TUPLE SPACE EXPLOSION: A DENIAL-OF-SERVICE ATTACK
AGAINST A SOFTWARE PACKET CLASSIFIER

Levente Csikor†, Dinil Mon Divakaran‡, Min Suk Kang†, Attila Kőrösi‡, Balázs Sonkoly†‡, Dávid Haja‡, Dimitrios P. Pezaros*, Stefan Schmid†, Gábor Rétvári‡

†National University of Singapore, *Trustwave, ‡Budapest University of Technology and Economics, †‡MTA-BME Network Softwarization Research Group, University of Glasgow, †Faculty of Computer Science, University of Vienna

ABSTRACT
Efficient and highly available packet classification is fundamental for various security primitives. In this paper, we evaluate whether the de facto Tuple Space Search (TSS) packet classification algorithm used in popular software networking stacks such as the Open vSwitch is robust against low-rate denial-of-service attacks. We present the Tuple Space Explosion (TSE) attack that exploits the fundamental space/time complexity of the TSS algorithm.

TSE can degrade the switch performance to 12% of its full capacity with a very low packet rate (0.7 Mbps) when the target only has simple policies such as, “allow some, but drop others”. Worse, an adversary with additional partial knowledge of these policies can virtually bring down the target with the same low attack rate. Interestingly, TSE does not generate any specific traffic patterns but only requires arbitrary headers and payloads which makes it particularly hard to detect.

Due to the fundamental complexity characteristics of TSS, unfortunately, there seems to be no complete mitigation to the problem. As a long-term solution, we suggest the use of other algorithms (e.g., HaRP) that are not vulnerable to the TSE attack. As a short-term countermeasure, we propose MGGuard that carefully manages the tuple space and keeps packet classification fast.

CCS CONCEPTS
• Security and privacy → Denial-of-service attacks.

KEYWORDS
security, denial-of-service, data plane, packet classifier, open vswitch, software-defined networking, tuple space search, algorithmic complexity

ACM Reference Format:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CoNEXT ’19, December 9–12, 2019, Orlando, FL, USA © 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6998-5/19/12 ...$15.00
https://doi.org/10.1145/3359989.3365431

1 INTRODUCTION
Packet classification is one of the fundamental building blocks of various security primitives. From access control implementation to network traffic isolation to denial-of-service defenses, they all require to classify incoming packets into multiple groups for different purposes, e.g., block/redirect certain traffic. Packet classification, thus, should be highly efficient, dependable, and available.

With the proliferation of virtualization techniques, packet classification has become softwareized and been widely used in virtualized systems today. As an example, Open vSwitch (OVS) [58] (along with many other software switches; e.g., VPP [27], HyperSwitch [60], GSwitch [77]) is universally used in virtualized environments. OVS is massively used for basic switching and firewall implementation for the tenants in cloud hosting systems enabling them to further specify custom firewall rules for their own purposes [15, 71] (§2).

Moreover, dynamic filter rules updates for denial-of-service defense can be implemented with the software-defined networking (SDN) capability [8], particularly OpenFlow, in OVS [46].

In this paper, we evaluate whether the de facto packet classification algorithm widely used in many popular software switches (OVS as our running example) is robust against denial-of-service (DoS) attacks (§3). Our study has great importance since the targeted packet classification algorithm, namely, Tuple Space Search scheme (TSS) [65], is extensively used not only in traditional appliances but also in wide range of virtualized networked systems; just to name a few, software based intrusion detection systems [44], stateful NAT implementation [34, 49], cloud management systems [26, 72].

Our findings are alarming. Not only effective DoS attacks against the TSS packet classification are possible but also detection and mitigation of the attacks are hard. First, we present a new low-rate DoS attack, which we call the Tuple Space Explosion (TSE) attack (§3), against the TSS scheme (§4). We demonstrate that as little as 670 kbps of attack traffic from a single traffic source can easily degrade a single OVS instance from its full capacity of 10 Gbps to 2 Mbps when an adversary has a partial control over (or knowledge of) the access control rules installed in the targeted OVS (§5). When an adversary has no such access to her target (§6), we show, she can still achieve a significant degradation of 88% from the maximum capacity with low attack traffic volume.

One interesting aspect of the TSE attack is that it does not demonstrate any specific patterns of its attack traffic. Unlike the existing low-rate DoS attacks (e.g., algorithmic-complexity DoS attacks [16, 62], shrew attacks [43]) that send carefully-crafted attack packet sequences with specific traffic patterns to the target system, the TSE attack only requires arbitrary packet header fields
and message contents, along with arbitrary packet arrival times. This makes its identification hard as it is not straightforward to define a specific signature of the attack traffic.

In fact that the TSE attack is highly effective only with randomly-generated inputs implies that the attack does not depend on specific attack strategies. Instead, it relies on the internal state of the target switch. That is, there exist certain states of the target switch that makes it vulnerable to DoS attacks with randomly-generated inputs. We investigate such system states in terms of Access Control Lists (ACLs) of the TSS packet classification. Unfortunately, some commonly used ACL patterns (e.g., WhiteList+DefaultDeny) are shown to be particularly vulnerable to the TSE attack.

Lastly, the very fact that the target of the TSE attack is the packet classifier itself makes it hard to design countermeasures. Defending against the TSE attack would require to have another packet classifier to filter out suspicious packets. However, unfortunately, the additional packet classifier is also likely vulnerable to the same DoS attack if implemented with the TSS algorithm. To mitigate the TSE DoS attack, thus, one has to deploy a different packet classifier that is robust against the TSE attacks. Yet, this suggests that this second packet classifier could have been simply used as the main classifier in the first place!

The TSE attack exploits the fundamental space/time complexity of the TSS algorithm and thus no complete mitigation of the problem seems possible (§7). Therefore, as a long-term solution, we suggest to use other packet classification algorithms that are not vulnerable to the TSE attack. Hierarchical tries [32], HaRP [59], and Hypercuts [10] packet classification algorithms seem to be unaffected by the TSE attack directly, although more in-depth study may be required for comprehensive analysis.

As a short-term solution, we present a cache management scheme, which we call MFCGuard, that dynamically monitors the number of entries in the tuple space and removes less important ones to lower the performance overhead of the packet classification (§8). We show that MFCGuard can limit the performance degradation for the packets that are eventually allowed to the system. This guaranteed performance for the allowed packets is achieved, however, at the expense of much increased processing time for the packets to be denied by the ACL rules. We discuss some operational concerns of MFCGuard, particularly when used in cloud hosting systems, where the increased computation overhead of MFCGuard may affect the operation of the tenants’ workloads in the system.

2 BACKGROUND

Here, we describe the operation and fundamental building blocks of the most typical virtual switches, particularly, Open vSwitch (OVS) [58], and present the packet classification, called Tuple Space Search (TSS), used in OVS. Readers familiar with the packet classification algorithms in software switches may continue from §3.

2.1 Switching Stacks for Virtualization

Enterprises increasingly offload business-critical workloads to the public cloud to benefit from low infrastructure costs, high availability, and flexible resource provisioning. Reliable and efficient service provisioning heavily depends on the ability to efficiently switch traffic between the tenants’ workloads and the outside world.

In this paper, we use OVS as our running example, but the presented vulnerabilities might affect other TSS-based software switches (e.g., VPP [27], Hyperswitch [60], GSitch [77]). OVS [58] is an open source, multilayer, production quality software switch that enables massive network automation through programmatic extensions [1]. It can be managed remotely through standardized control plane protocols [53, 57]. The OVS flow table describes the packet processing behavior to be implemented by the switching logic at a high level. Due to its flexibility, generality, and community support, OVS has been extensively used in cloud deployments [1].

The flow table of an OVS switch is an ordered set of flows, where each flow is a pair of (1) a wildcard rule, operating on specific protocol header fields (e.g., IP source address, ports) and designating packets that belong to the flow, and (2) an action, a set of packet processing primitives to be applied to packets matching the flow rule; e.g., “forward to port”, or “drop”.

Two flows in the flow table are said to overlap if there is a packet header that matches both. In this case, the matching flow that occurs first in the flow table takes precedence. For instance, in the sample ACL in Fig. 6, a packet with source IP address 10.0.0.1, source and destination ports 34521, and 443, respectively, matches both the second and the last flow entries with the first flow overriding the last one by higher priority.

In contrast, a flow table in which all rules are disjoint is order-independent because all packets have a single matching rule and equal priority (i.e., order is irrelevant). In general, this makes packet classification much simpler [42].

Most software switches (if not all) support order-dependent flow tables despite the performance benefit of order-independent tables because of the flexibility of the former. In virtualized environments (e.g., multiple tenants share a single software switch for access control), users with various networking knowledge configure the flow rules in the switches. The greater flexibility of order-dependent tables support rule wildcarding and flow priorities, which allow complex packet processing logics to be described concisely.

2.2 TSS for Fast Packet Classification

To cut down the prohibitive cost of packet classification, OVS adopts the well-known fast path/slow path separation principle [51]. The fast path comprises two layers of flow caches, and the slow path implements a complete representation of the flow table serving as a fallback when the fast path cannot decide on the fate of a packet. Only the first packet of each flow is subjected to full-blown flow-table processing, i.e., slow path, and the resulting flow-specific rules and actions are then registered in the flow caches; the rest of the flow’s packets take the fast path. This amortizes the cost of packet classification over subsequent packets of a flow, contributing to increased performance without loss of expressiveness and generality [13, 40, 47] (cf. §11.1 for the general flow-cache hierarchy).

Within the fast path, the microflow cache implements a transport-connection exact-match store where lookup occurs over all header fields, while the megafow cache (MFC) bundles multiple microflows into a single megaflow to impose common processing to the entire bundle [30, 49, 64]. In this design, the microflow cache merely serves as “short-term” memory and it is often exhausted even in normal operation (by default, it contains only a couple of
hundred entries). The lookup algorithm in the MFC relies on the TSS scheme [65], the prevailing packet classifier used to implement ACLs in other hypervisor switches as well (e.g., VPP [27], Hyper-Switch [60], GSwitch [77]). MFC generally saves on cache entries by a single megaflow covering, say, all incoming HTTP connections regardless of the source TCP port (i.e., TCP port wildcarded). In a nutshell, this is done by collecting the entries matching on the same set of header bits into a hash in which masked packet headers can be found fast. Then, masks and associated hashes are searched sequentially until the first matching entry is found.

Note that the TSS implementation in OVS does not know about flow priorities; thus the slow path ensures that MFC entries are all disjoint to make packet classification simpler yet introducing worst case exponential complexity (i.e., exhaustive linear search in the different masks). Correspondingly, as long as the number of masks is kept in a reasonable range (e.g., couple of hundreds masks), packet processing in the fast path is close to line rate. This property of the TSS is the very logic we aim to exploit in this paper.

3 TUPLE SPACE EXPLOSION: OVERVIEW
Here, we provide a high-level overview of the attack. First, we describe our threat model, then we show the essence of the proposed TSE attack and discuss the algorithmic complexity vulnerability of the TSS scheme using its implementation in OVS. Then, we show two different approaches of TSE, each posing different requirements and targets for the attacker.

3.1 Threat Model
We consider a general virtualized computing environment, where a targeted software switch is used for packet processing and basic network operations. This includes a typical multi-tenant cloud infrastructure whereby tenants lease resources in the cloud to deliver public services. Tenants may use cloud management system (CMS) APIs to set up their access-control list (ACL) rules in the underlying software switch to access-control, redirect, or log accesses to different resources [15, 36–38]. We consider that the internal algorithms of the data plane fabric is fully known to adversaries.

The attacker’s goal is to send some attack packets to the virtual switch, which when subjected to the implemented ACL will exhaust the underlying resources denying access to the rest of the users.

The adversary only needs to have the capability of crafting and sending IP packets with arbitrary legitimate headers without being filtered at the first hand, e.g., by her upstream or transit ISPs.

Note that we do not require any privilege of the target switch for the effective DoS attack. However, having some partial, internal state of the target switch (e.g., installed ACL rules) can further improve the efficiency (i.e., less number of required attack packets).

Here, we do not consider volumetric DoS attacks that congest the target’s network bandwidth with attack traffic.

3.2 DoS with Excessive MFC Masks
As mentioned in §2.2, the lesser the number of masks in the MFC the much faster the packet classification is.

To provide a simple but useful intuition oh how easily this number can be increased, consider that in a hypothetical protocol (say, HYP) having only 3 relevant header bits and consider a 3-bit-wide "Whitelist+DefaultDeny" type flow table shown in Fig. 1. The MFC is an unordered set of key-mask pairs $C = \{ (K, M) \}$ with entries $C = (K, M)$, here, $M$ is a bitmap to mask relevant header bits and $K$ is a key to be matched on the masked bits. According to the TSS scheme, we maintain a list of distinct masks $M$ (the "tuple space") plus, for each mask $M \in M$, a hash $H_M$ that will be used to store and lookup the keys with mask $M$.

Suppose that the switch receives a packet with HYP header $h_1 = \emptyset 01$. Since, initially, the MFC is empty, the packet is deferred to the slow path, which finds the first flow in the flow table to match, associates the action allow, and installs a new key-mask pair $C_1 = (\emptyset 01, 111)$ into the MFC; this amounts to adding the new mask 111 to the mask list $M$ and storing the key 001 in the respective hash $H_{111}$.

Now assume that a second packet arrives with header $h_2 = 111$. In this case, MFC lookup occurs as follows: take each mask $M \in M$ one by one, apply $M$ to the header and look up the resulting bitvector in the corresponding hash $H_{M'}$ if the lookup succeeds, then return a cache hit; otherwise resort to the next mask. In this case, there is only a single mask $M = 111$, so we look up the key $(h_2 \text{ AND } M) = 111$ in $H_{111}$. Since the lookup fails, this is a cache miss. The slow path will find the drop rule to match, and it will insert a new MFC entry into the fast path.

At this point, there are multiple choices to generate a new entry, each striking a different balance between space- and time-complexity (see details in §4); which one is taken in any particular case is the result of a rather involved construction of heuristics in the OVS slow path [58]. In a nutshell, when generating a new MFC entry $C$ for a packet with header $h$, OVS maintains the following two invariants:

- $\text{Inv}(1) \text{ Cover}: h \text{ matches } C$.
- $\text{Inv}(2) \text{ Independence}: C \text{ is disjoint from any } C' \in C$.

$\text{Inv}(1)$ simply states that an MFC entry will match the packet header that sparked its generation. $\text{Inv}(2)$ greatly simplifies the fast path code because lookup can early-exit once the first match is found, instead of having to tediously search through the entire mask list to check whether higher-priority matches occur later in the list (cf. Alg. 1 in §11.2). Hence, $h_2 = 111$ can spawns a new key-mask pair of either $C_2 = (111, 111)$ or $C_2 = (111, 100)$, covering 1 or 4 packets, respectively, according to different strategies (cf §4.1).

One might realize that due to these invariants, the number of key-mask pairs covering all possible packets significantly increases with the number and bit-width of the headers the ACL matches on. Particularly, if we establish a logical OR relation between the allow rules on more header fields (see a typical example in Fig. 6), it will in turn create an AND connection on the drop rule. Therefore, in order to test each header field at the same time, we need to test each combination of key-mask pairs for the individual headers resulting in a multiplicative increase in the tuple space (cf. §4.2).

This means that a typical ACL matching on the IP source address and TCP ports (e.g., ACL in Fig 6) can easily result in thousands of MFC masks. Consequently, this type of security policies/ACLs with an OR relation between the targeted header fields can become the sweet-spot for our attack; hence the name Tuple Space Explosion.

Next, we briefly present two different approaches of the TSE attack based on the partial control an adversary can have over the...
ACL. Then, in §4, we give a comprehensive overview of the look up algorithm, analyze its space- and time-complexity as the number of headers grows, and go through each case step-by-step to show how the MFC is exactly being managed.

3.3 Two Approaches of the TSE Attack

Before, we have seen that in order to practically populate the MFC with new entries and masks, we need to send a specially crafted packet sequence corresponding to the installed ACL. For instance, sending packets with (HYP) header (001) and (100) towards to the ACL shown in Fig. 1 will spawn entries #1 and #2 in the MFC as depicted in Fig. 3. However, a subsequent packet with header (101) will be also “caught by” entry #2, hence not increasing the number of masks in the MFC. Correspondingly, being aware of the ACL itself is a key aspect to the efficiency of the TSE attack. Thus, we present two different approaches of the TSE attack each posing different requirements and targets for the attacker.

In order to explain the main differences between them, and their practical targets, we need to understand a key abstraction in a cloud environment: the per-user virtual switches tenants configure to set up their ACLs. Tenants perceive these virtualised resources as their own physical switch, however switches are only logically separated and all of them are implemented and managed by the same individual software switch instance. Therefore, all workloads happened to be scheduled to the same hypervisor inherently share the switching fabric as well (e.g., the MFC).

**Co-located TSE.** We build on top of this abstraction: the attacker has leased resources in the cloud, which inherently makes him/her capable of installing ACLs into its own virtual switch. Then, the shared MFC can be easily populated with new masks by targeting the known ACLs (see details in §5). However, co-location comes at a price that only those tenants’ workloads are affected that happened to be scheduled to the same hypervisor.

**General TSE.** In this approach, we alleviate the restrictions of Co-located TSE: we consider the case when the attacker has neither resources in the cloud, nor knowledge about any ACLs. Here, we investigate how much more effort an attacker needs in order to achieve the same efficiency as Co-located TSE (see later in §6).

4 SPACE – TIME COMPLEXITY OF TSS

Next, we analyze the space- and time-complexity trade-offs of the TSS scheme using its implementation in OVS as a typical example. Then, we show how the TSS scheme manages its data structure, and how to maximize the number of masks in the MFC step-by-step.

Clearly, the most time-consuming step in Alg. 1 is the iteration through the mask list M, assuming that a hash lookup in $H_M$ is $O(1)$; the more the masks the slower the algorithm. The space-complexity is in turn driven by the sheer storage size of the MFC entries. Our observations are as follows.

**Observation 1.** The time-complexity of TSS lookup grows linearly with the number of distinct masks as $O(|M|)$ and the space-complexity grows linearly with the number of entries as $O(|C|)$.

Next, we demonstrate that the complexity of cache lookup can become prohibitive due to an algorithmic complexity vulnerability in the underlying TSS scheme. In particular, we show specific corner cases for which the MFC will exhibit exponential space- and/or time-complexity. For simplicity, we carry on with the hypothetical 3-bit protocol example (cf. Table 1), but bear in mind that MFC works the same for arbitrary wide bit-widths of header fields.

4.1 Maximize MFC Masks: Single Header

Packet classification is conceptually easier when there is only a single packet header field, e.g., IP protocol or destination address in the flow rules [31, 64]. First, we concentrate on this case, i.e., when the network policies match on a single header field only. Consider again the simplified policy on 3 bits mentioned in §3.2, i.e., when packets with header HYP 001 is allowed and everything else is denied. Next, we discuss some possible strategies to construct the MFC for this ACL using TSS.

**An invalid strategy** would be to install the flow table as is into the MFC resulting in two masks and two MFC entries. However, this would violate the independence invariant as the two entries overlap: a packet with HYP 001 would match both MFC entries, which would confuse the lookup algorithm. Consequently, in order to load a flow table into the MFC, it first needs to be converted into an order-independent form.

**Exact-match strategy.** One trivial order-independent transformation would be to cover the entire range of HYP with a single completely filled exact-matching hash, resulting in the TSS setup depicted in Fig. 2. Since we have a single mask, TSS lookup is extremely fast (cf. Observation 1). However, we need to add all possible 8 keys that can occur on 3 bits to the hash, yielding a larger memory footprint. Note that in general, the exact-match technique yields optimal time-complexity with exponential space-complexity.

**Wildcarding strategy.** The opposite extreme would be to wildcard as many bits as possible in order to get the broadest possible rules, and the fewest hashes, in TSS. Here, we obtain an exact-match entry for the allow-rule and separate key-mask pairs for testing each of the related 3 header bits to cover the whole tuple space. First, check whether the most significant bit is set and, if it is, then drop the packet; then, test for the second bit provided that the first bit is not set, and so on. One can easily check in Fig. 3 that the resulting MFC is order-independent and it is the smallest possible representation of this kind. We obtain 4 entries and 3 masks (the first and the last entries have the same mask), reducing the space complexity from 8 megaflows to just 4 at the cost of increasing classification time from a single iteration of the TSS lookup algorithm (Algorithm 1) to possibly 3 iterations for the 3 masks. Apparently, each strategy gives a different compromise between space- and time-complexity (in line with [28, 33]). The below theorem characterizes the attainable tradeoffs in general TSS (see proof in the Appendix).
Theorem 4.1. Given an ACL on a w-bit header field comprising a single exact-match allow rule and a lower-priority DefaultDeny policy, no TSS construction can achieve better than $O(k)$ time with $O(k^2 \log k)$ space complexity, for $1 \leq k \leq w$.

Here, the parameter $k$ balances between time- and space-complexity: for $k = 1$, we get optimal time-complexity ($O(1)$ time with $O(2^w)$ space), for $k = w$ we get optimal space-complexity ($O(w)$ time with $O(w)$ space), and different settings for $1 \leq k \leq w$ give different tradeoffs. While the compromise a particular TSS implementation realizes in a setup depends on a lot of unknowns, in practice OVS usually leans toward the “wildcarding” strategy ($k$ close to $w$), striving to minimize the memory footprint of the fast path classifier even at the cost of crippling lookup efficiency (see the comments in classifier.h in the OVS source code).

In essence, this means that for a single header of bit-width $w$, the MFC will have $w$ masks (in the worst case). However, in certain cases, OVS seems to optimize for the other extreme and minimizes lookup time at the cost of exponential space ($k$ close to 1); we have seen such behavior for ACLs including IPv6 address fields (cf. §5.4).

4.2 Maximize MFC Masks: Multi Headers

Next, we generalize the single-field technique to multiple fields to get the desired exponential complexity in any TSS implementation.

As mentioned before, an ACL that filters on the 32-bit IPv4 source address field ($w_1 = 32$), we can generate 32 masks and 33 entries. For the port field ($w_2 = 16$), the corresponding figures are 16 masks and 17 entries, respectively. Establishing a logical OR relation, on the other hand, between different header fields in the ACL at the same time will in turn create an AND connection on the drop rule.

Considering our example ACL in Fig. 4, this means that a packet can be dropped only if both HYP is not $\emptyset$ and HYP2 is not 1111. We can see that the most space-efficient way to test the deny case individually for the HYP field is to test each bit one by one and similarly for HYP2 field. However, collectively testing the two fields involves testing each combination of bit positions in the two fields (cf. Fig. 5), yielding $3 \times 4 + 1 = 13$ masks with roughly the same number of entries\(^\text{5}\). Observe that the first allow rule of the ACL is represented in the MFC in the same way, which does not apply for the second allow rule (hence +1 in the above equation).

Theorem 4.2. Given an ACL on $n$ header fields of bit-width $w_1$, $w_2$, . . . , $w_n$, comprising $n$ allow-rules, each exact-matching on a single header field, plus a lower-priority DefaultDeny policy, no TSS construction can achieve better than

$$O\left(\prod_i k_i\right)$$ time-complexity and

$$O\left(\prod_i k_i \frac{2^{w_i}}{2} - 1\right)$$ space-complexity

for any $1 \leq k_i \leq w_i$, $i \in \{1, \ldots, n\}$.

Here, again the formula allows to tune the space–time tradeoff, but this time separately for each field through setting $1 \leq k_i \leq w_i$. For the extreme choice $k_i = 1$ for all $i$, we again get optimal time

$$O(1) \text{ (time) with } O(2^{\sum_i w_i}) \text{ (space)}$$

\(^5\text{Note that if the second rule (in Fig. 4) also filtered on HYP, we would still have roughly the same masks; only allowance would change.}\)

CoNEXT ’19, December 9–12, 2019, Orlando, FL, USA

Figure 4: ACL on 2 headers

<table>
<thead>
<tr>
<th>#</th>
<th>HYP</th>
<th>HYP2</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>001</td>
<td>*</td>
<td>allow</td>
</tr>
<tr>
<td>2</td>
<td>1**</td>
<td>1111</td>
<td>allow</td>
</tr>
<tr>
<td>3</td>
<td>0**</td>
<td>1111</td>
<td>allow</td>
</tr>
<tr>
<td>4</td>
<td>000</td>
<td>1111</td>
<td>allow</td>
</tr>
<tr>
<td>5</td>
<td>1**</td>
<td>0***</td>
<td>deny</td>
</tr>
<tr>
<td>6</td>
<td>0**</td>
<td>0***</td>
<td>deny</td>
</tr>
<tr>
<td>7</td>
<td>0***</td>
<td>0***</td>
<td>deny</td>
</tr>
<tr>
<td>8</td>
<td>0***</td>
<td>0***</td>
<td>deny</td>
</tr>
<tr>
<td>9</td>
<td>0***</td>
<td>0***</td>
<td>deny</td>
</tr>
<tr>
<td>10</td>
<td>0***</td>
<td>0***</td>
<td>deny</td>
</tr>
</tbody>
</table>

Figure 5: Corresponding MFC (keys are masked)

$(O(1) \text{ (time) with } O(2^{\sum_i w_i})) \text{ (space)}$ and $k_i = w_i$ yields optimal space $(O(\prod_i w_i)$ for both time and space), and different settings for $k_i$ again provide different tradeoffs. In practice, we observed again OVS to lean towards space minimization yielding the required multiplicatively scaling $O(\prod_i w_i)$ space- and time complexity.

5 CO-LOCATED TSE

In this section, we discuss the Co-located TSE attack in detail. First we show the basic idea behind generating an “adversarial” packet sequence corresponding to the installed ACL. Then, we present a typical ACL tenants usually deploy for their services, and we apply our technique to show the efficiency of Co-located TSE. In particular, we evaluate Co-located TSE in various synthetic and live testbeds, and we show to what extent the tuple space explosion phenomenon can degrade the overall performance with considering several hardware offloading techniques.

5.1 Adversarial Packet Trace

As discussed in §3, to practically spawn the MFC entries, we need a specially crafted packet sequence. Note that such packets are completely legitimate and benign just as the useful user traffic. Taking the example ACLs in §4.1 and §4.2, we construct only a single header, then we turn to the case of multiple headers.

Single Header. We found the following packet trace generation strategy against the single header scenario to work well in practice: generate a packet that matches the allow rule, then add a packet with each of the relevant bits inverted one-by-one. In Fig. 1, this results in the following HYP header fields: \{001, 101, 011, 000\}. It can be seen that packets with these headers will exactly result in the 4 MFC entries and 3 masks shown in Fig. 3.

Multiple Headers. We generalize from the single-field case to the multi-field case with minimal modification. Consider the extended sample ACL in Fig. 4. First, create the HYP list $L_{HYP}$ according to the bit-inversion method, then create a similar list $L_{HYP2}$ for the HYP2 field, and then generate a packet with setting HYP and HYP2 from the outer product of the lists $L_{HYP} \times L_{HYP2}$. Ignoring the packets for the second allow rule that do not generate new masks (see entries #2 – #4 and #14 – #16 in Fig. 5), this technique gives exactly $4 \times 3 + 1 = 13$ packets and the same number of MFC masks. We omit the formal proofs here for brevity.

5.2 Practical ACLs

The effectiveness of the attack depends on the number of MFC masks the attacker can spawn in the data plane, which in turn
depends on the number, bit-width and the values of the header fields the installed network policies match on. Correspondingly, as a full-blown attack we can target 3 IP header fields out of the possible IP 5-tuple: the source IP address, the source transport port, and the destination port3 (see Fig. 6). Accordingly, we distinguish between several use cases, each with its own set of targeted header fields and its own expected effectiveness (the numbers are for IPv4).

First, in case of BASELINE, there is only one allow rule (apart from the simple DefaultDeny rule) in the flow table, which allows the tenant’s service to be reached (e.g., rule #1 in the ACL in Fig. 6). Accordingly, there is one benign traffic flow matching on that single rule only, and no malicious traffic enters the system; #MFC masks: 1. This scenario represents the full capacity of the switch in normal operation and serves as a baseline to measure service degradation for the rest of the use cases.

In case of Dp, we attack on a single header field, the 16-bit destination port. Accordingly, the ACL contains only Rule #1 and Rule #4 from the full-blown setup of Fig. 6, but now adversarial traffic against rule #1 is sent; #MFC masks: 16. Then, we gradually increase the number of headers and combine them, starting from the basic multi-field attacks of SpDp and StrDp, targeting both port (#MFC masks: ~256), and the IP source address with the destination port fields (#MFC masks: ~32 × 16 = 512), respectively.

Finally, as a full-blown attack, in SpStrDp we target all rules in Fig. 6 (#MFC masks: ~8200). Note that in all cases the traffic traces also include some additional random noise added to “unimportant” header fields (e.g., varying TTL) to increase the entropy hence using up the microflow cache.

### 5.3 Environments and Setup

Next, we present the setup used to evaluate Co-located TSE attack in various live environments. First, we present synthetic measurements on a standalone switch to show that the TSS implementation in OVS is vulnerable to TSE attacks. We also demonstrate that even if the TSS implementation is offloaded to the hardware, such a system is still vulnerable. Then, we study the performance of OVS when used as a hypervisor switch in a real OpenStack and a Kubernetes environment (small-in-house testbeds for ethical reasons). Table 1 lists the software and hardware configurations used for the tests.

In these environments, we measure the raw throughput of a standalone OVS by simulating the pipeline that would arise in a real cloud deployment. We created a simplified cloud infrastructure consisting of a small data center (DC) having 2 servers hosting the tenants’ workloads (see Fig. 7). In particular, the victim has a publicly available web service (V1 in Server 1), which s/he has installed an ACL (ACL-V) for. Furthermore, the victim also has another service (V2) used as a backend service of V1 scheduled to Server 2. On the other hand, the attacker also has a leased resource (A1) co-located with V1 in Server 1. Similarly, the attacker also defines an ACL (ACL-V) for his/her own service (again, A1). However, in this case the attacker installs the ACL used for the full-blown TSE attack (cf. Fig. 6), and will send a corresponding packet sequence to it in order to populate the MFC with an excess amount of masks; hence potentially degrading the quality of services of other tenants (i.e., V1 in this case). Furthermore, since the attainable number of masks is known, measurements do not require multiple runs.

### 5.4 Synthetic tests

Here, similarly to [60], our simplified DC consists of a system-under-test (SUT) which runs OVS acting as a hypervisor for two KVM virtual machines (Server 1), connected back-to-back to a similar “test” machine (Server 2). Instead of having a third machine for the incoming user traffic, for brevity, we only run an iperf3 session between V1 and V2 (cf. Fig. 7). This represents the “useful” benign traffic (e.g., frontend – backend communication), whose performance degradation will demonstrate the collateral damage. Furthermore, the attacker, from which we generate the malicious traffic (via replaying a pcap file like in [20]), is also cast to Server 2 in a second VM. The attack traffic, furthermore, contained the destination IP address of A1. (This setup gives a conservative estimate of the damage done; a single large “victim” flow can be handled in TSS using only a single cache entry ‘ideally’, while real workloads usually include thousands of flows requiring hundreds of TSS entries that will compete with the attacker’s adversarial cache entries.)

The OVS flow table was bootstrapped manually according to the ACL in Fig. 6.

Next, we show to what extent the throughput of OVS is affected as the number of MFC masks increases; we also evaluate the effect of several NIC driver offloading techniques.

The results when the victim generates TCP and UDP traffic are depicted in Fig. 9a, where the x axis shows the number of 3Even though benign traffic may consist of other types of flows (e.g., short-lived), they are out of scope of this study.

4Note, however, that in real deployments the attack might cannot be launched completely within a DC due to the IP spoofing protection mechanism of the CMS (e.g., OpenStack) that prevents attacks based on this header field.

![Figure 6: Simple ACL of a full-blown TSE attack.](image)

**Table 1: HW/SW and orchestrator versions.**

<table>
<thead>
<tr>
<th>Property</th>
<th>Synthetic</th>
<th>OpenStack</th>
<th>Kubernetes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5-2620 v3</td>
<td>64GB</td>
<td>2 x Intel i5-6300U</td>
</tr>
<tr>
<td>Memory</td>
<td>Intel X710</td>
<td>Intel 82598</td>
<td>2 x 3GB</td>
</tr>
<tr>
<td>NIC</td>
<td>Mellanox CX-4</td>
<td>Intel 82598</td>
<td>virtio</td>
</tr>
<tr>
<td>Kernel</td>
<td>4.13.15-1</td>
<td>4.4.0-112</td>
<td>4.4</td>
</tr>
<tr>
<td>OVS</td>
<td>2.9.2 (stable)</td>
<td>2.9.90 (unstable)</td>
<td>2.7</td>
</tr>
<tr>
<td>Orchestorator</td>
<td>OpenStack Queens</td>
<td>Kubernetes 1.7</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 7: Simplified cloud infrastructure model and overview of Co-located TSE (thin dot-dashed purple line) and General TSE (thick red dashed line) attacks.](image)
MFC masks, while the y axis plots the corresponding throughput; note that both axes are in log scale. In order to easily realize the maximum number of MFC entries attainable in each use case, we note the x tick labels for numbers 17, 260, 516 and 8200 for Dp, SpDp, SipDp, and SipSpDp, respectively. We observed dramatically different effects depending on various settings of the NIC driver. In particular, jumbo frames and generic transmit/receive offload support (GRO ON) let the NIC to assemble many small TCP packets into a single large TCP buffer [60], reducing the effective traffic rate seen by OVS to a couple of thousand pps which it can process efficiently even when the TSS classifier has excess masks. Furthermore, we also enabled full hardware offloading (FHO ON) via our Mellanox CX-4 NIC, which gave a huge boost to the overall performance (∼ 30 Gbps). However, the TSS classifier still remains vulnerable resulting in a significant performance drop as the number of MFC masks increases above 200. For UDP, these settings take no effect and performance degradation is clearly visible in all scenarios.

Our observations are as follows: i) 17 masks (max. in Dp use case) are sufficient to reduce the effective throughput to roughly 97%, 88%, and 53% of the Baseline with GRO ON, FHO ON, and GRO OFF, respectively. As the ii) number of masks reaches 260 (max. in SpDp), these numbers are as follows: 95%, 43%, and 10%, respectively. In case of having iii) more than 500 masks in the MFC (e.g., in SpDp), the increased packet classification time reduces the full capacity to its 76%, 29%, and 4.7%, respectively. Finally, iv) spawning more than 8000 masks (attainable in case of SpSpDp) results in a virtual complete denial of service attack in each case as the throughput drops down to 3.9%, 2.1%, and 0.2%, respectively. Hereafter, we present only the results for TCP with GRO OFF.

To present the increased packet processing time, the secondary y axis shows the flow completion time of 1GB TCP traffic with GRO OFF as the number of MFC masks grows; again, note the log scale. It can be seen that, on average, the flow completion time only increases half as high as the number of MFC masks.

Nevertheless, this experiment clearly marks the vulnerability of TSS to low-bandwidth DoS attacks.

Next, Fig. 8a gives the results for 3 parallel victim TCP flow in the SpDp scenario with TCP offloading disabled. In this benchmark, the attacker is active from t1 until t2 injecting 100 packets per second (50 Kbps), reducing victims’ aggregate traffic rate from 9.7 Gbps to below 0.5 Gbps. Observe the delay in the recovery of the victim rate that returns to full rate only after 10 seconds after t2; this is due to the 10 sec idle MFC timeout in OVS, keeping the attacker’s entries alive for an extended time.

The extent and type of the damage varies on a case by case basis, depending on the type of ACLs injected, the OVS version, the NIC configuration, etc. For instance, when we apply the SpDp attack vector over IPv6 we find that OVS applies the “wildcarding” TSS entry generation technique only to the TCP destination port field but seems to handle the IPv6 source address using exact matching, which can result in only a handful of masks but hundreds of thousands of MFC entries (irrespective of the TSE method). Hence, in this scenario the adversarial effect manifests itself not in the slowdown of the victim traffic but rather in excess memory and CPU consumption, with OVS taking up 8 CPU cores trying to uselessly reclaim megaflow memory occupied by the excess TSS entries. Restricting OVS to just 2 CPU cores then reduces victim traffic to 5% of its nominal rate.

Next, we evaluate the Co-located TSE attack in two smaller real testbeds (in-house for ethical concerns): OpenStack and Kubernetes.

5.5 OpenStack

Our OpenStack testbed (cf. Table 1 for details) uses the OVN integration [73]; this configuration is known to exhibit superior network performance compared to the default [4]. Workload isolation between the attacker and the victim was enforced by deploying the corresponding VMs using different OpenStack tenants.

The CMS API only allows the SpDp scenario, for which the results are given in Fig. 8b; here, the attacker starts sending at the beginning of the benchmark at 100 pps and stops in the 60-th second only to restart 30 seconds later; the victim joins with a full-rate UDP iperf session at the 30-th second. In line with the synthetic setup, in the OpenStack testbed we again see a substantial (more than 90%) useful performance reduction during the time interval when both the attacker and the victim are active. Again, the victim recovers 10 seconds after the attacker stops sending. Curiously, the re-activation of the attacker causes only a minor damage to the victim rate (about 10% drop); it seems that the attack is effective only against newly established target flows but causes minor harm to long-lasting flows already active at the moment when the attack starts. We observed this behavior consistently in this version of OpenStack; we have contacted the OVS authors regarding this behavior of this specific
unstable version of OVS, but the reasons were mostly unknown and migrating to a stable version was suggested.

5.6 Kubernetes
Our setup uses the OVN integration [69] and the topology is the same as in the synthetic tests; one server hosts the attacker and the victim source and another hosts the sinks, both provisioned in separate vagrant boxes connected by a virtual network link supporting 1 Gbps rate. Here, we could use the \textit{SipSpDp} attack scenario yielding the full ACL in Fig. 6; since Kubernetes/OVN currently does not support the full semantics of Calico network policies we injected the source port filtering rules manually via the CLI.

The results are shown in Fig. 8c. Initially no "malicious" ACL is set up, i.e., when the victim starts an \textit{iperf} session it quickly reaches 1Gbps rate. The attacker starts sending at \( t_1 \) at 1,000 pps (causing a minor glitch in the victim rate) and then injects the ACL in Fig. 6 at \( t_2 \), triggering thousands of MFC entries in the OVS data plane. In response, the victim rate rapidly drops by 80%. Then, at \( t_4 \), the attacker increases its sending rate to 2,000 pps, resulting in a full denial of network service to the victim. From that point, the victim rate drops close to 0 for 30 seconds, during which OVS can hardly push any useful packets through the data plane because of the malicious activity of the attacker. During our evaluations, we have seen similar cases of full-blown DoS under various scenarios, with cases when \textit{iperf} could not even establish a new TCP session for extended periods of time.

6 GENERAL TSE
Next, we scrutinize the efficiency of the TSE attack when having co-located resources and knowledge about the ACLs are not required. Such alleviated requirements make General TSE attack more appealing as any arbitrary service, i.e., ACL, can be attacked. However, this comfort comes at a price that the attack itself requires more effort (in terms of packet rate) to even approximate the efficiency of Co-located TSE yet keeping the attack rate low (≪ volumetric).

Correspondingly, in this section we first discuss how to generate a packet trace against an unknown ACL. Then, we show lower bounds on the estimated number of MFC masks we can achieve in all use cases against the ACL in Fig. 6), which we underpin later with practical measurements.

6.1 Adversarial Packet Trace
To target an unknown ACL, a naïve approach would generate sequentially all possible packets for the given header fields in order to spawn as many MFC masks as possible. Clearly, such approach would easily result in a volumetric attack since considering even the case of \textit{SipDp}, the required successful attack rate would be \( \sim 2.9 \text{ p(eta)bps}. \) Therefore, we need a better heuristic algorithm that tries to approximate the attainable number of MFC masks. Randomization has proven to be efficient many times in practice (e.g., evolutionary and genetic algorithms, runtime analysis, convergence [9]), thus we adapt this approach to our packet trace generation. First, we analyse what are the chances that a packet with random (but legitimate) header will spawn an MFC entry. Then, we show what is the expected number of MFC masks for a given number of random packets sent to an unknown ACL.

Single Header. One can see in Fig. 3 that for a header length \( h \) the probability that one packet will spawn a specific entry in the MFC is \( p_k(MFC) = \frac{2^k}{2^h} \), where \( k \) is the number of wildcarded bits the given MFC entry has; e.g., \#2 entry in Fig. 3 \( p_2(MFC) = \frac{2^2}{2^h} = 0.5 \). Generally, the probability that from \( n \) randomly generated packets there will be at least \( 1 \) packet that sparks an MFC entry for a given \( k \) is:

\[
P(k,n)(MFC) = 1 - (1 - p_k(MFC))^n, \tag{1}
\]

Accordingly, the expected value of the number of MFC masks can be formalized as follows:

\[
\mathbb{E}_n(k,n)(MFC) = \sum_{k=0}^h C_k * p(k,n)(MFC), \tag{2}
\]

where \( C_k \) notes the number of different MFC entries for a given \( k \).

Multiple Headers. Eq. 1 and Eq. 2 can be generalized to multiple headers; one only needs to pay attention to the number of possible different MFC mask combinations (\( C_k \)) for a given \( k \), which heavily depends on the width of the header the first flow rule matches on (see Appendix for more details).

Note that for each use case all related header fields were randomized (e.g., for \textit{SpDp} the packet trace contained packets with random source IP and destination port). Similarly to \textit{Co-located TSE}, the traces included additional random noise to exhaust the microflow cache. Recall, it is usually used up in normal operation.

6.2 Synthetic Tests
Next, we show to what extent a practical random packet trace generation is in par with the expected values above.

Note that in our evaluation we used the same ACLs as well as the same testbed presented in §5, however, the attacker targets the ACL installed by the victim (ACL-V). For brevity, we only present the results for the synthetic tests, but note that as long as the General TSE attack achieves the same amount of MFC masks as \textit{Co-located TSE}, the effects are the same irrespectively to the environment.

Results are depicted in Fig. 9b, where on the x axis the number of different random packets is shown, while the y axis depicts the expected (E) and measured (M) (averaged over 100 runs) MFC masks for each use case. Since the expected values are dominated by the width of the header the first flow rule of the ACL matches on (cf. §6.1), the difference between \textit{SipDp} and \textit{SpDp} was negligible; hence the latter is removed for brevity.

Observe that the more different header fields the ACL consists of the more MFC entries can be spawned with the same number of random packets. In particular, the maximum attainable MFC masks (with 50,000 packets) are approx. 16, 121, 122, and 581 in case of \textit{Dp}, \textit{SpDp}, \textit{SipDp}, and \textit{SipSpDp}, respectively. In terms of service degradation, these results mean that General TSE can reduce the full capacity with GRO OFF to 52% (97% with GRO ON, 88% with FHO, 60% with UDP), 12% (96% with GRO ON, 87% with FHO, 15.8% with UDP), and 1% (73.5% with GRO ON, 25.5% with FHO, 3.25% with UDP), respectively.

Recall that in \textit{Co-located TSE} the maximum attainable MFC masks (17 for \textit{Dp}, 256 for \textit{SpDp}, 512 for \textit{SipDp}, and 8195 for \textit{SipSpDp}, respectively) require the same amount of packets. This means that for the \textit{Dp} use case, General TSE can be as good as \textit{Co-located TSE} in
7 DISCUSSION

The technique extends to an arbitrary number of protocol fields. Each CMS imposes its own set of limitations on the possible ACLs that can be installed and the extent to which different packet header fields can be considered: by default, OpenStack and Kubernetes allow ingress policies to filter only on the source IP addresses and the destination port (TCP or UDP) [15, 71]. This gives a comfortable 32+16 = 512 excess masks in the MFC. Calico (a Kubernetes network plugin [66]) allows ingress security policies to also filter on the source port, yielding possibly 8192 masks (already enough for a full-blown DoS) to which egress policies introduce the destination IP address as well (~ 200 thousand masks).

All cloud deployments implementing ACLs in OVS are affected. OVS is extensively used in cloud-based systems (e.g., it is the most widely used hypervisor switch in OpenStack) and it increasingly takes over the responsibility of enforcing ACLs from iptables due to the raw performance edge, standard support, and ease in management [57]. The TSE attack is effective over the OVN backend for OpenStack Neutron [73], OpenStack/OpenDaylight [70], OpenStack/ONOS [68], and in Kubernetes/OVN as demonstrated in §5.5 and §5.6. Note, however, that default installations are not directly affected as ACLs are implemented in iptables, but this architecture tends to become legacy soon [4]. Furthermore, major cloud providers do not seem to be affected: for instance, Microsoft Azure does not use OVS in the AccelNet network virtualization framework [29] and, even though the Google Cloud Platform does include OVS in the Andromeda data plane, this seems to be a significantly stripped down version [22]. In any case, we did not perform specific tests in public cloud providers’ DCs for obvious ethical reasons. TSE generalizes beyond OVS. TSE exploits an algorithmic complexity deficiency in the venerable TSS scheme. Therefore, deployments relying on the TSS scheme for packet classification, e.g., OpenStack/Networking-vpp [72], Contiv/VPP Kubernetes [26], Xen/HyperSwitch [60], Netronome SmartNIC [50], might be also affected; the evaluation has bee left for a future study.

Furthermore, there is considerable base of network-function virtualization [34, 49], cloud gateway/load-balancer [49], campus and enterprise networks [48] that use TSS scheme for packet classifier to implement non-trivial packet processing pipelines. If any of the flow tables in these deployments contain the above adversarial pattern, then the DoS attacks presented here are effective.

8 MITIGATION

The above results suggest that TSE can be particularly damaging. Accordingly, we initiated a responsible disclosure process by providing code and methodology to reproduce the synthetic tests to the corresponding security teams [11, 18, 19]. Besides, several immediate yet impractical remedies might help: (i) deploying or offloading ACL implementations to a different hypervisor switch (e.g., [35, 49, 79]) or to the (ii) high-performing gateway appliance (e.g., [6]), (iii) switching the MFC completely off, or (iv) enabling advanced flow caching via DPDK-based OVS datapath [24]. However, each of the above has the following corresponding disadvantage: in case of (i) other implementations might suffer from the same attack (e.g., [26, 27, 75]) or (ii) they do not help against attacks initiated within the DC, for (iii) MFC has been the biggest performance improvement so far [56], and for (iv) the feature that may prevent the attack is available in select datapaths.

Forcing the use of jumbo frames and TCP buffers (cf. §5.4) can substantially decrease the effective packet rate, however packets coming from outside might be limited to the default MTU size. And they do not cover attack against other traffic; e.g., UDP, which is the underlying transport protocol in QUIC [67].

MFCGuard. As a more customized mitigation technique we developed MFCGuard, which monitors and modifies the MFC— if the
number of masks exceed a certain threshold, it looks for patterns corresponding to a possible TSE attack (cf. §3) in every 10 seconds, and wipes out those entries accordingly. We observed that monitoring (and modifying) the MFC has no overhead on the performance. In essence, removing an entry from the MFC means that matching packets will be processed in the slow path again. Since the slow path would spark the same MFC entries again, the idea behind MFCGuard was to constantly keep those entries out of the fast path. In practice, however, we observed that once an MFC entry is deleted then it will never be sparked again, i.e., matching packets will always be processed by the slow path. Such undesired, unexpected and undocumented behavior [12] can have serious performance penalties; although MFC entries can be manually re-injected.

There are several requirements MFCGuard needs to meet: (i) entries covering the useful traffic should never be deleted. Furthermore, (ii) according to the available resources, we can only remove select flows from the MFC to find a balance between the maximum performance of the fast path and the increased resource utilization by the slow path; both impacting the overall quality of the run services. Due to requirement (i) MFCGuard will only remove entries with drop action! With this simple yet important requirement, only adversarial packets will be subjected to the slow path, keeping the fast path accelerated for the useful traffic flows (cf. Alg. 2 in §11.4). In our current implementation, we have therefore focused on (i), leaving (ii) for future work [17].

We evaluated the efficiency of MFCGuard in all use cases (by deleting all drop rules) and observed that once the MFC is "cleaned", the performance of the victim's traffic goes back to its baseline. As the slow path is becoming much more involved, we evaluated the system's load in such cases. Results are depicted in Fig. 9c, where x and y axes show the attack rate and the corresponding CPU usage of the slow path daemon (ovs-vswitchd), respectively.

It can be seen that as long as the attack rate is less than 1,000 pps (< 1 Mbps) the slow path only consumes 15% of the CPU; recall, this packet rate is enough to bring down OVS in case of Co-located TSE). However, when the packet rate is 10,000 pps, the CPU load jumps up to ≈ 80% (this rate in case of General TSE would be enough to degrade the full capacity to 10%). We can conclude that our current MFCGuard implementation is already capable of efficiently mitigating both TSE attacks as long as the attacking rate is low. If the attack rate is much above 10,000 pps, the attack becomes a volumetric attack, for which there are multiple solutions to detect and handle (e.g., excess amount of packets and over-provisioning).

9 RELATED WORK

Whether or not to virtualize services is a complicated question many enterprises are facing today [25]; in a survey, 73% of responders said that security is a top challenge holding back cloud adoption [63], with the possibility of unmediated sharing and communications between different tenants' workloads being among the major concerns [54]. Such unmediated tenant-to-tenant interaction might be initiated by a malicious user by first launching a co-residency attack in order to co-locate a virtual machine with the target tenants' virtual machines on the same physical server [76], and then exploiting a side-channel effect [41, 45] to eavesdrop on sensitive information [23, 61].

Direct attacks on the cloud network infrastructure are less known; there has been work on fuzzing the data plane with considerable success [52, 74] and compromising SDN controllers [7]. Denial of service using algorithmic complexity attacks [2, 14, 16, 55] on the network data plane usually works by exploiting a vulnerable algorithm/data structure that is already in the targeted binary; e.g., [78] shows cache-collision attacks against the Linux IP stack and [21] targets stateful firewalls. Here, we showed a vulnerable data structure in the TSS scheme heavily used for packet classification in hypervisor switches. We showed that a typical ACL can be the vulnerable target itself in the data plane. Our finding can be exploited either remotely from the public Internet, or leasing a single virtual machine deployed in the cloud; detection and prevention techniques for algorithmic complexity attacks (see e.g., [5, 39, 55]) do not seem effective against it. Although, a mitigation technique for an algorithmic complexity attack in DPI engines [3] uses similar approach as our MFCGuard: a devised algorithm is used, which has a constant (but less than normal) throughput regardless of the input. The authors’ aim is to dynamically shift between algorithms for normal and malicious input. In contrast, MFCGuard does not need to change algorithms at all, and always provides the highest attainable throughput regardless of the low-rate input.

10 CONCLUSION

Highly efficient and resilient packet classification is crucial to many security primitives particularly in a virtualized environments. In this paper, we investigate to what extent the TSS algorithm used in many software switches is vulnerable against low-rate DoS attacks. Our TSE attack exploits the fundamental space/time complexity of the TSS algorithm, and degrades the switch performance to 12% with low attack rate (0.7 Mbps). We show that if an adversary has knowledge of the used classification policies, she can virtually bring down the packet classifier with the same attack rate. One key aspect of our TSE attack is that it is hard to detect the attack as it does not use any specific traffic pattern but some random packets. Furthermore, since we exploit a vital complexity characteristic, there seems to be no complete mitigation technique, unfortunately. As a short-term solution, we propose MFCGuard, a monitoring system that via carefully managing the entries in the tuple space can keep packet classification fast.

ACKNOWLEDGEMENTS

This research is supported by the National Research Foundation, Prime Minister’s Office, Singapore under its Corporate Laboratory@University Scheme, National University of Singapore, and Singapore Telecommunications Ltd.

This research has also been supported by European Cooperation in Science and Technology (COST) Action CA 15127: RECODIS – Resilient communication and service, the UK Engineering and Physical Sciences Research Council (EPSRC) projects EP/N033957/1, and EP/P004024/1. G. Rétvári has been supported by project no. 123957, 129589 and 124171 with the support provided from the National Research, Development and Innovation Fund of Hungary under the FK-17, KH-18 and K-17 funding schemes.
The OVS cache infrastructure

11.1 OVS cache infrastructure

The OVS flow cache infrastructure and the whole pipeline of processing a packet through the caches all the way up to the slow path (in case of the first packet of the flow) is shown in Fig. 10.

![Figure 10: The OVS flow cache infrastructure.](image)

11.2 Megaflow Lookup Algorithm

With the invariants (Inv(1)) and (Inv(2)) described in §3.2 is mind we summarize a rather simplified description of the MFC lookup algorithm implemented in the OVS fast path below:

Algorithm 1 Megaflow lookup. Input: packet header $h$

for $M \in M$ do

lookup $(h \text{ AND } M)$ in the hash $H_M$

if found then return cache hit

end for

return cache miss

Proof of Theorem 4.1. Let parameter $k = |M|$ denote the number of masks and let $B_i$ be the set of bitpositions used by the $i$-th mask. It is easy to see that the number of keys in the MFC is minimal if $B_i \cap B_j = \emptyset$ for $1 \leq i < j \leq k$. In this case, to cover each denied packet the $i$-th mask needs $2^{b_i} - 1$ keys (each key except the one that refers to the allow rule), where $b_i$ is the number of bit positions set in $B_i$. Thus, the number of keys in the MFC is $\sum_i \left(2^{k_i} - 1\right)$. Using the inequality between geometric and arithmetic means, the expression $\sum_i 2^{b_i}$ subject to $\sum_i b_i = w$ takes the minimal value when $k2^{\frac{w}{k}}$. Hence, for $k$ masks the number of keys is at least $k2^{\frac{w}{k}} - k = O(k2^{\frac{w}{k}})$.

Proof of Theorem 4.2. We focus only on the keys that refer to deny packets. Since there is an allow rule for each field, all masks must refer to every field on at least one position. One can see that to minimize the number of keys these positions should be a Cartesian product of separate solutions for the different fields. Let $k_i$ be the number of masks considering only the $i$-th field, then using the result of Theorem 4.1 we get the required result.

□
11.3 Probabilities and expected values

Multiple header fields with ACL unknown. Naively, one might think that for each \( k \) there is one combination where one of the headers has \( k - 1 \) wildcarded bits, while the other header has \( l (0 \leq l \leq k) \). However, as can be seen in Fig. 5, only the first allow rule appears in the MFC cache in the same way, i.e., with the other header fully wildcarded. If the second flow rule was also represented in the same way, then the MFC would violate the order-independent property as a packet with HYP 001 and HYP2 1111 header would match on both of the entries. Therefore, for \( k = l \) or \( k = 0 \) there is just one combination.

Furthermore, if \( k \) is greater than the length of the shortest header \((s)\), then again the number of possible combinations is less \( (\text{i.e., it simply cannot hold more wildcarded bits as its length } s)\). These observations can be summarized as follows: for two different header fields of length \( s, l \) \((s \leq l) \) : \( C_k = k + 2 \) if \( 0 \leq k < s \), \( C_k = s \) if \( s \leq k < l \), and \( C_k = (s + l) - (k + 1) \) if \( l \leq k \).

Thus, the most important factor in calculating the expected values is \( C_k \). In the following, how its calculation can be generalized to above 2 headers.

Assume that we have \( m + 1 \) different flow rules, where \( m \) rules match on \( m \) different headers with sizes of \( h_1, h_2, \ldots, h_m \), where \( h_1 \) is the highest priority rule, while \( h_m \) is the lowest one. Additionally, the last rule \((m + 1)^{th}\) is the low priority deny rule. The entries covering the \( i-1^{th} \) rule contain prefix rules for the previous headers, exact match for the \( i^{th} \) header, and wildcard for the remaining headers. Let \( f_{i-1}(u) \) be the number of combinations for prefix fields with \( u \) wildcarded bit. Then, it can be calculated by the following convolution:

\[
f_i(k) = \sum_{j=1}^{\min(k, h_j)} f_{i-1}(k-j),
\]

where \( f_0(k) = 1_{k=0} \). Let \( f_m \) be the same for the deny rule, and it can calculated with the same convolution. Furthermore, let \( c^{(i-1)}_k \) be the number of combinations that contain \( k \) wildcarded bits in the whole header, then it can be calculated as \( c^{(i-1)}_k = f_{i-1}(k - \sum_{j=i+1}^{m} h_j) \). Therefore, \( C_k = \sum_{i=0}^{m} c^{(i)}_k \).

11.4 Mitigation Algorithm

Below, we present the mitigation algorithm. First, it has two preset thresholds \((\text{the number of MFC masks (} m_{th} \text{) as well as for the acceptable CPU overhead (} c_{th} \text{)) as input parameters that can be fine-tuned according to the available resources. As indicated in Line 1, the algorithm runs every 10 seconds according to the MFC eviction policy. In Line 2, we check the number of masks in the MFC \((\text{it can be acquired via commands ovs-dpctl1 dump-flows or ovs-dpctl1 show})\). If the number of MFC masks is above the preset threshold \((\text{Line 3})\), then for each rule in the FlowTable we look for a pattern the TSE attack would generate \((\text{according to } \S 4)\) in the MFC \((\text{Line 4})\). If a pattern is found \((\text{Line 6})\), we remove the corresponding entries from the MFC \((\text{Line 7})\). Each time after removing some selected entries from the MFC, we check the increased CPU utilization \((\text{e.g., via command } \text{top in Line 9})\) and if it is below the threshold \((\text{Line 10})\), we keep removing MFC entries \((\text{if there are any})\); otherwise the system is considered to be balanced, i.e., no more entries from the MFC will be removed as it would cause too much packet processing overhead in the slow path.

![Algorithm 2 Mitigation](CoNEXT ’19, December 9–12, 2019, Orlando, FL, USA)

```python
Algorithm 2 Mitigation. Input: #MFC mask threshold \( m_{th} \), CPU utilization threshold \( c_{th} \)
1: for every 10 second do
2: \( m \leftarrow \) checkNumberOfMasks()
3: if \( m > m_{th} \) then
4: for rule in FlowTable do
5: \( \text{found} \leftarrow \) lookPatternInMFC(rule)
6: if \( \text{found} \) then
7: deleteMFCEntries(rule)
8: end if
9: \( \text{cpu}_\text{util} \leftarrow \) checkCPUUtilization()
10: if \( \text{cpu}_\text{util} \geq c_{th} \) then
11: return
12: end if
13: end for
14: end if
15: end for
```