DoubleGuard: Detecting Intrusions In Multi-tier Web Applications

Meixing Le
George Mason University

Angelos Stavrou
George Mason University

Brent ByungHoon Kang
George Mason University

***Abstract*—Internet services and applications have become an inextricable part of daily life, enabling communication and the management of personal information from anywhere. To accommodate this increase in application and data complexity, web services have moved to a multi-tiered design wherein the web server runs the application front-end logic and data is outsourced to a database or file server.**

In this paper, we present DoubleGuard, an IDS system that models the network behavior of user sessions across both the front-end web server and the back-end database. By monitoring both web and subsequent database requests, we are able to ferret out attacks that an independent IDS would not be able to identify. Furthermore, we quantify the limitations of any multi-tier IDS in terms of training sessions and functionality coverage. We implemented DoubleGuard using an Apache web server with MySQL and lightweight virtualization. We then collected and processed real-world traffic over a 15-day period of system deployment in both dynamic and static web applications. Finally, using DoubleGuard, we were able to expose a wide range of attacks with 100% accuracy while maintaining 0% false positives for static web services and 0.6% false positives for dynamic web services.

I. Introduction

Web-delivered services and applications have increased in both popularity and complexity over the past few years. Daily tasks, such as banking, travel, and social networking, are all done via the web. Such services typically employ a web server front-end that runs the application user interface logic, as well as a back-end server that consists of a database or file server. Due to their ubiquitous use for personal and/or corporate data, web services have always been the target of attacks. These attacks have recently become more diverse, as attention has shifted from attacking the front-end to exploiting vulnerabilities of the web applications [6], [5], [1] in order to corrupt the back-end database system [40] (e.g., SQL injection attacks [20], [43]). A plethora of Intrusion Detection Systems (IDS) currently examine network packets individually within both the web server and the database system. However, there is very little work being performed on multi-tiered Anomaly Detection (AD) systems that generate models of network behavior for both web and database network interactions. In such multi-tiered architectures, the back-end database server is often protected behind a firewall while the web servers are remotely accessible over the Internet. Unfortunately, though they are protected from direct remote attacks, the back-end systems are susceptible to attacks that use web requests as a means to exploit the back-end.

To protect multi-tiered web services, Intrusion detection systems (IDS) have been widely used to detect known attacks by matching misused traffic patterns or signatures [34], [30], [33], [22]. A class of IDS that leverages machine learning can also detect unknown attacks by identifying abnormal network traffic that deviates from the so-called "normal" behavior previously profiled during the IDS training phase. Individually, the web IDS and the database IDS can detect abnormal network traffic sent to either of them. However, we found that these IDS cannot detect cases wherein normal traffic is used to attack the web server and the database server. For example, if an attacker with non-admin privileges can log in to a web server using normal-user access credentials, he/she can find a way to issue a privileged database query by exploiting vulnerabilities in the web server. Neither the web IDS nor the database IDS would detect this type of attack since the web IDS would merely see typical user login traffic and the database IDS would see only the normal traffic of a privileged user. This type of attack can be readily detected if the database IDS can identify that a privileged request from the web server is not associated with user-privileged access. Unfortunately, within the current multi-threaded web server architecture, it is not feasible to detect or profile such causal mapping between web server traffic and DB server traffic since traffic cannot be clearly attributed to user sessions.

In this paper, we present DoubleGuard, a system used to detect attacks in multi-tiered web services. Our approach can create normality models of isolated user sessions that include both the web front-end (HTTP) and back-end (File or SQL) network transactions. To achieve this, we employ a lightweight virtualization technique to assign each user's web session to a dedicated container, an isolated virtual computing environment. We use the container ID to accurately associate the web request with the subsequent DB queries. Thus, DoubleGuard can build a causal mapping profile by taking both the web sever and DB traffic into account.

We have implemented our DoubleGuard container architecture using OpenVZ [14], and performance testing shows that it has reasonable performance overhead and is practical for most web applications. When the request rate is moderate (e.g., under 110 requests per second), there is almost no overhead in comparison to an unprotected vanilla system. Even in a worst case scenario when the server was already overloaded, we observed only 26% performance overhead. The container-based web architecture not only fosters the profiling of causal mapping, but it also provides an isolation that prevents future session-hijacking attacks. Within a lightweight virtualization environment, we ran many copies of the web server instances in different containers so that each one was isolated from the rest. As ephemeral containers can be easily instantiated and destroyed, we assigned each client session a dedicated container so that, even when an attacker may be able to compromise a single session, the damage is confined to the compromised session; other user sessions remain unaffected by it.

Using our prototype, we show that, for websites that do not permit content modification from users, there is a direct causal relationship between the requests received by the front-end web server and those generated for the database back-end. In fact, we show that this causality-mapping model can be generated accurately and without prior knowledge of web application functionality. Our experimental evaluation, using real-world network traffic obtained from the web and database requests of a large center, showed that we were able to extract 100% of functionality mapping by using as few as 35 sessions in the training phase. Of course, we also showed that this depends on the size and functionality of the web service or application. However, it does not depend on content changes if those changes can be performed through a controlled environment and retrofitted into the training model. We refer to such sites as "static" because, though they do change over time, they do so in a controlled fashion that allows the changes to propagate to the sites' normality models.

In addition to this static website case, there are web services that permit persistent back-end data modifications. These services, which we call dynamic, allow HTTP requests to include parameters that are variable and depend on user input. Therefore, our ability to model the causal relationship between the front-end and back-end is not always deterministic and depends primarily upon the application logic. For instance, we observed that the back-end queries can vary based on the value of the parameters passed in the HTTP requests and the previous application state. Sometimes, the same application's primitive functionality *(i.e., accessing a table)* can be triggered by many different web pages. Therefore, the resulting mapping between web and database requests can range from one to many, depending on the value of the parameters passed in the web request.

To address this challenge while building a mapping model for dynamic web pages, we first generated an individual training model for the basic operations provided by the web services. We demonstrate that this approach works well in practice by using traffic from a live blog where we progressively modeled nine operations. Our results show that we were able to identify all attacks, covering more than 99% of the normal traffic as the training model is refined.

II. Related Work

A network Intrusion Detection System (IDS) can be classified into two types: anomaly detection and misuse detection. Anomaly detection first requires the IDS to define and characterize the correct and acceptable static form and dynamic behavior of the system, which can then be used to detect abnormal changes or anomalous behaviors [26], [48]. The boundary between acceptable and anomalous forms of stored code and data is precisely definable. Behavior models are built by performing a statistical analysis on historical data [31], [49], [25] or by using rule-based approaches to specify behavior patterns [39]. An anomaly detector then compares actual usage patterns against established models to identify abnormal events. Our detection approach belongs to anomaly detection, and we depend on a training phase to build the correct model. As some legitimate updates may cause model drift, there are a number of approaches [45] that are trying to solve this problem. Our detection may run into the same problem; in such a case, our model should be retrained for each shift.

Intrusion alerts correlation [47] provides a collection of components that transform intrusion detection sensor alerts into succinct intrusion reports in order to reduce the number of replicated alerts, false positives, and non-relevant positives. It also fuses the alerts from different levels describing a single attack, with the goal of producing a succinct overview of security-related activity on the network. It focuses primarily on abstracting the low-level sensor alerts and providing compound, logical, high-level alert events to the users. DoubleGuard differs from this type of approach that correlates alerts from independent IDSes. Rather, DoubleGuard operates on multiple feeds of network traffic using a single IDS that looks across sessions to produce an alert without correlating or summarizing the alerts produced by other independent IDSs.

An IDS such as [42] also uses temporal information to detect intrusions. DoubleGuard, however, does not correlate events on a time basis, which runs the risk of mistakenly considering independent but concurrent events as correlated events. DoubleGuard does not have such a limitation as it uses the container ID for each session to causally map the related events, whether they be concurrent or not.

Since databases always contain more valuable information, they should receive the highest level of protection. Therefore, significant research efforts have been made on database IDS [32], [28], [44] and database firewalls [21]. These softwares, such as Green SQL [7], work as a reverse proxy for database connections. Instead of connecting to a database server, web applications will first connect to a database firewall. SQL queries are analyzed; if they're deemed safe, they are then forwarded to the back-end database server. The system proposed in [50] composes both web IDS and database IDS to achieve more accurate detection, and it also uses a reverse HTTP proxy to maintain a reduced level of service in the presence of false positives. However, we found that certain types of attack utilize normal traffics and cannot be detected by either the web IDS or the database IDS. In such cases, there would be no alerts to correlate.

Some previous approaches have detected intrusions or vulnerabilities by statically analyzing the source code or executables [52], [24], [27]. Others [41], [46], [51] dynamically track the information flow to understand taint propagations and detect intrusions. In DoubleGuard, the new container-based web server architecture enables us to separate the different information flows by each session. This provides a means of tracking the information flow from the web server to the database server for each session. Our approach also does not require us to analyze the source code or know the application logic. For the static web page, our DoubleGuard approach does not require application logic for building a model. However, as we will discuss, although we do not require the full application logic for dynamic web services, we do need to know the basic user operations in order to model normal behavior.

In addition, validating input is useful to detect or prevent SQL or XSS injection attacks [23], [36]. This is orthogonal to the DoubleGuard approach, which can utilize input validation as an additional defense. However, we have found that DoubleGuard can detect SQL injection attacks by taking the structures of web requests and database queries without looking into the values of input parameters (i.e., no input validation at the web sever).

Virtualization is used to isolate objects and enhance security performance. Full virtualization and para-virtualization are not the only approaches being taken. An alternative is a lightweight virtualization, such as OpenVZ [14], Parallels Virtuozzo [17], or Linux-VServer [11]. In general, these are based on some sort of container concept. With containers, a group of processes still appears to have its own dedicated system, yet it is running in an isolated environment. On the other hand, lightweight containers can have considerable performance advantages over full virtualization or para-virtualization. Thousands of containers can run on a single physical host. There are also some desktop systems [37], [29] that use lightweight virtualization to isolate different application instances. Such virtualization techniques are commonly used for isolation and containment of attacks. However, in our DoubleGuard, we utilized the container ID to separate session traffic as a way of extracting and identifying causal relationships between web server requests and database query events.

CLAMP [35] is an architecture for preventing data leaks even in the presence of attacks. By isolating code at the web server layer and data at the database layer by users, CLAMP guarantees that a user's sensitive data can only be accessed by code running on behalf of different users. In contrast, DoubleGuard focuses on modeling the mapping patterns between HTTP requests and DB queries to detect malicious user sessions. There are additional differences between these two in terms of requirements and focus. CLAMP requires modification to the existing application code, and the Query Restrictor works as a proxy to mediate all database access requests. Moreover, resource requirements and overhead differ in order of magnitude: DoubleGuard uses process isolation whereas CLAMP requires platform virtualization, and CLAMP provides more coarse-grained isolation than DoubleGuard. However, DoubleGuard would be ineffective at detecting attacks if it were to use the coarse-grained isolation as used in CLAMP. Building the mapping model in DoubleGuard would require a large number of isolated web stack instances so that mapping patterns would appear across different session instances.

III. Threat Model & System Architecture

We initially set up our threat model to include our assumptions and the types of attacks we are aiming to protect against. We assume that both the web and the database servers are vulnerable. Attacks are network-borne and come from the web clients; they can launch application-layer attacks to compromise the web servers they are connecting to. The attackers can bypass the web server to directly attack the database server. We assume that the attacks can neither be detected nor prevented by the current web server IDS, that attackers may take over the web server after the attack, and that afterwards they can obtain full control of the web server to launch subsequent attacks. For example, the attackers could modify the application logic of the web applications, eavesdrop or hijack other users' web requests, or intercept and modify the database queries to steal sensitive data beyond their privileges.

On the other hand, at the database end, we assume that the database server will not be completely taken over by the attackers. Attackers may strike the database server through the web server or, more directly, by submitting SQL queries, they may obtain and pollute sensitive data within the database. These assumptions are reasonable since, in most cases, the database server is not exposed to the public and is therefore difficult for attackers to completely take over. We assume no prior knowledge of the source code or the application logic of web services deployed on the web server. In addition, we are analyzing only network traffic that reaches the web server and database. We assume that no attack would occur during the training phase and model building.

A. Architecture & Confinement

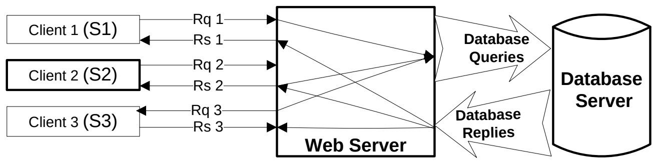

Fig. 1. Classic 3-tier Model. The web server acts as the front-end, with the file and database servers as the content storage back-end.

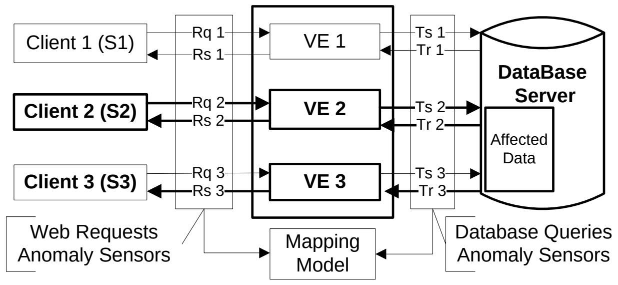

Fig. 2. Web server instances running in containers.

All network traffic, from both legitimate users and adversaries, is received intermixed at the same web server. If an attacker compromises the web server, he/she can potentially affect all future sessions (i.e., session hijacking). Assigning each session to a dedicated web server is not a realistic option, as it will deplete the web server resources. To achieve similar confinement while maintaining a low performance and resource overhead, we use lightweight virtualization.

In our design, we make use of lightweight process containers, referred to as "containers," as ephemeral, *disposable* servers for client sessions. It is possible to initialize thousands of containers on a single physical machine, and these virtualized containers can be discarded, reverted, or quickly reinitialized to serve new sessions. A single physical web server runs many containers, each one an exact copy of the original web server. Our approach dynamically generates new containers and recycles used ones. As a result, a single physical server can run continuously and serve all web requests. However, from a logical perspective, each session is assigned to a dedicated web server and isolated from other sessions. Since we initialize each virtualized container using a read-only clean template, we can guarantee that each session will be served with a clean web server instance at initialization. We choose to separate communications at the session level so that a single user always deals with the same web server. Sessions can represent different users to some extent, and we expect the communication of a single user to go to the same dedicated web server, thereby allowing us to identify suspect behavior by both session and user. If we detect abnormal behavior in a session, we will treat all traffic within this session as tainted. If an attacker compromises a vanilla web server, other sessions' communications can also be hijacked. In our system, an attacker can only stay within the web server containers that he/she is connected to, with no knowledge of the existence of other session communications. We can thus ensure that legitimate sessions will not be compromised directly by an attacker.

Figure 1 illustrates the classic 3-tier model. At the database side, we are unable to tell which transaction corresponds to which client request. The communication between the web server and the database server is not separated, and we can hardly understand the relationships among them. Figure 2 depicts how communications are categorized as sessions and how database transactions can be related to a corresponding session. According to Figure 1, if Client 2 is malicious and takes over the web server, all subsequent database transactions become suspect, as well as the response to the client. By contrast, according to Figure 2, Client 2 will only compromise the VE 2, and the corresponding database transaction set T_2 will be the only affected section of data within the database.

B. Building the Normality Model

This container-based and session-separated web server architecture not only enhances the security performances but also provides us with the isolated information flows that are separated in each container session. It allows us to identify the mapping between the web server requests and the subsequent DB queries, and to utilize such a mapping model to detect abnormal behaviors on a session/client level. In typical 3-tiered web server architecture, the web server receives HTTP requests from user clients and then issues SQL queries to the database server to retrieve and update data. These SQL queries are causally dependent on the web request hitting the web server. We want to model such causal mapping relationships of all legitimate traffic so as to detect abnormal/attack traffic.

In practice, we are unable to build such mapping under a classic 3-tier setup. Although the web server can distinguish sessions from different clients, the SQL queries are mixed and all from the same web server. It is impossible for a database server to determine which SQL queries are the results of which web requests, much less to find out the relationship between them. Even if we knew the application logic of the web server and were to build a correct model, it would be impossible to use such a model to detect attacks within huge amounts of concurrent real traffic unless we had a mechanism to identify the pair of the HTTP request and SQL queries that are causally generated by the HTTP request. However, within our container-based web servers, it is a straightforward matter to identify the causal pairs of web requests and resulting SQL queries in a given session. Moreover, as traffic can easily be separated by session, it becomes possible for us to compare and analyze the request and queries across different sessions. Section IV further discusses how to build the mapping by profiling session traffics.

To that end, we put sensors at both sides of the servers. At the web server, our sensors are deployed on the host system and cannot be attacked directly since only the virtualized containers are exposed to attackers. Our sensors will not be attacked at the database server either, as we assume that the attacker cannot completely take control of the database server. In fact, we assume that our sensors cannot be attacked and can always capture correct traffic information at both ends. Figure 2 shows the locations of our sensors.

Once we build the mapping model, it can be used to detect abnormal behaviors. Both the web request and the database queries within each session should be in accordance with the model. If there exists any request or query that violates the normality model within a session, then the session will be treated as a possible attack.

C. Attack scenarios

Our system is effective at capturing the following types of attacks:

Privilege Escalation Attack: Let's assume that the website serves both regular users and administrators. For a regular user, the web request r_u will trigger the set of SQL queries Q_u; for an administrator, the request r_a will trigger the set of admin level queries Q_a. Now suppose that an attacker logs into the web server as a normal user, upgrades his/her privileges, and triggers admin queries so as to obtain an administrator's data. This attack can never be detected by either the web server IDS or the database IDS since both r_u and Q_a are legitimate requests and queries. Our approach, however, can detect this type of attack since the DB query Q_a does not match the request r_u, according to our mapping model. Figure 3 shows

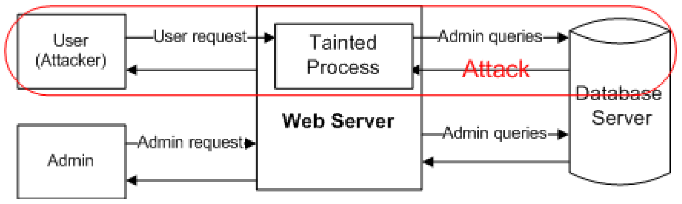

Fig. 3. Privilege Escalation Attack.

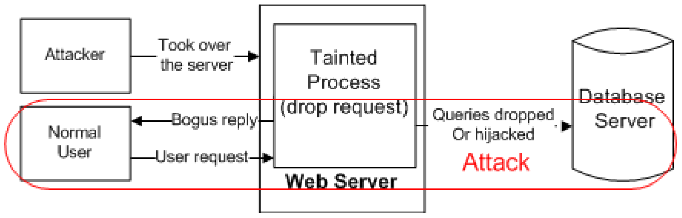

Fig. 4. Hijack Future Session Attack.

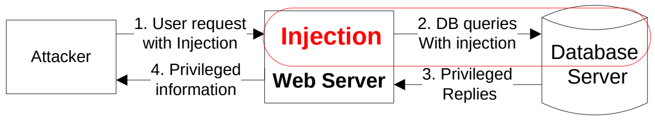
Attacker
1. User request with Injection
4. Privileged information
Injection
Web Server
2. DB queries With injection
3. Privileged Replies
Database Server

Fig. 5. Injection Attack.

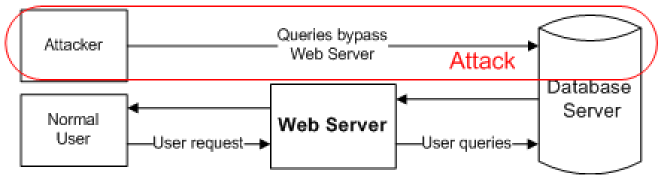

Fig. 6. DB Query without causing Web requests.

how a normal user may use admin queries to obtain privileged information.

Hijack Future Session Attack: This class of attacks is mainly aimed at the web server side. An attacker usually takes over the web server and therefore hijacks all subsequent legitimate user sessions to launch attacks. For instance, by hijacking other user sessions, the attacker can eavesdrop, send spoofed replies, and/or drop user requests. A session hijacking attack can be further categorized as a Spoofing/Man-in-the-Middle attack, an Exfiltration Attack, a Denial-of-Service/Packet Drop attack, or a Replay attack.

Figure 4 illustrates a scenario wherein a compromised web server can harm all the Hijack Future Sessions by not generating any DB queries for normal user requests. According to the mapping model, the web request should invoke some database queries (e.g., a Deterministic Mapping (section IV-A)), then the abnormal situation can be detected. However, neither a conventional web server IDS nor a database IDS can detect such an attack by itself.

Fortunately, the isolation property of our container-based web server architecture can also prevent this type of attack. As each user's web requests are isolated into a separate container, an attacker can never break into other users' sessions.

Injection Attack: Attacks such as SQL injection do not require compromising the web server. Attackers can use existing vulnerabilities in the web server logic to inject the data or string content that contains the exploits and then use the web server to relay these exploits to attack the back-end database. Since our approach provides a two-tier detection, even if the exploits are accepted by the web server, the relayed contents to the DB server would not be able to take on the expected structure for the given web server request. For instance, since the SQL injection attack changes the structure of the SQL queries, even if the injected data were to go through the web server side, it would generate SQL queries in a different structure that could be detected as a deviation from the SQL query structure that would normally follow such a web request.

Direct DB attack: It is possible for an attacker to bypass the web server or firewalls and connect directly to the database. An attacker could also have already taken over the web server and be submitting such queries from the web server without sending web requests. Without matched web requests for such queries, a web server IDS could detect neither. Furthermore, if these DB queries were within the set of allowed queries, then the database IDS itself would not detect it either. However, this type of attack can be caught with our approach since we cannot match any web requests with these queries. Figure 6 illustrates the scenario wherein an attacker bypasses the web server to directly query the database.

D. *DoubleGuard Limitations*

In this section, we discuss the operational and detection limitations of DoubleGuard.

Vulnerabilities Due to Improper Input Processing: Cross Site Scripting (XSS) is a typical attack method wherein attackers embed malicious client scripts via legitimate user inputs. In DoubleGuard, all of the user input values are normalized so as to build a mapping model based on the structures of HTTP requests and DB queries. Once the malicious user inputs are normalized, DoubleGuard cannot detect attacks hidden in the values. These attacks can occur even without the databases. DoubleGuard offers a complementary approach to those research approaches of detecting web attacks based on the characterization of input values [38].

Possibility of Evading DoubleGuard:

Our assumption is that an attacker can obtain "full control" of the web server thread that she connects to. That is, the attacker can only take over the web server instance running in its isolated container. Our architecture ensures that every client be defined by the IP address and port container pair, which is unique for each session. Therefore, hijacking an existing container is not possible because traffic for other sessions is never directed to an occupied container. If this were not the case, our architecture would have been similar to the conventional one where a single web server runs many different processes. Moreover, if the database authenticates the sessions from the web server, then each container connects to the database using either admin user account or non-admin user account and the connection is authenticated by the database. In such case, an attacker will authenticate using a non-admin account and will not be allowed to issue admin level queries. In other words, the HTTP traffic defines the privileges of the session which can be extended to the back-end database, and a non-admin user session cannot appear to be an admin session when it comes to back-end traffic.

Within the same session that the attacker connects to, it is allowed for the attacker to launch "mimicry" attacks. It is possible for an attacker to discover the mapping patterns by doing code analysis or reverse engineering, and issue "expected" web requests prior to performing malicious database queries. However, this significantly increases the efforts for the attackers to launch successful attacks. In addition, users with non-admin permissions can cause minimal (and sometimes zero) damage to the rest of the system and therefore they have limited incentives to launch such attacks.

By default, DoubleGuard normalizes all the parameters. Of course, the choice of the normalization parameters needs to be performed carefully. DoubleGuard offers the capability of normalizing the parameters so that the user of DoubleGuard can choose which values to normalize. For example, we can choose not to normalize the value "admin" in 'user = "admin"'. Likewise, one can choose to normalize it if the administrative queries are structurally different from the normal user queries, which is common case. Additionally, if the database can authenticate admin and non-admin users, then privilege escalation attacks by changing values are not feasible (i.e., there is no session hijacking).

Distributed DoS: DoubleGuard is not designed to mitigate DDoS attacks. These attacks can also occur in the server architecture without the back-end database.

IV. Modeling Deterministic Mapping and Patterns

Due to their diverse functionality, different web applications exhibit different characteristics. Many websites serve only static content, which is updated and often managed by a Content Management System (CMS). For a static website, we can build an accurate model of the mapping relationships between web requests and database queries since the links are static and clicking on the same link always returns the same information. However, some websites (e.g., blogs, forums) allow regular users with non-administrative privileges to update the contents of the served data. This creates tremendous challenges for IDS system training because the HTTP requests can contain variables in the passed parameters.

For example, instead of one-to-one mapping, one web request to the web server usually invokes a number of SQL queries that can vary depending on type of the request and the state of the system. Some requests will only retrieve data from the web server instead of invoking database queries, meaning that no queries will be generated by these web requests. In other cases, one request will invoke a number of database queries. Finally, in some cases, the web server will have some periodical tasks that trigger database queries without any web requests driving them. The challenge is to take all of these cases into account and build the normality model in such a way that we can cover all of them.

As illustrated in Figure 2, all communications from the clients to the database are separated by a session. We assign each session with a unique session ID. DoubleGuard normalizes the variable values in both HTTP requests and database queries, preserving the structures of the requests and queries. To achieve this, DoubleGuard substitutes the actual values of the variables with symbolic values. Figure 15 depicts an example of the normalizations of the captured requests and queries.

Following this step, session i will have a set of requests, which is R_i, as well as a set of queries, which is Q_i. If the total number of sessions of the training phase is N, then we have the set of total web requests REQ and the set of total SQL queries SQL across all sessions. Each single web request $r_m \in REQ$ may also appear several times in different R_i where i can be 1, 2 ... N. The same holds true for $q_n \in SQL$.

A. *Inferring Mapping Relations*

If several SQL queries, such as q_n, q_p, are always found within one HTTP request of r_m, then we can usually have an exact mapping of $r_m \rightarrow \{q_n, q_p\}$. However, this is not always the case. Some requests will result in different queries based on the request parameters and the state of the web server. For example, for web request r_m, the invoked query set can sometimes be $\{q_n,q_p\}$ or, at other times, $\{q_p\}$ or $\{q_q,q_n,q_s\}$. The probabilities for these queries are usually not the same. For 100 requests of r_m, the set is at $\{q_n,q_p\}$ 75 times, at $\{q_p\}$ 20 times, and at $\{q_q,q_n,q_s\}$ only 5 times. In such a case, we can find the mapping of $r_m \rightarrow q_p$ is 100%, with a $r_m \rightarrow q_n$ possibility of 80% and a $r_m \rightarrow q_s$ occurrence at 5% of all cases. We define this first type of mapping as deterministic and the latter ones as non-deterministic.

Below, we classify the four possible mapping patterns. Since the request is at the origin of the data flow, we treat each request as the mapping source. In other words, the mappings in the model are always in the form of one request to a query set $r_m \rightarrow Q_n$. The possible mapping patterns are as follows:

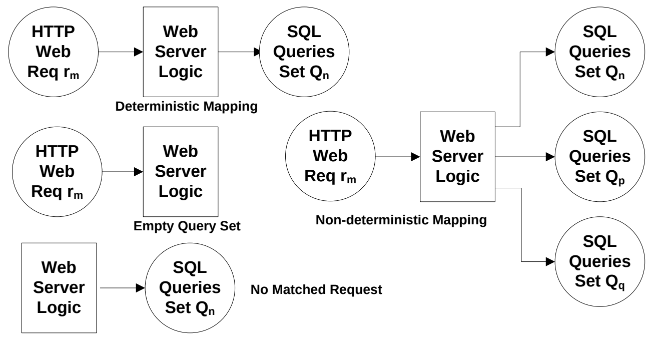

Fig. 7. Overall representation of mapping patterns.

Deterministic Mapping: This is the most common and perfectly-matched pattern. That is to say that web request r_m appears in all traffic with the SQL queries set Q_n. The mapping pattern is then $r_m \rightarrow Q_n$ ($Q_n \neq \emptyset$). For any session in the testing phase with the request r_m, the absence of a query set Q_n matching the request indicates a possible intrusion. On the other hand, if Q_n is present in the session traffic without the corresponding r_m, this may also be the sign of an intrusion. In static websites, this type of mapping comprises the majority of cases since the same results should be returned for each time a user visits the same link.

Empty Query Set: In special cases, the SQL query set may be the empty set. This implies that the web request neither causes nor generates any database queries. For example, when a web request for retrieving an image GIF file from the same web server is made, a mapping relationship does not exist because only the web requests are observed. This type of mapping is called $r_m \rightarrow \emptyset$. During the testing phase, we keep these web requests together in the set EQS.

No Matched Request: In some cases, the web server may periodically submit queries to the database server in order to conduct some scheduled tasks, such as cron jobs for archiving or backup. This is not driven by any web request, similar to the reverse case of the Empty Query Set mapping pattern. These queries cannot match up with any web requests, and we keep these unmatched queries in a set NMR. During the testing phase, any query within set NMR is considered legitimate. The size of NMR depends on web server logic, but it is typically small.

Non-deterministic Mapping: The same web request may result in different SQL query sets based on input parameters or the status of the web page at the time the web request is received. In fact, these different SQL query sets do not appear randomly, and there exists a candidate pool of query sets (e.g. $\{Q_n, Q_p, Q_q ...\}$). Each time that the same type of web request arrives, it always matches up with one (and only one) of the query sets in the pool. The mapping pattern is $r_m \rightarrow Q_i$ ($Q_i \in \{Q_n, Q_p, Q_q ...\}$). Therefore, it is difficult to identify traffic that matches this pattern. This happens only within dynamic websites, such as blogs or forum sites.

Figure 7 illustrates all four mapping patterns.

B. Modeling for Static Websites

In the case of a static website, the non-deterministic mapping does not exist as there are no available input variables or states for static content. We can easily classify the traffic collected by sensors into three patterns in order to build the mapping model. As the traffic is already separated by session, we begin by iterating all of the sessions from 1 to N. For each $r_m \in REQ$, we maintain a set AR_m to record the IDs of sessions in which r_m appears. The same holds for the database queries; we have a set AQ_s for each $q_s \in SQL$ to record all the session IDs. To produce the training model, we leverage the fact that the same mapping pattern appears many times across different sessions. For each AR_m, we search for the AQ_s that equals the AR_m. When $AR_m = AQ_s$, this indicates that every time r_m appears in a session then q_s will also appear in the same session, and vice versa.

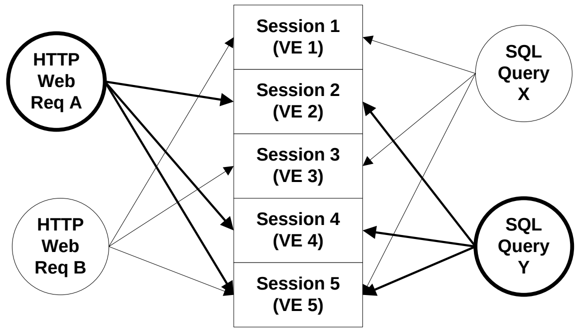

Fig. 8. Deterministic Mapping Using Session ID of the Container (VE).

Given enough samples, we can confidently extract a mapping pattern $r_m \rightarrow q_s$. Here, we use a threshold value t so that if the mapping appears in more than t sessions (e.g., the cardinality of AR_m or AQ_s is greater than t), then a mapping pattern has been found. If such a pattern appears less than t times, this indicates that the number of training sessions is insufficient. In such a case, scheduling more training sessions is recommended before the model is built, but these patterns can also be ignored since they may be incorrect mappings. In our experiments, we set t to 3, and the results demonstrate that the requirement was easily satisfied for a static website with a relatively low number of training sessions. After we confirm all deterministic mappings, we remove these matched requests and queries from REQ and SQL respectively. Since multiple requests are often sent to the web server within a short period of time by a single user operation, they can be mapped together to the same AQ_s. Some web requests that could appear separately are still present as a unit. For example, the read request always precedes the post request on the same web page. During the training phase, we treat them as a single instance of web requests bundled together unless we observe a case when either of them appears separately.

Our next step is to decide the other two mapping patterns by assembling a white list for static file requests, including JPG, GIF, CSS, etc. HTTP requests for static files are placed in the EQS set. The remaining requests are placed in REQ; if we cannot find any matched queries for them, they will also be placed in the EQS set. In addition, all remaining queries in SQL will be considered as No Matched Request cases and placed into NMR.

Figure 8 illustrates the use of the session ID provided by the container (VE) in order to build the deterministic mapping between http requests and the database requests. The request r_A has the set AR_A of {2,4,5}, which equals to AQ_Y. Therefore, we can decide a Deterministic Mapping $r_A \rightarrow q_Y$.

We developed an algorithm that takes the input of training dataset and builds the mapping model for static websites. For each unique HTTP request and database query, the algorithm assigns a hash table entry, the key of the entry is the request or query itself, and the value of the hash entry is AR for the request or AQ for the query respectively. The algorithm generates the mapping model by considering all three mapping patterns that would happen in static websites. The algorithm below describes the training process.

Algorithm 1 Static Model Building Algorithm.

```
Require: Training Dataset, Threshold $t$
Ensure: The Mapping Model for static website
 1: for each session separated traffic $T_i$ do
 2:     Get different HTTP requests $r$ and DB queries $q$ in this session
 3:     for each different $r$ do
 4:         if $r$ is a request to static file then
 5:             Add $r$ into set $EQS$
 6:         else
 7:             if $r$ is not in set $REQ$ then
 8:                 Add $r$ into $REQ$
 9:             Append session ID $i$ to the set $AR_r$ with $r$ as the key
10:     for each different $q$ do
11:         if $q$ is not in set $SQL$ then
12:             Add $q$ into $SQL$
13:         Append session ID $i$ to the set $AQ_q$ with $q$ as the key
14: for each distinct HTTP request $r$ in $REQ$ do
15:     for each distinct DB query $q$ in $SQL$ do
16:         Compare the set $AR_r$ with the set $AQ_q$
17:         if $AR_r = AQ_q$ and $Cardinality(AR_r) > t$ then
18:             Found a Deterministic mapping from $r$ to $q$
19:             Add $q$ into mapping model set $MS_r$ of $r$
20:             Mark $q$ in set $SQL$
21:         else
22:             Need more training sessions
23:             return False
24: for each DB query $q$ in $SQL$ do
25:     if $q$ is not marked then
26:         Add $q$ into set $NMR$
27: for each HTTP request $r$ in $REQ$ do
28:     if $r$ has no deterministic mapping model then
29:         Add $r$ into set $EQS$
30: return True
```

C. Testing for Static Websites

Once the normality model is generated, it can be employed for training and detection of abnormal behavior. During the testing phase, each session is compared to the normality model. We begin with each distinct web request in the session and, since each request will have only one mapping rule in the model, we simply compare the request with that rule. The testing phase algorithm is as follows:

1) If the rule for the request is Deterministic Mapping $r \to Q$ ($Q \neq \emptyset$), we test whether Q is a subset of a query set of the session. If so, this request is valid, and we mark the queries in Q. Otherwise, a violation is detected and considered to be abnormal, and the session will be marked as suspicious.
2) If the rule is Empty Query Set $r \to \emptyset$, then the request is not considered to be abnormal, and we do not mark any database queries. No intrusion will be reported.
3) For the remaining unmarked database queries, we check to see if they are in the set NMR. If so, we mark the query as such.
4) Any untested web request or unmarked database query is considered to be abnormal. If either exists within a session, then that session will be marked as suspicious.

In our implementation and experimenting of the static testing website, the mapping model contained the Deterministic Mappings and Empty Query Set patterns without the No Matched Request pattern. This is commonly the case for static websites. As expected, this is also demonstrated in our experiments in section V.

D. Modeling of Dynamic Patterns

In contrast to static web pages, dynamic web pages allow users to generate the same web query with different parameters. Additionally, dynamic pages often use POST rather than GET methods to commit user inputs. Based on the web server's application logic, different inputs would cause different database queries. For example, to post a comment to a blog article, the web server would first query the database to see the existing comments. If the user's comment differs from previous comments, then the web server would automatically generate a set of new queries to insert the new post into the back-end database. Otherwise, the web server would reject the input in order to prevent duplicated comments from being posted (i.e., no corresponding SQL query would be issued.) In such cases, even assigning the same parameter values would cause different set of queries, depending on the previous state of the website. Likewise, this non-deterministic mapping case (i.e., one-to-many mapping) happens even after we normalize all parameter values to extract the structures of the web requests and queries. Since the mapping can appear differently in different cases, it becomes difficult to identify all of the one-to-many mapping patterns for each web request. Moreover, when different operations occasionally overlap at their possible query set, it becomes even harder for us to extract the one-to-many mapping for each operation by comparing matched requests and queries across the sessions.

Since the algorithm for extracting mapping patterns in static pages no longer worked for the dynamic pages, we created another training method to build the model. First, we tried to categorize all of the potential single (atomic) operations on the web pages. For instance, the common possible operations for users on a blog website may include reading an article, posting a new article, leaving a comment, visiting the next page, etc. All of the operations that appear within one session are permutations of these operations. If we could build a mapping model for each of these basic operations, then we could compare web requests to determine the basic operations of the session and obtain the most likely set of queries mapped from these operations. If these single operation models could not cover all of the requests and queries in a session, then this would indicate a possible intrusion.

Interestingly, our blog website built for testing purposes shows that, by only modeling nine basic operations, it can cover most of the operations that appeared in the real captured traffic. For each operation (e.g., reading an article), we build the model as follows. In one session, we perform only a single read operation, and then we obtain the set of triggered database queries. Since we cannot ensure that each user perform only a single operation within each session in real traffic, we use a tool called Selenium [15] to separately generate training traffic for each operation. In each session, the tool performs only one basic operation. When we repeat the operation multiple times using the tool, we can easily substitute the different parameter values that we want to test (in this case, reading different articles). Finally, we obtain many sets of queries from one session and assemble them to obtain the set of all possible queries resulting from this single operation.

By placing each r_m, or the set of related requests R_m, in different sessions with many different possible inputs, we obtain as many candidate query sets $\{Q_n, Q_p, Q_q ...\}$ as possible. We then establish one operation mapping model $R_m \rightarrow Q_m$ ($Q_m = Q_n \cup Q_p \cup Q_q \cup ...$), wherein R_m is the set of the web requests for that single operation and Q_m includes the possible queries triggered by that operation. Notice that this mapping model includes both deterministic and non-deterministic mappings, and the set EQS is still used to hold static file requests. As we are unable to enumerate all the possible inputs of a single operation (particularly write type operations), the model may incur false positives.

E. Detection for Dynamic Websites

Once we build the separate single operation models, they can be used to detect abnormal sessions. In the testing phase, traffic captured in each session is compared with the model. We also iterate each distinct web request in the session. For each request, we determine all of the operation models that this request belongs to, since one request may now appear in several models. We then take the entire corresponding query sets in these models to form the set CQS. For the testing session i, the set of DB queries Q_i should be a subset of the CQS. Otherwise, we would find some unmatched queries. For the web requests in R_i, each should either match at least one request in the operation model or be in the set EQS. If any unmatched web request remains, this indicates that the session has violated the mapping model.

For example, consider the model of two single operations such as Reading an article and Writing an Article. The mapping models are $READ \rightarrow RQ$ and $WRITE \rightarrow WQ$, and we use them to test a given session i. For all the requests in the session, we then find that each of them either belongs to request set $READ$ or $WRITE$. (You can ignore set EQS here). This means that there are only two basic operations in the session, though they may appear as any of their permutations. Therefore, the query set Q_i should be a subset of $RQ \cup WQ$, which is CQS. Otherwise, queries exist in this session that do not belong to either of the operations, which is inconsistent with the web requests and indicates a possible intrusion. Similarly, if there are web requests in the session that belong to none of the operation models, then it either means that our models haven't covered this type of operation or that this is an abnormal web request. According to our algorithm, we will identify such sessions as suspicious so that we may have false positives in our detections. We discuss the false positive detection rate further in Section V.

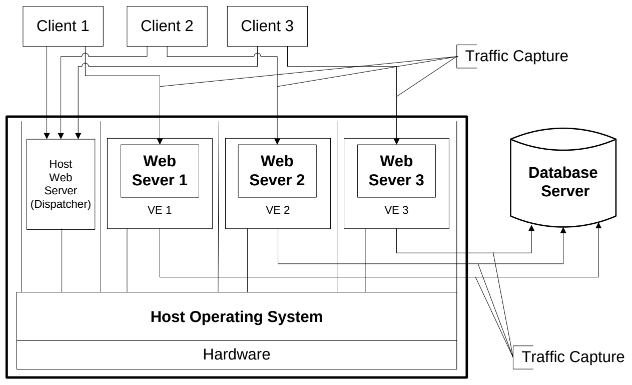

Fig. 9. The overall architecture of our prototype.

V. Performance Evaluation

We implemented a prototype of DoubleGuard using a web server with a back-end DB. We also set up two testing websites, one static and the other dynamic. To evaluate the detection results for our system, we analyzed four classes of attacks, as discussed in Section III, and measured the false positive rate for each of the two websites.

A. Implementation

In our prototype, we chose to assign each user session into a different container; however this was a design decision. For instance, we can assign a new container per each new IP address of the client. In our implementation, containers were recycled based on events or when sessions time out. We were able to use the same session tracking mechanisms as implemented by the Apache server (cookies, mod_usertrack, etc) because lightweight virtualization containers do not impose high memory and storage overhead. Thus, we could maintain a large number of parallel-running Apache instances similar to the Apache threads that the server would maintain in the scenario without containers. If a session timed out, the Apache instance was terminated along with its container. In our prototype implementation, we used a 60-minute timeout due to resource constraints of our test server. However, this was not a limitation and could be removed for a production environment where long-running processes are required. Figure 9 depicts the architecture and session assignment of our prototype, where the host web server works as a dispatcher.

Initially, we deployed a static testing website using the Joomla [10] Content Management System. In this static website, updates can only be made via the back-end management interface. This was deployed as part of our center website in production environment and served 52 unique web pages. For our analysis, we collected real traffic to this website for more than two weeks and obtained 1172 user sessions.

To test our system in a dynamic website scenario, we set up a dynamic Blog using the Wordpress [18] blogging software. In our deployment, site visitors were allowed to read, post, and comment on articles. All models for the received front-end and back-end traffic were generated using this data.

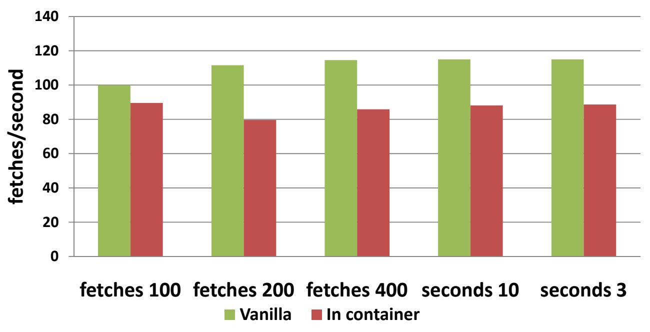

Fig. 10. Performance evaluation using http_load. The overhead is between 10.3% to 26.2%

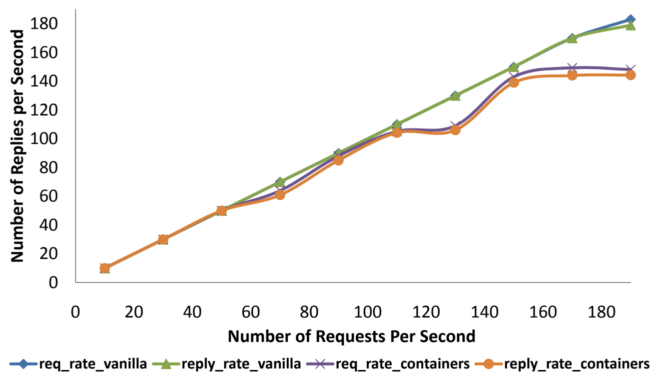

Fig. 11. Performance evaluation using autobench.

We discuss performance overhead, which is common for both static and dynamic models, in the following section. In our analysis, we did not take into consideration the potential for caching expensive requests to further reduce the end-to-end latency; this we left for future study.

B. Container Overhead

One of the primary concerns for a security system is its performance overhead in terms of latency. In our case, even though the containers can start within seconds, generating a container on-the-fly to serve a new session will increase the response time heavily. To alleviate this, we created a pool of web server containers for the forthcoming sessions akin to what Apache does with its threads. As sessions continued to grow, our system dynamically instantiated new containers. Upon completion of a session, we recycled these containers by reverting them to their initial clean states.

The overhead of the server with container architecture was measured using a machine with the following specifications: 4 cores 2.8GHz CPU, 8GB memory, 100MB/s NIC card, and CentOS 5.3 as the server OS. Our container template used Ubuntu 8.0.4 with Apache 2.2.8, and PHP 5.2.4. The size of the template was about 160MB, and Mysql was configured to run on the host machine. Our experiment showed that it takes only a few seconds for a container to start up, and our server can run up to 250 web server instances to form the pool of containers. Beyond this point, we observed a dramatic performance downgrade of the web server instances.

We evaluated the overhead of our container-based server against a vanilla web server. In order to measure throughput and response time, we used two web server benchmark tools: http_load [9] and autobench [4]. The testing website was the dynamic blog website, and both vanilla web server and the container-based web server connected to the same Mysql database server on the host machine. For the container-based server, we maintained a pool of 160 web server instances on the machine.

For the http_load evaluation, we used the rate of 5 (i.e., it emulated 5 concurrent users). We tested under the parameters of 100, 200, and 400 total fetches, as well as 3 and 10 seconds of fetches. For example, in the 100-fetches benchmark, http_load fetches the URLs as fast as it can 100 times. Similarly, in the 10 seconds benchmark, http_load fetches the URLs as fast as it can during the last 10 seconds. We picked 15 major URLs of the website and tested them against both servers. Figure 10 shows our experiment results.

The value of fetches per second in the http_load results is the most important indicator to reflect web server throughput performance. From the figure, we can observe that the overhead varied from 10.3% to 26.2%, under the full working load. When we put the parameters at 3 and 10 seconds, the overhead was about 23%.

We also tested using autobench, which is a Perl script wrapper around httperf [8]. It can automatically compare the performance of two websites. We tested demanding rate ranging from 10 to 190, which means that a series of tests started at 10 requests per second and increased by 20 requests per second until 190 requests per second were being requested; any responses that took longer than 10 seconds to arrive were counted as errors. We compared the actual requests rates and the replay rates for both servers.

Figure 11 shows that when the rate was less than 110 concurrent sessions per second, both servers could handle requests fairly well. Beyond that point, the rates in the container-based server showed a drop: for 150 sessions per second, the maximum overhead reflected in the reply rate was around 21% (rate of 130). Notice that 21% was the worst case scenario for this experiment, which is fairly similar to 26.2% in the http_load experiment. When the server was not overloaded, and for our server this was represented by a rate of less than 110 concurrent sessions per second, the performance overhead was negligible.

Figure 12 depicts the time needed for starting a container. As we opened 50 containers in a row, the average time was about 4.2 seconds.

C. Static website model in training phase

For the static website, we used the algorithm in Section IV-B to build the mapping model, and we found that only the Deterministic Mapping and the Empty Query Set Mapping patterns appear in the training sessions. We expected that the No Matched Request pattern would appear if the web application had a cron job that contacts back-end database

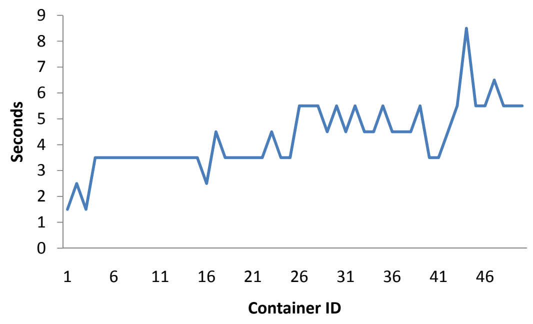

Fig. 12. Time for starting a new container.

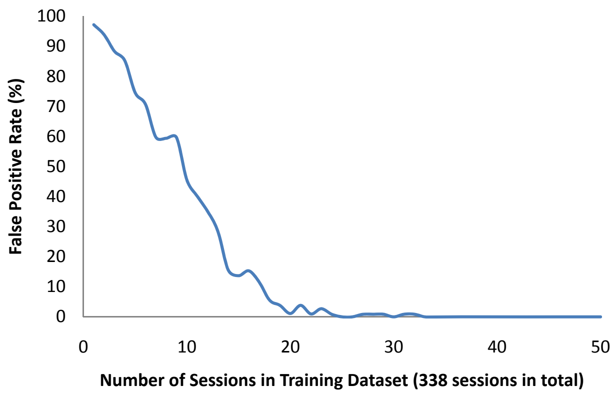

Fig. 13. False Positives vs Training Time in Static Website.

server; however, our testing website did not have such a cron job. We first collected 338 real user sessions for a training dataset before making the website public so that there was no attack during the training phase.

We used part of the sessions to train the model and all the remaining sessions to test it. For each number on the x-axis of Figure 13, we randomly picked the number of sessions from the overall training sessions to build the model using the algorithm, and we used the built model to test the remaining sessions. We repeated each number 20 times and obtained the average false positive rate (since there was no attack in the training dataset). Figure 13 shows the training process. As the number of sessions used to build the model increased, the false positive rate decreased (i.e., the model became more accurate). From the same figure, we can observe that after taking 35 sessions, the false positive rate decreased and stayed at 0. This implies that for our testing static website, 35 sessions for training would be sufficient to correctly build the entire model. Based on this training process accuracy graph, we can determine a proper time to stop the training.

D. Dynamic modeling detection rates

We also conducted model building experiments for the dynamic blog website. We obtained 329 real user traffic sessions from the blog under daily workloads. During this 7-day phase, we made our website available only to internal users to ensure that no attacks would occur. We then generated 20 attack traffic sessions mixed with these legitimate sessions,

Single Operation	No. of requests	No. of queries
Read an article	3	23
Post an article	10	49
Make Comment to an article	2	9
Visit next page	2	18
List articles by categories	3	19
List articles by posted months	3	16
Read RSS feed	1	2
Cron jobs	1	11
Visit by page number	2	18

TABLE I
SINGLE OPERATION MODELS EXAMPLE

and the mixed traffic was used for detection.

The model building for a dynamic website is different from that for a static one. We first manually listed 9 common operations of the website, which are presented in Table I. To build a model for each operation, we used the automatic tool Selenium [15] to generate traffic. In each session, we put only a single operation, which we iterated 50 times with different values in the parameters. Finally, as described in Section IV-D, we obtained separate models for each single operation. We then took the built models and tested them against all 349 user sessions to evaluate the detection performance. Figure 14 shows the ROC curves for the testing results. We built our models with different numbers of operations, and each point on the curves indicates a different *Threshold* value. The threshold value is defined as the number of HTTP requests or SQL queries in a session that are not matched with the normality model. We varied the threshold value from 0 to 30 during the detection. As the ROC curves depict, we could always achieve a 100% True Positive Rate when doing strict detection (threshold of 0) against attacks in our threat model. With a more accurate model, we can reach 100% sensitivity with a lower False Positive rate. The nature of False Positives comes from the fact that our manually extracted basic operations are not sufficient to cover all legitimate user behaviors. In figure 14, if we model 9 basic operations, we can reach 100% Sensitivity with 6% False Positive rate. In the case of 23 basic operations, we achieve the False Positive rate of 0.6%. This is part of the learning process illustrated in this paper, by extending the learning step to include more operations we can create a more robust model and further reduce the false positives.

E. Attack Detection

Once the model is built, it can be used to detect malicious sessions. For our static website testing, we used the production website, which has regular visits of around 50-100 sessions per day. We collected regular traffic for this production site, which totaled 1172 sessions.

For the testing phase, we used the attack tools listed in Table II to manually launch attacks against the testing website, and we mixed these attack sessions with the normal traffic obtained during the training phase. We used the sqlmap [16], which is an automatic tool that can generate SQL injection attacks. Nikto [13], a web server scanner tool that performs

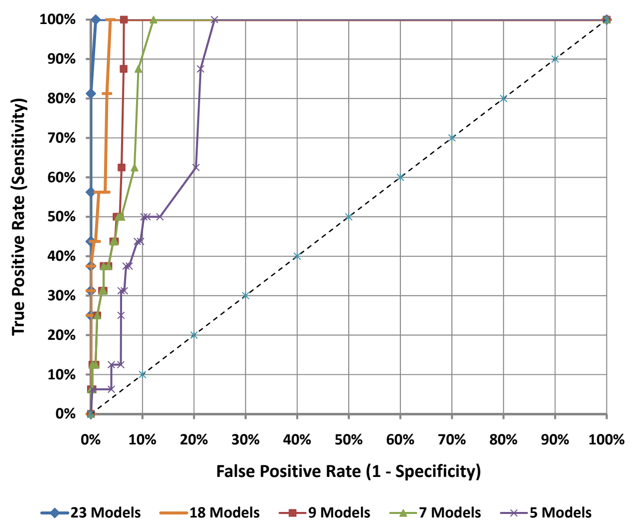

Fig. 14. ROC curves for dynamic models.

Operation	Snort	GSQL	DG
Privilege Escalation (WordPress Vul)	No	No	Yes
Web Server aimed attack (nikto)	Yes	No	Yes
SQL Injection (sqlmap)	No	Yes	Yes
DirectDB	No	No	Yes
linux/http/ddwrt_cgibin_exec∗	No	No	Yes
linux/http/linksys_apply_cgi∗	No	No	Yes
linux/http/piranha_passwd_exec∗	No	No	Yes
unix/webapp/oracle_vm_agent_utl∗	No	No	Yes
unix/webapp/php_include∗	Yes	No	Yes
unix/webapp/php_wordpress_lastpost∗	No	No	Yes
windows/http/altn_webadmin∗	No	No	Yes
windows/http/apache_modjk_overflow ∗	No	No	Yes
windows/http/oracle9i_xdb_pass∗	No	No	Yes
windows/http/maxdb_webdbm_database∗	No	No	Yes

TABLE II
DETECTION RESULTS FOR ATTACKS (GSQL STANDS FOR GREENSQL, AND DG STANDS FOR DOUBLEGUARD, ∗ INDICATES ATTACK USING METASPLOIT)

comprehensive tests, and Metasploit [12] were used to generate a number of web server-aimed http attacks (i.e., a hijack future session attack). We performed the same attacks on both DoubleGuard and a classic 3-tier architecture with a network IDS at the web server side and a database IDS at the database side. As there is no popular anomaly-based open source network IDS available, we used Snort [39] as the network IDS in front of the web server, and we used GreenSQL as the database IDS. For Snort IDS, we downloaded and enabled all of the default rules from its official website. We put GreenSQL into database firewall mode so that it would automatically whitelist all queries during the learning mode and block all unknown queries during the detection mode. Table II shows the experiment results where DoubleGuard was able to detect most of the attacks and there were 0 false positives in our static website testing.

Furthermore, we performed the same test for the dynamic blog website. In addition to the real traffic data that we captured for plotting the ROC curves, we also generated 1000 artificial traffic sessions using Selenium [15] and mixed the attack sessions together with all of them. As expected, the models for the dynamic website could also identify all of the same attack sessions as the static case. In the following section, we will discuss the experiment results in Table II in more detail based on these four attack scenarios in Section III-C.

1) Privilege Escalation Attack: For Privilege Escalation Attacks, according to our previous discussion, the attacker visits the website as a normal user aiming to compromise the web server process or exploit vulnerabilities to bypass authentication. At that point, the attacker issues a set of privileged (e.g., admin-level) DB queries to retrieve sensitive information. We log and process both legitimate web requests and database queries in the session traffic, but there are no mappings among them. IDSs working at either end can hardly detect this attack since the traffic they capture appears to be legitimate. However, DoubleGuard separates the traffic by sessions. If it is a user session, then the requests and queries should all belong to normal users and match structurally. Using the mapping model that we created during the training phase, DoubleGuard can capture the unmatched cases.

WordPress [18] 2.3.1 had a known privilege escalation vulnerability. As described in [19], there was a vulnerable check *"if (strpos($_SERVER['PHP_SELF'], 'wp-admin/') !== false) $this->is_admin = true;"* that used the PHP *strpos()* function to check whether the *$_SERVER['PHP_SELF']* global variable contained the string *"wp-admin/"*. If the *strpos()* function found the *"wp-admin/"* string within the *$_SERVER['PHP_SELF']* variable, it would return *TRUE*, which would result in the setting of the *"is_admin"* value to true. This ultimately granted the user administrative rights to certain portions of the web application. The vulnerable code was corrected to *"if (is_admin()) $this->is_admin = true;"* in a later version, which added a function to determine whether the user has administrative privilege. With the vulnerable code, an unauthorized user could input a forged URL like *"http://www.myblog.com/index.php/wp-admin/"* so as to set the value of variable *$this->is_admin* to *TRUE*. This would allow the unauthorized user to access future, draft, or pending posts that are administrator-level information.

According to our experimental results, DoubleGuard is able to identify this class of attacks because the captured administrative queries do not match any captured HTTP request. In addition, the crafted URLs also violate the mapping model of DoubleGuard, triggering an alert. In contrast, Snort fails to generate any alert upon this type of attack, as does GreenSQL. There are other privilege escalation vulnerabilities, such as the ones listed in NVD [2], [3], which prevent both a network IDS like Snort or a database IDS from detecting attacks against these vulnerabilities. However, by looking at the mapping relationship between web requests and database queries, DoubleGuard is effective at capturing such attacks.

2) Hijack Future Session Attack (Web Server aimed attack): Out of the four classes of attacks we discuss, session hijacking is the most common, as there are many examples that exploit the vulnerabilities of Apache, IIS, PHP, ASP, and cgi, to name

```
Legitimate HTTP request in training data:
GET:/sqlinjection.php?username=admin&password=123456
==========================================
Mapping Model:
GET:/sqlinjection.php?username=S&password=S
---------------Mapped SQL-----------------
SELECT * FROM users WHERE username='S' AND password='S'
...
```

Fig. 15. A trained mapping from web request to database queries

a few. Most of these attacks manipulate the HTTP requests to take over the web server. We first ran Nikto. As shown in our results, both Snort and DoubleGuard detected the malicious attempts from Nikto. As a second tool, we used Metasploit loaded with various HTTP based exploits. This time, Snort missed most of these attack attempts, which indicates that Snort rules do not have such signatures. However, DoubleGuard was able to detect these attack sessions. Here, we point out that most of these attacks are unsuccessful, and DoubleGuard captured these attacks mainly because of the abnormal HTTP requests. DoubleGuard can generate two classes of alerts. One class of alerts is generated by sessions whose traffic does not match the mapping model with abnormal database queries. The second class of alerts is triggered by sessions whose traffic violates the mapping model but only in regards to abnormal HTTP requests; there is no resulting database query. Most unsuccessful attacks, including 404 errors with no resulting database query, will trigger the second type of alerts. When the number of alerts becomes overwhelming, users can choose to filter the second type of alerts because it does not have any impact on the back-end database. Last, GreenSQL cannot detect these attacks.

DoubleGuard is not designed to detect attacks that exploit vulnerabilities of the input validation of HTTP requests. We argue that, if there is no DB query, this class of attacks cannot harm other sessions through the web server layer because of the isolation provided by the containers. However, as we pointed out in Section III-D, XSS cannot be detected nor mitigated by DoubleGuard since the session hijacking does not take place at the isolated web server layer.

3) Injection Attack: Here we describe how our approach can detect the SQL injection attacks. To illustrate with an example, we wrote a simple PHP login page that was vulnerable to SQL injection attack. As we used a legitimate username and password to successfully log in, we could include the HTTP request in the second line of Figure 15.

We normalized the value of ‘admin’ and ‘123456’, and repeated the legitimate login process a few times during the training phase. The mapping model that was generated is shown in Figure 15 (S stands for a string value), where the generalized HTTP request structure maps to the following SQL queries. After the training phase, we launched an SQL injection attack that is shown in Figure 16. Note that the attacker was not required to know the user name and password because he/she could use an arbitrary username the password 1’ or ’1=1, which would be evaluated as true.

The HTTP request from the SQL injection attacker would look like the second line in Figure 16. The parameter shown

```
GET:/sqlinjection.php?username=guess&
        password=1%27+or+%271%3D1
===========================================
Generalized caputred HTTP request:
GET:/sqlinjection.php?username=S&password=S
-------------------------------------------
Generalized caputred DB query:
SELECT * FROM users WHERE username='S' AND
        password='S' or 'S'
...
```

Fig. 16. The resulting queries of SQL injection attack.

in the box is the injected content. After normalizing all of the values in this HTTP request, we had the same HTTP request as the one in Figure 15. However, the database queries we received in Figure 16 (shown in box) do not match the deterministic mapping we obtained during our training phase.

In another experiment, we used sqlmap [16] to attack the websites. This tool tried out all possible SQL injection combinations as a URL and generated numerous abnormal queries that were detected by DoubleGuard. GreenSQL was also effective at detecting these attacks, which shows its ability to detect SQL injection attacks. Regarding Snort, although it is possible to write user-defined rules to detect SQL injection attack attempts, our experiments did not result in Snort reporting any SQL injection alerts.

SQL injection attacks can be mitigated by input validation. However, SQL injection can still be successful because attackers usually exploit the vulnerability of incorrect input validation implementation, often caused by inexperienced or careless programmers or imprecise input model definitions. We establish the mappings between HTTP requests and database queries, clearly defining which requests should trigger which queries. For an SQL injection attack to be successful, it must change the structure (or the semantics) of the query, which our approach can readily detect.

4) Direct DB attack: If any attacker launches this type of attack, it will easily be identified by our approach. First of all, according to our mapping model, DB queries will not have any matching web requests during this type of attack. On the other hand, as this traffic will not go through any containers, it will be captured as it appears to differ from the legitimate traffic that goes through the containers. In our experiments, we generated queries and sent them to the databases without using the web server containers. DoubleGuard readily captured these queries. Snort and GreenSQL did not report alerts for this attack.

VI. CONCLUSION

We presented an intrusion detection system that builds models of normal behavior for multi-tiered web applications from both front-end web (HTTP) requests and back-end database (SQL) queries. Unlike previous approaches that correlated or summarized alerts generated by independent IDSes, DoubleGuard forms a container-based IDS with multiple input streams to produce alerts. Such correlation of different data streams provides a better characterization of the system for anomaly detection because the intrusion sensor has a more precise normality model that detects a wider range of threats.

We achieved this by isolating the flow of information from each web server session with a lightweight virtualization. Furthermore, we quantified the detection accuracy of our approach when we attempted to model static and dynamic web requests with the back-end file system and database queries. For static websites, we built a well-correlated model, which our experiments proved to be effective at detecting different types of attacks. Moreover, we showed that this held true for dynamic requests where both retrieval of information and updates to the back-end database occur using the web-server front end. When we deployed our prototype on a system that employed Apache web server, a blog application and a MySQL back-end, DoubleGuard was able to identify a wide range of attacks with minimal false positives. As expected, the number of false positives depended on the size and coverage of the training sessions we used. Finally, for dynamic web applications, we reduced the false positives to 0.6%.

ACKNOWLEDGEMENTS

This work was supported in part by the United States Air Force Research Laboratory (AFRL) through contract FA8650-10-C-7024 and NSF grants CNS-TC 0915291, CNS 1118355, ETRI (B551179-09-01-00), and a research gift from Google Inc. Any opinions, findings, and conclusions expressed are those of the authors and do not represent the views of the funding sponsors.

REFERENCES

[1] http://www.sans.org/top-cyber-security-risks/.
[2] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-4332.
[3] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-4333.
[4] autobench. http://www.xenoclast.org/autobench/.
[5] Common vulnerabilities and exposures. http://www.cve.mitre.org/.
[6] Five common web application vulnerabilities. http://www.symantec.com/connect/articles/five-common-web-application-vulnerabilities.
[7] greensql. http://www.greensql.net/.
[8] httperf. http://www.hpl.hp.com/research/linux/httperf/.
[9] http_load. http://www.acme.com/software/http_load/.
[10] Joomla cms. http://www.joomla.org/.
[11] Linux-vserver. http://linux-vserver.org/.
[12] metasploit. http://www.metasploit.com/.
[13] nikto. http://cirt.net/nikto2.
[14] Openvz. http://wiki.openvz.org.
[15] Seleniumhq. http://seleniumhq.org/.
[16] sqlmap. http://sqlmap.sourceforge.net/.
[17] Virtuozzo containers. http://www.parallels.com/products/pvc45/.
[18] Wordpress. http://www.wordpress.org/.
[19] Wordpress bug. http://core.trac.wordpress.org/ticket/5487.
[20] C. Anley. Advanced sql injection in sql server applications. Technical report, Next Generation Security Software, Ltd, 2002.
[21] K. Bai, H. Wang, and P. Liu. Towards database firewalls. In *DBSec 2005*.
[22] B. I. A. Barry and H. A. Chan. Syntax, and semantics-based signature database for hybrid intrusion detection systems. *Security and Communication Networks*, 2(6), 2009.
[23] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th international conference on World wide web*, 2010.
[24] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns.
[25] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In *RAID 2007*.
[26] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8), 1999.
[27] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium*, 2010.
[28] Y. Hu and B. Panda. A data mining approach for database intrusion detection. In H. Haddad, A. Omicini, R. L. Wainwright, and L. M. Liebrock, editors, *SAC*. ACM, 2004.
[29] Y. Huang, A. Stavrou, A. K. Ghosh, and S. Jajodia. Efficiently tracking application interactions using lightweight virtualization. In *Proceedings of the 1st ACM workshop on Virtual machine security*, 2008.
[30] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium*, 2004.
[31] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10^{th} ACM Conference on Computer and Communication Security (CCS '03)*, Washington, DC, Oct. 2003. ACM Press.
[32] Lee, Low, and Wong. Learning fingerprints for a database intrusion detection system. In *ESORICS: European Symposium on Research in Computer Security*. LNCS, Springer-Verlag, 2002.
[33] Liang and Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *SIGSAC: 12th ACM Conference on Computer and Communications Security*, 2005.
[34] J. Newsome, B. Karp, and D. X. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2005.
[35] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical prevention of large-scale data leaks. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2009.
[36] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *RAID 2005*.
[37] S. Potter and J. Nieh. Apiary: Easy-to-use desktop application fault containment on commodity operating systems. In *USENIX 2010 Annual Technical Conference on Annual Technical Conference*.
[38] W. Robertson, F. Maggi, C. Kruegel, and G. Vigna. Effective Anomaly Detection with Scarce Training Data. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.
[39] M. Roesch. Snort, intrusion detection system. *http://www.snort.org*.
[40] A. Schulman. Top 10 database attacks. http://www.bcs.org/server.php?show=ConWebDoc.8852.
[41] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*. The Internet Society, 2009.
[42] A. Seleznyov and S. Puuronen. Anomaly intrusion detection systems: Handling temporal relations between events. In *RAID 1999*.
[43] Y. Shin, L. Williams, and T. Xie. SQLUnitgen: Test case generation for SQL injection detection. Technical report, Department of Computer Science, North Carolina State University, 2006.
[44] A. Srivastava, S. Sural, and A. K. Majumdar. Database intrusion detection using weighted sequence mining. *JCP*, 1(4), 2006.
[45] A. Stavrou, G. Cretu-Ciocarlie, M. Locasto, and S. Stolfo. Keep your friends close: the necessity for updating an anomaly sensor with legitimate environment changes. In *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence, 2009*.
[46] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *ACM SIGPLAN Notices*, 39(11), Nov. 2004.
[47] F. Valeur, G. Vigna, C. Krügel, and R. A. Kemmerer. A comprehensive approach to intrusion detection alert correlation. *IEEE Trans. Dependable Sec. Comput*, 1(3), 2004.
[48] T. Verwoerd and R. Hunt. Intrusion detection techniques and approaches. *Computer Communications*, 25(15), 2002.
[49] G. Vigna, W. K. Robertson, V. Kher, and R. A. Kemmerer. A stateful intrusion detection system for world-wide web servers. In *ACSAC 2003*. IEEE Computer Society.
[50] G. Vigna, F. Valeur, D. Balzarotti, W. K. Robertson, C. Kruegel, and E. Kirda. Reducing errors in the anomaly-based detection of web-based attacks through the combined analysis of web requests and SQL queries. *Journal of Computer Security*, 17(3):305–329, 2009.
[51] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS 2007*.
[52] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Symposium on Security and Privacy (SSP '01)*, May 2001.