A Declarative Approach to Hardening Services Against QoS Vulnerabilities

Young-Woo Kwon and Eli Tilevich
Dept. of Computer Science
Virginia Tech
Blacksburg, VA 24060
Email: {ywkwon,tilevich}@cs.vt.edu

***Abstract*—The Quality of Service (QoS) in a distributed service-oriented application can be negatively affected by a variety of factors. Network volatility, hostile exploits, poor service management, all can prevent a service-oriented application from delivering its functionality to the user. This paper puts forward a novel approach to improving the reliability, security, and availability of service-oriented applications. To counter service vulnerabilities, a special service detects vulnerabilities as they emerge at runtime, and then hardens the applications by dynamically deploying special components. The novelty of our approach lies in using a *declarative framework* to express both vulnerabilities and hardening strategies in a domain-specific language, independent of the service infrastructure in place. Thus, our approach will make it possible to harden service-oriented applications in a disciplined and systematic fashion.**

I. Introduction

The mainstream software paradigm has been transitioning from software-as-a-product (SaaP) to software-as-a-service (SaaS). SaaS is a computing modality that comprises a collection of services—encapsulated units of computing functionality accessed by clients through public interfaces. Distributed service-oriented applications—accessed remotely across the network—are rapidly becoming the preferred building blocks for the majority of modern computing domains. The popularity of distributed service-oriented applications stems from the general software engineering benefits of SaaS, including low coupling, strong encapsulation, ease of discovery, and reduced maintenance costs.

Despite their numerous benefits, distributed services may not provide the requisite levels of reliability and security, particularly when operated in volatile network environments, attacked by hackers, or not properly maintained and managed. To that end, this paper describes a new approach that can harden distributed service-oriented applications against three major threats to the QoS: (1) network volatility—services are often accessed through disconnected and limited networks, for which the service must be properly adapted; (2) security exploits—a distributed service-oriented application can be exploited by an adversary for nefarious purposes, and must be protected against all known and future exploits; (3) administrative mismanagement—a distributed service-oriented application and its clients may be upgraded according to conflicting schedules, thus requiring runtime adaptation to avoid service protocol mismatches.

To harden distributed service-oriented applications against the important vulnerability classes described above, we propose *declarative hardening*. Specifically, we design and implement a DSL for expressing *hardening policies*. DSL combine high expressiveness, conciseness, and simplicity by providing constructs that are custom tailored for a given domain. In our case, the target domain is hardening distributed service-oriented applications. A service compiler translates the policies to a hardening components for a target service infrastructure in place. Finally, a hardening framework should be able to integrate the generated hardening components with a distributed service-oriented application, thereby equipping it with the capacity to counteract the specified vulnerabilities. Thus, this approach harmoniously combines several state-of-the-art technologies to address an important set of vulnerabilities that plague distributed service-oriented applications.

The uniqueness of the proposed approach lies in the following advantages over the current state of the art: (1) a declarative approach—we introduce a domain-specific language for describing both service vulnerabilities and the hardening strategies to eliminate them; (2) a compilation of declarative hardening specifications—our compiler is capable of generating working code for Open Service Gateway Initiative (OSGi) service infrastructure; (3) reusable hardening components—strategies are reusable across multiple applications and domains; (4) separation of concerns—reliability/security specialists can focus on their respective areas of expertise.

As our experimental platform, we use a well-known distribution middleware system—CXF-DOSGi—which enable service-oriented computing in Java. We have created a hardening framework which can harden any remote OSGi application, enabling it to cope with network volatility, security exploits, and mismanaged API. We provide a new programming language for expressing hardening policies and strategies, which can also be reused across applications. The programmer describes hardening policies and strategies. Then, the hardening framework handles all the underlying machinery required to harden the remote OSGi application.

In our experiments, we have executed a realistic OSGi application to measure efficiency and performance. By comparing the execution of the original and hardened version, we have assessed their respective ability to complete the execution, the total time taken to arrive to a result, and the overhead of the

hardening functionality. Our results indicate that it is feasible and useful to systematically harden existing service oriented applications with the ability to cope with vulnerabilities.

The rest of this paper is structured as follows. Section II introduces the concepts and technologies used in this work. Section III describes our proposed approach, including the proposed hardening language and hardening framework. Section IV evaluates the utility and efficiency of the proposed approach through a performance benchmark and a case study. Section V compares our approach to the existing state of the art. Finally, Section VI presents future research directions and concluding remarks.

II. Background

In the following discussion, we first describe service oriented architectures and their distributed versions. Then, we introduce the three types of vulnerabilities that can affect the QoS of distributed service-oriented applications and are addressed by our approach.

A. Service Oriented Architecture

Service-Oriented Architecture (SOA) has been recently employed as a means of providing uniform access to a variety of computing resources across multiple application domains. In SOA, software components are provided as services, self-encapsulated units of functionality accessed through a public interface. The core principles of SOA can be summarized as follows [9]:

- **Loose Coupling:** Services minimize dependencies and are aware only of each other.
- **Abstraction:** Services abstract away their underlying implementation details from their clients.
- **Reusability:** Services provide reusable functionality.
- **Autonomy:** Services control their environment and resources to provide consistency and reliability during the execution.
- **Statelessness:** Services avoid maintaining any state to facilitate failure recovery and minimize resource consumption.
- **Discoverability:** Services can be effectively discovered and interpreted through standard protocols.
- **Composability:** Services compose effectively regardless of their size and complexity.

This work uses the following service technologies.

1) OSGi: The Open Service Gateway Initiative (OSGi) provides a platform for implementing services [21]. It allows any Java class to be used as a service by publishing it as a service bundle. OSGi manages published bundles, allowing them to use each other's services. OSGi manages the lifecycle of a bundle (i.e., moving between install, start, stop, update, and delete stages) and allows it to be added and removed at runtime.

OSGi is a mature software component platform. It has been widely adopted by multiple industry and research stakeholders, organized into the OSGi Alliance. OSGi is used in large commercial projects, including the Spring framework[1] and Eclipse[2], which use this platform to update and manage plug-ins. The OSGi standard is currently implemented by several open-source projects, including Apache Felix[3], and Knopflerfish[4]. Despite its versatility, OSGi was mainly used for inter-bundle communication within a single host.

2) OSGi Remote Services: Recently, the OSGi alliance released the OSGi R4.2 specification that describes how remote OSGi services can be discovered and used [21]. The OSGi R4.2 specification does not specify how remote OSGi services should be accessed. Instead, the specification codifies only how remote service interfaces should be discovered and retrieved. Once a remote service interface is obtained, it is up to the implementor of this specification how interface methods are to be invoked at a remote OSGi framework and how their results are to be transferred back to the caller. The first reference implementation of R4.2 is Apache CXF-DOSGi[5], which implements the specification as Web services, using SOAP over HTTP for transmission and WSDL contracts for exposing services. In addition, RBI-OSGi [17] is the first non-RPC implementation of the OSGi R4.2 specification. RBI-OSGi does not require any changes to remote service interfaces, which are discovered and bound using a standard OSGi registry. Furthermore, R-OSGi [24] that was introduced prior to the standard OSGi remote services enables proxy-based distribution for services, providing proxies as OSGi bundles.

B. QoS Vulnerabilities

A distributed service-oriented application becomes vulnerable to several threats. Specifically, the network connecting remote services may be subject to volatility—temporary network outages. Rendering a service remotely accessible can make it vulnerable to security exploits—although security is a vast research area, here we focus only on the security issues pertaining to distribution. A recent study has determined that out of the known 39 OSGi vulnerabilities, as many as 20 vulnerabilities (e.g., exposing internal representation, flaws in parameter validation, and invalid work flow) [22] are specific to accessing services remotely. Finally, when services are remote to each other, they can evolve independently, thus causing version inconsistency problems.

1) Network Volatility: A remote service can be accessed through various networks, which are subject to network volatility due to various conditions such as random channel errors, node mobility, and congestion. For example, WiFi networks transmit radio signals, which are volatile, often making it impossible to reach a 100% reliability. Another condition causing network volatility is congestion, which occurs when radio channels interfere with each other or multiple data is transmitted concurrently over the same radio link.

[1] http://www.springsource.org/
[2] http://www.eclipse.org/
[3] http://felix.apache.org/
[4] http://www.knopflerfish.org/
[5] http://cxf.apache.org/distributed-osgi.html

When the underlying network fails, a distributed service-oriented application will typically signal an error to the end user, who can then decide on how to proceed. The user, for example, could choose to check the network connection and restart the application. The purpose of hardening strategies is to enable a distributed service-oriented application to continue executing when the underlying network becomes unavailable. A recent survey [18] classifies disconnected operation techniques as well as how they can be applied to improve the overall system dependability. Specifically, the most common disconnected operations are: *caching*—that employs caching techniques to store a subset of remote data locally, so that it could be retrieved and used by remote service requests when the network becomes unavailable; *hoarding*—that prefetches all the remote data needed for successfully completing any remote service invocation; *queuing*—that intercepts and records remote requests made to an unreachable remote service, and the recorded requests are then replayed when the service becomes available; and *replication*—that maintains a local copy of a remote component, so that when the remote component becomes unreachable, the local copy is used.

2) Security Exploits: The openness engendered by SaaS is a double-edged sword. On the one hand, any client can access a distributed service-oriented application through its public interfaces. On the other hand, unless a proper authentication scheme is put in place, the distributed service-oriented application can become exposed to malicious clients. According to the literature [22], distributed service-oriented applications are particularly vulnerable to the following threats:

a) Software Defects: can cause faults and failures in distributed service-oriented applications. For example, a logic bug may allow clients access certain methods without authentication (i.e., bypassing the invocation of `authenticate()`). Such bugs expose systems to security exploits that can render services unavailable. Eventually, the bugs should be fixed by modifying the source code, but a hardening strategy can also be developed to handle such software defects. For example, recently proposed approaches accomplish that through runtime verification which monitor systems and synthesize corrective functionality [6].

b) Improper Parameter Validation: exposes service methods to malicious clients that can pass illegal parameters, thus leading to undesirable outcomes. For example, accessing a parameter exceeding the available memory can render the service unavailable.

c) Invalid Access: has two types of vulnerability patterns—exposing internal representation and accessing from malicious clients. Exposed internal representation enables to execute code that should be hidden, thereby triggering unexpected system behavior or enabling malicious clients to access sensitive data. In addition, remote services should be protected from malicious clients. To deal with such invalid accesses, authorization and access control are commonly used techniques.

3) Service Mismanagement: Service-oriented applications rely on loosely-coupled remote interfaces, each of which could evolve independently. When the vendor releases a new version of the application, the users can update different services at different times. As a result, the device can try to communicate with the old service interface. If the service interface has changed, the requested service methods may no longer be available.

C. Domain-Specific Language and Security Policies

A domain-specific language (DSL) is a programming language designed to solve problems in a particular domain. Compared to general-purpose languages (e.g., C, C++, Java, etc.) DSLs are custom tailored for the domain at hand, providing expressivenesses and ease of use advantages. DSL encapsulates its domain expertise, making it easier for non-expert programmers to craft effective solutions for problems in the target domain.

In security research, domain-specific policy languages have been proposed to describe authorization and access control [8], [13]. These languages address the poor fit of general purpose languages to describe all the numerous low-level security issues that occur at the systems level. Equipped with such a DSL, programmers can easily express sophisticated security configurations.

III. Declarative Hardening

Next, we first outline our proposed solution and then describe our hardening language and hardening framework, respectively.

A. Solution Overview

To harden distributed service-oriented applications against the important vulnerability classes described above, we propose *declarative hardening*. Figure 1 depicts how our approach leverages the expressive power of DSLs, flexibility of the service compilation, and the adaptivity of the hardening framework. Specifically, we design and implement a DSL for expressing *hardening policies*. DSL combine high expressiveness, conciseness, and simplicity by providing constructs that are custom tailored for a given domain. In our case, the target domain is hardening distributed service-oriented applications. A service compiler translates the policies to a hardening components for a target service infrastructure in place. In our case, the target service infrastructure is the OSGi framework. Finally, our hardening framework seamlessly integrate the generated hardening components with a distributed service-oriented application, thereby equipping it with the capacity to counteract the specified vulnerabilities. Thus, our approach harmoniously combines several state-of-the-art technologies to elegantly address an important set of vulnerabilities that plague distributed service-oriented applications.

B. Hardening Policy Language—HPL

One of the key novelties of our approach is using a DSL for describing vulnerabilities and their hardening strategies. We call our language *Hardening Policy Language (HPL)*. In designing HPL, we aim at combining both expressiveness and ease of use. The specific design goals include:

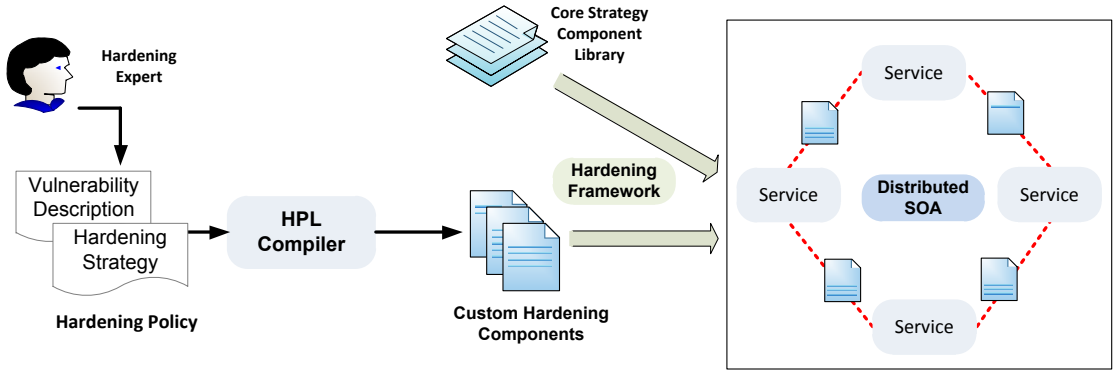

Fig. 1. Approach Overview.

```
Policy (ServiceConfig | NetworkHardening | SecurityHardening
        | ServiceAPIHardening | HardeningStrategy)
Begin
    PolicyName => [name] ;
    ServiceName => [name] ;
    Config => ([config_types] is [type])+ ;

    Condition => //Network Volatility
      (NetworkEvent([event]))
        When (Execution | Call) [From | To] [flow])+ ;
      Config => ([config_types] is [type])+ ;
      Then => ([strategy]) ;

    Condition => //Security Exploits
      ((Execution | NotExecution | Call | NotCall)
        [flow] [From | To] [url])+ ;
      ((ParamChecking [flow] [From | To] [url]
        Using Strategy [strategy])+ ;
      (Access From [url])+ ;
      Then => ([strategy]) ;

    Condition => //Mismanaged Service Interface
      (Exception([exception])
        When (Execution | Call) ([flow]))+ ;
      ((Execution | Call) [flow] [From|To] [url])+ ;
      Then => ([strategy]) ;
End
```

Fig. 2. Language constructs.

- *Expressiveness*—a reliability/security expert should be able to express any kind of vulnerability easily, with the resulting code being easy to understand, maintain, and evolve.
- *Extensibility*—it should be possible to integrate existing security and reliability policies with HPL policies.
- *Platform Independence*—HPL policies should be platform independent, with the same policy compilable to any service platform.

Figure 2 shows how the HPL is constructed. To provide fault-tolerance and security defense to distributed service-oriented applications, what the programmer should do is only to write a policy script in HPL. First of all, hardening policies consist of five types of policy, including Service Configuration, Network Hardening, Security Hardening, Service API Hardening, and Hardening Strategy. Each policy mainly consists of a set of conditions which describes specific vulnerable situations and applicable hardening strategies. In the following sections, we detail how our HPL can effectively express remote services, vulnerabilities and hardening strategies.

An HPL policy can then be compiled to a specific service platform. For example, if the platform is Java-based, our HPL compiler generates hardening Java components that implement interface HardeningEventListener:

```
public interface HardeningEventListener {
  public Object eventNotified(HardeningEvent event);
}
```

The interface is implemented by our core strategy component library (See Figure 1), which supplies OSGi-specific hardening components. Reliability/security experts also is able to extend the library with new hardening components that handle newly discovered vulnerabilities. The method eventNotified takes HardeningEvent which contains invocation information, including a service object, method information, URL, vulnerability type, exception, etc.

1) Hardening Services with HPL:

Service Configurations: Figure 3 shows an HPL policy that can configure different operational environments. In particular, the programmer can specify device types (e.g., mobile, server, etc), network types (e.g., WiFi, 3G, LAN, etc), network conditions (e.g., delay, loss, and jitter), service types (e.g., conversational, streaming, interactive, background), and a required QoS-level (e.g., best-effort, guaranteed, etc). Through configuration settings, the programmer can detail characteristics of the remote service, thereby making it possible to provide different hardening scenarios according to dynamically changing environment.

Network Volatility: Figure 4 presents an HPL hardening policy that can make a service resilient network volatility. The policy is identified by its name and the `NetworkHardening` type. The same policy can be applied to multiple services by using different service identifiers. The policy contains vulnerability conditions and a hardening strategy description. The `NetworkEvent` keyword describes system network events such as disconnection, reconnection, packets loss, and normal operation. The optional `When` keyword monitors all the exceptions or events related to a specific method. The

```
Policy ServiceConfig
Begin
  PolicyName => [policy_name] ;
  ServiceName => [service_name] ;
  Config =>
    DeviceType is [mobile | server | ... ]
    NetworkType is [WiFi | 3G | LAN | ... ]
    NetworkCondition.{delay,loss, jitter }
      is {([high|med|low])+}
    ServiceType is
      [ conversational |streaming| interactive |background]
    QoS is [best−effort | guaranteed | ... ] ;
End
```

Fig. 3. A script describing service configurations.

programmer specifies the conditions using the `Execution` and `Call` keywords. These keywords specify the execution locations to be monitored. The HPL compiler generates aspects to intercept application-level exceptions and system events, raised in response to experiencing volatility.

An HPL policy can be configured for different operational environments by specifying the distributed service-oriented application's device types, network links (i.e., bandwidth/latency), and required QoS.

A repository of readily-available hardening components for network volatility will be reusable out-of-the-box and will also serve as building blocks for custom strategies. For network volatility, the hardening components will be based on widely used disconnected operations.

```
Policy NetworkHardening
Begin
  PolicyName => [policy_name];
  ServiceName => [service_name];
  Condition => NetworkEvent([event])
    When (Execution | Call) [From | To] [flow])+ ;
    Config =>
      DeviceType is [mobile | server | ... ]
      NetworkType is [WiFi | 3G | LAN | ... ]
      NetworkCondition.{delay, loss , jitter } is
        {([high | med | low])+}
      ServiceType is
        [ conversational | streaming | interactive | background]
      QoS is [best−effort | guaranteed | ... ] ;
    Then => Apply Strategy([strategy_name]);
End
```

Fig. 4. Hardening a service against network volatility.

Security Exploits: Figure 5 depicts an HPL policy for hardening a distributed service-oriented application against security exploits. In particular, we aim at *application level* security exploits of distributed service-oriented applications. Defenses against low-level attacks, such as sniffing, spoofing, etc., have been thoroughly integrated with modern network stacks. To detect software defects, we adopt the notion of a legitimate program control flow—allowable sequences of service method calls—expressed through the `Execution`, `NotExecution`, `Call`, `NotCall` keywords. These keywords parameterize our HPL compiler to generate runtime monitors that can detect and counteract exploits.

To defend a distributed service-oriented application against malicious clients, HPL features the `Access`, `Call`, and `Execution` keywords. By controlling the control flow of a service, our approach prevents malicious clients from exploiting the openness espoused by SaaS architectures. After a service's public interface is published, traditional service platforms exercise little control over how clients use this interface. Our approach adds auditing capabilities to the execution of a service by enforcing its control flow and access control.

To guard the execution of a service against improper service method parameters, HPL features the `ParamChecking` keyword that can be used to generate parameter inspection components. Parameters can be verified to hold certain values or not to surpass certain allocated memory thresholds.

In terms of the specific hardening strategies, suspicious clients can be handled by expressing in HPL a custom written component that will be invoked to counteract the detected exploits. For example, the client's connection can be terminated, a security enhancer strategy can be installed to prevent future exploits, or a service can be registered to be resuscitated if the detected penetration does end up bringing it down.

Security enhancers strategies encapsulate well-known security mechanisms such as security protocols, cryptography, authentication, and authorization schemes. Because these schemes incur a performance cost, one could choose to activate them only if necessary. For example, if unauthorized use of a service is detected, the client's connection will be terminated and an access control strategy can be deployed to control which clients can use the service in the future.

Finally, service resuscitators attempt to return a service to a clean state before or after encountering a fault [28]. Among the strategies that can be useful are *micro-restart* [5] and *checkpoint-restart* [15]. Upon detecting a potentially illegal parameter in the example above, a restart strategy can be installed to restart the service if the illegal parameter does bring down the service.

```
Policy SecurityHardening
Begin
  PolicyName => [policy_name];
  ServiceName => [service_name];
  Condition =>
    ((Execution | NotExecution ) ([flow]) From [url])+;
    Then => Apply Strategy([strategy_name]);
  Condition =>
    ((Call | NotCall) ([flow]) To [url ])+;
    Then => Apply Strategy([strategy_name]);
  Condition =>
    ParamChecking([flow]) Using Strategy([strategy_name]);
    Then => Apply Strategy([strategy_name]);
  Condition =>
    Access (From | To) [url]+ ;
    Then => Apply Strategy([strategy_name]);
End
```

Fig. 5. Hardening a service against security vulnerabilities.

Service Mismanagement: Figure 6 shows an HPL policy to harden a distributed service-oriented application against being mismanaged during upgrades. The `Exception` keyword

adds monitoring capabilities to invoking a service through an obsolete public interface. In response to detecting such version mismatch, a hardening strategy can automatically generate a service adapter, initiate a dynamic upgrade, or schedule an upgrade at a later point. The `Execution` or `Call` keywords provide fine-grained capabilities in monitoring for service mismanagement (e.g., at the method or client location levels).

The problem of mismanaged service interface in distributed service-oriented applications is well-known [4]. Our approach explores how this vulnerability can be handled systematically. Handling this problem is closely related to managing API evolution, a highly-active area of recent research. Recent approaches include explicit documentation, automatic inference and refactorings, compatibility layers, etc. These approaches provides valuable insights for the design of hardening strategies to handle mismanaged service interfaces.

```
Policy ServiceAPIHardening
Begin
  PolicyName => [policy_name];
  ServiceName => [service_name];
  Condition =>
    Exception([exception]
      When (Execution|Call) ([flow] (From|To) [url ]))+;
    Then => Apply Strategy([strategy_name);
  Condition =>
    ((Execution | Call) [flow] (From | To) [url ])+ ;
    Then => Apply Strategy([strategy_name);
End
```

Fig. 6. Hardening a service against mismanaged service interfaces.

2) Hardening Strategy: Fig 7 shows an HPL script that expresses a hardening strategy. To that end, HPL features several keywords that define basic execution directives—`Execute`, `Reject`, `Throw`, `Stop`, `Delegate`, `Replace`, etc. The directives constitute atomic operational units and are expected to be provided as part of the core component library. Using the directives, the programmer can implement service-specific strategies or extend the existing hardening strategies. A strategy script starts with the `HardeningStrategy` keyword, followed by a policy name and service identifier. Then, the `Implements` block describes strategy implementations that consist of the predefined execution directives and custom components that have to be custom implemented for the service platform in place. Our HPL compiler translates HPL scripts to components and distributed aspects that is integrated with distributed service-oriented applications.

C. Dynamically Composable Hardening Framework

In the following section, we discuss the system architecture of the hardening framework. The key objective of this work is to explore how policies can be interpreted and instantiated in the hardening framework and applied to an existing distributed service-oriented application that may have been written without fault-tolerance capabilities in mind.

The purpose of the hardening framework is to harden a distributed service-oriented application with resiliency to cope with vulnerabilities.

```
Policy HardeningStrategy
Begin
  PolicyName => [policy_name];
  ServiceName => [service_name];

  Implements {
    Method public Object eventNotified(HardeningEvent event)
    {
      @Execute(event);
      @Reject(event);
      @Throw(Exception());
       ... //Implement custom hardening strategies
    } ;
  }
End
```

Fig. 7. Describing a hardening strategy.

In designing the hardening framework, we pursue the following goals:

1) *Transparency*—any hardening strategy and the hardening framework should not affect the core functionality of the underlying OSGi framework and applications.
2) *Flexibility*—the hardening framework should be capable of adding or removing policies at any time without having to stop the application.
3) *Efficiency*—the hardening framework should not affect significantly the performance of the OSGi application .

Next, we discuss the system architecture of the hardening framework. Modern state-of-the-art middleware infrastructures reports various low-level symptoms of something going wrong in the execution of remote services (e.g., link failure, node mobility, non-existing service methods, etc.) by means of application-level exceptions. The hardening framework intercepts such application-level exceptions as well as the events signaling some changes in low-level service execution (e.g., a successful network reconnection). Then, the hardening framework handles application-level exceptions by triggering a hardening strategy.

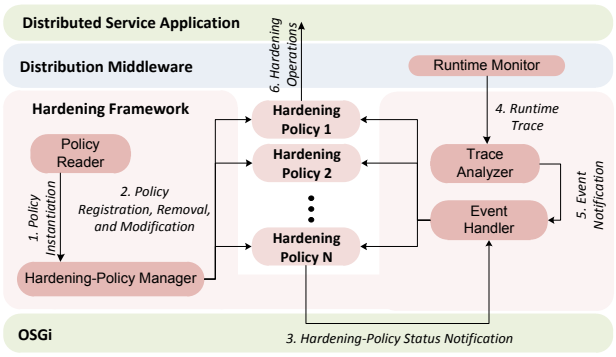

Fig. 8. The hardening framework.

Figure 8 shows how the hardening framework was integrated with the OSGi framework and existing services. The hardening framework periodically reads hardening policies

from the specified policy repository. Then, our HPL compiler translates hardening policies and strategy descriptions to XML documents and runtime binaries (e.g., Java bytecode). The hardening policy manager instantiates vulnerability conditions, so that the hardening framework can detect vulnerabilities by comparing vulnerability conditions with runtime traces. A hardening strategy is a standard OSGi service that implements HardeningEventListener interface. Then, the dynamically generated or pre-deployed hardening strategies are registered to the OSGi framework, and the hardening policy manager keeps track of their statuses (e.g., registration, unregistration, update, etc) for dynamic loading and unloading.

In addition, according to the standard OSGi specification, ServiceHook enables other services to intercept OSGi framework events. Thus, when a distributed service-oriented application starts its remote service, the hardening framework creates a runtime monitor which intercepts remote service invocations and catches exceptions and events. Such traces are analyzed by the trace analyzer and forwarded to the registered hardening strategies to counteract the found vulnerabilities.

D. Discussion

The approach described has specific engineering objectives, creating pragmatic new technologies that can make distributed service-oriented applications more available, reliable, and secure. One important question concerns whether availability, reliability, and security can be effectively reasoned about and implemented as orthogonal cross-cutting concerns, separate from the core functionality of a given distributed service-oriented application. The scientific consensus has been that it is impossible to achieve this objective in full generality. However, these concerns can be quite effectively separated in certain domains and execution environments.

Being specifically tailored to address the problems of a given domain, DSLs can be powerful and effective tools. However, learning a new DSL takes an additional effort that may negatively affect programmer productivity. Although we designed HPL to be easy to learn and use, programmers tend to differ in their ability to learn new languages. As a result, introducing HPL in the programmer's tool chain may initially inconvenience some programmers.

Finally, to yield its intended benefits, our approach relies on the existence of state-of-the art adaptation facilities of the underlying middelware infrastructure. Although OSGi has all the facilities required to support our approach, other middleware platforms may lack some advanced features such as deploying and undeploying services at runtime. In future work, we plan to explore how generalizable our approach is.

IV. Evaluation

We evaluated the effectiveness and performance of our hardening framework through a micro benchmark and a larger case study.

A. Micro Benchmark

For this experiment, we have used Lucene, a widely-used Java search engine library distributed as an OSGi bundle. Among the capabilities provided by Lucene are indexing files and retrieving indexes of a given search word. We used Lucene to implement a dictionary service that given a word can return its definition, synonyms and neighboring words.

All the experiments were conducted on the client machine running 3.0 GHz Intel Dual-Core CPU, 2 GB RAM, Windows XP, JVM 1.6.0 13 (build 1.6.0 13-b03), and the server machine running 1.8 GHz Intel Dual-Core CPU, 2.5 GB RAM, Windows 7, JVM 1.6.0 16 (build 1.6.0 16-b01, connected via a local area network (LAN) with a 100Mbps bandwidth, and 1ms latency.

In this benchmark, we measured the performance overhead for CXF-DOSGi middleware platform. Specifically, we examined how the service can be effectively executed, in terms of the total execution time when our declarative service hardening module is introduced. Each benchmark method calls three services in sequence, repeating each service call 100 times, and then reporting the total execution time. The results show that as the number of policies grows, 30 hardening policies experience a performance overhead of about 10%, which shows that our approach is practical. If a service-oriented application can afford to run 10% slower, it can benefit from our approach.

B. Case Study: OneBusAway

OneBusAway[6] is a bus information system that enables passengers of the local transportation system to track the location and movement of commuter buses over the Internet and using mobile devices [10]. OneBusAway system provides several APIs for different devices, including REST APIs for web applications, iPhone APIs, and SMS APIs.

```
Policy ServiceConfig
Begin
  PolicyName => onebusaway_configs;
  ServiceName => OneBusAway;
  Config =>
    DeviceType is mobile &&
    NetworkCondition.{dealy, loss,  jitter }
      is {high, high, high} &&
    NetworkType is 3G &&
    ServiceType is  interactive  &&
    QoS is best−effort ;
End
```

Fig. 9. An HPL policy describing OneBusAway configurations.

1) Describing OneBusAway Configurations: Figure 9 shows how a OneBusAway client service can be configured. The service configurations are used for both the server and clients to determine an appropriate hardening strategy. In this case study, since the OneBusAway service aims at providing bus schedule in real time at any location, we assume that a client is a mobile device using a 3G network. Thus, network conditions such as delay, loss, and jitters are relatively high. The service type is `interactive` and the required QoS level is `best-effort`. Of course, the service configuration can be differently set according to changes of network conditions or types of a client device.

[6]http://www.onebusaway.org/

```
Policy NetworkHardening
Begin
  PolicyName => onebusaway_net_hardening;
  ServiceName => OneBusAway;
  Condition =>
    NetworkEvent(Disconnection && Normal)
      When Execution(List<StopBean> StopsForLocation(*));
      Config =>
        QoS is best-effort &&
        DeviceType is mobile &&
        NetworkType is 3G &&
        ServiceType is interactive &&
        NetworkCondition.{delay, loss, jitter }
          is {high, high, high} ;
    Then => Apply Strategy(Caching);
End
```

Fig. 10. An HPL policy against network volatility.

2) Hardening OneBusAway Against Network Volatility: Figure 10 depicts an HPL policy to harden the OneBusAway service against network volatility. We harden the method List<StopBean> StopsForLocation(*), which immediately returns bus stops' information for given location. When network events are raised from a distribution middleware system, the `Caching` strategy will be applied. The caching strategy stores all remote method invocation requests and results when the network is operating normally. Then, the strategy retrieves results from the cache. Thus, `NetworkEvent` takes two types of events—`Disconnection` and `Normal`.

3) Hardening OneBusAway Against Security Vulnerabilities: Figure 11 shows an HPL policy to harden the OneBusAway service against security exploits. This policy script describes four types of security vulnerabilities. First, to hide the remote method CurrentTime(), the remote service rejects all requests. Typically, removing a method from public service interface requires changing the interface's source code. Our hardening policy, however, makes it possible to hide service methods as needed. This is accomplished by declining all the client calls to the removed methods.

Second vulnerability is passing improper parameters to the method List<StopBean> StopsForLocation(*). Since this method does not validate location data, it throws NullPointerException in case of that location data (i.e., longitude and latitude) are out of range. To inspect parameters, we use the `Checker` strategy.

The third vulnerability is a logic flow that can allow malicious clients to bypass authentication. For this experiment, we created a new method **void** authenticate(*) for checking clients' credentials. Thus, before calling any method in OneBusAway, clients should first set their user name that is subsequently used for authenticating all service method invocations from that client. If the method **void** authenticate(*) is not invoked, all requests are denied. The last vulnerability suspicious clients potentially misusing a service. To counter this vulnerability, the Access keyword monitors connected clients and can reject all requests from any specified URL.

```
Policy SecurityHardening
Begin
  PolicyName => onebusaway_sec_hardening;
  ServiceName => OneBusAway;
  Condition =>
    Execution(TimeBean CurrentTime()) ;
    Then => Apply Strategy(Reject);
  Condition =>
    ParamChecking(List<StopBean> StopsForLocation(*))
      Using Strategy(Checker);
    Then => Apply Strategy(Reject);
  Condition =>
    NotExecution(void authenticate (*)) && Execution(*);
    Then => Apply Strategy(Reject);
  Condition =>
    Access From [malicious_url];
    Then => Apply Strategy(Reject);
End
```

Fig. 11. An HPL policy against security exploits.

```
Policy ServiceAPIHardening
Begin
  PolicyName => onebusaway_API_hardening;
  ServiceName => OneBusAway;
  Condition =>
    Exception(NoSuchMethodException)
      When Execution(List<StopBean> StopsForLocation(*)) ;
    Then => Apply Strategy(Adapter);
End
```

Fig. 12. An HPL policy against the mismanaged service.

4) Hardening OneBusAway Against Mismanaged Service: Figure 12 shows an HPL policy to harden the OneBusAway service against mismanaged service. For this experiment, we added an integer argument for logging client's ID to the method List<StopBean> StopsForLocation(). Thus, when clients request List<StopBean> StopsForLocation() without specifying their ID, the remote service will throw NoSuchMethodException. The `Adapter` strategy supplies the missed parameter and then invokes the updated method.

5) Describing a Hardening Strategy: Figure 13 presents an HPL policy that creates a `Caching` strategy to be used for network volatility hardening. In this example, we simply store execution results in a HashTable. This caching strategy stores all results during normal operations and then retrieves their results when the network becomes unavailable.

V. Related Work

Although modern society intrinsically depends on software systems, all computing systems are prone to unreliability. Complex distributed systems often fail to deliver the expected quality of service (QoS), when their constituent components fail. This lack of reliability negatively affect the overall system's trustworthiness. Indeed, defects in deployed software systems cost the US economy billions of dollars annually [26]

Our approach is related to several research domains, which include automated fault tolerance, security hardening, adaptive and fault-tolerant middleware, and aspect oriented software construction. This work synthesizes and enhances some existing common hardening strategies. In the following discussion, we outline the main research domains from which this work

```
Policy HardeningStrategy
Begin
  PolicyName => onebusaway_caching_strategy;

  Implements {
    Method public Object eventNotified(HardeningEvent event) {
      if(@Caching == null) {
        @CreateStorage
          (@Caching, HashTable<HardeningEvent, Object>);
      }
      if(event.TYPE == NETWORK_NORMAL) {
        Object result = @Execute(event);
        @Store(@Caching, event, result );
      } else if(event.TYPE == NETWORK_DISCONNECTION) {
        Object result = @Retrieve(event);
        if( result != null) { return result ; }
        else { @Throw(Exception("Network Disconnection"));}
      }
    } ;
  }
End
```

Fig. 13. Describing a caching hardening strategy.

draws inspiration and borrows well-established and verified solutions.

a) DSL for Reliability and Security: Much research explored DSLs to solve reliability problems. In the field of security, policy-based approach has been widely explored in the last decade. Among recently introduced policy languages are including Ponder [8] and Rei [13]. Ponder defines authorization and security management policies. Because policies are separated from a system, it can adapt to changing requirements by disabling or replacing policies without restarting. Rei can be used to define different kinds of policies, including security, privacy, management, and conversation. These policy languages have inspired the design of HPL. However, HPL focuses on application-level security and also aims at availability and reliability.

GRAFT [29] automatically specializes middleware for fault-tolerance. It employs Component Availability Modeling Language (CAML) to annotate a distributed application's model, and then automatically specializes the application's middleware for domain-specific fault-tolerant requirements. GRAFT also uses a DSL to express the requested fault-tolerance functionality. Although similar to our approach in terms of adopting domain-specific approach, GRAFT only copes with reliability problems. On the other hand, our approach counteracts availability and security, as well as reliability.

Business Process Execution Language (BPEL) is a standard language that defines business processes for Web services. A BPEL program can, for example, express that a Web service be composed through a business process involving some existing Web services. To handle failures in BPEL processes, various monitoring techniques have been proposed [11], [2], [19], [3]. Our approach shares the same goal with these techniques, but we strive to achieve greater transparency in detecting anomalies and flexibility in deploying solution components. Unlike the prior state of the art, our approach does not require any modification to the underlying middleware infrastructure (e.g., Web service runtime or the BPEL execution engine). Our approach also deploys special-purpose components to counter the detected vulnerabilities. Furthermore, our approach is flexible and dynamic: special failure-handling components can be deployed at runtime without having to interrupt the execution.

Some of recent research has focused on providing failure handling mechanisms at runtime by using the Aspect-Oriented Programming (AOP) technique, which enables inserting failure handling modules into an unmodified BPEL. However, whenever weaving occurs at deployment time [2], new failure types cannot be handled dynamically. Although reference [19], [3] presents a runtime failure handling mechanism, it can only handle restricted failure types (e.g., service failure) because they were built on top of the existing BPEL specification. As compared to these approaches, our framework includes both a dynamically composable failure handling language and its execution runtime system. Our approach thus equips programmers with the ability to cope with various service QoS vulnerabilities by simply describing a new policy script and dynamically instantiating the required hardening strategies. Finally, most BPEL-based approaches have focused on handling failures at a service provider. However, our approach enables failure handling at both the server and client parts of an service-based application. Thus, whenever a service provider cannot be modified, the service can still be hardened by deploying our framework only at the service consumer side.

b) Fault-Tolerant Middleware: A number of techniques for making existing systems fault tolerant [12], [23], [16] are related to our approach. JReplica [12] expresses via AOP how adaptable fault tolerance can be added through replication. Reference [23] describes how fault tolerance can be added to CORBA components by automatically instantiating distributed replicated components. DR-OSGi [16] is a component framework to harden distributed service-oriented applications against network volatility. DR-OSGi avoids modifying source code explicitly and enables the reuse of disconnected operations across different applications. Arora and Kulkarni [1] have shown that fault-tolerant systems feature two types of components that they called *detectors* and *correctors*. They have argued that enhancing a fault-tolerant system with a set of fault-tolerant components will lead to a fault-tolerant system. They have also suggested that this division can serve as a basis for designing component-based fault tolerant systems.

Our approach based on above techniques enables the programmer to harden distributed service-oriented applications without having to modify their source code explicitly. By avoiding ad-hoc modification that can be tedious and error-prone, our approach not only hardens distributed service-oriented applications more systematically, but also enables greater reuse of the hardening strategies across different distributed service-oriented applications.

c) Security as a Separate Concern: Our approach treats security as a separate concern. A popular technology for modularizing cross-cutting concerns is AOP [14], which has been successfully used in prior systems for introducing

security-related functionality [30]. In addition, several special security libraries and frameworks are AOP-based, including Java Security Aspect Library(JSAL) [7], Security Annotation Framework [25], and Spring Security [27]. What makes AOP a promising technology for implementing our approach is its ability to weave in concerns at runtime, without restarting the application. This runtime adaption ability aligns well with the dynamic nature of the OSGi infrastructure. AOP is not the only approach for encapsulating security functionality. A middleware-based approach such as CORBA Security Service [20] has been shown successful for modularizing security functionality, including authentication, authorization, confidentiality, integrity, and auditing.

VI. Future Work and Conclusions

In future work, we plan to focus on increasing the generality and heterogeneity of our approach, so that the same HPL policy could be flexibly compiled into platform-specific component instantiations. Another direction will explore the complexities of applying multiple hardening strategies to the same service: the strategies should be able to coexist without interference.

In this paper, we have introduced *Declarative Hardening*, a promising approach for systematically hardening service applications to cope with network volatility, security exploits, and service mismanagement. Our HPL language is an expressive a powerful abstraction for the programmer to describe various hardening policies. The HPL compiler translates policy scripts to hardening components, which are applied to distributed service-oriented applications at runtime. The micro benchmark and case study showed effectiveness of our approach. As we rely on greater numbers of network-enabled devices with network volatility, security exploits, and service mismanagement remain a permanent presence. Declarative hardening explores how these vulnerabilities can be handled declaratively, providing a systematic and reusable solution.

Acknowledgments

The authors would like to thank the MESOCA anonymous reviewers, whose comments helped improve this paper's presentation. This research is supported by the National Science Foundation through the grant CCF-1116565.

References

[1] A. Arora and S. Kulkarni. Detectors and correctors: a theory of fault-tolerance components. In *The 1998 18 th International Conference on Distributed Computing Systems*, pages 436–443, 1998.

[2] L. Baresi and S. Guinea. Towards dynamic monitoring of ws-bpel processes. In B. Benatallah, F. Casati, and P. Traverso, editors, *Service-Oriented Computing - ICSOC 2005*, volume 3826 of *Lecture Notes in Computer Science*, pages 269–282, 2005.

[3] L. Baresi, S. Guinea, M. Pistore, and M. Trainotti. Dynamo + astro: An integrated approach for bpel monitoring. volume 0, pages 230–237, Los Alamitos, CA, 2009.

[4] K. Becker, A. Lopes, D. S. Milojicic, J. Pruyne, and S. Singhal. Automatically determining compatibility of evolving services. In *ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services*, pages 161–168, Washington D.C., 2008.

[5] G. Candea and A. Fox. Recursive restartability: turning the reboot sledgehammer into a scalpel. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 125–130, May 2001.

[6] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 569–588, New York, NY, USA, 2007. ACM.

[7] M. H. Chunlei, C. Wang, and L. Zhang. Toward a reusable and generic security aspect library. In *In AOSD:AOSDSEC 04: AOSD Technology for Application-level Security*, 2004.

[8] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. A language for specifying security and management policies for distributed systems. *Imperial College Research Report DoC*, 1, 2000.

[9] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[10] B. Ferris, K. Watkins, and A. Borning. Onebusaway: results from providing real-time arrival information for public transit. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 1807–1816, 2010.

[11] S. Guinea, L. Baresi, G. Spanoudakis, and O. Nano. Comprehensive monitoring of bpel processes. *IEEE Internet Computing*, 99, 2009.

[12] J. L. Herrero, F. Sanchez, O. Sanchez, and M. Toro. Fault tolerance AOP approach. In *Workshop on AOP and Separation of Concerns*, pages 44–52, 2001.

[13] L. Kagal, T. Finin, and A. Joshi. A policy language for a pervasive computing environment. *Policies for Distributed Systems and Networks, IEEE International Workshop on*, 0:63, 2003.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming(ECOOP 97)*, 1997.

[15] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[16] Y.-W. Kwon, E. Tilevich, and T. Apiwattanapong. DR-OSGi: Hardening distributed components with network volatility resiliency. In *Proceedings of the ACM/IFIP/USENIX* 10^{th} *International Middleware Conference (Middleware 2009)*, 2009.

[17] Y.-W. Kwon, E. Tilevich, and W. R. Cook. An assessment of middleware platforms for accessing remote services. In *Proceedings of the 2010 IEEE International Conference on Services Computing*, SCC '10, 2010.

[18] M. Mikic-Rakic and N. Medvidovic. A classification of disconnected operation techniques. In *Proceedings of the 32nd EUROMICRO Conference on Software engineering and Advanced Applications (EUROMICRO-SEAA'06)*, 2006.

[19] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 815–824, New York, NY, USA, 2008. ACM.

[20] Object Management Group. The CORBA security service specification. Specification, Object Management Group, 2002.

[21] OSGi Alliance. OSGi release 4.1 specification. Specification, 2010.

[22] P. Parrend and S. Frénot. Classification of component vulnerabilities in java service oriented programming (sop) platforms. In *CBSE '08: Proceedings of the 11th International Symposium on Component-Based Software Engineering*, pages 80–96, 2008.

[23] A. Polze, J. Schwarz, and M. Malek. Automatic generation of fault-tolerant CORBA-services. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 2000)*, 2000.

[24] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: Distributed applications through software modularization. In *Proceedings of the ACM/IFIP/USENIX* 8^{th} *International Middleware Conference*, Newport beach, CA, USA, November 2007.

[25] Security Annotation Framework. http://safr.sourceforge.net/.

[26] D. Scott. Assessing the costs of application downtime. Technical report, Gartner Group, 1998. www.gartner.com.

[27] Spring Security. http://static.springsource.org/spring-security/site/.

[28] M. Sullivan and R. Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, pages 2–9, Jun 1991.

[29] S. Tambe, A. Dabholkar, J. Balasubramanian, and A. Gokhale. Automating middleware specializations for fault tolerance. In *Proceedings of the International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, March 2009.

[30] J. Viega, J. T. Bloch, and P. Ch. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14:31–39, 2001.