SARDOS: Self-Adaptive Reaction against Denial of Service in the Internet of Things

Marco Tiloca
RISE Research Institutes of Sweden
Isafjordsgatan 22, Kista, Sweden
Email: marco.tiloca@ri.se

Rikard Höglund
RISE Research Institutes of Sweden
Isafjordsgatan 22, Kista, Sweden
Email: rikard.hoglund@ri.se

Syafiq Al Atiiq
KTH Royal Institute of Technology
Isafjordsgatan 22, Kista, Sweden
Email: atiiq@kth.se

***Abstract*—Denial of Service (DoS) is a common and severe security issue in computer networks. Typical DoS attacks overload servers with bogus requests, induce them to worthlessly commit resources, and even make them unable to serve legitimate clients. This is especially relevant in Internet of Things scenarios, where servers are particularly exposed and often equipped with limited resources. Although most countermeasures focus on detection and mitigation, they do not react to dynamically adapt victims' behavior, while at the same time preserving service availability. This paper presents SARDOS, a reactive security service that leverages detection mechanisms from different communication layers, and adaptively changes the operative behavior of victim servers while preserving service availability. We experimentally evaluated SARDOS with a prototype implementation running on an underclocked Raspberry Pi server. Our results show that, when running SARDOS, a server under attack displays considerably lower memory and CPU usage, while still ensuring (best-effort) fulfillment of legitimate requests.**

***Index Terms*—Security, Denial of Service, Internet of Things.**

I. Introduction

Internet of Things (IoT) refers to a network scenario with billions of hosts interconnected and available over the Internet [1][2]. This includes huge deployments of everyday objects and constrained devices with limited resources, e.g. sensors and actuators. The IoT concept is fostering several Internet-oriented use cases, such as smart home and smart building, factory automation, e-health and environmental monitoring.

The peculiar nature of the IoT also resulted in several efforts from research, industry and standardization communities. In fact, such large deployments of heterogeneous devices require to redesign most of the previously available solutions. This led to developing management, communication and security protocols that effectively and sustainably fulfill the requirements of IoT applications. Among others, the IETF especially released the Constrained Application Protocol (CoAP) [3] and the Datagram Transport Layer Security (DTLS) suite [4][5].

Nevertheless, IoT devices are highly exposed to *Denial of Service* (DoS), i.e. an attack that injects (high-volume) invalid traffic against victim hosts. The attack aims at exhausting victims' resources through worthless demanding operations, thus degrading performance or even making victims unable to serve legitimate requests. DoS concerns many application and technology areas, e.g. wireless sensor networks [6], mobile computer environments [7], and power server systems [8]. Recently, it was confirmed in the top IoT security threats [9].

Several countermeasures are designed to run on the potential victims, rely on DoS-resilient management of resources, and often leverage the detection of invalid messages, enforced through security protocols operating at different layers, e.g. [4][10]. That is, received messages that do not pass security checks are discarded, thus avoiding further processing at the higher layers. However, these detection mechanisms act exclusively as deterrent and only avoid the worst impacts possible. In fact, victims under attack may still remain worthlessly fully active and available, and thus use resources mostly to process invalid service requests. That is, a strong enough attack may still harm service availability, and even run the victims out of resources. Thus, detection mechanisms alone are not flexible in *adjusting* the service availability and resource usage on the victims, based on a dynamic observation of attack conditions.

In this paper, we fill this gap and present SARDOS, our adaptive counteraction against DoS. SARDOS builds on the following rationale. When experiencing a DoS attack, it is not convenient for a victim server to remain *fully* and *directly* available to serve requests. In fact, resources would be used mostly to process attack messages, thus compromising availability and performance anyway. Instead, the server may rather *gradually* and *adaptively* enforce a trade-off between protection from DoS and direct service availability. According to this rationale, SARDOS is designed to achieve two goals.

First, SARDOS limits the impact of DoS on the server's resources. This is achieved by regularly assessing the amount of received invalid messages, and then adjusting the operative state of the server based on the current attack intensity. Second, SARDOS preserves a (best-effort) capability to serve requests from legitimate clients. This is achieved by means of a trusted Proxy that temporarily assists a server under attack, through message relaying during mild/intermittent DoS, or through message caching during intense/persistent DoS. To the best of our knowledge, SARDOS is the first host-based approach that dynamically adapts the service behavior of a server under attack, while preserving a (best-effort) service availability.

Also, SARDOS has the following advantages. First, it is *self-adaptive*, as it autonomously adapts the victim's behavior based on the observed attack intensity. Second, it builds upon mechanisms for detecting invalid messages, and leverages their indications to dynamically adjust the victim's operative state when under attack. SARDOS is not devoted to particular secu-

Copyright of IEEE, 2018. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The official version can be found at: https://ieeexplore.ieee.org/document/8554819

rity mechanisms for detecting invalid messages, i.e. multiple detection mechanisms can be used, possibly at different layers.

We developed a prototype implementation of SARDOS [11] for the library Californium/Scandium [12], and used it to experimentally evaluate the validity and effectiveness of our approach, considering DoS attacks of different intensities. In particular, we evaluated the memory and CPU usage of an underclocked Raspberry Pi 3 server under DoS attack, as well as the impact on Request-Response exchanges for client hosts. Our results show that, even with no particular optimization, SARDOS effectively limits the worthless usage of resources on a server under attack, while at the same time preserving (best-effort) service to legitimate clients.

The paper is organized as follows. We overview the related work in Section II, and background concepts in Section III. We introduce the application scenario and adversary model in Section IV. We present our adaptive solution against DoS in Section V, and our experimental evaluation in Section VI. Finally, in Section VII, we draw our conclusive remarks.

II. Related Work

Denial of Service (DoS) attacks aim at making victim hosts unable to effectively serve requests. This is achieved by exhausting the victim's resources, so jeopardizing performance and service availability. DoS attacks are classified as *benign*, *malignant*, and *service request* [7]. Benign DoS induces valid but resource-harvesting processing; malignant DoS includes compromising victims to alter their behavior; service request DoS repeatedly sends requests to victims to trigger worthless processing. DoS has targeted different application and technology areas, e.g. mobile computer environments [7], Wireless Sensor Networks [6] and even powerful server systems [8].

Solutions are classified as *router-based* and *host-based* [13]. Router-based solutions rely on mechanisms running on routers, to detect and block DoS traffic [14] or to trace the attacker [15]. However, they require all routers to coordinate and to use packet marking techniques [16]. Host-based solutions are enforced on victim hosts, and rely on DoS-resilient management of resources [17]. This paper falls into this latest category.

SMACK is a host-based countermeasure that reduces the impact of service request DoS [10], thanks to a short Message Authentication Code (MAC) included in request messages sent to server devices. The recipient checks the conveyed short MAC, and thus can promptly identify and discard invalid messages, as likely part of a DoS attack. The authors provide an adaptation of SMACK for the IoT protocol CoAP, where the short MAC is *embedded* in CoAP requests, thus resulting in no message expansion. However, SMACK does not take any reactive action to further protect a device under DoS attack.

In the TCP SYN flooding attack, spoofed TCP SYN packets are sent to servers to trigger TCP three-way handshakes and create half-open TCP sessions. Detection relies on traffic analysis [18], while countermeasures use complex router-based solutions [19] and mitigating host-based solutions, such as *SYN cache* [20], *SYN Cookies* [21] and *Client Puzzles* [22].

Another target for DoS is also the *Handshake* protocol of TLS [23] and DTLS [4] used by two peers to establish a secure channel. That is, the adversary induces a server to start several half-open (D)TLS sessions, and thus to run out of resources. Countermeasures rely on: client puzzles to make the attack less efficient [24]; server-generated *Cookies* to be echoed by clients [4]; authenticated *ClientHello* messages enabling only legitimate clients to open (D)TLS sessions with servers [25].

Compared to pure detection/mitigation approaches, this paper takes a step forward for further reducing the attack impact and protecting victim devices. That is, SARDOS builds upon mechanisms for detecting invalid (DoS) messages, and relies on their indications to dynamically adjust the victim's operative state. At the same time, it preserves service availability to the best extent possible, even under heavy DoS attack. To the best of our knowledge, this is the first host-based approach that dynamically adapts the service behavior of a server under attack, while preserving (best-effort) service availability.

III. Background

In this section, we provide an overview of the main background concepts referred throughout the rest of the paper.

A. CoAP

The Constrained Application Protocol (CoAP) is an asserted IoT standard for message transfer at the application layer [3]. It refers to a client-server model where network nodes, namely *endpoints*, can also switch role. CoAP is affordable in constrained IoT environments based on machine-to-machine communication, e.g. building automation and sensor networks.

While using the REST architecture like HTTP, CoAP is not session-based, runs on UDP, uses an asynchronous messaging model, and handles delayed and lost messages. It is designed to work with proxies, which are entrusted with forwarding request and response messages, possibly performing caching. CoAP does not provide security procedures, and recommends using DTLS [4] to achieve secure communication, i.e. authentication, integrity and confidentiality of exchanged messages.

CoAP messages start with: i) 4 bytes including information such as a Message ID; ii) an optional *Token* to match request and response messages; and, possibly, iii) a number of *options* specified according to a Type-Length-Value format and used to control additional features. For instance, they can specify for how long a message is valid, indicate message fragmentation, or instruct a proxy on how to handle relayed messages.

B. SMACK

SMACK is a security service that enables early and efficient detection of invalid (DoS) messages [10]. It relies on a short Message Authentication Code (MAC) seamlessly embedded in transmitted messages. A recipient verifies the short MAC and determines if the received message is genuine and coming from a legitimate sender, or is invalid and to be discarded.

In [10], SMACK was explicitly adapted to work for CoAP. That is, the short MAC is embedded in the Token of the CoAP header, thus resulting in no communication overhead and no

changes to the message format. This paper considers this adaptation of SMACK to detect DoS attacks when communications between two CoAP endpoints occur over an insecure channel.

SMACK relies on a Key Distribution Center (KDC), which is in a trust relation with the recipient device and shares with it a long-term key. Upon request, the KDC provides a sender device with: i) a nonce to use as initial Message ID; and ii) unique key material valid for a SMACK session, which expires after a fixed number of sent messages. The sender increments the Message ID after each message transmitted to that recipient in the SMACK session, and uses the session key material to compute the short MAC. By using the Message ID and the long-term key shared with the KDC, the recipient derives the same key material and verifies that the short MAC is correct.

C. DTLS

The Datagram Transport Layer Security (DTLS) standard [4] provides communication security at the transport layer, when message exchanges rely on connectionless transport protocols, e.g. UDP. It is designed as a close copy of the TLS standard [23] and fulfills equivalent security requirements, i.e. it prevents eavesdropping, tampering and message forgery.

Two DTLS peers first run the DTLS *Handshake* protocol, so exchanging network information and establishing cryptographic key material to protect communications. Specifically, one device acts as *client*, whereas the other one acts as *server*. The *Handshake* can be based on certificates as default option, on raw public keys, or on symmetric pre-shared keys. After the Handshake, the two peers share a secure session and can communicate through the DTLS *Record* protocol, which securely transports data and connection state information, and concretely provides secure and reliable message transfer.

IV. Application Scenario and Adversary Model

Hereafter, we consider the scenario in Figure 1, where a Server S and a Client C communicate using CoAP, either over an insecure channel or using DTLS. Also, an adversary A performs a DoS attack, i.e. it repeatedly sends invalid CoAP request messages to S, inducing their parsing and processing. We refer to these messages as *attack messages*. This *service request* attack induces S to worthlessly commit resources, thus endangering responsiveness or even availability altogether.

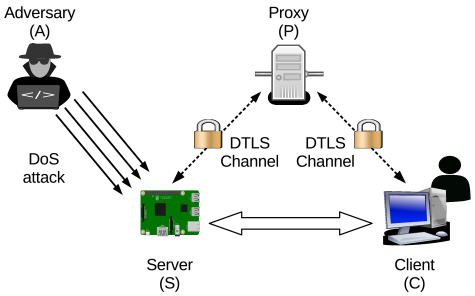

Fig. 1: Application scenario.

When under attack, S should identify and discard attack messages. To this end, S can use different detection mechanisms to distinguish between messages from legitimate clients and *invalid messages*. This paper focuses on two detection mechanisms, i.e. SMACK [10] or the DTLS Record protocol [4]. While invalid messages may be due to accidental corruption or processing errors, we refer to a conservative policy where S considers each invalid message as an attack message.

The Proxy P acts as intermediary between C and S, although *only* during DoS attacks. That is, P does *not* normally participate in the communications between C and S, as long as S is not under attack and is operating in normal conditions, i.e. it directly communicates with C. We detail the behavior of P in Section V, when presenting our countermeasure to DoS. A server is associated to one Proxy, which can be associated to multiple servers. We assume that both C and S securely communicate with P using CoAP over DTLS. If S uses SMACK, then P acts also as KDC (see Section III-B).

We consider P trustworthy and designed as reliable and secure, thus practically infeasible to compromise. P can be centralized or based on a distributed architecture. A centralized approach is easier to adopt, but leads to a single point a failure. However, P is supposed to be robust and reliable by design, and the literature provides established techniques to achieve security and reliability in these systems [26][27]. While it is impractical to apply such techniques to each and every server, it is feasible to use them for relatively few deployed units such as P. Instead, a distributed approach prevents single points of failure, and improves robustness and availability. However, it introduces logical replica to keep synchronized and consistent. This work is not devoted to any particular architectural design, and further related details are out of the scope of this paper.

Note that several IoT scenarios do rely on intermediary units that support asynchronous communication models and offload effort from servers. Typically, intermediaries are used as: i) proxies, for message forwarding and caching [3]; ii) authorization servers, to enforce access control policies [28][29]; iii) key managers, to revoke and distribute cryptographic material.

V. Reaction against Denial of Service

This section presents SARDOS, our adaptive counteraction to DoS. SARDOS builds on the following rationale. When under DoS attack, it is not convenient for a server to be fully and directly available to serve requests. In fact, server's resources would be mostly used to handle and process attack messages, thus worsening availability and performance anyway. Instead, the server can *adaptively* and *gradually* enforce a trade-off between direct service availability and protection from DoS. Based on this, we designed SARDOS to achieve two goals.

First, SARDOS aims at limiting the impact of DoS on the server's resources. To this end, the server regularly collects the number of received invalid messages, then assesses the attack intensity, and finally adjusts its operative state accordingly. Invalid messages can be detected with multiple, possibly co-existing, security mechanisms at different layers. Second, SARDOS aims at preserving a (best-effort) capability to serve requests from legitimate clients. To this end, it relies on the trusted Proxy to assist the server when under attack. The Proxy relays messages between clients and the server during

mild/intermittent DoS, and caches client messages to be later forwarded to the server, during intense/persistent DoS.

The following describes the server operative states, the transition among states, and how they affect the client experience.

A. Server operational perspective

In SARDOS, the server S can be in one of three operative states, namely NORMAL, PROTECTED and OFF. In the following, we describe how S operates in each different state.

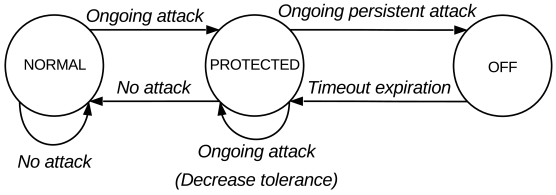

Fig. 2: Server state machine.

Figure 2 shows the state machine adopted on the server. The NORMAL state denotes a *typical* behavior where S serves client requests at its earliest convenience. The PROTECTED state denotes a *limited* behavior, where S is under a mild and intermittent DoS, and serves clients only through P acting as message relay. The OFF state denotes a *best-effort* behavior, where S is under an intense and persistent DoS, has its network interface turned off, and relies on P to cache client requests to be served once back to PROTECTED. Upon changing state, S informs P by exchanging dedicated SARDOS control messages over the pre-established DTLS channel (see Section IV). We detail the process of state transition in Section V-B. Hereafter, we refer to the SARDOS parameters in Table I.

1) Server in NORMAL state: While in NORMAL, S receives requests directly from clients and serves them at its earliest convenience. Once moved back from PROTECTED, S notifies P to be resuming its operations in NORMAL state.

2) Server in PROTECTED state: While in PROTECTED, S processes client requests only if relayed by P, and provides related responses through P. So doing, S reduces its effort when under mild/intermittent DoS. Before entering this state, i.e. from NORMAL or OFF, S notifies P to be operating in PROTECTED state for t_1 s. If moving back from OFF, S gets cached requests from P, to be served while in PROTECTED.

3) Server in OFF state: While in OFF, S keeps its network interface turned off, in order to greatly reduce its effort during intense/persistent DoS. Before entering this state, S notifies P to be switching to OFF for t_2 s, $t_2\_MIN \leq t_2 \leq t_2\_MAX$, during which P caches client requests to be served later on.

While in NORMAL or PROTECTED, S regularly checks the reception rate of invalid messages, and determines whether to switch to a different state (see Section V-B). Intuitively, S considers consecutive time windows, during each of which it maintains a counter of invalid messages X. That is, S resets X to 0 at the beginning of each time window, and then increments X for each invalid message received during that time window. At the end of each time window, S compares X against a state attack threshold Th. Then, S accordingly updates Th to adjust its attack tolerance, and possibly switches to a different state.

TABLE I: List of SARDOS parameters.

Name	Description	Unit
X	Counter of invalid messages	Messages
W_0	Length of NORMAL time windows	s
W_1	Length of PROTECTED time windows	s
Th_0	Attack threshold in NORMAL state	Messages
Th_0_MIN	Minimum size of Th_0	Messages
Th_0_MAX	Maximum size of Th_0	Messages
Th_1	Attack threshold in PROTECTED state	Messages
Th_1_MIN	Minimum size of Th_1	Messages
Th_1_MAX	Maximum size of Th_1	Messages
k	Corrective exponent	-
k_MAX	Maximum value of k	-
t_1	Value for timer T1	s
t_2	Value for timer T2	s
t_2_MIN	Maximum value for timer T2	s
t_2_MAX	Maximum value for timer T2	s

Invalid messages can be detected through different security mechanisms, possibly co-existing at different layers. In this work, we considered SMACK and the DTLS Record protocol (see Section III). However, SARDOS is not devoted to any particular detection mechanism, and different approaches than SMACK and DTLS can be seamlessly used. In fact, regardless the specific mechanisms and their rationale, each received message assessed as invalid is silently discarded and results in X incremented by 1. That is, S enforces a conservative policy where all invalid messages are considered attack messages.

B. Server state transitions

Upon startup, S establishes a DTLS session with P (see Section IV), with which it registers as a server running SARDOS. Then, state transitions occur as described below.

When entering NORMAL, S performs the actions in Algorithm 1. Upon startup initialization, S sets the attack threshold Th_0 to Th_0_MAX. Instead, if PROTECTED was the previous state, S sets Th_0 to Th_0_MIN, so taking a conservative behavior. Either case, S sets the attack message counter X to 0, sends to P a SARDOS control message informing to be in NORMAL state, and starts operating in NORMAL state.

Algorithm 1 Enter the NORMAL state

1: **if** (Server initialization) **then**
2: $\quad Th_0 \leftarrow Th_0\_MAX$
3: **else if** (Previous state was PROTECTED) **then**
4: $\quad Th_0 \leftarrow Th_0\_MIN$
5: **end if**
6: $X = 0$
7: <Inform P to be in NORMAL>

While in NORMAL, S assesses the occurrence of DoS and a possible transition to PROTECTED, as per Algorithm 2. That is, at the end of each NORMAL time window, i.e. after every W_0 s, S checks the following three mutually exclusive conditions. (1) As per Lines 2-5, S checks whether it is under no attack or negligible attack, i.e. $X < Th_0\_MIN$. If so, S relaxes its attack assessment by doubling the threshold Th_0,

Algorithm 2 Assess transition to PROTECTED state

1: <At the end of each NORMAL time window>
2: **if** ($X < Th_0\_MIN$) **then**
3: $Th_0 \leftarrow min(2 \cdot Th_0, Th_0\_MAX)$
4: $X = 0$
5: // Move to the next NORMAL time window
6: **else if** ($Th_0\_MIN \leq X < Th_0$) **then**
7: $Th_0 \leftarrow max(Th_0/2, Th_0\_MIN)$
8: $X = 0$
9: // Move to the next NORMAL time window
10: **else if** ($X \geq Th_0$) **then**
11: <Move to PROTECTED state>
12: **end if**

up to Th_0_MAX. Then, S resets X to 0 and moves to the next NORMAL time window. (2) Otherwise, as per Lines 6-9, S checks whether it is under a mild yet non-negligible attack, i.e. $Th_0\_MIN \leq X < Th_0$. If so, S hardens its attack assessment by halving the threshold Th_0, possibly rounding it to Th_0_MIN. Then, S resets X to 0 and moves to the next NORMAL time window. (3) Otherwise, as per Lines 10-12, S checks whether it is under a non-negligible and persistent attack, i.e. $X \geq Th_0$. If so, S moves to PROTECTED.

When moving to PROTECTED, S performs the following actions, as per Algorithm 3. If NORMAL was the previous state, S initializes the threshold Th_1 to Th_1_MAX. Instead, if OFF was the previous state, S initializes Th_1 to Th_1_MIN, so taking a more conservative behavior. Either case, S sets both the counter X and the exponent k to 0. Then, S sends to P a SARDOS control message informing to be in PROTECTED state for t_1 s. If NORMAL was the previous state, S must re-transmit the control message until it gets an acknowledgment, in order to ensure synchronization with P. After that, S starts the timer T_1 set to t_1 s and begins operating in PROTECTED state. If the timer T_1 expires, S sends a SARDOS control message to P, and moves to NORMAL (see Algorithm 1).

While in PROTECTED, S assesses the occurrence of DoS and a possible transition to OFF, as per Algorithm 4. That is, at the end of each PROTECTED time window, i.e. after every W_1 s, S checks the following three mutually exclusive conditions. (1) As per Lines 2-5, S checks whether it is under no attack or negligible attack, i.e. $X < Th_1\_MIN$. If so, S relaxes its attack assessment by doubling the threshold Th_1 up to Th_1_MAX. Then, S resets X to 0 and moves to the next PROTECTED time window. (2) Otherwise, as per Lines 6-10, S checks whether it is under a mild yet non-negligible attack, i.e. $Th_1\_MIN \leq X < Th_1$. If so, S hardens its attack assessment by reducing Th_1, possibly rounding it to Th_1_MIN as minimum possible value. That is, the exponent k is incremented by 1 up to k_MAX, and Th_1 is decremented by 2^k. This allows S to perform *smooth* adjustments, while avoiding continuous and unstable swings between PROTECTED and OFF. Note that k is reset to 0 only when switching to PROTECTED from a different state (see Algorithm 3). Then, S resets X to 0 and moves to the next PROTECTED time window. (3) Otherwise, as per Lines 11-13, S checks whether it is under a non-negligible and persistent attack, i.e. $X \geq Th_1$. If so, S moves to OFF.

When moving to OFF, S performs the following actions.

Algorithm 3 Enter the PROTECTED state

1: **if** (Previous state was NORMAL) **then**
2: $Th_1 \leftarrow Th_1\_MAX$
3: **else if** (Previous state was OFF) **then**
4: $Th_1 \leftarrow Th_1\_MIN$
5: **end if**
6: $X = 0$
7: $k = 0$
8: <Inform P to be in PROTECTED and start timer T_1>

Algorithm 4 Assess transition to OFF state

1: <At the end of each PROTECTED time window>
2: **if** ($X < Th_1\_MIN$) **then**
3: $Th_1 \leftarrow min(2 \cdot Th_1, Th_1\_MAX)$
4: $X = 0$
5: // Move to the next PROTECTED time window
6: **else if** ($Th_1\_MIN \leq X < Th_1$) **then**
7: $k \leftarrow min(k + 1, k\_MAX)$
8: $Th_1 \leftarrow max(Th_1 - 2^k, Th_1\_MIN)$
9: $X = 0$
10: // Move to the next PROTECTED time window
11: **else if** ($X \geq Th_1$) **then**
12: <Stop timer T_1 and move to OFF state>
13: **end if**

First, it randomly generates t_2, $t2\_MIN \leq t_2 \leq t2\_MAX$. Then, S sends to P a SARDOS control message informing to be in OFF state for t_2 s. Note that S must retransmit the control message until it gets an acknowledgment, in order to ensure synchronization with P. After that, S starts the timer T_2 set to t_2, and turns off its network interface. When the timer T_2 expires, S turns on its network interface and sends a SARDOS control message to P, in order to fetch possible cached messages sent by legitimate clients. Finally, S moves back to PROTECTED (see Algorithm 3). Note that the amount of time t_2 spent in OFF state changes at each transition to OFF, thus preventing the adversary to dynamically adapt the attack profile based on otherwise predictable state transitions.

C. Client operational perspective

This section describes how each operative state on the server side reflects on the service experienced on the client side.

1) Server in NORMAL state: While in this state, S normally processes client requests, and directly replies at its earliest convenience. If C believes S to be in PROTECTED or OFF, and sends a request to P with S as final recipient, P replies with a SARDOS control message, telling C to transmit its requests directly to S, which is in fact in NORMAL.

2) Server in PROTECTED state: While in this state, S accepts only requests that are relayed by P. If S receives a request *directly* from C while in this state, two cases can occur.

First, in case C has already an established communication session with S, i.e. through SMACK or DTLS, S verifies the request to be a valid message, and silently discards it anyway. Then, S replies to C with a SARDOS control message, signaling to be currently operating in PROTECTED.

Second, if there is no established communication session, S does not process the request, counts it as an invalid message, i.e. increments the counter X, and discards it. Then, S does not reply to C, which would re-transmit the same request until it reaches its retransmission limit. The network administrator should properly configure the SARDOS parameter on the

server side, to avoid that relatively few retransmissions from legitimate clients can unfairly induce transitions to OFF state.

Either case, C assumes that S is in PROTECTED, and starts transmitting to P all requests addressed to S, indicating to relay them directly to S. Then, C also receives the related responses as relayed by P. Note that C assumes S to be in PROTECTED until P notifies that S has switched to a different state.

3) Server in OFF state: While in this state, S has its network interface turned off and cannot serve any request.

If C believes S to be in NORMAL and contacts S directly, it does not get a reply and would re-transmit a request until reaching its retransmission limit. After that, C assumes S to be in PROTECTED, and transmits to P its requests addressed to S, indicating to not cache them and to relay them directly to S. Alternatively, C might have already known S to be in PROTECTED. If so, C sends to P its requests addressed to S, indicating to not cache them and to relay them directly to S.

Either case, P replies to C with a SARDOS control message, signaling that S is actually in OFF. Then, C may still transmit to P all its requests addressed to S, indicating to cache them and to relay them to S when it switches back to PROTECTED. When doing so, C can provide an indicative amount of time t_OFF_C that it is fine to wait for getting a response from S, though relayed through P. Then, P accordingly caches such requests if the residual time that S spends in OFF state is less than the amount of time indicated by C. Otherwise, P discards those request messages. Either case, P informs C of the residual amount of time to wait until S switches back to PROTECTED state. This prevents C from performing further worthless request transmissions during that time interval. Upon receiving responses to cached requests, C assumes S to be in PROTECTED and accordingly sends new requests through P.

VI. Experimental Evaluation

To evaluate SARDOS, we developed a prototype implementation for the library Californium/Scandium [12], and tested it on a Raspberry Pi server. Our implementation is available as open source at [11]. This section presents results from our experimental evaluation, considering different attack intensities and two different detection mechanisms. We show that, even without particular optimizations, SARDOS effectively limits the worthless usage of resources on a server under DoS attack, while still providing (best-effort) service to legitimate clients.

For our experiments, we referred to the scenario in Figure 1 deployed over a local Ethernet network. We used a commodity laptop where the three clients and the adversary ran as four separate processes. We used a second commodity laptop as CoAP proxy P. Both laptops were equipped with 12 GB of RAM and a Quad Core 2.1 GHz CPU, and ran Ubuntu Linux. As server S, we used a Raspberry Pi 3 model B [30] running Raspbian Linux 9, and equipped with 1 GB of RAM and a Quad Core 1.2 GHz 64-bit CPU. To make the Raspberry Pi constrained and more sensitive to DoS, we switched off 3 of its 4 CPUs, and underclocked the active one to 850 MHz. The laptops and the Raspberry Pi ran an extended version of the library Californium/Scandium implementing CoAP and DTLS.

The same experiments were performed in five test cases. 1) In T_NO_SEC, SARDOS is *not* used, and communications between clients and S are not secured. 2) In T_SMACK, SARDOS is *not* used, communications between clients and S are not secured, and S uses SMACK. 3) In T_DTLS, SARDOS is *not* used, and communications between clients and S are secured with DTLS. 4) In T_SMACK+, SARDOS is used, communications between clients and S are not secured, and S uses SMACK. 5) In T_DTLS+, SARDOS is used, and communications between clients and S are secured with DTLS. Also, S always discards invalid messages. When SARDOS is used, S considers each invalid message as an attack message.

For each test case, one experiment consists in each client performing 50 Request-Response exchanges with S. An experiment is completed when each client has received the response to its last request, or has reached the retransmission limit for its last request. Each client generates a new request every 2.5 s, i.e. a 23-byte CoAP request if DTLS is not used, or a 53-byte DTLS Record encapsulating the request otherwise. A client sends a new request once received the response to the latest one, or upon reaching the retransmission limit for the latest one. As per the CoAP standard [3], a client transmits a request up to 4 times, with an overall response timeout of 93 s.

If SMACK is used, we consider sessions of 127 messages in length. If DTLS is used, we consider the cryptosuite TLS_PSK_WITH_AES_128_CBC_SHA256. If clients and S use DTLS or SMACK, they have a previously established DTLS channel or SMACK session. If SARDOS is used, clients and S communicate with P on pre-established DTLS channels. These exchanges include SARDOS control messages to/from P, whose payload follows a Type-Subtype-Value scheme and may convey CoAP messages that P relays or caches.

We considered four attack configurations, where the adversary sends attack messages with different attack rates R, i.e. 0 msg/s (no attack occurs), 70 msg/s, 350 msg/s and 1000 msg/s. The adversary sends plain 23-byte CoAP requests in T_NO_SEC, T_SMACK and T_SMACK+, or invalid 53-byte DTLS Records encapsulating a CoAP request in T_DTLS and T_DTLS+. To model a worst case condition, in T_SMACK, T_DTLS, T_SMACK+ and T_DTLS+ the adversary sends spoofed invalid messages, with source IP address of a client sharing a SMACK or DTLS session with S. This induces S to process attack messages to the maximum possible extent.

In T_SMACK+ and T_DTLS+, the SARDOS parameters were configured as: Th_0_MAX = 180 msg; Th_0_MIN = 100 msg; Th_1_MAX = 385 msg; Th_1_MIN = 325 msg; $W_0 = W_1 = 3$ s; t_1 = 15 s; t_2_MAX = 12 s; t_2_MIN = 8 s. Clients refer to t_OFF_C = 40 s. While these values were good to support our experiments, they should not be intended as optimal. Finding optimal values, e.g. through an analytical model, is out of the scope of this paper. As per Algorithms 2 and 4, SARDOS assesses the current attack conditions based on the invalid messages detected by SMACK or the DTLS Record protocol, possibly enforcing a state transition on S.

We evaluated: i) the impact on Request-Response exchanges with S experienced by a client; ii) the time spent by S in

Test case	Direct	Via P	Via P (cached)
T_NO_SEC	50	-	-
T_SMACK	50	-	-
T_DTLS	50	-	-
T_SMACK+	50	0	0
T_DTLS+	50	0	0

TABLE II: Served client requests (Attack rate R = 0 msg/s).

Test case	Direct	Via P	Via P (cached)
T_NO_SEC	50	-	-
T_SMACK	50	-	-
T_DTLS	50	-	-
T_SMACK+	7	43	0
T_DTLS+	7	43	0

TABLE III: Served client requests (Attack rate R = 70 msg/s).

Test case	Direct	Via P	Via P (cached)
T_NO_SEC	7	-	-
T_SMACK	50	-	-
T_DTLS	50	-	-
T_SMACK+	6	1	43
T_DTLS+	5	1	44

TABLE IV: Served client requests (Attack rate R = 350 msg/s).

Test case	Direct	Via P	Via P (cached)
T_NO_SEC	3	-	-
T_SMACK	50	-	-
T_DTLS	50	-	-
T_SMACK+	5	1	44
T_DTLS+	5	1	44

TABLE V: Served client requests (Attack rate R = 1000 msg/s).

each SARDOS state; and iii) the RAM and CPU usage on S. We averaged results over independent repetitions, and never observed any particular differences among the three clients.

A. Results

Tables II-V show the number of requests successfully served for each client, i.e. the number of requests for which that client receives a response. The columns "Via P" and "Via P (cached)" are relevant only for the test cases T_SMACK+ and T_DTLS+, where SARDOS is used. We recall that each client sends 50 CoAP requests intended to S. As we did not observe any particular difference among the different clients, in the following we simply refer to C as the considered one.

As shown in Table II, when $R = 0$ msg/s, i.e. no attack occurs, all 50 requests are always responded to, and especially directly by S. As shown in Table III, when $R = 70$ msg/s, all 50 requests are also always responded to. However, in the test cases T_SMACK+ and T_DTLS+, 7 responses are received directly from S, while the remaining 43 are relayed by P. Note that, since S does not go to OFF, no requests are cached on P.

Table IV considers $R = 350$ msg/s. In T_NO_SEC, only the first 7 requests are served, after which S becomes congested and unresponsive to serve further requests. After that, all the following client requests time out. In the other test cases, S serves all the 50 requests. Of course, in the T_SMACK and T_DTLS test cases, all requests are responded to directly by S. Instead, in T_SMACK+ and T_DTLS+, the initial requests are also responded to directly, while the following ones are responded to via P. In particular, almost all such requests are initially cached by P, due to S being also in OFF. Once back to PROTECTED, S serves each cached request after having retrieved it from P. Similar considerations hold for Table V, i.e. when $R = 1000$ msg/s. However, in T_NO_SEC test case, only the first 3 requests are served, before S becomes congested and unresponsive to serve further requests.

When SARDOS is used under attack rates $R = 350$ and $R = 1000$ msg/s, the overall response time for cached requests equals to: i) the response time of non-cached requests relayed by P; plus ii) the request cache time on P, which is in turn influenced by the variable time t_2 that S spends in OFF. For our settings, such overall response time ranges between 10 and 14 s, i.e. up to 15% of the overall response timeout.

Figure 3 covers the test cases T_SMACK+ and T_DTLS+, and shows the distribution of server time among the different SARDOS states. When S is under attack, the time spent in NORMAL decreases, S operates also in PROTECTED, and it spends time mostly in OFF for high attack rates. For the considered settings, both the attack rates $R = 350$ and $R = 1000$ msg/s result in a very similar time distribution. Also, given a same attack rate, we did not observe significant differences between the test cases T_SMACK+ and T_DTLS+.

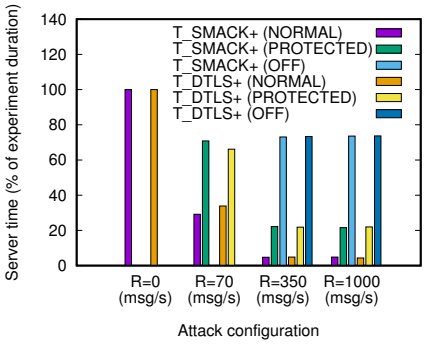

Fig. 3: Server time in SARDOS states.

Figure 4 shows the average RAM occupancy on S. The RAM used in T_NO_SEC is comparable with the RAM used in T_SMACK, T_DTLS and T_DTLS+ for $R = 0$ and $R = 70$ msg/s, while about 24% lower than in the same test cases for $R = 350$ and $R = 1000$ msg/s. Instead, in every attack configuration, the RAM used in T_SMACK+ is much lower than the RAM used in the other test cases, i.e. about 26% less than in T_SMACK, and even up to 41% lower than in T_DTLS and T_DTLS+. For high attack rates, the T_DTLS and T_DTLS+ test cases display a RAM consumption which is about 23% higher than in T_NO_SEC. This condition is mostly due to the memory-harvesting DTLS, and is only slightly ameliorated by additionally using SARDOS. On the contrary, when relying on SMACK as detection mechanism, the additional usage of SARDOS in T_SMACK+ greatly reduces the RAM usage. This effect is more intense for high attack rates, where T_SMACK+ displays a RAM usage which is even much lower than in the T_NO_SEC test case.

Figure 5 shows the average CPU usage on S. In the T_NO_SEC test case, the CPU usage reaches 77% for $R = 70$

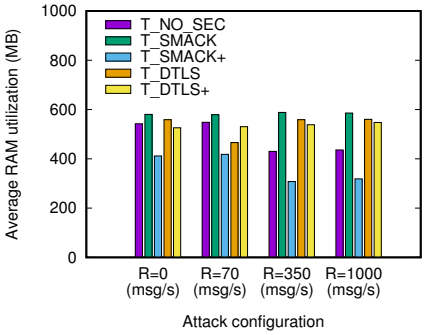

Fig. 4: Memory usage on the server.

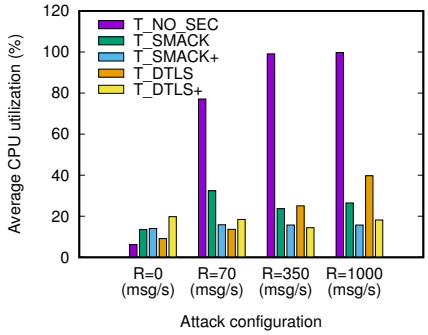

Fig. 5: CPU usage on the server.

msg/s, and 99% for higher attack rates. The other test cases display the following. If $R = 0$ msg/s, the detection mechanisms and possibly SARDOS result in higher CPU usage than in T_NO_SEC, due to computing overhead which is not compensated by any attack counteraction. Instead, they greatly reduce the CPU usage for higher attack rates. That is, if $R = 70$ msg/s and S relies on SMACK, the additional usage of SARDOS in T_SMACK+ *further* greatly reduces the CPU usage. For the highest attack rates, the benefits of SARDOS become even more considerable. In fact, for $R = 350$ msg/s, we observe that, in T_SMACK+ (T_DTLS+), SARDOS reduces the already limited CPU usage of T_SMACK (T_DTLS) by a *further* 33.5% (42.5%). Similarly, for $R = 1000$ msg/s, we observe that, in T_SMACK+ (T_DTLS+), SARDOS reduces the already limited CPU usage of T_SMACK (T_DTLS) by a *further* 40.62% (54.2%). Therefore, under intense and persistent DoS, SARDOS results in an average CPU usage on S which is up to 15.75% (18.19%) in T_SMACK+ (T_DTLS+).

VII. Conclusion

We have presented SARDOS, a reactive security service against Denial of Service. SARDOS dynamically adapts the operative state of victim servers, so limiting worthless usage or resources, while still ensuring a (best-effort) fulfillment of requests from legitimate clients. To this end, it leverages detection mechanisms from different layers, and considers current attack conditions to adjust the operational state of the victim server. We experimentally confirmed the effectiveness of SARDOS through our prototype implementation, tested on an underclocked Raspberry Pi server running a typical IoT stack. That is, SARDOS considerably limits the usage of memory and CPU on a server under attack, while preserving service availability. Future works will focus on adapting SARDOS to resource-constrained platforms running the Contiki OS, while exploiting power-saving modes of battery-powered servers.

Acknowledgment

This work was supported by the EIT-Digital HII ACTIVE.

References

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, October 2010.

[2] G. Kortuem, F. Kawsar, V. Sundramoorthy, and D. Fitton, "Smart objects as building blocks for the Internet of things," *IEEE Internet Computing*, vol. 14, no. 1, pp. 44–51, January-February 2010.

[3] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252, 2014.

[4] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC 6347, 2012.

[5] H. Tschofenig and T. Fossati, "Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things," RFC 7925, 2016.

[6] D. R. Raymond, R. C. Marchany, M. I. Brownfield, and S. F. Midkiff, "Effects of Denial-of-Sleep Attacks on Wireless Sensor Network MAC Protocols," *IEEE Transactions on Vehicular Technology*, vol. 58, no. 1, pp. 367–380, January 2009.

[7] T. Martin, M. Hsiao, D. Ha, and J. Krishnaswami, "Denial-of-Service Attacks on Battery-powered Mobile Computers," in *IEEE PerCom 2004*. IEEE Computer Society, March 2004, pp. 309–318.

[8] Z. Wu, M. Xie, and H. Wang, "On Energy Security of Server Systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 6, pp. 865–876, November-December 2012.

[9] ENISA, "Baseline Security Recommendations for IoT," 2017.

[10] C. Gehrmann, M. Tiloca, and R. Höglund, "SMACK: Short message authentication check against battery exhaustion in the Internet of Things," in *IEEE SECON 2015*. IEEE, June 2015, pp. 274–282.

[11] "Adaptive DoS," https://bitbucket.org/rhogl/adaptivedos.

[12] "Californium CoAP framework," http://www.eclipse.org/californium/.

[13] C. J. Haining Wang and K. G. Shin, "Defense Against Spoofed IP Traffic Using Hop-Count Filtering," *IEEE/ACM Transactions on Networking*, vol. 15, no. 1, pp. 40–53, February 2007.

[14] H. Beitollahi and G. Deconinck, "A Cooperative Mechanism to Defense against Distributed Denial of Service Attacks," in *IEEE TrustCom 2011*, November 2011, pp. 11–20.

[15] J. Li, M. Sung, J. Xu, and L. Li, "Large-scale IP traceback in high-speed Internet: practical techniques and theoretical foundation," in *The 2004 IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 2004, pp. 115–129.

[16] M. T. Goodrich, "Efficient Packet Marking for Large-scale IP Traceback," in *ACM CCS 2002*. ACM, November 2002, pp. 117–126.

[17] X. Qie, R. Pang, and L. Peterson, "Defensive Programming: Using an Annotation Toolkit to Build DoS-resistant Software," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 45–60, December 2002.

[18] J. Mirkovic and P. Reiher, "D-WARD: A Source-End Defense Against Flooding Denial-of-Service Attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 3, pp. 216–232, July 2005.

[19] D. T. and O. R., "Hot Spares for DoS attacks," *The Magazine of USENIX and SAGE*, vol. 25, no. 4, p. 3, July 2000.

[20] J. Lemon, "Resisting SYN Flood DoS Attacks with a SYN Cache," in *BSD Conference 2002*. USENIX Association, 2002, pp. 1–9.

[21] A. Zuquete, "Improving the Functionality of SYN Cookies," in *The IFIP TC6/TC11 Sixth Joint Working Conference on Communications and Multimedia Security: Advanced Communications and Multimedia Security*. Kluwer, B.V., September 2002, pp. 57–77.

[22] T. Aura, P. Nikander, and J. Leiwo, "DOS-Resistant Authentication with Client Puzzles," in *Revised Papers from the 8th International Workshop on Security Protocols*. Springer-Verlag, 2001, pp. 170–177.

[23] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, 2008.

[24] D. Dean and A. Stubblefield, "Using Client Puzzles to Protect TLS," in *The 10th Conference on USENIX Security Symposium - Volume 10*, ser. SSYM'01. USENIX Association, 2001, pp. 1–8.

[25] M. Tiloca, C. Gehrmann, and L. Seitz, "On Improving Resistance to Denial of Service and Key Provisioning Scalability of the DTLS Handshake," *International Journal of Information Security*, vol. 16, no. 2, pp. 173–193, April 2017.

[26] R. J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems, 2nd Edition*. Wiley Publishing Inc., 2008.

[27] K. P. Birman, *Guide to Reliable Distributed Systems. Building High-Assurance Applications and Cloud-Hosted Services*. Springer, 2012.

[28] L. Seitz, G. Selander, E. Wahlstroem, S. Erdtman, and H. Tschofenig, "Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth)," draft-ietf-ace-oauth-authz-13, March 2018, (work in progress).

[29] L. Seitz, G. Selander, and C. Gehrmann, "Authorization framework for the Internet-of-Things," in *D-SPAN workshop of IEEE WoWMoM 2014*. IEEE Computer Society, June 2013, pp. 1–6.

[30] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.