being involved in any activity with them.

6.5.2. Service Level Design

A local online evidence derivation web service called EDWS generates service level evidences. As shown in Figure 6.11, it receives the envelopes from the deliver process at TTPs prior to their service invocations it produces, and prevents or marks relevant messages. EDWS Alert clients send alerts to CEG web services so that they might help through some further investigations. EDWS Message clients invoke the deliver process back to continue their actual invocations.

I propose an application level prevention model, that is, the Deliver process conducts early detection by invoking the EDWS. The process pushes the deliver messages into locally implemented EDWS that runs queries defined in streamSQL earlier. First, the receive activity initiates instances for each request and pushes the messages into local detection service that checks the messages using reasonable window sizes to look for misuse patterns. The detected messages are marked and all messages are sent back to the process.

In accordance with the policies employed by the service logical operator in Figure 6.2, one option is that a termination is performed over the instance or branches other way and continues as the rest of the process instructs. However, in any case, the alert message is produced and sent to CEGWS for further investigation. This is done by the alert client as illustrated below.

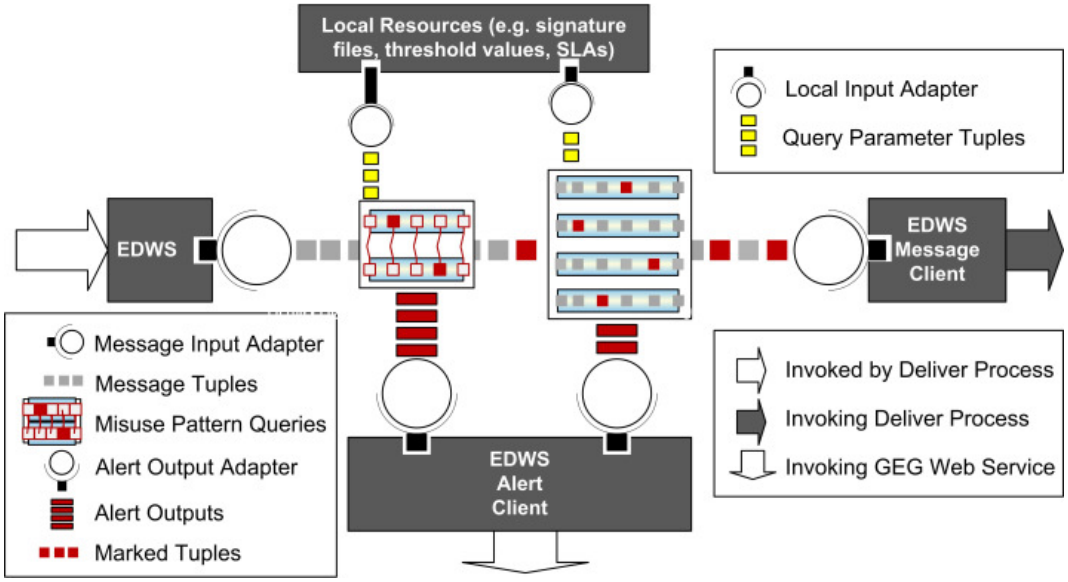


Figure 6.11. Online EDWS Architecture

EDWS is invoked by the Deliver processes at TTPs at service invocation times. The records are processed by the StreamSQL pattern queries for service misuses such as instantiation flooding or malicious content. The tuples are marked if they feature the misuse pattern and in accordance with the policies, the deliver process either terminates those sessions or lets them run.

The architecture employs Regex readers to read malicious signatures from a signature file and the related query loads them into a memory table for further lookups through the detection process.

Other types of readers or local input adapters parse other service level agreement (SLA) files in order to load significant threshold values into lookup tables queried during detection.

The **EDWS Message Client** marks the messages involved in any misuse. As a typical web service client invokes the deliver process back addressing the second receive activity (remember Figure 6.2) in the process.

The **EDWS Alert Client** is different from the CEG alert client because it connects to only CEG web services rather than endpoint web services. It creates store messages in MEI format with a specific *msg* value "alert" and *content* value as defined in alert

message definition above. On the contrary of TTP stations, EDWS clients send alert messages in MEI format inside store messages.

6.6. Related Work

As mentioned in previous chapter, Widder et.al. [14] propose a new approach based on the discriminant analysis of events grouping them if they represent an unknown pattern. They envision their approach would help recognizing new patterns of credit card transaction use case scenarios and the fraud activities related to them. Their approach, however, does not promise any live detection and alert mechanism.

Sense & response service architecture (SARESA) of [66] provides real time business intelligence (BI) unlike traditional BIs. SARESA introduces a complete process of detecting, interpreting, automation, and response to business partners. Rather than my live external observations of communications it proposes an event-driven architecture that is collecting the events from members. Therefore, it is incapable of prevention. On the other hand, leaving the partners free to send events to the system would deny it being more sensitive in detection which is not the case through my model. However, SARESA has the advantage of serving diverse IT architectures and is not bound to only web services.

Ari [67] describes a data mining model management system that addresses model outdates, scalable management, semantic differences between models, and business process integration for real time BI over SOA. Although the work in [67] does not propose or address specific real time architecture it helps those systems to be multi dimensional in time, syntax, or semantics regarding the models they can use. Such technique is similar to my abstraction method only in one dimension, which is, bridging the syntax gap.

6.7. Conclusions

To abstract detection queries I generalized them based on web service roles and message types. Using such queries I introduced a more promising online detection model for web services. Proposing a dependency tree generation query I narrowed the domain of potential web services to be effected and should be alerted based on detection query outputs. Categorizing misuses at the service and business level I briefly designed a prevention model for service misuses and an alert model for the business level.

CHAPTER 7

EVALUATION OF EXPERIMENT RESULTS AND VALIDATION STATEMENT

7.1. Introduction

Through my dissertation, I introduced a three layered platform to generate evidences of web services misuses. The services at the bottom and middle layers have been studied by others with a narrower scope [9, 41, 69, and 70]. Therefore, I tended to validate the third layer of my evidence generation framework in which I introduced novel queries in Chapter 5. I summarized those queries and the patterns they follow along with the names I used through the chapter in Table 7.1 below.

Table 7.1. Query-Pattern Mapping

Query Name	Pattern	Pattern Name
GenereateCHOR-Investing	$(\text{invest} \otimes_p \text{authorize});_p \text{confirm};_p \text{deliver}$	**invest-chor**
DetectRecruits	$\text{invest};_p \text{pay}$	**invest-pay**
ClimbRecruitPath	$\text{invest1};_p \text{invest2}$	**invest-invest**
GenerateRecruitTree	$\text{invest1};_p \text{invest2}$	**invest-invest**

In order to test their accuracy and performance rates. I generated synthetic data and used a special simulation platform as my test environment.

7.2. Data Characteristics

I introduced an illegal business scheme, which is novel to the best of my knowledge. Therefore, I was unable to access real data featuring such scheme. However, I generated special synthetic data in MEI format called MEI-I that reflect Ponzi/Pyramid Schemes, which I introduced in Chapter 5. The MEI-I, having 1 record per second, contains 193827 records in total spreading over three days (January 19, 2006 – January 21, 2006). The MEI-I is a file in CSV (Comma Separated Values) format. I focused on four aspects which one can observe through real life records of a potential Ponzi scheme while generating the MEI-I that are defined and explained shortly. The first is the *density* of messages over the course of the entire life cycle of the scheme. The second is the *proximity* rates between selected messages. The third is the *overlapping* rate of overlapping transactions and choreographies. And finally the *capacity* rate of data that hides such a complex scheme among its records.

Density: Typical Ponzi records reflect a tree structure which builds on hierarchical recruits in depth and horizontal recruits in breadth (fan). Through the infant phase, that is early levels, the tree has few recruits and grows extremely fast in its mid way; and towards the end of tree, it starts being unsustainable, and therefore fewer recruits start to appear and consequently lower the spread. The chart in Figure 7.1 illustrates how Ponzi records scatter over the synthetic data. In order to measure the spread, I define density as

the number of records obtained per time interval. In my study, I used 12 hour time intervals to compute density.

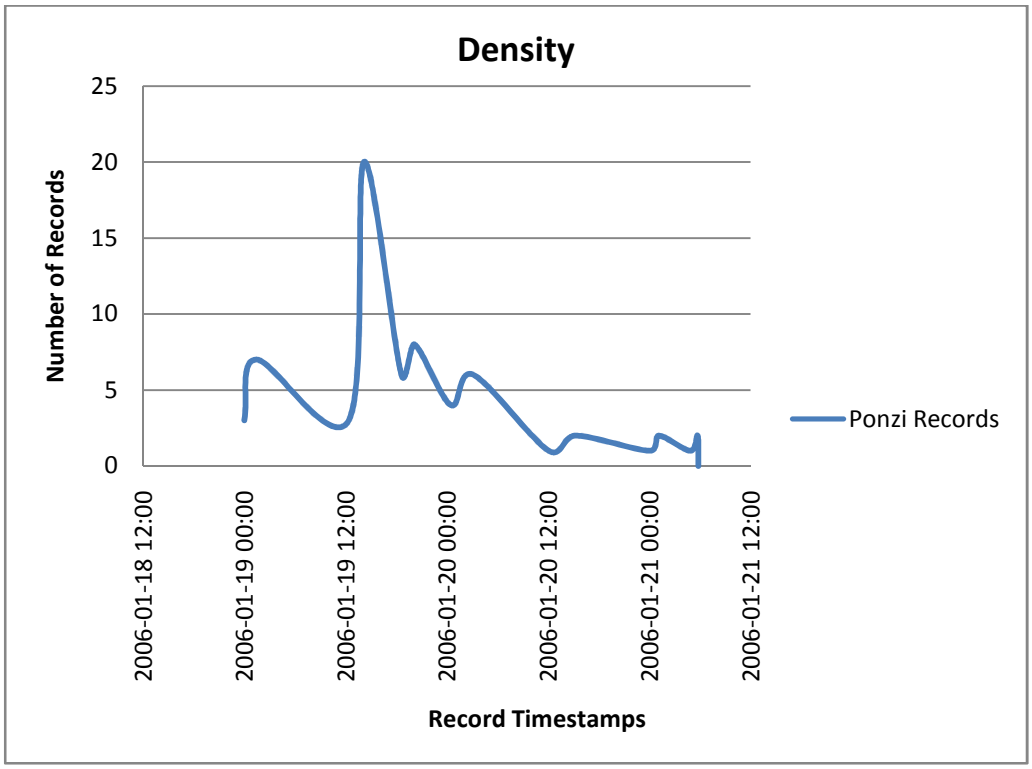


Figure 7.1. Density of Ponzi Records over MEI-I

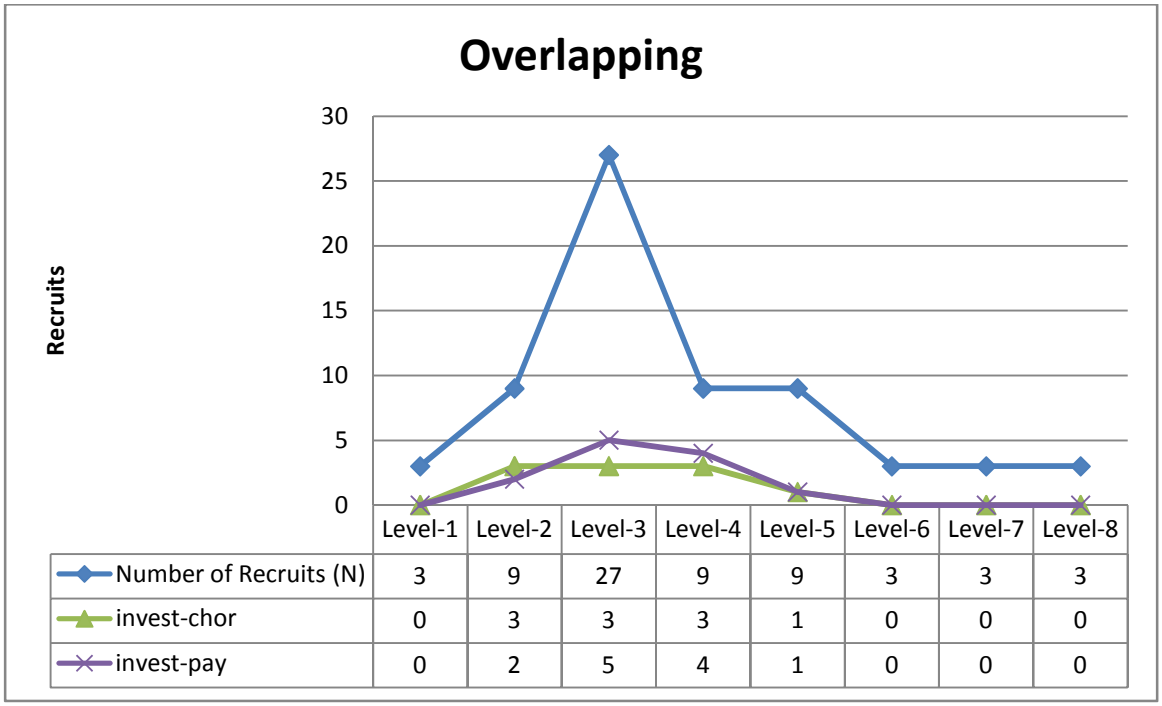
Overlapping: As mentioned, Ponzi records scatter based on fan and depth. While one can observe overlapping recruits in fan, it is impossible to see overlapping recruits over the same recruit paths in depth, because the records are time sequentialized within a path. Assuming R1, R11, and R12 are recruiters; and R1 recruits both R11 and R12, thus constituting a recruit tree of fan 2 and depth 1. For such a recruit tree, while invest-$chor_{R11}$ and invest-$chor_{R12}$ choreography instances can overlap each other, neither invest-$chor_{R11}$ nor invest-$chor_{R12}$ can overlap invest-$chor_{R1}$ choreography instance over MEI records. I measure overlaps simply counting pairs of choreography instances of which

message records cannot be isolated from one another when they are sorted based on their timestamps. The sample overlapping records are shown in Table 7.2. According to the table, while invest-chor$_{O31}$ - invest-chor$_{O33}$ and invest-chor$_{O32}$ - invest-chor$_{O33}$ are overlapping, invest-chor$_{O31}$ - invest-chor$_{O33}$ is an example for totally isolated instances. Therefore, given the table below, I count 2 overlaps. Based on this measurement, I counted the overlaps through MEI-I and obtained the numbers listed in Figure 7.2 for invest-chor and invest-pay choreographies.

Table 7.2. Overlapping Invest Choreography Instances

Time	Sender	Receiver	Msg	Content
2006-01-19 01:28:26	O31	Bank	authorize	invID=31
2006-01-19 01:30:08	O31	O	invest	promoter=O3 invID=31
2006-01-19 01:31:47	Bank	O	confirm	invID=31
2006-01-19 01:32:03	O32	Bank	authorize	invID=32
2006-01-19 01:33:04	O	O31	deliver	invID=31
2006-01-19 01:34:41	O32	O	invest	promoter=O3 invID=32
2006-01-19 01:34:54	Bank	O	confirm	invID=32
2006-01-19 01:35:03	O33	O	invest	promoter=O3 invID=33
2006-01-19 01:35:06	O33	Bank	authorize	invID=33
2006-01-19 01:35:10	Bank	O	confirm	invID=33
2006-01-19 01:35:13	O	O33	deliver	invID=33
2006-01-19 01:35:17	O	O32	deliver	invID=32

Towards the mid life of a Ponzi scheme, when the scheme spreads rapidly the number of overlapping records, are expected to be high. Figure 7.2 shows that recruit levels from 2 to 5 records are so dense and so are their overlapping rates. Notice that the numbers of overlapping records for invest-chor and invest-pay patterns are variables of my detection queries.



	Level-1	Level-2	Level-3	Level-4	Level-5	Level-6	Level-7	Level-8
Number of Recruits (N)	3	9	27	9	9	3	3	3
invest-chor	0	3	3	3	1	0	0	0
invest-pay	0	2	5	4	1	0	0	0

Figure 7.2. Overlapping Ponzi Records over MEI-I

Proximity: I define proximity as distance (seconds) in time between the first message and the last message of related pattern (choreography) instance through MEI records. As I employed previously defined patterns for dynamics of my queries, one essential criteria for their success would be the proximity of pattern instances. For example, if an invest message is unexpectedly further away from a pay message, then this pattern may go undetected, thus leading to higher false negative rates subsequently. Figure 7.3 shows proximity values for each pattern regarding their message evidence indexes over my test data. I show minimum, average, and maximum values in seconds that may further help in determining query parameters in general and window sizes in particular.

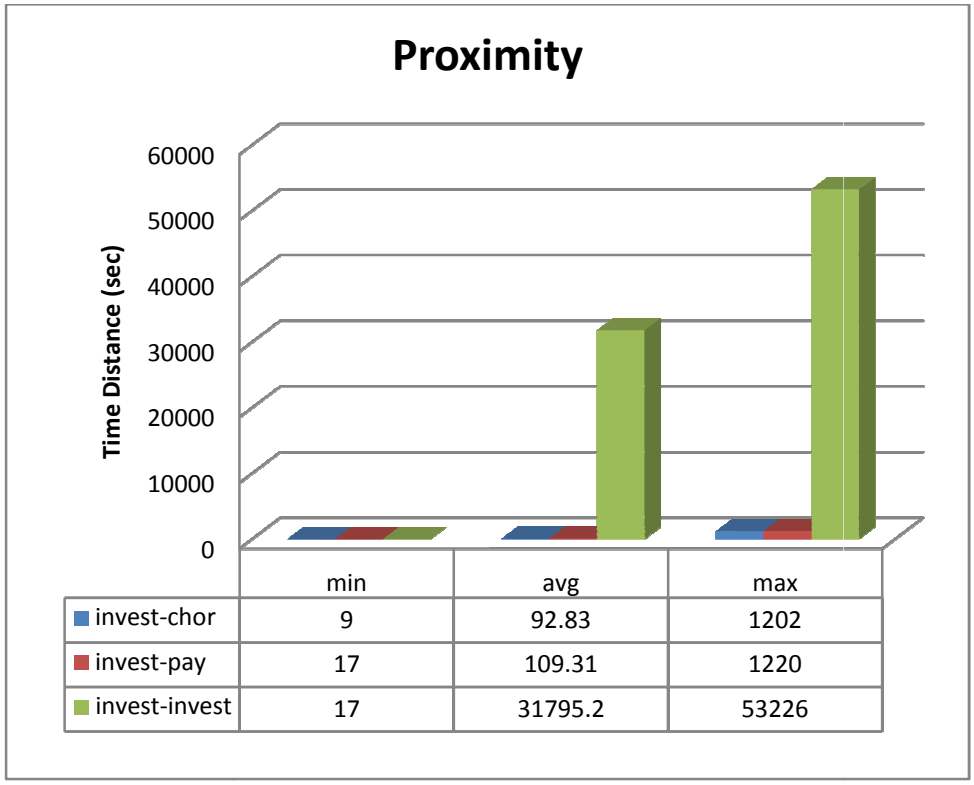


Figure 7.3. Proximity Values of Records over MEI-I

Capacity: The properties I described so far could be helpful in evaluating the accuracy and performance of queries, because they directly effect how those patterns spread over my test records. They, however, have little use in performance evaluation. For performance evaluation, developing a simple data generation code and created three more data sets using the seed data (MEI-I) I originally created. MEI-X is a data set that contains the same Ponzi malicious activity. However it is 10 times bigger than MEI-I in number of records. MEI-XX and MEI-L are other two sets that are 20 and 50 times bigger respectively as shown in Figure 7.4.

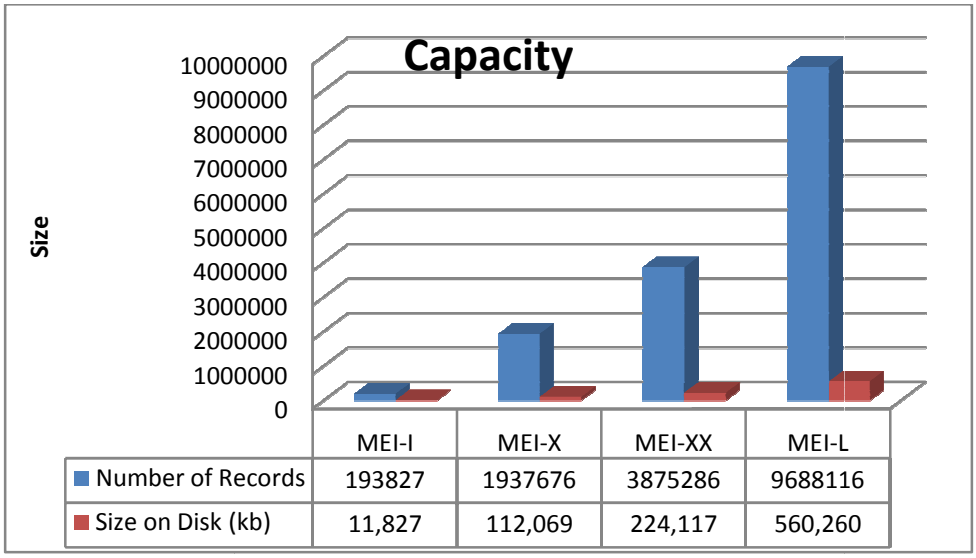


Figure 7.4. Capacity Values of Test Data

7.3. Test Environment

As I mentioned earlier, I defined use/misuse patterns using StreamSQL and subsequently employed StreamBase platform to detect those patterns. I used specially generated data described above in MEI structure. I implemented StreamBase's feed simulation platform accepting those data in CSV files. The platform empowers users to run their StreamSQLs over any user-defined file satisfying the data schema expected by the query. The platform also provides observing outputs of query runs. Using the StreamBase Manager, one can also observe CPU and Memory usage during the course of query executions. Although StreamBase encourages using their enterprise servers for benchmarking and better performance, I observed that their feed simulation platform was adequate to test my queries even over vast amount of data at reasonable resource allocation rate. Table 7.3

shows the details of my test environment, which is identically, used for all further experiments.

Table 7.3. Test Environment

Hardware	
CPU	:Intel Core2 T7400 2.16 GHz, 4MB L2 Cache, 667 MHz FSB
Physical Memory	:2 Gigabyte, 995 MHz
Harddisk	:250 GB, 7200 rpm

Software	
Operating System	:Windows XP SP2
JVM	:SUN JDK 1.5.0.15
StreamBase Studio	:6.4 Version
Max Heap Size	:1024 MB

7.4. Accuracy and Performance

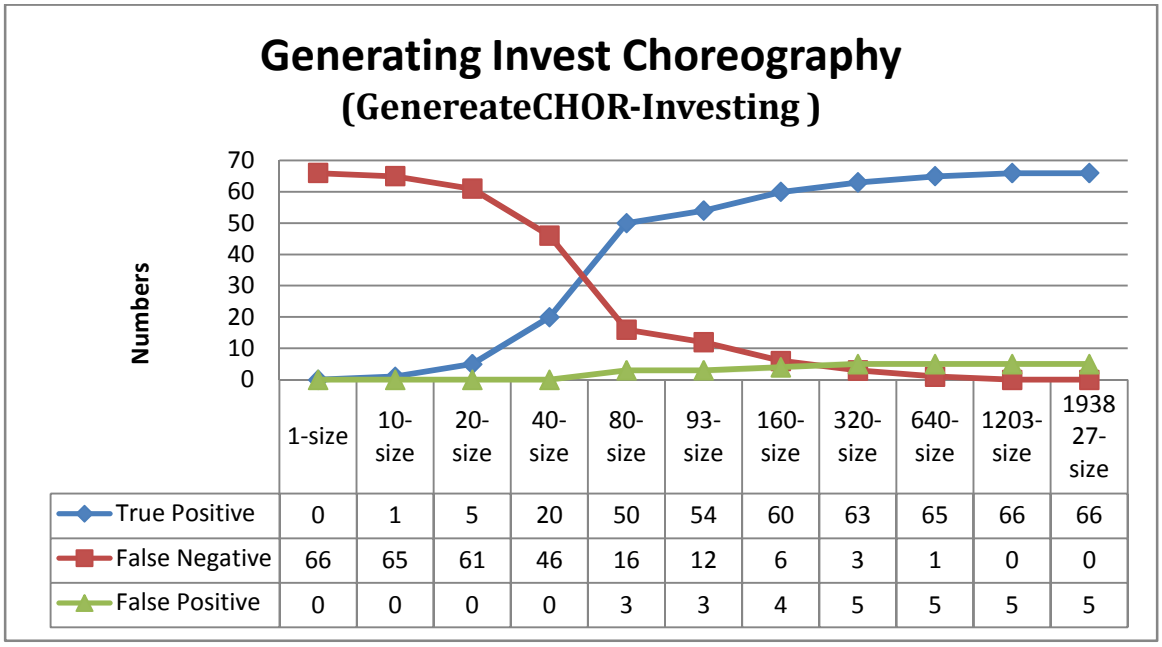
In order to test accuracy and performance using the environment described above, I tested four major queries listed in introduction section over appropriate data set/s.

7.4.1. Testing Accuracy

Effective tuning on queries would lead to more reasonable accuracy rates, such as optimization on window sizes or adding/reducing message attribute equations for approximating queries from property-based patterns to key-based patterns.

Generating Evidence for Invest Choreography: Through the records in MEI-I, I have hidden 66 invest choreography instances of which proximity and overlapping rates have been given earlier in this chapter. After running the **GenereateCHOR-Investing**

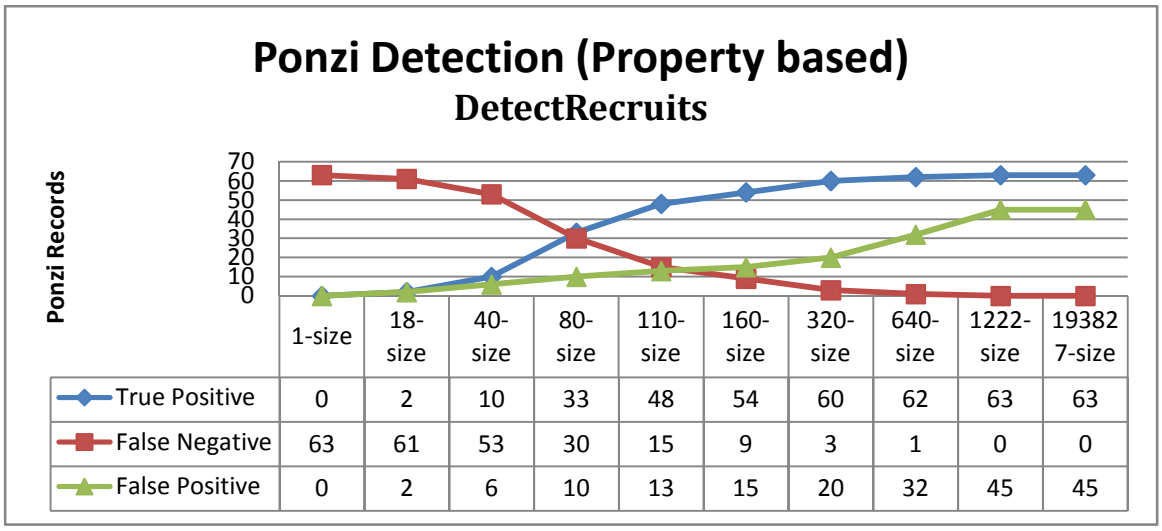
streamSQL given in Chapter 5 for generating evidence of invest choreographies, observed accuracy values were invariant based on the window size used. Figure 7.5 shows those True Positive, False Negative, and False Positive values, which show that bigger window sizes decrease False Negatives while increasing True Positives. I observed that I cannot avoid false positives as long as the number of overlapping invest-chor instances is above 0 with bigger window sizes. False positives were based on overlapping invest choreography instances. Because the query is property based, it is not resistant to overlapping records. When I employed records that are key based and slightly changed the query to detect based on key attributes over records then I observed 0 false positives. This, however, may not be the case in real life all the time.



	1-size	10-size	20-size	40-size	80-size	93-size	160-size	320-size	640-size	1203-size	1938 27-size
True Positive	0	1	5	20	50	54	60	63	65	66	66
False Negative	66	65	61	46	16	12	6	3	1	0	0
False Positive	0	0	0	0	3	3	4	5	5	5	5

Figure 7.5. Accuracy Rates for GenereateCHOR-Investing

Generating Evidence for Ponzi-like recruits: Through the records in MEI-I, I have hidden 63 invest-pay hidden choreographies of which proximity and overlapping rates have been given earlier (notice the values for invest-pay patterns in Figures 7.2 and 7.3) in this chapter. After running the **DetectRecruits** streamSQL given in Chapter 5 for generating evidence of Ponzi-like choreographies observed accuracy values were invariant based on the window size used. Figure 7.6 shows those True Positive, False Negative, and False Positive values, which show that bigger window sizes decrease False Negatives while increasing True Positives. I observed that I cannot avoid false positives as long as the number of overlapping invest-pay instances is above 0 when bigger window sizes are used. False positives were based on overlapping invest choreography instances. Because the query is property based, it is not resistant to overlapping records.

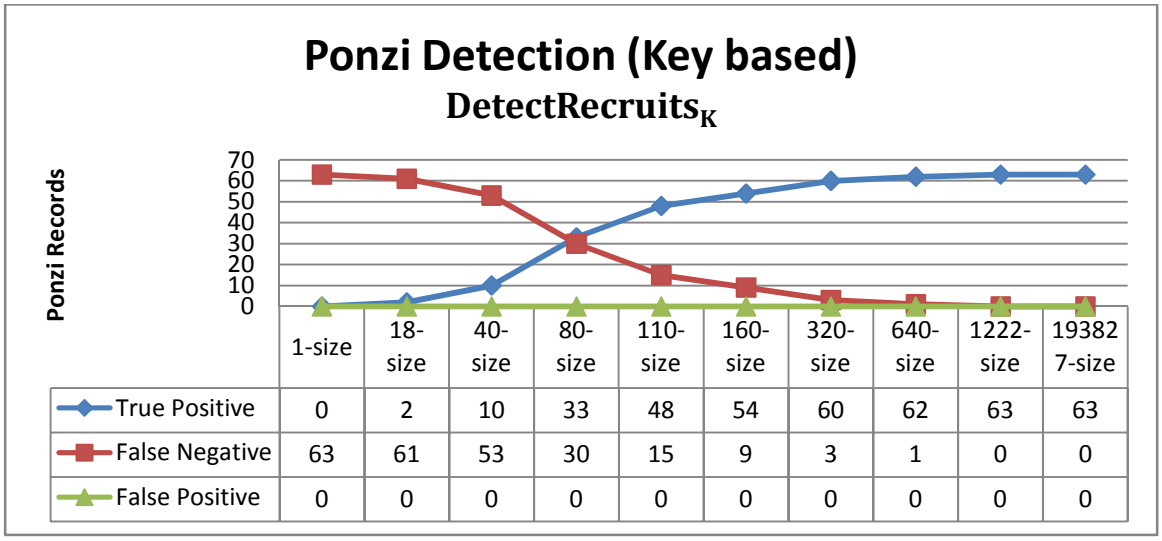


	1-size	18-size	40-size	80-size	110-size	160-size	320-size	640-size	1222-size	193827-size
True Positive	0	2	10	33	48	54	60	62	63	63
False Negative	63	61	53	30	15	9	3	1	0	0
False Positive	0	2	6	10	13	15	20	32	45	45

Figure 7.6. Accuracy Rates for DetectRecruits

When I employed records that are key based and slightly changed the query to detect based on key attributes over records then, I observed 0 false positives. To upgrade Ponzi detection query to a key-based query, **DetectRecruits$_K$**, I added below clause and the results showed as in Figure 7.7. While the original query only matches a receiver of a pay message of the preceding invest message content, this new predicate looks to match a sender of an invest message with subsequent pay message content, thus mutually correlating invest-pay message pair based on a key property. However, this may not be the case in real life all the time.

```
AND regexmatch(".*"+"recruit="+invest.sender+".*", pay.content)
```



	1-size	18-size	40-size	80-size	110-size	160-size	320-size	640-size	1222-size	193827-size
True Positive	0	2	10	33	48	54	60	62	63	63
False Negative	63	61	53	30	15	9	3	1	0	0
False Positive	0	0	0	0	0	0	0	0	0	0

Figure 7.7. Accuracy Rates for DetectRecruits$_K$

Climbing Recruit Path: Starting from any promoter (a.k.a. recruiter), the **ClimbRecruitPath** query climbs up to the early records of the scheme using invest-invest pattern, thus pointing out the orchestrator's first activation. To my observation,

unlike other queries, a missing record through the path prevents the query from computing the path although it is still runs and wastes CPU time.

Generating Recruit Trees: The **GenerateRecruitTree** uses invest-invest pattern as well. The proximity values for this pattern have been shown earlier in this chapter. Because invest-invest records defined in this pattern can never overlap due to its hierarchical (recall $invest_{R1}$; $invest_{R12}$ described earlier) structure. Consequently the false positives are non-existent due to zero overlaps. Running the **GenerateRecruitTree** streamSQL given in Chapter 5 for generating recruit trees there are three properties one can observe regarding the success of such query, *number of recruits* in fan 1 and depth 1 (e.g. a recruiter->recruitee pair) detected, *number of distinct sub trees* varying in fan and depth, *detected recruit rate*, and the *completeness rate* of the tree revealed. Detected recruits are expected to constitute a tree structure. However, missing branches due to partial outputs might create distinct sub trees without creating the main recruit tree. Queries running within small window sizes may generate a large number of unconnected sub recruit trees thereby decreasing the chance to compute the main recruit tree. For example, the query may generate R1->R11, R1->R12, R1->13, R12->R121, and R122->R1222 recruiter->recruitee pairs, where while one can observe first four pairs constitute a tree with four branches in fan 3 and depth 2. The R122->R1222 pair, however, cannot link to the previous tree because of missing R12->R122 recruit, thus constituting an independent tree in fan 1 and depth 1. Therefore, the number of distinct sub trees is 2 for this output. Using number of detected recruits (R) and the total number of recruits (N) of

the main tree, I compute the detected recruit rates (RR=R/N*100) dividing the number of detected recruits by the total number of recruits. I also compute the *completeness rate (CR=RR/T)* by dividing the detected recruit rate by the number of distinct sub trees (T). I introduce the Completeness Rate formula, because neither the detected recruit rate nor the number of distinct sub trees is adequate to measure the ability to completely determine the recruit tree. I observed high rates in detected recruits where the number of distinct and unconnected sub trees were higher as well, thus decreasing the completeness of the generated tree. Table 7.4 shows the results within various window sizes and their corresponding outcomes. The graph in Figure 7.8 also illustrates how completeness rate deviates from detected recruit rate within various window sizes.

Table 7.4. Test Results for GenerateRecruitTree

Recruit Tree (N=63)	Detected Recruits (R)	Distinct Sub-Trees (T)	Detected Recruit Rate (RR=R/N*100)	Completeness Rate (CR=RR/T)
18-size	1	1	1.59	1.59
500-size	4	1	6.35	6.35
5000-size	7	1	11.11	11.11
10000-size	9	1	14.29	14.29
20000-size	14	4	22.22	5.56
31796-size	30	5	47.62	9.52
45000-size	47	4	74.60	18.65
53227-size	63	1	100.00	100.00
100000-size	63	1	100.00	100.00
193827-size	63	1	100.00	100.00

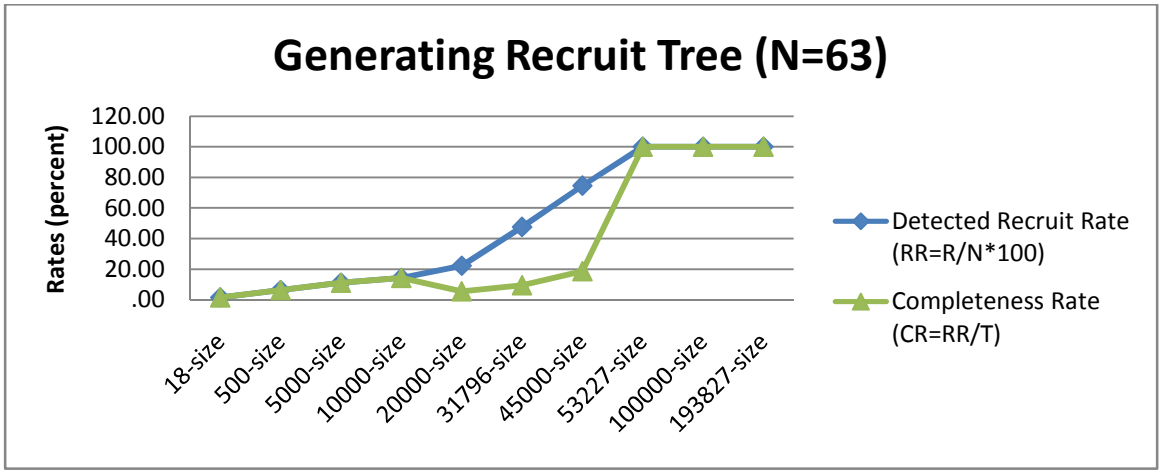


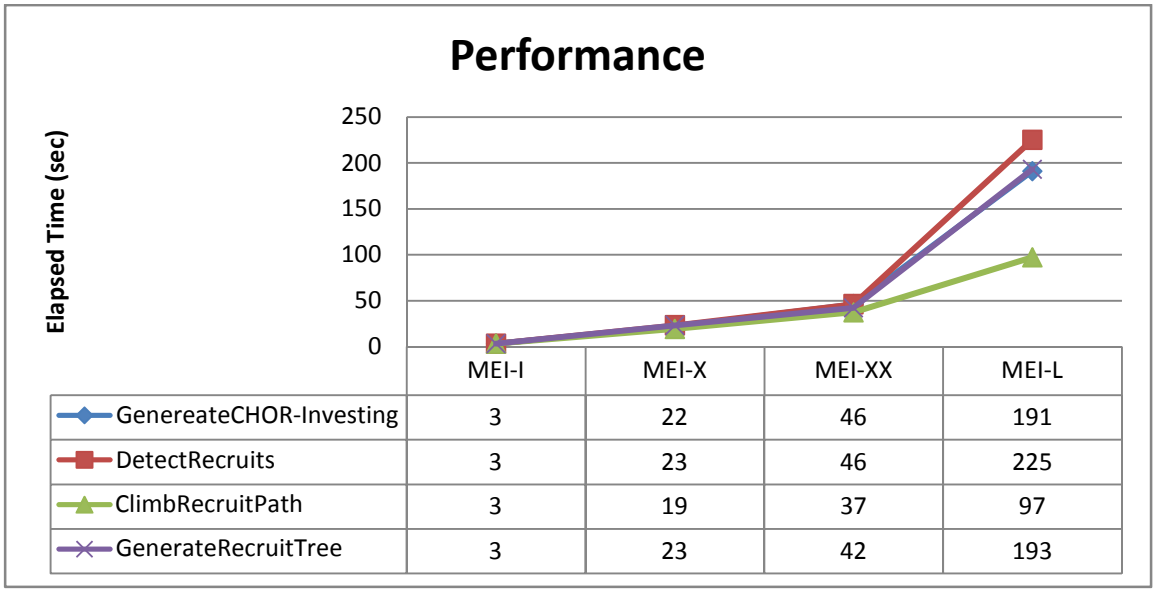
Figure 7.8. Accuracy Rates for GenerateRecruitTree

In terms of optimizations based on window sizes, I observed using a value slightly bigger than maximum proximity value of a pattern gives the best value. During my tests, using window sizes at maximum proximity values gave best results in terms of True Positive and Completeness Rates. However, this is valid when I can correctly estimate this maximum value and when this value is not an outlier.

7.4.2. Testing Performance

As mentioned earlier, I observed maximum proximity rates give the best decision during my tests. Therefore, I built my performance tests upon those values per query. I used the environment described earlier over data sets in different sizes listed in Figure 7.4. According to my tests, queries for generating invest choreographies and recruit trees showed similar results as illustrated in Figure 7.9. The query execution times for detecting Ponzi-like recruits are more than others over the largest data set, MEI-L.

Recruit path climbing algorithm which traverses backward over records showed best performance where there is no need to use larger window sizes.



	MEI-I	MEI-X	MEI-XX	MEI-L
GenereateCHOR-Investing	3	22	46	191
DetectRecruits	3	23	46	225
ClimbRecruitPath	3	19	37	97
GenerateRecruitTree	3	23	42	193

Figure 7.9. Performance Test Results

7.5. Validation Statement

I used StreamBase Studio Feed Simulation platform for performing my tests during the validation. Although there is still higher performance opportunities promised by vendors, StreamBase in particular, I observed my queries reveal reasonable outcomes in terms of performances tests even using this IDE based test environment rather than an enterprise environment. I also observed the queries result in reasonable accuracy values given the specially crafted synthetic data sets. Given the data set, I successfully determined a reasonable window size for window based queries. Using attributes, I also successfully

could converge the evidence outcomes tuning the force of time, property, and key-based patterns directing those queries.

CHAPTER 8

CONCLUSION

8.1. Conclusions

Critical applications, such as those used by business, crave for accountability, which can only be achieved by having of forensically sound evidence. As a successor to business implementations, existing implementations of service-oriented architectures have little promise in providing sound evidences. As I mentioned earlier, for service-oriented architectures I consider evidences as sound only if they are neutral, comprehensive, and reliable because of interdependencies between services and the ability to build global services using composed services.

I extended the existing evidences layer notion of Herzberg et al. to the web services paradigm as a state-of-the-art technology in SOA. The evidences layer proposed by me includes non-repudiation protocols implementing Trusted Third Parties (TTPs) which is unavoidable in achieving neutrality of interactive evidence. They also promise reliability because of the cryptographic backing used during the process. I enhanced the existing notion of evidence collection with the *distributed* collection of evidences residing in

many TTPs that I refer to as FWS-TTP in order to reach comprehensive evidence to address the need to reveal global views of composed service executions. Through this work, I based the collection process on actual log records (LR) and their indexes (LRI) residing on FWS-TTPs rather than on a central repository. Although I was unable to test my distributed collection algorithm, it holds promise in narrowing the scope of examination regarding any incident.

Placing the FWS-TTPs at the bottom I upgraded my evidence management model into a three-layered Evidence Generation Framework (EGF). The new model proposed a central approach in storing and collecting evidences regarding global model executions. I designed endpoint modules that can be used to integrate existing web services without any custom re-engineering and without polluting the existing business logic at endpoints.

Although I explained how my framework can generate pair-wise evidences through prototype architecture, this would have no promise with respect to generating evidences against global models for either use or misuse cases. To generate comprehensive evidences, at the top layer of my framework, I designed queries mining use and misuse case patterns of web service choreographies out of message evidence indexes (MEI) stored at a central repository. I precisely defined the Ponzi/Pyramidal business scheme as a misuse pattern along with corresponding queries. Because it was not possible to obtain real data, I tested queries using a set of synthetic data specifically prepared for revealing the success of queries.

Finally, in order to respond to the detected misuses, as they occurred, I introduced a live detection/prevention/alert model. I first generalized the queries to run in a live environment and address a larger set of misuses. Categorizing the architecture for service and business level designs: I explained how FWS-TTPs should feed upper layers while generating pair-wise evidences at service invocation times, how feedback results are used from service level for prevention, and how business level generates alerts based on live MEIs.

In my opinion, such a framework would help its member web services in many ways, such as providing the basis for many global model specifications and solving disputes among partners. In addition, it might also be an effective platform for revealing ongoing global business misuses and alerting members of their occurrence.

8.2. Future Work

As mentioned during the case study in Chapter 3, the collection of dependent messages and services might traverse over multiple-TTPs, that is, the need may arise to make the EGF scalable all over the Internet. In order to achieve similar tasks, WS-Trust and SAML based federation and delegation mechanisms should be reified over the framework.

I have already mentioned two types of basic evidence based on service level agreements (SLA) in Chapter 4: (1) Evidence of Violation and (2) Evidence of Availability. In addition, many different types of Evidence of Violations could be generated from SLAs and EGFs. Using a predefined format for SLAs, such as WSLA (Web Service Level Agreement) [69], a generic algorithm would be helpful to create evidence of violations in the case of an unexpected behavior of one or more of the endpoints. Some SLA monitor/detection mechanisms [70] could also evidently run over the framework.

Although I have designed an agent using Axis2's extensible handling mechanism, as of the time of this writing, there is no actively running adapter module working for evidence-mindful web services. An industrial effort in creating an "Evidence as Service" concept would reduce the effort in enhancing the reliability of business logic at endpoint services.

APPENDIX

Publications

The contributions of my dissertation have been published/submitted in international venues as listed below:

[1] M. Gunestas, D. Wijesekera, and A. Singhal, "Forensic Web Services," presented in *Fourth Annual IFIP WG 11.9 International Conference on Digital Forensics* Kyoto, Japan, 2008.

[2] M. Gunestas, D. Wijesekera, and A. Elkhodary, "An Evidence Generation Model for Web Services," presented in *The IEEE International Conference on System of Systems Engineering (SoSE '09)*, 2009.

[3] M. Gunestas, D. Wijesekera, and A. Singhal, "Forensics over Web Services: The FWS", Accepted for *Web Services Security Development and Architecture: Theoretical and Practical Issues*, IGI Global, 2009.

[4] M. Gunestas and D. Wijesekera, "Detecting Illegal Business Schemes in Choreographed Web Services: The Ponzi/Pyramidal Case," To-be-submitted to *Sixth Annual IFIP WG 11.9 International Conference on Digital Forensics* Hong Kong, 2010.

[5] M. Gunestas and D. Wijesekera, "Online Detection and Alert Model for Misuses over Web Services," Under review in *CollaborateCom 2009 - The 5th International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 11-14 November, Crystal City, Washington D.C., USA, 2009.

REFERENCES

REFERENCES

[1] K. Mandia, and C. Prosise, *Incident Response: Investigating Computer Crime*: McGraw-Hill, Inc. 2001

[2] D. A. Schum, *The evidential foundations of probabilistic reasoning*. New York: Wiley. 1994

[3] "SOAP Specifications," available at www.w3.org/TR/soap/

[4] "Web Service Definition Language," available at www.w3.org/TR/wsdl

[5] R. Khalaf, N. Mukhi, and S. Weerawarana, "Service-Oriented Composition in BPEL4WS," in *Twelfth International World Wide Web Conference*, Budapest, Hungary, 2003.

[6] S. M. S. Cruz, M. L. M. Campos, P. F. Pires, and L. M. Campos, "Monitoring e-business Web services usage through a log based architecture," in *IEEE International Conference on Web Services Proceedings, 2004.* 2004, pp. 61-69.

[7] A. Herzberg and I. Yoffe, "The Delivery and Evidences Layer," Cryptology ePrint Archive Report 2007/139, 2007.

[8] S. Kremer, O. Markowitch, and J. Zhou, "An Intensive Survey of Non-repudiation protocols," *Computer Communications,* vol. 25, pp. 1606-1621, 2002.

[9] P. Robinson, N. Cook, and S. Shrivastava, "Implementing fair non-repudiable interactions with Web services," in *EDOC Enterprise Computing Conference, 2005 Ninth IEEE International*, 2005, pp. 195-206.

[10] M. Rouached and C. Godart, "Analysis of Composite Web Services Using Logging Facilities," in *Second International Workshop on Engineering Service-Oriented Applications: Design and Composition (WESOA'06)* Chicago, USA, 2006.

[11] S. M. S. Cruz, L. M. Campos, M. L. M. Campos, P. F. Pires, "A Data Mart Approach for Monitoring Web Services Usage and Evaluating Quality of Services," in *XVIII Brazilian Symposium of Data Bases*, Brazil, October 2003.

[12] J. C. Sremack, "Investigating real-time system forensics," 2005, pp. 25-32.

[13] W. D. Yu, P. Supthaweesuk, and D. Aravind, "Trustworthy Web services based on testing," in *IEEE International Workshop on Service-Oriented System Engineering, 2005. SOSE 2005.* 2005, pp. 159-169.

[14] A. Vorobiev and H. Jun, "Security Attack Ontology for Web Services," in *Second International Conference on Semantics, Knowledge and Grid, 2006. SKG '06.* 2006, pp. 42-42.

[15] Y. Demchenko, L. Gommans, C. de Laat, and B. Oudenaarde, "Web services and grid security vulnerabilities and threats analysis and model," in *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.* 2005, p. 6 pp.

[16] D. W. Green, "Attacking and Defending Web Services," in *The NEbraskaCERT Conference*, Omaha, NE USA, 2006.

[17] S. Faust, "SOAP Web Services Attacks," SPI Dynamics, 2005.

[18] W. Negm, "Anatomy of a Web Services Attack," Forum Systems, Inc. 2004.

[19] J. Rosenberg and D. L. Remy, *Securing Web Services with WS-Security*: Sams Publishing, 2004.

[20] M. McIntosh and P. Austel, "XML signature element wrapping attacks and countermeasures," in *Proceedings of the 2005 workshop on Secure web services* Fairfax, VA, USA: ACM Press, 2005.

[21] M. O'Neill, "Hardening Web Services," in *Hardening Network Security* McGraw-Hill Professional, 2005.

[22] R. Dwibedi, "XPath injection in XML databases," 2005.

[23] A. Singhal and T. Winograd, "Guide to Web Services Security," National Institute of Standarts and Technology 2009.

[24] Z. Jianyin, S. Sen, and Y. Fangchun, "Detecting Race Conditions in Web Services," in *International Conference on Internet and Web Applications and Services/Advanced International Conference on Telecommunications, 2006. AICT-ICIW '06.* 2006, pp. 184-184.

[25] W. Du, *Cross-Site Scripting (XSS) Attack Lab*: Laboratory for Computer Security Education, Syracuse University. 2006

[26] R. Salz, “Understanding XML Digital Signature,” available at http://msdn.microsoft.com/en-us/library/ms996502.aspx, 2003

[27] IBM, "JAX-RPC handlers collection," WebSphere Application Server Information Center 2007.

[28] ORACLE, "Using JAX-RPC Handlers," Oracle® Application Server Web Services Developer's Guide B28974-01, 2006.

[29] BEA, "Specifying SOAP Handlers for a Web Service," BEA WEBLOGIC WORKSHOP HELP.

[30] "Apache Axis2 Architecture Guide," The Apache Software Foundation 2006.

[31] P. Srinath, H. Chathura, E. Jaliya, C. Eran, R. Ajith, J. Deepal, W. Sanjiva, and D. Glen, "Axis2, Middleware for Next Generation Web Services," in *International Conference on Web Services, 2006. ICWS '06.* 2006, pp. 833-840.

[32] WS-Trust V1.0 Working Draft, OASIS Web Services Secure Exchange TC, http://www.oasisopen.org/committees/download.php/16138/oasis-wssx-ws-trust-1.0.pdf, 2006

[33] OASIS, "WS-SecureConversation 1.3," 2007.

[34] H. Chet, "Digital evidence bag." vol. 49: ACM Press, 2006, pp. 69-70. 28

[35] M. B. Juric, *Business Process Execution Language for Web Services*, Second Edition ed. Birmingham, UK: Packt Publishing, 2006.

[36] S. T. King and P. M. Chen. “Backtracking Intrusions,” in *Proceedings of the 2003 Symposium on Operating Systems Principles (SOSP)*, pages 223-236, October 2003.

[37] W. Wang and T. E. Daniels, "Building evidence graphs for network forensics analysis," in *Computer Security Applications Conference, 21st Annual*, 2005, p. 11 pp.

[38] US Department of Justice, “Digital Evidence: Standards and Principles,” Forensic Science Communications, available at http://www.fbi.gov/hq/lab/fsc/backissu /april2000 /swgde.htm

[39] T. Coffey, P. Saidha, “Non-repudiation with mandatory proof of receipt,” ACMCCR: Computer Communication Review 26.

[40] J. A. Onieva, Z. Jianying, M. Carbonell, and J. Lopez, "Intermediary non-repudiation protocols," in *IEEE International Conference on E-Commerce, 2003. CEC 2003.* 2003, pp. 207-214.

[41] M. Bilal, J. P. Thomas, M. Thomas, and S. Abraham, "Fair BPEL processes transaction using non-repudiation protocols," in *2005 IEEE International Conference on Services Computing,* 2005, pp. 337-340 vol.1.

[42] C. M. Jayalath and R. U. Fernando, "A Modular Architecture for Secure and Reliable Distributed Communication," in *The Second International Conference on Availability, Reliability and Security, 2007. ARES 2007.* 2007, pp. 621-628.

[43] K. Shanmugasundaram, N. Memon, A. Savant, and H. Bronnimann, "ForNet: A Distributed Forensics Network," in *Proceedings of the Second International Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security*, St. Petersburg, Russia, 2003.

[44] S. Micali, "Certified E-mail with invisible post offices," presented at the RSA '97 conference (1997).

[45] W. Xu, D. Chadwick, and S. Otenko, "A PKI Based Secure Audit Web Service" in *IASTED Communications, Network and Information and CNIS*, Phoenix, USA, November 2005

[46] R.E. Johnson, and B. Foote, "Designing reusable classes," in Journal of Object-Oriented Programming 1, 5 (June/July 1988), 22-35.

[47] M. Fowler and K. Scott, *UML distilled: applying the standard object modeling language*: Addison-Wesley Longman Ltd., 1997.

[48] B. Catriel, E. Anat, M. Tova, and P. Alon, "BP-Mon: query-based monitoring of BPEL business processes." vol. 37: ACM, 2008, pp. 21-24.

[49] L. Ardissono, R. Furnari, A. Goy, G. Petrone, and M. Segnan, "Monitoring Choreographed Services," in *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*, 2007, pp. 283-288.

[50] M. Zuckoff, *Ponzi's scheme : the true story of a financial legend*. New York: Random House, 2005.

[51] U.S. Security and Exchange Commission, "'Ponzi' Schemes," available at http://www.sec.gov/answers/ponzi.htm

[52] SavvySugar. “Help! My Friend Is Part of a Pyramid Scheme,” in *MainStreet*, available at http://www. mainstreet.com/article/smart-spending/help-my-friend-part-pyramid-scheme, 2009.

[53] R. M. Steward, “South Africa investigates alleged pyramid scheme,” in *The Wall Street Journal*, available at http://online.wsj.com/article/SB12450772607 9615295.html, 2009.

[54] Los Angeles Times, “Internet Pyramid Scheme Alleged,” available at http://articles.latimes.com/2006/sep/28/ business/fi-wrap28.3, 2006.

[55] D. A. Valentine, "Pyramid Schemes," in *International Monetary Fund’s Seminar on Current Legal Issues Affecting Central Banks*, 1998

[56] M. Jensen, N. Gruschka, and N. Luttenberger, "The Impact of Flooding Attacks on Network-based Services," in *Third International Conference on Availability, Reliability and Security, 2008. ARES 08.* 2008, pp. 509-513.

[57] A. Barros, G. Decker, M. Dumas, and F. Weber, "Correlation Patterns in Service-Oriented Architectures," in *Fundamental Approaches to Software Engineering*, 2007, pp. 245-259.

[58] J. M. Zaha, M. A. Dumas, H. M. ter Hofstede, A. Barros, and G. Decker, “Bridging Global and Local Models of Service-Oriented Systems,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 38, pp. 302-318, 2008

[59] StreamSQL, available at http://blogs.streamsql.org/

[60] StreamBase Technical Documentation, available at http://www.streambase.com

[61] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison–Wesley Publishing Company, Reading, MA, 2002.

[62] A. Widder, R. v. Ammon, P. Schaeffer, and C. Wolff, "Identification of suspicious, unknown event patterns in an event cloud," in *Proceedings of the 2007 inaugural international conference on Distributed event-based systems* Toronto, Ontario, Canada: ACM, 2007.

[63] W. De Pauw, R. Hoch, and Y. Huang, "Discovering Conversations in Web Services Using Semantic Correlation Analysis," in *IEEE International Conference on Web Services, 2007. ICWS 2007.* 2007, pp. 639-646.

[64] C. Westphal, *Data Mining for Intelligence, Fraud & Criminal Detection: Advanced Analytics & Information Sharing Technologies*. Frederick, Maryland, USA: CRC Press. 2008.

[65] Open Web Application Security Project, "Cross Site Scripting Prevention Cheat Sheet," available at http://www.owasp.org/index.php/XSS_Cross_Site_Scripting_Prevention_Cheat_Sheet

[66] S. Basu, F. Casati, and F. Daniel, "Toward Web Service Dependency Discovery for SOA Management," in *Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 2*: IEEE Computer Society, 2008.

[67] N. Tho Manh, S. Josef, and A. M. Tjoa, "Sense \& response service architecture (SARESA): an approach towards a real-time business intelligence solution and its use for a fraud detection application," in *Proceedings of the 8th ACM international workshop on Data warehousing and OLAP* Bremen, Germany: ACM, 2005.

[68] I. Ari, J. Li, A. Kozlov, and M. Dekhil, "Data Mining Model Management to Support Real-Time Business Intelligence in Service-Oriented Architectures" in *HP Software University Association Workshop*, Morocco, June 2008.

[69] A. Keller and H. Ludwig, "The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services," *Journal of Network and Systems Management,* vol. 11, pp. 57-81, 2003.

[70] A. Sahai, V. Machiraju, M. Sayal, A. van Moorsel, and F. Casati, "Automated SLA Monitoring for Web Services," in *Management Technologies for E-Commerce and E-Business Applications: 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2002, Montreal, Canada, October 21-23, 2002. Proceedings*, 2002, pp. 28-41.

CURRICULUM VITAE

Murat Gunestas is a major at General Directorate of Security in Ankara, Turkey. He has eleven years of experience in the field of computer and information systems. His general research interests include web services security, computer and network forensics, and component-based software engineering. In addition to the PhD from George Mason University, he has Bachelor of Science degree in Security Science and Master of Science degree in Software Engineering. He developed software for General Directorate of Security by 2003 and lead software teams afterwards. He currently holds software project lead position at the same department along with appointments of deploying appropriate security policies.