Stealthy Denial of Service Strategy in Cloud Computing

Massimo Ficco and Massimiliano Rak

Abstract—The success of the cloud computing paradigm is due to its on-demand, self-service, and pay-by-use nature. According to this paradigm, the effects of Denial of Service (DoS) attacks involve not only the quality of the delivered service, but also the service maintenance costs in terms of resource consumption. Specifically, the longer the detection delay is, the higher the costs to be incurred. Therefore, a particular attention has to be paid for stealthy DoS attacks. They aim at minimizing their visibility, and at the same time, they can be as harmful as the brute-force attacks. They are sophisticated attacks tailored to leverage the worst-case performance of the target system through specific periodic, pulsing, and low-rate traffic patterns. In this paper, we propose a strategy to orchestrate stealthy attack patterns, which exhibit a slowly-increasing-intensity trend designed to inflict the maximum financial cost to the cloud customer, while respecting the job size and the service arrival rate imposed by the detection mechanisms. We describe both how to apply the proposed strategy, and its effects on the target system deployed in the cloud.

Index Terms—Cloud computing, sophisticated attacks strategy, low-rate attacks, intrusion detection

1 Introduction

Cloud Computing is an emerging paradigm that allows customers to obtain cloud resources and services according to an on-demand, self-service, and pay-by-use business model. Service level agreements (SLA) regulate the costs that the cloud customers have to pay for the provided quality of service (QoS) [1]. A side effect of such a model is that, it is prone to Denial of Service (DoS) and Distributed DoS (DDoS), which aim at reducing the service availability and performance by exhausting the resources of the service's host system (including memory, processing resources, and network bandwidth) [2]. Such attacks have special effects in the cloud due to the adopted pay-by-use business model. Specifically, in cloud computing also a partial service degradation due to an attack has direct effect on the service costs, and not only on the performance and availability perceived by the customer. The delay of the cloud service provider to diagnose the causes of the service degradation (i.e., if it is due to either an attack or an overload) can be considered as a security vulnerability. It can be exploited by attackers that aim at exhausting the cloud resources (allocated to satisfy the negotiated QoS), and seriously degrading the QoS, as happened to the BitBucket Cloud, which went down for 19h [3]. Therefore, the cloud management system has to implement specific countermeasures in order to avoid paying credits in case of accidental or deliberate intrusion that cause violations of QoS guarantees.

Over the past decade, many efforts have been devoted to the detection of DDoS attacks in distributed systems. Security prevention mechanisms usually use approaches based on rate-controlling, time-window, worst-case threshold, and pattern-matching methods to discriminate between the nominal system operation and malicious behaviors [4]. On the other hand, the attackers are aware of the presence of such protection mechanisms. They attempt to perform their activities in a "stealthy" fashion in order to elude the security mechanisms, by orchestrating and timing attack patterns that leverage specific weaknesses of target systems [5]. They are carried out by directing flows of legitimate service requests against a specific system at such a low-rate that would evade the DDoS detection mechanisms, and prolong the attack latency, i.e., the amount of time that the ongoing attack to the system has been undetected.

This paper presents a sophisticated strategy to orchestrate stealthy attack patterns against applications running in the cloud. Instead of aiming at making the service unavailable, the proposed strategy aims at exploiting the cloud flexibility, forcing the application to consume more resources than needed, affecting the cloud customer more on financial aspects than on the service availability. The attack pattern is orchestrated in order to evade, or however, greatly delay the techniques proposed in the literature to detect low-rate attacks. It does not exhibit a periodic waveform typical of low-rate exhausting attacks [5], [6], [7], [8]. In contrast with them, it is an iterative and incremental process. In particular, the attack potency (in terms of service requests rate and concurrent attack sources) is slowly enhanced by a patient attacker, in order to inflict significant financial losses, even if the attack pattern is performed in accordance to the maximum job size and arrival rate of the service requests allowed in the system. Using a simplified model empirically designed, we derive an expression for gradually increasing the potency of the attack, as a function of the reached service degradation (without knowing in advance the target system capability). We show that the features offered by the cloud provider, to ensure the SLA negotiated with the customer (including the load balancing and auto-scaling mechanisms), can be maliciously exploited by the proposed

• *The authors are with the Department of Industrial and Information Engineering, Second University of Naples (SUN), Via Roma 29, 81031, Aversa, Italy. E-mail: {massimo.ficco, massimiliano.rak}@unina2.it.*

Manuscript received 17 July 2013; revised 1 May 2014; accepted 6 May 2014. Date of publication 30 June 2014; date of current version 11 Mar. 2015. Recommended for acceptance by A. Martin.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCC.2014.2325045

stealthy attack, which slowly exhausts the resources provided by the cloud provider, and increases the costs incurred by the customer.

The proposed attack strategy, namely *Slowly-Increasing-Polymorphic DDoS Attack Strategy* (SIPDAS) can be applied to several kind of attacks, that leverage known application vulnerabilities, in order to degrade the service provided by the target application server running in the cloud. The term polymorphic is inspired to polymorphic attacks which change message sequence at every successive infection in order to evade signature detection mechanisms [9]. Even if the victim detects the SIPDAS attack, the attack strategy can be re-initiate by using a different application vulnerability (polymorphism in the form), or a different timing (polymorphism over time).

In order to validate the stealthy characteristics of the proposed SIPDAS attack, we explore potential solutions proposed in the literature to detect sophisticated low-rate DDoS attacks. We show that the proposed slowly-increasing polymorphic behavior induces enough overload on the target system (to cause a significant financial losses), and evades, or however, delays greatly the detection methods. Moreover, in order to explore the attack impact against an application deployed in a cloud environment, this paper focuses on one of the most serious threats to cloud computing, which comes from XML-based DoS (X-DoS) attacks to the web-based systems [10]. The experimental testbed is based on the mOSAIC framework, which offers both a 'Software Platform', that enables the execution of applications developed using the mOSAIC API, and a 'Cloud Agency', that acts as a provisioning system, brokering resources from a federation of cloud providers [11].

The rest of this paper is organized as follows. Background and related work are presented in Section 2. Section 3 illustrates several examples of attacks, which can be leveraged to implement the proposed attack pattern. Section 4 describes the proposed strategy to build the stealthy attacks, and presents the attack pattern, whose detailed implementation is reported in Section 5. Section 6 introduces the X-DoS attack used as case study. Section 7 shows the experimental results obtained running the attack pattern. Validation of stealthy characteristics is provided in Section 8. Some considerations about countermeasures against the proposed strategy are illustrated in Section 9. Conclusions and future work are described in Section 10.

2 Background and Related Work

2.1 Related Work

Sophisticated DDoS attacks are defined as that category of attacks, which are tailored to hurt a specific weak point in the target system design, in order to conduct denial of service or just to significantly degrade the performance [7], [12]. The term *stealthy* has been used in [13] to identify sophisticated attacks that are specifically designed to keep the malicious behaviors virtually invisible to the detection mechanisms. These attacks can be significantly harder to detect compared with more traditional brute-force and flooding style attacks [5].

The methods of launching sophisticated attacks can be categorized into two classes: *job-content-based* and *jobs arrival pattern-based*. The former have been designed in order to achieve the worst-case complexity of $O(n)$ elementary operations per submitted job, instead of the average case complexity of $O(1)$ [14], [15], [16]. The jobs arrival pattern-based attacks exploit the worst case traffic arrival pattern of requests that can be applied to the target system [7], [17]. In general, such sophisticated attacks are performed by sending a low-rate traffic in order to be unnoticed by the DDoS detection mechanisms. In recent years, variants of DoS attacks that use low-rate traffic have been proposed, including *Shrew* attacks (LDoS), *Reduction of Quality* attacks (RoQ), and *Low-Rate DoS attacks against application servers* (LoRDAS).

The terminology '*stealthy DDoS*' mainly refers to Shrew attacks first introduced in [6], which was followed by a series of related research [18], [19]. It refers to a periodic, pulsing, and low-rate attack traffic against the TCP protocol. Specifically, it has been used to exploit TCP's retransmission time-out (RTO) mechanism, provoking a TCP flow to repeatedly enter in a RTO state. This is obtained by sending high rate but short-duration bursts (having round trip time scale burst length), and repeating periodically at slower RTO time-scales [5], [16].

RoQ attacks target the dynamic operation of the adaptation mechanisms widely adopted to ensure that the workload would be distributed across the system resources to optimize the overall performance [7], [12], [20]. By using a specific attack pattern, RoQ induces constant oscillations between the overload and underload states, without sustaining the attack traffic. This is achieved by timing the attack traffic and its magnitude in order to exploit the dynamics of the system.

As for LoRDAS, the applications are usually more vulnerable to DDoS attacks due to their complexity. Specifically, LoRDAS attacks have no necessary deviations in terms of network traffic volumes or traffic distribution with respect to normal traffic. Due to its high similarity to legitimate network traffic and much lower launching overhead than classic DDoS attack, this new assault type cannot be efficiently detected or prevented by existing network-based solutions [21], [22]. Therefore, in recent years, the target of DDoS attacks has shifted from network to application server resources and procedures. Several LoRDAS attack models against application server have been proposed [8], [23], [24]. In particular, they aim at keeping the service queue of the target application servers completely full of requests coming from the attacker, so that any new incoming request sent by legitimate users is discarded. Macia-Fernandez et al. [24] present an evolution of the low-rate DDoS attack against iterative application servers, and extends its capabilities to concurrent systems. They assume the target server has a finite service queue, where the incoming service requests are temporarily stored to be served by the corresponding application process or thread. The attack takes advantage of the capacity to forecast the time at which the responses to incoming requests for a given service occur. This capability is used to schedule an intelligent pattern in such a way that the attacked server becomes busy the most time in processing of the malicious requests instead of those

from legitimate users. The actual availability of the service is thus reduced, while the data rate is low to elude potential defense mechanisms deployed against high-rate DDoS attacks at the server side. However, the major attention was paid to the mechanism for forecasting of the application server response times. Regarding this, Xiaodong et al. [8] propose a modified strategy, according to which the attack differs in behavior at different stage of the process. Specifically, the attack process has two different stages, before and after the service queue is filled up. The attacker monitors the number of attack requests that have been declined by the server within a recent period. If the requests are widely rejected, the attacker can assume that the service queue has been filled, and the attack is suspended. However, both the described attacks exhibit the typical periodic waveform of the low-rate DDoS attack, which consists of ON-OFF sequences of attack messages. Several works have proposed approaches to detect attacks that exhibit such a periodic and pulsing behavior [25], [26], [27].

Ben-Porat et al. [12] propose a vulnerability metric. It accounts for the maximal performance degradation (damage) that sophisticated attackers can inflict on the system using a specific amount of resources, normalized by the performance degradation attributed to regular users (using the same resources). In particular, they evaluate the vulnerability of both Open Hash and Closed Hash systems combined with some queuing mechanism, commonly used in computer networks. They show that under attack, the regular user still suffers from performance degradation, even after the attack has ended (such degradation is not due to the extra load of the attack, but due to its sophistication). However, the proposed metric has been defined to support the system designer to evaluate the relationship between the maximal job size allowed in the system and the vulnerability of the system to malicious attacks, and to choose the desired balance between the two. In contrast, our contribution aims at presenting an approach to build up sophisticated attack patterns, which cause significant damages (service performance degradation), even if the attack pattern is performed in accordance with the maximal job size and arrival rate allowed in the system (stealthy behavior). In particular, SIPDAS is a low-rate attach pattern that leverages the strengths of both LoRDAS and RoQ. On the one hand, it is designed to exploit a common vulnerability in application design or implementation [28]; on the other hand, it is able to target the dynamic operation of the adaptation mechanisms used to ensure the system performance.

To the best of our knowledge, none of the works proposed in the literature focus on stealthy attacks against application that run in the cloud environment.

2.2 Cloud Resources Provisioning

Cloud providers offer services to rent computation and storage capacity, in a way as transparent as possible, giving the impression of '*unlimited resource availability*'. However, such resources are not free. Therefore, cloud providers allow customers to obtain and configure suitably the system capacity, as well as to quickly renegotiate such capacity as their requirements change, in order that the customers can pay only for resources that they actually use. Several cloud providers offer the '*load balancing*' service for automatically distributing the incoming application service requests across multiple instances, as well as the '*auto scaling*' service for enabling consumers to closely follow the demand curve for their applications (reducing the need to acquire cloud resources in advance). In order to minimize the customer costs, the auto scaling ensures that the number of the application instances increases seamlessly during the demand spikes (to maintain the contracted performance), and decreases automatically during the demand lulls. For example, by using Amazon EC2 cloud services, the consumers can set a condition to add new computational instances when the average CPU utilization exceeds a fixed threshold. Moreover, they can configure a cool-down period in order to allow the application workload to stabilize before the auto scaling adds or removes the instances [29].

In the following, we will show how this feature can be maliciously exploited by a stealthy attack, which may slowly exhaust the resources provided by the cloud provider for ensuring the SLA, and enhance the costs incurred by the cloud customer.

2.3 The mOSAIC Framework

The mOSAIC project aimed at offering a simple way to develop and manage applications in a multi-cloud environment [11]. It provides a framework composed of two main components: the *cloud agency* and the *software platform*. The cloud agency acts as a provisioning system, brokering resources from a federation of cloud providers. The mOSAIC user develops the application on its local machine, then it uses a local instance of the cloud agency in order to start-up the process of remote resource acquisition and to deploy the Software Platform and the developed application. The Platform enables the execution of the developed applications on the acquired cloud resources.

A Java-based API is provided to develop software components in the form of *Cloudlets*. A mOSAIC application is a collection of Cloudlets, which are interconnected through communication resources, such as queues or shared key-value stores. The Cloudlets run on a dedicated operating system, named mOSAIC Operating System (mOS), which is a small Linux distribution. At runtime, the Software Platform transparently scales the Cloudlets instances on the acquired virtual machines (VM) on the base of the resource consumption (auto scaling). As an example, when the Platform detects that a Cloudlet is overloaded (e.g., it has too messages on the intercommunicating queues), it may choose to start a new Cloudlet instance. The Platform assumes such a decision on the base of policies defined by the application developer (through specific mOSAIC features). Finally, a load balancing mechanism automatically balances the application service requests among the instances.

3 DoS Attacks against Cloud Applications

In this section are presented several attack examples, which can be leveraged to implement the proposed SIPDAS attack pattern against a cloud application. In particular, we consider DDoS attacks that exploit application vulnerabilities

[10], [12], [30], including: the Oversize Payload attack that exploits the high memory consumption of XML processing; the Oversized Cryptography that exploits the flexible usability of the security elements defined by the WS-Security specification (e.g., an oversized security header of a SOAP message can cause the same effects of an Oversize Payload, as well as a chained encrypted key can lend to high memory and CPU consumptions); the Resource Exhaustion attacks use flows of messages that are correct regarding their message structure, but that are not properly correlated to any existing process instance on the target server (i.e., messages that can be discarded by the system, but at the expense of a huge amount of redundant work, such as the Business Process Execution Language (BPEL) based document, which must be read and processed completely, before they may safely be discarded); and attacks that exploit the worst-case performance of the system, for example by achieving the worst case complexity of Hash table data structure, or by using complex queries that force to spend much CPU time or disk access time.

In this paper, we use a Coercive Parsing attack as a case study, which represents one of the most serious threat for the cloud applications [10]. It exploits the XML verbosity and the complex parsing process (by using a large number of namespace declarations, oversized prefix names or namespace URIs). In particular, the *Deeply-Nested XML* is a resource exhaustion attack, which exploits the XML message format by inserting a large number of nested XML tags in the message body. The goal is to force the XML parser within the application server, to exhaust the computational resources by processing a large number of deeply-nested XML tags [30].

4 Stealthy DoS Characterization and Modeling

This section defines the characteristics that a DDoS attack against an application server running in the cloud should have to be stealth.

Regarding the quality of service provided to the user, we assume that the system performance under a DDoS attack is more degraded, as higher the average time to process the user service requests compared to the normal operation. Moreover, the attack is more expensive for the cloud customer and/or cloud provider, as higher the cloud resource consumption to process the malicious requests on the target system. From the point of view of the attacker, the main objective is to maximize the ratio between the amount of '*damage*' caused by the attack (in terms of service degradation and cloud resources consumed), and the the cost of mounting such an attack (called '*budget*') [7].

Therefore, the first requirement to design an efficient DDoS attack pattern is the ability of the attacker to assess the damage that the attack is inflicting to the system, by spending a specific budget to produce the malicious additional load. The attack damage is a function of the '*attack potency*', which depends on the number of concurrent attack sources, the request-rate of the attack flows, and the job-content associated to the service requests to be processed. Moreover, in order to make the attack stealthy, the attacker has to be able to estimate the maximum attack potency to be performed, without that the attack pattern exhibits a behavior that may be considered anomalous by the mechanisms used as a protection for the target system.

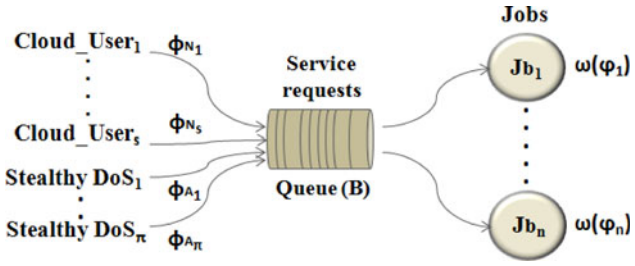

Fig. 1. Attack scenario.

In the following sections, starting from a synthetic representation of the target system, we describe the conditions the attack pattern has to satisfy to minimize its visibility as long as possible, and effectively affect the target system performance in the cloud environment.

4.1 Server under Attack Model

In order to assess the service degradation attributed to the attack, we define a synthetic representation of the system under attack. We suppose that the system consists of a pool of distributed VMs provided by the cloud provider, on which the application instances run. Moreover, we assume that a load balancing mechanism dispatches the user service requests among the instances. The instances can be automatically scaled up or down, by monitoring some parameter suitable to assess the provided QoS (e.g., the computational load, the used memory, and the number of active users). Specifically, we model the system under attack with a comprehensive capability ζ_M, which represents a global amount of work the system is able to perform in order to process the service requests. Such capability is affected by several parameters, such as the number of VMs assigned to the application, the CPU performance, the memory capability, etc. Each service request consumes a certain amount w_i of the capability ζ_M on the base of the payload of the service request. Thus, the load C_N of the system at time t can be modeled by a queuing system $M/M/n/n$ with Poisson arrivals, exponentially distributed service times, multiple servers, and n incoming requests in process (system capability). Moreover, the auto scaling feature of the cloud is modeled in a simple way: when new resources (e.g., VMs) are added to the system, the effect is an increase of the system capability ζ_M.

Therefore, given η legitimate type of service requests $\theta = (\vartheta_1, \ldots, \vartheta_\eta)$, and denoted w as the cost in terms of cloud resources necessary to process the service request $\varphi \in \theta$, an attack against a cloud system can be represented as in Fig. 1. Specifically, Fig. 1 shows a simple illustrative attack scenario, where the system is modeled as: (i) a queue (that conceptually represents the load balancing mechanism), in which are queued both the legitimate user request flows ϕ_{N_j} and the DDoS flows ϕ_{A_j} (attack sources), and (ii) a job for each service request that is currently processed on the system.

4.2 Stealthy Attack Objectives

In this section, we aim at defining the objectives that a sophisticated attacker would like to achieve, and the requirements the attack pattern has to satisfy to be stealth. Recall that, the purpose of the attack against cloud applications is not to necessarily deny the service, but rather to inflict significant degradation in some aspect of the service (e.g., service response time), namely *attack profit* P_A, in order to maximize the cloud *resource consumption* C_A to process malicious requests. In order to elude the attack detection, different attacks that use low-rate traffic (but well orchestrated and timed) have been presented in the literature. Therefore, several works have proposed techniques to detect low-rate DDoS attacks, which monitor anomalies in the fluctuation of the incoming traffic through either a time- or frequency-domain analysis [18], [25], [26]. They assume that, the main anomaly can be incurred during a low-rate attack is that, the incoming service requests fluctuate in a more extreme manner during an attack. The abnormal fluctuation is a combined result of two different kinds of behaviors: (i) a periodic and impulse trend in the attack pattern, and (ii) the fast decline in the incoming traffic volume (the legitimate requests are continually discarded). Therefore, in order to perform the attack in stealthy fashion with respect to the proposed detection techniques, an attacker has to inject low-rate message flows $\phi_{A_j} = [\varphi_{j,1}, \ldots, \varphi_{j,m}]$, that satisfy the following optimization problem:

Stealthy DDoS attack pattern in the cloud - Denote π the number of attack flows, and consider a time window T, the DDoS attack is successful in the cloud, if it maximizes the following functions of profit and resource consumption:

$$\begin{cases} maximize \quad P_A = \sum_{j=1}^{\pi} \sum_{i} g(\varphi_{j,i}), \\ maximize \quad C_A = \sum_{j=1}^{\pi} \sum_{i} w(\vartheta_{j,i}), \end{cases} \tag{1}$$

and it is performed in stealthy fashion, if each flow ϕ_{A_j} satisfies the following conditions:

$$\begin{cases} (c_1) \quad minimize \quad \delta_j, \forall j \in [1..\pi], \\ (c_2) \quad s.t.to \quad \varphi_{j,i} \in \theta, \\ (c_3) \quad s.t.to \quad exhibits\ a\ pattern\ neither\ periodic\ nor\ impulsive, \\ (c_4) \quad s.t.to \quad exhibits\ a\ slowly\ increasing\ intensity, \end{cases} \tag{2}$$

where:

- g is the profit of the malicious request $\varphi_{j,i}$, which expresses the service degradation (e.g., in terms of increment of average service time t_S to process the user requests with respect to the normal operation);
- δ_j is the average message rate of the flow ϕ_{A_j},
- w is the cost in terms of cloud resources necessary to process $\varphi_{j,i} \in \theta$.

Cond. (2.c_1) implies that the flow ϕ_{A_j} has to be injected with a low-rate δ_j. Cond. (2.c_2) assumes that all attack messages have to be legitimate service requests.

4.3 Creating Service Degradation

Considering a cloud system with a comprehensive capability ζ_M to process service requests φ_i, and a queue with size B that represents the bottleneck shared by the customer's flows ϕ_{N_j} and the DoS flows ϕ_{A_j} (Fig. 1). Denote C_0 as the load at time the onset of an attack period T (assumed to occur at time t_0), and C_N as the load to process the user requests on the target system during the time window T. To exhaust the target resources, a number n of flows ϕ_{A_j} have to be orchestrated, such that:

$$C_0(t_0) + C_N(T) + C_A(T) \geq \zeta_M * T, \tag{3}$$

where $C_A(T)$ represents the load to process the malicious requests φ_i during the period T. If we assume that (1) the attack flows are not limited to a peak rate due to a network bottleneck or an attacker's access link rate, and (2) the term C_N can be neglected during the attack ($C_A \gg C_N$), the malicious resource consumption C_A can be maximized if the following condition is verified:

$$C_A(T) \geq \zeta_M * T - C_0(t_0) \quad with \quad C_A \gg C_N. \tag{4}$$

Moreover, assume that during the period T, the requests $\varphi_i \in \phi_A$ burst at an average rate δ_A, whereas the flow ϕ_N bursts at an average rate δ_N. Denote B_0 as the queue size at time t_0, and d as the time that the queue becomes full, such that:

$$d = \frac{B - B_0}{\delta_A + \delta_N - \delta_p}, \tag{5}$$

where δ_p is the average rate of requests processed on the target system (i.e., the system throughput during the period T). After d seconds, the queue remains full if $\delta_A + \delta_N \geq \delta_p$. In particular, under attack, if $d < T$ and $C_A(\Omega) \geq \zeta_M * \Omega - C_0(t_0 + d)$, the attacker can archive the best profit P_A during the time window $\Omega = [t_0 + d, T]$ (i.e., there will be a high likelihood for the user requests to be neglected, forcing the client to perform a service request retransmission).

4.4 Minimize Attack Visibility

According to the previous stealthy attack definition, in order to reduce the attack visibility, Conditions (2) have to be satisfied. Therefore, through the analysis of both the target system and the legitimate service requests (e.g., the XML document structure included within the HTTP messages), a patient and intelligent attacker should be able to discover an application vulnerability (e.g., a Deeply-Nested XML vulnerability), and identify the set of legitimate service request types $\vartheta_k \subset \theta$ (Cond. (2.c_2)), which can be used to leverage such vulnerability. For example, for an X-DoS attack, the attacker could implement a set of XML messages with different number of nested tags $n_{T_i} = 1, \ldots, N_T$. The threshold N_T can be either fixed arbitrarily, or possibly, estimated during a training phase, in which the attacker injects a sequence of messages with nested XML tags growing, in order to identify a possible limitation imposed by a threshold-based XML validation schema. A similar approach can be used to estimate the maximum message rate δ_T with which injecting the service requests φ_i. Then, the attacker

has to define the minimal number π of flows ϕ_A characterized by malicious requests injected with:

- an average message rate lower than δ_T, in order to evade rate-controlling- and time-window-based detection mechanisms (Cond. (2.c_1)), and
- a polymorphic pattern (described in the next section), in order to evade low-rate detection mechanisms (Conditions (2.c_3 and 2.c_4)),

such that maximize the functions P_A and C_A (Eq. (1)).

5 Slowly-Increasing-Polymorphic DDoS Attack Pattern against Cloud Application

In this section, on the basis of the attack objectives and characterization described in Section 4, we present a strategy to implement attack patterns that optimize the problem described by Conditions (1) and (2).

5.1 Slowly-Increasing Polymorphic Attack Strategy

Denote w as the payload in terms of cloud resources ζ_i (e.g., average CPU load), necessary to process the service request $\varphi_i \in \theta$ during a time period Δ_i (i.e., $w(\varphi_i) = \zeta_i * \Delta_i$), Eq. (4) can be formulated as $C_A(T) = \sum \zeta_i * \Delta_i \geq \zeta_M * T - C_0(t_0)$. Moreover, assuming that n is the average number of malicious requests $\Phi_n = [\varphi_1, \ldots, \varphi_n]$ (with $\varphi_i \in \theta$) sent by the attacker during a time window T, and m is the average number of malicious requests Φ_m actually processed by the target system during T; in order to maximize the functions P_A and C_A, the DDoS attack has to satisfy the following conditions:

$$\begin{cases} \frac{n}{\sum_{i=1}^{n} t_{I_i}} \geq \frac{m}{\sum_{i=1}^{m} t_{S_i}} & with\ m \leq n, \\ \sum_{i=1}^{m} \zeta_i * \Delta_i \geq \zeta_M * T - C_0(t_0) & with\ C_A \gg C_N, \end{cases} \quad (6)$$

where, t_{I_i} is the inter-arrival time between two consecutive requests $\varphi_i \in \Phi_n$, and t_{S_i} is the average service time to process $\varphi_i \in \Phi_m$. Eq. (6) expresses that, in order to identify a suboptimal solution of the problem described by Functions (1), the attacker should know in advance the comprehensive system capacity ζ_M and the payload w for each type of request $\varphi_i \in \theta$. The knowledge of such information would allow to orchestrate an attack that, on the one hand, achieves the desired service degradation, on the other hand, minimizes the attack potency in order to hide the malicious behavior and to reduce the budget needed to mount the attack.

To implement an attack pattern that maximizes P_A and C_A, as well as satisfies Conditions (2), without knowing in advance the target system characteristics, we propose a Slowly-Increasing Polymorphic attack strategy, which is an *iterative* and *incremental* process. At the first iteration only a limited number π of flows ϕ_A are injected. The value π is increased by one unit at each iteration p, until the desired service degradation is achieved. During each iteration, the flows ϕ_A exhibit the *attack intensity* shown in Fig. 2. Specifically, each flow ϕ_{A_j} consists of burst of messages, in which the parameter $I_0(p)$ means the initial attack intensity at the iteration p (which can be orchestrated by varying the number and type of injected requests), T is the length of the burst period, and ΔI is the increment of the attack intensity each time a specific condition V is false. V is tested at the end of each period T. The satisfaction of the condition V identifies the achievement of the desired service degradation.

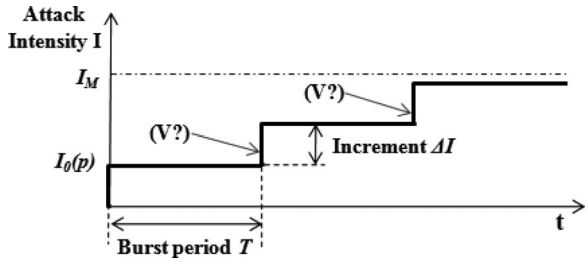

Fig. 2. SIPDAS lowly-increasing intensity behavior.

The purpose of using an iterative and incremental approach is related to the inability of knowing in advance the target system capability ζ_M and the payload $w(\varphi_i)$. The parameter ΔI sets the hypothetical overload that the attacker would like to add on the target system. The value ΔI has to be manipulated by the attacker, and controlled within a very small range to hide the attack behavior, and prolong the attack detection latency. The intensity $I(t)$ is periodically increased until it exceeds a threshold I_M, beyond which the attack may be detected. In such case, a new attack iteration $p+1$ is performed, in which another flow ϕ_{A_j} is added, and a new attack intensity $I_0(p+1)$ is computed for each flow ϕ_{A_j} (as will be described in Section 5.1.1). Therefore, in order to inject a number of requests (Cond. (2.c_1)) strictly necessary for achieving a certain level of service degradation (Eq. (6)), the intensity $I(t)$ and the number of involved flows ϕ_A are slowly enhanced. Moreover, each burst is a sequence of legitimate messages φ_i, randomly chosen within the set θ (polymorphism in the form), injected with an inter-arrival time t_I that is proportional to the alleged load associated to the injected message.

In the following is described as the SIPDAS-based attacks can be implemented, and how to estimate their effects on the target system.

5.1.1 Attack Approach

In order to implement SIPDAS-based attacks, the following components are involved:

- a *Master* that coordinates the attack;
- π *Agents* that perform the attack (each Agent injects a single flow of messages ϕ_{A_j}); and
- a *Meter* that evaluates the attack effects.

Algorithm 1 describes the approach implemented by each Agent to perform a stealthy service degradation in the cloud computing. It has been specialized for an X-DoS attack. Specifically, the attack is performed by injecting polymorphic bursts of length T with an increasing intensity until the attack is either successful or detected. Each burst is formatted in such a way as to inflict a certain average level of load C_R. In particular, we assume that C_R is proportional to the attack intensity of the flow ϕ_{A_j} during the period T.

Therefore, denote I_0 as the initial intensity of the attack, and assuming $\Delta C_R = \Delta I$ as the increment of the attack intensity. For each attack period, fixed the maximum number of nested tags (*tagThreshold*), the routine *pickRandomTags*(...) randomly returns the number of nested tags n_T for each message (row 4). Based on n_T, the routine *computeInterarrivalTime* uses a specific algorithm (Eq. (12) described in Section 6.3) to compute the inter-arrival time for injecting the next message (row 5). At the end of the period T, if the condition '$attackSuccessful$' is false (evaluated by the Meter and described in the next Section 5.1.2), the attack intensity is increased (row 10). If the condition '$attackSuccessful$' is true, the attack intensity is maintained constant until either the attack is detected (e.g., the target system is no longer reachable due to a reaction performed by the security administrator or by an Intrusion Prevention System), or the auto-scaling mechanism enabled in the cloud adds new cloud resources (row 12). The attack is performed until it is either detected, or the average message rate of the next burst to be injected is greater than δ_T (row 21). In this last case, the Agent notifies to the Master that the maximum average message rate is reached (row 26), and continues to inject messages formatted according to the last level of load C_R reached (row 30).

If all the current Agents reach the threshold δ_T, the Master replaces them with new Agents. Specifically, assuming that at the generic iteration p were active p Agents, the Master replaces them with $p+1$ Agents, with initial attack intensity $I_0(p+1) = C_{R_M}(p)/(p+1)$, where $C_{R_M}(p)$ is the maximum level of intensity achieved by the previous p Agents. In general, the attack starts with a limited number of Agents (e.g., a single Agent) and increased by one unit at each iteration.

5.1.2 Attack Effect Estimation

During the attack, in order to determine if the current flows ϕ_A are generating a service degradation, the Meter injects a flow ϕ_M of requests φ_i overlapped to the attack flows ϕ_A, and estimates the service time t_S to process each message φ_i on the target system. In particular, if we assume that the flow ϕ_M is not limited by a network bottleneck, and the network latency is negligible, then, we can approximate t_S with the response time of the target application. Therefore, during a training phase, the attacker can estimate an approximation of the actual distribution of the response time t_R, for each message of type $\vartheta_k \subset \theta$, and then, uses it to evaluate the service degradation achieved.

Since the actual response time distribution may have a large variance during the attack, the estimation model has to be in charge of identifying significant deviations. Therefore, supposing that $\mu_R(\vartheta_k)$ and $\sigma_R(\vartheta_k)$ are the mean and standard deviation of the response time t_R for the messages type ϑ_k, empirically estimated during the training phase, the Meter can adopt the following $Chebyshev$'s inequality to compute deviation of the service time $t_S(\varphi_i)$ during the attack:

$$p(|t_S(\varphi_i) - \mu_R(\vartheta_k)| \geq \lambda * \sigma_R(\vartheta_k)) \leq \frac{1}{\lambda^2} \quad with \quad \varphi_i \in \vartheta_k. \quad (7)$$

The Chebyshev's inequality establishes an upper bound for the percentage of samples that are more than λ standard deviations away from the population mean. For example, by Eq. (7) can be shown that with $\lambda = 2$, at least 25 percent of the samples would fall outside two standard deviations from the mean. The Chebyshev's inequality can be used to compute an upper limit (an outlier detection value) $\varsigma(\vartheta_k) = \mu_R(\vartheta_k) + \lambda * \sigma_R(\vartheta_k)$ beyond which the sample t_S can be considered to be an outlier. Specifically, we assume that the attack is successful, if starting from a certain time t_0, Eq. (8) is verified for each φ_i:

$$t_S(\varphi_i) > \varsigma(\vartheta_k) \quad with \quad \varphi_i(t) \in \vartheta_k \quad and \quad t > t_0. \quad (8)$$

On the other hand, the service degradation may be due to a temporary user's peak load. Moreover, since even if the attack is successful, a short subset of requests may be not affected by the attack (as will be shown in Fig. 5 of Section 6.2), we can assume that the provided service is degraded whether, given a sequence of messages $\varphi_i \in \phi_M$, $i = 1, \ldots, n$, Eq. (8) has to be verified at least for a large number of service requests φ_i. This assumption can be verified by an approach based on a $count - and - threshold$ mechanism. It relies on a simple γ filtering function defined as follows:

$$\begin{cases} \gamma(0) = 0; & \\ \gamma(i) = \gamma(i-1) + 1 & if\ t_S(\varphi_i) > \varsigma(\vartheta_k); \\ \gamma(i) = \gamma(i-1) * D & if\ t_S(\varphi_i) \leq \varsigma(\vartheta_k); \\ with\ D \in]0,1\ [\ and\ \varphi_i \in \vartheta_k; & \end{cases} \quad (9)$$

where γ is a score that tracks the number of requests that exhibit a t_S degraded according to the condition (8). Specifically, the attack is successful if $\gamma(i)$ exceeds a certain threshold H, which represents the minimum number of consecutive requests that present a t_S sufficiently large to consider the service degraded. D represents the ratio with which $\gamma(i)$ is decreased for each t_S that is comparable with the t_R under normal condition. The values to assign to H and D depend on the required accuracy. In particular, a higher value of H may improve the probability of correctly recognizing the service degradation due to the attack rather than a short workload peak of the user. A higher value of D enhances the weight of requests that are processed with a service time comparable with that estimated during normal operation. Moreover, the values of H and D have to be chosen on the base of the number of requests injected during the attack period T. Assuming that the peak loads due to the user workload are short and impulsive, the attacker should choose a sufficiently large period T, for example, ranging between 10 minutes and one hour. Moreover, we empirically observed that, if the attack is successful, the number of messages unaffected by the attack is low (less than 10 percent of the injected messages). Therefore, we can choose a value D close to 1 (e.g., $D = 0.98$), and a threshold H greter than the 50 percent of the requests injected in the period T).

In summary, according to the proposed approach, the Meter is the component enabled to inject the flow ϕ_M, as well as to compute the filtering function γ used to set the condition $attackSuccessful$ in Algorithm 1. Specifically, at the end of each attack period T, the Meter sets the condition $attackSuccessful$, and the Agent increases the attack intensity whether such a condition is false.

Algorithm 1: Core Algorithm of SIPDAS Agent

Require: $Integer\ timeWindow \Leftarrow T$ {Burst period.}
Require: $Integer\ n_T \Leftarrow 0$ {Nested tags within each message.}
Require: $Integer\ tagThresold \Leftarrow N_T$ {Nested tags threshold.}
Require: $Integer\ rateThreshold \Leftarrow \delta_T$ {Attack rate threshold.}
Require: $Integer\ attackIncrement \Leftarrow \Delta I$ {Attack intensity increment.}
Require: $Integer\ C_R \Leftarrow I_0$ {Initial attack intensity.}
1: **repeat**
2: $t \Leftarrow 0$;
3: **while** $t \leq T$ **do**
4: $n_T \Leftarrow pickRandomTags(tagThresold)$;
5: $t_I \Leftarrow computeInterarrivalTime(C_R, n_T)$;
6: $sendMessage(n_T, t_I)$;
7: $t \Leftarrow t + t_I$;
8: **end while**
9: **if** $!(attackSuccessful)$ **then**
10: $C_R \Leftarrow (C_R + attackIncrement)$; {Attack intensification}
11: **else**
12: **while** $!(attack\_detected)$ **and** $attackSuccessful$ **do**
13: {Service degradation achieved; attack intensity is fixed}
14: $n_T \Leftarrow pickRandomTags(tagThresold)$;
15: $t_I \Leftarrow computeInterarrivalTime(C_R, n_T)$;
16: $sendMessage(n_T, t_I)$;
17: **end while**
18: **end if**
19: $t_{I_{M(C_R)}} = computeInterarrivalTime(C_R, N_T)$;
20: $t_{I_{m(C_R)}} = computeInterarrivalTime(C_R, 1)$;
21: **until** $(2/(t_{I_M} - t_{I_m}) < rateThreshold)$ **and** $!(attack\_detected)$
22: **if** $attack\_detected$ **then**
23: {Notify to the Master that the attack has been detected}
24: **print** $'Attack\_detected'$;
25: **else**
26: {Notify to the Master the attack has reached the threshold δ_T and archived the intensity $C_R = C_{R_M}$}
27: **print** $'Threshold\_reached'$;
28: {Continue the attack by using the previous C_R value}
29: $C_R = C_R - attackIncrement$;
30: **loop**
31: $n_T \Leftarrow pickRandomTags(tagThresold)$;
32: $t_I \Leftarrow computeInterarrivalTime(C_R, n_T)$;
33: $sendMessage(n_T, t_I)$;
34: **end loop**
35: **end if**

However, in order to make stealth the whole attack, even the flow ϕ_M must satisfy Conditions (2). Therefore, the sequence of the messages ϕ_M has to exhibit a polymorphic behavior in terms of message form and over time. In order to generate a polymorphic form, the Meter builds a sequence of legitimate messages $\phi_M(t) = [\varphi_1(t), \ldots, \varphi_m(t)]$, randomly chosen within the set θ. As for the inter-arrival time t_I, the average message rate has to be lower than the message rate of flows ϕ_A, but enough large to make a proper performance assessment of the system under attack (e.g., a request per second). Specially, the requests are injected with an inter-arrival time t_I such that, fixed m random number χ_i sorted in ascending order in the range $[0, T]$:

$$\forall \chi_i \in [0, T[\Rightarrow \varphi_i(t_k + \chi_i) \quad with \quad i = [1, \ldots, m], \chi_i \in \Re, \tag{10}$$

where m is the number of requests sent in the period $T_k = [t_k, t_k + T[$, and $\chi_i < \chi_j$ with $i < j$.

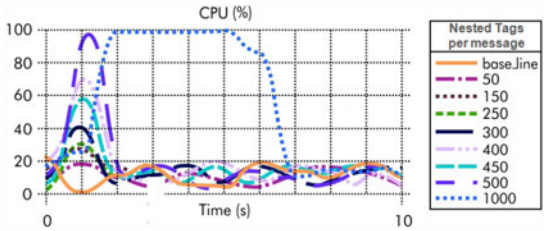

Fig. 3. CPU consumption with different nested XML tags.

6 The Case Study

In order to explain how to leverage a DDoS attack described in Section 3 to implement a SIPDAS pattern, the Deeply-Nested XML attack is adopted as case study. We first analyze the CPU consumption in presence of the classic brute-force X-DoS attack against a simple application server (Section 6.1). Then, the requirements to perform an X-DoS attack in the cloud environment are derived (Section 6.2). Finally, Section 6.3 describes how this simple X-DoS attack can be orchestrated in order to exhibit a polymorphic behavior.

6.1 XML-Based DoS Attack Example

In a previous work [32], we presented the effects of a Deeply-Nested XML attack against a simple web application built on Apache Axis2. The attacked system is a single VM with 2.8 GHz processor, 4 GB of RAM, and Ubuntu 9.10 Linux. During the experimental campaign, we analyzed the CPU consumption depending on the number of nested XML tags and the frequency with which the malicious messages are injected. In particular, in Fig. 3 is shown the CPU consumption on the target system to parse messages containing XML tags with different nested depth. [t] The results showed that a message of 500 nested tags is sufficient to produce a peak of CPU load of about 97 percent, whereas with 1,000 tags the CPU is fully committed to process the message for about 3 seconds. Moreover, we performed several attacks. For each attack, we injected a homogeneous X-DoS flow ϕ_{A_j}, i.e., a sequence of messages with a fixed number of nested tags n_{T_j} and a fixed message rate δ_{A_j}. Assuming that 20 seconds are the maximum time observed experimentally to reach a steady state value of CPU under attack (namely C_R), and denoting *'base_line'* as the average CPU load in absence of user load (about 9 percent). We have empirically estimated a function Y, which represents an approximation of the values C_R reached (after 30 s) by each attack, varying n_T (different inter-arrival time

TABLE 1
Attack Parameters to Achieve an Overload C_R of About 30 Percent

n_{T_j}	5	63	159	182	211	273
t_{I_j}	10 ms	50 ms	100 ms	250 ms	500 ms	1000 ms

$t_I = 1/\delta_A$ are also considered). We can observe that, fixed the inter-arrival time, C_R enhances with a trend that can be approximated as an exponential function $Y(n_T, t_I) = A(t_I) * e^{[n_T * B(t_I)]}$. In particular, in order to simplify the achieved results, assuming that $B(t_I)$ is a steady value equal to ϵ, the computational load on the target system during the attack increases according to the following expressions:

$$\begin{cases} CPU = Base\_line + C_R(n_T, t_I) \\ C_R(n_T, t_I) = A(t_I) * e^{\epsilon n_T} \\ A(t_I) = \alpha t_I^{-\beta}. \end{cases} \quad (11)$$

Finally, we empirically observed that the same level of CPU overload can be achieved with several flows ϕ_{A_j}, orchestrated with different combination of n_T and t_I. For example, in order to obtain an increment of CPU load of about 30 percent with respect to the *base_line*, we can equally use flows ϕ_A orchestrated according to the pair (n_{T_j}, t_{I_j}) reported in Table 1. As will be described in Section 6.3, such consideration can be exploited to build the polymorphic behavior of the SIPDAS attack pattern.

In the second set of experiments, in order to evaluate the effect of a distributed X-DoS attack, we injected concurrent flows ϕ_{A_j} with the same number of nested tags n_T and inter-arrival time t_I. Empirical results showed that C_R increases linearly with respect to the number k_F of concurrent flows, such that $CPU = Base\_line + (m * k_F)$ with the slope $m < 1$.

6.2 Creating X-DoS Based Degradation in Cloud Computing

In this section, we analyze the effects of an X-DoS attack on the cloud system in terms of provided quality of service. The mOSAIC-based testbed designed to test the attack effectiveness consists of two independent cloud applications, which represent the 'Attacker' and the target application server, namely SUA. Both applications are designed according to the mOSAIC paradigm and run on independent set of VMs. The SUA application emulates a typical XML-based web application: it receives the XML messages via HTTP, and performs the XML parsing (using a DOM approach), and other simple elaborations (e.g., it computes the total number of nested tags). The mOSAIC implementation of such a scenario is represented in Fig. 4: a *HTTPgw* Cloudlet manages the HTTP messages and forwards them to the *XML Analyzer*, which parses the XML document, and stores the results (in a Key-Value store). The advantage of using mOSAIC is that, we are able to automatically scale the application when the virtual node is overloaded (starting new VMs). Moreover, thanks to the mOSAIC monitoring tools [11], we are able to evaluate the resource consumption of each involved VM and the number of retrieved messages (XML documents) to be processed.

In [31], we adopted the TPC Benchmark W (TPC-W) is adopted in order to assess the effectiveness of the X-DoS attack with respect to the provided quality of service [33]. It is a transactional web benchmark used to simulate the activities of a business oriented transactional web server (i.e., the execution of multiple transaction types that span a breadth of complexity). Each web interaction involves a different computational cost, and it is subjected to a response time constraint. The performance metric reported by TPC-W is the number of web interactions processed per second, namely WIPS. In this paper, the target application is an ad-hoc web service, therefore, we cannot use the TPC-W workload. In order to perform similar evaluations, we have built a TPC-W emulator, which generates a similar workload for the SUA application. Moreover, it measures the number of operations processed per second (WIPS), and retransmits the same request to the application when it is not processed before a fixed timeout. It should be clear that this testbed does not aim at correctly emulating the TPC-W benchmark. It is only used as a basis to build up a generic workload, under which we are able to analyze the attack effect.

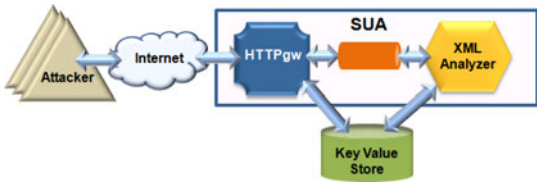

Fig. 4. Architecture of the mOSAIC-based testbed.

We perform a dedicated testing campaign, by injecting flows ϕ_{A_j} with a fixed number of nested tags n_{T_j} and a fixed message rate δ_{A_j}. In Fig. 5 is shown the effect of a flow of messages with 350 nested tags every 100 ms. Specifically, it represents a qualitative analysis of the WIPS variation with respect to the time, during several time windows. The first window shows the WIPS variation in absence of the attack (i.e., in normal operation the considered testbed processes about 190 transaction per second). In the second time window, the X-DoS attack is activated (at time t_0). After a few seconds from t_0, the system is completely overloaded (100 percent CPU), and the number of interactions processed become very low (about 17 per second). At time t_1, the attack is intensified, in particular, the message rate is increased from 100 to 10 ms, which further reducing the number of transactions processed (about 9 per second). Finally, at time t_2, we forced the mOSAIC auto-scaling

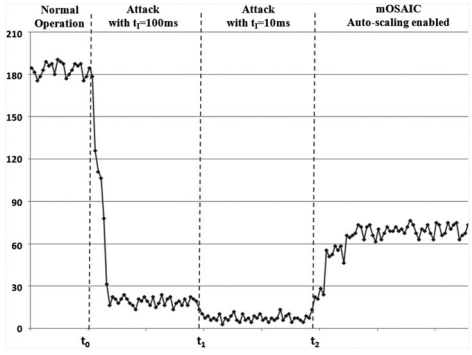

Fig. 5. WIPS evaluation during an X-DoS (with 350 nested XML tags) against a cloud application.

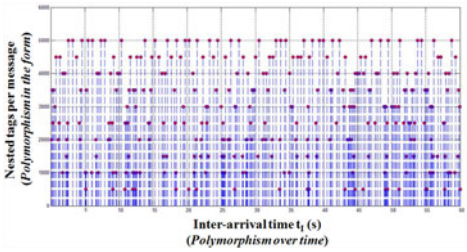

Fig. 6. SIPDAS burst.

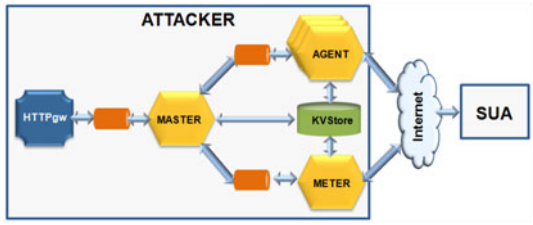

Fig. 7. Attacker implementation by the mOSAIC paradigm.

mechanism to add three new VMs, on which the XML Analyzer instances are enabled; thus, the number of processed requests progressively increases from 9 to 71. However, since the attack is already in progress, the provided service still results degraded compared with the normal operation.

Such results show that, (i) if the X-DoS attack is overlapped on the user traffic, although the attack achieves a substantial service degradation, a subset of user's requests is either unaffected, or at most processed in a longer time; and (ii) once the system has reached a certain level of service degradation, the attacker has to greatly increase the attack potency (i.e., the request-rate and/or number of concurrent sources), in order to produce a slight additional damage. Therefore, on the basis of the considerations described in Section 5.1, it is necessary to identify the maximum attack intensity I_M that the attacker can inflict to the target system, while respecting Cond. (2).

6.3 SIPDAS-Based Attack Burst

In Fig. 6 is shown an example of a SIPDAS-based X-DoS attack burst, in which the number of nested tags n_T and the inter-arrival time for each injected requests $\varphi_i \in \theta$ have been chosen according to a uniform distribution. Specifically, supposing that N_T and δ_T are the maximum number of tags that can be nested and the maximum request rate respectively (empirically estimated as described in Section 4.4), and assuming that the level of CPU load increases according to the trend described by Eq. (11):

$$C_R(n_T, t_I) = \alpha t_I^{-\beta} * e^{(\epsilon n_T)} \Rightarrow t_I = \sqrt[\beta]{\frac{\alpha e^{(\epsilon n_T)}}{C_R}}, \tag{12}$$

fixing the value C_R (the increment of CPU load inflicted by the attack), the burst is implemented by orchestrating a sequence of messages, in which each message includes a random number of nested tags $n_T \leq N_T$, injected with an inter-arrival time t_I computed by Eq. (12) and included within the range $[t_{I_m}(C_R), t_{I_M}(C_R)]$:

$$t_{I_m}(C_R) = t_I(1, C_R) \leq t_I(n_T, C_R) \leq t_I(N_T, C_R) = t_{I_M}(C_R), \tag{13}$$

$$with \begin{cases} t_I(1, C_R) = \frac{\sqrt[\beta]{\alpha e^{\epsilon}}}{C_R}, \\ t_I(N_T, C_R) = \frac{\sqrt[\beta]{\alpha e^{\epsilon N_T}}}{C_R}. \end{cases} \tag{14}$$

Clearly, we cannot expect that the real CPU load on the target system will exactly follow the function described by Eq. (11). Nevertheless, such a model may be used to identify the sequence of pairs $[n_T, t_I(n_T)]$, which can be used to build the polymorphic behavior of the burst (that produces a load C_R on the target system).

Moreover, each burst is a sequence of requests $\varphi_i(n_T)$ (with n_T equally distributed in the range $[1, N_T]$), which exhibits an average message rate equal to:

$$\delta_A(C_R) = \frac{1}{[t_{I_M}(C_R) - t_{I_m}(C_R)]/2}. \tag{15}$$

Therefore, the maximum attack intensity I_M (beyond which the flow ϕ_A may not be more stealth) is bounded by the condition $\delta_A(C_R) < \delta_T$.

7 Attack Evaluation

In this section, first we present an example of attack implemented by using the paradigm offered by the mOSAIC framework. Then, we study the impact of the SIPDAS-based attack pattern on the quality of service provided by the target application, by varying the number of the involved Agents and the resources of the application server.

7.1 Attack Implementation

The implementation of a SIPDAS-based attack can be done in several ways. In this work, we use the same cloud framework adopted for building up the target server application SUA (described in Section 6.2). As a result, the implemented attack can be offered *as a services* through a simple web interface. Fig. 7 shows the architecture of the attacker application in terms of the mOSAIC Cloudlets. A web interface is used to setup the attack parameters and observe the status of the attack. When the attack is activated by the web interface, a set of parameters is sent to the Master, including the target system URL, the attack intensity I_0, the attack increment ΔI, the thresholds N_T and δ_T (e.g., the maximum number of nested tags and service request rate), and the attack period T. The Master coordinates the attack, by enabling the Agent instances, and interacting with the Meter that performs legitimate requests to the server under attack, and differently from the Agents, evaluates the response time t_S. The KV store shared among the Cloudlets, maintains all the information related to the attack state, including the parameters used by the Agents and the attack results (in terms of reached service degradation) evaluated by the Meter. The Master periodically acquires information from the 'KV store', and sends messages to Agents in order to update their actions, according to the attack strategy described in Section 5.1.1.

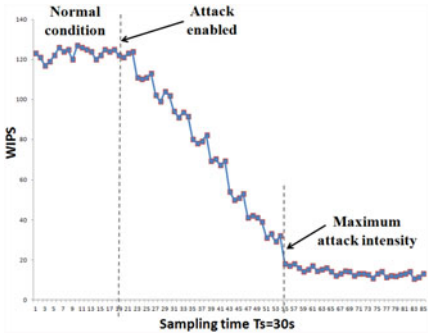

Fig. 8. SIPDAS effect (with a single Agent) on the mOSAIC-based application running on a single VM (auto-scaling disabled).

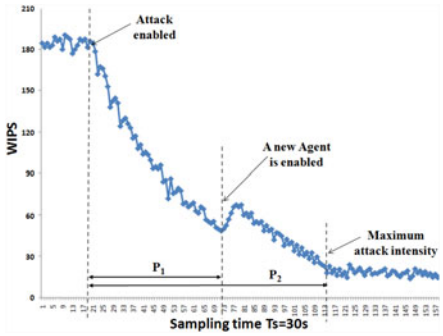

Fig. 9. SIPDAS effect (with two Agents) on the mOSAIC-based application running on two VMs (auto-scaling disabled).

7.2 Experimental Evaluation

In the following experiments, we assume that during the normal operation the target application SUA runs on a certain number of VMs (with 2 CPU x86, 32 bit, 2.0 Ghz with 1 GB of memory) in a mOSAIC-based private cloud. The auto-scaling mechanism is enabled by the mOSAIC Platform when the average CPU load on the involved VMs exceeds the 90 percent for a time period greater than 10 minutes. Moreover, we adopt the developed TPC-W emulator (described in Section 6.2) both to simulate the customer workload and to evaluate the attack effect. The TCP-W emulator and the attacker application are deployed on different VMs and connected to the target cloud through a private network (100 Mb/s Ethernet LAN). The settings for the attack evaluation are selected as following: $I_0 = 10$ (initial attack intensity), $\Delta I = 10$ (attack intensity increment), and the maximum number of nested tags $N_T = 40$ (we assume that it is imposed by a prevention mechanism based on a validation schema). Moreover, in order to achieve a small evaluation time, the attack period is chosen to be $T = 120\,s$. Finally, during the first two experiments the mOSAIC auto scaling mechanism is disabled.

During the first experiment, we evaluate the maximum message rate δ_T necessary to inflict a substantial service degradation. According to the filtering function γ described by Eq. (9) (with $D = 0.98$ and $\lambda = 3$), we assume that the attack is successful if $t_s(\varphi_i) > \mu_R + 3\sigma_R$ for a number of consecutive service requests greater that $H = 60$. In order to show the attack effects, Fig. 8 shows the WIPS variation with respect to the time, achieved with a single Agent against SUA deployed on a single VM on the server side. In order to make more clear the achieved results, the WIPS values are aggregated at a fixed time interval $T_S = 30\,s$ and the average value is shown. Experimental results show that are sufficient about nine attack periods (i.e., about $t = 9 * T = 18$ minutes) to satisfy Eq. (7), as well as to achieve a service degradation greater than 90 percent. The smallest reached inter-arrival time between two consecutive message (in the attack sequence) is $t_I = 26\,\mathrm{ms}$, whereas the average value is $t_I = 73\,ms$.

In the second experiment, we set the threshold δ_T to the average value t_I reached during the previous experiment ($\delta_T = 1/73\,ms$). Fig. 9 shows the WIPS variation achieved when the SUA is deployed on two VMs. Results show that a single Agent is not able to inflict a significant service degradation. Specifically, at sample $\#73$ the Agent reaches the maximum achievable attack intensity C_{R_M} with the fixed δ_T (after a period $P_1 = 50 * T_S$ from the attack activation). At this point, the Master enables another Agent and sets a new initial attack intensity of the two Agents to $I_0 = C_{R_M}/2$. As Fig. 9 shows, with two Agents the maximum service degradation is achieved after a time period $P_2 = 96 * T_S = 48$ minutes.

In the third experiment, the mOSAIC auto-scaling mechanism is enabled. We assume that in normal conditions the target application runs on two VMs, whereas in case of overloading due to a workload peak, the auto-scaling mechanism can incrementally add other five VMs. Experimental results show that after about 3 hours the attack inflicts the maximum service degradation with five Agents.

8 Validation

In order to validate the proposed stealthy strategy, we analyze the traffic distribution exhibited by the presented attack pattern.

Recall that, the detection methods proposed in the literature assume the main anomaly incurred during a low-rate attack is that, the incoming service requests fluctuate in a more extreme manner during an attack (it is due to a huge number of messages periodically injected to the normal traffic in a very slow pace), as well as the incoming traffic volume suffers a fast decline. On the other hand, according to the proposed SIPDAS strategy, the overall incoming traffic volume may or may not decline during the attack, because the attack flows are legitimate message sequences $\varphi_i \in \theta$ injected with a rate $\delta_A < \delta_T$, comparable with the normal customer's flows, as well as it does not exhibit an impulsive pattern.

In order to prove that such kind of methods cannot be applied to detect SIPDAS attack pattern, we exploit a recent work that proposes real-time detection approach for low-rate DDoS attacks. Specifically, Lui [27] propose to compute the distribution of the flow connection entropy (FCE) series to obtain a coarse-grained estimation of the message traffic, and then use the time-series decomposition method to divide the FCE series into a '*trend*' component and a '*steady random*' component. Authors propose to analyze such components to detect the anomalies of the long-term and short-term trends in the message traffic, caused by the low-rate DDoS attacks.

The FCE of a set of incoming flows is defined as:

$$FCE = -\sum_{j=1}^{n} p(f_i) log_2 p(f_i), \tag{16}$$

where $p(f_i)$ is the probability of receiving a message belonging to the flow f_i. According to a study conducted by Guang et al. [34], a DDoS attack produces an abnormal increase in the FCE of the message traffic. Generally, a long time-scale traffic exhibits the characteristics of trend, period, mutation, and randomness. The first two components belong to the long-term changes, which represent the smooth process of message traffic behavior, whereas the other two components belong to the short-term changes, reflecting uncertainty in the message traffic. As for the short period of time, we may assume that the mutation component and the period component are negligible with the proposed attack pattern, i.e., the messages traffic within a short window can be considered as the consequence of the interaction between the trend component and the random component only. This assumption simplifies the real-time detection approach. Under this assumption, the FCE series is modeled as:

$$\begin{cases} FCE_i = \overline{FCE_i} + R_i; \\ R_i = \eta_i + \epsilon_i; \end{cases} \tag{17}$$

where $\overline{FCE_i}$ denotes the trend component, R_i is the random component, η_i represents the steady random component, and ϵ_i represents the measurement error (white noise).

To estimate the trend component, an exponentially weighted moving average can be adopted. The estimation of FCE is obtained by combining the current FCE with the FCE from the previous period corrected for trend as follows:

$$\begin{cases} \overline{FCE_i} = \kappa_i FCE_i + (1-\kappa_i)\overline{FCE_i}, \\ \kappa_i = \kappa_{max}(1 - e^{-Q\mu_i}), \\ \mu_i = \frac{|FCE_i - FCE_{i-1}|}{FCE_{i-1}}, \end{cases} \tag{18}$$

where μ_i reflects the extent of the fluctuation. Generally, if FCE series are subject to large changes, then the proportion of the history should be small so as to quickly attenuate the effect of the old observations. If FCE series smoothly change, the proportion of the history should be large to minimize random variations. The value of the parameter Q can be determined empirically through experiments. Finally, subtracting the long-term trend component from the original series, the remainder of the series can be considered as the random part of the traffic $R_i = FCE_i - \overline{FCE_i}$. Therefore, given a series of FCE data, we first compute the trend and random components (the trend component will reflect the majority of slowly-increasing signal, whereas the steady random signal is contained in the random component), then, specific anomaly detection methods are applied to each component separately. In particular, we apply the non-parametric CUSUM method to the random component [34]. It can be used to determine whether the observed time series are statistically homogeneous, and to identify the time when the changes occur. The basic idea of CUSUM is to accumulate those small fluctuations during the detection process to amplify the varying statistical feature, and thus, improve the detection sensitivity. It can detect a small deviation of the mean effectively. As for the trend component,

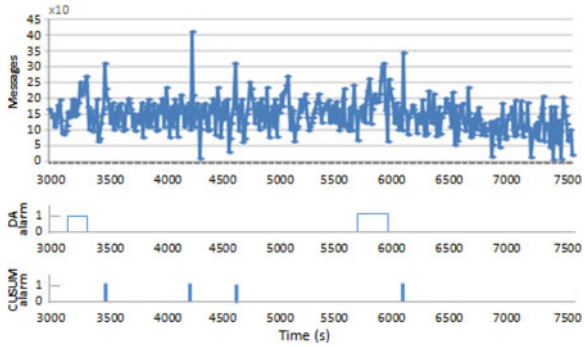

Fig. 10. Detection results achieved with FCE series (auto scaling disabled).

we calculate the double auto-correlation (DA) coefficient series and examine the first N_{max} elements in such series [35]. If the behavior in the trend component has an evident increasing or declining tendency, then those N_{max} values will all exceed a certain threshold:

$$\rho'_k(i) > T_H, \; with \; 2 \leq k \leq N_{max}, \tag{19}$$

where $\rho'_k(i)$ is the DA coefficient series at phase i. Generally, since the window size used in the detection algorithm is small (for attack latency reduction), the normal trend series should exhibit a little fluctuation.

During the experimental campaign, we evaluated the accuracy and latency of the detection approach based on FCS series. In order to generate the normal traffic, we use actual HTTP traffic collected by a web server of our university. The traffic we tested is synthetically generated by merging the attack traffic and the normal traffic. The default setting for the adopted parameters are the same used in [35]: $\Delta t = 5\,s$ for sampling FCE series, $\kappa_i = 0.4$, $Q = 10$ for time series decomposition, $N_{max} = 6$, and $T_H = 0.4$. Generally, the size of the detection sliding window W should be kept small (e.g., $W = 10$ minutes) to adapt to real-time detection requirements. Moreover, with larger window size, more memory resources will be consumed [34].

During the first experiment, we consider the second attack scenario of Section 7.2 ($T = 20$ s, auto scaling disabled, two VMs client-side). We analyze separately the two components to detect long-term and short-term anomalous changes of the incoming message traffic. We used alarm series to represent the detecting results, in which the value is 1 when an anomaly is detected and 0 otherwise. As Fig. 10 shows, several anomalous behaviors are observed from the FCE series, including short increasing tendencies and random components. However, all the alarms are false positives due to short outbursts or increasing-intensity of the normal traffic. The attack traffic is injected with a very slow-increasing trend, which is not amplified from the background traffic. It does not exhibit sudden fluctuations. Moreover, the incoming traffic volume declines very slowly. By finding a better combination of FCE parameter values, this last phenomena could be detected, but at the expense of the detection accuracy (with an increase of false positives of about 270 percent in the considered example). Moreover, recall that, the increasing trend of the attack potency depends from the number of involved flows and the attack period; even if the victim detects the attack, the malicious

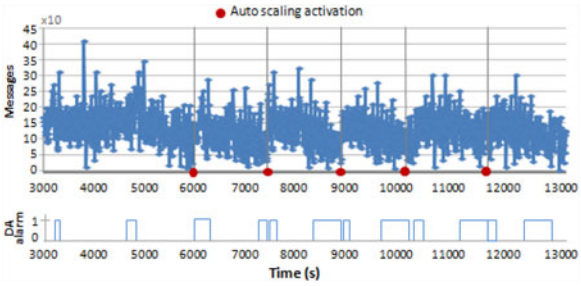

Fig. 11. Detection results achieved with FCE series (auto scaling enabled).

process can be re-initiate (polymorphic over time). In particular, enlarging suitably the attack period (T greater than the detection sliding window W), the incoming traffic volume exhibits a slow-decreasing trend, such that the FCE series do not exhibit fluctuation as time goes.

In the second experiment, we consider the third attack scenario of Section 7.2 ($T = 20$ s, auto scaling enabled, seven VMs client-side). Fig. 11 shows that anomalous trends are observed before and after the system scales up. Specifically, when an VM is added, the incoming traffic volume increases quickly, which is an obvious consequence of the improvement in overall system performance. Then, since the attack is already in progress, the performance decreases quickly again. However, even in this case, experimental results have shown that, by using a larger attack period T (greater than 10 minutes), this second phenomenon cannot be detected by the FCE series. Finally, the attacker could slowly increase the attack potency in order to induce the system to scale up, but without deny the service, or however, excessively degrading the response time. Specifically, by using the Meter, the attacker can check the reached service degradation (estimated by the function described in Section 5.1.2). As Fig. 11 shows, when the system scales up, the system performance tends to improve. Therefore, once the system is scaled up, the attacker can fix the attack potency to the value achieved, without further overloading the system. In this way, the attacker can inflict a significant financial cost, limiting the response time degradation.

9 Host-Based Countermeasures

This section presents some considerations about host-based countermeasures against the presented attack strategy. In particular, several recent works have proposed host-based solutions for virtual environment application monitoring of cloud resources utilization [36], [37]. However, they claim that, the proposed detection techniques, which adopt predefined target performance measures and empirically derived thresholds, require a good understanding of the target application. In a dynamic environment such as the cloud, it is difficult to identify which resources (such as memory, computing and communication units, server thread pools, database connections, locks, queues, hash tables, etc.) and measures (e.g., response time, CPU utilization, etc.) need to be monitored and how to set accurate threshold values for them [38]. The particular resources and measures to monitor depend on the nature of the applications, their workload, and the their potential vulnerability. The model-based techniques also face a big challenge in modeling the highly dynamic of the cloud application behavior. Another drawback of host-based techniques is that monitoring agents have to be deployed on the target VMs, along with a need for a scalable monitoring infrastructure, which represent an additional real cost for the victims (in the cloud). Moreover, collecting several measures frequently in real time, from all the involved virtual nodes, can be costly.

Regarding the X-DoS attack used as case study, it is an exhaustion attack that inflicts a sustained consumption of computational resources. A possible detection method could consist in monitoring the overall CPU consumption of the target application deployed over the distributed VMs. We adopt a window-based state monitoring that triggers state alerts only when the normal state is continuously violated for a time window W [39]. W is essentially the tolerable time of abnormal state, which must be established by the service provider. Given the threshold T_R, the size W of the monitoring window, and n monitored VMs with CPU values $C_i(t)$ at time t, an alert is triggered only when $\sum_{i=1}^{n} C_i(t-j) > T_R$, $\forall j \in [0, W[$ at any t. In order to validate the adopted technique, we assume that the alarm must be triggered when the overall CPU load on the involved VMs exceeds 80 percent for a time window $W = 10$ minutes. Thus, if we consider the attack scenario of the second experiment of Section 7.2 (in which two VMs are involved and the auto scaling mechanism is disabled), the attack is detected after 41 minutes (detection latency). In the second experiment, we consider the third attack scenario of Section 7.2 (in which two VMs run in normal condition, and five VMs are added incrementally by the auto scaling mechanism). In order to set the auto scaling mechanism, we were inspired by the policy for the auto scaling activation suggested for Amazon Cloud [40]. We assume that the system scales up by one VM, if CPU utilization is greater than 60 percent for 5 minutes (we call this time 'scaling time'). Based on this setting, the attack is detected only after all the additional VMs have been enabled, which is about 137 minutes from the start of the attack.

The experimental results have shown that the adopted detection technique may be more accurate than the traffic-based technique presented in Section 7.2, although with a longer detection latency. On the other hand, the fixed thresholds can be identified and exploited by sophisticated attackers in order to implement a SIPDAS-based attack pattern. The proposed strategy allows to define a sequence of bursts which, while respecting the fixed thresholds, inflicts a system load that is considered normal by the detection technique, and at the same time consuming more resources possible. In particular, the attacker can gradually increase the attack potency, forcing the cloud service to scaling up, until reaching the maximum computational capacity (all available VMs have been enabled). At this point, the attacker can slightly reduce the attack intensity until the system scales down (e.g., by a single VM). If this process is repeated in an iterative manner, with a period smaller than the detection time window W and an attack period T greater than the scaling time, the overall CPU consumption can be kept below the detection threshold fixed to 80 percent (none alarm is triggered).

Based on the performed analysis, a possible approach to detect SIPDAS should correlate both host- and traffic-based information, such as an excessive sustained consumption of

computational resources, with no corresponding increase in incoming traffic [41]. However, the proposed attack strategy is generic enough to be applied to several kind of attacks that leverage known application vulnerabilities. Therefore, even if the victim detects the attack, the attack process can be re-initiate by exploiting a different application vulnerability (polymorphism in the form), or a different timing (polymorphism over time), in order to inflict a prolonged consumption of resources.

10 Conclusions

In this paper, we propose a strategy to implement stealthy attack patterns, which exhibit a slowly-increasing polymorphic behavior that can evade, or however, greatly delay the techniques proposed in the literature to detect low-rate attacks. Exploiting a vulnerability of the target application, a patient and intelligent attacker can orchestrate sophisticated flows of messages, indistinguishable from legitimate service requests. In particular, the proposed attack pattern, instead of aiming at making the service unavailable, it aims at exploiting the cloud flexibility, forcing the services to scale up and consume more resources than needed, affecting the cloud customer more on financial aspects than on the service availability.

In the future work, we aim at extending the approach to a larger set of application level vulnerabilities, as well as defining a sophisticated method able to detect SIPDAS-based attacks in the cloud computing environment.

Appendiex A
Table of Notions

Notions	Meaning
ζ_M	comprehensive system capacity
ω_i	computational load of a service request
C_N	normal system load
θ	legitimate service request set
ϑ_i	service request type
P_A	attack profit
C_A	attack resource consumption
C_R	desired level of load
$\phi_{A(N)}$	attack (normal) flow
π	number of concurrent flows
φ_i	service request
n_T	number of nested tags per message
g	profit of a malicious service request
δ	average rate of flow ϕ
t_I, t_S	inter-arrival time and service time
B	queue size
T, I	attack period and intensity
δ_T, N_T	attack thresholds (rate and nested tags)
p	attack iteration
W	detection time window

Acknowledgements

This research was partially supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under Grant Agreements no. 610795 (SPECS), and the MIUR under Project PON02 00485 3487784 "DISPLAY" of the public private laboratory "COSMIC" (PON02 00669).

References

[1] M. C. Mont, K. McCorry, N. Papanikolaou, and S. Pearson, "Security and privacy governance in cloud computing via SLAS and a policy orchestration service," in *Proc. 2nd Int. Conf. Cloud Comput. Serv. Sci.*, 2012, pp. 670–674.

[2] F. Cheng and C. Meinel, "Intrusion Detection in the Cloud," in *Proc. IEEE Int. Conf. Dependable, Autonom. Secure Comput.*, Dec. 2009, pp. 729–734.

[3] C. Metz. (2009, Oct.). DDoS attack rains down on Amazon Cloud [Online]. Available: http://www.theregister.co.uk/2009/10/05/amazon_bitbucket_outage/S

[4] K. Lu, D. Wu, J. Fan, S. Todorovic, and A. Nucci, "Robust and efficient detection of DDoS attacks for large-scale internet," *Comput. Netw.*, vol. 51, no. 18, pp. 5036–5056, 2007.

[5] H. Sun, J. C. S. Lui, and D. K. Yau, "Defending against low-rate TCP attacks: Dynamic detection and protection," in *Proc. 12th IEEE Int. Conf. Netw. Protocol.*, 2004, pp. 196-205.

[6] A. Kuzmanovic and E. W. Knightly, "Low-rate TCP-Targeted denial of service attacks: The shrew vs. the mice and elephants," in *Proc. Int. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2003, pp. 75–86.

[7] M. Guirguis, A. Bestavros, I. Matta, and Y. Zhang, "Reduction of quality (RoQ) attacks on internet end-systems," in *Proc. IEEE Int. Conf. Comput. Commun.*, Mar. 2005, pp. 1362–1372.

[8] X. Xu, X. Guo, and S. Zhu, "A queuing analysis for low-rate DoS attacks against application servers," in *Proc. IEEE Int. Conf. Wireless Commun., Netw. Inf. Security*, 2010, pp. 500–504.

[9] L. Wang, Z. Li, Y. Chen, Z. Fu, and X. Li, "Thwarting zero-day polymorphic worms with network-level length-based signature generation," *IEEE/ACM Trans. Netw.*, vol. 18, no. 1, pp. 53–66, Feb. 2010.

[10] A. Chonka, Y. Xiang, W. Zhou, and A. Bonti, "Cloud security defense to protect cloud computing against HTTP-DOS and XML-DoS attacks," *J. Netw. Comput. Appl.*, vol. 34, no. 4, pp. 1097–1107, Jul. 2011.

[11] D. Petcu, C. Craciun, M. Neagul, S. Panica, B. Di Martino, S. Venticinque, M. Rak, and R. Aversa, "Architecturing a sky computing platform," in *Proc. Int. Conf. Towards Serv.-Based Int.*, 2011, vol. 6569, pp. 1-13.

[12] U. Ben-Porat, A. Bremler-Barr, and H. Levy, "Evaluating the vulnerability of network mechanisms to sophisticated DDoS attacks," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2008, pp. 2297–2305.

[13] S. Antonatos, M. Locasto, S. Sidiroglou, A. D. Keromytis, and E. Markatos, "Defending against next generation through network/endpoint collaboration and interaction," in *Proc. IEEE 3rd Eur. Int. Conf. Comput. Netw. Defense*, 2008, vol. 30, pp. 131–141.

[14] R. Smith, C. Estan, and S. Jha, "Backtracking algorithmic complexity attacks against a NIDS," in *Proc. Annu. Comput. Security Appl. Conf.*, Dec. 2006, pp. 89–98.

[15] C. Castelluccia, E. Mykletun, and G. Tsudik, "Improving secure server performance by re-balancing SSL/TLS handshakes," in *Proc. ACM Symp. Inf.*, Apr. 2005, pp. 26–34.

[16] Y. Zhang, Z. M. Mao, and J. Wang, "Low-rate TCP-targeted DoS attack disrupts internet routing," in *Proc. 14th Netw. Distrib. Syst. Security Symp.*, Feb. 2007, pp. 1–15.

[17] U. Ben-Porat, A. Bremler-Barr, and H. Levy, "On the exploitation of CDF based wireless scheduling," in *Proc. IEEE Int. Conf. Comput. Commun.*, Apr. 2009, pp. 2821–2825.

[18] A. Kuzmanovic and E. W. Knightly, "Low-rate TCP-targeted denial of service attacks and counter strategies," *IEEE/ACM Trans. Netw.*, vol. 14, no. 4, pp. 683–696, Aug. 2006.

[19] S. Ebrahimi-Taghizadeh, A. Helmy, and S. Gupta, "TCP vs. TCP: A systematic study of adverse impact of short-lived TCP flows on long lived TCP flows," in *Proc. IEEE Int. Conf. Comput. Commun.*, Mar. 2005, pp. 926–937.

[20] M. Guirguis, A. Bestavros, I. Matta, and Y. Zhang, "Reduction of quality (RoQ) attacks on dynamic load balancers: Vulnerability assessment and design tradeoffs," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2007, pp. 857–865.

[21] Y. Xuan, I. Shin, M. T. Thai, and T. Znati, "Detecting application denial-of-service attacks: A group-testing-based approach," *IEEE Tran. Parallel Distrib. Syst.*, vol. 21, no. 8, pp. 1203–1216, Aug. 2010.

[22] V. Durcekova, L. Schwartz, and N. Shahmehri, "Sophisticated denial of service attacks aimed at application layer," in *Proc. 9th Int. Conf. ELEKTRO*, 2012, pp. 55–60.

[23] G. Macia-Fernandez, E. Diaz-Verdejo, and P. Garcia-Teodoro, "LoRDAS: A low-rate DoS attack against application servers," in *Proc. 2nd Int. Conf. Critical Inf. Infrastruct. Security*, 2008, vol. 5141, pp. 197–209.
[24] G. Macia-Fernandez, J. E. Diaz-Verdejo, and P. Garcia-Teodoro, "Mathematical model for low-rate DoS attacks against application server," *IEEE Trans. Inf. Forensics Security*, vol. 4, no. 3, pp. 519–529, Sep. 2009.
[25] X. Luo and R. K. Chang, "On a new class of pulsing denial-of-service attacks and the defense," in *Proc. Netw. Distrib. Syst. Security Symp.*, Feb. 2005, pp. 61–79.
[26] Y. Chen and K. Hwang, "Collaborative detection and filtering of shrew DDoS attacks using spectral analysis," *J. Parallel Distrib. Comput.*, vol. 66, no. 9, pp. 1137–1151, Sep. 2006.
[27] H. Liu. "Real-time detection of stealthy ddos attacks using time-series decomposition," in *Proc. Int. Conf. Commun.*, 2010, pp. 1–6.
[28] A. Jumratjaroenvanit and Y. Teng-amnuay, "Probability of attack based on system vulnerability life cycle," in *Proc. IEEE Int. Conf. Electron. Commerce Security*, Aug. 2008, pp. 531–535.
[29] Amazon EC2—Auto Scaling Feature. Giu. (2012) [Online]. Available: http://aws.amazon.com/autoscaling/
[30] M. Jensen, N. Gruschka, and R. Herkenh, "A survey of attacks on web services," *Comput. Sci.*, vol. 24, no. 4, pp. 185–197, 2009.
[31] M. Ficco and M. Rak, "Intrusion tolerance of stealth DoS attacks to web services," in *Proc. Int. Conf. Inf. Security Privacy*, 2012, vol. 376, pp. 579–584.
[32] M. Ficco and M. Rak, "Intrusion tolerant approach for denial of service attacks to web services," in *Proc. IEEE Int. Conf. Data Compression, Commun. Process.*, Jun. 2011, pp. 285–292.
[33] TPC Benchmark W (TPC-W). A transactional web benchmark. (2013) [Online]. Available: at http://www.tpc.org/tpcw/
[34] C. Guang, G. Jian, and D. Wei, "A time-series decomposed model of network traffic," in *Proc. 1st Int. Conf. Adv. Natural Comput.*, 2005, pp. 338–345.
[35] B. Brodsky and B. Darkhovsky, *Nonparametric Methods in Change-point Problems*. Norwell, MA, USA: Kluwer, 1993.
[36] Y. Mei, L. Liu, and X. Pu, "Performance analysis of network I/O workloads in virtualized data centers," *IEEE Trans. Services Comput.*, vol. 6, no. 1, pp. 48–63, Jan. 2013.
[37] C. C Thimmarayappa and A. Jayadharmarajan, "VMON-virtual environment application monitoring," in *Proc. Int. Conf. Adv. Comput., Commun. Inf.*, 2013, pp. 1936–1941.
[38] H. Wu, A. N. Tantawi, and T. Yu, "A self-optimizing workload management solution for cloud applications," in *Proc. IEEE 20th Int. Conf. Web Serv.*, 2013, pp. 483–490.
[39] S. Meng, T. Wang, and L. Liu, "Monitoring continuous state violation in datacenters: Exploring the time dimension," in *Proc. IEEE 26th Int. Conf. Data Eng.*, 2010, pp. 968–979.
[40] Auto Scaling in the Amazon Cloud. (2012, Jan.) [Online]. Available: http://techblog.netflix.com/2012/01/auto-scaling-in-amazoncloud. html
[41] M. Ficco, "Security event correlation approach for cloud computing," *Int. J. High Perform. Comput. Netw.*, vol. 7, no. 3, pp. 173–185, 2013.

Massimo Ficco received the degree in computer engineering from the University 'Federico II' in 2000, and the PhD degree in information engineering from the University of Parthenope. He is an assistant professor at the Department of Industrial and Information Engineering of the Second University of Naples (SUN). From 2000 to 2010, he was a senior researcher at the Italian University Consortium for Computer Science. His current research interests include software engineering architecture, security aspects, and mobile computing. He has been involved in several EU funded research projects in the area of security and reliability of critical infrastructures.

Massimiliano Rak received the degree in computer science engineering at the University of Naples Federico II in 1999. In November 2002, he received the PhD degree in electronical engineering at Second University of Naples. He is an assistant professor at Second University of Naples. His scientific activity is mainly focused on the analysis and design of high performance system architectures and on methodologies and techniques for distributed software development. He has been involved in several EU funded research projects in the area of cloud computing.