# Architecture-Based Self-Protecting Software Systems

Eric Yuan, Sam Malek
George Mason University
4400 University Drive
Fairfax, VA, 22030
{eyuan, smalek}@gmu.edu

Bradley Schmerl, David Garlan, Jeff Gennari
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA, 15213
{schmerl, garlan, jgennari}@cs.cmu.edu

## ABSTRACT

Since conventional software security approaches are often manually developed and statically deployed, they are no longer sufficient against today's sophisticated and evolving cyber security threats. This has motivated the development of *self-protecting software* that is capable of detecting security threats and mitigating them through runtime adaptation techniques. In this paper, we argue for an *architecture-based self- protection (ABSP)* approach to address this challenge. In ABSP, detection and mitigation of security threats are informed by an architectural representation of the running system, maintained at runtime. With this approach, it is possible to reason about the impact of a potential security breach on the system, assess the overall security posture of the system, and achieve defense in depth. To illustrate the effectiveness of this approach, we present several architecture adaptation patterns that provide reusable detection and mitigation strategies against well-known web application security threats. Finally, we describe our ongoing work in realizing these patterns on top of Rainbow, an existing architecture-based adaptation framework.

## Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

## Keywords

Self-Protection; Software Architecture; Software Security

## 1. INTRODUCTION

Security is increasingly the principal concern that drives the design and construction of modern software systems. Conventional, often manually developed and statically employed, techniques for securing software systems are no longer sufficient. As adversaries become more agile in devising new threats, so should the mechanisms for securing the software systems. This has motivated the development of *self-protecting software*—a new kind of software, capable of detecting security threats and mitigating them through runtime adaptation techniques.

Most self-protection research to-date has focused on specific line(s) of defense (e.g., network, host, or middleware) within a software system. Specifically, such approaches tend to focus on a specific type or category of threats, implement a single strategy or technique, and/or protect a particular component or layer of the system [31]. As a result, the "big picture" understanding of overall security posture and globally coordinated defense strategies appear to be lacking. In addition, growing threats of insider attack call for new mechanisms to complement traditional perimeter-based security (i.e., securing the system at its boundaries) that has been the main focus of prior research. Finally, due to the added complexity of dynamically detecting and mitigating security threats, the construction of self-protecting software systems has shown to be significantly more challenging than traditional software [7]. Lack of engineering principles and repeatable methods for the construction of such software systems has been a major hindrance to their realization and adoption by industry.

In this paper, we argue for an *architecture-based self-protection (ABSP)* approach to address the aforementioned challenges. In ABSP, detection and mitigation of security threats are informed by an architectural representation of the software that is kept in synch with the running system. An architectural focus enables the approach to assess the overall security posture of the system and to achieve defense in depth, as opposed to point solutions that operate at the perimeters. By representing the internal dependencies among the system's constituents, ABSP provides the means to tackle issues such as insider attack. The architectural representation also allows the system to reason about the impact of a security breach on the system, which would inform the recovery process.

As concrete evidence of how ABSP promotes a disciplined and repeatable process for engineering self-protecting software systems, we will present several *architecture-level self-protection patterns*. These patterns provide reusable detection and adaptation strategies for solving well-known security threats. We illustrate their application in dealing with commonly encountered security threats in the realm of web-based applications.

Finally, we describe our work in realizing some of these patterns on top of an existing architecture-based adaptation framework, namely Rainbow [9]. The resulting framework holds promise for increasing reuse across applications/domains and reducing the effort required in realizing self-protection capabilities. In the process of extending Rainbow, we have faced several challenging research questions that will frame future research in this area.

The remainder of paper is organized as follows: in Section 2 we set the stage with a simple web application as a motivating example, so that we can illustrate in a concrete setting how ABSP can help thwart security attacks. In Section 3 we briefly outline related self-protection research and point out their limitations, before making the case for the architecture-based approach in Section 4. In section 5 we proceed to elaborate the details of ABSP using a number of repeatable architecture adaptation patterns. In Section 6 we provide an overview of our ongoing work in implementing the security adaptation patterns in the Rainbow framework. Finally, we conclude the paper with a summary of our key contributions, a discussion of remaining challenges, and areas of future research.

## 2. MOTIVATING EXAMPLE

Based on real sites like cnn.com, Znn [6] is a news service that serves multimedia news content to its customers. Architecturally, Znn is a web-based client-server system that conforms to an N-tier style. As illustrated in Figure 1, the service provides web browser-based access to a set of clients. To manage a high number of client requests, the site uses a load balancer to balance requests across a pool of replicated servers (two shown), the size of which can be configured to balance server utilization against service response time. For simplicity sake we assume news content is retrieved from a database and we are not concerned with how they are populated. We further assume all user requests are stateless HTTP requests and there are no transactions that span across multiple HTTP requests. This base system does not yet have any architectural adaptation features, but serves as a good starting point for our later discussions.

To illustrate the ABSP approach as well as self-adaptation concepts, we augment Znn's basic web application architecture with a "meta component" called ARchitecture Manager (ARM), shown in Figure 1, which is responsible for monitoring the components and connectors in the system and adapting them in accordance with self-protection goals. To achieve this, the ARM implements the Monitor, Analyze, Plan, Execute (MAPE) loop [14], and is connected to "sensors" throughout the system to monitor their current status and "effectors" that can receive adaptation commands to make architecture changes to the base system at runtime. As described later in the paper, the Rainbow platform [9] serves as a good implementation of the ARM component using architecture-based techniques combined with control and utility theories.

In the real world, a public website like Znn faces a wide variety of security threats. Malicious attacks, both external and internal, grow more complex and sophisticated by the day. The Open Web Application Security Project (OWASP) maintains a Top Ten list [22] that provides a concise summary of what was considered to be the ten most critical security risks at the application level. They are listed in Table 1. Another list published by the MITRE Corporation, the Top 25 Common Weakness Enumeration (CWE) [26], covers many similar threats with more details.

We will use Znn throughout the paper as a running example to illustrate how architecture-based approaches may be utilized to help counter such threats in an autonomous and self-managed fashion. Note that we focus our attention on the OWASP Top 10 and CWE Top 25 threats due to their prevalence on the Internet and their relevance to the Znn web application. However the same general approach can be applied to counter other security threats not covered in the two lists, such as Denial of Service attacks, buffer overflows, privilege escalation, etc.

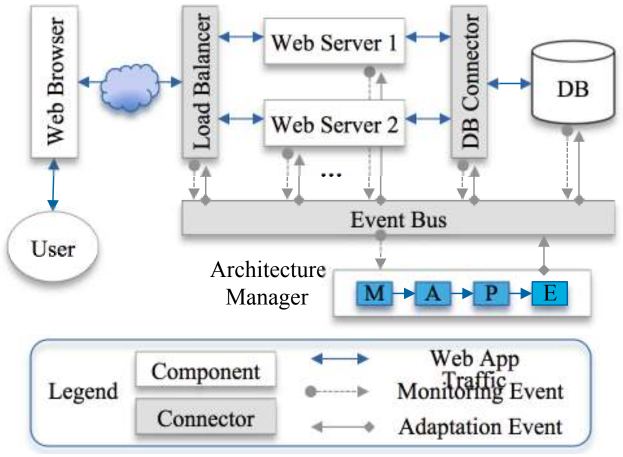

**Figure 1: Znn Self-Adaptive Architecture**

**Table 1: OWASP Top 10, 2010 Edition**

| | |
|---|---|
| A1 | Injection |
| A2 | Cross-Site Scripting (XSS) |
| A3 | Broken Authentication and Session Management |
| A4 | Insecure Direct Object References |
| A5 | Cross-Site Request Forgery (CSRF) |
| A6 | Security Misconfiguration |
| A7 | Insecure Cryptographic Storage |
| A8 | Failure to Restrict URL Access |
| A9 | Insufficient Transport Layer Protection |
| A10 | Unvalidated Redirects and Forwards |

## 3. STATE OF THE ART

In SEAMS 2012 the authors presented a preliminary study of self-protecting software systems [31]. We have since expanded that study following a systematic literature review process proposed by [15]. This has broadened the scope of our study from 32 to 107 papers and strengthened the validity of our conclusions. To the best of our knowledge, this study [32] is the most comprehensive investigation of the literature in this area of research.

One of the most evident findings in our survey is that most self-protection research to-date focuses on specific layers or "lines of defense" in a software system, namely:

- *Network*, involving security of communication links, networking protocols, and data packets.
- *Host*, involving the host environment on a machine, including hardware/firmware, OS, and in some occasions hypervisors that support virtual machines.
- *Middleware* such as application servers, object brokers, messaging hubs and service buses.
- *Application* level, concerning programming language security and application-specific measures.

Such research, no matter how effective, tends to be point solutions that lack the overall picture of the system's security posture. For example, the business context in which the system is deployed and inter-component collaboration to help defend against security breaches are not considered by these solutions. On the other hand, self-adaptive approaches that focus on software architecture as a whole (and as such can be applied to any/all component layers in a coordinated and consistent fashion), can provide a context for reasoning about these business- and application-specific concerns.

A related finding in our survey also showed that self-protection research has been predominantly perimeter- focused, aiming to protect the system at its boundary. The historical emphasis of the computer security community on network intrusion detection and intrusion prevention (ID/IP) helps to explain this. Even though we observed encouraging trends from intrusion detection systems to intrusion tolerant systems (ITS) [18] and automated intrusion response systems (IRS) [25] in the past decade, the continued influence of the intrusion-centric mindset is evident. Systems relying solely on perimeter security, however, are often rendered helpless when the perimeter is breached; nor can they effectively deal with threats that originate from inside of the system. With growing threats of insider attacks [23], we see a pressing need for architectural approaches that monitor and adapt overall system behavior.

Of equal relevance to this paper are the recurring self-protection patterns revealed in the same survey. Architecture and design patterns in general have been well-researched and documented, but we observed a number of interesting patterns have emerged as being especially effective in establishing self-protecting behavior. We identified a number of *structural patterns* and *behavioral patterns*, a subset of which will be applied to ABSP in Section 5.

## 4. THE CASE FOR ARCHITECTURE-BASED SELF-PROTECTION

Mature and effective defense mechanisms are readily available to make the Znn news site more secure. At the network and host levels, for example, one can:

- Place a traffic-filtering firewall at the WAN edge before the load balancer to filter all incoming TCP/IP traffic. Illegal access to certain ports and protocols from certain source IP addresses can be easily blocked.
- Install a network-based intrusion detection device on the local LAN that examines network traffic to detect abnormal patterns. Repeated requests to a single server that exceed a threshold, for example, may trigger an alarm for Denial of Service (DoS) attack.
- Install a host-based intrusion detection system in the form of software agents on all servers, which monitors system calls, application logs, and other resources. Access to a system password file without administrator permissions, for instance, may indicate the server has been compromised.

At the web application level, OWASP has also recommended a rich set of detection techniques and prevention countermeasures aimed at mitigating the aforementioned Top 10 risks. Some key practices include [22]:

- Conduct code reviews and blackbox penetration testing to find security flaws proactively;
- Employ static and dynamic program analysis tools to identify application vulnerabilities. For SQL injection risks, for instance, one can conduct static code analysis to find all occurrence of the use of SQL statement interpreters;
- Use whitelist input validation to ensure all special characters in form inputs are properly escaped to prevent Cross-Site Scripting (XSS) attacks;
- Follow good coding practices and use well-tested API libraries.

The traditional approaches, however, are not without limitations. First, most are labor-intensive and require significant manual effort during development and/or at runtime. Second, the testing tools and preventive countermeasures do not provide complete and accurate defense against rapidly changing attack vectors, as admitted in the OWASP Top 10 report. Many approaches find it difficult to balance the competing goals of maximizing detection coverage (reducing false negatives) and maintaining detection accuracy (reducing false positives). Furthermore, for web attacks such as XSS and CSRF, they are partly caused by careless and unwitting user behavior, which is impossible to completely eliminate. Last but not the least, a large percentage of the web applications today are so-called "legacy" applications for which development had ended some time ago. Even when vulnerabilities are found, the fixes are going to be difficult and costly to implement.

The Architecture-based Self-Protection (ABSP) approach does not seek to replace the mainstream security approaches but rather to complement them. ABSP focuses on securing the architecture as a whole, as opposed to specific components such as networks or servers. Working primarily with constructs such as components, connectors and architecture styles, the ABSP approach protects the system against security threats by (a) modeling the system using machine-understandable representations, (b) incorporating security objectives as part of the system's architectural properties that can be monitored and reasoned with, and (c) making use of autonomous computing principles and techniques to dynamically adapt the system at runtime in response to security threats, without necessarily modifying any of the individual components.

The Benefits of ABSP are as follows:

- Defense in depth. Most self-protection approaches that have originated in the systems community are perimeter based. By modeling the internal structure of a software system in ABSP, it provides an effective mechanism to deal with threats in multiple stages and at different levels.
- Impact Analysis. ABSP allows us to reason about threat impacts as well as the trustworthiness of a software system at the granularity of its elements (components, connectors). The dependencies among the system's constituents would help us track the impact of a compromised element on the other parts of the system and formulate globally coordinated defense strategies.
- Insider attack. Modeling and monitoring the software system in terms of its architectural constituents make it possible to detect abnormal behavior inside the software system. This allows us to mitigate security threats that originate from within the system.

- Reuse. By implementing self-protection patterns as orthogonal concerns, separate from application logic, the former may be easily reused in other applications.

- Dynamism. Separation of concerns also allows the self-protection mechanisms to evolve independent of the application logic, to quickly adapt to emerging threats.

## 5. ARCHITECTURE PATTERNS FOR SELF-PROTECTION

In this section, we use patterns as a convenient way to illustrate how ABSP may be used to bring self-securing capabilities to a system such as Znn. For each pattern, we briefly describe the security threat(s) it can be effective for, how the threat could be detected, and finally how it could be dealt with through adaptation. It is worth noting that the self-protection patterns described here represent architectural level adaptation strategies, and are therefore different from previously identified reusable security patterns [12, 30], which constitute for the most part implementation tactics such as authentication, authorization, password synchronization, etc.

### 5.1 Protective Wrapper Pattern

This simple pattern involves placing a security enforcement proxy, wrapper, or container around the protected resource, so that request to / response from the resource may be monitored and sanitized in a manner transparent to the resource. Protective wrappers are not uncommon in past self-protection research. The SITAR system [29] protects COTS servers by deploying an adaptive proxy server in the front, which detects and reacts to intrusions. Invalid requests/responses trigger reconfiguration of the COTS server. Virtualization techniques are increasingly being used as an effective protective wrapper platform. VASP [33], for example, is a hypervisor-based monitor that provides a trusted execution environment to monitor various malicious behaviors in the OS.

#### *5.1.1 Architectural Adaptation*

One straightforward way to employ this pattern for Znn is to place a new connector called "Application Guard" in front of the load balancer, as shown in Figure 2. To support this change, the ARM not only needs to connect to the application guard via the event bus, but also needs to update its architecture model to define additional monitoring events for this new element (e.g. suspicious content alerts) and additional effector mechanisms for its adaptation (e.g. blocking a user).

#### *5.1.2 Threat Detection*

The application guard serves as a protective wrapper for the Znn web servers by performing two key functions: attack detection and policy enforcement. By inspecting and if necessary sanitizing the incoming HTTP requests, the application guard can detect and sometimes eliminate the threats before they reach the web servers. Injection attacks (OWASP A1), for example, often contain special characters such as single quotes which will cause erroneous behavior in the backend database when the assembled SQL statement is executed. By performing input validation (e.g. using a "white list" of allowed characters) or using proper escape routines, the wrapper can thwart many injection attempts.

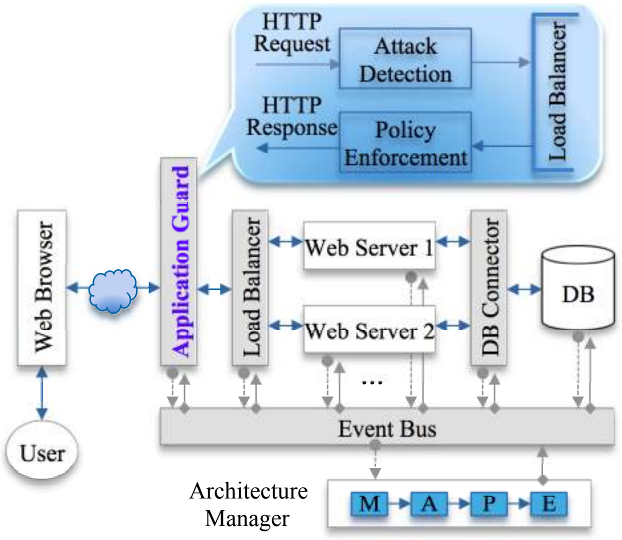

**Figure 2: Znn Protective Wrapper Architecture**

As its name suggests, this protective wrapper works at the application level, in contrast to conventional network firewalls that focus on TCP/IP traffic. It communicates with and is controlled by the model-driven ARM "brain", and as such can help detect more sophisticated attack sequences that are multi-step and cross-component. For example, the ARM can signal the application guard to flag and stop all access requests to the web server document root, because a sensor detected a buffer overflow event from the system log of the web server host. The latter may have compromised the web server and placed illegitimate information at the document root (e.g., a defaced homepage, or confidential user information). The detection may be achieved through incorporating an attack graph in the ARM's architecture model, as described in [8].

#### *5.1.3 Threat Mitigation*

A second function performed by the application guard is policy enforcement as directed by the ARM. Take Broken Authentication and Session Management (OWASP A3) for example; web sites often use URL rewriting which puts session IDs in the URL: `http://znn.com/premium/newsitem;sessionid=SIRU3847YN9W38475N?newsid=43538` When this URL is either shared or stolen, others will be able to hijack the session ID to access this user's content or even his/her personal information. The application guard can easily prevent this by applying encryption / obfuscation techniques so session IDs cannot be identified from the URL, or by tying session IDs with user's MAC addresses so that session IDs cannot be reused even if they are stolen. Similar mechanisms at the application guard may also help patch up other vulnerabilities such as Insecure Direct Object References (OWASP A4) and Failure to Restrict URL Access (OWASP A8).

More adaptive enforcement actions are possible thanks to the ARM component that is aware of the overall system security posture. After the ARM senses the system is under attack, it can instruct the application guard to dynamically cut off access to a compromised server, switch to a stronger encryption method, or adjust trustworthiness of certain users. Adaptation strategies may be based on heuristic metrics indicating the overall system's security posture, which are computed in real time. As a concrete example, in section 6 we will show how this pattern is employed against denial of service (DoS) attacks.

## 5.2 Software Rejuvenation Pattern

As defined by [13], the Software Rejuvenation pattern involves gracefully terminating an application instance and restarting it at a clean internal state. This pattern is part of a growing trend of proactive security strategies that have gained ground in recent years. By proactively "reviving" the system to its "known good" state, one can limit the damage of undetected attacks, though at the cost of extra hardware resources. The TALENT system [20], for example, addresses software security and survivability using a "cyber moving target" approach, which proactively migrates running applications across different platforms on a periodic basis while preserving application state. The Self-Cleansing Intrusion Tolerance (SCIT) architecture [17] uses redundant and diverse servers to periodically "self-cleanse" the system to pristine state. the Proactive Resilience Wormhole (PRW) effort [24] also employs proactive rejuvenation for intrusion tolerance and high availability.

### *5.2.1 Architectural Adaptation*

When applying this pattern to the Znn application, we will update the ARM's system representation to establish two logical pools of web servers: in addition to the active server pool connected to the load balancer, there will also be an idle web server pool containing running web servers in their pristine state, as shown in Figure 3. These server instances could be either separate software processes or virtual machine instances. In the simplest case, the ARM will issue rejuvenation commands at regular intervals (e.g., every 5 minutes, triggered by a timer event), which activate a new web server instance from the idle pool and connects it to the load-balancer. At the same time, an instance from the active pool will stop receiving new user requests and terminate gracefully after all its current user sessions log out or time out. The instance will then be restarted and returned to the idle pool.

The ARM "brain" may also pursue more complex rejuvenation strategies, such as:

- Use threat levels or other architectural properties to determine and adjust rejuvenation intervals at runtime
- Perform dynamic reconfigurations and optimizations (e.g. restart a server instance with more memory based on recent server load metrics)
- Mix diverse web server implementations (e.g. Apache and Microsoft IIS) to thwart attacks exploiting platform-specific vulnerabilities

Note that, the rejuvenation process, short as it may be, temporarily reduces system reliability. Extra care is needed to preserve application state and transition applications to a rejuvenated replica.

### *5.2.2 Threat Detection*

A unique characteristic about the rejuvenation pattern is that it neither helps nor is dependent upon threat detection, but for the most part used as a mitigation technique.

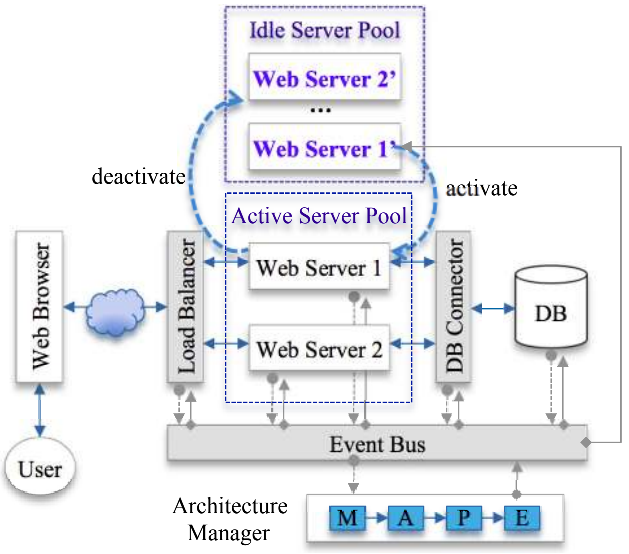

**Figure 3: Znn Software Rejuvenation Architecture**

### *5.2.3 Threat Mitigation*

Although the rejuvenation pattern doesn't eradicate the underlying vulnerability, it can effectively limit potential damage and restore system integrity under injection (OWASP A1), reflective XSS (OWASP A2), and to some extent CSRF (OWASP A5) attacks, detected or undetected. These are considered among the most vicious and rampant of web application threats, in part because the attack vector is often assisted by careless or unsuspecting users. Clicking on a phishing URL is just one of the many examples. When a fragment of malicious code is sent to the server, such as the following that steals user cookies [21]:

```
<SCRIPT type="text/javascript">
var adr = 'example.com/evil.php?cakemonster='
          + escape (document.cookie);
</SCRIPT>
```

This piece of injected code may be stored in server memory or (in a worse case) in the database, and then used for malicious intents such as stealing confidential user information, hijacking the server to serve up malware, or even defacing the website - and continue doing so *as long as the server is running.*

With a rejuvenation pattern in place, a server may only be compromised for up to the rejuvenation interval. In mission-critical operations, the interval can be as short as a few seconds, drastically reducing the probability and potential damage from these attacks even when detection sensors fail.

Our pattern implementation as depicted in Figure 3 does have some limitations. First, for persistent attacks such as DoS, rejuvenating the web server will not be effective because the DoS traffic will simply be directed to the new web server instance and overwhelm it. In such cases rejuvenation must be carried out in conjunction with other countermeasures such as blocking the attacking source. Secondly, caution must be taken so that corrupted state is not migrated to the new instances. For example, when malicious code is stored in the database, simply recycling the web server will not eradicate the root of the threat because restarting a database server instance will only clean up transient, in-

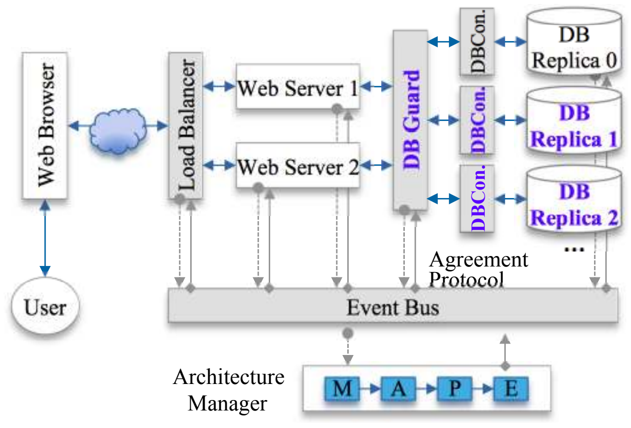

**Figure 4: Znn Agreement-Based Redundancy Architecture**

memory storage but has no effect on data changes already committed to permanent storage. This pattern, therefore, is not an effective mechanism against stored XSS attacks.

## 5.3 Agreement-based Redundancy

As pointed out in the previous subsection, proactively "hot swapping" active and possibly tainted web servers with new pristine instances can effectively limit the damage of scripting attacks that seek to inject malicious code in the web server, but the technique can offer little relief to attacks that have succeeded in permanently altering the system state, particularly in the database. To address the latter challenge, we consider another architecture pattern, Agreement-based Redundancy, which maintains multiple replica of a software component at runtime in order to tolerate Byzantine faults and ensure correctness of results. A prime example of this pattern comes from the seminal work by [4] described a Byzantine Fault Tolerance (BFT) algorithm that can effectively tolerate $f$ faulty nodes with $3f+1$ replicas within a short window of vulnerability. Similar agreement-based voting protocols have been used in many other systems such as SITAR and [28]. The strengths of this pattern is many-fold - it is easy to implement, performs well, helps meet both system security and availability goals, and is effective against unknown attacks.

### *5.3.1 Architectural Adaptation*

In the Znn example we choose to apply this pattern to the database layer, as shown in Figure 4. First, we update the ARM's architecture representation to maintain a number of identical database instances (along with their respective database connectors), all active and running concurrently. Secondly, a new connector called DB Guard is introduced to handle database requests from web servers using an agreement-based protocol. The ARM communicates the agreement-based protocol specifics to the DB Guard, such as the number of replicas and quorum thresholds. The ARM can dynamically adapt the protocol as needed at runtime.

### *5.3.2 Threat Detection*

Given the heavy reliance on databases in today's software applications, it is no surprise SQL injection is ranked as the number one threat in both OWASP Top 10 and CWE Top 25. The ABR pattern can effectively detect and stop the SQL variants of the injection attack (OWASP A1) and stored XSS (OWASP A2) attack when they contain illegal writes to the Znn database. Consider a simplified scenario where the Znn web server attempts to retrieve news articles from a database table based on keyword:

```
...
string kw = request.getParameter("keyword");
string query = "SELECT * FROM my_news
  WHERE keyword = '" + kw + "'";
...
```

Note that many database servers allow multiple SQL statements separated by semicolons to be executed together. If the attacker enters the following string:

```
xyz'; DELETE FROM news; --
```

Then two valid SQL statements will be created (note the training pair of hyphens will result in the trailing single quote being treated as a comment thus avoiding generating a syntax error, see [27] for details):

```
SELECT * FROM my_news WHERE keyword='xyz';
DELETE FROM news;
```

As a result, a seemingly harmless database query could be used to corrupt the database and result in loss of data. Good design and coding techniques, along with static analysis tools can help identify vulnerabilities. As mentioned earlier in the paper, however, such efforts are labor-intensive and not bullet-proof. Using the Agreement-based Redundancy pattern, we take an non-intrusive approach that does not require code changes to the web application nor the database. Instead. we execute the following algorithm in the DB Guard connector:

1. Treat each database request $R$ as a potential fault-inducing operation, and execute it first on the primary node (replica 0). The selection of the primary node is arbitrary.
2. Use a voting algorithm to check predefined properties of the database. For example, one property may be the number of news articles. The ARM is responsible for defining and monitoring these properties and making them available to the database connector. When quorums can be reached on all properties and the primary node is part of the quorum, proceed to next step; otherwise flag $R$ as invalid and revert the primary node to its state before $R$, either by rolling back the transaction or by making a copy of another replica.
3. Execute $R$ on all other replicas 1 to $n$, bringing all replicas to the same state.
4. Adjudicate the results from all replicas using the voting algorithm. If a quorum is reached, return the result to client; otherwise consider the system in a compromised state and raise flag for human administrator attention.

It is easy to see that when the above attack string gets sent to the DB Guard and executed in the primary node, the number of news articles is reduced to zero after the delete command, therefore different from the quorum. The request will be aborted and the system reverted to its valid state. Our implementation, however, comes with a caveat: it is not effective when the SQL injection seeks only to read data (i.e. the compromise is in system confidentiality, not integrity). In the latter case, the protective wrapper pattern can still help inspect and detect the anomaly.

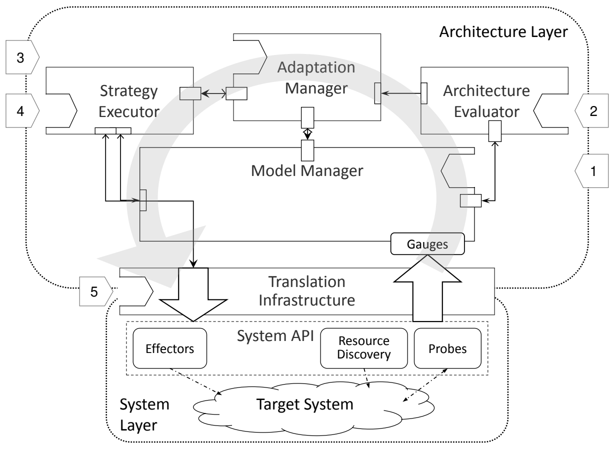

**Figure 5: The Rainbow Architecture.**

### 5.3.3 *Threat Mitigation*

As we have seen from the above scenario, the SQL injection attack is effectively stopped after it is detected in the algorithm. To complete the full sequence for threat mitigation, we only need to furbish a few more details:

- Once an invalid and potentially malicious request is detected, the DB Guard will notify the ARM that can deploy countermeasures such as nullifying the associated user session, notifying the system administrator, or even disabling the user account in question.
- When the adjudication of the results (step 4 of the algorithm) is not unanimous, raise a flag about the minority server instance. If further diagnostics confirm the instance is not in a valid state, destroy and regenerate this instance.

## 6. REALIZING SELF-PROTECTION PATTERNS IN RAINBOW

In the previous sections, we have argued that architecture-based self-protection can provide a principled and repeatable approach to constructing self-protecting systems, and given some examples of patterns for ABSP. In this section, we outline how to implement this approach in an architecture-based self-adaptive framework called Rainbow. We begin by providing an overview of Rainbow, and then continue by discussing how the Protective Wrapper pattern can be realized by the framework.

### 6.1 Rainbow Framework Overview

The Rainbow framework has demonstrated how architecture models of the system, updated at runtime, can form the basis for effective and scalable problem detection and correction. Architecture models represent a system in terms of its high level components and their interactions (e.g., clients, servers, data stores, etc.), thereby reducing the complexity of those models, and providing systemic views on their structure and behavior (e.g., performance, availability, protocols of interaction, etc.). In the context of this paper, the Rainbow framework can be viewed as Architecture Manager capable of evaluating and adapting the underlying system to defend against threats.

The Rainbow framework uses software architectures and a reusable infrastructure to support self-adaptation of software systems. Figure 5 shows the adaptation control loop of Rainbow. Probes are used to extract information from the target system that update the architecture model via Gauges, which abstract and aggregate low-level information to detect architecture-relevant events and properties. The architecture evaluator checks for satisfaction of constraints in the model and triggers adaptation if any violation is found, i.e., an adaptation condition is satisfied. The adaptation manager, on receiving the adaptation trigger, chooses the "best" strategy to execute, and passes it on to the strategy executor, which executes the strategy on the target system via effectors.

The best strategy is chosen on the basis of stakeholder utility preferences and the current state of the system, as reflected in the architecture model. The underlying decision making model is based on decision theory and utility [19]; varying the utility preferences allows the adaptation engineer to affect which strategy is selected. Each strategy, which is written using the Stitch adaptation language [5], is a multi-step pattern of adaptations in which each step evaluates a set of condition-action pairs and executes an action, namely a tactic, on the target system with variable execution time. A tactic defines an action, packaged as a sequence of commands (operators). It specifies conditions of applicability, expected effect and cost-benefit attributes to relate its impact on the quality dimensions. Operators are basic commands provided by the target system.

As a framework, Rainbow can be customized to support self-adaptation for a wide variety of system types. Customization points are indicated by the cut-outs on the side of the architecture layer in Figure 5. Different architectures (and architecture styles), strategies, utilities, operators, and constraints on the system may all be changed to make Rainbow reusable in a variety of situations. In addition to providing an engineering basis for creating self-adapting systems, Rainbow also provides a basis for their analysis. By separating concerns, and formalizing the basis for adaptive actions, it is possible to reason about fault detection, diagnosis, and repair. For example, many of the standard metrics associated with classical control systems can, in principle, be carried over: settling time, convergence, overshoot, etc. In addition, the focus on utility as a basis for repair selection provides a formal platform for principled understanding of the effects of repair strategies.

In summary, Rainbow uses architectural models of a software system as the basis for reasoning about whether the system is operating within an acceptable envelope. If this is not the case, Rainbow chooses appropriate adaptation strategies to return the system to an acceptable operating range. The key concepts of the approach are thus: (a) the use of abstract architectural models representing the runtime structures of a system, that make reasoning about system-wide properties tractable, (b) detection mechanisms that identify the existence and source of problems at an architectural level, (c) a strategy definition language called Stitch that allows architects to define adaptations that can be applied to a system at runtime, and (d) a means to choose appropriate strategies to fix problems, taking into consideration multiple quality concerns to achieve an optimal balance among all desired properties.

## 6.2 Realizing the Protective Wrapper Pattern

In this section, we describe how the Protective Wrapper Pattern in Section 5.1 is implemented in Rainbow to protect Znn against a denial of service (DoS) attack. DoS has been extensively researched in the past, including recent efforts using adaptive approaches [1, 16]. Our focus in this paper is not so much on advancing the state of the art for DoS attack mitigation, but on illustrating how the problem may be addressed at the architectural level using repeatable patterns and a runtime self-adaptive framework.

### 6.2.1 Architecture Adaptation

At the system level, the protective wrapper is placed in front of the load balancer of Znn to achieve two levels of protection: 1) it maintains a list of black-listed IPs that are considered to be malicious, and ignores all requests from them; and 2) it maintains a list of suspicious clients that are generating an unusually large amount of traffic and limits the requests that get forwarded to the load balancer. Each of these are manipulated via effectors in Znn (in reality, scripts that can run on the load balancer) that introduce and configure the wrapper. Each script is associated with an architectural operator that can be used by tactics in Stitch to implement the mitigation.

The architecture model of Znn is annotated with properties to determine the request rates of clients, and the probability that a client is engaging a DoS (i.e., being malicious). Gauges report values for these properties (described below), and constraints check that clients have reasonable request rates and low probabilities of maliciousness, and if not, are throttled or on the blacklist. If these constraints fail, then the mitigation strategy above is applied.

In terms of customization of Rainbow, the model and its annotation with the above properties corresponds to customization point 1 in Figure 5, and the constraints that check the correctness of the model to point 2.

### 6.2.2 Threat Detection

The DoS attack is detected by probes that monitor the traffic in the system, and correspond partially to customization point 5 in Figure 5. Rainbow aggregates this data into actionable information within gauges, and then uses this information to update the architectural model to reflect operating conditions. To determine the probability of a client participating in a DoS, we follow the approach described in [3, 2]. We define transactions representing request behaviors in the architectures that are derived from low level system events, and an Oracle that analyzes the transactions from each client and, using a method called Spectrum-based Fault Multiple Fault Localization, reports the probability of each client acting suspiciously as a *maliciousness* property on each component in the model.

Furthermore, probes and gauges keep track of which clients are in the blacklist or being throttled, allowing the constraints in the model to fail only on clients that haven't been dealt with yet.

### 6.2.3 Threat Mitigation

When a threat is detected and reported in the architectural model, causing a constraint to fail, the Rainbow Adaptation Manager is invoked. It selects and executes adaptations to maximize the utility of the system. In the case of a DoS attack, maximizing utility means stopping the attack with minimal client service disruption. That is, the DoS response must take care not deny access to clients without cause.

Consider the scenario where attackers may be dealt with differently depending on the frequency with which they attack (i.e. repeat offenders) and the duration of the attack. Rules to determine how these factors influence response could be encoded into a security policy with the following logic:

- The traffic for previously unknown attackers is throttled (i.e., some requests are ignored) to limit the impact of the attack without totally cutting off service. This approach minimizes the chances of disrupting the service of possibly legitimate clients.
- Repeat offenders are *blackholed* meaning all that client's traffic is filtered at the load balancer. Known malicious clients are given less mercy than those who have not previously attacked Znn.
- Long-running attacks are not tolerated under any circumstances.

These rules are encoded as the Rainbow strategy shown in Listing 1. The applicability of the FixDosAttack strategy depends on the state of the system as it is represented in the architectural model. The strategy is only applicable if the cUnderAttack condition is true in the model.[1] Conditions such as cLongAttack (line 3) and cFreqAttacker (line 9) reflect the state of architectural properties, such as whether an attack is ongoing, and act as guards in the strategy's decision tree. This strategy captures the scenarios where infrequent and frequent attackers are dealt with by throttling or blackholing the attack respectively, unless the attack is long running. This is consistent with the logic described above.

**Listing 1: An example strategy for implementing the DoS Wrapper.**

```
1  strategy FixDoSAttack [cUnderAttack] {
2    t0: (cInfreqAttacker) -> throttle() @[2000(/*ms*/ ]
           {
3      t0a: (cLongAttack) ->  blackhole () @[2000] {
4        t0ai: (default) -> done;
5      }
6      // No more steps to take
7      t0a: (default) -> TNULL;
8    }
9    t1: (cFreqAttacker) -> blackhole() @[2000/*ms*/] {
10     t1a: (cLongAttack) -> blackhole() @[2000] {
11       t1ai: (default) -> done;
12     }
13   }
14 }
```

While strategies determine which action to take, tactics are responsible for taking the action. If a condition is true in the strategy, then the subsequent tactic is applicable. For example, an infrequent attacker would cause the cInfreqAttacker condition to be true invoking the throttle() tactic (line 2). Tactics are the specific actions to take to transform the architecture into a desired state. The tactics used in the FixDoSAttack strategy are: throttle and blackhole.

Consider the blackhole tactic shown in Listing 2. When executed, this tactic will change the ZNN system to discard traffic from specified clients (i.e., put them in a blackhole).

[1]The details of this condition are elided for space, but are written in the first-order predicate langauge of Acme [10]

The tactic has three main parts: the applicability condition that determines whether the tactic is valid for the situation, the tactics action on the architecture, and the tactic's anticipated affect on the model of the system.

**Listing 2: Tactic to black hole an attacker.**

```
tactic blackholeAttacker () {
 condition {
   // check any malicious clients
   cUnblackholedMaliciousClients
 }
 action {
  // Collect all attackers
  set evilClients =
    { select c : T.ClientT in M.components |
      c.maliciousness > M.MAX_MALICIOUS};
  for (T.ClientT target : evilClients) {
    // black hole the malicious attacker
    Sys.blackhole(M.lbproxy, target);
  }
 }
 effect {
  // all the malicious clients blackholed.
  !cUnblackholedMaliciousClients
 }
}
```

The tactic's applicability condition relies on whether or not a client is currently attacking, indicated by a maliciousness threshold property in the architecture. If a gauge sets the maliciousness property of the suspected attacker above the maliciousness threshold, then blackholing is a viable action to take. In this sense, the tactic provides an additional checkpoint that more closely aligns with the architecture.

The action part of the tactic places clients that are identified as attackers in the blackhole by invoking the Sys.blackhole(...) operation. This operation is bound to an effector that actually modifies the Znn system to drop attacker packets, by adding the client to the blacklist, adding the filter, and restarting the load balancer.

Finally, the effect section verifies that the action had the intended impact on the system. In terms of acting as a protective wrapper, this tactic adapts Rainbow that it can intercept and discard malicious traffic before it reaches the Znn infrastructure.

## 7. CONCLUDING REMARKS

This paper has illustrated the benefits of evaluating the security properties of a software system using architectural models, and in particular at runtime. In support of our argument, we illustrated (1) how existing ad hoc techniques to self-protection can be formulated as architecture-level patterns, thus paving the way for a systematic engineering approach to construction of such systems, and (2) how those patterns could be realized in Rainbow. Our experiences with extending Rainbow to protect web applications have corroborated some of the hypothesized benefits of architecture-based self-protection (recall section 4). For instance, by explicitly separating the DoS detection and mitigation capability from the application logic, our solution can be easily reused in any other web application managed by Rainbow. In addition, by representing the self-protection capabilities as architectural elements (e.g., a protective wrapper connector), our solution allows us to reason about the security posture of a system in terms of its architectural configuration. This also enables adaptation of self-protection mechanism itself, e.g., addition and removal of new wrapper connectors.

While our experiences have been very positive so far, a number of research challenges remain:

*Quantifying security.* One of the difficulties in automatically making adaptation decisions is the lack of established and commonly accepted metrics for the quantification of security. Good security metrics are needed to enable comparison of candidate adaptations, evaluate the effect adaptations have on security properties, and quantify overall system security posture. However, there are few security metrics that can be applied at an architectural level.

Architectural-level metrics are preferred because they reflect the system security properties affected by adaptations. Such metrics could include measures to classify security based on applied adaptations and evaluate the impact of adaptations on certain security properties (e.g., attack surface, number least privilege violations). We have done some preliminary work in quantifying the attack surface with respect to an architecture in [11]. With the aid of such metrics, in our future work we plan to develop better mechanisms to better select adaptations that promote desired properties and quantify the impact on system security.

*Quality attribute tradeoffs.* In fielded systems security must be considered with other, possibly conflicting, quality attributes. Rainbow makes tradeoffs between quality attributes with each adaptation, selecting an adaptation that maximizes the overall utility of the system. Principled mechanisms are needed to evaluate the impacts of these tradeoffs as the system changes. Consider that almost all self-protection patterns described in Section 5 come at the expense of other quality attributes (e.g. response time, availability). Mechanisms to automatically evaluate competing quality attributes is critical for effective self adaptation.

Software architecture is the appropriate medium for evaluating such tradeoffs automatically because it provides a holistic view of the system. Rainbow reasons about a multi-dimensional utility function, where each dimension represents the user's preferences with respect to a particular quality attribute, to select an appropriate strategy. We plan to evaluate this approach with respect to security, which requires quantifying security as above.

*Protecting the self-protection logic.* Most of the research to date has assumed the self-protection logic itself is immune from security attacks. One of the reasons for this simplifying assumption is that prior techniques have not achieved a disciplined split between the protected subsystem and the protecting subsystem. In our approach, the inversion of dependency and clear separation of application logic from Rainbow present us with an opportunity to address this problem. The inversion of dependency allows us to layer the self-protection logic recursively, and thus have one instance of Rainbow protect another instance of it. The fact that the two subsystems are separated allows us to leverage techniques such as virtualization, such that Rainbow would execute on a separate virtual machine, thereby reducing the likelihood of it being compromised by the same threats targeted at the application logic.

As we explore the avenues of future research outlined above, we hope the paper provides an impetus for others to do the same.

## 8. ACKNOWLEDGMENTS

This work is supported in part by awards W911NF-09-1-0273 from Army Research Office, D11AP00282 from Defense

Advanced Research Projects Agency, and CCF-1217503 and CCF-1252644 from National Science Foundation.

## 9. REFERENCES

[1] Barna, C. et al. Model-based adaptive dos attack mitigation. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (2012), pp. 119–128.

[2] Casanova, P. et al. Diagnosing architectural run-time failures. To appear in SEAMS, 2013.

[3] Casanova, P. et al. Architecture-based run-time fault diagnosis. In *Proceedings of the 5th European Conference on Software Architecture* (2011).

[4] Castro, M., and Liskov, B. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst. 20*, 4 (Nov. 2002), 398–461.

[5] Cheng, S.-W., and Garlan, D. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software, Special Issue on State of the Art in Self-Adaptive Systems 85*, 12 (December 2012).

[6] Cheng, S.-W. et al. Evaluating the effectiveness of the rainbow self-adaptive system. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09* (May 2009), pp. 132 –141.

[7] Chess, D. M. et al. Security in an autonomic computing environment. *IBM Systems Journal 42*, 1 (2003), 107–118.

[8] Foo, B. et al. ADEPTS: adaptive intrusion response using attack graphs in an e-commerce environment. In *International Conference on Dependable Systems and Networks, 2005. DSN 2005. Proceedings* (July 2005), pp. 508–517.

[9] Garlan, D. et al. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer 37*, 10 (Oct. 2004), 46–54.

[10] Garlan, D. et al. Acme: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*, G. Leavens and M. Sitaraman, Eds. Cambridge University Press, 2000, pp. 47–68.

[11] Gennari, J., and Garlan, D. Measuring attack surface in software architecture. Tech. Rep. CMU-ISR-11-121, Institute for Software Research, School of Computer Science, Carnegie Mellon University, 2011.

[12] Hafiz, M. et al. Organizing security patterns. *IEEE Software 24*, 4 (Aug. 2007), 52–60.

[13] Huang, Y. et al. Software rejuvenation: analysis, module and applications. In *, Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers* (June 1995), pp. 381–390.

[14] Kephart, J., and Chess, D. The vision of autonomic computing. *Computer 36*, 1 (Jan. 2003), 41–50.

[15] Kitchenham, B. Procedures for performing systematic reviews. *Keele, UK, Keele University 33* (2004).

[16] Li, M., and Li, M. An adaptive approach for defending against ddos attacks. *Mathematical Problems in Engineering* (2010).

[17] Nagarajan, A. et al. Combining intrusion detection and recovery for enhancing system dependability. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)* (June 2011), pp. 25 –30.

[18] Nguyen, Q., and Sood, A. A comparison of intrusion-tolerant system architectures. *IEEE Security Privacy 9*, 4 (Aug. 2011), 24–31.

[19] North, D. A tutorial introduction to decision theory. *IEEE Transactions on Systems Science and Cybernetics 4*, 3 (1968), 200–210.

[20] Okhravi, H. et al. Creating a cyber moving target for critical infrastructure applications using platform diversity. *International Journal of Critical Infrastructure Protection 5*, 1 (Mar. 2012), 30–39.

[21] OWASP.org. Cross-site scripting (XSS) - OWASP. https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[22] OWASP.org. Owasp top ten project. https://www.owasp.org/index.php/Category: OWASP_Top_Ten_Project.

[23] Sibai, F., and Menasce, D. Defeating the insider threat via autonomic network capabilities. In *2011 Third International Conference on Communication Systems and Networks (COMSNETS)* (Jan. 2011).

[24] Sousa, P. et al. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems 21*, 4 (Apr. 2010), 452–465.

[25] Stakhanova, N. et al. A taxonomy of intrusion response systems. *International Journal of Information and Computer Security 1*, 1 (Jan. 2007), 169–184.

[26] The MITRE Corporation. CWE - 2011 CWE/SANS top 25 most dangerous software errors. http://cwe.mitre.org/top25/.

[27] The MITRE Corporation. CWE-89: improper neutralization of special elements used in an SQL command ('SQL injection'). http://cwe.mitre.org/data/definitions/89.html.

[28] Valdes, A. et al. An architecture for an adaptive intrusion-tolerant server. In *Security Protocols*, B. Christianson, B. Crispo, J. Malcolm, and M. Roe, Eds., vol. 2845 of *Lecture Notes in Computer Science.* Springer Berlin / Heidelberg, 2004, pp. 569–574.

[29] Wang, F. et al. SITAR: a scalable intrusion-tolerant architecture for distributed services. In *Foundations of Intrusion Tolerant Systems, 2003* (2003), pp. 359–367.

[30] Yoshioka, N. et al. A survey on security patterns. *Progress in Informatics 5*, 5 (2008), 35–47.

[31] Yuan, E., and Malek, S. A taxonomy and survey of self-protecting software systems. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (2012).

[32] Yuan, E. et al. A survey of self-protecting software systems. In *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* (June 2013).

[33] Zhu, M. et al. VASP: virtualization assisted security monitor for cross-platform protection. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (2011), pp. 554–559.