Hindawi
Security and Communication Networks
Volume 2019, Article ID 8368473, 13 pages
https://doi.org/10.1155/2019/8368473

Research Article

Application-Level Unsupervised Outlier-Based Intrusion Detection and Prevention

Omar Iraqi [1,2] **and Hanan El Bakkali** [1]

[1] *Rabat-IT Center, ENSIAS, Mohammed V University, Rabat, Morocco*
[2] *School of Science and Engineering, Al Akhawayn University, Ifrane, Morocco*

Correspondence should be addressed to Omar Iraqi; o.iraqi@aui.ma

Received 31 December 2018; Revised 22 February 2019; Accepted 12 March 2019; Published 28 July 2019

Guest Editor: Stefano Calzavara

As cyber threats are permanently jeopardizing individuals privacy and organizations' security, there have been several efforts to empower software applications with built-in immunity. In this paper, we present our approach to immune applications through application-level, unsupervised, outlier-based intrusion detection and prevention. Our framework allows tracking application domain objects all along the processing lifecycle. It also leverages the application business context and learns from production data, without creating any training burden on the application owner. Moreover, as our framework uses runtime application instrumentation, it incurs no additional cost on the application provider. We build a fine-grained and rich-feature application behavioral model that gets down to the method level and its invocation context. We define features to be independent from the variable structure of method invocation parameters and returned values, while preserving security-relevant information. We implemented our framework in a Java environment and evaluated it on a widely-used, enterprise-grade, and open-source ERP. We tested several unsupervised outlier detection algorithms and distance functions. Our framework achieved the best results in terms of effectiveness using the Local Outlier Factor algorithm and the Clark distance, while the average instrumentation overhead per intercepted call remains acceptable.

1. Introduction

With the ever growing landscape of ubiquitous computing, cyberattacks are continually on the rise. The reputational and financial damages are just catastrophic for both organizations and governments. A class of security solutions that has attracted researchers over the last twenty years consists of enabling software applications to become immune against attacks [1–4]. This is a challenging area as it integrates several domains including anomaly-based intrusion detection and prevention [5–11], application partitioning and sandboxing [12–15], automatic error detection and patching [3, 4], as well as collaborative application communities [1, 16, 17]. To this end, not only can these fields be leveraged and combined in different ways, but they can also be approached from different perspectives and using different techniques. In particular, there is a plethora of anomaly-based intrusion detection approaches. They can be categorized based on:

(1) What data is collected: network packets, system calls [18], library calls [7], control flow graphs [6, 10, 11], etc.;

(2) Who collects such data: a network sensor, a system kernel hook [9], a dynamic runtime environment hook [10], or an application-level interceptor [1];

(3) From where such data is collected: the network, system audit trails, application source code [11], bytecode [10], or binaries [18];

(4) When the behavioral model is learned: before program execution (statically) [11] or during program execution (dynamically) [6, 10, 18]; and

(5) How the model is constructed: finite state automaton [6, 10, 11, 18], or machine learning including supervised [2, 19], semi-supervised and unsupervised learning.

In this paper, we present our framework for immune applications, where applications themselves play a *central* and *active* role in the intrusion detection and response processes. Moreover, such applications can leverage our framework

Table 1: Comparison of different anomaly-based intrusion detection approaches.

	What				*Who*			*Where*		*When*			*How*	
	SC	LC	CFG	IRF	OS	DRE	APP	SRC/B	BIN	BE	DE	FSA	SML	UnML
Feng [6]			x		x				x		x	x		
Jones [7]		x					x		x		x	x		
Ghosh [8]					x				x		x	x	x	
Ko [9]	x				x				x		x	x	x	
Hawkins [10]			x			x			x		x	x		
Wagner [11]	x			x				x		x		x		
Our approach	*x*	*x*	*x*	*x*			*x*	*x*			*x*			*x*

to unravel the targeted vulnerabilities, including zero-day, and track them back to their specific location in the source code, at the method level. In order to achieve these goals, the collection of relevant data shall be performed by the *application itself*, at the level of the instrumented application *source* code, or *bytecode* for managed languages such as Java, C# and Python. Runtime application instrumentation relieves the application provider from adding or changing anything in the software development process or code. It is up to the application owner to launch the application with an instrumentation agent implementing our framework. In Java for example, starting an application with an instrumentation agent is as easy as specifying an option in the command-line. Finally, as we would like to take account and advantage of the applications production and business context, we enable these applications to build their behavioral model *dynamically*, using *unsupervised* learning.

This work makes the following contributions:

(i) An approach to application intrusion detection and prevention, where applications play a *central* and *active* role, at no additional cost on the application provider;

(ii) An application of unsupervised outlier detection techniques to intrusion detection;

(iii) A fine-grained and rich-feature application behavioral model that gets down to the method level and even to its invocation context level. This level of granularity reduces false positives;

(iv) A Java implementation of our framework;

(v) An evaluation of our framework using popular, open-source and enterprise-class software products and tools.

This paper is organized as follows. Section 2 reviews the related work. Section 3 describes our approach to application-level, unsupervised, outlier-based intrusion detection and prevention. Section 4 gives an overview of our framework implementation and Section 5 describes its experimental evaluation. Section 6 gives and discusses the results in terms of effectiveness and efficiency. Finally, we conclude our paper by stating future work and direction.

2. Related Work

2.1. Application Anomaly-Based Intrusion Detection. We would like to start by defining what we exactly mean by application intrusion detection (AppID). Does it refer to any process that aims at detecting intrusions based on data that is gathered from applications (*what*), regardless of *who* gathers it, *when* it is gathered, or from *where* it is gathered? Consider for example using a standalone tool for extracting system calls, issued by an application, from an audit trail and analyzing them for intrusion detection. Consider also using kernel-level hooks for tracking these system calls and analyzing them. Since in both cases, the application is not even aware of the process and plays a passive role, can we still talk about application intrusion detection? According to Sielken who first coined the term in 1999, the answer is yes [5]. Since then, many researchers have been adopting this understanding [19].

In this work, we would like to take a step back regarding Sielken definition. More specifically, we are concerned about making a clear distinction between host-based intrusion detection and OS-level intrusion detection on one hand, and application-level intrusion detection on the other hand. To this end, we define application intrusion detection as the process that aims at detecting application-level intrusions by gathering and analyzing application data, whereby the application shall *play an active role* in the data gathering and analysis process. For performance and modularity considerations, the application may delegate the analysis to an external party. And to avoid cross-cutting concerns, data gathering can be applied orthogonally to the application functional code by a dynamic runtime environment, or through code instrumentation [1]. In both cases, data gathering interceptors – or advices – shall be invoked from within the application user space.

Table 1 shows a comparison between different approaches to anomaly-based application intrusion detection in its broad sense, as defined by Sielken. This comparison is based on:

(1) What data is collected: system calls (SC), library calls (LC), control flow graphs (CFG) or invocation rich features (IRF) such as call arguments;

(2) Who collects it: the OS, the dynamic runtime environment (DRE) or the application itself (APP);

(3) From where it is collected: at the level of the source code or bytecode (SRC/B) or at the level of the binaries (BIN);

(4) When it is collected: statically before execution (BE) or dynamically during execution (DE); and

(5) How it is analyzed: using finite state automata (FSA), supervised machine learning (SML) or unsupervised machine learning (UnML).

We can see through Table 1 that our approach is, to the best of our knowledge, the only one to use method invocation rich features, which are dynamically extracted by the application itself, at the bytecode level, and analyzed using unsupervised machine learning. In fact, even though Wagner uses call rich features, his approach is statically applied to the source code. Moreover, the so called features are limited to system call arguments that can be statically predicted [11].

2.2. Unsupervised Outlier Detection. Barnett & Lewis define an outlier as an observation, or subset of observations, which appear to be inconsistent with the remainder of that set of data [20]. Aggarwal and Yu note that outliers may be considered as noise points lying outside a set of defined clusters. These outliers behave differently from the norm [21]. In the context of intrusion detection, we consider Hawkins definition as the most pertinent: an outlier is an observation that deviates so much from other observations as to arouse suspicion that it was generated by a different mechanism [22]. In the literature, several terms have been used to refer to *outlier detection*: novelty detection, anomaly detection, noise detection, deviation detection or exception mining [23]. Outlier detection has been used in a variety of domains such as fraud detection, loan application processing, intrusion detection, activity monitoring, novelty detection in text and images, and medical condition monitoring [24].

There are many techniques and approaches to outlier detection. In particular, *unsupervised* machine learning is one of the most promising, mainly for two reasons. The first one is that intrusions are very rare compared to genuine requests. So we don't have balanced classes suitable for classification. The second reason is that we want the learning process to take account of the application, system or network context. Therefore, it shall be based on real life data from the production environment, not the development or test environment. Supervised learning would put an undesirable training burden on the application, system or network owner. We want to avoid that.

Many researchers have actually adopted unsupervised outlier detection for intrusion detection. For example, Lazarevic et al have applied unsupervised outlier detection algorithms to network intrusion detection [24]. One way of categorizing unsupervised outlier detection algorithms can be based on:

(i) Whether they are local or global, meaning whether they are based on the density of the neighborhood surrounding the record, or on the whole dataset for each record;

(ii) Whether they consider "outlierness" a binary property or a real value (degree of "outlierness");

(iii) Their performance and scalability; and

(iv) Their incremental capability to support data streams.

3. Our Application-Level Unsupervised Outlier-Based Intrusion Detection and Prevention

Our approach uses application-level intrusion detection to access and track application data all along its processing lifecycle, not just during I/O operations. Moreover, data is made available as domain model objects, structures and high-level programming language types, not just as raw strings or bytes. On the other hand, unsupervised outlier-based detection allows leveraging the application business context and real data, without creating any training burden on the application owner. This unique combination of application-level and unsupervised outlier-based intrusion detection allows our approach to leverage otherwise inaccessible context, granularity and data [1].

We target method invocations as these are the core units of authorized as well as unauthorized computing. They also provide rich features and context for intrusion detection. Our approach defines such features and context, and empowers applications to extract, gather and analyze them.

3.1. Method Invocation Features. For each method invocation, we extract meaningful features from a security standpoint. These are security-relevant metadata about parameters and the returned value if any, and whether an exception has been thrown or not. Parameters and the returned value may vary from simple primitive types to complex graphs of objects. For the same method and parameter or returned value, the structure of these graphs may vary a lot across invocations, with an arbitrary number of nodes, links, and loops.

We have specifically and carefully defined the metadata of each parameter and the returned value to be independent from this variable structure, while remaining relevant to intrusion detection. To this end, we start by generating a spanning tree from the graph of each invocation parameter and returned value, where the leaves contain primitive values: strings and numbers. Then we identify the most unusual leaf path, the most unusual n-grams (the values of n can be specified at the level of implementation), the most unusual string length, and the most unusual number. This is what allows us to detect malicious input as well as data leakage, no matter how they are hidden in the complex and dynamic structures of parameters or returned values.

The rationale behind identifying unusual n-grams is to spot attacks of type SQL/Code injection and XSS among others. Indeed, such attacks use commands and syntax that are not supposed to be there. Likewise, identifying unusual numbers, string lengths and paths is motivated by the need to uncover fraudulent operations as well as data leakage. It also helps unraveling buffer overflow/read attacks among others, which try to allocate and use more or less space than usual.

For example, Heartbleed bug exploits a vulnerability at the level of OpenSSL. More specifically, it provides *tls1_process_heartbeat(SSL ∗s)* method with a malicious parameter claiming a payload length in *s⟶s3⟶rrec.data* that is larger than its real value as reflected by *s⟶s3⟶rrec.length* [25]. This difference creates a buffer overread. While the request/response payload can be up to

```
public class Checkout {
        public void process( ShoppingCart shoppingCart ) throws Exception {
                //implementation
        }
}
public class ShoppingCart {
        private User user ;
        private ArrayList < SelectedProduct > selectedProducts ;
        private float totalPriceWithoutTax ;
        private float tax;
        private float totalWeight ;
        //more code...
}
public class SelectedProduct {
        private Product product ;
        private int selectedQuantity ;
        //more code...
}
public class Product {
        private int id;
        private String barcode;
        private String name;
        private String category;
        private float priceWithoutTax ;
        private flot taxRate ;
        private float discount;
        private float weight;
        private HashMap <String, String> characteristics;
        //more code...
}
public class User {
        private int id;
        private String fullName ;
        private String address;
        //more code...
}
```

Code 1: An e-commerce application example.

2^{14} – (1 + 2 + 16) [26], genuine implementations adopt a smaller length, e.g. 18 bytes for OpenSSL [27]. However, to steal maximum data, an attacker may send an even smaller or zero-length payload and claim the maximum value. This leads to a $s \longrightarrow s3 \longrightarrow rrec.length$ that is unusually low and causes the method to leak encryption key material whose length is unusually high [25]. Even if the attacker sent a payload with a typical length, leaked data length would still be unusually high.

We compute the minimum frequency of all paths (Feature 1 – F1), the minimum frequency of n-grams (F2[n]), the maximum variation of strings lengths (F3), the maximum variation of numbers (F4), and the variation of the length of the parameter's JSON representation (F5). These frequencies and variations are computed with respect to leaves at the same path in previous invocations. As for number variation, it is computed as the difference between the current observation and the mean observation divided by the standard deviation of observations at the same path in previous invocations. If the path of an n-gram or a number has not been seen before, then the frequency of the n-gram is set to 1 (nothing unusual), and the variation of the number is set to 0 (nothing unusual). However, the invocation is penalized by the fact that the path is very unusual. Indeed, its frequency is set as low as 1 (first time seen) over the number of invocations made so far, which increases the invocation "outlierness".

3.1.1. Example – An E-Commerce Application. To better illustrate these method invocation features, we consider an e-commerce application with a *Checkout* class acting on the user *ShoppingCart* through a *process* method as shown in Code 1.

We focus on the *process* method of the *Checkout* procedure for two reasons. The first one is that the *process* method better illustrates how invocation features are extracted from a complex parameter like *ShoppingCart*. The second reason is that the *process* method could be an effective checkpoint to detect fraudulent transactions like those containing an unusual number of products or quantities, and consequently an unusually high total price. Such a situation may arise for

```
{"user":{"id":1001,"login":"Login_1","fullName":"User_1","address":"Address_1"},
"selectedProducts":[{"product":{"id":1,"barcode":"Barcode_1","name":"Product_1",
"category":"Category_1","priceWithoutTax":100.0,"taxRate":20.0,"discount":5.0,"weight":0.0,
"characteristics":{"characteristic11":"value_11","characteristic13":"value_13",
"characteristic12":"value_12"}},"selectedQuantity":1},{"product":{"id":2,"barcode":"Barcode_2",
"name":"Product_2","category":"Category_2","priceWithoutTax":200.0,"taxRate":20.0,"discount":
5.0,"weight":0.0,"characteristics":{"characteristic22":"value_22","characteristic21":
"value_21","characteristic23":"value_23"}},"selectedQuantity":2},{"product":{"id":3,
"barcode":"Barcode_3","name":"Product_3","category":"Category_3","priceWithoutTax":300.0,
"taxRate":20.0,"discount":5.0,"weight":0.0,"characteristics":{"characteristic33":"value_33",
"characteristic32":"value_32","characteristic31":"value_31"}},"selectedQuantity":3}],
"totalPriceWithoutTax":1330.0,"tax":266.0,"totalWeight":0.0}
{"user":{"id":1002,"login":"Login_2","fullName":"User_2","address":"Address_2"},
"selectedProducts":[{"product":{"id":1,"barcode":"Barcode_1","name":"Product_1","category":
"Category_1","priceWithoutTax":100.0,"taxRate":20.0,"discount":5.0,"weight":0.0,
"characteristics":{"characteristic11":"value_11","characteristic13":"value_13",
"characteristic12":"value_12"}},"selectedQuantity":2},{"product":{"id":2,"barcode":
"Barcode_2","name":"Product_2","category":"Category_2","priceWithoutTax":200.0,"taxRate":
20.0,"discount":5.0,"weight":0.0,"characteristics":{"characteristic22":"value_22",
"characteristic21":"value_21","characteristic23":"value_23"}},"selectedQuantity":4},
{"product":{"id":3,"barcode":"Barcode_3","name":"Product_3","category":"Category_3",
"priceWithoutTax":300.0,"taxRate":20.0,"discount":5.0,"weight":0.0,"characteristics":
{"characteristic33":"value_33","characteristic32":"value_32","characteristic31":"value_31"}},
"selectedQuantity":6},{"product":{"id":4,"barcode":"Barcode_4","name":"Product_4",
"category":"Category_4","priceWithoutTax":400.0,"taxRate":20.0,"discount":5.0,"weight":0.0,
"characteristics":{"characteristic43":"value_43","characteristic42":"value_42",
"characteristic41":"value_41"}},"selectedQuantity":8}],"totalPriceWithoutTax":5700.0,"tax":
1140.0,"totalWeight":0.0}
```


Excerpt 1: Two sample invocations of *process* method—spanning tree of the *ShoppingCart* parameter as a JSON object.

Table 2: Examples of spanning tree paths to leaves and values.

Spanning tree paths to leaves	Invocation 1	Invocation 2
p_0_user_login	Login_1	Login_2
p_0_selectedProducts_0_product_barcode	Barcode_1	Barcode_1
p_0_selectedProducts_0_product_characteristics_characteristic11	value_11	value_11
p_0_selectedProducts_0_selectedQuantity	1	2
p_0_totalPriceWithoutTax	1330.0	5700.0
p_0_selectedProducts_3_product_characteristics_characteristic42	*Inexistent path*	Value_42 *(1st time seen)*
p_0_selectedProducts_3_product_taxRate	*Inexistent path*	20 *(1st time seen)*

example when a malicious user has a stolen credit card and does not care about the total price.

We consider a scenario whereby two users checked out after selecting three and four products respectively. The first user has selected one item of *Product_1,* two of *Product_2* and three of *Product_3*. The second user has selected two items of *Product_1*, four of *Product_2*, six of *Product_3* and eight of *Product_4*. Excerpt 1 shows two spanning trees of the *ShoppingCart* parameter as JSON objects that are generated respectively out of the two corresponding invocations of the *process* method.

Table 2 shows examples of the spanning tree paths to leaves for each invocation. The *p_i_* prefix identifies the parameter at position *i*. In our example, *p_0_* prefix identifies *ShoppingCart*, the first and only parameter.

Values at the level of the same leaf are compared across invocations. As showcased in Table 2, the generated spanning tree may vary across invocations in terms of structure. For example, the breadth, depth, paths and number of leaves depend in our case on the number of products added by the user to his/her *ShoppingCart*. This reflects on the existence and size of the *selectedProducts* array and paths under its elements. Despite this changing structure of parameter's spanning tree across invocations, we always extract the same invocation features: minimum frequencies and maximum variations throughout the whole tree.

Excerpt 2 shows features extracted from the two sample invocations of the *process* method. In the second invocation, *selectedProducts* array contains an additional element, and consequently new paths/leaves in the tree:

```
1.0 1.225 0.5 0.5 0.333 0.333 0.0 Thread_1422880298849000568
0.5 1.853 0.5 1.0 0.143 0.143 0.0 Thread_8140491395022634864
On each line, features in this order: min path frequency, max
number variation, max string length variation, parameter's
JSON length variation, min 3-gram frequency, min 1-gram
frequency, exception (0/1), thread label
```

Excerpt 2: Features extracted from two sample invocations of *process* method.

p_0_selectedProducts_3_∗. Each new path has 0.5 frequency: 1 (first time seen) over 2 invocations. Hence, the minimum path frequency for the second invocation is 0.5.

In terms of maximum number variation, the second invocation exhibits a higher value (1.853) than the first invocation (1.225). This is due to the higher number of products and quantities selected by the second user with respect to the first one. Total prices are also different accordingly. In the first invocation, there is a number variation (1.225) because we also compare numbers among siblings and relatives under the same array-aggregated paths in the same invocation. For example, we determine the maximum variation of prices among all *selectedProducts* in that first invocation. We give more details on siblings, relatives and array-aggregated paths in Section 3.2 below.

As for the maximum string length variation, its value is the same for both invocations (0.5) because in our example, strings at the level of each leaf have the same length across invocations and across siblings and relatives. Moreover, whenever a value is distant from the mean by less than the standard deviation, we smooth its variation to 0.5. We give more details about this variation smoothing in Section 3.3 below.

For the same reason, the parameter's JSON length variation is equal to 0.5 for the first invocation. However, since the *ShoppingCart* of the second user contains more products, its JSON representation is longer than in the first invocation, which reflects on its length variation (1), i.e. distant from the mean by exactly the standard deviation.

3.2. Sibling/Relative Grouping and Comparison. When a node in the spanning tree of a parameter or of the returned value is an array, its elements represent sibling leaves or branches. Each leaf in each branch may have relatives in other branches. From the e-commerce application example, we can mention several relatives such as *p_0_selectedProducts_0_selectedQuantity* and *p_0_selectedProducts_1_selectedQuantity*, as well as *p_0_selectedProducts_0_product_barcode* and *p_0_selectedProducts_1_product_barcode*.

These siblings and relatives hold comparable values respectively. As such, we can group each set of siblings and relatives under an aggregated path in order to compare their values. Without such a grouping and inter-sibling/relative comparison, some malicious or leaked data could be easily hidden in an array element. Indeed, if the element path is seen for the first time, then previous invocations will be helpless. Apart from detecting the unusual path, we would not be able to detect the anomalous data without comparing it with its siblings or relatives across invocations.

From the e-commerce application example, *p_0_selectedProducts_3_product_taxRate* path is seen for the first time in the second invocation. If it held malicious data, then we would be able to spot it only if we compared it with its relatives, i.e. *p_0_selectedProducts_0_product_taxRate, p_0_selectedProducts_1_product_taxRate* and *p_0_selectedProducts_2_product_taxRate*.

In this context, we aggregate paths leading to siblings/relatives by mapping them to their common ancestor array and skipping the array indexes of siblings or of the branches leading to relatives. From the e-commerce application example, all *p_0_selectedProducts_[index]_selectedQuantity* paths lead to relatives, hence can all be aggregated under *p_0_selectedProducts_selectedQuantity*. From the same example, all *p_0_selectedProducts_[index]_product_barcode* paths can be aggregated under *p_0_selectedProducts_product_barcode*.

3.3. Variation and Frequency Smoothing. The idea of computing the variation of numbers and string lengths is to spot unusual values that deviate so considerably from the mean to arise suspicion. Similarly, the idea of computing the frequency of n-grams is to identify unusual sets of characters that are so rare to arise suspicion. However, while conducting experiments, we found out that in some cases, the maximum variations of numbers or string lengths are so small that they make corresponding records erroneously appear as outliers. We also found out that in some cases, the minimum frequencies of n-grams are so high that they make corresponding records erroneously look like outliers.

To address this issue, we apply a smoothing effect on small variations and high frequencies. Whenever a number or a string length deviates from the mean value across invocations by less than the standard deviation, we set its variation to 0.5. This way, all such variations share a common value (0.5) and avoid raising suspicion. Similarly, whenever a minimum frequency is higher than the average minimum frequency across invocations, we set it to the average minimum frequency. Therefore, such high minimum frequencies get aligned with the average minimum frequency and avoid raising suspicion.

3.4. Method Invocation Context. The same method may be invoked at different locations of a program, for different purposes and by different types of users. This is the invocation context, and we consider the call stack a key indicator of

such a context. Indeed, under different call stacks, the same method may take completely different sets of inputs, follow different control flows, and yield different sets of outputs. To have a meaningful monitoring of method invocations, we take into account the invocation context by augmenting the tree path mentioned above by the invocation call stack. We call it the contextualized path. This way, we discriminate the different invocation values at the same tree path, but belonging to different call stacks. We end up learning and building a model on a per method and call stack basis. In other words, if we monitored m methods $(M_i)_{1..m}$, and each M_i were reachable through $CS(M_i)$ call stacks, then we would have $\sum_{i=0}^{m} CS(M_i)$ models.

From the e-commerce example above, we can mention two invocation contexts of the *process* method. The first context would be related to normal customers, while the second context would be related to special customers. The latter would benefit from special treatment, discounts and services. They would also have a dedicated checkout screen, which would use the *process* method through a different call stack. We assign unique IDs, e.g. *CSNC* and *CSSC*, to call stacks leading to the *process* method invoked on behalf of normal and special customers respectively. In this scenario, for each tree path we will have two contextualized paths. For example, we will augment *p_0_selectedProducts_0_selectedQuantity* path with the two call stack IDs to generate the corresponding contextualized paths: *CSNC_p_0_selectedProducts_0_selectedQuantity* and *CSSC_p_0_selectedProducts_0_selectedQuantity*. We will also build two models, one per call stack, and each contextualized path will be under one of them.

3.5. Outlier Detection Algorithms. We have experimented with different unsupervised outlier detection algorithms, including EM Outlier Detection, Angle-based Outlier Detection [28] and Local Outlier Factor (LOF) [29]. This latter has proven to be very effective as shown by the precision and recall values in Section 6. The accuracy of LOF measured in this work is consistent with other research results. For example, in a comparative study of anomaly detection schemes in network intrusion detection, Lazarevic et al have found that LOF exhibited the best accuracy among all tested algorithms [24].

This can be explained by the fact that, as opposed to many outlier detection algorithms that consider "outlierness" a binary property, LOF assigns to each record in the dataset an "outlierness" score or factor. Moreover, this factor is based on the density of the neighborhood surrounding the record not all the dataset. This locality feature makes LOF outperform algorithms based on a global approach, especially when the dataset is made of scattered clusters [29]. LOF algorithm can also be augmented with the incremental capability, as demonstrated by Pokrajac et al. [30]. This capability makes LOF optimized for live data streams, generated by our application-level intrusion detection.

Formally, the *LOF* of a record R is the average of the ratio of the local reachability densities of R's k-nearest neighbors and that of R. As shown in Equation (1), lower is R's local reachability density, and the higher are the local reachability densities of R's k-nearest neighbors, the higher is the *LOF* value of R. Equation (2) defines the local reachability density *LRD* of a record R as the inverse of the average reachability distance based on the k-nearest neighbors of R [29].

$$LOF_k(R) = \frac{\sum_{O \in N_k(R)} \left(LRD_k(O) / LRD_k(R) \right)}{\left| \left(N_k(R) \right) \right|} \tag{1}$$

$$LRD_k(R) = \frac{1}{\left(\sum_{O \in N_k(R)} ReachDist_k(R, O) / \left| N_k(R) \right| \right)} \tag{2}$$

$$ReachDist_k(R, O) = \max \left(Distance_k(O), d(R, O) \right) \tag{3}$$

In equation (3), $\text{ReachDist}_k(R,O)$ represents the reachability distance between records R and O. $\text{Distance}_k(O)$ denotes the k-distance of record O, and it is defined as the distance d(O,P) between O and a record $P \in D$ (Dataset) such that:

(i) for at least k objects $P' \in D \setminus \{O\}$ it holds that $d(O,P') \leq d(O,P)$, and

(ii) for at most k-1 objects $P' \in D \setminus \{O\}$ it holds that $d(O,P') < d(O,P)$.

In a three dimensional space, $\text{Distance}_k(O)$ can be seen as the radius of the sphere whose center is O and contains at least k records both inside and on its surface, but at most k – 1 records inside.

Defining $\text{ReachDist}_k(R,O)$ as the maximum of $\text{Distance}_k(O)$ and d(R,O) means that for "sufficiently" close records R and O, their actual distance d(R,O) is replaced by $\text{Distance}_k(O)$. This allows to reduce statistical fluctuations of d(R,O) [29]. Different distance functions can be used to compute the distance d(R,O) between the records R and O as shown in the subsection below.

3.6. Distance Functions. While keeping the LOF algorithm, we tried different distance functions such as the Clark distance, the Manhattan distance, the Pearson correlation distance, as well as the Euclidean distance. Equations (4), (5), (6) and (7) define these distances between two d-dimension records P and Q [31, 32]. The Euclidean distance has outperformed the other distances and has given the best results. These are detailed in Section 6 below.

$$EuclideanDistance(P, Q) = \sqrt{\sum_{i=1}^{d} \left(P_i - Q_i \right)^2} \tag{4}$$

$$ClarkDistance(P, Q) = \sqrt{\sum_{i=1}^{d} \left(\frac{P_i - Q_i}{P_i + Q_i} \right)^2} \tag{5}$$

$$ManhattanDistance(P, Q) = \sum_{i=1}^{d} \left| P_i - Q_i \right| \tag{6}$$

$$PearsonCorrelationDistance(P, Q) = 1 - \frac{covariance(P, Q)}{\sigma_P \sigma_Q} \tag{7}$$

σ being the standard deviation

```
public static class MethodAdvice {
    public static FeatureExtractor featureExtractor = FeatureExtractor.getSingleton();
    @Advice.OnMethodEnter
    public static Invocation onEnter(@Advice.Origin String fullyQualifiedMethodName,
                @Advice.AllArguments Object[] params) {
//if first invocation in processing cycle, generate and append a unique label to //thread name
return new Invocation(fullyQualifiedMethodName, params);
    }
    @Advice.OnMethodExit
    public static void onExit(@Advice.Enter Invocation invocation,
                              @Advice.Return Object result, @Advice.Thrown throwable) {
invocation.update(result, throwable != null);
FeatureRecord featureRecord = featureExtractor.extract(invocation);
//send featureRecord tagged with the label generated above, to ELKI-based analysis and
//intrusion detection
    }
  }
```

Code 2: Advice to instrument target method.

3.7. Security Categorization, Outlierness, and Intrusion Prevention. One of the advantages of intrusion *detection* is that it offers the possibility to not block the processing, which may even be performed offline in another machine. This reduces drastically the performance overhead incurred by intrusion *detection* in comparison with intrusion *prevention*. Moreover, false positives in intrusion detection don't cause a denial of service, as opposed to intrusion prevention. Nevertheless, organizations are more and more interested in intrusion prevention systems for their capability to not only detect intrusions but to also stop them. Therefore, we support both modes of operation and allow the application owner to specify which one to use.

This choice will depend on the application business case, the organization risk profile and appetite, as well as the security budget it is willing to allocate to compensate for performance overhead. Moreover, when intrusion prevention is adopted, we take into consideration the application security categorization, according to a framework such as FIPS-199 [33]. More specifically, if the application has higher confidentiality or integrity requirements than its availability requirements, then a detected intrusion shall promote stopping that processing line and preventing the intrusion. Otherwise, a detected intrusion shall promote logging the intrusion and raising an alarm, without stopping the processing line. Other factors also promote one direction or the other. Higher the "outlierness" is, the more it shall promote stopping the processing line. Finally, the more alarms have been raised in the same processing line, the more they shall promote stopping that line.

4. Implementation Overview

4.1. Tools and APIs. We have used ELKI 0.7: Environment for Developing KDD-Applications Supported by Index-Structures [34] to integrate and test several unsupervised outlier detection algorithms. One main advantage of ELKI is the separation between the dataset representation, the data normalization methods, the outlier detection algorithms and the distance functions. In particular, different alternatives can be tested in each one of these dimensions independently of the others.

We have also used Google Gson 2.8.5 for extracting spanning trees from complex object graphs and serializing them as JSON objects. Through its simple API, Gson supports complex objects, even those with deep inheritance hierarchies and extensive use of generics. Moreover, the source code of such objects may not be available [35]. Last but not least, we have leveraged Byte Buddy 1.8.3 on top of Java instrumentation API. Byte Buddy is a code generation and manipulation library for creating and modifying classes at runtime [36]. For compatibility issues, we used OpenJDK 10 [37].

4.2. Data Structures and Code. We have defined an *Invocation* data structure that represents raw metadata about an intercepted method invocation such as parameters, returned value, etc. At the core of our code, we designed a *FeatureExtractor* singleton that serializes *Invocation* instances into JSON trees using Gson. Then, it processes them recursively and builds corresponding security-relevant *FeatureRecord* instances. As explained in Section 3 above, these instances are independent from the variable structure of parameters and returned values.

As shown in Code 2, the *FeatureExtractor* singleton is dynamically hooked into the application code using a *MethodAdvice*. This latter dynamically intercepts invocations of selected methods using Java instrumentation API with the support of Byte Buddy. It applies instrumentation logic just before and after method invocation, at method entry and exit respectively.

At the entry of every selected method, the *MethodAdvice* creates an *Invocation* instance initialized with the method fully qualified name, parameters and entry time. At the exit point, the *MethodAdvice* updates the *Invocation* instance with the returned value if any or thrown exception, the call stack ID and the exit time. Then, *MethodAdvice* uses

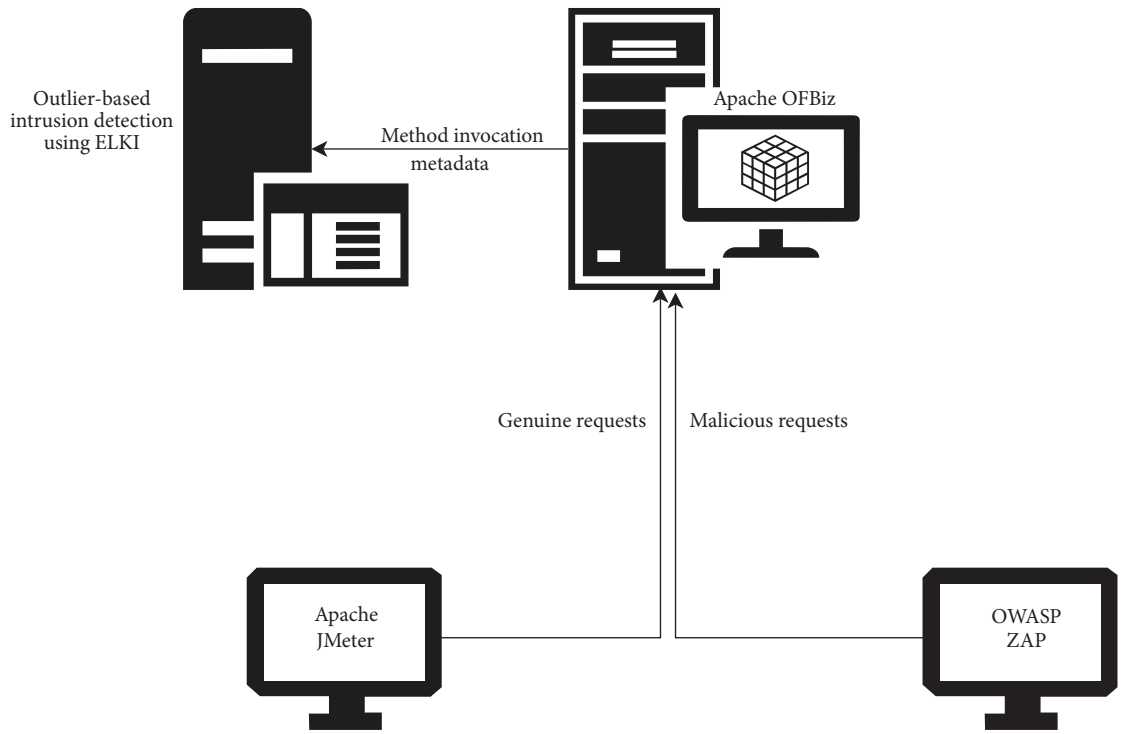

Figure 1: Test environment.

FeatureExtractor to transform the *Invocation* instance to a *FeatureRecord* instance and sends it to ELKI-based module for analysis and intrusion detection.

4.3. Lazy Serialization. Serializing invocation instances into JSON trees may in some cases interfere with, and affect the application logic. For example, serializing *java.sql.ResultSet* objects requires iterating over all query results, fetching data from the database, and moving each time the cursor forward. On one hand, the application may not need to fetch all the records, and on the other hand, we need to return the cursor back to its original position once done. Depending on the DBMS driver, this operation may not be supported. So in such cases, we defer serialization until the application itself tries to access the underlying records. This is what we call *lazy* serialization.

4.4. Request Processing Cycle Labeling. As multi-threaded applications may use thread pooling for performance optimization, we can't rely on thread IDs to uniquely identify request processing cycles. Therefore, we generate a unique label and append it to the thread name in the first method invocation of each processing cycle, as depicted in Code 2. This technique also addresses single-threaded applications. Moreover, when asynchronous, non-blocking (I/O) operations are performed, we mark each *promise* with the label generated for the processing cycle that owns it.

5. Experimental Evaluation

5.1. Test Environment. As shown in Figure 1, we adopted Apache OFBiz [38] as the cornerstone of our test environment, in addition to Apache JMeter [39] and OWASP Zed Attack Proxy (ZAP) [40]. Apache OFBiz is a free, open-source and widely-used enterprise resource planning (ERP) software. As such, it integrates several enterprise-grade applications/modules such as customer relationship management (CRM), accounting, human resource management, content and project management, etc. Apache JMeter is very popular as a free and open-source software performance evaluation tool, while OWASP ZAP is widely-known as a free and open-source vulnerability detection and penetration testing tool.

Based on our framework, we enable Apache OFBiz to become aware of intrusions and play an *active* role in their detection. As a Gradle project, it can be instrumented either at build time through a Gradle plugin, or at runtime through a Java agent. We use the agent alternative as it allows redefining JVM bootstrap classes, unlike the plugin alternative. We also increase the maximum size of Java heap size to 4 GB to support the high loads of testing.

All used stations are equipped with Intel i5 processor, as well as 8 GB DDR4 RAM. The stations hosting Apache OFBiz and ELKI-based software are running 64-bit Linux Ubuntu 16.04, while the stations hosting Apache JMeter and OWASP ZAP are running 64-bit Windows 7. All stations are connected to the same LAN to minimize network delays and jitter and consequently have an accurate evaluation of performance overhead.

5.2. Evaluation Process. We designed two test scenarios. The first scenario aims at evaluating the effectiveness of our framework in terms of precision and recall. The second scenario aims at evaluating the efficiency of our framework in terms of response time and memory usage.

In the first scenario, we use OWASP ZAP to launch attacks against Apache OFBiz through its invoice update web form. These attacks are of different types: buffer overflow, CRLF injection, XSS, CSRF, SQL injection, remote OS command injection, etc. Before launching the attacks, we run first JMeter to simulate normal traffic and allow our framework to build a sense of it. We do so by randomly generating posted data values using Javascript functions supported by JMeter for this purpose. In particular, we use *__Random(lower, upper)* to generate random numbers, as well as *__RandomString(length, chars)* to generate random strings. We even randomize the length of strings using the first function. More importantly, we specify the intervals and character sets to be plausible and coherent with the business domain. For example, we randomize the invoice message by varying its length between 0 and 30 and picking its characters from lower and upper case letters in addition to numbers and punctuation marks.

In the second scenario, we use JMeter to simulate a user logging into Apache OFBiz, searching for an invoice and updating it, then searching for a payment and updating it. We apply this scenario on both the original version of Apache OFBiz and its instrumented version, while varying the number of concurrent users and runs (loops of the same scenario).

We instrument Apache OFBiz at different levels: the controller where common request management tasks are performed and the model where business-specific processing is performed. At the controller level, we instrument *org.apache.ofbiz.webapp.control.ControlFilter* in order to label threads as explained in Section 4.4 above. At the level of the model and based on our evaluation scenarios, we focus on the invoicing and payment modules. Therefore, we instrument all public methods of *org.apache.ofbiz.accounting.invoice* and *org.apache.ofbiz.accounting.payment* packages. We also instrument the data access object *org.apache.ofbiz.entity.GenericDAO* that is responsible for managing all persistent entities and database operations.

5.3. Evaluation Metrics. While the "outlierness" score gives comparable and more accurate information about analyzed records than just a binary value, score interpretation remains a challenge [41]. Indeed, computed scores do not convey a specific meaning and may exhibit a low contrast. Moreover, their range may be completely different from one dataset to another, from one algorithm to another, and from one distance function to another. To address this issue, we can either normalize the scores and use the precision and recall metrics or use the R-precision metric without the need for score normalization.

5.3.1. Precision and Recall with Outlier Score Normalization. To normalize the scores, we can use different functions such as Gamma scaling and Sigmoid scaling as described in [42]. These functions shall fulfill three requirements:

(1) preserve the score order;

(2) yield values in the [0, 1] range that are interpretable as outlier probabilities; and

(3) increase contrast, i.e., yield high densities near 0 and 1, respectively.

We would then measure the precision and recall for different normalized score cutoff values such as 0.5, 0.6, 0.7, 0.8, and 0.9.

5.3.2. R-Precision. Another alternative consists of measuring the R-precision, which corresponds to the precision @ the top-R outliers [42]. R represents the a priori known number of relevant records, which are true intrusion attempts in our case. Based on this definition, the R-precision is also equal to the "R-recall": the recall @ the top-R outliers. Therefore, by measuring the R-precision, we measure both the precision and recall. More importantly, score interpretation is no more relevant. Indeed, all what matters is the score order so that we can pick the top-R outliers. Consequently, score normalization is no more needed. For this reason mainly, we adopted the R-precision as our evaluation metric.

6. Results and Discussion

6.1. Effectiveness Evaluation Results. We measured the R-precision for different distance functions (DF) and different k-nearest neighbors (LOF.k) values following the first evaluation scenario presented in Section 5.2 above. Table 3 shows the R-precision values that we obtained for a relevant method invocation context.

The Clark distance outperformed the other distances for LOF.k values in the [150 .. 1000] interval and achieved the best R-precision (97.44%) for LOF.k values in the [400 .. 700] interval. Beyond LOF.1000, the R-precision reached by the Clark distance remained constant and equal to the one achieved by both the Euclidean and Manhattan distances (92.31%). In the [70 .. 100] interval, Pearson correlation distance outperformed the other distances and reached its best R-precision (88.46%) for LOF.k values in the [70 .. 90] interval.

In addition to the R-precision achieved by our framework, these results unravel other interesting findings. First, there is no single optimum LOF.k value or interval for all distances. Likewise, there is no single best distance for all LOF.k values. Also, for a given distance, the R-precision does not necessarily evolve monotonically with LOF.k values. For example, the R-precision achieved by Pearson correlation distance increases in the [50 .. 90] interval before decreasing in the [90 .. 700] interval. It increases again in the [700 .. 850] interval, remains constant in the [850 .. 1250] interval, and decreases again in the [1250 .. 1500] interval.

Finally, as our framework exceeded 97% R-precision, it would be interesting to see how it stands against popular IDS/IPS solutions. To this end, a thorough and meaningful comparative study needs to be conducted using carefully-planned test scenarios and environment.

6.2. Efficiency Evaluation Results. We measured the average response time (ART) as well as the peak memory usage (PMU) following the second evaluation scenario presented in Section 5.2 above. As shown in Table 4, we varied the number

Table 3: R-precision (%) for different distance function (DF) and k-nearest neighbors (LOF.k) values.

LOF.k DF	50	60	70	80	90	100	150	250	400	700	850	1000	1250	1500
Clark	14.10	14.10	58.97	75.64	76.92	82.05	*89.74*	*96.15*	*97.44*	*97.44*	*94.87*	*93.59*	*92.31*	*92.31*
Euclidean	70.51	78.21	*87.18*	85.90	76.92	76.92	67.95	55.13	44.87	35.90	*92.31*	*92.31*	*92.31*	*92.31*
Manhattan	69.23	69.23	74.36	74.36	65.38	60.26	55.13	46.15	38.46	26.92	*92.31*	*92.31*	*92.31*	*92.31*
Pearson	34.62	61.54	*88.46*	*88.46*	*88.46*	*87.18*	84.62	83.33	75.64	61.54	*91.03*	*91.03*	*91.03*	84.62

Table 4: Instrumentation overhead versus increasing number of concurrent users.

Concurrent Users		5		10		15		20	
Runs per User		120		60		40		30	
Performance Indicator		ART [ms]	PMU [GB]	ART [ms]	PMU [GB]	ART [ms]	PMU [GB]	ART [ms]	PMU [GB]
Original (Noninstrumented) Apache OFBiz		142	3.05	210	3.88	303	3.98	400	4.15
Payment Module Instrumented 6 intercepted calls per run x 600 runs = 3600 intercepted calls	Indicator Value	165	3.05	243	4.07	337	4.23	456	4.31
	Overhead (%)	16.20	0.00	15.71	4.90	11.22	6.28	14.00	3.86
	Average Overhead per Intercepted Call (%)	*2.7*	*0.00*	*2.62*	*0.001*	*1.87*	*0.002*	*2.33*	*0.001*
Invoice Module Instrumented 48 intercepted calls per run x 600 runs = 28800 intercepted calls	Indicator Value	251	3.95	439	5.16	906	5.53	1056	5.62
	Overhead (%)	76.76	29.51	109.05	32.99	109.01	38.94	164.00	35.42
	Average Overhead per Intercepted Call (%)	*1.6*	*0.001*	*2.27*	*0.001*	*2.27*	*0.001*	*3.42*	*0.001*
Payment Module & Invoice Module Instrumented 54 intercepted calls per run x 600 runs = 32400 intercepted calls	Indicator Value	268	4.11	480	5.22	908	5.66	1080	5.66
	Overhead (%)	88.73	34.75	128.57	34.54	199.67	42.21	170.00	36.39
	Average Overhead per Intercepted Call (%)	*1.64*	*0.001*	*2.38*	*0.001*	*3.70*	*0.001*	*3.15*	*0.001*

of concurrent users, runs per user, and the instrumented Apache OFBiz modules.

In terms of response time, the overhead reported in Table 4 is the average overhead experienced by each user in each run. We call it *O(time, run)*. However, in terms of memory usage, the overhead reported in Table 4 is caused by all runs. We call it *O(memory, all)*. Therefore, we compute the average instrumentation overhead per intercepted call in terms of time *O(time, ic)* and memory usage *O(memory, ic)* according to (8) and (9), respectively.

$$O(time, ic) = \frac{O(time, run)}{NIC} \tag{8}$$

$$O(memory, ic) = \frac{O(memory, all)}{NIC * NR} \tag{9}$$

NIC: number of intercepted calls per run

NR: number of runs

As shown in Table 4, while instrumenting more modules and methods incur a higher cost, a promising result shows that the average instrumentation overhead per intercepted call is limited to 3.7% in terms of response time and as low as 0.01‰–0.0.2‰ in terms of memory usage. Moreover, it is worth mentioning that this evaluation is based on a proof of concept implementation, which has many opportunities for improvement and optimization.

7. Conclusion

In this paper, we presented our framework for application-level, unsupervised, outlier-based intrusion detection and prevention. More specifically, we have defined a fine-grained, rich-feature application behavioral model that gets down to the method level and its invocation context.

We implemented our framework in a Java environment and evaluated its effectiveness and efficiency on a widely-used, enterprise-grade, and open source ERP, while comparing several outlier detection algorithms and distance functions. The Local Outlier Factor algorithm, combined with the Clark distance, exceeded 97% R-precision. It also exceeded 92% R-precision when combined with the Euclidean distance, as well as the Manhattan distance. The average instrumentation overhead per intercepted call remains limited to 3.7% in terms of response time and less than 0.02‰ in terms of memory usage.

We have started experimenting with ideas that leverage our intrusion detection framework to enable applications unravel the targeted vulnerabilities and track them back

to the method level. To this end, we are testing the idea of correlating alarms raised by different method invocations in the same execution line to identify the vulnerable method. We will also explore mimicry attacks and optimize our framework accordingly. Moreover, we will conduct a thorough comparison of our framework against popular IDS/IPS solutions. We will also adopt evaluation metrics other than the R-precision by normalizing outlier scores and measuring the precision and recall at different cutoff values. Finally, we are intending to design, implement, and evaluate a cloud-based, distributed, and collaborative system that allows applications exchange intrusion and vulnerability-related information and alarms.

Data Availability

The source code of our framework along with the datasets generated and analyzed during the current study is publicly available on Github: https://github.com/oiraqi/immune-apps.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] O. Iraqi and H. E. Bakkali, "Toward third-party immune applications," in *Computer Network Security*, pp. 348–359, Springer, Cham, Switzerland, 2017.

[2] D. A. Fernandes, M. M. Freire, P. A. Fazendeiro, and P. R. Inácio, "Applications of artificial immune systems to computer security: a survey," *Journal of Information Security and Applications*, vol. 35, pp. 138–159, 2017.

[3] N. Perino, "A framework for self-healing software systems," in *Proceedings of the 2013 35th International Conference on Software Engineering, ICSE 2013*, pp. 1397–1400, IEEE Press, Piscataway, NJ, USA, May 2013.

[4] C. Watson, M. Coates, J. Melton, and D. Groves, "Creating attack-aware software applications with real-time defenses," *CrossTalk*, vol. 24, no. 5, pp. 14–18, 2011.

[5] R. S. Sielken, *Application Intrusion Detection*, University of Virginia, Charlottesville, Va, USA, 1999.

[6] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Proceedings of the 2003 IEEE Symposium on Security And Privacy*, pp. 62–75, IEEE Computer Society, USA, May 2003.

[7] A. K. Jones and Y. Lin, "Application intrusion detection using language library calls," in *Proceedings of the 17th Annual Computer Security Applications Conference, ACSAC 2001*, pp. 442–449, USA, December 2001.

[8] A. K. Ghosh, C. Michael, and M. Schatz, "A real-time intrusion detection system based on learning program behavior," in *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, pp. 93–109, Springer, London, UK, 2000.

[9] C. Ko, T. Fraser, and L. Badger, "Detecting and countering system intrusions using software wrappers," in *Proceedings of the 9th conference on USENIX Security Symposium*, vol. 9, Denver, Colo, USA, 2000.

[10] B. Hawkins and B. Demsky, "ZenIDS: introspective intrusion detection for PHP applications," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering, ICSE 2017*, pp. 232–243, IEEE, Argentina, May 2017.

[11] D. Wagner and D. Dean, "Intrusion detection via static analysis," *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, pp. 156–168, 2001.

[12] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. V. Styp-Rekowsky, "Seamless in-app ad blocking on stock android," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*, pp. 691–706, USENIX Association, Washington, DC, USA, 2015.

[13] M. Sun and G. Tan, "NativeGuard: protecting android applications from third-party native libraries," in *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec 2014*, pp. 165–176, ACM, UK, July 2014.

[14] Y.-Y. Hong, Y.-P. Wang, and J. Yin, "NativeProtector: protecting android applications by isolating and intercepting third-party native libraries," in *ICT Systems Security and Privacy Protection*, J.-H. Hoepman and S. Katzenbeisser, Eds., pp. 337–351, Springer International Publishing, 2016.

[15] F. Wang, Y. Zhang, K. Wang, P. Liu, and W. Wang, "Stay in your cage! a sound sandbox for third-party libraries on android," in *Computer Security – ESORICS 2016*, I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, Eds., pp. 458–476, Springer International Publishing, 2016.

[16] L. Kahloul, D. Boukhlouf, and O. Kazar, "Network security: distributed intrusion detection system using mobile agent technology," *International Journal of Communication Networks and Distributed Systems*, vol. 17, no. 4, pp. 335–347, 2016.

[17] F. Hosseinpour, S. Ramadass, A. Meulenberg, P. Vahdani Amoli, and Z. Moghaddasi, "Distributed agent based model for intrusion detection system based," *International Journal of Digital Content Technology and its Applications*, vol. 7, 2003.

[18] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automation-based method for detecting anomalous program behaviors," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp. 144–155, IEEE Computer Society, Washington, DC, USA, May 2001.

[19] L. Viljanen, "A survey on application level intrusion detection," Tekninen Raportti, Helsingin yliopisto, Tietojenkäsittelytieteen laitos, 2005.

[20] V. Barnett and T. Lewis, "Outliers in Statistical Data. 3rd edition. J. Wiley & Sons 1994, XVII. 582 pp., £49.95," *Biometrical Journal*, vol. 37, pp. 256-256, 2007.

[21] C. C. Aggarwal and P. S. Yu, "Outlier detection for high dimensional data," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pp. 37–46, ACM, USA, May 2001.

[22] D. M. Hawkins, *Identification of Outliers*, Springer, Netherlands, 1980.

[23] V. J. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial Intelligence Review*, vol. 22, no. 2, pp. 85–126, 2004.

[24] A. Lazarevic, L. Ertoz, V. Kumar, A. Ozgur, and J. Srivastava, "A comparative study of anomaly detection schemes in network intrusion detection," in *Proceedings of the 2003 SIAM International Conference on Data Mining*, pp. 25–36, SIAM, Philadelphia, Pa, USA, 2003.

[25] "Heartbleed Bug," http://heartbleed.com/.

[26] R. Seggelmann, M. Tuexen, and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension," RFC Editor RFC6520, 2012.

[27] "OpenSSL Heartbeat implementation," https://git.openssl.org/gitweb/?p=openssl.git;a=blob;f=ssl/t1_lib.c;h=b82fadace66e764b47ab2d854621ad89b804e8d2.

[28] H.-P. Kriegel, M. Schubert, and A. Zimek, "Angle-based outlier detection in high-dimensional data," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2008*, pp. 444–452, ACM, USA, August 2008.

[29] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: identifying density-based local outliers," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD '00)*, vol. 2 of *ACM Sigmod Record*, pp. 93–104, ACM, Dallas, Tex, USA, May 2000.

[30] D. Pokrajac, A. Lazarevic, and L. J. Latecki, "Incremental local outlier detection for data streams," in *Proceedings of the 1st IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2007*, pp. 504–515, USA, April 2007.

[31] M.-M. Deza and E. Deza, *Dictionary of Distances*, Elsevier Science, 1st edition, 2006.

[32] S.-H. Cha, "Comprehensive survey on distance/similarity measures between probability density functions," *International Journal of Mathematical Models and Methods in Applied Sciences*, vol. 1, p. 8, 2007.

[33] "Standards for security categorization of federal information and information systems," Tech. Rep. FIPS PUB 199, National Institute of Standards and Technology. Federal Information Processing Standards Publication, 2004, http://csrc.nist.gov/publications/fips/fips199/FIPS-PUB-199-final.pdf.

[34] E. Schubert, A. Koos, T. Emrich, A. Züfle, K. A. Schmid, and A. Zimek, "A framework for clustering uncertain data," in *Proceedings of the 3rd Workshop on Spatio-Temporal Database Management, STDBM 2015, Co-located with the 32nd International Conference on Very Large Data Bases, VLDB 2015*, pp. 1976–1979, Kohala Coast, Hawaii, USA, August 2015.

[35] "Gson: A Java serialization and deserialization library to convert Java Objects into JSON and back," https://github.com/google/gson.

[36] "Byte Buddy: A runtime code generation for the Java virtual machine," http://bytebuddy.net.

[37] OpenJDK, http://openjdk.java.net/.

[38] "OFBiz - Apache," https://ofbiz.apache.org/.

[39] "Apache JMeter - Apache JMeter^{TM}," http://jmeter.apache.org/.

[40] "OWASP Zed Attack Proxy Project - OWASP," https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

[41] H.-P. Kriegel, P. Kroger, E. Schubert, and A. Zimek, "Interpreting and unifying outlier scores," in *Proceedings of the 2011 SIAM International Conference on Data Mining*, B. Liu, H. Liu, C. Clifton, T. Washio, and C. Kamath, Eds., pp. 13–24, Society for Industrial and Applied Mathematics, 2011.

[42] C. Buckley and E. M. Voorhees, "Evaluating evaluation measure stability," *ACM SIGIR Forum*, vol. 51, p. 8, 2017.