International Journal of Innovative
Computing, Information and Control
Volume **16**, Number **1**, February **2020**

 ISSN 1349-4198
pp. 137–151

AN IMMUNITY-ENHANCING SECURITY MODULE FOR CLOUD SERVERS

Takeshi Okamoto

Department of Information Network and Communication
Kanagawa Institute of Technology
1030 Shimo-ogino, Atsugi, Kanagawa 243-0292, Japan
take4@nw.kanagawa-it.ac.jp

Received July 2019; revised November 2019

Abstract. *Cyberattacks on a vulnerability in a server application are threats to cloud servers. Advanced endpoint security techniques can detect and mitigate cyberattacks on known and unknown vulnerabilities, but their detection causes the server application to terminate, resulting in a denial of service. Intrusion detection systems with machine learning combine high accuracy and high speed of detection, but they require a wide variety of samples for learning. To solve these drawbacks, this paper proposes an immunity-enhancing module that adaptively acquires immunity to known and unknown cyberattacks without the need for prior learning of attack data. The module consists of innate and adaptive immune functions. The innate immune function detects known and unknown cyberattacks using advanced endpoint security techniques, whereas the adaptive immune function learns and detects the cyberattacks identified by the innate immune function using a gradient boosting classifier, thereby preventing a denial of service due to the innate immune function. This paper describes implementation of the module for a DNS server application. Its performance was evaluated by attacking vulnerabilities of CVE-2015-5477 and CVE-2016-2776. The module showed a detection accuracy of 99.94%, including a true negative rate of 100.00% and a true positive rate of 99.88%, and an overhead of 2.70%.*

Keywords: Cloud server, Intrusion detection system, Adaptive machine learning, DNS, Gradient boosting

1. **Introduction.** Advances in cloud computing technologies have motivated many organizations to move their on-premises servers to the cloud. The dominant characteristics of cloud servers are their availability, scalability, and ubiquity, reducing downtime, allowing on-demand scale-up/down of service resources, and permitting ubiquitous access, respectively. In addition, cloud providers offer security features for data protection, such as authentication, access control, and encryption, and some have optional features that enhance security from cyberattacks, such as content delivery networks (CDNs) and intrusion detection systems (IDSs) including web application firewalls (WAFs). Despite these security features, challenges remain in cybersecurity, including preventing cyberattacks on vulnerabilities of server applications. Such cyberattacks can be handled by IDSs, classified as network-based and host-based IDSs.

Network-based IDSs detect and prevent cyberattacks by monitoring byte sequences in network traffic. Most commercial IDSs, however, are based on signature-matching methods, which may have difficulties detecting unknown cyberattacks, such as zero-day attacks, especially cyberattacks on vulnerabilities in original server applications. In contrast, these

DOI: 10.24507/ijicic.16.01.137

IDSs are better able to detect cyberattacks on vulnerabilities in commercial-off-the-shelf (COTS) server applications.

Unknown cyberattacks may be detected by machine learning based IDSs, with state-of-the-art IDSs found to be highly accurate in detecting cyberattacks when coupled with high-speed classification and learning [1-4]. These systems, however, require a wide variety of samples for learning and they may have difficulty collecting samples of cyberattacks on original server applications. In contrast, some anomaly based IDSs, including machine learning-based IDSs, do not require any samples of attack data. An immunity-based anomaly detection system showed higher accuracy of detection than other anomaly detection methods, and at a low computational cost [5]. However, these immunity-based IDSs require large numbers of normal samples to build an identification model. Difficulties may be encountered in ensuring that all the normal samples are actually normal.

Host-based systems detect and mitigate known and unknown cyberattacks on vulnerabilities in an application by monitoring the behavior of the application process, such as system call sequences [6-8] and control flow integrity [9-14]. Upon detecting cyberattacks, these host-based systems cause the application to terminate, preventing these cyberattacks, but resulting in a denial of service.

Intrusion-tolerant systems tolerate cyberattacks on vulnerabilities in server applications, using different implementations of the same server specification [15-19], as all implementations may not be affected by the same vulnerabilities [20,21]. Intrusion-tolerant systems require considerable resources to run diverse server applications, drastically increasing their operating costs, including those involved in the setup and maintenance of all configurations for all server applications.

To enhance the resilience of a server application against cyberattacks on its vulnerabilities, we propose an immunity-enhancing module that satisfies the following requirements.

- The module requires no additional physical or virtual machines such as security appliances.
- The module requires only one sample of normal data in advance; i.e., the module requires no samples of attack data, including known and unknown attack data.
- The module adaptively detects and prevents known and unknown cyberattacks in real-time.
- The module autonomously recovers the service of the server application compromised by cyberattacks.

The immunity-enhancing module has two properties of self-adaptive software [22]: *self-protecting*, defined as the ability to detect attacks and recover from their effects; and *self-healing*, defined as the ability to detect and diagnose failures, and subsequently recover from them. Studies on an IDS with self-protecting features showed that they can adaptively detect unknown attacks by continuously learning them using machine learning [23,24]. However, they may cause erroneous learning due to false detections, thereby reducing the accuracy of detection. In contrast, the immunity-enhancing module can prevent erroneous learning by diagnosing the result of detection. A study on self-healing for cybersecurity proposed a software self-healing framework that detects vulnerable code in an application and then repairs it by transforming it to secure code based on public vulnerabilities data [25], while the immunity-enhancing module can recover the application without its source code and public vulnerabilities data.

Section 2 of this paper presents a design of the immunity-enhancing module, and Section 3 describes implementation of a prototype for Berkeley Internet name domain (BIND) provided by Internet systems consortium (ISC), the most commonly used server implementation of domain name system (DNS), as the DNS service is an essential component

of Internet infrastructure. Section 4 shows evaluations of the performance of the prototype, including its accuracy of detection and overhead. The accuracy of detection was evaluated by exploiting two vulnerabilities of the server application: CVE[1]-2015-5477 and CVE-2016-2776. The overhead of the module was evaluated by comparing round trip times between a client and a server with and without the module. Section 5 is the conclusion.

2. **Design of an Immunity-Enhancing Module.** An immunity-enhancing module allows a server application to adaptively acquire immunity to known and unknown cyberattacks without the need for prior learning of cyberattacks and to autonomously recover from the effects of cyberattacks. That is, the immunity-enhancing module provides a server application with high resilience against cyberattacks. The immunity-enhancing module is equipped with both innate and adaptive immune functions. These functions are incorporated into a server application by modifying the source code of that server application or by runtime injection into a process of that server application (Figure 1).

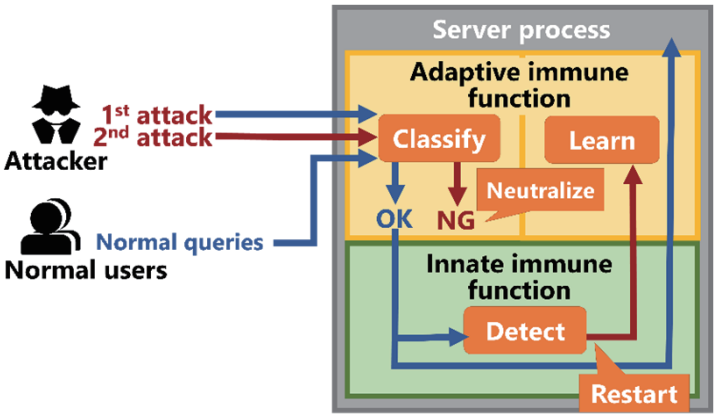


FIGURE 1. Overview of an immunity-enhancing module

Similar to innate immunity in biological systems, which non-specifically identifies and eliminates pathogens by recognizing molecules that are broadly shared by pathogens, the innate immune function of the server application non-specifically detects and prevents cyberattacks on known and unknown vulnerabilities by recognizing behaviors common to those cyberattacks. Subsequently, the innate immune function presents those cyberattacks to the adaptive immune function. Upon detecting a cyberattack, however, the innate immune function restarts the server application, because the server application process may lose its own normal execution control or may be unable to receive requests and send responses due to that cyberattack.

Similar to adaptive immunity in biological systems, which generates immunological memory of a specific antigen, resulting in faster and stronger elimination upon repeat exposure to pathogens bearing that antigen, the adaptive immune function learns a specific cyberattack upon initial detection by the innate immune function. This prevents the innate immune function from restarting the server application.

2.1. **Innate immune function.** The innate immune function has the ability to detect cyberattacks on known and unknown vulnerabilities in a server application that cause denial of service (DoS) and remote code execution (RCE). Upon detecting a cyberattack, the innate immune function notifies the adaptive immune function of the detection of that

[1]CVE: Common Vulnerabilities and Exposures

cyberattack. The innate immune function subsequently restarts the server application process by creating a new server application process and terminating the compromised server application process.

2.1.1. *Detection of DoS attacks.* A remote attacker may place a server application into a DoS state. This type of DoS attack can be classified into three types:

- Exploiting a vulnerability in a server application that causes DoS (e.g., a cyberattack on CVE-2016-2776 and CVE-2017-7651);
- Exhausting the internal resources of the server, such as memory, sockets, and threads (e.g., an SYN flood attack and a slow read attack);
- Exhausting the network bandwidth of the server by continuously sending a flood of requests or responses (e.g., a UDP-based amplification attack).

The first type of DoS attack can be detected by signature matching, response check, and assertion check methods.

Signature matching methods, such as Snort and YARA, look for patterns of byte sequences specific to cyberattacks in network traffic. These methods detect and prevent known attacks before a server application process loses its normal execution control, thereby sustaining the service of the server application without a need to restart it. Signature matching methods have difficulty detecting unknown attacks due to their lack of known signatures. These unknown attacks can be detected by response check and/or assertion check methods.

Response check methods look for the absence of a response, as DoS attacks force a server application process not to send a response. These methods, however, cannot prevent DoS attacks, because these attacks have been successfully completed at the time of their detection.

Assertion check methods validate the contents of a request received by a server application. These methods, however, cannot prevent DoS attacks, because assertion failure aborts the execution of the server application.

Upon detecting the first type of DoS attack, the response and assertion checks require that the server application be restarted to recover its service. To detect subsequent similar attacks without restarting, the innate immune function must identify a request used by the first attack and then pass the contents of that request to the adaptive immune function.

The second type of DoS attack is mitigated by security features, such as SYN cookies and firewalls, built into operating systems. These attacks, however, are not learned by the adaptive immune function, because these attacks are blocked by the operating system before a server application receives a request.

The third type of DoS attack cannot be prevented, because the innate immune function cannot stop attackers from sending a flood of requests or responses. This type of attack can be mitigated by an external service, such as CDN and IP anycast on a commercial cloud provider, or by a distributed DoS open threat signaling (DOTS) service [26] on Internet service providers.

2.1.2. *Detection of remote code execution attacks.* An attacker remotely executes malicious code by exploiting a vulnerability in a server process, compromising the most important properties of information security; i.e., confidentiality, integrity, and availability. The process of such an RCE attack passes through three stages:

1) Hijacking program control of an application process by exploiting a vulnerability in the application, such as stack buffer overflow;

2) If necessary, bypassing data execution prevention (DEP) by invoking a critical memory function for security, such as the `VirtualProtect` function on Windows and the `mprotect` function on Linux, via return-oriented programming (ROP);
3) Performing malicious operations, such as downloading and executing malware by executing a small piece of code, called "shellcode".

The first stage can be detected and prevented by a signature matching method (Section 2.1.1) and/or a vulnerability-based method. The latter method detects malicious modification of critical data, such as a return address on a stack memory. An RCE attack on a stack buffer overflow can be detected and prevented by stack canary [27,28] and SafeSEH [29]. They require compiling the source code of the application to enable their feature. An RCE attack on a heap buffer overflow can be mitigated by methods built into Windows [30].

The second stage can be detected and prevented using a control flow integrity (CFI) detection method [9], which permits only a legitimate control flow path of an application during execution. CFI detection methods can be classified into two granularity detection levels: fine-grained CFI [9] and coarse-grained CFI [10,11,13] detection methods.

The third stage can be detected and prevented using the signature matching method, the CFI detection method, or a behavior-based method. The latter method prevents behaviors specific to shellcode during execution by monitoring critical data structures, such as the PE header of NTDLL.DLL in an application process [31], or Windows APIs used in shellcode [32].

Upon detecting an RCE attack, all these methods, except for signature matching methods, raise an exception, resulting in the cessation of the server application. The innate immune function subsequently restarts the server application to recover its service. To detect subsequent similar attacks without restarting, the innate immune function identifies a request used by the first attack and then passes the contents of that request to the adaptive immune function.

2.2. **Adaptive immune function.** The adaptive immune function has the ability to classify and learn a request using a supervised machine learning method, such as support vector machines, neural networks, naïve Bayes, and random forests. Although supervised machine learning requires a wide variety of supervised data for learning before classification, the adaptive immune function requires only one normal request as supervised data, because additional supervised data are continuously supplied by the innate immune function while the server application is running. This continuous supply of supervised data is the key mechanism of the adaptive immune function, resulting in adaptive learning without the need for prior learning of attack requests, i.e., the more requests the adaptive immune function learns, the more it will improve the accuracy of detection.

Supervised data include both normal and attack requests. An attack request is supplied to a queue for attack requests just after the innate immune function detects a cyberattack, whereas a normal request is supplied to a queue for normal requests just after the server application sends a response corresponding to that normal request without error. The maximum sizes of the two queues are parameters of the immunity-enhancing module, as described in Section 4.3.

All the requests in the two queues are learned every time the innate immune function restarts the server application rather than every time a normal request is received, because the latter may impose a heavy load on the server application due to the large number of normal requests in normal times.

A request is classified just after being received by the server application (i.e., before it is detected by the innate immune function). If the request is classified as an attack,

it is discarded or neutralized by the adaptive immune function before it breaches the vulnerability (i.e., before it is detected by the innate immune function), thereby preventing the innate immune function from restarting the server application. If the request is classified as normal, it is passed to the server application.

3. **Prototype for ISC BIND.** The domain name system (DNS) is an essential component of Internet infrastructure. The application most widely used by the DNS server is Berkeley Internet name domain (BIND), provided by Internet systems consortium (ISC). Unfortunately, many critical vulnerabilities have been found in the application, with some, such as CVE-2016-2776 and CVE-2017-3145, causing denial of service. Such vulnerabilities may be found in the near future.

In this study, a prototype was designed for ISC BIND 9 to protect its vulnerability from cyberattacks. The prototype was implemented as a dynamic link library (DLL) module for Windows, IMMUNITY.DLL, which is equipped with both innate and adaptive immune functions (Figure 2). Upon creation of a BIND process, the module is automatically injected into the BIND process via a fix library, known as a shim, created by the Microsoft Application Compatibility Toolkit [33]. Once the DLL file is injected into the address space of the process, the `DllMain` function is called on the process. The `DllMain` function is responsible for API hooking, which allows injection of new features into Windows APIs by intercepting their calls. Thus, both the innate and adaptive immune functions are incorporated into Windows APIs used by the BIND process. The APIs are hooked using mhook [34], an inline hooking library for ×86 and ×64 platforms.

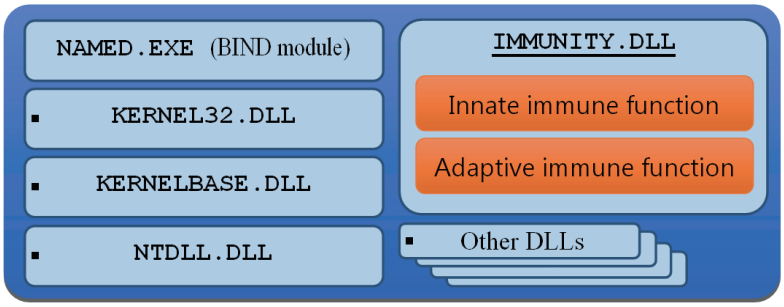


FIGURE 2. Layout of a BIND process

3.1. **Retrieval of a query and a response.** The prototype requires a query and a response to learn and classify a query. The BIND process receives a query and sends a response using "I/O completion ports", which provides a threading model for processing asynchronous I/O requests. Although the BIND process calls the `WSARecvFrom` function to receive a query, this function is unable to receive any query. Rather, the `WSARecvFrom` function sends an I/O request to a network device driver, with the I/O request subsequently added to an I/O completion queue. The BIND process retrieves a query from multiple receiving buffers using the `GetQueuedCompletionStatus` function instead of the `WSARecvFrom` function. The prototype therefore retrieves a query by hooking the `GetQueuedCompletionStatus` function.

The BIND process sends a response using the `WSASendTo` function. The second parameter of the `WSASendTo` function contains the contents of the response. The prototype therefore retrieves the response by hooking the `WSASendTo` function.

The contents of each query are stored in a query table, allowing their retrieval. The query table is indexed by the transaction ID (TID) of a query. A query corresponding to the specific response is retrieved from the query table by specifying the TID of the

response. This is based on the specification of the DNS protocol that the TID of a query is identical to the TID of the response to that query [35]. However, when the BIND process simultaneously receives several queries with the same TID, the contents of one query will be overwritten by those of another in the query table, causing erroneous learning by the adaptive immune function. The overwriting of a query is prevented by replacing its TID with a cryptographic hash-based TID derived from the contents of that query. This hash-based TID is truncated to the upper two bytes of the SHA256 hash value, as the size of each TID filed is two bytes. The original contents of a query are stored in the query table, the key to which is the hash-based TID of that query. The query with the hash-based TID is passed to the BIND process. When the BIND process sends a response, the hash-based TID of that response is restored to its original TID, which is retrieved from an original query corresponding to that response using the query table.

3.2. **Detection of DoS attacks.** The first type of DoS attack exploits a vulnerability in a server application (Section 2.1.1), preventing the server application from sending a response. This type of DoS attack can therefore be detected by checking for the absence of a response. However, a recursive request to DNS servers can result in the absence of responses due to failures of authoritative DNS servers. Thus, the innate immune function of the prototype may yield false positive results.

To prevent false positives, the innate immune function of the prototype uses assertion check, which allows the innate immune function to detect the DoS attack. DoS attacks on vulnerabilities in the BIND server application, such as CVE-2016-2776 and CVE-2016-9444, cause an assertion failure owing to assertion check, which raises an exception by calling the abort function. Because the exception can be regarded as an indicator of a DoS attack, the innate immune function detects DoS attacks by handling the exception. However, the innate immune function cannot identify a query used by the DoS attack, as no information about that query is present in the context of an exception handler.

To identify a query used by the DoS attack, the innate immune function of the prototype checks for the absence of a response using a thread pool timer by calling the `CreateThreadpoolTimer` function, which executes a thread after a specified time. The specified time represents the time-out period of a query, which was set to 10 sec. in the prototype. A thread pool timer is created every time the BIND process receives a query. After the specified time, the thread pool timer executes a notification thread with a hash-based TID of a query. If the above exception occurs, the exception handler suspends its own thread using the `Sleep` function, resulting in execution of the notification thread. The notification thread recognizes a query with the hashed-based TID as an attack. The attack query is retrieved from the query table and the adaptive immune function is notified, followed by restarting of the BIND process.

If the BIND process sends a response before the notification thread is executed, the innate immune function recognizes a query corresponding to that response as normal. The normal query is retrieved from the query table by specifying the hash-based TID of that response, followed by notification of the adaptive immune function. At the same time, the innate immune function deletes the thread pool timer corresponding to the normal query to prevent execution of the above notification thread.

3.3. **Detection of remote code execution attacks.** Methods of detection of cyber-attacks on vulnerabilities that cause RCE can be classified into three categories (Section 2.1.1). The prototype uses SecondDEP [32], classified in the third category (i.e., shellcode detection), because this method can detect more extensive and varied RCE attacks than the other methods. SecondDEP detects and prevents shellcode execution by checking Windows API calls. This is based on behavior, in which normal executable code calls

Windows APIs in executable code areas into which executable files have been mapped. In contrast, shellcode calls Windows APIs in data areas in which code execution is permitted.

SecondDEP cannot identify a query used by RCE attacks, as there is no information on that query in the context of shellcode detection. To identify a query used by RCE attacks upon their detection, the innate immune function of the prototype suspends its own thread using the Sleep function, resulting in execution of the notification thread. The notification thread identifies that query and notifies the adaptive immune function of it.

3.4. **Classifier for an adaptive immune function.** The adaptive immune function uses LightGBM [36], a gradient boosting algorithm, to learn and classify queries. Our previous work showed that the gradient boosting algorithm had the highest accuracy of detection [37]. In addition, LightGBM is suitable for the prototype, because both the prototype and LightGBM are implemented with C++, enabling the adaptive immune function to directly call LightGBM APIs for learning and classification. Other classifier algorithms are discussed in Section 4.2, as they affect not only the accuracy of detection but the speed of classification.

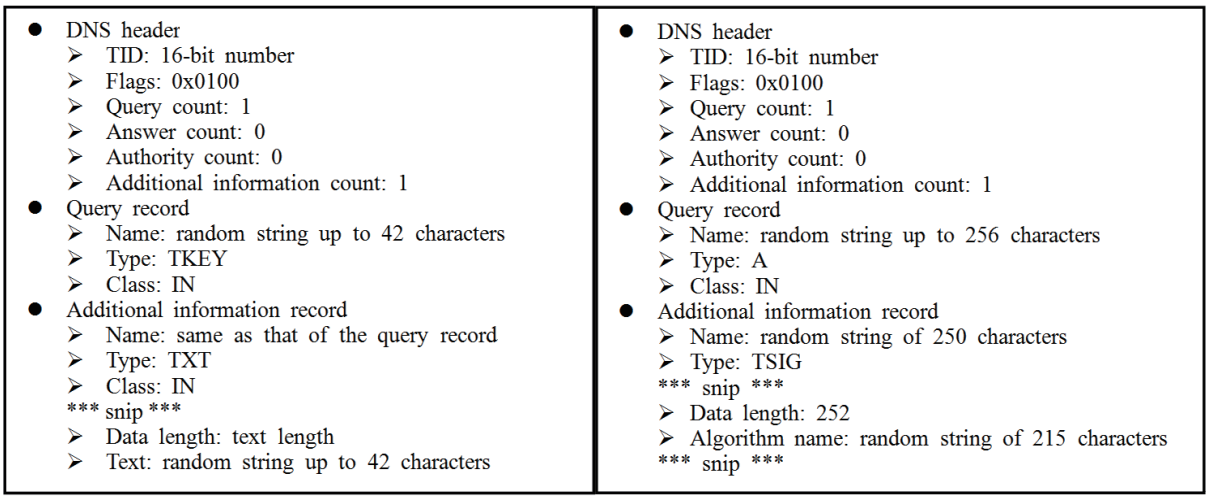
Classification of a query is performed by calling the `Booster::GetPredictRawData` member function of LightGBM. If the adaptive immune function classifies a query as an attack, it neutralizes that query by zero-filling its contents. If not, it passes that query to the BIND process. Queries are learned by repeatedly calling the `Booster::TrainOneIter` member function of LightGBM up to the maximum number of boosting iterations. All the parameters of LightGBM are set by default, except for three: `objective`, `min_data_in_leaf`, and `metric`. These parameters are set to "`binary`", "`2`", and "`binary, auc`", respectively.

In our previous prototypes [37,38], the fuzzy hash value [39] of a raw query was fed into the classifier. In this prototype, however, the byte sequence of a raw query is fed into the classifier to improve the accuracy of detection. The byte sequence is mapped to an N-dimensional vector, in which each element is initialized with a value of 0×100. This value indicates that there is no element at that position in the vector. If the length of the query is greater than the number of dimensions, the adaptive immune function discards the query by zero-filling it. The number of dimensions in the prototype is set at 1,000.

4. **Performance Tests.** The accuracy of detection and the overhead of the prototype were evaluated using ISC BIND 9.7.0, which has vulnerabilities that can be exploited using Metasploit Framework.

4.1. **Performance test data and scenario.** Each item of normal data is a DNS query containing one of eight resource records (A, AAAA, NS, SOA, CNAME, TXT, PTR, and MX), which are frequently queried to a cache server. In contrast, each item of attack data is a query that causes denial of service by exploiting either CVE-2015-5477 [40] or CVE-2016-2776 [41]. These vulnerabilities cause the BIND process to crash with an assertion failure. Figure 3 represents the contents of each attack query, based on auxiliary modules provided by Metasploit Framework [42,43]. All performance tests were conducted using the following three steps.

1) The process of BIND was created as a cache server and received 80 normal queries: 8 types $\times 10$ queries.
2) Cyberattacks on the above vulnerabilities caused denial of service, resulting in learning of the 80 normal queries and the attack query, followed by restarting of the BIND process.
3) The BIND process continuously received normal and attack queries.



(a) CVE-2015-5477 (b) CVE-2016-2776

FIGURE 3. Query data of attacks on (a) CVE-2015-5477 and (b) CVE-2016-2776

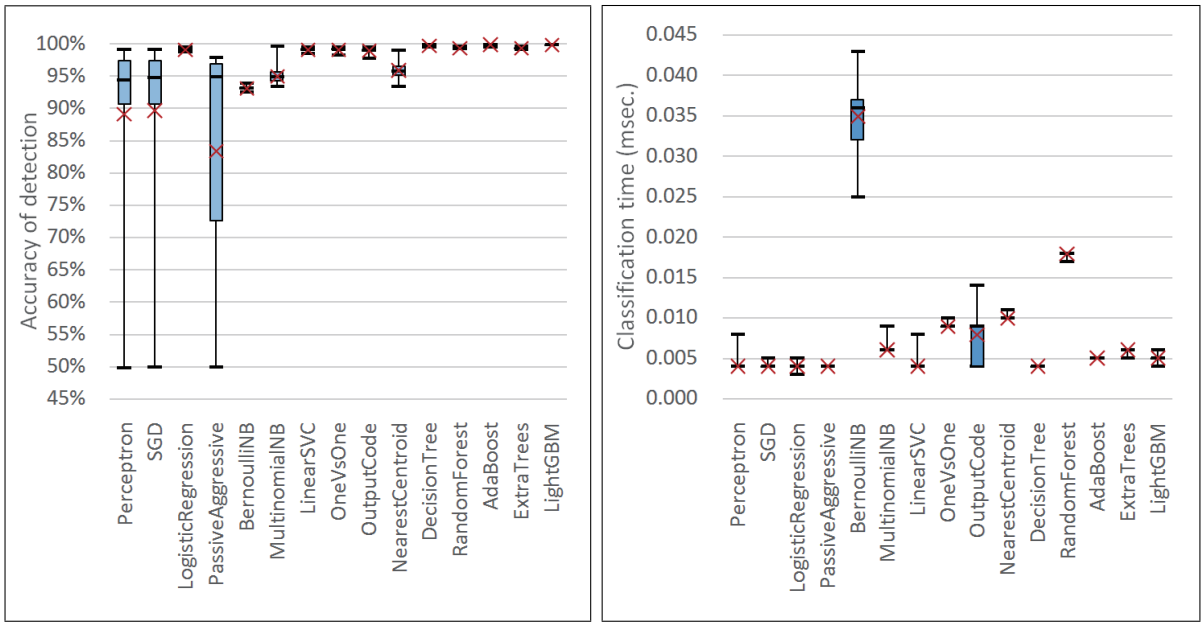
The accuracy of detection and the overhead of the prototype were evaluated during the third step. The test data from the third step consisted of 1,600 randomly generated queries. Half of these queries were normal, consisting of 100 each of the above eight resource records. The remaining half of these queries were attack queries, consisting of 400 each of the above two vulnerabilities. Each trial was performed by rearranging the order of transmission of queries for the test data.

4.2. **Comparison of classifier algorithms.** The algorithm of the classifier will affect the accuracy of detection and the speed of classification. Various algorithms were examined using scikit-learn, which provides a Python library for machine learning. Because the prototype used LightGBM, LightGBM was used instead of the gradient boosting algorithm of scikit-learn. All parameters of these algorithms were set by default. LightGBM parameters were the same as those described in Section 3.4 and were identical to the default parameters of the gradient boosting algorithm provided by scikit-learn. The algorithms were evaluated by simulating the performance tests described in Section 4.1. Table 1 shows the specifications of the simulation machine.

TABLE 1. Specifications of a simulation machine

CPU	AMD Opteron 6234 2.4 GHz (2 processors × 12 cores)
Memory	32 GB
Benchmark score [44] (Single thread score)	10,079 (914)

Figure 4 shows box-and-whisker plots of the accuracy of detection and the speed of classification of 100 trials for each algorithm, except for algorithms with low accuracy ($< 85\%$) or slow speed (time > 0.04 msec.). The five most accurate algorithms were tree-based, with LightGBM being the most accurate and the third-fastest. The speeds of the tree-based algorithms were almost the same, except for the random forests algorithm. LightGBM is widely-used in many winning solutions of machine learning competitions [45]. Therefore, LightGBM was the most suitable algorithm for the prototype.



(a) Accuracy of detection (b) Classification time

FIGURE 4. Comparison of 15 algorithms, each assessed over 100 trials (each red cross indicates a mean value)

4.3. **Size of learning queues.** We previously reported that the size of the learning queue affected the accuracy of detection [37,38]. To determine a suitable query size, the accuracy of detection and the time required for learning were evaluated for 1-800 queries using the simulating performance tests described in Section 4.1. The accuracy of detection of each was $\geq 99.91\%$ when the queue size was set at greater than three, suggesting that the prototype had to learn only three normal and three attack queries, much fewer than the 450 queries required in our previous prototype. This improvement was achieved by replacing the random forests algorithm with the gradient boosting algorithm. In addition, the time required by each for learning was nearly equal to 50 msec. over all queue sizes, suggesting that queue size had no effect on learning time.

4.4. **Accuracy of detection.** The accuracy of detection of the prototype using only the innate immune function was evaluated. All attacks were detected, with no false positives. However, the BIND process was restarted after every attack as there was no adaptive immune function. The restart resulted in loss of cache in the BIND process and temporal stop of the service. The adaptive immune function can play an important role in preventing restarting by adaptively learning attack queries supplied by the innate immune function.

The accuracy of detection of the prototype using both the innate and adaptive immune functions was evaluated, with queue size set at 80. Figure 5 shows line plots of the average true negative rate (TNR), true positive rate (TPR) and detection accuracy of 100 trials of 1-40 queries. The initial TPR was quite low, as the adaptive immune function learned only one query during the second step of the test scenario described in Section 4.1. Only the first 10 queries markedly increased the accuracy of detection with large fluctuations. The accuracy reached 99.9375% before the final query, number 1,600, was received. This result indicated that the prototype adaptively detected and prevented attacks on the BIND server application without the need for prior learning of attack queries.

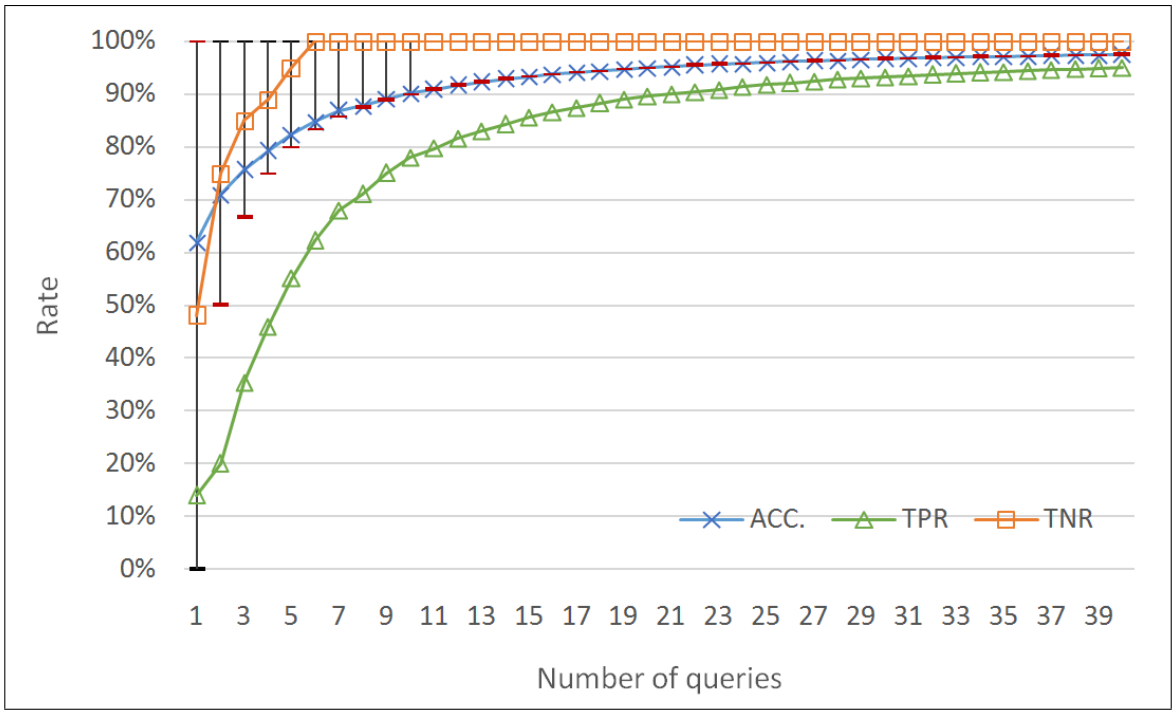


FIGURE 5. Average TNR, TPR, and accuracy as a function of the number of queries (each vertical black line indicates a range from minimum to maximum accuracy, and each red bar indicates median accuracy)

Table 2 shows the statistics for the final TNR, TPR, and detection accuracy of 100 trials. The prototype showed no false positives and only one false negative in each trial. At the final query in all trials, the queue for attack queries contained only two queries: one an attack on CVE-2015-5477 and the other an attack on CVE-2016-2776. This result indicated that only one learning of an attack enabled the adaptive immune function to detect all attacks on the same vulnerability as that first attack.

TABLE 2. Final TNR, TPR, and detection accuracy of 100 trials

Rate	Mean ± SD
Accuracy	99.9375 ± 0.0000
TNR	100.0000 ± 0.0000
TPR	99.8750 ± 0.0000

To investigate detection performance specific to each vulnerability, attacks on each vulnerability were evaluated, assuming that either CVE-2015-5477 or CVE-2016-2776 was fixed with security updates. Table 3 shows detection performances for CVE-2015-5477, CVE-2016-2776, and both together. The detection performances were almost the same, indicating that the prototype was not dependent on the type of vulnerability.

TABLE 3. TNR, TPR, and detection accuracy for cyberattacks on each vulnerability alone and the two together

Rate	Two CVEs	CVE-2015-5477	CVE-2016-2776
Accuracy	99.9375 ± 0.0000	99.9994 ± 0.0001	99.9375 ± 0.0000
TNR	100.0000 ± 0.0000	99.9988 ± 0.0001	100.0000 ± 0.0000
TPR	99.8750 ± 0.0000	99.9875 ± 0.0000	99.8750 ± 0.0000

4.5. **Overhead of the prototype.** The immunity-enhancing module increases the overhead of the BIND server application due to its having both innate and adaptive immune functions, such as response check and classification. The overhead of the prototype was evaluated by comparing round trip times (RTTs) of the BIND server with and without the prototype. The RTT was measured by dnsperf, a collection of DNS server performance testing tools [46]. The dnsperf command was executed on a DNS client connected to a BIND server via a layer 2 Gigabit Ethernet switch. Table 4 shows the specifications of the server machine.

Table 4. Specifications of a server machine

CPU	Intel Core i5-4300U 1.9 GHz (2 cores × 2 threads)
Memory	8 GB
Benchmark score [44] (Single thread score)	3,741 (1,569)

Table 5 shows the statistics of RTTs of 100,000 trials for the BIND process with and without the prototype. The overhead of the prototype was 2.70% (+0.004 msec.), much lower than the overhead of our previous prototype, which was 10.81% [38]. This reduction was achieved by replacing a random forests classifier (ranger [47]) with a gradient boosting classifier (LightGBM). Disabling the adaptive immune function alone showed that 80% of the overhead was shared by the innate immune function and 20% by the adaptive immune function. Further investigation of the overhead of the innate immune function indicated that this function spent too much time detecting DoS attacks using a thread pool timer, rather than detecting RCE attacks using SecondDEP. The overhead of the innate immune function may be reduced by improving detection of DoS attacks. The overhead of the adaptive immune function may also be reduced by tuning the parameters of LightGBM for classification, such as the maximum number of bins.

Table 5. RTTs of BIND 9 with/without a prototype (msec.)

Test cases	Mean ± SD
BIND 9 without a prototype	0.148 ± 0.002
BIND 9 with a prototype	0.152 ± 0.003

4.6. **Cyberattacks on an adaptive immune function.** Adaptive machine learning was shown to have potential risks of security violations of integrity, availability, and privacy [48]. Cyberattacks on adaptive machine learning can be taxonomically divided into two types.

The first type of attack can result in an attack query being misclassified as a normal query. This type of attack first sends various normal queries similar to the attack query, and then sends an attack. Fortunately, even if the adaptive immune function misclassifies an attack query as a normal query, it will be detected by the innate immune function, allowing the adaptive immune function to learn the attack query and to detect subsequent similar attacks.

The second type of attack can result in a normal query being misclassified as an attack query. This type of attack first sends an attack query very similar to a normal query before the adaptive immune function learns that normal query, resulting in denial of normal queries similar to that attack query. Evaluation of the risk of the second type of

misclassification resulted in false positives, but the more attack queries detected by the innate immune function resulted in better accuracy of detection by the adaptive immune function. Fortunately, no normal query was similar to attack queries of CVE-2015-5477 and CVE-2016-2776, because commonly used normal queries differ significantly from those attack queries.

5. **Conclusions.** This paper has proposed a design of an immunity-enhancing module that adaptively acquires immunity to known and unknown cyberattacks without learning attack data in advance. The module consists of innate and adaptive immune functions. The innate immune function detects known and unknown cyberattacks using SecondDEP, whereas the adaptive immune function learns and detects cyberattacks detected by the innate immune function using LightGBM.

To evaluate the performance of the module, a prototype for ISC BIND was implemented. The performance of the prototype was evaluated by attacking two vulnerabilities of ISC BIND, CVE-2015-5477 and CVE-2016-2776. The accuracy of detection was 99.94%, corresponding to a true negative rate of 100.0% and a true positive rate of 99.88%, and the overhead of the prototype was 2.70%. The overhead may be reduced by tuning the parameters of LightGBM.

Future work includes implementation and evaluation of the module for server applications of IoT, such as a message queueing telemetry transport (MQTT) broker, to enhance the protection of IoT services.

REFERENCES

[1] A. L. Buczak and E. Guven, A survey of data mining and machine learning methods for cyber security intrusion detection, *IEEE Commun. Surv. Tutor.*, vol.18, no.2, pp.1153-1176, 2016.

[2] W. C. Lin et al., CANN: An intrusion detection system based on combining cluster centers and nearest neighbors, *Knowledge-Based Systems*, vol.78, pp.13-21, 2017.

[3] R. A. R. Ashfaq et al., Fuzziness based semi-supervised learning approach for intrusion detection system, *Information Sciences*, vol.378, pp.484-497, 2017.

[4] S. Mohammadi et al., Cyber intrusion detection by combined feature selection algorithm, *Journal of Information Security and Applications*, vol.44, pp.80-88, 2019.

[5] M. Mohammadi et al., A fast anomaly detection system using probabilistic artificial immune algorithm capable of learning new attacks, *Evolutionary Intelligence*, vol.6, no.3, pp.135-156, 2014.

[6] S. A. Hofmeyr et al., Intrusion detection using sequences of system calls, *Journal of Computer Security*, vol.6, no.3, pp.151-180, 1998.

[7] W. Haider et al., Detecting anomalous behavior in cloud servers by nested arc hidden SEMI-Markov model with state summarization, *IEEE Trans. Big Data*, 2017.

[8] W. Khreich et al., Combining heterogeneous anomaly detectors for improved software security, *Journal of Systems and Software*, vol.137, pp.415-429, 2018.

[9] M. Abadi et al., Control-flow integrity principles, implementations, and applications, *ACM Trans. Information and System Security*, vol.13, no.1, pp.1-40, 2009.

[10] I. Fratrić, *ROPGuard: Runtime Prevention of Return-Oriented Programming Attacks*, Technical Report, https://www.ieee.hr/_download/repository/ Ivan_Fratric.pdf, 2012.

[11] V. Pappas et al. Transparent ROP exploit mitigation using indirect branch tracing, *Proc. of the 22nd USENIX Security Symposium*, pp.447-462, 2013.

[12] Y. Cheng et al., ROPecker: A generic and practical approach for defending against ROP attack, *Proc. of the 21st Annual Network and Distributed System Security Symposium, Internet Society*, pp.1-14, 2014.

[13] T. Okamoto, ROPGuard bypass prevention method using last branch recording facilities, *IPSJ Journal*, vol.57, no.9, pp.1933-1943, 2016.

[14] M. Rahman et al., Origin-sensitive control flow integrity, *Proc. of the 28th USENIX Security Symposium*, pp.195-211, 2019.

[15] A. Saidane et al., The design of a generic intrusion-tolerant architecture for web servers, *IEEE Trans. Dependable and Secure Computing*, vol.6, no.1, pp.45-58, 2009.

[16] P. Sousa et al., Highly available intrusion-tolerant services with proactive-reactive recovery, *IEEE Trans. Parallel and Distributed Systems*, vol.21, no.4, pp.452-465, 2010.
[17] H. Seondong et al., Designing and implementing a diversity policy for intrusion-tolerant systems, *IEICE Trans. Information and Systems*, vol.100, no.1, pp.118-129, 2017.
[18] T. Okamoto, F. Sano, I. Winarno, Y. Hata and Y. Ishida, Implementation and evaluation of an intrusion-resilient system for a DNS service, *International Journal of Innovative Computing, Information and Control*, vol.13, no.5, pp.1735-1749, 2017.
[19] J. Zheng et al., Security evaluation of a VM-based intrusion-tolerant system with pull-type patch management, *IEEE Proc. of the 19th International Symposium on High Assurance Systems Engineering (HASE)*, pp.156-163, 2019.
[20] I. Gashi et al., Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers, *IEEE Trans. Dependable and Secure Computing*, vol.4, no.4, pp.280-294, 2007.
[21] M. Garcia et al., OS diversity for intrusion tolerance: Myth or reality?, *The 41st International Conference on Dependable Systems and Networks*, pp.383-394, 2011.
[22] M. Salehie et al., Self-adaptive software: Landscape and research challenges, *ACM Trans. Autonomous and Adaptive Systems*, vol.4, no.2, pp.14:1-14:42, 2009.
[23] Y. Al-Nashif et al., Multi-level intrusion detection system (ML-IDS), *IEEE Proc. of 2008 International Conference on Autonomic Computing*, pp.131-140, 2008.
[24] M. Danforth, WCIS: A prototype for detecting zero-day attacks in web server requests, *Proc. of the 25th Large Installation System Administration Conference (USENIX LISA'11)*, 2011.
[25] A. ur R. Jaffar, M. Nadeem, M. Alenezi and Y. Javed, Using public vulnerabilities data to self-heal security issues in software systems, *ICIC Express Letters*, vol.13, no.7, pp.557-567, 2019.
[26] R. Dobbins et al., *Use Cases for DDoS Open Threat Signaling*, Internet draft, IETF, https://datatracker.ietf.org/doc/draft-ietf-DOTS-use-cases/, 2018.
[27] C. Cowan et al., Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks, *Proc. of the 7th USENIX Security Symposium*, pp.63-78, 1998.
[28] N. Burow et al., SoK: Shining light on shadow stacks, *Proc. of 2019 IEEE Symposium on Security and Privacy*, vol.1, pp.985-999, 2019.
[29] Microsoft Developer Network, */SAFESEH (Image has Safe Exception Handlers)*, https://msdn.microsoft.com/en-us/library/9a89h429.aspx, 2016.
[30] Microsoft SWI, *Software Defense: Mitigating Heap Corruption Vulnerabilities*, https://blogs.technet.microsoft.com/srd/2013/10/29/software-defense-mitigating-heap-corruption-vulnerabilities/, 2013.
[31] Microsoft, *Enhanced Mitigation Experience Toolkit 5.52 User Guide*, http://download.microsoft.com/download/B/F/B/BFBFDAB1-225C-4ECD-906F-C1DF61D7DB64/EMET%205.5.2%20User's%20Guide_J.pdf, 2016.
[32] T. Okamoto, SecondDEP: Resilient computing that prevents shellcode execution in cyber-attacks, *Procedia Computer Science*, vol.60, pp.691-669, 2015.
[33] Windows Dev Center, *Application Compatibility Toolkit (ACT)*, https://docs.microsoft.com/en-US/windows/desktop/win7appqual/application-compatibility-toolkit–act-, 2018.
[34] A. Marton, *mhook, an API Hooking Library*, v2.4, https://github.com/martona/mhook, 2014.
[35] P. Mockapetris, *RFC-1035 Domain Names – Implementation and Specification*, Network Working Group, 55, 1987.
[36] G. Ke et al., Lightgbm: A highly efficient gradient boosting decision tree, *Advances in Neural Information Processing Systems*, pp.3146-3154, 2017.
[37] M. Tarao, Toward an artificial immune server against cyber attacks: Enhancement of protection against DoS attacks, *Procedia Computer Science*, vol.96, pp.1137-1146, 2016.
[38] T. Okamoto, Implementation and evaluation of an immunity-enhancing module for ISC BIND9, *Procedia Computer Science*, vol.126, pp.1405-1414, 2018.
[39] J. Kornblum, Identifying almost identical files using context triggered piecewise hashing, *Digital Investigation*, vol.3, pp.91-97, 2006.
[40] M. McNally, CVE-2015-5477: An error in handling TKEY queries can cause named to exit with a REQUIRE assertion failure, ISC Knowledge Base, https://kb.isc.org/article/AA-01272, 2015.
[41] B. Conry, CVE-2016-2776: Assertion failure in buffer.c while building responses to a specifically constructed request, ISC Knowledge Base, https://kb.isc.org/article/AA-01419, 2016.
[42] J. Foote et al., *BIND TKEY Query Denial of Service*, https://github.com/rapid7/metasploit-framework/blob/master/modules/auxiliary/dos/dns/bind_tkey.rb, 2015.
[43] M. Rocha et al., *BIND TKEY Query Denial of Service*, https://github.com/rapid7/metasploit-framework/blob/master/modules/auxiliary/dos/dns/bind_tsig.rb, 2017.

[44] PassMark Software, *CPU Benchmarks*, https://www.cpubenchmark.net/cpu_list.php, 2019.
[45] Microsoft, *Machine Learning Challenge Winning Solutions, LightGBM/Examples, GitHub*, https://github.com/microsoft/LightGBM/blob/master/examples/README.md, 2019.
[46] DNS-OARC, *dnsperf*, https://www.dns-oarc.net/tools/dnsperf, 2019.
[47] M. N. Wright, Ranger: A fast implementation of random forests for high dimensional data in C++ and R, *Journal of Statistical Software*, vol.77, no.1, pp.1-17, 2017.
[48] L. Huang et al., Adversarial machine learning, *The 4th ACM Workshop on Security and Artificial Intelligence*, pp.43-58, 2011.