Mitigating Low-volume DoS Attacks with Data-driven Resource Accounting

ChangSeok Oh[†] Sangho Lee[*] Wen Xu[†] Rohan Devang Vora[†] Taesoo Kim[†]

[†]*Georgia Institute of Technology* [*]*Microsoft Research*

Abstract

Low-volume Denial-of-Service (μDoS) attacks have been demonstrated to fundamentally bypass traditional DoS mitigation schemes based on the flow and volume of network packets. Their three characteristics—low capacity (volume), slow speed (velocity) and benign looking (legitimacy), make it infeasible to tame them by simply observing external network activities, demanding a finer-grained scheme that can monitor the internal activities. Recent μDoS attacks appear to be not just stealthy but also destructive so that they often result in severe consequences to the victim machine: e.g., SegmentSmack and FragmentSmack attacks demonstrated in 2018 can result in a kernel panic or system hang while requiring virtually no resource from the attacker.

In this paper, we propose a data-driven approach, called ROKI, that accurately tracks internal resource utilization and allocation associated with each packet (or session), making it possible to tame resource exhaustion caused by μDoS attacks. Since ROKI focuses on capturing the symptom of DoS, it can effectively mitigate previously unknown μDoS attacks. To enable a finer-grain resource tracking, ROKI provided in concept the accounting capabilities to each packet itself, so we called *data-driven*: it monitors resource utilization at link, network, transport layers in the kernel, as well as application layers, and attributes back to the associated packet. Given the resource usages of each packet, ROKI can reclaim (or prevent) the system resources from malicious packets (or attackers) whenever it encounters system-wide resource exhaustion. To provide a light-weight resource tracking, ROKI carefully multiplexes hardware performance counters whenever necessary. Our evaluation shows that ROKI's approach is indeed effective in mitigating real-world μDoS attacks with negligible performance overheads—incurring 3%–4% throughput and latency overheads on average when the system is throttled.

1 Introduction

Denial-of-Service (DoS) attacks are ongoing, evolving threats to the Internet. Basically, they throttle a victim server with a tremendous number of network packets, so that the server becomes unresponsive to other benign requests. To maximize the number of attack packets, they often abuse a large number of zombie computers [9], also known as distributed DoS (DDoS) attacks. Existing defenses or mitigation approaches either rely on a good, reactive infrastructure to handle higher network capacity, limit the rate of packets to which each client can respond, or filter out the packets with known bad signatures [6,7,18]. These approaches, however, make DoS attacks evolve into two other extremes: either maximizing attack volume to overwhelm the defenses (e.g., the 1.7 Tbps Memcached reflection attack [31]), or minimizing attack volume to bypass them (e.g., the SegmentSmack [39,41] and FragmentSmack [40] attacks). This paper focuses on the latter attack as known as a *low-volume DoS attack* (μDoS).

μDoS requires low-volume, slow-rate attack traffics, yet still results in destructive consequences to the victim machines. Instead of merely flooding a victim's network, μDoS attacks send either crafted exploit packets to trigger DoS vulnerabilities of target software, or expensive, yet legitimate requests that can easily exhaust important system resources, such as CPU cycles, memory or file descriptors. Conventional DoS defense schemes such as improving network capacity, restricting the packet rates or even blocking known bad requests, are, unfortunately, less effective against μDoS attacks. For example, recent μDoS attacks, SegmentSmack [39,41] and FragmentSmack [40], rely only on legitimate packets that can exhaust the CPU cycles of a victim's servers by stressing the re-assembling logic that handles out-of-order TCP segments and incomplete IP fragments. More traditional μDoS attacks, such as Billion Laugh [30] and Slowloris [8], can cause memory exhaustion or limit the number of active sessions of the victim's server, while relying only on benign-looking requests that can bypass the existing DoS defense schemes.

Existing countermeasures against μDoS focus only on fixing specific instances of attacks since by definition, all μDoS attacks are hardly identifiable in terms of network traffics and legitimacy. It is important to address the root cause of μDoS attacks, but our communities are desperately looking for more universal, practical solutions that can mitigate the emerging μDoS attacks [8,11]. To be deployable in practice, it is also important to address the μDoS problems at the end points (i.e., hosts) without requiring modification of network infrastructure.

In this paper, we propose ROKI, a data-driven resource accounting system to mitigate μDoS attacks by monitoring how each packet (or session) accounts to the system resource us-

ages, and reacting to the packet causing resource exhaustion. The key idea of ROKI is to focus on identifying the *symptom* of μDoS (i.e., resource exhaustion, so DoS), rather than identifying specifics of attacks, such as exploit methods or types of DoS vulnerabilities. Once ROKI concludes that the systems are throttled or potentially under a DoS attack, it reacts to the current situation based on the amount of the system resources attributed to each packet (or session). As a reaction, ROKI attempts to reclaim the resources of the most exhaustive packets (or sessions) in order, and prevent the future attacks by blacklisting their origins.

To make this data-driven resource accounting possible, ROKI attempts to accomplish three goals: precision, performance and generality. First, ROKI implements a finer-grained resource accounting scheme. It provides in concept resource accounting capabilities to each packet (or session); it monitors resource utilization at link, network, transport layers in the kernel, as well as application layers, and attributes back to the associated packet. This allows ROKI to reason about μDoS attacks targeting specific network layers. Second, ROKI minimizes the performance overheads required for the fine-grained resource accounting by carefully multiplexing hardware performance counters (HPCs). Third, ROKI focuses on identifying the symptom of the DoS attacks and mitigates them without knowing the details of the attack methods nor targeted resources for exhaustion. Most importantly, ROKI is designed to tackle μDoS at the end hosts without requiring the modification of the network infrastructure.

Our evaluation shows that ROKI can identify real-world μDoS attacks (e.g., FragmentSmack, Apache Range Header, and Slowloris) targeting various types of system resources (e.g., CPU, memory, and connection pool) at different layers (e.g., network, transport, and application layers). Also, ROKI can mitigate on-going μDoS attacks by selectively blocking exhaustive requests. That is, it can continue to serve legitimate requests by dropping up to 16% of requests even under active μDoS attacks. It also imposes negotiable performance overheads: when the system is throttled and ROKI is applied, it incurs only 3.5%–4.8% of latency and throughput overheads.

The summarized contributions of this paper are as follows:

- **Data-oriented resource-usage profiling.** To the best of our knowledge, ROKI is the first study that detects and blocks suspicious requests according to their high system resource usages. ROKI enables data-oriented resource-usage profiling to accurately identifies how many resources have been used to process each request at each network layer, detecting suspicious clients regardless of which unknown exploit techniques they use.
- **Hardware-based efficient profiling.** ROKI uses HPCs for efficient profiling of per-packet per-layer resource usage. More specifically, ROKI uses the performance monitoring unit (PMU) to check the number of retired CPU instructions at each layer, and the memory bandwidth monitoring (MBM) to check the number of memory accesses at each layer for processing each packet.
- **Universal defense.** ROKI is effective against various μDoS attacks targeting different resources. ROKI monitors various system resources simultaneously using the same technique to detect and avoid any of their exhaustion.

The remainder of this paper is organized as follows. §2 defines μDoS attacks and explains previous studies. §3 introduces our motivation, research goal, and challenges. §4 depicts the detailed design of ROKI. §5 describes how we implemented ROKI. §6 explains our case studies on μDoS attacks and evaluates the performance overhead and the effectiveness of blocking in ROKI. §7 discusses some limitations of ROKI and potential solutions. §8 introduces related work and §9 concludes this paper.

2 Background

We define a μDoS attack and characterize their properties by using real-world μDoS attacks as examples. We also explain existing approaches and their limitations, highlighting the motivation of ROKI's approach.

2.1 Low-volume DoS (μDoS) Attack

μDoS attacks aim to make victim servers unavailable with a small number of attack packets. They often rely on carefully crafted, benign looking packets to effectively exhaust the important system resources of victim servers, by exploiting their performance bugs or heavy operations. For example, the event handler poisoning attack [12, 55] overloads a single-threaded event-driven server (e.g., Node.js) by requesting expensive computations such as evaluating complicated regular expressions. Second, to exhaust both memory and CPU of a victim server, some attacks exploit its performance bugs. The Black-Nurse attack [57] exploits a vulnerability of a Linux-based firewall that consumes many CPU cycles and much memory to process "Destination Unreachable/Protocol Unreachable" messages of the ICMP protocol.

Existing countermeasures against μDoS attacks tend to be attack-specific, as discussed in [4]. For example, to mitigate the SSL/TLS Renegotiation attack, it is recommended to disable the SSL/TLS Renegotiation protocol [45]. Similarly, to address the event handler poisoning attack described above, typical countermeasures are to estimate the complexity of each regular expression. Not surprisingly, it is recommended to restrict the number of active sessions a client can initiate in order to mitigate the session-pool μDoS attacks [8, 12, 55]. We believe countering μDoS attacks by exploiting attack-specific characteristics falls short in mitigating unforeseen, constant threats of μDoS attacks.

2.2 Defenses Based on Resource Profiling

Two recent countermeasures [12, 36] have been demonstrated to identify suspicious requests for CPU exhaustion by profiling resource usages. Rampart [36] measures per-function CPU time for handling individual requests. It is effective in detecting anomalous requests that incur drastically longer execution time than normal requests. Node.cure [12] aims to mitigate event handler poisoning attacks using timeout exceptions. Node.cure defines wall-clock timeout values for specific event handlers, so they would not spend more time to process requests than the defined timeout value.

These two approaches are effective in detecting and mitigating the described μDoS attack scenarios, but are hardly possible to generalize further to mitigate other types of μDoS attacks, for the following reasons. First, they profile resource usages either in a too course-grained manner (i.e., per process)—failed to attribute specific types of resource exhaustion, or only at a too high level (i.e., application layer)—failed to accurately accommodate system-wide noises such as context switching and interrupt timing. Second, they fail to handle μDoS attacks targeting kernel-level resource exhaustion such as link, network, and transport layers, which rapidly become popular in recent years (e.g., SegmentSmack and FragmentSmack in 2018 [39, 40]). Since commodity operating systems such as Linux and Windows are monolithic, it is challenging to profile each network layer in a non-intrusive manner. SplitStack [4] attempts to overcome this problem by splitting the network stack layers and profiling each of them. However, it demands huge kernel modification. Third, they focus on a single type of hardware resources (i.e., CPU), thereby failing to provide an accurate view to multiple hardware resources as well as software-abstracted resources such as the connection pool. Since it is not uncommon to exhaust multiple resources simultaneously [28], we need better a μDoS mitigation scheme that can accurately reason about a diverse set of resource consumption together.

2.3 Hardware Performance Counter (HPC)

HPCs are hardware units to count low-level events of micro-architecture during runtime [61]. With HPCs, we can efficiently profile various micro-architectural events related to program execution, such as retired CPU instructions, cache hits or misses, and branch predictions and misses. The number of HPC registers, however, is quite small (e.g., four in Intel CPUs [23] and six in AMD and ARM CPUs [1, 2]) such that careful scheduling is necessary to fully leverage them to monitor various events simultaneously.

3 Motivation and Challenges

In this section, we explain our motivation and research goal, and the challenges we have to overcome.

3.1 Motivation and Research Goal

Although μDoS attacks are serious security threats, we still lack an effective and general defense mechanism against them (§2). We believe resource-usage profiling is a promising direction to cope with μDoS attacks because, regardless of which tricks they exploit, they eventually aim to exhaust important system resources. Existing approaches, however, are not accurate enough to figure out the direct relationship between each request and resource usage in different layers for various types of resources (§2.2).

ROKI is designed to solve this challenging problem by enabling *data-driven resource tracking* to avoid heavy resource usage rather than attempting to detect attacks. Since μDoS attacks exploit benign yet expensive operations or unknown performance bugs, it is almost impossible to figure out their intention. Instead, ROKI identifies which data (i.e., network packets) lead to heavy resource consumption to block or postpone processing further packets from the origin. This protects any type of system resources from potential exhaustion. From the viewpoint of system resources, whether a packet intended to attack them is meaningless because in either case they cannot serve the packet and subsequent ones if they are (almost) saturated.

3.2 Challenges

Realizing a data-driven resource tracking system, ROKI, is challenging, especially because we aim to make it highly accurate, efficient, and universal. We specify the three critical challenges of ROKI and explain how we tackle each one.

C1. Accurate resource usage tracking at each layer. ROKI aims to prevent each packet from exhausting system resources at each network stack layer. This is because some μDoS attacks (e.g., BlackNurse, SegmentSmack, and FragmentSmack [39, 40, 57]) tend to exhaust resources only at a specific layer, which can be hidden when we solely monitor system-wide resource usage. ROKI solves this challenge by probing the resource usage at the entry functions of each layer that every packet should go through (§4.4).

C2. Efficient resource usage tracking. Per-packet and per-layer resource usage profiling can induce significant computation and memory overhead. ROKI solves this challenge by using HPCs to accurately and efficiently monitor CPU and memory usage (§4.4).

C3. Universal mitigation. Universal defense mechanisms against μDoS attacks are necessary to deal with variants or unknown attacks. Fixing individual problems and monitoring specific resources cannot achieve such goals. ROKI solves this challenge by monitoring various system resources at different layers simultaneously to detect and block suspicious packets with heavy resource consumption.

4 ROKI

ROKI is designed with three main design principles: monitoring resource usage in a fine-grained (per-packet and per-layer in the kernel as well as the application) and unified manner while minimizing performance overhead. We first explain ROKI's threat model and depict its design in detail.

Threat model. ROKI aims to mitigate μDoS attacks with low capacity, slow speed, and benign looking. Considering a large volume of network traffic, i.e., DDoS, and signature- and behavior-based filtering are out of its scope. Different countermeasures [38, 54, 65] can be used to deal with them. ROKI differentiates and blocks each suspicious host according to its IP address, implying that it might be vulnerable to IP spoofing attacks. Preventing IP spoofing attacks is a challenging problem requiring other advanced mechanisms (e.g., ingress and egress filtering [48] or IP authentication header [27]). We plan to adopt such mechanisms to make ROKI be robust against IP spoofing. Lastly, since ROKI relies on HPCs to profile resource usages, it requires either bare-metal machines [3, 44] or virtualized performance counters [20].

4.1 Overview

Figure 1 shows a design overview of ROKI. ROKI consists of three components: *resource profiler*, *system watchdog*, and *data handler*. The resource profiler, the main component of ROKI, monitors the resource usages for handling each packet at each layer of the network stack, including the link, network, transport, and application layers. To achieve this goal, the resource profiler injects probing code into all of the layers to keep track of resource usage. The probing code leverages HPCs to efficiently profile resource usage, such as performance monitoring unit (PMU) and memory bandwidth monitoring (MBM). The data handler retrieves per-packet resource-tracking information for further analysis and resource-exhaustion mitigation, such as temporal blocks on suspicious clients that routinely send expensive packets.

4.2 Components

We explain the three main components of ROKI: resource profiler, system watchdog and data handler. They run asynchronously to minimize performance degradation.

Resource profiler. The resource profiler monitors the resource consumption of each packet throughout the system, following its data flow from the link layer to the application layer. To avoid directly modifying the Linux kernel and server applications, we implemented the resource profiler on top of the bcc framework [24] along with customization for MBM tracing. The bcc project seamlessly integrates probing technologies (e.g., kprobe, uprobe, usdt, and eBPF [10, 14, 15, 19, 35]) into a single framework, which helps us write the tracing code.

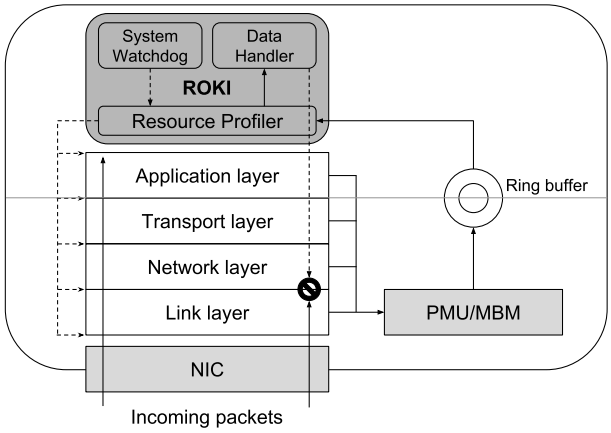

Figure 1: Design overview. ROKI consists of resource profiler, system watchdog, and data handler. It performs data-oriented resource profiling with HPCs while blocking suspicious clients based on profiling results.

During initialization, the resource profiler populates eBPF code into probing points located at the entry and exit points of each layer. The inserted code reads the PMU and MBM counters whenever a packet hits any of these probing points, calculates the difference between the two probing points of every protocol layer, and stores them in a key-value store. Whenever a packet is completely processed, its profiling result is delivered to the resource profiler running in user space.

System watchdog. The system watchdog monitors system-wide resources to selectively activate blocking. A small Python module periodically checks system-wide CPU and virtual memory usage, and the number of established connections. Based on the given thresholds for each system-wide resource, the watchdog activates the data handler's blocking feature when it observes any suspicious resource usage. Later, if ROKI remedies an attack such that the resource pressure is relieved, the watchdog deactivates the blocking function to minimize benign clients who encounter the block.

Data handler. The data handler has the two roles: logging and blocking. Logging allows an administrator to analyze the behaviors of attacks during real time or off time. Also, the data handler blocks suspicious clients. Using the mitigation algorithm discussed in §4.5, the data handler examines clients with corresponding profiling information to decide what client among them should be blocked.

We used passive and active approaches together to block a certain client. Once ROKI determines a client as suspicious, the data handler spawns a thread to block the client asynchronously. The blocking thread first blocks the suspicious IP address with iptables [42] or nftables [43], and then actively disconnects established sessions with the blocked IP address by injecting a shutdown system call into the target process that manages the sessions via Frida [50]. After the blocking time configured by the administrator passes, the thread unblocks

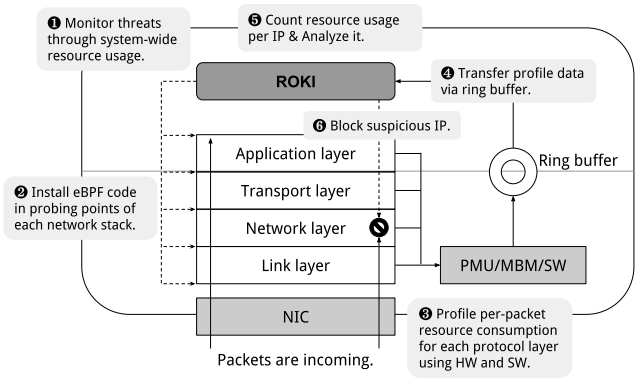

Figure 2: Overall workflow. When ROKI begins, it compiles and populates eBPF code at the entry points of each network layer, which reads PMU and MBM counters whenever packets hit the probing points. These values are conveyed via a ring buffer to the user space daemon of ROKI. Based on these profile results, ROKI determines what client is suspicious and blocks it for given seconds.

Layer	Entry functions
Application	
Apache	`ap_invoke_handler`, `ssl_hook_pre_connection`
Transport	
TCP	`tcp_v4_rcv`
UDP	`udp_rcv`
ICMP	`icmp_rcv`
Network	`ip_rcv`
Link	`__netif_receive_skb`

Table 1: Probing points used by ROKI. For the transport layer, ROKI uses three probing points: `tcp_v4_rcv`, `udp_rcv` and `icmp_rcv`. `ap_invoke_handler` and `ssl_hook_pre_connection` are used by Apache, as probing points for the application layer.

the client.

4.3 End-to-end Workflow

Figure 2 shows a detailed workflow of ROKI. First, ROKI starts with basic policies including the default blocking time, inspecting window period, whitelist, system resource thresholds for turning on and off the resource profiler, and resource thresholds to restrict the resource usage of each host. Then, the system watchdog and the resource profiler start to monitor system-wide resources and per-packet resource usage, respectively. If the watchdog detects abnormal activities against the given policies for those system resources, it activates the blocking function of the data handler.

The resource profiler monitors the kernel and the target application simultaneously but asynchronously. More specifically, it consists of two threads that listen to the kernel and application, respectively. During initialization, the resource profiler installs eBPF code into probing points, i.e., the entry functions of each layer, shown in Table 1.

Once a packet arrives at the server, it is processed throughout each layer of the network stack in order where ROKI

```
int will_enter_net_transport_layer(struct pt_regs *ctx)
{
    u32 cpu = bpf_get_smp_processor_id();
    PacketData* packet_data = packet_data_percpu.lookup(&cpu);
    if (!packet_data || !packet_data->src_addr)
        return 0;

    u64 value = 0;
    value = instructions.perf_read(cpu);
    packet_data->instructions[NET_LAYER_TRANSPORT] = value;

    value = mbm.perf_read(cpu);
    packet_data->mbm[NET_LAYER_TRANSPORT] = value;
    return 0;
}

int will_leave_net_transport_layer(struct pt_regs *ctx)
{
    u32 cpu = bpf_get_smp_processor_id();
    PacketData* packet_data = packet_data_percpu.lookup(&cpu);
    if (!packet_data || !packet_data->src_addr)
        return 0;

    u64 from = data->instructions[NET_LAYER_TRANSPORT];
    u64 diff = from ? instructions.perf_read(cpu) - from : 0;
    packet_data->instructions[NET_LAYER_TRANSPORT] = diff;

    from = data->mbm[NET_LAYER_TRANSPORT];
    diff = from ? mbm.perf_read(cpu) - from : 0;
    packet_data->mbm[NET_LAYER_TRANSPORT] = diff;
    return 0;
}
```

Figure 3: Simplified eBPF code snippet that profiles resource usage at the transport layer. It invokes the `perf_read` to retrieve PMU and MBM values. To keep consistency, we extended the bcc framework to allow accessing the MBM counter.

populates eBPF code. When a packet hits a probing point, the eBPF code resolves the IP address of the packet first, reads the PMU and MBM counters afterward, and stores the counts as initial values in a key-value store at the end. Figure 3 shows the simplified code that performs this process at the transport layer in the kernel. If the server has multiple active CPU cores, it needs multiple key-value stores for each core. The initial value in the key-value store is used to calculate resource usage for a single network layer when the processing packet leaves the current layer or proceeds to the next layer. Since the CPU core used to process a packet may change, we also monitor this activity via a separate probe for `finish_task_switch()`. When a CPU switch occurs, we calculate intermediate resource usage on the leaving core, save the intermediate result in the key-value store, and aggregate resource usage on the switched core into it later. Likewise, by installing an additional probe for `inet_csk_accept()`, we monitor the number of connections newly established for a client while a request sent by the client is processed. All of the profiling information will be conveyed to the user-space data handler via a ring buffer for further analysis and mitigation.

The data handler groups per-packet profiled data from the resource profiler based on the IP address, and holds them in a key-value store. Meanwhile, it discards stale data in the key-value store according to configuration. To decide which client is suspicious, the data handler uses the algorithm described in §4.5. With the data received in the past, the handler ranks

IP addresses per resource and determines which client ranks at the top in terms of each resource usage. If the sum of resource usage of the top ranked IP violates a policy that the administrator sets, the data handler concludes that the top ranked address is suspicious and prevents the corresponding client from accessing the server for a while. The data handler uses passive and active approaches for this temporal blocking: updating iptables rules to block the suspicious IP address and forcefully disconnecting already established sessions.

4.4 Data-oriented Resource Usage Profiler

For each packet entering into a server, ROKI tracks how many resources have been used to process them by following its data flow within the server from the link layer to the application layer. A naïve approach to achieve this goal is to instrument all functions that receive a network packet as input to profile resource usage, which results in too many changes and heavy performance overhead. Rather, ROKI focuses on the *entry functions* of each layer (e.g., `ip_rcv`, `tcp_v4_rcv`) that every network packet goes through [58]. By measuring resource usage at the entry functions of different layers and comparing them, ROKI is able to determine how many resources have been utilized for each packet at each layer. Also, instead of directly modifying or instrumenting the entry functions, ROKI uses the Linux kernel's tracing functionalities (e.g., kprobe, uprobe, and eBPF [10, 14, 19, 35]) to dynamically inject probing code to them (§5). This flexible approach allows much portability for developers to apply ROKI to any distributed system beyond a single end point machine by defining data to track across the distributed system.

To accurately and efficiently profile resource usage, ROKI's probing code uses the PMU to measure how many CPU instructions were retired and the MBM to measure how much memory was accessed during processing each packet.

4.5 Mitigating Resource-exhaustion

The data handler and resource profiler of ROKI work together to mitigate system resource exhaustion resulting from potential μDoS attacks, by temporarily blacklisting suspicious clients identified by ROKI that heavily occupy system resources. Our approach consists of two main steps: (1) identifying which client uses resources the most and (2) determining if its resource usage violates an administrator's policy. When a packet arrives on the server, ROKI profiles how it affects system resource usage and maintains this information for each client that has a unique IP address. The gathered data is kept for a certain time frame determined by an administrator. Then, ROKI ranks the clients according to how many resources they have used. We anticipate that the top-ranked client will continuously spend many resources as usual such that it is the most beneficial candidate to blacklist.

The next step is to determine whether the resource usage of the top-ranked client violates the given policies by administrators. Since ROKI always monitors the resource usages of each request, it can calculate average resource usages to determine reasonable threshold. Administrators would use static or dynamic threshold to determine suspicious clients. For simplify, this paper assumes that they use static threshold.

It is worth noting that blocking clients based on resource usage is mainly to keep a server live longer, not for detecting μDoS attacks. ROKI blocks clients only when the server is almost out of resources, implying that the server eventually fails to serve further requests from other clients. In addition, those temporarily blocked clients can always retry the failed requests later when the server is no longer busy.

5 Implementation

We implemented ROKI for securing the Apache (v2.2.13 and v2.4.18) running in a Linux machine (Fedora 27 powered by kernel version 4.13.9). Note that we consider multiple versions of Apache to reproduce μDoS attacks with their original targets. ROKI is implemented with 2,522 lines of Python code and 2,953 lines of C-like eBPF code. We will open source the entire code of ROKI.

6 Evaluation

In this section, we evaluate ROKI to answer the following three questions:

- **Attack detection:** How effective is ROKI in detecting real-world μDoS attacks targeting CPU (§6.1.1), memory (§6.1.2), and connection pool (§6.1.3)?
- **Quality of Service (QoS):** How effective is ROKI in maintaining latency? (§6.2)
- **Performance overhead:** How much performance overhead does ROKI incur to profile packets? (§6.3)

Experimental setup. We evaluated ROKI in a 1GbE local network that consists of four machines, acting as server, attacker, benign client, and latency monitor. The victim server protected by ROKI had two Intel Xeon E5-2687W v4 CPUs (24 cores, 3GHz) and 252 GB of memory. The server ran Linux kernel version 4.13.9. In our evaluation, we intentionally enabled only four cores of the server to easily exhaust it unless otherwise stated. The attacker machine had an Intel Xeon CPU E7-4820 CPU (16 cores, 2GHz) and 125 GB of memory. The benign client machine was equipped with an Intel Core i7-6600U CPU (4 cores, 2.6GHz) and 19 GB of memory. The latency monitor machine had an Intel Xeon CPU E7-4820 CPU (16 cores, 2GHz) and 252 GB of memory. To check response time, the monitor periodically sent the server an ApacheBench (ab) request every 0.5 seconds and calculated the round-trip time.

Configuration	Default value
Blocking time	5 seconds
Windowing time for inspection	3 seconds
Monitoring interval for system resource	0.1 seconds
Ring buffer size per core	16 MB
Conditions to enable/disable Resource Profiler	
CPU usage	75% / 35%
Memory usage	75% / 50%
Connection pool	75% / 35%
Instruction Thresholds	
Application	300,000
Transport	45,000,000,000
Network	1,000,000,000
Link	80,000,000,000
MBM Thresholds	
Application	1,000,000,000
Transport	50,000,000,000
Network	500,000,000
Link	1,500,000,000,000
Connection Threshold	6

Table 2: Default configuration used for our evaluation.

Default configuration. ROKI is fully configurable for administrators in terms of its blocking period, windowing time for inspection, system resource monitoring period for the watchdog, and thresholds for each network layer. We summarize the configuration used for our evaluation in Table 2.

Interpretation of experimental results. Before diving into each experiment in detail, we explain how to interpret the profiling results from ROKI. Figure 4 shows the CPU utilization of the server (i.e., the number of retired CPU instructions) for handing individual network packets at each network layer under Apache Range Header attacks profiled by ROKI. We monitored the server for 60 seconds where the μDoS attack was conducted. Here, gray dots represent the packets from the attacker machine and black dots represent the packets from the benign client machine and the latency monitor. The top-most plot, which indicates the latency measured at the latency monitor, is provided for reference and not used by ROKI for detection and mitigation. When ROKI was fully initialized, the benign client started sending requests to the server from the 5.5th second (black dots) and the attacker started the attack from the 8.5th second (gray dots) and repeated it every 4.5 seconds. Each attack lasted for 3 seconds. During the attack, the server could not handle benign requests such that its latency increased. However, in respite from the attack, it restarted serving benign requests.

By carefully analyzing the result, we found that the server utilized more CPU resources for handling the packets from the Apache Range Header attack than those for handing benign packets at the application layer. Thus, by blocking the clients sending expensive packets, ROKI has a chance to mitigate this μDoS attack.

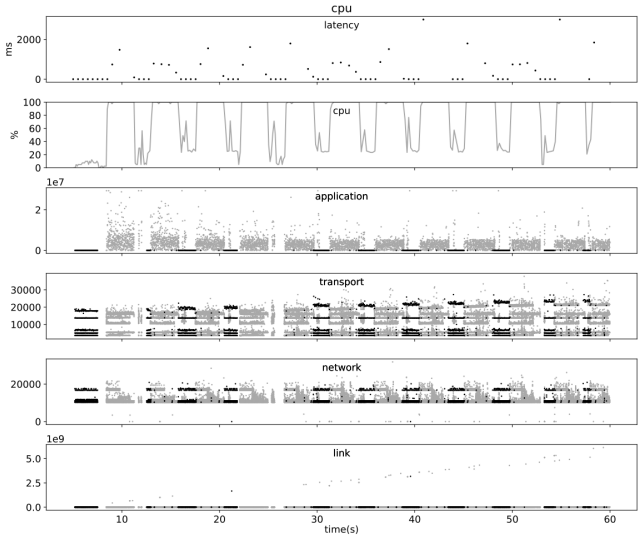

Figure 4: Apache Range Header attack profiled by ROKI without intervention. Gray and black dots represent the number of retired CPU instructions for handling each packet from the attacker and benign client at each layer, respectively. The server's latency, system CPU usage, and number of instructions used in the four network layers are displayed in order from the top-most plot.

6.1 Mitigating Real-world μDoS Attacks

To know the effectiveness of ROKI, we reproduce three representative real-world attacks, FragmentSmack (CPU), Apache Range Header (memory), and Slowloris (connection pool), against a server protected by ROKI.

6.1.1 FragmentSmack

FragmentSmack [40] is a CPU exhaustion attack that exploits a performance bug in reassembling IP fragments at the network layer. In the Internet, senders can break large packets (i.e., bigger than the maximum transmission unit) into a number of fragments, which will be reassembled by receivers in regular sequence. Until the entire fragments arrive, receivers have to queue arrived fragments while arranging them in order according to their offset values. FragmentSmack aims to prolong this process by creating and transmitting small fragments with arbitrary offsets while discarding last fragments, whose value of the more fragment (MF) bit is zero, to prohibit complete packet reconstruction.

Methodology. We reproduced FragmentSmack according to [37, 40, 51, 52]. With Scapy [49], we created 64 KB UDP packets and broke them into 8 B fragments with the same ID but arbitrary offsets. Then, we used 64 workers to transmit the fragments to a victim concurrently in arbitrary order while discarding fragments with MF=0. Single IP FragmentSmack saturates one CPU core [51], so, in this experiment, we activated two cores of the victim server for resource saturation. We increased the high and low thresh-

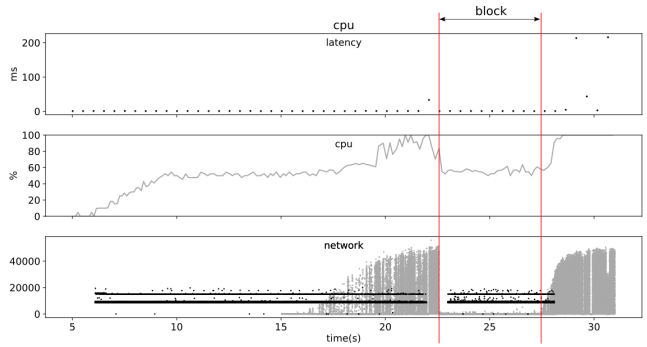

Figure 5: FragmentSmack under ROKI's protection. FragmentSmack attacks the IP fragment reassembly at the network layer. ROKI detects its heavy resource usage at the network layer and blocks it.

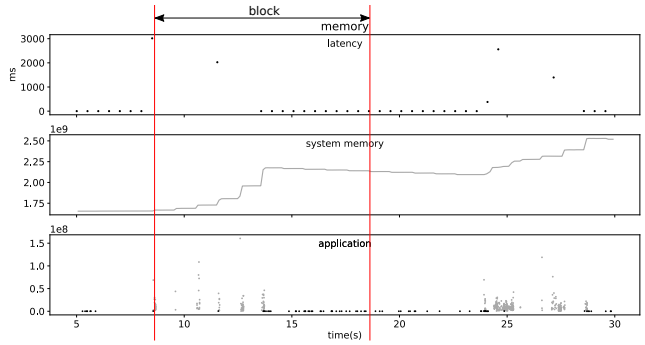

Figure 6: The Apache Range Header attack exhausting memory in the application layer. ROKI prevented this attack from exhausting the system memory.

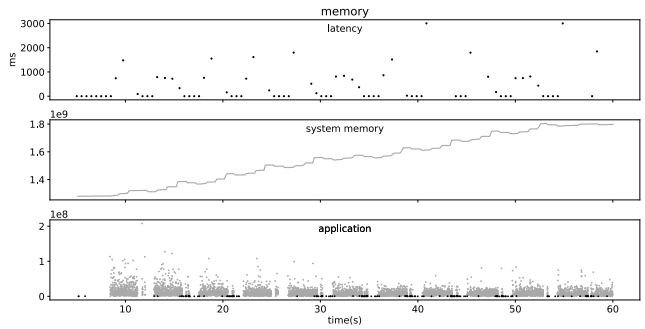

Figure 7: The Apache Range Header attack profiled by ROKI without blocking. The system memory usage kept increasing due to the attack.

old for IP fragmentation, `net.ipv4.ipfrag_high_thresh` and `net.ipv4.ipfrag_low_thresh`, to 64 MB and 48 MB, respectively, at the victim server to make it have a queue long enough to incur busy defragmentation. Instead of iptables that does not handle fragments, we used nftables [43] to block the source IP addresses of suspicious fragments.

Experimental results. Figure 5 shows how ROKI mitigated FragmentSmack. The victim server received benign requests from the 6th second and malicious requests from the 15th second. As fragments from the attacker were queued, we observed that the victim's CPU usage increased. ROKI blocked the origin of these fragments at the 25th second because they consumed CPU beyond the given threshold at the network layer. During the blocking period, the victim server was able to serve benign requests because nftables discarded malicious fragments as soon as they arrived.

6.1.2 Apache Range Header attack

The Apache Range Header attack [17] is a μDoS attack that overloads the CPU and memory of a victim server by exploiting a protocol design flaw in the Apache web server. More specifically, the HTTP protocol allows a client to request multiple overlapped ranges in a single request, which makes the server perform large fetches that are inefficiently kept in memory. Although the Apache Range Header attack exhausts both CPU and memory, we focus on memory implication of the attack in our evaluation.

Methodology. We used the Apache Killer script [29] against httpd-2.2.13 which is vulnerable to the Apache Range Header attack. For this experiment, we used ten-second blocking periods to clearly see ROKI's effectiveness.

Experimental results. Figure 6 shows how ROKI mitigated the Apache Range Header attack. When malicious requests arrived in the server at the 8th second, ROKI successfully detected the abnormal memory movement in the application layer via high MBM values even before the overall system memory usage increased. The suspicious IP was blocked at the 8th second, but the active memory of the system increased during the blocking period. This was because the server had to process arrived requests before the IP was blocked. However, the active memory steadily decreased later (i.e., after the 14th second) until the next attack arrived. Without ROKI, the overall memory usage kept increasing (Figure 7).

6.1.3 Slowloris

The Slowloris attack [8] aims to occupy as many connections as possible to prohibit establishing further benign connections, by sending partial requests that do not complete.

Methodology. To reproduce Slowloris, we applied the Slowloris script [22] against httpd-2.4.18. We used netstat [60] for checking the total number of established connections. We also counted the number of new connections for each client, as explained in §4.3.

Experimental results. Figure 8 shows how ROKI mitigates the Slowloris attack. The benign requests (gray bars) that appeared since the 6th second consistently made a single connection. In contrast, the malicious requests (blue bars) that appeared since the 10th second abnormally established many connections in a very short time. This difference allowed ROKI to detect the malicious requests. Once the attacker was

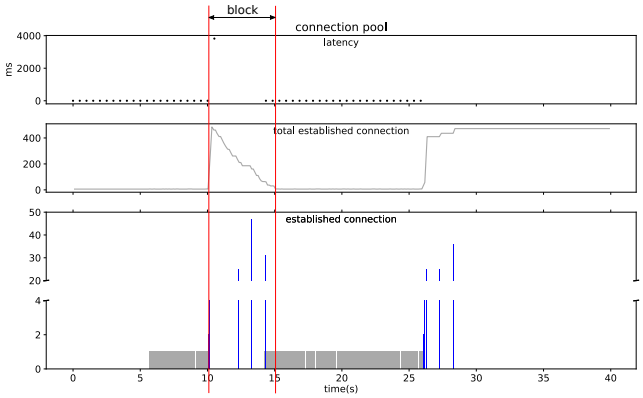

Figure 8: ROKI detected Slowloris by counting newly established connections in real time. Gray and blue bars represent the number of established connections with a benign client and an attacker, respectively. After ROKI blocked the attack at the 10th second, the connections from the attacker were closed such that the benign client could make new connections from the 14th second.

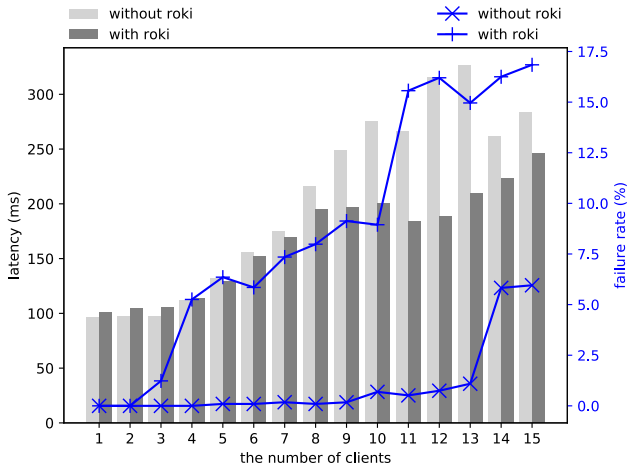

Figure 9: Average latency versus request failure rate with the increasing number of concurrent clients. ROKI maintains reasonable latency for clients even when a server is throttled. Whenever the system is under exhaustive resource uses, ROKI drops the most expensive requests at the moment, which indeed help system's resource allocation and so the QoS of the service.

identified, ROKI actively withdrew suspicious connections and injected a shutdown system call into web server processes via Frida [50]. The number of established connections kept decreasing from the 10th second to the 15th second, and ROKI served the benign client from the 14th second until the next attack occurred at the 26th second.

6.2 Latency versus Failure Rate

ROKI maintains good QoS even when a server is fully saturated with μDoS attacks or other benign but expensive requests by selectively blocking clients who consume the most resources. To verify whether ROKI satisfies this goal, we conducted an experiment to see how the average latency and failure rate (i.e., request drop rate) vary according to the load of a web server protected by ROKI. The server was serving mirrored Wikipedia pages [63], and multiple clients from 1 to 15 concurrently requested random Wikipedia pages from the server for 30 seconds. We emphasize that we set no attacker in this experiment because μDoS attacks can be constituted with legitimately formatted packets so differentiating attacker's packets from benign ones is not ROKI's main concern. We measured (1) the average latency only for *successful requests* (i.e., requests not dropped by ROKI) and (2) the failure rate (i.e., requests dropped by ROKI). Figure 9 shows the results. ROKI started to drop some requests when the number of concurrent clients was larger than 2. Without ROKI, the average latency of the server sharply increased in proportion to the number of concurrent clients. However, with ROKI, the slope was gentle with some failure rates. When the number of concurrent clients was 12, ROKI improved the average latency by up to 1.67× while dropping 16.2% of requests (122 out of 753 requests). This shows ROKI maintains QoS well by dropping some requests as expected. Interestingly, when the number of concurrent clients was larger than or equal to 14, we observed failed requests even without ROKI due to server saturation, improving the latency. These arbitrary packet drops, however, are problematic because, usually, there are more normal clients than attackers (or heavy clients) such that benign clients would observe more failures compared with the others.

6.3 Profiling Overheads

ROKI's fine-grained resource accounting unavoidably demands additional CPU and memory resources, resulting in degraded latency and throughput, and more memory consumption. In this section, we evaluate the profiling overheads by using Apache Benchmark and calculating additional memory requirements.

Latency and throughput. We measured how ROKI affected the latency and throughput of a web server by running http-2.4.33 on the server machine protected by ROKI and Apache Benchmark (ab) through a client machine. The web server enabled 48 cores and was serving a static page. ab kept requesting the static page while varying concurrency from 10 to 100, and 512 that was the maximum number of sessions the web server supported. ROKI does not block any connections for this microbenchmark to measure its pure profiling overhead. We repeated each experiment 10 times and averaged the results. Figure 10 shows the latency and throughput of the server with and without ROKI whose overhead stably moved between 3%–5% according to the number of concurrent connections. The minimum overhead in latency was 3.48% when the number of connections were 60, and the one in throughput was 3.55% when the concurrent connections were 30. In

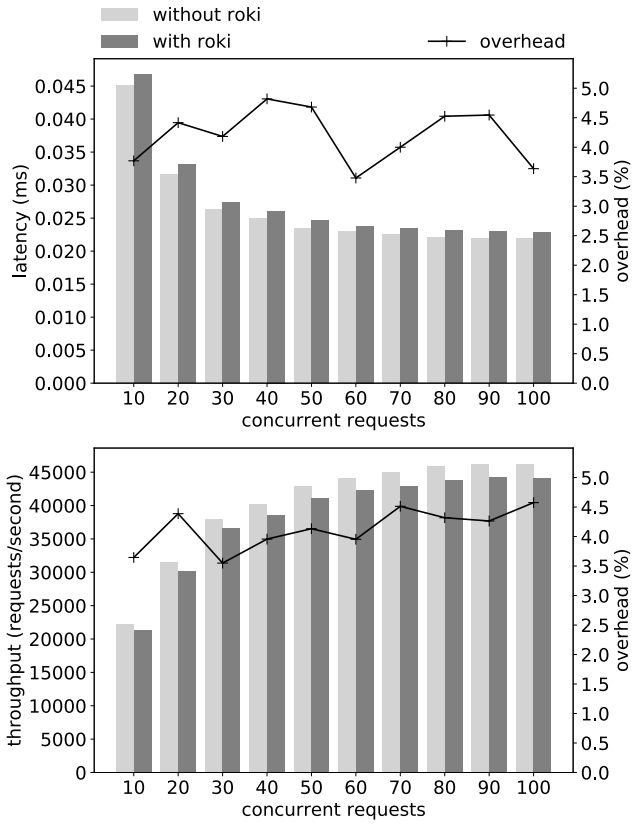

Figure 10: Apache benchmark result with ROKI. The top figure shows the overhead caused by ROKI in latency and the bottom one shows the overhead in throughput. Light gray and dark gray bars represent results without and with ROKI, respectively. ROKI introduced 3.48%–4.82% overhead in latency and 3.55%–4.58% overhead in throughput while the number of concurrent connections was varied from 10 to 100.

the worst case where the number of concurrent requests was 512, the overhead in latency was 5.98% and the overhead in throughput was 4.19%.

Memory overheads. For each IPv6 packet, ROKI requires 126 bytes of per-packet profiling data; meaning that for queuing 100,000 packets for analysis, it requires around 12 MB of additional memory uses. Currently, ROKI uses a 16 MB ring buffer for each core to satisfy this requirement. The per-packet profiling data of kernel space consists of timestamp (8 B), CPU ID (4 B), source IP address (4 or 16 B), instruction counters for three layers (3×8 B), MBM counters for three layers (3×8 B), and that of userspace has one more field to store program ID (4 B).

7 Discussion

In this section, we discuss some limitations of ROKI and its optimal configuration.

Ring buffer overflow. ROKI is not able to comprehensively analyze or block suspicious packets if a ring buffer overflows. ROKI uses a ring buffer to deliver per-packet profiling information collected at each layer of the in-kernel network stack to the user-space resource profiler. Since the ring buffer is finite, it can overflow if too many concurrent packets are delivered or the resource profiler does not efficiently consume profiling information, resulting in data loss. Thus, an administrator needs to configure the ring buffer's size sufficiently to not suffer from buffer overflow unless there is exceptional network traffic (likely due to a DDoS attack).

As explained in Figure 6.3, the current ring buffer size ROKI is using, 16 MB is enough to handle normal HTTP requests. We also want to emphasize that the ring buffer overflow does not hurt the server's normal operations. This is because it only maintains additional profiling information used by ROKI not by the kernel or server application.

HPC scalability. The HPC has a limitation in scalability: that is, the number of performance events it can concurrently monitor is limited. For example, the Intel Xeon E5-2687W v4 CPU that we use for evaluation supports three fixed-function performance counters and four programmable counters per logical core [23]. Currently, ROKI only uses two programmable counters to monitor retired CPU instructions and memory bandwidth, so it does not suffer from a scalability problem. If we want to monitor other performance events as well (e.g., cache misses and branch instructions) to detect other types of resource exhaustion, we need to rely on the kernel's time multiplexing of HPCs.

Optimal policy. Figuring out the optimal policy to determine when to block what clients is important. However, the optimal policy heavily depends on server configurations and administrators' performance goal. Without real environment setup and data, determining the policy makes no sense. Our best-effort strategy to determine threshold values was calculating average response time while varying request drop rates to find acceptable balance between them. We do not claim that it was the best approach, but we think that it is one of the feasible approaches to determine acceptable threshold.

8 Related Work

In §2.2, we discuss Rampart [36], Node.cure [12], and Split-Stack [4] to detect or mitigate μDoS attacks. Apart from these strategies, other studies inspire us to design and implement ROKI. We describe them in the rest of this section.

Tracing based profiling techniques. To detect abnormality in resource usage, ROKI relies on a data-oriented resource profiling technique that records resource consumption, tracing unique data. Other researchers also consider how to profile a system based on tracing techniques for different goals. Magpie [47] models CPU workload in distributed system, tracing events incurred by a request. After correlating OS level events, Magpie can figure out CPU usage per request. However, un-

like ROKI, it is based on event tracing, and relatively coarse-grained (i.e., no per-layer resource profiling). Also, it does not profile memory usage. Pip [46] proposed a bug finding technique for applications running on distributed system by comparing actual behavior and expected behavior. To check application behavior, Pip records paths by tracking explicit path identifiers. Unfortunately, Pip is not helpful to defeat μDoS attacks whose behavior is almost legitimate. A general tracing framework was introduced by X-Trace [16]. It enables reconstruction of user's task tree and a comprehensive view across layers and applications. However, it does not deal with a resource accounting technique to protect the system from μDoS attacks as ROKI does. Besides, it requires source code change to embed metadata into target software for tracking. Retro [25] presents a resource management framework for a distributed system of which resource is shared by multiple tenants. It aims to achieve desired performance guarantee or fairness for each tenant through profiling per-workflow resource load. Retro's granularity for resource profiling is coarser than ROKI's one, and its per-workflow resource profiling does not work for μDoS attacks since the attack tends to exhaust resources on an end-point machine. Monitoring resources across distributed machines hinders the attack from being detected.

DoS/DDoS defenses. DoS and DDoS attacks aim to send a large number of packets to a victim server. Especially, DDoS relies on a large number of zombie machines (or bots) to generate an excessive volume of network traffic. Since they are old and popular attacks, many researchers have already analyzed them and proposed effective countermeasures [38, 54, 65]. For example, researchers extract statistical patterns from the massive attack traffic to generate filtering rules. Also, they analyze command-and-control traffic between bots and their masters to take down the botnet. To defend against DDoS attacks, a large body of research explored monitoring and mitigation schemes based on packet signatures. Such schemes include Randomize-Then-Optimize randomization [64], detecting period pulse [56], spectral analysis [5], and modeling [32, 53]. Unlike schemes that are designed for specific protocols or rely on packet signatures, ROKI detects resource exhaustion using data collected from system performance monitors. DDoS defense mechanisms can complement ROKI to make a server secure against both DDoS and μDoS attacks.

μDoS attacks and defenses. Pioneering research on μDoS attacks [32] demonstrated that a vulnerability in the TCP timeout mechanism can be exploited with periodic, short-lived, low-volume traffic. Such attacks can be extended to Pulsing DoS (PDoS) attacks [34], which exploit the Adaptive-Increase-Multiplicative-Decrease (AIMD) algorithm implemented in the TCP protocol. The Boarder Gateway Protocol (BGP), which is used to perform routing sessions on commercial routers, was shown to be vulnerable to μDoS attacks [66]. μDoS attacks have been further generalized as Reduction of Quality (RoQ) attacks [21], which cause a system to perform below capacity.

Initial work on μDoS attacks proposed two approaches: router-assisted and end-point min-Retransmission Time Out randomization [32]. However, experiments performed in the initial work on μDoS showed that by limiting the peak rate and burst length of an attack, the proposed attack could still severely degrade throughput without being detected by the popular DoS detection algorithm Random Early Detection, Preferential Drop (RED-PD) [32].

Vanguard [33] detects μDoS attacks by monitoring anomalies in network events, such as abnormal traffic of outgoing TCP ACK signals or an imbalance of incoming and outgoing ACK signals. ROKI mitigates a wider range of attacks than Vanguard, with low overhead by comprehensively monitoring resource exhaustion using HPCs.

Hardware-based resource monitoring. Researchers started to use HPCs for security applications. SlowFuzz [59] aims to automatically detect the most expensive inputs for diverse, well-known algorithms by continuously measuring their resource usage with HPCs in a domain-independent manner. That is, it is a proactive approach to find performance bugs of a program, which can complement ROKI.

Several malware studies [13, 26, 62] use HPCs to check low-level, accurate behaviors of malware. They monitor a process with HPCs to determine whether its resource usage behaviors are similar to the behaviors of known malware. This line of research, however, needs to be improved because it is difficult to associate low-level HPC values with high-level user intention [67]. In contrast, ROKI does not suffer from this challenging problem because its goal is to determine which packet consumes many system resources, rather than to infer some intention from the packet's resource usage pattern.

9 Conclusion

A μDoS attack is challenging to defeat because of its low capacity, low speed, and legitimacy. ROKI protects an end-point server from this sophisticated attack with data-oriented resource usage tracking. It accurately monitors resource usages along the data flow per request, recognizes resource usage anomalies due to the request, and temporarily block the request origin in a unified manner. Evaluation shows that ROKI is effective against real-world μDoS attacks targeting CPU, memory, and connection pool in either kernel or user space with acceptable overhead.

References

[1] AMD. AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://www.amd.com/system/files/TechDocs/24593.pdf.

[2] ARM. ARM® Cortex®-A57 MPCore™ Processor Technical Reference Manual. http:

//infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488c/BIICBJEC.html.

[3] Jeff Barr. Amazon EC2 Bare Metal Instances with Direct Access to Hardware. https://aws.amazon.com/blogs/aws/new-amazon-ec2-bare-metal-instances-with-direct-access-to-hardware/.

[4] Ang Chen, Akshay Sriraman, Tavish Vaidya, Yuankai Zhang, Andreas Haeberlen, Boon Thau Loo, Linh Thi Xuan Phan, Micah Sherr, Clay Shields, and Wenchao Zhou. Dispersing Asymmetric DDoS Attacks with SplitStack. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)*, Atlanta, GA, November 2016.

[5] Yu Chen and Kai Hwang. Collaborative detection and filtering of shrew DDoS attacks using spectral analysis. *Journal of Parallel and Distributed Computing*, 2006.

[6] Cloudflare. Advanced DDoS Protection and Mitigation | Cloudflare. https://www.cloudflare.com/ddos/.

[7] Cloudflare. Cloudflare Rate Limiting. https://www.cloudflare.com/rate-limiting/.

[8] Cloudflare. Slowloris DDoS Attack. https://www.cloudflare.com/learning/ddos/ddos-attack-tools/slowloris.

[9] Evan Cooke, Farnam Jahanian, and Danny McPherson. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Proceedings of the Steps to Reducing Unwanted Traffic on the Internet on Steps to Reducing Unwanted Traffic on the Internet Workshop*, SRUTI'05, pages 6–6, Berkeley, CA, USA, 2005. USENIX Association.

[10] Jonathan Corbet. Extending extended BPF, 2014. https://lwn.net/Articles/603983.

[11] DARPA. Extreme DDoS Defense (XD3). https://www.darpa.mil/program/extreme-ddos-defense.

[12] James C. Davis, Eric R. Williamson, and Dongyoon Lee. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[13] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the Feasibility of Online Malware Detection with Performance Counters. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Tel-Aviv, Israel, June 2013.

[14] Matt Fleming. A thorough introduction to eBPF, 2017. https://lwn.net/Articles/740157.

[15] Matt Fleming. Using user-space tracepoints with BPF, 2018. https://lwn.net/Articles/753601.

[16] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. X-Trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, April 2007.

[17] Apache Foundation. Range header DoS vulnerability Apache HTTPD prior to 2.2.20, 2011. https://httpd.apache.org/security/CVE-2011-3192.txt.

[18] Google. Google | Project Shield | Free DDoS protection. https://projectshield.withgoogle.com/public/.

[19] Sudhanshu Goswami. An introduction to KProbes, 2005. https://lwn.net/Articles/132196.

[20] Brendan Gregg. The PMCs of EC2: Measuring IPC. http://www.brendangregg.com/blog/2017-05-04/the-pmcs-of-ec2.html.

[21] Mina Guirguis, Azer Bestavros, Ibrahim Matta, and Yuting Zhang. Reduction of quality (RoQ) attacks on internet end-systems. In *24th Annual Joint Conference of the IEEE Computer and Communications Societies*, 2005.

[22] Robert "RSnake" Hansen. Slowloris HTTP DoS, 2009. https://web.archive.org/web/20090822001255/http://ha.ckers.org/slowloris.

[23] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html.

[24] IO Visor Project. BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more. https://github.com/iovisor/bcc.

[25] Rodrigo Fonseca Jonathan Mace, Peter Bodik and Madanlal Musuvathi. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Oakland, CA, May 2015.

[26] Mikhail Kazdagli, Vijay Janapa Reddi, and Mohit Tiwari. Quantifying and Improving the Efficiency of Hardware-based Mobile Malware Detectors. In *Proceedings of the 49th Annual IEEE/ACM International*

Symposium on Microarchitecture (MICRO), Taipei, Taiwan, October 2016.

[27] Stephen Kent. IP Authentication Header. https://www.ietf.org/rfc/rfc4302.txt.

[28] Sean Michael Kerner. Defending Against The 'Apache Killer' Exploit, 2011. https://www.esecurityplanet.com/network-security/Defending-Against-The-Apache-Killer-Exploit-3939081.htm.

[29] Kingcope. Apache Killer, 2011. http://seclists.org/fulldisclosure/2011/Aug/175.

[30] Amit Klein. Multiple vendors XML parser (and SOAP/WebServices server) Denial of Service attack using DTD, 2002. https://www.securityfocus.com/archive/1/303509.

[31] Mohit Kumar. 1.7 Tbps DDoS Attack – Memcached UDP Reflections Set New Record. https://thehackernews.com/2018/03/ddos-attack-memcached.html.

[32] Aleksandar Kuzmanovic and Edward W. Knightly. Low-rate TCP-targeted Denial of Service Attacks: The Shrew vs. The Mice and Elephants. In *Proceedings of the 14th ACM SIGCOMM*, pages 75–86, Karlsruhe, Germany, August 2003.

[33] Xiapu Luo, Edmond WW Chan, and Rocky KC Chang. Vanguard: A new detection scheme for a class of TCP-targeted denial-of-service attacks. In *Network Operations and Management Symposium*, 2006.

[34] Xiapu Luo and Rocky KC Chang. On a new class of pulsing denial-of-service attacks and the defense. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2005.

[35] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference*, San Diego, CA, 1993. Winter USENIX.

[36] Wei Meng, Chenxiong Qian, Shuang Hao, Kevin Borgolte, Giovanni Vigna, Christopher Kruegel, and Wenke Lee. Rampart: Protecting Web Applications from CPU-Exhaustion Denial-of-Service Attacks. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[37] Microsoft. ADV180022 | Windows Denial of Service Vulnerability. https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/ADV180022.

[38] Jelena Mirkovic and Peter Reiher. A Taxonomy of DDoS Attack and DDoS Defense Mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2):39–53, April 2004.

[39] MITRE. CVE-2018-5390. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-5390.

[40] MITRE. CVE-2018-5391. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-5391.

[41] MITRE. CVE-2018-6022. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6022.

[42] netfilter.org. iptables project. https://netfilter.org/projects/iptables/index.html.

[43] netfilter.org. The netfilter.org "nftables" project. https://netfilter.org/projects/nftables/.

[44] Oracle Cloud Infrastructure. Bare Metal Cloud Computing. https://cloud.oracle.com/bare-metal/features.

[45] Jorge Orchilles. SSL/TLS Renegotiation DoS attack, 2011. https://www.ietf.org/mail-archive/web/tls/current/msg07553.html.

[46] Charles Killian Patrick Reynolds and Janet L. Wiener. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[47] Rebecca Isaacs Paul Barham, Austin Donnelly and Richard Mortier. Using Magpie for request extraction and workload modeling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[48] Daniel Senie Paul Ferguso. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. https://www.ietf.org/rfc/rfc2267.txt.

[49] Philippe Biondi and the Scapy community. Scapy: Packet crafting for Python2 and Python3. https://scapy.net.

[50] Ole André V. Ravnås. FRIDA. https://www.frida.re.

[51] Red Hat, Inc. SegmentSmack and FragmentSmack: IP fragments and TCP segments with random offsets may cause a remote denial of service. https://access.redhat.com/articles/3553061.

[52] Remco Verhoef. Back to the 90's: FragmentSmack. https://isc.sans.edu/forums/diary/Back+to+the+90s+FragmentSmack/23998.

[53] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking algorithmic complexity attacks against a NIDS. In *Proceedings of the 22th Annual Computer Security Applications Conference (ACSAC)*, Miami Beach, FL, USA, December 2006.

[54] Gaurav Somani, Manoj Singh Gaur, Dheeraj Sanghi, Mauro Conti, and Rajkumar Buyya. DDoS attacks in cloud computing: Issues, taxonomy, and future directions. In *Computer Communications Vol.107*, pages 30–48, July 2017.

[55] Cristian-Alexandru Staicu and Michael Pradel. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[56] Haibin Sun, John Lui, and David KY Yau. Defending against low-rate TCP attacks: dynamic detection and protection. In *12th IEEE International Conference on Network Protocols (ICNP)*, 2004.

[57] TDC Security Operations Center. BLACKNURSE IT CAN BRING YOU DOWN, 2016. http://blacknurse.dk.

[58] The linux foundation. Kernel flow. https://wiki.linuxfoundation.org/networking/kernel_flow.

[59] Angelos D. Keromytis Theofilos Petsios, Jason Zhao. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[60] TLDP. Linux Network Administrators Guide. https://www.tldp.org/LDP/nag2/x-087-2-iface.netstat.html.

[61] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting hardware performance counters. In *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 59–67. IEEE, 2008.

[62] Xueyang Wang, Sek Chai, Michael Isnardi, Sehoon Lim, and Ramesh Karri. Hardware Performance Counter-Based Malware Identification and Detection with Adaptive Compressive Sensing. *ACM Trans. Archit. Code Optim.*, 13(1):3:1–3:23, March 2016.

[63] Wikipedia. Wikipedia:Mirrors and forks. https://en.wikipedia.org/wiki/Wikipedia:Mirrors_and_forks.

[64] Guang Yang, Mario Gerla, and MY Sanadidi. Defense against low-rate TCP-targeted denial-of-service attacks. In *9th International Symposium on Computers and Communications (ISCC)*, 2004.

[65] Saman Taghavi Zargar, James Joshi, and David Tipper. A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks. In *IEEE Communications Surveys & Tutorials (Volume: 15, Issue: 4, Fourth Quarter 2013)*, March 2013.

[66] Ying Zhang, Zhuoqing Morley Mao, and Jia Wang. Low-Rate TCP-Targeted DoS Attack Disrupts Internet Routing. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2007.

[67] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware Performance Counters Can Detect Malware: Myth or Fact? In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi'an, China, May–June 2016.