Application-Agnostic Offloading of Datagram Processing

Oliver Hohlfeld†, Helge Reelfs†, Jan Rüth†, Florian Schmidt‡,
Torsten Zimmermann†, Jens Hiller†, Klaus Wehrle†

†COMSYS, RWTH Aachen University ‡NEC Laboratories Europe
†{lastname}@comsys.rwth-aachen.de ‡florian.schmidt@neclab.eu

***Abstract*—As network speed increases, servers struggle to serve all requests directed at them. This challenge is rooted in a partitioned data path where the split between the kernel space networking stack and user space applications induces overheads. To address this challenge, we propose Santa, an architecture to optimize the data path by enabling server applications to (partially) offload packet processing to a generic rule processor. We exemplify Santa by showing how it can drastically accelerate UDP packet processing in the Linux kernel—a currently neglected domain. Our evaluation focuses on accelerating DNS traffic for which we find a performance increase by a factor of 5.5 on real-world request pattern.**

I. Introduction

Increasing line rates challenge the packet processing performance of current server systems. These performance challenges can be attributed to two main overhead factors in network stacks: *i)* memory allocations and copy operations, and *ii)* overheads by performing system calls and the required context switches [1]–[3]. These costs are particularly significant at high line rates (e.g., multiple 10 G interfaces), but are already apparent at lower rates if many (small) requests need to be processed (e.g., for DNS traffic). Thus, current OS *data path* designs significantly challenge packet processing performance in commodity hard- and software where CPU speeds do not scale with increasing line speeds.

The problem of speeding-up server systems is currently addressed by alternative data path designs that entirely bypass the kernel-level network stack as the bottleneck. One line of research proposes to offload packet processing to dedicated hardware for improved processing performance (see e.g., [4], [5]). A very active line of research proposes to shift packet processing to user-land stacks and thereby also removes the kernel from the data path (see e.g., [2], [6]). Performance improvements by this approach can be attributed to *i)* omitted copy operations and context switches between user space and kernel space and *ii)* benefits due to optimized and tailored microstacks. Realizations of HTTP and DNS servers as example applications utilizing user-land networking showed drastic performance increases (see e.g., [3], [6]).

In this paper, we describe a different strategy to accelerate server systems with a data path architecture that enables applications to (partially) offload packet processing into an application-agnostic rule-processor (which we refer to as Santa). This rule processor handles frequent requests on behalf of the application in kernel space and therefore short-cuts the data path. Its principal design enables it to reside in various parts of the network, e.g., in programmable NICs or middleboxes. However, it can also reside within traditional kernel stacks, which is the example application domain in this paper. From there, it can accelerate packet processing by avoiding costly copy operations and context switches that challenge server performance in the first place. This way—inspired by SDN—we propose to split the current data path into a control and data-plane. Based on the example set by applying our approach to accelerating kernel-level packet processing, we are able to show how the optimization of existing stacks—a domain that is currently neglected—can complement bypassing approaches.

The main contributions of this paper are as follows:

- We present Santa, a data path design utilizing an application-*agnostic* architecture to enable user-level applications to offload replies to common requests to a generic rule processor, e.g., to accelerate kernel-level stacks. Benefiting from Santa only requires minimal changes to applications that want to employ it.
- We exemplify the benefits of Santa to accelerate a DNS server for UDP-based packet processing. Our evaluation is based on using an ISP-level DNS traffic trace as real-world example. Santa increases the number of processed DNS requests per second by a factor of up to 5.5.

The good news is: there is still some life left in kernel-level packet processing. By applying Santa to the kernel, we unlock the speed of kernel space networking for legacy server software without requiring extensive changes or specialized implementations. We thus aim to pave the way for new packet processing pipelines and complement the ongoing discussion on user-level network stacks and kernel-bypassing techniques.

II. Santa Architecture

We accelerate packet processing by splitting it into a *data* and a *control* plane, similar to SDN. This split is based on short-cutting the traditional data-plane by inserting an application-agnostic rule processor—which we refer to as Santa [7]—that allows applications to (partially) offload their application processing logic. Santa can reside in various parts of the network, e.g., in the kernel or in middleboxes (see Fig. 1). Applying Santa to the kernel has the potential (by limiting context switches and copy operations) to drastically accelerate traditional packet processing—a domain that is currently neglected in the presence of proposals to bypass the kernel entirely (see Section VI). By applying Santa to the Linux kernel, we show that bypassing approaches are not

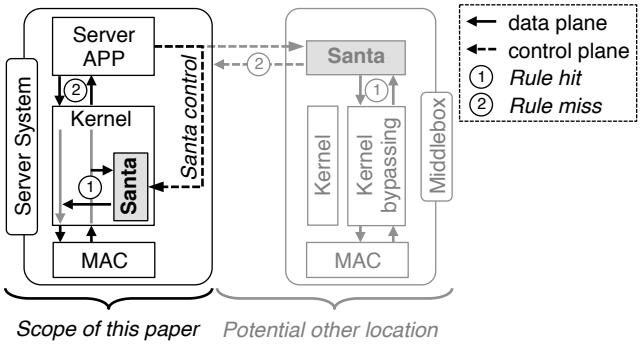


Fig. 1: Example execution positions of Santa: in-kernel (the focus of this paper) and in middleboxes (e.g., accelerated with kernel bypassing techniques such as DPDK or netmap).

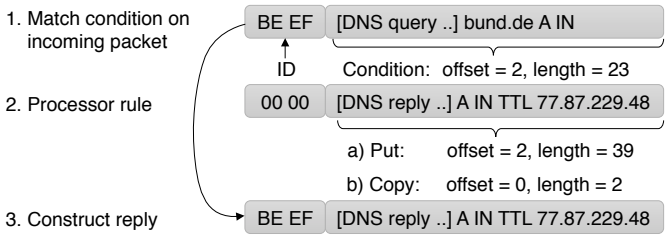


Fig. 2: Example DNS processing rule for an `A` record request for `bund.de`. If the rule matches, it constructs a response using put and copy rules specified in the processor template. The *put* operation will put the static part (A record) and the *copy* operation will paste the query-specific ID.

always necessary providing a trade-off decision to application developers and operators.

Unlike caching infrastructures, the control over the offloaded rules remains at the applications using Santa. That is, we provide a *control plane* offering applications the ability to add, modify, and delete rules in the Santa rule processor. Further, the control plane can be used for querying the rule processor, e.g., to retrieve access statistics that are relevant to content providers. This way, the application remains in full control of the offloaded packet processing.

Santa is most beneficial if a significant part of an application's workload consists of repeatedly serving the same requests with fixed responses (e.g., DNS, HTTP, Memcached, or database workloads). This highlights the main focus of Santa: it will only work well for (temporarily) static request–response pairs, but in those cases, we show that it works exceedingly well. Furthermore, we show the relevance of such static serving even in today's Internet full of highly dynamic content.

A. Rules: Definition and Expressiveness

The offloaded rules comprise a *condition* that checks whether an incoming packet matches an anticipated request and a *processor* that constructs a reply in that case. For each packet that is destined for an application using Santa, the rule processor checks whether the packet matches a rule. If a rule matches, Santa replies with the offloaded response instead of forwarding the packet to the application. Otherwise, the packet is handed over to the application (e.g., via the standard socket interface in case of a kernel-based implementation, or via a dedicated control channel as in SDN) and the application will handle the packet itself, as traditionally without Santa.

A *condition* comprises an offset, a length, and the pattern to be matched. For example, a DNS server can construct a rule that recognizes packets querying an `A` record of a domain. A *processor*, likewise, comprises an offset and a length, and, additionally, either a put or a copy instruction. To continue the DNS example (see Figure 2), the response to the requested `A` record is constructed with two successive processors: a) one that contains the reply, most importantly the IP address of the requested domain, and b) one that copies the transaction ID identifying the request into the response packet so that the requesting host can correlate the received reply.

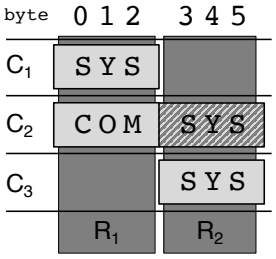
This very basic functionality has several advantages. By only allowing very specific operations when matching requests and constructing replies, we circumvent security issues that are likely if we put complex server implementations into potentially critical locations (e.g., kernel space or on shared middleboxes). Furthermore, since conditions base on basic packet content comparison, and processors either put predefined data into a packet or copy parts of the request into the response, the rules are application-*agnostic*. In our evaluation, we show that this simple ruleset suffices to offload DNS responses.

B. Efficient Rule Matching

As Santa targets high-performance applications, handling high packet rates is a fundamental requirement. Consequently, evaluating whether or not Santa has to apply a rule on an incoming packet becomes the main challenge.

This is done by checking all rules' conditions using hashes which thus is an elemental building block of our system. However, Santa should handle conditions with arbitrary offsets and lengths. Thus, a naive hashing approach, i.e., hashing each condition's pattern, would require calculating many hashes for each incoming packet (one hash for each offset-length combination that occurs in at least one condition). To reduce this overhead, Santa identifies small, non-overlapping *regions*, each defined by an offset and a length, and calculates hashes only for these regions. Then, for each rule, Santa assigns the first region that shares the offset with the rule and that is smaller or equal in length compared to the whole condition to be matched. For this, we calculate the hash of the (sub-)pattern (induced by the region on the pattern to be matched) and store a reference to the full condition and to the corresponding region in a hash table (cf. Figure 3) using the calculated hash as the identifier. This enables to identify conditions that could match to reduce the number of required full condition checks.

To check a packet, Santa calculates, for each region, the hash of the corresponding packet's sub-part and checks for the occurrence of this hash in the hash table. If such an entry exists, Santa performs a byte-wise comparison between the packet and the full condition, as a region may cover only a subset of the



Hash table

Identifier	Value
h_n(SYS)	(C_1, R_1, h_{32}(SYS)) (C_3, R_2, h_{32}(SYS))
h_n(COM)	(C_2, R_1, h_{32}(COM))

Fig. 3: Conditions C_1, C_2, C_3 are inserted into Santa. The algorithm identifies two regions (R_1, R_2) for matching. For each condition's starting region, the 32-bit FNV-1a hash (h_{32}) is calculated and put into the hash table. Here, we keep track of the condition, region, and h_{32} value for match verification.

full condition and hash collisions may occur. When a rule, and therefore the corresponding condition is added, modified, or deleted, the algorithm incrementally updates affected regions and executes necessary bookkeeping.

We explain the algorithm by means of an example given in Figure 3. In this example, two conditions start at the same offset, and a third starts three bytes later. Conditions C_1 and C_2 would yield one shared region R_1 with the same offset and the shorter length of both conditions. C_3, however, is responsible for producing another region R_2. The conditions C_1 and C_2 are linked to their starting region R_1, whereas C_3 is linked to R_2. We store these links in the hash table with the hash value of the condition's region-defined sub-pattern as the identifier.

Due to having only two regions, we only need to calculate at most two hashes for an incoming packet (instead of three, one for each condition), one for bytes 0–2 and one for bytes 3–5. The first region would hence cover all conditions having the same starting offset and length of the overall shortest condition of this set.

Naturally, in hash tables, collisions can occur. The number of collisions may be de- or increased through the choice of the hash's length. The hash's length also influences the hash table's size which again is an important factor. For an n-bit hash, the table size is $2^n p$ with p denoting the size of a pointer. Thus, e.g., a 32-bit hash table already requires 32 GB on a 64-bit architecture. However, by employing a 22-bit hash as the primary key, the hash table's size is only 32 *MB*. For our implementation, we use the FNV-1a [8] hash which is fast to compute. To reduce the footprint of the hash table, we cut the hash to the required key size.

Still, to cope with a possibly large number of hash collisions, we employ a two-stage collision resolution. First, we calculate a 32-bit hash and keep it for each condition's region-induced sub-pattern. With this knowledge, we ensure a matching 32-bit hash value. Second, we also ensure a matching region for each candidate. For example, the collision in Figure 3 (SYS) cannot be resolved by the 32-bit hash, because the collision stems from two conditions having the exact same hashing input. However, the collision can be resolved by looking at the regions. If the hash of SYS was calculated from R_1 of the incoming packet, (C_3, R_2) cannot be a valid result. Note

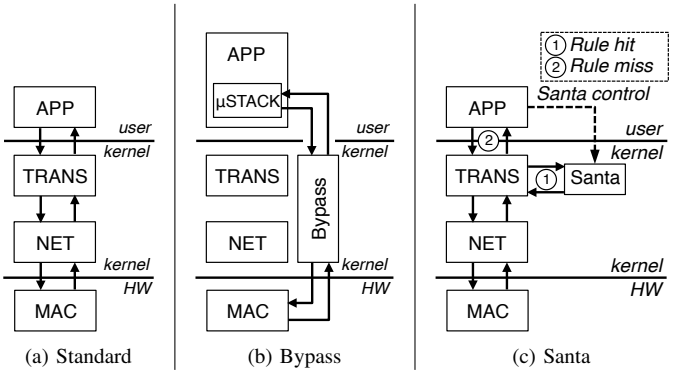


Fig. 4: Comparison of in-kernel Santa to (a) a traditional network stack and (b) user-land stacks bypassing the kernel data path. Santa provides a rule processor answering common requests without passing the packet to the application.

that the same sub-pattern from C_2 is disregarded in R_2 as the condition does not start here (cf. hatched area Figure 3). Finally, the whole condition found in the hash table has to be checked against the incoming packet to ensure that it is a correct match. This is not only due to potential hash collisions, but also because we compute the hash possibly only on a *subset* of a condition. For example, an incoming packet that contains the value COMSIS in bytes 0–5 would show up as a candidate for C_2 by containing the value COM in R_1 and is checked. Nevertheless, it is obviously not a correct match.

III. Unused Potential of Kernel Stacks

Server applications largely rely on using kernel-level stacks for packet processing. However, the packet processing performance of current network stacks suffers from a partitioned data path (see Figure 4a). In this partition, switching from the hardware to the kernel space and from the kernel space to the user space involves costly context switches and copy operations. These operations can largely degrade the packet processing performance of server systems, especially for small packets [1], [2]. To address this issue, current works propose to bypass the (inefficient) kernel entirely—and thus remove the kernel from the data path (see Section VI).

In this paper, we complement this stream of work by shortcutting the data path with Santa to accelerate packet processing in the traditional Linux kernel. We chose this example to highlight that kernel-level packet processing can be accelerated without having to abandon well-maintained and feature-rich stacks from the data path.

Therefore, we start by motivating why the kernel is slow in the remainder of this section before we describe the kernel-level implementation of Santa in Section IV.

A. The Price of Partitioning

From a system perspective, the receive data path processing can be split into three parts, each of which is handled separately as shown in Figure 4a: (1) At the bottom, the PHY and (often) MAC processing is efficiently done in specialized hardware. (2) The central portion of the processing, the network and

transport layer, is done in the operating system's network stack. (3) Finally, the application is in charge of its own specific protocols. This tripartition is mirrored in the setup of a classic protocol stack. Between each of the partitions, a context switch has to occur. Between hardware and kernel, this is done by the hardware raising an interrupt, which enables the kernel's ISR (Interrupt Service Routine) to copy data from the network hardware's data queue into main memory and trigger further processing. Once network and transport layer protocols have been traversed, the packet payload is assigned to the application it belongs to and is eventually added to the corresponding socket queue. Finally, the application can issue a system call such as `read` to receive the queued data.

This partition has the advantage of providing well-defined interfaces between each of the building blocks. This is highlighted by the fact that the generally-used Berkeley socket interface has been in use since the 1980s. However, one large disadvantage is the loss of efficiency. At each of the borders, data has to be copied from one memory area to the other. This turns out to be a problem as computing performance outpaces memory access speed.

This becomes even more apparent when considering data copy operations between user and privileged kernel space. Whenever an application needs to read or write data from or to the kernel, the memory region has to be copied. Even if not, the mode switch between the user and kernel space is a large bottleneck. Depending on the data that is transmitted, the system calls providing the interface between user and kernel space can easily account for a third of the processing time of the packet within the host [1].

B. Reducing the Partitioning Overhead

Reducing this sizable partitioning overhead in packet processing is a core motivation for many works in the field. We discuss these works in Section VI. However, approaches that have seen much exposure in recent years are netmap [2] and DPDK [9], which form a part of high-performance network server solutions such as Sandstorm [3] or optimized user space stacks such as Seastar [10].

The idea of these bypass approaches, which is visualized in Figure 4b, is to completely bypass the kernel's network stack and instead directly map data from the hardware queues into the application. The performance gain stems from two sources: First, by completely skipping network processing in the kernel, data can be directly transferred from and to user space, eliminating one copy operation. Second, the application is required to not only process application-layer protocols, but also the network and transport layer ones. By creating a specialized microstack which only contains the functionality required for the specific setup, packet processing can be further optimized for specialized use cases. They have been shown to support line rate throughput using small computing capacity for a wide range of packet sizes and use-cases [3], [10], [11].

However, one of the advantages of bypassing is also its disadvantage. By requiring the application to provide and use its own network stack for network and transport layer processing, bypassing abandons the exceedingly well-tested, well-maintained, and feature-rich network stack available inside the OS. In addition, new APIs and different programming paradigms hinder a widespread adoption in well-established networking applications.

IV. SANTA IN THE LINUX KERNEL

Given the potential performance gains, while maintaining compatibility with legacy software, we decided to implement Santa for the Linux kernel. Santa comprises a main part, which is independent of the main kernel files and resides in its own subtree, and several hooks in relevant places inside the network stack (i.e., the socket and UDP layer). The footprint of these hooks is quite small, only about 270 lines of code, of which a large portion is due to the extension of the socket options enabling the control plane (see Figure 4(c)).

A. Inserting Santa into the Receive Path

Received packets are processed in the Linux kernel within the `NET_RX` softirq. Under normal conditions (and also for Santa, if no rule matches), the packet is eventually passed into the socket buffer, ready to be read by an application's system call. If on the other hand, Santa finds that the packet matches a rule, it creates a response from the respective processor(s) and then sends out the newly created response packet(s).

We inevitably increase the time the system spends in softirq context because the send path is now also traversed during the softirq in case of a match. This is not a problem in and of itself; it is merely a result of skipping system calls and thus a key contributor to our performance increase. However, it is important to understand that this moves the potential bottleneck of the system. Normally, under high load, the system is not able to reach line rate because it spends the vast amount of its CPU time switching back and forth between application and kernel, copying data between them, and processing requests in user space. Once we remove this bottleneck, the softirq processing is potentially the next bottleneck to cope with. This may happen earlier than one might expect because, by default, the `NET_RX` softirq is only processed on the CPU the hardware interrupt was triggered on, which, depending on the hardware, might only be a single core in the system.

Thus, we require parallel processing of `NET_RX` softirqs to unlock the full potential of Santa. With receive-side scaling (RSS) and receive packet steering (RPS), the kernel already provides such a mechanism. Both mechanisms enable to process packets of the same flow on the same CPU, thus eliminating limitations of simple hardware IRQ distribution. We do not go into further details of RSS and RPS here; for this paper, it is merely important to understand that both allow distributing the processing of incoming packets efficiently over several cores, unlocking the full potential of Santa.

B. UDP Processing

We insert Santa into the UDP receive path. After a packet has been associated with an actual socket, we intercept it to check whether the incoming datagram matches a rule. This is

important as applications insert rules for specific sockets, hence, we need to know to which socket a packet should be delivered to. Otherwise, an application could hijack other applications' packets by inserting rules that match those packets. If a rule matches, Santa constructs a reply from the rule's processors, inserts it into the network stack's transmit path and discards the packet that triggered the rule. That way, it is neither handed over to the application, resulting in a duplicate answer nor triggers the costly traversal of the user–kernel barrier.

Nevertheless, we collect statistics per rule that can be made available to the user space application as a feedback channel, such that Santa does not operate fully under the radar.

C. User space API

To enable applications to offload packet processing tasks to Santa, we realized a simple user space API. Once a socket is created, the application can attach a *processor* and a *condition* (cf. Section II) that define the matching and the resulting action. We illustrate the usage of this interface by an example shown in Listing 1.

```
1  #include "santa-userspace.h" /*userspace part*/
2  struct santa_processor p = {0};
3  struct santa_condition c = {0};
4  p.type = SANTA_COPY_TEMPLATE;
5  p.buf = "COM"; p.len = 3; p.offset = 0;
6  c.buf = "SYS"; c.len = 3; c.offset = 0;
7  /*set processor for condition to specific socket*/
8  c.p = setsockopt(fd, SOL_SOCKET, SO_SANTA_P, &p,
       sizeof(p));
9  /*add the condition*/
10 setsockopt(fd, SOL_SOCKET, SO_SANTA_C, &c,
       sizeof(c));
```

Listing 1: Santa usage example – we match `COM` while responding with `SYS` on socket `fd`.

After defining the type of the processor, i.e., copy (put) data from a predefined template (cf. lines 4 and 5), this processor is attached to the socket `fd` via the `setsockopt` system call. This call returns an identifier of the newly added processor, which can be bound to a condition (cf. lines 6 and 8). Finally, this condition is also attached to the socket; from now on Santa will intercept packets matching the condition and replies with the defined answer. All other packets destined for this socket not matching the condition are forwarded to the application as usual. Update and delete methods are handled likewise.

Note that a non-root application can only alter Santa properties of its own sockets to isolate applications for security.

V. Evaluation: DNS Server

Santa enables *frequently accessed* and (temporarily) static content to be served at *lower latency* and *higher throughput* in the number of requests. We demonstrate this ability by using Santa to accelerate a DNS server as a widely-used UDP-based application [12]. Since performing name resolutions are the first steps in many Internet transactions, optimizing DNS performance helps to optimize the performance of Internet-based applications. In particular, drastic increases in throughput enable highly loaded servers to reduce the overall load and to serve a much larger number of clients with the same hardware.

A. Testbed Setup

We evaluate the performance of Santa in a testbed study. The testbed consists of a single server, equipped with a Quad-Core Intel i7 CPU running at 3.6 GHz and 16 GB of RAM. This server runs the Santa Linux kernel and a BIND 9.10.2 DNS server. We extended BIND to utilize the Santa socket options for installing rules into Santa. Four load generating clients are connected via 10 G Ethernet over a Netgear switch to the server. The selected number of clients enable creating an overload scenario fully utilizing the bottleneck server-link. The reason to focus on a high-load / overload scenario is that this challenges the performance of traditional user space packet processing the most. Due to the small packet size of DNS requests, we introduce a high amount of per-packet processing overhead. Therefore, we expect performance optimizations to be the most pronounced in this scenario. Our load generation is based on replaying DNS requests according to pre-configured popularity distributions using *DNSPerf* [13]. The workload generation is subject to artificial test and two realistic popularity distributions that we describe later in this section.

B. Baseline Performance: BIND

We start by showing that our Santa extension has no performance drawback over an unmodified vanilla Linux kernel in the absence of matching rules. That is, we compare the baseline performance of our modified kernel to the unmodified kernel when no Santa rules are installed and all requests are handled by BIND. This evaluation serves two purposes. First, it provides an intuition on the achievable performance in our testbed with unmodified standard software. Second, it helps to ensure that our Santa kernel modification has no negative performance implications on the standard kernel.

To this end, we measure the BIND 9 performance on a vanilla Linux kernel and on our modified Santa kernel striving for a maximum number of replies. We thus configured BIND to serve only a simple DNS request for a single `A`-record for a predefined domain. We expect that this minimum setup results in the least processing overhead for BIND, and thus presents an upper performance bound.

As workload, we generate DNS requests to only that resource record. We then measure the performance over 45 s intervals in which all clients perform requests in parallel while using a warmup period of ≈ 15 s. We repeat the experiment 30 times, both for the vanilla kernel and for the Santa kernel. As a performance metric, we measure the requests served by BIND per second by capturing the received DNS traffic on the clients.

We show the distribution of the achieved performance for both kernels as green and red bars in Figure 5 (see the 1 zone entry bars). For both kernel configurations (i.e., the unmodified kernel and the Santa kernel), we show the sum of DNS replies received by all the 4 workload generating clients. We observe that the maximum performance of BIND in our setup is below 640 k replies per second. Furthermore, both kernels perform virtually identical. This shows that the Santa extension has no negative performance implication on the standard Linux packet processing performance.

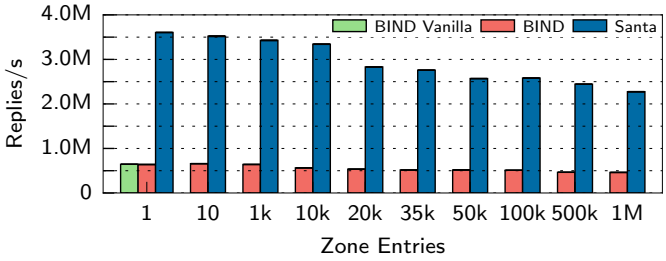
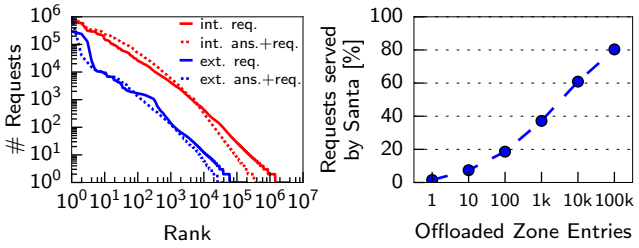


Fig. 5: Compared to BIND, Santa increases throughput by factor 5 throughout any number of installed rules. The throughput, however, decreases with higher amounts of rules.



(a) DNS RR Popularity (b) Santa Hit Rate

Fig. 6: Requests to an ISP internal and external DNS server follow a power-law (a). Thus, handling few heavy-hitters with Santa yields high hit-rates (b).

C. Baseline Performance: Santa

We next move to evaluate the achievable performance of our Santa extension. To use a realistic request structure (i.e., length and domain pattern), we extract domain names from the *Alexa* top 1 M list [14] given its widespread use [15]. To evaluate Santa's performance with respect to the number of installed rules, we extract n unique DNS names (from 1 to 1 million) from the Alexa list creating n rules and measure the reply throughput.

As a request pattern, each client requests a permutation of these n DNS names over and over. This request pattern serves as an example workload of a DNS resolver, where each entry is equally popular (we show the performance for more realistic, power-law distributed requests in Section V-E). All four clients replay the respective permutation using DNSPerf to saturate the server. On the server side, we offloaded rules for the set of n hosts, thus guaranteeing only rule hits. Preliminary tests (not shown) revealed that a hash table size of 22 bit suffices to reach near best performance in our test cases.

Figure 5 shows the sum of served requests per second while increasing the number of rules installed in Santa from 1 to 1 M entries of the Alexa top 1 M list. As in the previous section, we measure the performance over intervals of 45 s while repeating the experiment 30 times for rule set size n. Finally, we repeat the experiment with BIND and create n zone entries.

This evaluation shows a performance increases by a factor of 4.9 to 5.6 compared to BIND for all tested numbers of installed rules. Specifically, up to 10 k configured hosts, we observe a stable number of 3.3 M to 3.6 M replies per second. The performance starts dropping at 10 k configured zone entries. We attribute the observed performance drop to emerging hash collisions. That is, due to all DNS request match conditions starting at the same offset and a presumably short region, domain names with the same prefix will create the same hash values for their region. The performance further decreases with an increasing number of rules. Nevertheless, Santa clearly outperforms BIND.

This baseline evaluation already shows the potential of our approach. We remark that this use-case is artificial as no single DNS server will likely serve as many entries—and more importantly, not all entries will be equally popular. Therefore, we will now focus on evaluating Santa based on the request pattern of a real-world DNS resolver.

D. Properties of Real-World DNS Traffic

To base the evaluation of Santa on realistic real-world DNS traffic, we next analyze DNS request patterns of end-users. The data that this analysis was based on was captured in a small segment of an ISP's residential access network over the course of 60 h in May 2015. To preserve the privacy of the end-users, IP addresses were anonymized and the requested DNS resource records were hashed. As such, we could only extract access popularities of the anonymized requests.

We focus on two DNS resolvers: *i)* an ISP-internal DNS that serves as default resolver for the connected users and *ii)* an external DNS resolver voluntarily configured by *some* users. For the ISP-internal (external) DNS resolver, we observe 42.6 M (2 M) requests to 727 k (50 k) distinct resource records, respectively. The external DNS server receives fewer requests since it needs to be explicitly configured by the users.

The request frequency of the resolved resource records follows a power-law, i.e., very few DNS records receive the bulk of the requests. Processing these heavy hitters with Santa has the potential to significantly improve the overall performance of a DNS server. We show the request frequency of each resource record ordered by decreasing frequency as solid lines in Figure 6a. The power-law distribution is indicated by an almost straight line in the log-log space. As the internal (default) DNS resolver receives more traffic, the distributions are shifted.

However, DNS request-to-response mappings are only stable for limited time spans defined by the records' TTL (e.g., minutes in the case of CDNs). Changing mappings requires updates to the Santa rules. To understand how many requests can be satisfied with one mapping before an update is necessary, we next investigate combinations of (resource record, record type, answer). Each such combination corresponds to a rule that we would need to insert or update. This grouping results in 3.6 M (193 k) distinct combinations for the internal (external) DNS server, respectively. As for the requests, the combination of requests and answers also follows a power law, depicted by the dashed lines in Figure 6a.

The observed power law suggests that *stable* mappings receive substantial hits before rule updates are required. Thus, the overall DNS server performance can be significantly optimized by an accelerated processing of heavy-hitters. By

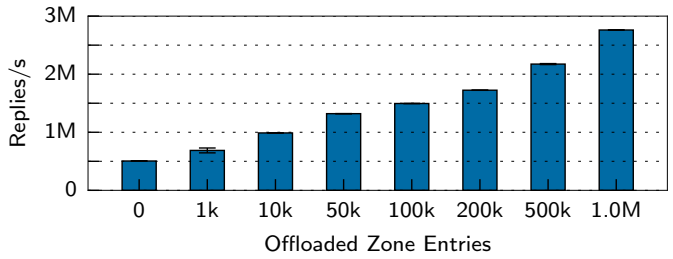


Fig. 7: BIND & Santa parallel operation: BIND offloads the top $n \in \{0, \ldots, 1\,M\}$ requests of our DNS trace (see Figure 6a) to Santa. Due to the power-law distributed popularities, this results in a more significant performance increase the more responses are offloaded to Santa by a factor of up to 5.5.

assuming an optimal offloading strategy and a-priori populating Santa with the n most popular objects, we depict the achievable rule hit rate in Figure 6b. That plot indicates that offloading as few as 100 of the most popular DNS resource records to Santa already yields a rule hit rate of 18.6 %. Increasing the amount of Santa rules to include the top 10 k requested DNS records already yields a rule hit rate of 60.8 %. Thus, a substantial amount of DNS requests has the potential to benefit from Santa accelerations. We empirically show this benefit in a testbed-driven evaluation in the remainder of this section.

We further remark that authoritative and root DNS servers offer an even greater potential to benefit from Santa acceleration due to lower and more stable set of resource records.

E. Applying Santa to Real-World DNS Traffic

We next evaluate Santa with a realistic workload pattern derived from our measurements. This workload pattern enables generating different mixtures of traffic served by BIND and Santa in parallel. That is, we configure BIND to offload the n most popular records to Santa.

We base this evaluation on the same testbed setup as used in the previous evaluations. However, this time, regardless of the offloaded request set, each client picks requests from all the Alexa 1 M hosts at random, weighted according to the probability distribution observed in the internal ISP trace (see Figure 6a). As a result and unlike in Section V-C, only the offloaded entries are served by Santa, whereas the remaining traffic is served by BIND. As the DNS trace was anonymized, we establish a canonical mapping between the hashed DNS names and hosts in the Alexa top 1 M list, i.e., the most prominent DNS hash is assigned to the first entry in the Alexa list and so on. Thus, we generate realistic rule hit rates and hence workload patterns from the anonymized data. We replay the requests to the server from all four clients while measuring 30 times for a period of 45 s for the evaluation. Again, a warmup period of ≈ 15 s was added for measuring in a stable region and we remain with our previous 22 bit hash table.

Figure 7 shows the results as the number of observed replies per second for all four clients for a varying amount of offloaded requests. Recall that this resembles a mixed scenario in which *both* BIND and Santa are active simultaneously, i.e., offloaded entries are served by Santa and the other entries by BIND. Thus, the performance is denoted as the *overall* server performance. When Santa is not active (i.e., amount of rules is 0) we see that BIND performs comparably to our previous evaluation by serving about 500 k requests per second. The slightly worse performance compared to Figure 5 is due the larger working set of resource records. Once we start offloading the most frequent requests to Santa, we observe drastic performance improvements similar to the power-law distribution of the request pattern. Concretely, we see that already offloading the 1 k most requested hosts increases the overall performance by 36 %. This speedup factor ever increases as Santa serves more and more requests. At 1 M entries, Santa serves all requests and reaches 2.8 M replies/s, a factor of 5.5, in line with the results shown in Figure 5. We remark that the overall performance of the system is primarily degraded in this mixed scenario due to BIND also using a significant portion of the computation time while serving requests. However, Santa's performance ever increases with an increasing amount of offloaded requests.

F. Takeaway

Our results show that, based on different synthetic and real-world DNS traffic patterns, Santa can substantially increase DNS server throughput by factors of 4.9 to 5.6 due to reduced per-packet processing times. Our results thus indicate that the performance of highly loaded DNS servers can be drastically improved by Santa. This can not only increase the performance on a set of existing hardware, but it can also reduce the hardware requirements for an existing workload, thereby reducing the hardware and energy cost of a server deployment: with Santa, a higher number of requests can be served with less hardware.

VI. Related Work

Classical approaches to optimize packet processing on end-hosts and servers are kernel optimizations and alternative network APIs. Proposed optimizations involve *i)* channelizing processing [16], [17], *ii)* alternative socket abstractions [17], or *iii)* using batching to reduce overheads [17], [18]. While these optimizations are application independent, improved packet processing performance can also be obtained by moving the entire server logic into the kernel. To this end classic network processing tasks, steered by the user space, such as firewalling and demultiplexing have long been a kernel feature [19] often enabled via packet filters [20] such as BPF [21] but also recent advancements in NFV make use of this concept of executing user-level code in the kernel environment [22], [23]. Other approaches implement an HTTP cache in kernel [24], [25]. They split up static and dynamic HTTP content to be handled by the kernel and a user space web server respectively. Serving static content from a kernel space web server showed to nearly double the performance [25]. While Santa benefits from similar performance improvements due to in-kernel packet processing, it is application agnostic and enables every application to offload packet processing tasks expressed via rules. To reduce the load on servers based on commonly requested items, i.e.,

HTTP or peer-to-peer content, [26] proposes an extension of TCP that allows the content provider to label cacheable items. Routers on the path cache these labeled packets and serve them directly to clients if a request is intercepted. Santa is able to process packets based on the aforementioned rules, without the need for protocol modifications for tagging such as labels. In this way, we provide a generalized and application-agnostic framework for offloading packet processing, without altering standardized protocols or requiring on path assistance.

More radical approaches involve bypassing the kernel in the data-plane by either *i)* offloading packet processing to specialized hardware, such as GPUs [4], [5], [27], [28] or NetFPGAs [29], or by *ii)* shifting packet processing to user-land stacks [2], [3], [6], [11]. The latter achieves drastic performance increases and lower CPU footprints by avoiding kernel-based packet processing overheads [3]. These advances have proven to be useful for accelerating software switches [30], HTTP [3], [6], and DNS servers [3]. However, bypassing the kernel comes at the cost of abandoning a well-maintained kernel network stack offering central administration. Likewise, new OS designs propose to generally remove the kernel from the data-plane [31], [32]. Conversely, StackMap [33] combines the standard Linux network stack with fast netmap-based network I/O, but still requires dedicating network cards to applications with limited isolation among each other, while Santa can support any number of applications and keep strong isolation between them.

Closest to our approach are application specific handlers (ASHs) [34]. ASHs enable applications to offload generic code into the kernel to handle message arrivals. These handlers are user-written routines that have to be checked for bounded execution and if exceptions are prevented. We build upon this motivation to eliminate expensive copy operations and user space/kernel space switches. In contrast, our more restricted rule-based language avoids potentially costly (security) checks required when executing generic code. It further offers the potential to be executable on a spectrum of heterogeneous target platforms, e.g., the kernel, NICs, or middleboxes.

VII. Discussion

While we discussed the design and implementation decisions and trade-offs in detail in Sections II and IV, some additional general points deserve further discussion.

Santa vs. caching. While Santa shares similarities with traditional caching, both concepts are fundamentally different. A traditional cache, be it a CPU, an OS, or a web proxy cache, acts independently of the application. It manages its cache based on heuristic rules that try to keep frequently accessed items while removing unpopular or outdated ones. In contrast, Santa's capabilities are controlled by the application. It is not fixed to a certain size, and elements are inserted and removed explicitly, not implicitly via a cache control algorithm. Furthermore, updates to outdated information are also triggered by the application itself. This eliminates two disadvantages of caching. First, it prevents that outdated information is being delivered from the cache. Second, there is no cache thrashing, because items are not added and removed based on a generic popularity metric, and there is no pressing need to remove items to add other ones.

Encrypted Traffic. A continuously increasing amount of Internet traffic is encrypted [35]—e.g., noticeable by the growth in the HTTPS ecosystem [36] or HTTP2 [37]—and can challenge rule matching performed in kernel space. An emerging example of an encrypted transport protocol is QUIC [38]. Here, encryption is applied *above* the lower transport layer (UDP) posing a particular challenge to Santa (unlike e.g., IPSec that happens before Santa matching). Since decryption is performed in user space, rules cannot match in kernel space. Although methods for matching on encrypted traffic (see e.g., [39]) or enabling a selected party to access and modify traffic (see e.g., [40]) exist, these add overhead and require client modifications.

Expressiveness of rules. Santa's rules were deliberately designed to be very simple. First, this simplicity allows less complex code in possibly sensitive privileged areas such as the kernel and thus increases security. Further, this simplicity allows finding matching rules with high efficiency, as described in Section II-B. This hinges on the fact that matches have an offset and a length so that static areas inside incoming packets are checked. On the other hand, this means that more complex protocols using variable length encoding cannot be matched. Following our QUIC example, the length, occurrence, and order of different headers are not fixed, i.e., it is not possible to match them in every conceivable position and combination without a prohibitive overhead. In such cases, a string search rule checking for the existence of a match *anywhere* in the packet would be desirable. However, the processing overhead of such rules would be much larger, and our hashing approach would also break. Thus, to implement such rules, we would need a different concept of matching.

Protocol independence. Our use case shows the common scenario for employing Santa, offloading replies from servers. This is exemplified by realizing Santa for UDP packet processing to accelerate DNS servers. However, since Santa rules are protocol agnostic and can match on arbitrary parts of incoming packets, there is no need to speak any already existing protocol. In fact, Santa also supports hooking rules earlier into the packet processing than presented in this paper, e.g., on the network layer. Matching at such an early point in the stack allows parsing all packets (not only those for a certain socket) and thus opens up the possibility to hijack packets. Therefore, installing such rules should require root access, similar to opening raw sockets. Nonetheless, inserting rules in this way opens up new possibilities, e.g., it enables to adapt or implement new transport layer protocols by defining rules that encompass the protocol behavior, without needing to update the kernel code.

Enabling in-network processing. While this paper focuses on partially offloading application logic to the kernel, it is not limited to in-kernel processing. Concretely, application logic expressed by simple SDN-inspired Santa rules can be conceptually executed in *any* network device, e.g., ranging from the NIC to programmable edge switches. Thus, this paves the way for enabling lightweight in-network processing.

VIII. Conclusion

This paper described Santa, an architecture to accelerate server systems. Santa enables applications to (partially) offload their (frequent) packet processing tasks into an application-agnostic rule-processor that can reside in various parts of the network, e.g., in an OS kernel. It thus essentially optimizes the data path by shortening it. We demonstrate this potential by implementing Santa in the Linux network stack—an application domain that is currently neglected in the presence of the proposal to remove kernel-stacks from the data path. From there, it can accelerate packet processing by avoiding costly copy operations and context switches that challenge server performance in the first place. We evaluate its benefit on the example of UDP packet processing for accelerating a DNS server, subject to an ISP traffic pattern. Our evaluations highlight that Santa can increase the number DNS requests served by a common BIND DNS server of a factor up to 5.5.

By the example application of Santa to the kernel, we show that still performance increases can be reached. Thus, the good news of this paper is: there is still life left in kernel-level packet processing. While we exemplified our approach in this domain, its design is not limited to the kernel. This protocol-agnostic approach, despite its ease of use, opens up the possibility for performance improvement to all applications that struggle under heavy load from semi-static request-response pairs. Thus, it can both increase potential throughput on existing machines, or reduce the number of servers required to handle the same workload.

Acknowledgements

This work was funded by the DFG as part of the CRC 1053 MAKI and SPP 1914 REFLEXES and the European Union's Horizon 2020 research and innovation program 2014-2018 under grant agreement no. 644866 (SSICLOPS). It reflects only the authors' views and the funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] S. Larsen, P. Sarangam, R. Huggahalli, and S. Kulkarni, "Architectural Breakdown of End-to-End Latency in a TCP/IP Network," *International Journal of Parallel Programming*, vol. 37, no. 6, pp. 556–571, 2009.

[2] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," in *USENIX Security Symposium*, 2012.

[3] I. Marinos, R. N. Watson, and M. Handley, "Network Stack Specialization for Performance," in *ACM SIGCOMM*, 2014.

[4] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated Software Router," in *ACM SIGCOMM*, 2010.

[5] I. Pratt and K. Fraser, "Arsenic: a user-accessible gigabit Ethernet interface," in *IEEE INFOCOM*, 2001.

[6] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo, "Rekindling Network Protocol Innovation with User-level Stacks," *SIGCOMM CCR*, vol. 44, no. 2, pp. 52–58, Apr. 2014.

[7] F. Schmidt, O. Hohlfeld, R. Glebke, and K. Wehrle, "Santa: Faster Packet Delivery for Commonly Wished Replies," in *ACM SIGCOMM Poster*, 2015.

[8] G. Fowler, L. C. Noll, K.-P. Vo, and D. Eastlake, "The FNV Non-Cryptographic Hash Algorithm," Internet Draft draft-eastlake-fnv-09, Apr. 2015.

[9] "Intel DPDK," http://dpdk.org.

[10] "Seastar project," http://www.seastar-project.org.

[11] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a highly scalable user-level TCP stack for multicore systems," *USENIX NSDI*, 2014.

[12] P. Richter, N. Chatzis, G. Smaragdakis, A. Feldmann, and W. Willinger, "Distilling the Internet's Application Mix from Packet-Sampled Traffic," in *Passive and Active Measurement (PAM)*, 2015.

[13] Nominum, "DNSPerf," http://nominum.com/measurement-tools/.

[14] Alexa, "Top 1M sites," https://www.alexa.com/topsites.

[15] Q. Scheitle, O. Hohlfeld, J. Gamba, J. Jelten, T. Zimmermann, S. D. Strowes, and N. Vallina-Rodriguez, "A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists," in *arXiv preprint arXiv:1805.11506*, 2018.

[16] V. Jacobson and B. Felderman, "Speeding up Networking," *linux.conf.au*, 2006, http://ns1.lemis.com/grog/Documentation/vj/lca06vj.pdf.

[17] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "MegaPipe: A New Programming Interface for Scalable Network I/O," in *OSDI*, 2012.

[18] T. Marian, K. S. Lee, and H. Weatherspoon, "NetSlices: Scalable Multi-core Packet Processing in User-space," in *ACM/IEEE ANCS*, 2012.

[19] "netfilter," http://www.netfilter.org.

[20] J. Mogul, R. Rashid, and M. Accetta, "The Packer Filter: An Efficient Mechanism for User-level Network Code," in *ACM SOSP*, 1987.

[21] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *USENIX*, 1993.

[22] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea, "Enabling End-Host Network Functions," in *ACM SIGCOMM*, 2015.

[23] S. Pathak and V. S. Pai, "ModNet: A Modular Approach to Network Stack Extension," in *USENIX NSDI*, 2015.

[24] M. Bar, "Kernel Korner: kHTTPd, a Kernel-Based Web Server," *Linux Journal*, vol. 2000, no. 76, Aug. 2000.

[25] C. Lever, M. A. Eriksen, and S. P. Molloy, "An analysis of the TUX web server," Univ. of Michigan, Tech. Rep., 2000.

[26] P. Sarolahti, J. Ott, K. Budigere, and C. Perkins, "Poor man's content centric networking (with TCP)," Aalto University, Tech. Rep., 2011.

[27] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "GASPP: a GPU-accelerated stateful packet processing framework," in *USENIX Annual Technical Conference*, 2014.

[28] S. Kim, S. Huh, Y. Hu, X. Zhang, A. Wated, E. Witchel, and M. Silberstein, "GPUnet: Networking abstractions for GPU programs," in *OSDI*, 2014.

[29] M. Flajslik and M. Rosenblum, "Network interface design for low latency request-response protocols," in *USENIX Annual Technical Conference*, 2013.

[30] L. Rizzo and G. Lettieri, "VALE, a Switched Ethernet for Virtual Machines," in *CoNEXT*, 2012.

[31] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The Operating System is the Control Plane," in *USENIX OSDI*, 2014.

[32] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A Protected Dataplane Operating System for High Throughput and Low Latency," in *USENIX OSDI*, 2014.

[33] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs," in *USENIX Annual Technical Conference*, 2016.

[34] D. A. Wallach, D. R. Engler, and M. F. Kaashoek, "ASHs: Application-specific Handlers for High-performance Messaging," in *ACM SIGCOMM*, 1996.

[35] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste, "The Cost of the "S" in HTTPS," in *CoNEXT*, 2014.

[36] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, "Analysis of the HTTPS Certificate Ecosystem," in *ACM IMC*, 2013.

[37] T. Zimmermann, J. Rüth, B. Wolters, and O. Hohlfeld, "How HTTP/2 Pushes the Web: An Empirical Study of HTTP/2 Server Push," in *IFIP NETWORKING*, 2017.

[38] J. Rüth, I. Poese, C. Dietzel, and O. Hohlfeld, "A First Look at QUIC in the Wild," in *Passive and Active Measurement (PAM)*, 2018.

[39] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "Blindbox: Deep packet inspection over encrypted traffic," in *ACM SIGCOMM*, 2015.

[40] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, and P. Steenkiste, "Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS," in *ACM SIGCOMM*, 2015.