A Hybrid System for Run-time Protection inside Java Application

ABSTRACT

Web attacks pose a significant security threat to web applications. While Web Application Firewall (WAF) technology is widely used to block requests outside the application, Runtime Application self-protection (RASP) technology can offer more comprehensive protection by analyzing the program at a deeper level.

To this end, we have developed the HP-RASP (High-Performance RASP) system using the Java programming language. Our system provides effective in-application protection against various attacks, such as SQL Injections, XSS attacks, and Java Deserialization attacks. In addition, we have integrated the BERT model to enhance our system's protection against SQL requests.

Our experimental results demonstrate that the HP-RASP system has achieved excellent performance in terms of detection rate and resource consumption, particularly in preventing SQL Injection attacks. Moreover, our system outperforms other related open-source systems on most evaluation metrics.

KEYWORDS

RASP, BERT model, Software Security, Web application

1 INTRODUCTION

For decades, the risk of web attacks has posed a significant threat to network security. According to a report released by SANS in 2022, attacks targeting web applications rank in the top 5 of 'Top Action Vectors in System Intrusion Incidents' [19]. Despite the widespread deployment of security technology today, the report emphasizes that web security cannot be taken lightly. Among the many web attacks, SQL injection attacks and XSS (cross-site scripting) attacks are two of the most well-known and harmful methods [2, 14, 18, 39], making them the first-tier killers of web application security.

Web application firewall (WAF) is the most commonly used protection technique for web applications. Its main idea is to filter and monitor web traffic between applications and the Internet based on the traffic's external features. Over the last two decades, researchers have made significant strides with WAF techniques. WAFs can be divided into two categories based on time: signature-based WAFs and machine-learning-based WAFs [4, 7]. Signature-based firewalls were a widely accepted technique before the widespread use of machine learning. However, Razzaq et al. [35] summarized and proposed that web security protection based on signature alone is insufficient. In the last decade, with the popularity of machine learning techniques, many anomaly-based WAFs [1, 3, 24, 29, 30, 32, 42] and hybrid-based WAFs [33, 41] have been implemented, greatly advancing software protection technology.

While WAF technology has made significant contributions in identifying some types of malicious behavior, it has limitations, even with machine-learning and deep-learning techniques [5, 9, 45]. Specifically, WAFs do not perform well against never-seen-before malicious traffic and obfuscated traffic because they are positioned outside of a web application and only analyze external characteristics of the traffic, intercepting any requests deemed malicious without analyzing how the application processes them.

To address this issue, RASP technology can be used to monitor and block attacks in real-time and enable the application itself to have self-protection capabilities. Compared to WAF technology, RASP has no requirements for extensive testing and configuration and performs more stability. Gartner first proposed the idea of a built-in firewall in 2012, and since then, a considerable amount of related work has been conducted [6, 36, 37, 46].

In particular, machine learning can be applied to RASP to identify latent features of malicious attacks and block them selectively. By parsing a large number of payloads from the RASP system and applying natural language processing technologies, the accuracy of intercepting malicious requests can be improved. Prior research has investigated malicious payload detection using natural language processing, including deep learning models, graph convolutional networks, and the Transformer model. The BERT model, in particular, has been effective in identifying malicious payloads by fine-tuning it to specific tasks, such as classifying SQL statements [11, 16, 26, 27].

Overall, RASP technology combined with machine learning and natural language processing technologies holds great potential in improving the performance of certain security technologies and better protecting web applications from malicious attacks.

Therefore, we have designed and implemented a new hybrid system that combines the RASP technology and the BERT (Bidirectional Encoder Representations from Transformers) model together.[1] Our system can protect the applications against three types of attacks, which are SQL Injections, XSS attacks, and Java Deserialization attacks. In the system, run-time parameters and external features of the network requests are generated, and we can identify them using detection rules in our system with the parameters sent back from the monitors. In the RASP system, payloads are generated at the same time as rule detection, and we can feed them into the pre-trained BERT model and get the probability that they may act maliciously. The BERT model has been pre-trained and fine-tuned only for SQL Injections because we do not have a large dataset of XSS payloads now for training, the same as Java Deserialization attacks.

Here are our contributions:

i. We develop a comprehensive set of attack rules libraries and blacklists for the three types of attacks: SQL Injections, XSS attacks, and Java Deserialization attacks.

ii. Additionally, we design and implement the HP-RASP system, a High-Performance RASP System. This system includes a well-designed hook point mechanism and detection rules that allow for the accurate detection of potential attacks.

iii. To further enhance the system's accuracy, we design the Deep Semantic Understanding Module. We fine-tune the BERT model on the identification task of the malicious payloads and then embed it into our HP-RASP system.

iv. Finally, we compare the performance of our system with that of the OpenRASP system, taking detection rate, time consumption,

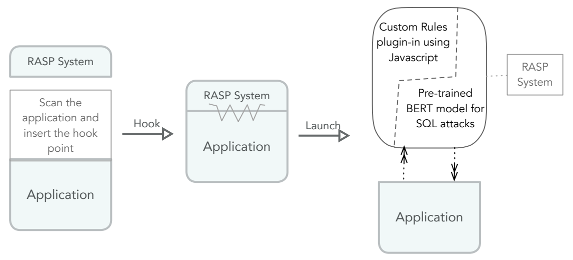
[1]The source code for our project is available on our code repository at : https://github.com/liuchengjie01/HPRASP

and system resource consumption into account. The results demonstrate the advantages of the HP-RASP system in terms of improved performance and detection accuracy.

Overall, this research contributes to the field of cybersecurity by providing a comprehensive set of attack rules libraries, and blacklists, as well as a highly effective system for detecting potential attacks.

2 STRUCTURE

We have implemented the system using Java and focus on the protection of Java web applications. To ensure that the system starts protection before the execution of the application, a very significant preceding step in the operation of the system is to scan the entire application and get the location of some key methods to construct appropriate hook points. After the hook points have been inserted, the two core detection modules will start up: the Detection Rules module and the BERT model module.



RASP System
Scan the application and insert the hook point
Application
Hook
RASP System
Application
Launch
Custom Rules plugin-in using Javascript
Pre-trained BERT model for SQL attacks
RASP System
Application

Figure 1: System Start-up Procedure: Insert the Hook Points and Launch the Two Main Detection Modules

Figure 1 shows the structure of the system. The Detection Rules module uses effective detection rules to make an identification using the parameters delivered by hook points. The Deep Semantic Understanding module will classify the requests into two categories to identify whether the request is malicious or not. Detailed explanations on how to find and insert hook points will be given in Section 3.1. The implementation of the two modules can be found below in Section 3.2 and Section 4.

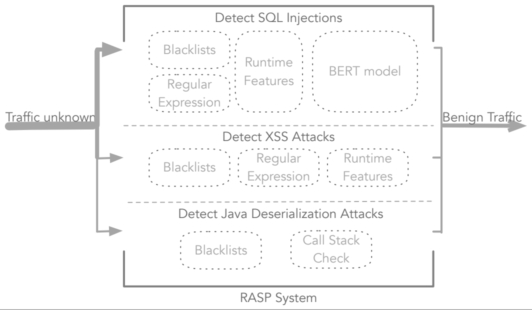


Figure 2: Overview of RASP System: Block malicious traffic and let benign traffic go away

After the completion of the initialization process, the system proceeds to safeguard the applications from potential web attacks. As depicted in Figure 2, the system comprises distinct sub-modules that are tailored to tackle three kinds of web attacks. To combat SQL Injections and XSS attacks, the system employs blacklists and regular expressions to detect malicious payloads and utilizes runtime features to analyze the behavior of the incoming requests. On the other hand, to counter Java Deserialization attacks, the system utilizes blacklists to match classes and analyze call stacks to identify hidden risk points.

The Deep Semantic Understanding module of the system is particularly noteworthy for its implementation of a fine-tuned BERT model. This model enhances the system's capability to understand the context of SQL statements at a deep semantic level, particularly those that are obfuscated or unknown. The module's robust design and efficient implementation contribute to the system's overall effectiveness in safeguarding against web attacks.

In conclusion, the system's multi-faceted approach to web attack prevention, coupled with the advanced techniques employed in its sub-modules, contribute to a comprehensive defense mechanism that provides a high level of protection to applications.

3 IMPLEMENTATION

In this section, we will give an overall description of how the RASP system works, and in the subsections, we will show some more details on some important and technical foundations of the system.

The system supports defense against three attack types: SQL Injection, XSS attack, and Java Deserialization attack. Corresponding hook mechanisms have been made for these three attack types. At the same time, for SQL Injections, we further filtered the received traffic through a pre-trained BERT model to achieve higher recognition accuracy and security.

In the initialization stage of the RASP system, as shown on the left of Figure 3, we modify the *premain* method, which performs the initialization and class loading operations and will be called before the *main* method. Then, we use the *java.lang.instrument* package to hook the key methods. In the *LoadClass* process, as shown on the right of Figure 3, the bytecode of the class will be handed to the module $Transformer$ module when a Java class is to be loaded. Then, the $Transformer$ module will decide whether this class is a class need to be hooked and send the class selected to ASM framework[17, 22]. The ASM framework will gradually analyze the bytecode of each method in the class according to the event-driven architecture. When the method we need to hook is triggered, we will insert the bytecode into the detection function at the beginning or end of the method. Finally, at the end of the startup stage of the system, we hand back the hooked bytecode of the class to $Transformer$ and the JVM will load it and run it as a normal Java application.

3.1 Hook Point

The hook points are the foundation of the RASP system, and we inject the bytecode into the protected application, whose basic idea is just similar to the AOP technology[40] in Java. The RASP system uses Java Instrumentation Interface[31] to perform AOP weaving before the bytecode runs. The biggest role of the Java

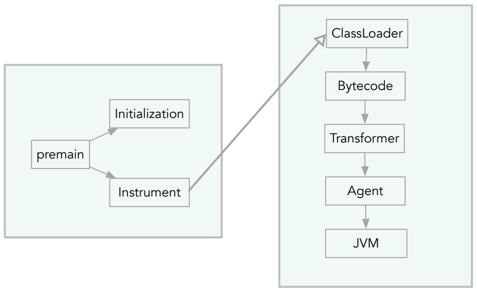


Figure 3: RASP System Startup Procedure

Instrumentation Interface is to dynamically change and operate class definitions to monitor applications running on the JVM.

We start the Instrumentation agent by specifying a specific jar file (containing the Instrumentation agent) through the ˘*javaagent* parameter when a normal Java application (Java class with *main* function) is running, and this operation makes it possible to run applications that have hook points without modifying the source code. The core code of the hook process is listed below in Algorithm 1. Firstly, we have to get the source destination of the static class which is in processing. Then, we check whether the methods we have read from the class are not null pointers. Finally, we use the *insertBefore* given by *CtBehavior* to insert bytecode at the beginning of the method body. The parameters mentioned in the algorithm come from the initial scanning of the classes in the application to be protected. So, in the hook process, what we have to do is to write the hook class and insert it into the entry function corresponding to different vulnerabilities through the *premain* method and its subsequent methods, and perform intrusion detection processing before the entry method runs.

Algorithm 1 The Core of Hook Point Insert Process

```
src = getInvokeStaticSrc(class, method, parameters);
methods = getMethod(class);
if methods != null && methods.size() > 0 then
    for method : methods do
        //Inserts at the beginning or end of the method body
        insert(method, src);
    end for
else
    Error: No method in this class
end if
```

In addition to some common hook rules, such as some methods that data flows through multiple times, and some entry methods, which we recorded in the risk class library in our project, we provided some special hook rules for three kinds of attacks to be detected according to the attack signature library and the risky class library besides the common rules like some common entry methods and execution methods. Different classes and method calls would be hooked related to the request types.

Hook points for SQL Injections To deal with several types of SQL Injections in Java web applications, we built libraries for the following risky classes, Table 1 shows the typical class related to SQL Injections. In this table, we can see that for the regular call path of SQL requests in Java web application, we have inserted hook points in every key node on the call path, such as classes like *SQLDriverManager*, *SQLConnection*, *SQLStatement* and *SQLResultSet*, which is important interfaces of various popular database software like MySql, Oracle, etc. What's more, we added *fileuploadHook* which supports SQL multi-part upload. The hook points are not only for SQL Injections when practical, and they sometimes make a contribution to others attacks' prevention.

Table 1: SQL Risk Classes (part)

Class	Description
com.mysql.jdbc.NonRegisteringDriver	ConnectionHook
com.mysql.jdbc.NonRegisteringDriver	DriverManagerHook
oracle.jdbc.driver.OracleStatement	StatementHook
org.postgresql.jdbc.PgPreparedStatement	StatementHook
...	...
com.ibm.db2.jcc.am	SQLResultSetHook
com.mysql.jdbc.ConnectionImpl	ConnectionPreparedHook
org.hsqldb.jdbc.JDBCConnection	ConnectionPreparedHook
org.apache.fileupload.FileUploadBase	MultipleHook

Hook points for XSS attacks Special hooks were made for the server and app front-end to ensure that the system can obtain sufficient parameters that can prove that the request has XSS attacks or not. Table 2 shows the typical class related to XSS attacks. *BESResponseBodyHook* was used to block malicious XSS requests by analyzing the parameters from the front end of the application, and *ServerRequestHook* focused on the server part, which generates the features of the input scripts and identifies them. What's more, we build some special hook points for some famous frameworks such as Catalina and Spring.

Table 2: XSS Risk Classes (part)

Class	Description
com.bes.OutputBuffer	BESResponseBodyHook
io.undertow.ServletRequestContext	ServerRequestHook
io.undertow.AttachmentKey	ServerRequestHook
org.apache.coyote.Response	ServerResponseBodyHook
...	...
org.apache.catalina.connector.Request	ServerParamHook
apache.catalina.FilterChain	ServerRequestEndHook
apache.catalina.CoyoteAdapter	ServerPreRequestHook
org.springframework.RequestWrapper	SpringHook

Hook points for Java Deserialization attacks To deal with this kind of attack, we implement two kinds of hook points: native Java Deserialization class hook and FastJSON hook. We focused on the *java.io.ObjectInputStream* class, which deserializes primitive data and objects previously written using an ObjectOutputStream and is used to recover those objects previously serialized. Most of the deserialized data can be obtained from this hook point. The FastJSON hook is relatively special. It is a Java library that can be used to convert Java Objects into their JSON representation, which is widely used. When we want to monitor the data flow of the application building a process, we insert the hook point *java.lang.ProcessImpl*, which is the function related to *process.init*, and is often used by RCE attacks.

Apart from the three types of attacks, We construct the special hook points for three different operating systems: Windows, Linux, and macOS.

Overall, The bytecode is modified through the java agent and ASM framework, which can achieve the purpose of inserting the hook point. After the hook point insert process is completed, the RASP startup process can be carried out.

3.2 Program Semantic Feature Detection Module

In this section, we will show you how we deal with these three specific types of attacks in the RASP system after all the hook points are inserted. The Detection rules module plays a linking role in the whole RASP system. It receives the data hook points sent, analyzes them, and blocks the requests which have obvious malicious features. Similarly, through feature analysis and simple payload analysis, we get requests that have very little possibility of malicious behavior and help them bypass the detection of the following BERT model to reduce the overall burden on the system. After the traffic passes through this module, only those requests that may have malicious behavior have to be further confirmed and diverted through BERT model detection.

We implement a Javascript plugin, which can be seen as a filter to process the parameters hook points sent and use JSON to match some malicious features. Here are some representative examples of the rules.

Detection rules for SQL Injections The following rules are constructed to identify the SQL requests. Firstly, we simply scan the SQL payload and check its length and syntax. The lengths of SQL statements should exceed 8 bytes, and they should contain keywords unique to SQL statements like *SELECT*, *FROM*, etc. The most important thing is to make sure whether the contents of the statements cause the logic of the SQL statement to change. Secondly, we pay attention to the features of the request. Many features like whether to execute multiple statements, whether it contains hexadecimal strings, whether it has version number comments, the frequency of sensitive methods, whether it has read or write files' operation, and whether it has $information\_schema$ related read operations have been taken into consideration. We will comprehensively evaluate these features, and give a score to affect the final response: benign, malicious, or not sure. Finally, we focus on the status of the database by monitoring the $error\_code$ of the database the application used, and it helps us better backtrack past security threats.

Detection rules for XSS attacks Same as the SQL's rules, we first scan the payloads input from outside. The length of user input may not exceed 15 bytes, for excessively long XSS payloads can be a sign of attack. Then, We use some regular expressions which were carefully crafted from past attacks to match the requests. For example, we have a regex '<![\\to -\\[A-Za-z]|<([A-Za-z]1,12)[\\/>\\x00-\\x20]', which is used for matching the payload for scanning. When the length of the user input exceeds 15, and the number of parameters collected from hook points matching the regex exceeds 10, it is judged as a scanning attack and is directly intercepted. XSS attacks are mainly divided into Reflected XSS attacks and Stored XSS attacks. The above ideas are aimed at Reflected XSS attacks, and we implemented some special rules and regex for different databases for Stored XSS attacks. Some dangerous payloads capable of permanently modifying stored data are identified using some SQL related hook points, such as *SQLStatementHook* and *SqlConnectionPreparedHook*, for the reason that it must execute some operations related to databases. The existence of stored XSS attacks can be judged through the assistance of conventional XSS hook points and some SQL hook points, which is an untapped territory in related open-source tools.

Detection rules for Java Deserialization attacks We get the call stack of the deserialized class through the hook points and compare the methods in the call stack with the blacklist we collected. This rule helps us determine whether the request contains a vulnerable component. If the methods in the blacklist reach a certain proportion and have a certain relationship, then we will consider this request to be malicious.

3.3 Deep Semantic Understanding Module

Considering that the detection of rules above stays at the level of keyword matching and data statistics, it is impossible for it to understand the payloads' context semantics well. So, we use the BERT4 model to extract and classify the semantic features of the payload, such as some malicious parts and their variants.

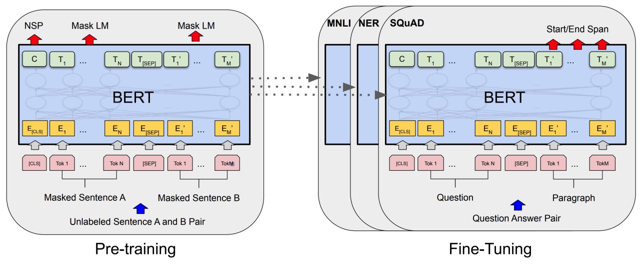


Figure 4: Overall pre-training and fine-tuning procedures for BERT.[11]

Due to the limitations of datasets, it is difficult for us to make a convincing dataset for the payloads of java Deserialization attacks and XSS attacks. So, we only made the relevant design for SQL statement payloads using the model.

The training dataset for the BERT model comes from the SQLi dataset[38] in the Kaggle dataset. The dataset contains a total of about 30,000 SQL statements, of which about 62 percent are normal data, marked as benign, and the others are marked as malicious. We divided the statements (about 90 percent of the whole dataset, the rest are used for system testing) into a training set and a validation set at a ratio of 8:1, then we compared several parameter combinations and selected the best-performing model.

There are two main problems to be solved when we tried to use the BERT model. Firstly, SQL statements, which have their own unique grammatical structure and expression, are different from natural languages. We need to pre-train the model to adapt to the syntax of the SQL statement. Secondly, the malicious part commonly does not take up a lot, so we need better use of the attention mechanism in the model to find malicious behavior in the payloads captured. The first problem needs a large amount of corpus which we can not afford now, so we just focus on the second problem and will research the former one in the future.

To address these two major problems, we have made the following efforts: Firstly, we carefully studied the contextual characteristics of the payloads and selected the $bert - base - uncased$ model[10] as a benchmark for fine-tuning.

Secondly, we use the SQL statement dataset to fine-tune the pre-trained model. There are several steps: i) extract the sentence features of the training data, ii) convert the payloads into high-dimensional vectors, iii) Then Mean-pooling is performed on the feature vectors, iv) connect the pooled vector matrix to the Fully Connected Layer, the length of the obtained vectors is the length of the labels at this time, v) normalize the vectors to obtain the probability distribution of each label, vi) the vector with the highest probability is selected as a classification result.

Thirdly, we try the model conversion. The saved model is in PyTorch format, which needs to be converted to an ONNX model format. While effectively improving the RASP's accuracy rate, the forward propagation of the network in the ONNX format does not bring a large burden on response time.

Finally, we use the result to further identify whether the request is malicious.

4 EXPERIMENT

We deploy the RASP system on a Ubuntu Server which has 2 vCPUs. In the experiment, we start a process to run the script sending the request and read the output result of the system in JSON format. The environments of the experiment are listed below:

Table 3: Experimental Environment Configuration

System/Tool	Description	Version/Config
Ubuntu	RASP Environment	16.04 LTS
CPU	RASP Environment	2 vCPUs, 2.9GHz
Java	Programming language	1.8
Tomcat	Simulate requests	5.8
Python	Build BERT model	3.7
PyTorch	Build BERT model	1.12

4.1 Detection Rate Evaluation

In this section, we will compare the detection rate (Precision Score, Recall Score, and F1-Score) with the OpenRASP system for SQL Injections, and then compare the experiment results for XSS results and Java Deserialization attacks.

For SQL Injections, we write scripts and test the system with the data in the SQLi dataset, and we finally have used 3060 pieces of samples as the test data of the system for SQL Injections, 1939 among them are benign, and others are malicious. We deploy a tomcat server and use it to send benign and malicious requests by running scripts.

Table 4: Comparison among Different Solutions (SQL): OpenRASP, HP-RASP, and HP-RASP with BERT model

Solution	Precision Score	Recall Score	F1-score
OpenRASP	0.443	0.979	0.610
HP-RASP	1.000	0.572	0.728
HP-RASP with BERT	0.819	0.983	0.893

Table 5: Comparison among Different Solutions (XSS): OpenRASP and HP-RASP

Solution	Total number attacks	Blocked attacks (Reflected)
OpenRASP	6606	6408
HP-RASP	6606	6598
Solution	Number of attacks	Blocked attacks (Stored)
OpenRASP	6606	NaN
HP-RASP	6606	6598

To evaluate the results more scientifically, we use evaluation metrics commonly used in machine learning. By the way, We define the **Precision Score** as how many of all predicted malicious samples are malicious samples and define the **Recall Score** as the proportion of how many malicious samples can be predicted among all malicious samples. The **F1-score** is the harmonic mean of the previous two results. Assuming that the system regards letting go of benign requests as equally important as blocking malicious requests, then the F1-score can be used as an important number to evaluate the interception effectiveness of the system.

Table 4 shows the result of our experiments on SQL Injections. We can see that the HP-RASP with the BERT model had the best overall interception accuracy because it had the highest F1 score as shown in the table above. Compared with the model, The OpenRASP system judges too many benign samples as malicious samples, so its Precision Score is not satisfied. On the other hand, the HP-RASP without the deep-learning model has disadvantages in finding malicious behavior in the requests.

The experiment result proves that the model has excellent comprehensive performances against SQL Injections, and the addition of the BERT model can effectively increase the accuracy of malicious request detection.

For XSS attacks, we have used the $xss - payload - list$ dataset in the payloadbox repository in Github. This dataset has more than 6000 pieces of XSS payloads including Reflected XSS attacks and Stored XSS attacks. The payloads of XSS attacks aim to make the javascript engine execute illegal statements, for example, payload '<html onMouseLeave html onMouseLeave="javascript:javascript:al -ert(1)"></html onMouseLeave>' tries to execute the pop-up command on the web page. There are several types of payloads in the dataset, such as payloads like splicing, creating anonymous functions, creating pseudo-protocols, etc.

Compared with the OpenRASP system, which only implements the detection rules for the Reflected XSS attacks, we successfully implement the detection mechanism for Stored XSS attacks. The experiment results are listed in Table 5.

We can see that compared with the prototype OpenRASP system, the HP-RASP system not only implemented stored XSS attacks but also has a preliminary improvement in the detection rate.

For Java Deserialization attacks, we used the ysoserial[12] tool to generate datasets. Ysoserial is a collection of utilities and property-oriented programming "gadget chains" discovered in common java libraries. For example, it contains some payloads targeting some certificate versions of widely used frameworks, such as commons-collections4:4.0, json-lib:jar:jdk15:2., spring-core:4.1.4, spring-beans:4.1.4, slf4j-api:1.6.4, javax.servlet-api:3.1.0, etc. A certain payload contains the bytecode of the classes. Figure 5 shows the details of the example payload.

```
0000000: aced 0005 7372 0032 7375 6e2e 7265 666c  ....sr.2sun.refl
0000010: 6563 742e 616e 6e6f 7461 7469 6f6e 2e41  ect.annotation.A
0000020: 6e6e 6f74 6174 696f 6e49 6e76 6f63 6174  nnotationInvocat
...
0000550: 7672 0012 6a61 7661 2e6c 616e 672e 4f76  vr..java.lang.Ov
0000560: 6572 7269 6465 0000 0000 0000 0000 0000  erride..........
0000570: 0078 7071 007e 003a                      .xpq.~.:
```

Figure 5: Bytecode of the Java Deserialization Payload

We have verified the effectiveness of the Deserialization part through simulation. The Recall Scores of both the OpenRASP system and the HP-RASP system are **0.8462**, which shows that the system can effectively intercept Java Deserialization attacks.

4.2 Time Consumption

Since the RASP system will inevitably cause some time loss, the time consumption index is also important in engineering. We defined several evaluation parameters to evaluate the time consumption in various ways. We send 20000 requests in 200 threads for every single solution's experiment. Table 6 shows the experiment of time consumption. The index $90\%line$ means that the response time of 90% requests achieved, only 10% of requests' response time exceeded the number. The definition of the 95% line is the same.

In this table, we can see that the Throughout Capacity among the solution are basically the same, which means the deployment of the system had little impact on the operation of the application. Compared with the OpenRASP system, the HP-RASP system had better performance and can respond to requests outside more quickly. Compared with the time usage of the raw application, 90% of requests' additional response time does not exceed 20% of the original response time, which is undoubtedly acceptable. It can also be seen from the table that when the BERT model is added, the additional response time does not change much. Overall, the system we implemented performed well in terms of time consumption.

4.3 System Resource Consumption

Finally, we use scripts to monitor the resource usage of each solution in the face of benign and malicious samples at runtime. The detailed results (Figure 7 and Figure 6)are listed below.

We can see from the results that the Network IO Metrics of the system on each solution were comparable when faced with benign and malicious requests, which shows that the HP-RASP system's sending and receiving are stable in face of diverse external requests. The two figures named CPU Metrics show that our system has significantly decreased the consumption of computing resources compared with the OpenRASP system, and it has little impact on the raw application. In terms of system storage resource consumption, a certain amount of memory resources are occupied by the BERT model, which is predictable. What's more, with some exceptions, the disk resource consumption of different solutions is in a stable range. Overall, the HP-RASP system performs well in resource consumption experiments, which has a controllable increase over the raw application and has a greater improvement compared to the prototype system.

5 LIMITATION

Applicability Incorporating the RASP system and implementing real-time monitoring, analysis, and judgment entail the allocation of additional system resources, a factor that cannot be overlooked. Furthermore, the deep learning model employed for identifying potentially malicious attacks in SQL statement payloads may introduce delays in application runtime. Thus, to balance the demands of security and time efficiency, Java software that meets exceptionally high-security standards is the most appropriate choice for leveraging this technology for robust security protection.

Dataset for unknown attacks We have compiled a distinctive library of attack signatures for the three types of attacks, based on extensive research data and our practical experiences. However, this approach may not be entirely effective in safeguarding against undisclosed and unknown security threats since such attacks are not included in the attack library. To address this issue, we have leveraged a deep learning model for analyzing SQL payloads. This model has significantly enhanced our ability to identify and characterize SQL Injection attacks by accurately summarizing their patterns. Additionally, the model has enabled us to detect the attack behaviors of previously unknown threats following their intrusion attempts.

6 CONCLUSION

This paper presents the HP-RASP system, which we systematically demonstrate from three main directions: research background, system design and implementation, and experiments.

In the research background section, we highlight the unique advantages of RASP over WAF technology, specifically its ability to better capture the behavior of network traffic inside the application. To implement HP-RASP, we conducted extensive research on SQL Injections, XSS attacks, and JAVA Deserialization attacks based on OpenRASP. We also leveraged the ability of RASP to unpack complex requests into SQL payloads to utilize the BERT model, which is widely used in natural language processing.

Next, in the system design and implementation section, we discuss the insertion process of crucial hook points in the RASP system and introduce detection rules in the JavaScript plugins, which we then combine. In the BERT detection module, we pre-trained the model to familiarize it with the characteristics of SQL statements and fine-tuned it to distinguish between benign and malicious SQL statements.

Table 6: Comparison among Different Solutions (Time Consumption): Raw Application, Application with OpenRASP, Application with HP-RASP, and Application with HP-RASP and BERT model

Solution	50% Line	90% Line	95% Line	Throughout Capacity	Receive Rate(KB/s)	Send Rate(KB/s)
Raw application (benign)	4	6	6	200.1	1016.11	29.9
Raw application (malicious)	5	5	5	200	1025.18	33.1
OpenRASP (benign)	5	6	7	199.8	1029.22	29.86
OpenRASP (malicious)	5	9	30	199.8	1062.84	36.2
HP-RASP without BERT model (benign)	5	7	10	199.9	1028.87	29.87
HP-RASP without BERT model (malicious)	5	7	9	199.9	1059.99	34.77
HP-RASP with BERT model (benign)	5	7	10	200	1028.93	29.87
HP-RASP with BERT model (malicious)	5	7	9	200.1	1069.78	34.79

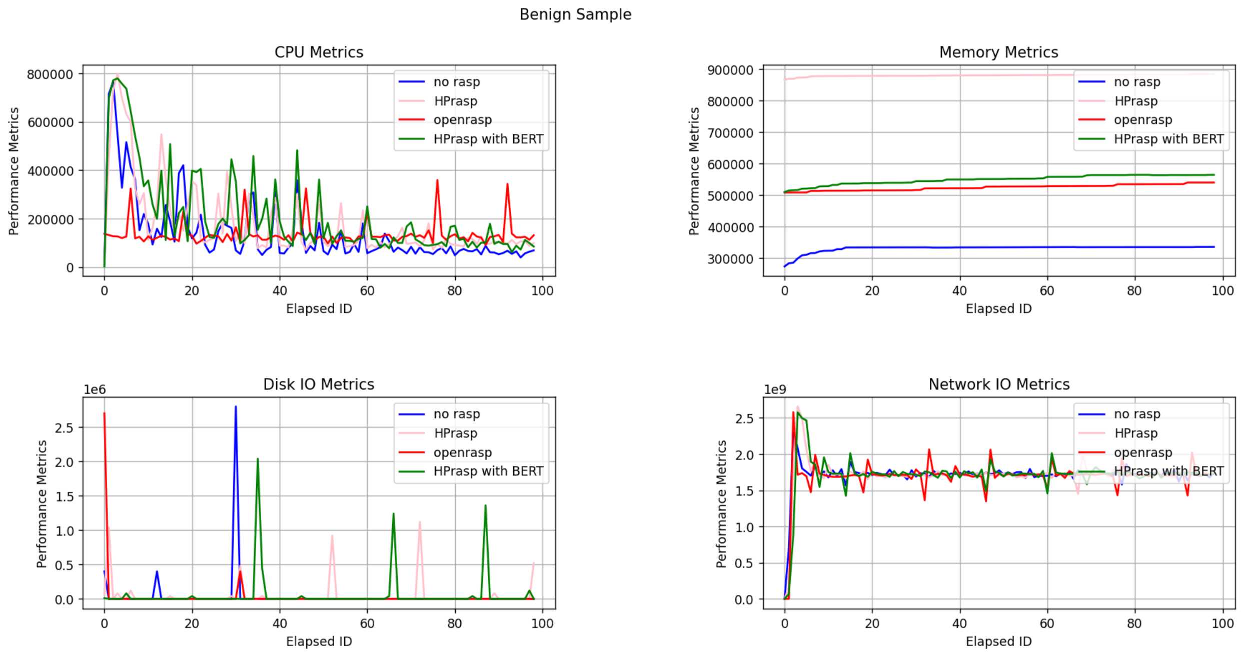


Figure 6: Comparison among Different Solutions (System Resource Monitoring when Receiving Good Samples): Raw Application, Application with OpenRASP, Application with HP-RASP, and Application with HP-RASP and BERT model

In the experimental section, we evaluate the HP-RASP system's performance from three dimensions: detection accuracy, time consumption, and system resource consumption. Our experimental results show that the HP-RASP system significantly improves detection accuracy compared to the prototype system, with the BERT model playing a critical role in this improvement. Additionally, the HP-RASP system imposes an acceptable burden on time consumption and system resource consumption.

In future research, we plan to focus on three main areas. First, we aim to collect enough datasets on XSS attacks and apply the BERT model to detect them. Second, we plan to optimize the system efficiency and improve its engineering applicability by using cleaner rules and more efficient structures. Finally, we intend to explore the possibility of porting the RASP system to other programming language platforms, including low-code platforms.

7 RELATED WORK

RASP Technology According to Gartner[13], RASP technology was first proposed in 2012. It was described as being built or linked into an application or its runtime environment and is capable of detecting and preventing real-time attacks by controlling the application. In the same year, E. Yuan[47] elaborated the main taxonomy and survey of self-protecting systems. In 2016, more detailed frameworks were proposed by Petar et al.[49] and Adrian et al.[23].

In recent years, there have been several developments in RASP technology. For instance, Yin et al.[46] developed a scheme on RASP against script injection attacks based on data flow analysis and automatic insertion of filters before relevant sink statements. Salemi et al.[36] modified the WAF using the RASP in order to improve the security level of this solution. Moreover, RASP technology has been applied to the blockchain security research area[25, 44], using traditional security technology to solve blockchain security issues.

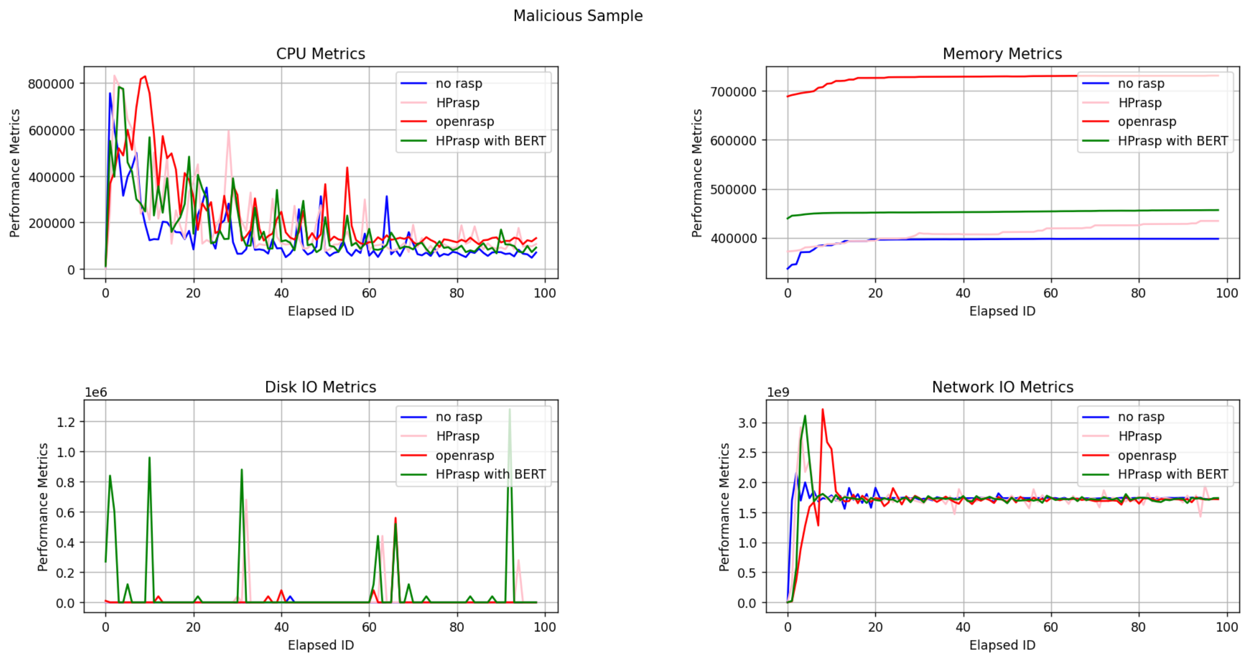


Figure 7: Comparison among Different Solutions (System Resource Monitoring when Receiving Bad Samples): Raw Application, Application with OpenRASP, Application with HP-RASP, and Application with HP-RASP and BERT model

SQL Injections Detection Technology SQL Injection occurs when the attacker tries to insert malicious code into the Web Applications' database. For blind SQL injection, which our RASP mainly aims to prevent, there have been many pieces of research and products developed before. For example, AMNESIA[15] is a tool that stands for analysis and monitoring for neutralizing SQL injection attacks. Komiya et al.[20] used machine learning algorithms to improve the performance of the detection.

To deal with SQL injection attacks along with the XSS attack, which is mostly done with the help of JavaScript, several researchers have developed methods for protection. For instance, Rattipong et al.[34] exploited the cookie table using XSS Attack with the SQL Injection. Zhang et al.[48] used the execution flow mechanism in order to protect from JavaScript based XSS attack along with SQL injection attacks.

XSS Attacks Detection Technology Cross-site scripting (XSS) attacks have been a persistent problem for web applications since the early 2000s [**?**]. There are two main technologies for detecting XSS attacks: server-side detection and client-side detection. Our RASP system can be classified as a client-side detection technology. Mitropoulos et al. attempted to defend against XSS attacks driven by malicious JavaScript by collecting the elements corresponding to a script from the browser's JavaScript engine and generating their fingerprints, ultimately verifying the fingerprints [28]. Weissbacher et al. proposed ZigZag, which automatically hardens JavaScript-based web applications using anomaly detection techniques that monitor the client-side code's execution traces [43]. These approaches can complement the client-side detection provided by our RASP system.

Java Deserialization Attacks Detection Technology Deserialization in Java is a process of reconstructing an object from structured data. Nowadays, the most common data format used for serializing data is JSON. In order to detect Java Deserialization and object injection vulnerabilities in web applications, ObjectMap, a tool proposed by Koutroumpouchos et al.[21], can be used. For protection against Deserialization attacks, Cristalli et al.[8] presented a new sandboxing approach that defines the Deserialization behavior based on trusted execution paths for Java applications.

BERT Model and its Pre-training In 2018, Devlin et al.[11] introduced a new language representation model called BERT, which stands for Bidirectional Encoder Representations from Transformers. It was designed to pre-train deep bidirectional representations from the unlabeled text by jointly conditioning on both the left and right context in all layers.

REFERENCES

[1] Ali Ahmad, Zahid Anwar, Ali Hur, and Hafiz Farooq Ahmad. 2012. Formal reasoning of web application Firewall rules through ontological modeling. In *2012 15th International Multitopic Conference (INMIC)*. IEEE, 230–237.
[2] Zainab S Alwan and Manal F Younis. 2017. Detection and prevention of SQL injection attack: A survey. *International Journal of Computer Science and Mobile Computing* 6, 8 (2017), 5–17.
[3] Dennis Appelt, Cu D Nguyen, Annibale Panichella, and Lionel C Briand. 2018. A machine-learning-driven evolutionary approach for testing web application firewalls. *IEEE Transactions on Reliability* 67, 3 (2018), 733–757.
[4] Simon Applebaum, Tarek Gaber, and Ali Ahmed. 2021. Signature-based and machine-learning-based web application firewalls: A short survey. *Procedia Computer Science* 189 (2021), 359–367.
[5] Mohammed Babiker, Enis Karaarslan, and Yasar Hoscan. 2018. Web application attack detection and forensics: A survey. In *2018 6th international symposium on digital forensic and security (ISDFS)*. Ieee, 1–6.
[6] Petar Čisar and Sanja Maravić Čisar. 2016. The framework of runtime application self-protection technology. In *2016 IEEE 17th International Symposium on*

Computational Intelligence and Informatics (CINTI). IEEE, 000081–000086.

[7] Victor Clincy and Hossain Shahriar. 2018. Web application firewall: Network security models and configuration. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 835–836.

[8] Stefano Cristalli, Edoardo Vignati, Danilo Bruschi, and Andrea Lanzi. 2018. Trusted execution path for protecting java applications against deserialization of untrusted data. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 445–464.

[9] Luca Demetrio, Andrea Valenza, Gabriele Costa, and Giovanni Lagorio. 2020. Waf-a-mole: evading web application firewalls through adversarial machine learning. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 1745–1752.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[12] Chris Frohoff. 2018. ysoserial. https://github.com/frohoff/ysoserial.

[13] Gartner. 2012. IT Glossary. http://www.gartner.com/it-glossary/runtimeapplication-self-protection-rasp/.

[14] Teddy Surya Gunawan, Muhammad Kasim Lim, Mira Kartiwi, Noreha Abdul Malik, and Nanang Ismail. 2018. Penetration testing using Kali linux: SQL injection, XSS, wordpres, and WPA2 attacks. *Indonesian Journal of Electrical Engineering and Computer Science* 12, 2 (2018), 729–737.

[15] William GJ Halfond and Alessandro Orso. 2005. AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 174–183.

[16] Hong Ye He, Zhi Guo Yang, and Xiang Ning Chen. 2020. PERT: payload encoding representation from transformer for encrypted traffic classification. In *2020 ITU Kaleidoscope: Industry-Driven Digital Transformation (ITU K)*. IEEE, 1–8.

[17] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. 2014. {ASM}: A Programmable Interface for Extending Android Security. In *23rd USENIX Security Symposium (USENIX Security 14)*. 1005–1019.

[18] Hsiu-Chuan Huang, Zhi-Kai Zhang, Hao-Wen Cheng, and Shiuhpyng Winston Shieh. 2017. Web application security: threats, countermeasures, and pitfalls. *Computer* 50, 6 (2017), 81–85.

[19] Terry John. 2022. SANS 2022 Top New Attacks and Threat Report. (2022).

[20] Ryohei Komiya, Incheon Paik, and Masayuki Hisada. 2011. Classification of malicious web code by machine learning. In *2011 3rd International Conference on Awareness Science and Technology (iCAST)*. IEEE, 406–411.

[21] Nikolaos Koutroumpouchos, Georgios Lavdanis, Eleni Veroni, Christoforos Ntantogian, and Christos Xenakis. 2019. ObjectMap: Detecting insecure object deserialization. In *Proceedings of the 23rd Pan-Hellenic Conference on Informatics*. 67–72.

[22] Eugene Kuleshov. 2007. Using the ASM framework to implement common Java bytecode transformation patterns. *Aspect-Oriented Software Development* (2007).

[23] Adrian Lane. 2016. Understanding and Selecting RASP: Technology Overview. https://securosis.com/blog/understanding-and-selecting-rasp-technology-overview.

[24] Jingxi Liang, Wen Zhao, and Wei Ye. 2017. Anomaly-based web attack detection: a deep learning approach. In *Proceedings of the 2017 VI International Conference on Network, Communication and Computing*. 80–85.

[25] Clifford Liem, Eslam Abdallah, Chidinma Okoye, John O'Connor, Md Swawibe Ul Alam, and Stacy Janes. 2017. Runtime self-protection in a trusted blockchain-inspired ledger. In *15th Escar Europe*.

[26] Hongyu Liu, Bo Lang, Ming Liu, and Hanbing Yan. 2019. CNN and RNN based payload classification methods for attack detection. *Knowledge-Based Systems* 163 (2019), 332–341.

[27] Zhonglin Liu, Yong Fang, Cheng Huang, and Jiaxuan Han. 2022. GraphXSS: an efficient XSS payload detection approach based on graph convolutional network. *Computers & Security* 114 (2022), 102597.

[28] Dimitris Mitropoulos, Konstantinos Stroggylos, Diomidis Spinellis, and Angelos D Keromytis. 2016. How to train your browser: Preventing XSS attacks using contextual script fingerprints. *ACM Transactions on Privacy and Security (TOPS)* 19, 1 (2016), 1–31.

[29] Asaad Moosa. 2010. Artificial neural network based web application firewall for SQL injection. *International Journal of Computer and Information Engineering* 4, 4 (2010), 610–619.

[30] Hai Thanh Nguyen, Carmen Torrano-Gimenez, Gonzalo Alvarez, Slobodan Petrović, and Katrin Franke. 2011. Application of the generic feature selection measure in detection of web attacks. In *Computational Intelligence in Security for Information Systems*. Springer, 25–32.

[31] Oracle. [n. d.]. Interface Instrumentation. https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html.

[32] Dariusz Pałka and Marek Zachara. 2011. Learning web application firewall-benefits and caveats. In *International Conference on Availability, Reliability, and Security*. Springer, 295–308.

[33] Parikshit Prabhudesai, Aniket A Bhalerao, and Rahul Prabhudesai. 2019. Web Application Firewall: Artificial Intelligence Arc. *International Research Journal of Engineering and Technology (IRJET)* (2019).

[34] Rattipong Putthacharoen and Pratheep Bunyatnoparat. 2011. Protecting cookies from cross site script attacks using dynamic cookies rewriting technique. In *13th International Conference on Advanced Communication Technology (ICACT2011)*. IEEE, 1090–1094.

[35] Abdul Razzaq, Ali Hur, Sidra Shahbaz, Muddassar Masood, and H Farooq Ahmad. 2013. Critical analysis on web application firewall solutions. In *2013 IEEE Eleventh International Symposium on Autonomous Decentralized Systems (ISADS)*. IEEE, 1–6.

[36] Marco Salemi, Ramin Sadre, and Axel Legay. 2020. " Automated rules generation into Web Application Firewall using Runtime Application Self-Protection. (2020).

[37] Aishwarya Seth et al. 2022. Comparing Effectiveness and Efficiency of Interactive Application Security Testing (IAST) and Runtime Application Self-Protection (RASP) Tools. (2022).

[38] Syed Saqlain Hussain Shah. 2019. sql injection dataset. https://www.kaggle.com/datasets/syedsaqlainhussain/sql-injection-dataset.

[39] Piyush A Sonewar and Nalini A Mhetre. 2015. A novel approach for detection of SQL injection and cross site scripting attacks. In *2015 International Conference on Pervasive Computing (ICPC)*. IEEE, 1–4.

[40] Éric Tanter, Rodolfo Toledo, Guillaume Pothier, and Jacques Noyé. 2008. Flexible metaprogramming and AOP in Java. *Science of Computer Programming* 72, 1-2 (2008), 22–30.

[41] ADEM Tekerek and OF Bay. 2019. Design and implementation of an artificial intelligence-based web application firewall model. *Neural Network World* 29, 4 (2019), 189–206.

[42] Carmen Torrano-Gimenez, Alejandro Perez-Villegas, and Gonzalo Alvarez. 2009. A self-learning anomaly-based web application firewall. In *Computational Intelligence in Security for Information Systems*. Springer, 85–92.

[43] Michael Weissbacher, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2015. {ZigZag}: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities. In *24th USENIX Security Symposium (USENIX Security 15)*. 737–752.

[44] WenChuan Yang and Jing Peng. 2020. Research on EVM-based smart contract runtime self-protection technology framework. In *Workshops of the International Conference on Advanced Information Networking and Applications*. Springer, 617–627.

[45] Imrana Abdullahi Yari, Babangida Abdullahi, and Steve A Adeshina. 2019. Towards a framework of configuring and evaluating Modsecurity WAF on Tomcat and Apache web servers. In *2019 15th International Conference on Electronics, Computer and Computation (ICECCO)*. IEEE, 1–7.

[46] Zhongxu Yin, Zhufeng Li, and Yan Cao. 2018. A Web Application Runtime Application Self-protection Scheme against Script Injection Attacks. In *International Conference on Cloud Computing and Security*. Springer, 566–577.

[47] Eric Yuan and Sam Malek. 2012. A taxonomy and survey of self-protecting software systems. In *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 109–118.

[48] Qianjie Zhang, Hao Chen, and Jianhua Sun. 2010. An execution-flow based method for detecting cross-site scripting attacks. In *The 2nd International Conference on Software Engineering and Data Mining*. IEEE, 160–165.

[49] Petar Čisar and Sanja Maravić Čisar. 2016. The framework of runtime application self-protection technology. In *2016 IEEE 17th International Symposium on Computational Intelligence and Informatics (CINTI)*. 000081–000086. https://doi.org/10.1109/CINTI.2016.7846383