Online Detection of Web Choreography Misuses

Murat Gunestas and Duminda Wijesekera

Abstract—**Web service choreographies can be misused or abused in two different ways. The first is to exploit the underlying choreography to attack web services, that we call service misuses. Instantiation flooding is one such attack. The second is using web service choreographies to conduct illegal businesses, that we call business misuses. Ponzi and Pyramidal schemes belong in this second category. Due to the web service interdependencies created in choreographed web services both these kinds are difficult to detect. We show an online detection system that would generate alerts of both these kinds of misuses.**

Index Terms—**Service oriented architecture; web service; choreography; service misuse; business misuse; detection; alert; sense and response**

I. Introduction

DYNAMIC service invocations and message content specific operations among choreographed web servers are being deployed across many industries. Consequently, exploiting these dynamic inter-dependencies could lead to a new class of mal-actors – namely those that exploit these new features and those that use them to conduct illegal business schemes.

In addition to legal business schemes, illegal schemes can be implemented in service oriented architecture. Such schemes would be difficult to understand through microscopic observation because most of the participants of such schemes would be unaware of the macroscopic business scam.

There can be many illegal schemes. Two popular ones—pyramid and Ponzi schemes—are difficult to differentiate from Multi Level Marketing that either runs real business or invests in others. However, the basic dynamics of Ponzi/pyramid schemes is *rob Paul to pay George*. Ponzi [1] ran such a scheme in recent history. For many years he collected money and gave papers back promising return on investment in 90 days. He actually returned the money to the early investors using the money invested by late joiners. He never ran a business that produced any profit or incurred any loss because he did not invest in any other business.

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 work reported in [2] proposes using 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 a 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. Online detection capabilities provide an opportunity to prevent endpoint services from engaging in identifiable malicious activities, such as dataflow or denial of service attacks.

Through the paper, we describe how web service choreographies can be misused in service oriented architectures in Section 2. Section 3 describes our framework for online detection of choreography misuses. In Section 4 we describe potential software design architectures for the proposed framework. Section 5 describes our online detection model and different types of misuses that can be detected in real time. Section 6 describes an alert model that can warn potential intended endpoints, such as potential investors that can be affected. Section 7 discusses related work and Section 8 concludes the paper.

II. Choreography Misuses

We consider misuses in choreographed web services twofold: *business level choreography misuses* and *service level choreography misuses*.

A. Business Misuses

As described earlier, we here mention that semantic capabilities of web services can be exploited by mal-actors in several ways. The mal-actors may even be orchestrators of large business schemes that abuse choreography models, or they can play the role as a partner in a choreography that becomes malicious, and thus, deviating from the expected choreographic behavior.

Deviating choreographies are those in which one or more parties behave contrary to the specification. Such behaviors may work for some partners' benefit. For example, a travel agency may serve their best offer list in favor of chosen hotels, car-rental companies, etc. They may also behave in a double-faced way to conceal the misbehavior. The external behaviors of services have to be observed in order to detect such behaviors.

B. Service Misuses

Here we mention the exploits of some design flaws on static choreographic models and the importance of detecting those

Manuscript received August 1, 2009.

Murat Gunestas was with George Mason University. He is now with the General Directorate of Security, Ankara, Turkey, (phone: +90 312-412-1709; e-mail: mgunestas@egm.gov.tr).

Duminda Wijesekera is with the Computer Science Department, George Mason University, Fairfax, VA 22030 USA, (e-mail: dwijesek@gmu.edu).

exploits even if they are unavoidable. Mal-actors in those cases abuse the inter-dependency spanning over the web services employed by static choreography models.

A *Dataflow attack* is a special type of service level exploit that is difficult to detect because the attacker leaks malicious code into services with good data. Unless the services check the content, malicious data would pass between the systems. [3] Describes a XSS (cross site scripting) attack scenario that leaves a stepping stone as a suspected attacker. Such a scenario again increases the need for externally observable messages flows and exculpating the stepping stones from unjust accusations. Some receiver web services, may also consider detection, prevention, or using an online alert mechanism.

Instantiation flooding [4] is a typical denial of service (DoS) attack that can be launched on web service compositions. Here, the attacker repeatedly invokes the *receiver* operation of a process at the target web service. In response, the target services engine attempts to instantiate every request, thereby creating a DoS attack. Mostly, the hierarchical service compositions may experience this attack. In such a case, an attacker that knows the involved service interdependencies can target a specific process. The attacker floods the process and thereby alters the state of the choreography engine, leading to resource-consumption on the engine. More lethal attackers may target a different web service that runs another process invokable by a stepping stone service. As designed, the stepping stone may invoke the process that damages the actual target. After flooding both services either of these would crash; in any case, the actual attacker can blame the stepping stone. Conversely, the web services that are in need of maximum throughput may consider a detecting and/or preventing such misuses.

III. Overview of Evidence Generation Framework

In order to facilitate and base evidence generation on a reliable infrastructure that can convince the services that want accountability on their transactions and fast detection when misused, we proposed designing an *Evidence Generation Framework (EGF)* that preserves appropriate evidence to recreate the composed web service invocations independent of the partners of the transaction. This would have a greater chance of being accepted at any stage of disputes resolution. EGF is based on a three level model, as shown in Figure 1. The bottom layer provides pair-wise evidence. The middle layer derives new evidences from pair-wise evidence. The purpose of these derivations is to either refute or justify claims of SLA violations or to prove violations of their partners. EGF will provide on-line evidence management capabilities to other web services as a web service itself. To gain these capabilities, the EGF would be integrated (see *WS-Evidence interface* and *evidence adapter modules*) with *endpoint web services* that require their services – referred to as member web services of EGF. In order to do so, it provides a centralized service access point to its customer web services. This information, retained by EGF acting as a trusted third party, can be directly given to their members, assigned arbiters, or forensic examiners.

In order to be successful, other web services need to use our *evidence generation framework* as a neutral third. If all parties subscribe to our proposed service, the evidence layer establishes communication channels transparent to other parties to collect and preserve communications at the bottom layer of our framework. In order to do so, we design evidence adapters that intercept invocation from/to the services and forward them to the evidence framework. In our previous work, we stored communication evidence in cryptographically secure repositories in order to generate and derive evidence to substantiate or refute claims of misbehavior [3]. In this version, we upgrade them to function online by caching evidence streams and querying these caches. To do so, the bottom layer passes evidence indexes into upper layers at service invocation time, thus feeding these services with live evidences for mining evidence of complex actions out of them. Having employed cache based live queries onto those layers we propose a twofold response model: alert and prevent (see dashed arrows in Figure 1).

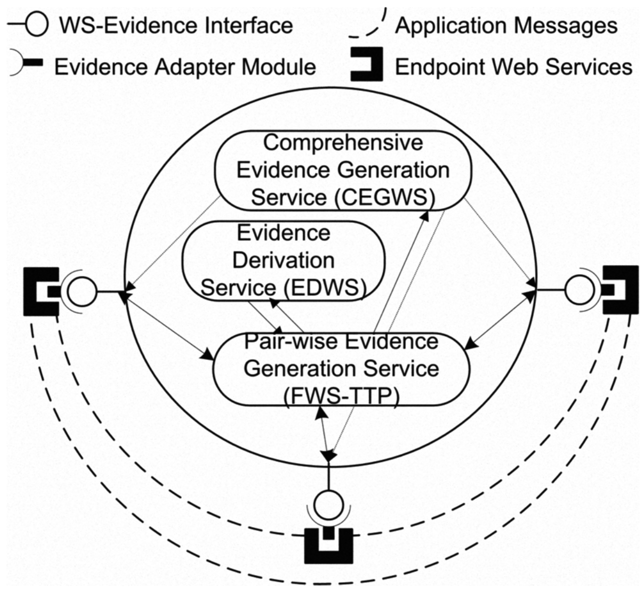
WS-Evidence Interface
Application Messages
Evidence Adapter Module
Endpoint Web Services
Comprehensive Evidence Generation Service (CEGWS)
Evidence Derivation Service (EDWS)
Pair-wise Evidence Generation Service (FWS-TTP)

Fig. 1. The EGF in online mode. Two sided arrows illustrate Pair-wise Evidence Generation messages while one-sided arrows depict invocations between TTP (Trusted Third Party) and both EDWS and CEGWS at service invocation time that lead immediate alerts for endpoint services and TTPs.

A. Pair-wise Evidence Generation

Our previous work [5] shows a prototype implementation of the pair-wise evidence generation model and presents many protocols based on two types of message exchange patterns (MEP): One-Way and Request-Response. There, we provided *non-repudiation*, *fairness*, and *timeliness* in our pair-wise evidence. We used digital signatures to provide proof of receipt and delivery, link a message to its creator/sender and provide message integrity.

For accountability, we use fair non-repudiation mechanisms that utilize Trusted Third Parties (TTP because, although there are fair exchange protocols for two participants that do not use a TTP (for e.g. Markowitch [6]), these protocols assume that the participants have prior knowledge of the message contents. We do not use them because web services may not always know expected contents prior to receipt. We require timeliness because of the time sensitive nature of most business transactions. We base our 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)* 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.

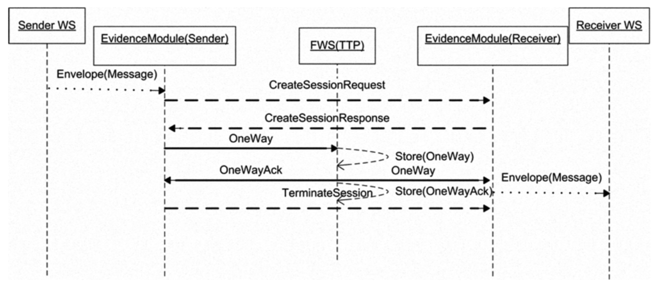

Fig. 2. The sequence diagram illustrates Pair-wise Evidence Generation. Notice the scenario is one-way using inline TTP.

Figure 2 illustrates a one-way SELP protocol and follows the steps below:

1. The evidence module in the Sender side intercepts the request of an envelope and pause the message.
2. Creates and sends a *CreateSessionRequest* to the receiver web service for the target operation.
3. On receipt, the evidence module on the receiver side receives creates and sends a response message back to the sender. It also creates a session.
4. Sender sends a one way (<#session| #message |#signature$_{\text{Sender-K}}$ (#session|"4"|# env)>) message to the TTP.
5. The TTP receives the one way message, stores the message, and forwards it to the intended receiver. It also creates, signs, stores, and sends OneWayAck (<#session|#message|#signature$_{\text{TTP-K}}$ (#session|"5" |#env)>) to the sender.
6. The receiver's side evidence module intercepts the message, validates and extracts the message envelope to release it to the expected receiver operation.
7. Sender's evidence module creates and sends a *TerminateSession* message to the receiver web service for the related session, thus, terminating the session.

B. 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 we rely on the evidences stored at TTPs. Evidences gathered this way can be used by a web service to exculpate it from accusations. Depending upon the service level agreements, the number of evidences would increase. Our previous work [5] explains samples for evidence of violations against time-out agreements and scheduled invocations between two endpoints. We here demonstrate how Evidence Derivation Services (EDWS) can derive evidences regarding service level online misuses, thus leading to immediate feedbacks to the bottom layer for probable prevention.

C. Comprehensive Evidence Generation

The top layer can use a rule engine or 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 by deriving the evidences at the second layer. Through the paper, we demonstrate how online evidences of complex scenarios can be mined from evidence of observed interactions of pair-wise communications. We generate alerts for potential victims, investigators, or arbiters of such global misuses so that they can take immediate actions.

D. WS-Evidence Layer

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

--A new layer called WS-Evidence is along with the XML schema for used messages and stored records.

-- A specific evidence generation adapter module to re-route all transactional messages of member web services of the FWS framework through FWS servers. Those adapters absorb WS-Evidence alert messages so that the endpoint web service can take appropriate actions

--A specific evidence generation adapter module to re-route all transactional messages of member web services of the FWS framework through FWS servers. Those adapters absorb WS-Evidence alert messages so that the endpoint web service can take appropriate actions.

--The underlying layer to WS-Evidence should be applied to provide a trust base and cryptographic services (e.g. any WS-Security module).

--Specific patterns and the queries are designed to provide evidence generation at various levels. Online versions of those detection queries are designed for generating evidence against service misuses and global misuses.

--An alert and prevention model is designed to keep relevant member services up-to-date regarding misuse activities currently emerging.

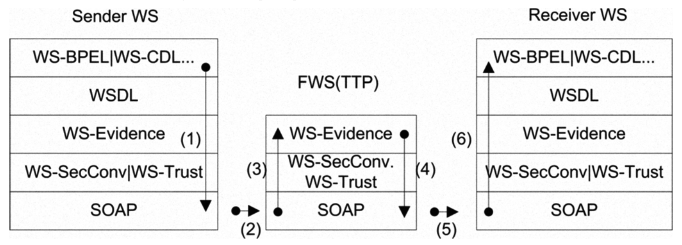

Fig. 3. WS-Evidence as a layer for web services stack. Notice that security specifications such as WS-SecureConversation and WS-Trust underpin WS-Evidence as well as it can underpin any upper level specifications (e.g. WS-BPEL for business processes and WS-CDL for choreography modeling) represented by WSDL (Web Service Definition Language).

Figure 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 wire communication at SOAP level; and 3 and 4 represent inputs and outputs in TTP. Therefore, TTPs can manage the protocols between web service pairs.

E. Delivery Process for Online EGFs

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

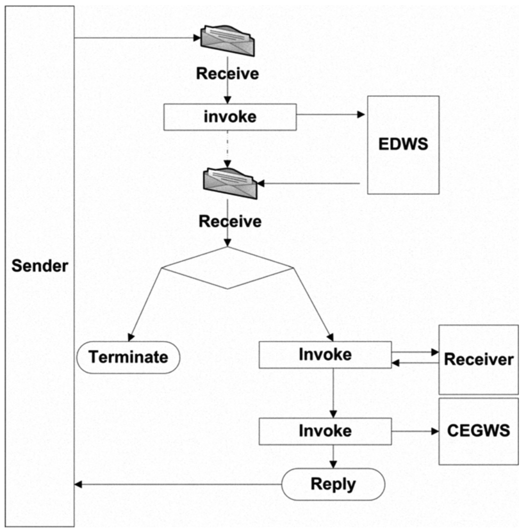

Fig. 4. Deliver process running at TTP for online EGF. Notice activities invoking both EDWS and CEGWS for online detection.

In order to enrich the EGF with online capability, we propose enhancing the pair-wise evidence generation process at TTPs (remember step 5 in section III.A). The pseudo BPEL diagram in Figure 4 shows a typical delivery process connecting to online services at service invocation time, thus leading to online detection, prevention, or alert mechanisms on demand.

The pseudo BPEL process above shows sender-receiver interactions and live detection invocations rather than detailed activities through the process. The delivery process at TTPs extracts each message received (notice the first *Receive* in Figure 4) in MEI format where *sender* and *receiver* fields are 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 FWS stations. In order to glue the process to online services for detection, prevention, and alerting we enrich the process as described above. That is, we forward MEIs into EDWS (notice the *Invoke* after first *Receive*) and CEGWS (notice the *Invoke* before *Reply*) thus empowering the EGF framework live detection capabilities.

TABLE I
MESSAGE EVIDENCE INDEX TABLE (MEI)

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

IV. ONLINE EGF ARCHITECTURE

The EGF introduces two potential architectures for online detection, prevention, and alerting. The first is for business misuses, thereby employable remotely on a central system that gathers all messages. The second is for service misuses, employable at TTP stations mostly for prevention purposes or marking the malicious activity for detection purposes.

A. Business Level Design

A central online web service called CEGWS generates comprehensive evidence of business misuses. As shown in Figure 5, during alerts, CEGWS collects Message Evidence Index records from TTPs at their service invocation times. CEG alert clients send alerts to relevant web services that could become a potential victim.

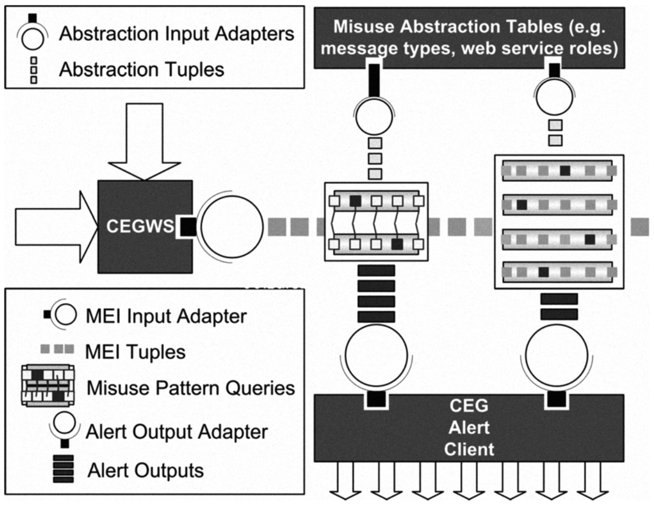

Fig. 5. Online CEGWS architecture. Notice MEI tuples are processed along with abstraction tables for the sake of more generic detection of misuses.

The *CEG (Comprehensive Evidence Generation) Web Service* collects MEI records from other FWS-TTP stations. These records are stored and sent to MEI input adapters to be processed for evidence of business misuses or generating other comprehensive evidence. The web service accepts one way WS-Evidence messages including MEIs from TTP stations.

The **MEI Input Adapter** is a live input adapter accepting SOAP messages and emit MEI tuples in real time into applications that detects misuses. As the StreamBase input adapter, this API (e.g. JMS, IBM MQ) can extract index records from WS-Evidence SOAP messages and pass the MEI records into StreamSQL for querying. **MEI tuples** are produced by MEI input adapters. The **Sort Operator,** using an on-demand window size, sorts incoming records before they are fed to StreamSQL based queries.

Misuse Pattern Queries are based on a particular pattern query language, streamSQL, and as shown earlier, discover misuse. **Alert outputs** are mostly designed in SELECT parts of streamSQL queries that mine for 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 be 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 a misuse is detected. It creates a SOAP message stream with endpoint web services using WS-Evidence alert messages. It fires one-way alert messages as including the details described below.

Alert messages contain information related to a business misuse. Business misuses may be in several types (misuse_type) and each types of misuse may feature various schemes (misuse_code). We, therefore, propose coding each scheme distinctly. Each misuse scheme consists of one or more malicious actors, and can be of the following form.

{misuse_type, misuse_code, {mal_actor1, mal_actor2…}}

Evidence modules at endpoints or specifically designed XML firewalls can absorb the alert messages in the above format. They can take immediate actions such as creating a rule ignoring messages coming from suspected malicious actors or creating a black list of web servers.

B. *Service Level Design*

A local online evidence derivation web service, that we refer to as the EDWS generates service level evidences. As shown in Figure 6, it receives the envelopes from the deliver process at TTPs prior to their service invocations, and prevents or marks relevant messages. EDWS Alert clients send alerts to CEG web services so that they might be used in further investigations.

We propose an application level prevention model, that is, the Deliver process conducts early detection by invoking the EDWS. The process pushes messages into locally implemented EDWS that run StreamSQL queries that in turn can invoke a local detection service that marks suspected messages.

As described in Figure 4, one option is to terminate suspicious instance or branches of the choreography or continue with the rest of the process. However, in either case, an alert message is produced and sent to CEGWS for further investigation. This is done by the alert client as illustrated in Figure 6.

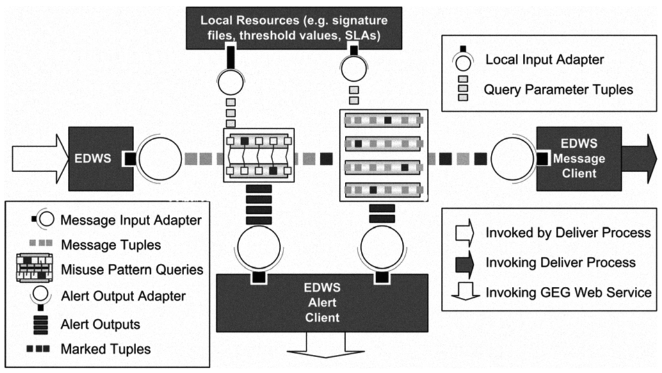

Fig. 6. Online EDWS architecture. Notice that local resources are involved in the process of detecting service misuses.

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 lookup in 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 receives activity (recall Figure 4) in the process.

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

V. The Online Detection Model

Most intrusion detection either detects anomalies or misuses, of which we use the latter. In addition, we categorize the misuses over web services. We 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*.

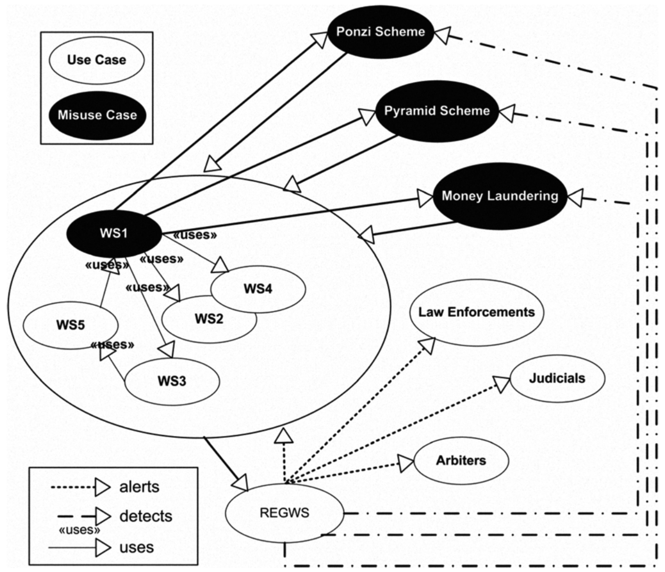

Fig. 7. Business misuse case. Notice a malicious service (WS1) misuses others using particular illegal business schemes. Timely invoked CEGWS detects the malicious activity and produces alerts.

A. Online Detection of Business Misuses

Web services build choreographies and perform illegal business activities such as Ponzi schemes, pyramid schemes, or global money-laundering schemes. Our detection model looks for specific misuse patterns featuring such activities. We propose a heuristic algorithm for discovering Ponzi-like misuse of choreographies in our previous work [2]. However, we need more abstract patterns to detect business level misuses without being dependent on any specific scenarios.

Abstracting Misuse Patterns

The query proposed in [2] is concrete and can be strictly applied to a specific domain of misuses. For example, our heuristically discovered patterns would apply only to one specific orchestrator of the scheme. Here, we propose abstracting those concrete patterns to address more misuses that may occur than those based on a single orchestrator. To achieve this, we use role based abstraction to generalize the endpoint web services' involvement in the service choreography. We also use type based abstraction to generalize messages flowing between endpoints. In order to determine the roles and types of web services and messages pertain, we 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 time. Using these techniques, we create type and role tables for messages and web services, respectively. Through this paper, we assume we already have those classification tables and PEG services having access those tables.

Mapping Messages to Types

SOAP envelopes carry message names. But different XML schemas may use different names for the same entity, such as a pay activity being called "payInput" or "sendPay" in Table II. Hence, we use a type table that is a one-to-many mapping of names to types.

TABLE II
MESSAGE TYPE TABLE (MTT)

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

Given Table II and a MEI record, instead of MEI.msg=="invest" or MEI.msg=="pay", we abstract invest and pay message types using the following pattern queries:

MTT.message==MEI.msg AND MTT.type=="Invest"
MTT.message==MEI.msg AND MTT.type=="Pay" (1)

Mapping Web Services to Roles

Similarly, we need to map we services to roles for which we use a role table that maps a service to its potential roles. A (WSRT) is a one-to-many mapping of web service to roles. As shown in Table III, service A acts as an investment company accepting invest messages from investors and as an investor investing in others.

TABLE III
WEB SERVICE ROLE TABLE (WSRT)

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

Given Table III we achieve abstract the roles using the following queries:

WSRT.service==MEI.receiver AND WSRT.role=="Investee"
WSRT.service==MEI.sender AND WSRT.role=="Investor" (2)

Using More Abstract Content Linkage

Our previous work uses exact XPaths of linkage parameters through the message. Here we generalize this by using regular expression matching through the entire message content. The following example searches for a promoter reference through an invest message without binding to a specific path of a specific schema.

regexmatch(".*"+payMEI.receiver +".*", investMEI.content) (3)

Hereafter we show how to mine business misuse instances of given patterns over live MEIs of all observed web transactions using streamSQL [8] and a StreamBase platform [9]. 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 live input streams from a file or a database and produce outputs. StreamSQL has several commands, of which we describe a few that we 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. Query 1 is in streamSQL and that can detect an abstract misuse pattern, say, of what happens in real time. For example, below, we can define a real life Ponzi-like pattern for "B robs A to pay C" using invest and pay message types:

invest; pay where
pay.reciever=C and invest.sender=A and
invest.reciever=B and pay.sender=B and
invest.recruiter=pay.reciever (4)

invest; pay notation in (4) denotes that invest message type is required to observe before pay message type where the rest of the criteria in (4) should also meet. Query 1 looks up invest and pay types using actual message names over MTT as in (1) as well as investor and investee roles using actual sender and receiver service names over WSRT as in (2) for abstraction. In order to find recruiter link between invest and pay message, the query uses abstract content linkage method as in (3).

DetectRecruits
Description: Glides over MEIs using window size 3 to detect pattern (4) along with the predicates specified in WHERE clause.
Input:MEI tuples
Output: Emits 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;
```

Query 1. Detecting recruits of Ponzi schemes.

Query 1 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.

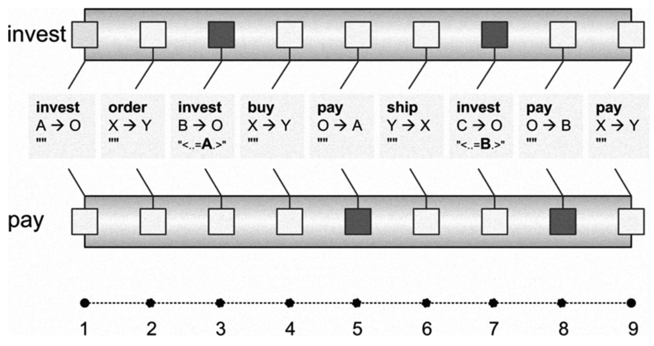

Fig. 8. Detecting Ponzi-like patterns. Notice two rectangles illustrating snapshots of detected invest;pay patterns.

Figure 8 shows how two Ponzi-like recruits are detected using the window of size 3 from a collection of 9 MEIs. It depicts 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.

Detecting few Ponzi-like recruits may provide little confidence in declaring that a Ponzi scheme was detected. In order to increase the confidence we can define some minimum support value as threshold thus alerting only when it meets. The query below can be added to strengthen the previous detecting query along with a predefined minimum support value. Query 2 alerts each time at least 6 Ponzi-like recruits are detected over the output of the previous query. Query 2 can be added to Query 1 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;
```

Query 2. Enhancing Ponzi detection.

B. *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 we briefly describe *malicious content* and *instantiation of flooding* techniques and propose detection queries for them.

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 our previous work [3]. Detecting such attacks needs less complicated queries than scanning the content using a comprehensive library of malicious scripts. Assuming we already have such a library, Query 3 detects messages that contain malicious scripts. Those script signatures might be derived from some prevention cheat sheets such as OWASP's (Open Web Application Security Project) [17].

TABLE IV
SIGNATURE TABLE (ST)

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

```
   DetectMalicousContent
   Description: Checks every message content if there is a
   malicious content.
   Input: MEI tuples
   Output: Emits 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 misue FROM SignatureTable AS SIG, MEI
8    WHERE regexmatch(".*"+SIG.signature+".*", MEI.content)
9    INTO AttacksOut;
```

Query 3. Detecting malicious content.

Query 3 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.

Instantiation of Flooding

Another service level misuse is denial of service. Below the SOAP layer the problem is the same with typical DoS over HTTP services. However, for web services, the transport layer may vary, thus a SOAP layer solution would be helpful. Assuming that TTP processes run over hardware with high computation power, we 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 5, 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 V
WEB SERVICE THRESHOLD TABLE (WSST)

ID	Service	Threshold
1	A	4
2	B	19
3	C	99
4	D	23

```
    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: Emits alert containing the count and time interval
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;
```

Query 4. Detecting instantiation flooding

Query 4 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 are readers and database clients to pass tuples into the input adapter, thus allowing them to be used in matching thresholds. Because actual frequency of the messages determines whether a set of messages is declared malicious, Query 4 uses a time window rather than timestamp values that are inserted 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.

VI. THE ALERT MODEL

Having generated enough evidences regarding a misuse, our framework sends targeted alerts. In order to introduce a stable and robust alerting framework we have to address two issues: First, we need to define the domain of web services that can benefit from such alerts. Second, we need to tune the queries to produce only one alert for each detected misuse and minimize false positives. Our answer to the first question is that alert type can be used to determine the web services that can benefit from the alert. As an answer to the second question, we describe a method to scale down the number of alerts.

A. Scaling Down the Alert Domain

We describe three ways to narrow the alert recipient domain. First, depending upon misuses we consider alerting only dependent web services – that is those that depend on the services of a potentially misbehaving service. In order to determine that, given a message we create a *dependency tree* starting from that message as the root. Second, for some misuses we may need to know the *types of web services* that have the potential to become a victim. As described earlier, Ponzi schemes target investor web services (i.e. that have the type of investor). The third is that some actors that may invoke other Use Cases also need to be alerted. Now we provide the details.

```
    GenerateDependencyTree
    Description: Tracing forward the MEIs, outputs an appropriate
    table to create tree-view of the downstream dependents.
    Input: MEI tuples
    Output:
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;
```

Query 5. Generating dependency tree (forward)

Dependency Tree Generation

Learning dependencies over past message interactions help in detecting potentially affected services. First, one can trace back to learn possible web service invocations causing the suspicious activity; and second, tracing forward could allow one to learn downstream invocations possibly caused by the misbehaving service. However, the observed messages in these two directions 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 work to exactly correlate message sequences in choreographies [10]-[11]. Basu, Casati, and Daniel [12] proposes a probabilistic model that attempts to predict message correlations with various probabilities. However, we do not need exact causal correlations to obtain the downstream dependents of a message. Instead, we propose Query 5 that learns downstream dependencies of a web service. The query expects to start from a record in the past and traverses the records forward generating a table of dependents.

Query 5 accepts MEI records and a root web service as shown in line 2. The query loads (lines 5-6) the root service into the **DependencyTable** which is created in memory and is appended each time a new dependent service is 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.

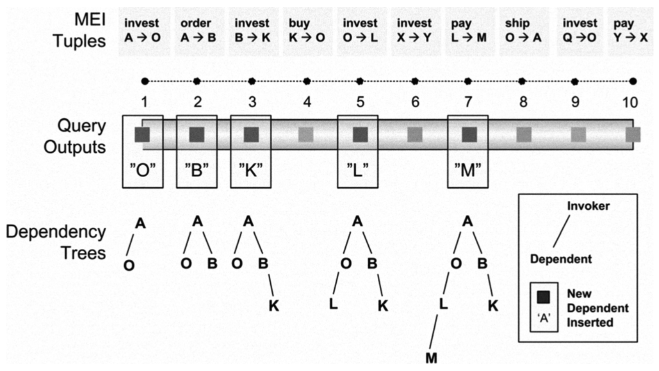

Fig. 9. Generating dependency tree in action. Notice Query Outputs and corresponding Dependency Trees as query traverses forward in time over MEI records.

Figure 9 illustrates the actions that Query 5 takes over a set of MEI tuples listed at the top. Tuples at times 1,2,3,5, and 7 meet the criteria of the query, and. it emits outputs including invoker and dependent fields. These two fields lead to building dependencies in tree form.

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 detected Ponzi-like misuses.

Potential Members

When a suspicious activity is observed with respect to a choreography model, all possible web services that can participate in the choreography are candidates to be alerted. This might be achieved in two ways: First, early association with choreographies (use patterns) and second extracting choreographies along with all possible branches of choice points and branches of such choreographies.. The first can be performed during the first registration of use pattern. The latter requires that the syntax of the choreographies be examined in detail, with all logical splits, choice points, loops, etc.

VII. Related Work

Luckham [13] proposes Rapide, an event pattern language that defines complex patterns and has been implemented in some service oriented architectures. To the best of our knowledge, none of them provide non-repudiable messages. Luckham [13] also provide 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, Ammon, Schaeffer, and Wolff [14] proposed a new approach based on the discriminant analysis of events, grouping them if they represent an unknown pattern. They expect their approach to recognize new patterns of credit card transaction use case scenarios and the fraudulent activities related to them. Their approach depended on having an accurate account of events. They implement an experimental environment based on a CEP engine.

Sense & response service architecture (SARESA) of [15] provides real time business intelligence (BI). SARESA introduces a comprehensive process to detect, interpret, automate, and respond to business partners. Unlike collecting externally observed events, it proposes using an event-driven architecture that collects the events from member services. Therefore, it is incapable of preventing misuses. Conversely, giving the partners the freedom to send events records to the system would allow the partners to control the misuses that can detected. However, SARESA has the advantage of serving diverse IT architectures and is not bound to only web services.

Ari [16] describes a data mining model management system that addresses model outdates, scalability of management structures, semantic differences between models, and business process integration for real time BI over SOA. Although the work reported in [16] does not propose or address specific real time architectures, it helps those systems to be multi dimensional in time, syntax and semantics.

VIII. Conclusions

In this paper we have shown a method to detect and potentially prevent the harm caused by service choreographies. We do by abstractly specifying service choreographies. Then we used several criteria to determine potential benefactors of detected misuses. And reduce the number of alert messages that can be sent to chosen web services. Finally we proposed potential software architectures that may be constructed to host our detection mechanisms.

IX. References

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

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

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

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

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

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

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

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

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

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8346
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8346