Redundancy Elimination in Microservice Communication

Jonathan Browne, Peter Gao

1 Introduction

There is an increasing interest from the industry to shift from monolithic application architectures to microservice-based architectures [10]. A microservice architecture consists of multiple interconnected, self-contained services ("microservices"), each responsible for one small task; such an architecture enables significant scalability and modularity in application design and deployment. The traffic between microservices usually exhibits (or can be designed to exhibit) more well-defined patterns than general network traffic, simply by the fact that microservices are components in the same system. How do we take advantage of such patterns as microservice architecture designers? One way is through redundancy elimination (RE), which aims to reduce bandwidth consumption by identifying and eliminating duplicate data in communication.

There has been plenty of work on redundancy elimination for general, end-to-end traffic (e.g., [14] and [30]). However, work specifically targeting microservice traffic is relatively scarce, despite the opportunity to make use of the known patterns of, and control over, such traffic. In this paper, we will:

- examine some existing works on redundancy elimination and how they relate to RE under the microservice setting;
- propose 3 RE algorithms for microservice architectures and explain their implementation in a specific architecture (*Istio*'s *BookInfo* [6][4]);
- evaluate our Transport-layer RE implementation
- discuss our design choices and possible next steps.

The source code of our implementations is available on GitHub:

- Application-layer RE: https://github.com/peter1357908/HTTP-RE-for-Istio-BookInfo
- Transport-layer RE: https://github.com/JBYoshi/istio-proxy

2 Background and Approaches

For this paper, we focus on the sidecar-based microservice architectures. In such architectures, services run in "containers" or "pods" (in Kubernetes terms [11]), each consisting of one service coupled with a "sidecar". The service container primarily contains application logic independent of the network behavior. The communication logic (such as routing and telemetry) is outsourced to the sidecar, which serves as a mediator for all traffic into and out of the service. Having defined our focus, we are ready to examine some related works on RE for how they fit in the microservice context and how they inspired our approaches.

2.1 Web Proxy Caching

There are some widely adopted application-layer RE algorithms that can serve as good RE baselines. For example, *web proxy caching* is an *object-level caching algorithm* that operates at the application layer. It stores caches of static web content ("object") close to the clients (usually at a proxy server) so as to avoid duplicate requests for and transmissions of the same data from the servers, reducing latency and bandwidth at the same time. This idea has a few limitations, namely the fact that it relies on the object being static and cannot take advantage of redundancy over multiple objects – an object different by just one byte would need to be cached separately (e.g., an edited version of a previously sent text message).

3.2.2 demonstrates an implementation that incorporates this approach into the microservice architecture, caching static HTTP responses from an upstream service using the sidecar of a downstream service. Since sidecars essentially function as proxies, this approach is natural and simple to implement, and it bears some benefits specific to the microservice architecture that we'll discuss in 3.2.2.

2.2 HTTP Compression

Another popular application-layer RE algorithm is *HTTP compression*, which focuses on reducing the size of individual HTTP requests and responses. HTTP compression can be implemented as a combination of two forms, body compression and header compression. Header compression applies specifically to the HTTP headers. In HTTP 2.0, header compression is able to track redundancies across multiple requests and responses. [24] However, it requires that both the client and server support HTTP/2, and it does not address redundancies in payloads across requests. In body compression, a compression algorithm is applied to an individual

HTTP request or response payload to optimize for redundancy within the specific payload. [2] This RE approach is orthogonal to Web Proxy Caching, but has a similar limitation – it cannot take advantage of redundancy over multiple HTTP bodies. Further, some bodies are too small to benefit from compression.

3.2.1 demonstrates an implementation of custom HTTP body compression that's transparent to the communicating microservices (typical HTTP body compression/decompression would be done by the endpoints themselves as opposed to by proxies).

3.3, in turn, also addresses the aforementioned limitations by essentially aggregating the HTTP bodies and compressing the aggregation, building on the intuition that service-to-service communications tend to have similar data types – it specifically targets redundancy across multiple bodies, which comes with a desirable side effect: now bodies that are too small to compress individually may introduce room for compression as part of the aggregation.

2.3 Packet-level RE

There are also some protocol-independent RE techniques for the general network. One interesting approach introduced by [29] works at the network layer, aiming to achieve *packet-level redundancy elimination* that takes advantage of redundancy over multiple packets. Their algorithm compresses and decompresses packets on a hop-by-hop basis. Each node using the algorithm would cache a queue of most recent packets. For each packet seen, it would have computed a set of *fingerprints* by hashing each fixed-length substring of the packet (it uses Rabin fingerprints [27] to compute such hashes efficiently), storing a number of those fingerprints as indices to the cached packets. If a new packet contains byte strings that are shared by cached packets, the algorithm will be able to identify those through matching fingerprints and then replace them with short shims to achieve compression – downstream nodes will be able to decompress the packet with their own cached packets and information stored in the shims.

This algorithm's assumptions fit the sidecar-based architectures well – the sidecars have full control over the network layer and the service-to-service communication are hop-by-hop in nature. However, this approach doesn't really take advantage of properties of the microservices architectures: the microservice traffic won't benefit from it much more than a general network traffic would. Further, this algorithm has only been evaluated in an ideal, unconstrained setting and face some practical challenges [17][18]. All factors considered, we decided not to build on this approach.

3 Implementation

As mentioned in the previous section, we took inspiration from existing RE approaches and implemented 2 application-layer RE algorithms and 1 transport-layer RE algorithm. In this section, we will first go over our implementation environment (*Istio* and *BookInfo*) and then explain the details of our implementations.

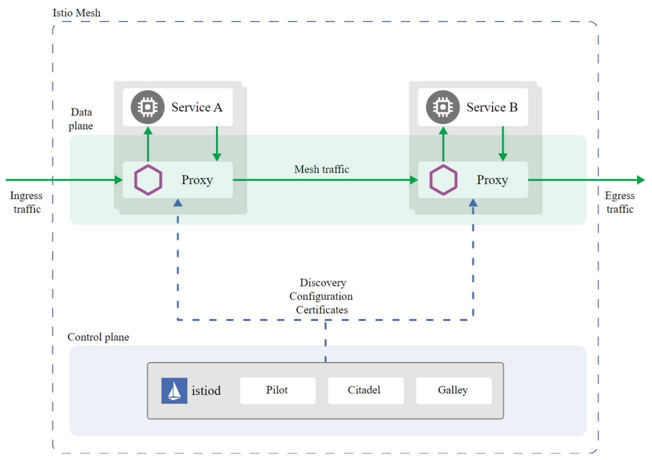

Figure 1: Istio Architecture [3]

3.1 Environment

3.1.1 Istio and Envoy

We chose to implement our algorithms in the Istio service mesh [6] in the form of "Envoy Filters". The Istio service mesh (Figure 1) consists of a data plane and a control plane: the data plane is supported by the Envoy proxies deployed as sidecars, while the control plane manages and configures those proxies. Specifically, each Envoy proxy can be figured individually through the application of "filters" that act on the traffic passing through the proxy. Oftentimes multiple filters are chained together, in the same fashion as multiple Linux commands chained by pipes. Istio allows operators to apply or modify existing Envoy filters through its "Envoy Filter" interface [5] (confusingly named). We configured existing programmable filters to implement our application-layer RE algorithms, while we built a custom Envoy filter to implement our transport-layer RE algorithm.

3.1.2 BookInfo

BookInfo is a sample Istio application composed of four separate microservices [4]; it displays information about a book, having different underlying microservices contribute different components of that information. As shown in Figure 2, ingress traffic to the BookInfo application arrives at the `productpage` microservice (akin to a front-end server), which requests data from the `details` and `reviews` microservices. There are three versions of the `reviews` microservice, two of which request data from the `ratings` microservice. We tested all of our implementations on the BookInfo application.

3.2 Application-layer RE

3.2.1 HTTP Compression

First, we'll go over our HTTP compression algorithm, implemented by configuring the programmable Lua filter [9]. The

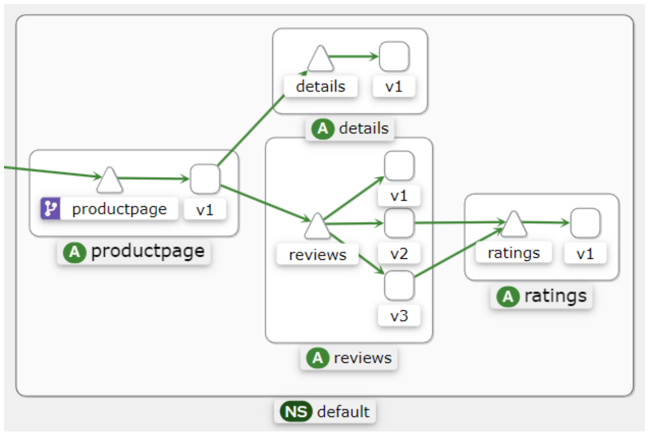

Figure 2: “Versioned app graph” of the BookInfo Application generated by Kiali

Lua filter allows running custom Lua scripts on both HTTP requests and responses. We implemented our algorithm as the following Lua scripts:

- for a response (by the current microservice) to an inbound request (from a downstream microservice), compress the HTTP body
- for a response (from an upstream microservice) to an outbound request (by the current microservice), decompress the HTTP body.

The main advantages of this approach compared with regular HTTP body compression is that the compression is handled in the sidecars and transparent to the microservices. Complying with the segregation philosophy behind sidecar-based architectures, the compression algorithm can be configured dynamically, separate from the application logic.

A downside specific to our implementation is that we need to inline the compression algorithm, since the Lua filters do not retain state across different invocations [7] and importing an external library is not well supported [8]. Of course, such a downside is not fundamental to the algorithm and can be avoided through alternative, albeit more involved implementation choices (like recompiling Envoy with a custom C++ filter as in 3.3 or compiling Rust/Go code to WebAssembly and use the Wasm filter [13]).

During evaluation, we observed that the payloads from BookInfo microservices are too small to compress. Specifically, the response body from the v3 `reviews` microservice and the `details` microservice only have 437 bytes and 178 bytes, respectively; our chosen compression algorithm [12] ended up refusing to replace the original body with the “compressed” body since the latter would be longer (due to the compression overhead). We would like to contrast this result with the high compression yield from our transport-layer RE in 4. The contrast demonstrates the importance of taking advantage of patterns specific to microservice architectures (by compressing across multiple bodies rather than one body).

3.2.2 *HTTP Caching*

Next, we present our HTTP Caching algorithm, building on top of the existing `SimpleHttpCache` filter [1]. We configured the `SimpleHttpCache` filter to cache HTTP responses to outbound requests in-memory, respecting the `Cache-Control` header specified in the response (which specifies, e.g., the cache’s Time-to-Live). Meanwhile, we configured a Lua filter for the upstream sidecars to control the `Cache-Control` header. For example, using this algorithm, when `productpage` tries to request data from `reviews` and the sidecar for `productpage` contains a valid cache of that data, it will directly return that response to `productpage` and prevent a duplicate request from being sent.

Similar to our HTTP Compression algorithm, our HTTP Caching algorithm executes in the sidecars and is transparent to the microservices, thus granting modularity. One interesting use case of this algorithm is to serve as a part of another service mesh control algorithm. For example, we can expand the Lua script to allow adjusting a cache’s TTL based on the performance of the service mesh (we can cache some content longer if we want to reduce bandwidth consumption, at the cost of freshness of that content).

The evaluation of this implementation is trivial in our setup, because BookInfo’s responses are fully static. To clarify, while there are multiple versions of the `reviews` microservice (routed to in a Round Robin fashion), caching a response from one version prevents requests to all versions of that microservice, by design.

Note that we cannot use Lua filters to write the caching algorithm itself, due to the aforementioned statelessness of Lua filters.

3.3 Transport-layer RE

Finally, we showcase our transport-layer RE, a compression algorithm that involves combining HTTP messages between each pair of services into a single stream of data that is compressed as a whole. The intuition behind this approach is that since each node runs a specific service, requests made between a pair of nodes are likely to model similar data types. These requests and responses will likely have repeated keys and data structures, so the repeated requests will have redundant data. By combining the requests and responses along a single stream of data, we can apply conventional data compression algorithms to detect and eliminate those redundancies, regardless of the specific format of the data being passed. A graphical diagram of this approach is shown in in Figure 3.

To handle connection multiplexing, we configured the Envoy proxies within Istio to reroute all internal HTTP traffic through HTTP/2. In HTTP 1.0, every request is made over a separate TCP connection. HTTP 1.1 added support for keep-alive connections, which allow multiple requests to be processed in sequence along a single connection; however, the server must send all responses in the same order that the requests were made. This means that a single slow request will

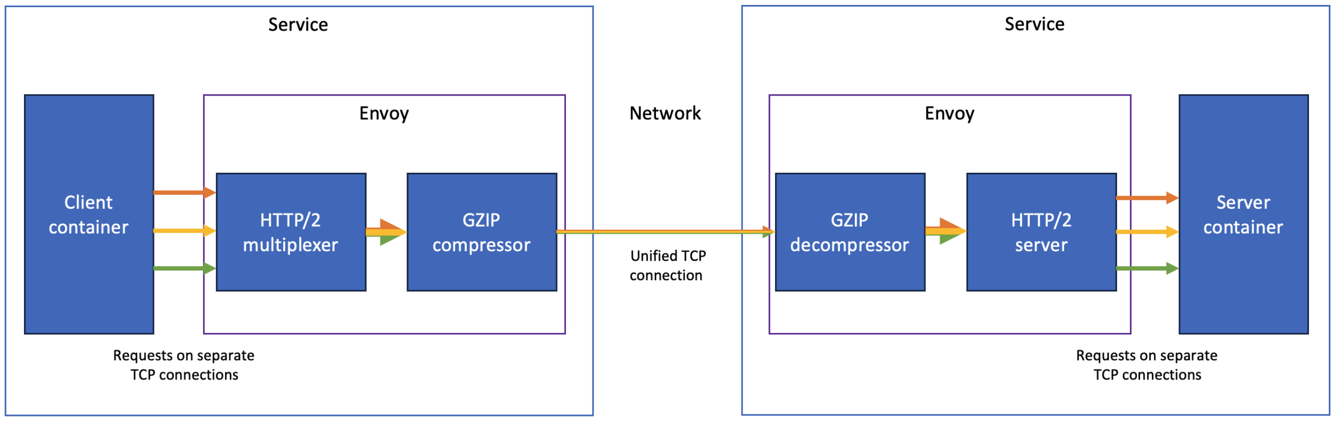

Figure 3: Architecture of our transport-layer RE algorithm.

block all subsequent requests on the same connection; even if the server processes requests in parallel, it cannot send response data for later requests until it finishes processing the earlier request. HTTP 2.0 allows requests and responses to be sent asynchronously and in different orders.

Since not all HTTP software uses HTTP/2 by default, we set up Envoy to intercept all outbound HTTP requests in the sidecar and combine those along a connection to the same service. By default, Envoy passes along outbound HTTP requests unmodified, so a service that makes calls to another service over HTTP 1.0 or 1.1 would end up with multiple connections to the destination service, and stream compression algorithms would not be able to detect redundancies across the different connections. Our implementation detects those HTTP requests in Envoy before being sent across the network and passes them through an Envoy HTTP connection pool set to only use HTTP/2, reducing the number of independent connections.

Using HTTP/2 provides some level of redundancy elimination by itself, even before doing further compression on our part. HTTP/2 uses a binary encoding for headers instead of the text-based format used by HTTP 1.0 and 1.1, which takes up less space in the output. Headers within a connection are compressed using the HPACK algorithm, which detects repeated header names and values and replaces them with references to previous occurrences of those values. For example, if services authenticate to each other using persistent access tokens in an HTTP header, HTTP/2 would only transmit the full token once, and then subsequent requests would indicate that the original value should be reused. For services with patterns of many requests with short payloads, this reduction can be significant in and of itself.

To handle redundancies within the payloads, we added a further layer of compression around the entire HTTP/2 stream. Our implementation for this uses the well-known GZIP compression format, which is flexible enough to work with arbitrary payloads. GZIP is implemented on top of the DEFLATE algorithm, which uses a combination of Huffman coding to compact individual bytes and backreferences to detect repeated bytes. We implemented this compression as an Envoy network filter written in C++ that compresses and decompresses data as it passes through the sidecar. Compression is applied after the requests are combined into an HTTP/2 connection, and decompression is applied as the connection is received, before the individual requests are split up and routed to the service.

Notably, our algorithm differs from the standardized GZIP support in HTTP due to its ability to detect redundancies across different requests. Most HTTP clients and servers already support GZIP compression, but compression is applied independently to individual request and response payloads. In communications with end hosts, which have traditionally been more common with HTTP, different payloads correspond to different objects that don't have many redundancies in common. For example, when a web browser requests a web page, some response bodies will contain HTML markup, while others will contain supplementary JavaScript, CSS, fonts, or images. In microservice environments, payloads are more uniform, so there are more redundancies between different file types that standard HTTP compression will not detect. Implementing compression on the entire stream allows our algorithm to work more effectively on the microservice use case.

4 Evaluation

We evaluated our transport-layer compression system on Istio's sample BookInfo application, which is designed to show the interaction between different microservices within Istio. We configured the microservice mesh to apply our compression algorithm on all communications within the application, including both inter-service communication and communication between the Istio ingress gateway and the service implementing the application front-end.

4.1 Compression

To test the effectiveness of our algorithm, we made 500 HTTP requests in series to the HTML service endpoint. We measured the amount of data transferred through the service mesh using a logging system in each Envoy proxy, in terms of both compressed and uncompressed data. We did not request supplementary CSS, JavaScript, or images that would be displayed in a browser, since those would only be served by the frontend service and would not demonstrate the interactions between microservices. We also ran the same test on the default Istio configuration as a control, only modified to log the amount of data transferred.

With no compression applied, our system transferred 7059 KB of data between the nodes. Recordings of the traffic transferred during the experiment indicated that all requests in this test were made over HTTP 1.1 and that no requests reused existing connections.

Our full compression scheme transferred 498 KB of data during the test after compression, for a reduction of 92.9%. This indicates that there was a very high level of redundancy in the requests and responses transferred in the test.

When the effects of the GZIP stage were excluded, our test transferred 3104 KB of uncompressed data. This provided a 56.0% reduction in data transferred over the unmodified test. Since HTTP/2 does not modify request and response payloads, this indicates that much of the data transferred in the unmodified case was through the HTTP 1.1 header format. Comparing this amount to the amount of data transferred after compression indicates that adding GZIP compression reduced the amount of data transferred by 84.0%, so the header format alone did not account for all of the redundancy.

4.2 Performance

To test the performance of our transport-layer compression system, we re-ran the compression tests while measuring the end-to-end response time of the requests. To minimize the effect of network latency, we made these requests from a machine in the same network as the service mesh, directing those requests through the ingress gateway to the frontend.

With compression disabled, the average end-to-end request time was 1.18 seconds. With our full compression algorithm enabled, the average end-to-end request time was 1.12 seconds, for a time savings of 5.1%. This suggests that our compression algorithm can cause small improvements in overall application performance.

5 Discussion

5.1 GZIP

For the transport layer redundancy elimination algorithm, we chose GZIP because it is a standardized and well-known format for compression. However, alternative compression methods have been proposed that improve on compression ratio, speed, and memory usage (such as EndRE [14] and Brotli). In this paper, we focused solely on compression ratio since that was easiest to measure. For future work, it would be interesting to compare the effects of different algorithms specifically on microservice traffic.

5.2 Implications of Aggregation

The aggregation process of our transport layer RE algorithm can also have negative effects on performance in certain circumstances. Unless service clients natively use HTTP/2, the sidecar on the outgoing machine needs to re-parse each request and send it as HTTP/2, which requires additional processing power on the sidecar. In addition, because HTTP/2 sends multiple requests over a single TCP connection, if one packet is dropped, then the stream needs to wait for that packet to be retransmitted before any later requests can be handled. However, in data center environments, round trip times are lower than over the general Internet due to the physical proximity of servers [15], so this may not be as significant of an issue. The recently-standardized HTTP version 3.0 [19] uses UDP for the transport layer to allow different streams to run independently and ensure that dropped packets only affect the specific connections that those packets contain data for, not for other connections in the stream. Our system cannot directly adapt to these changes since general-purpose compression algorithms like GZIP can only compress on one stream at a time. Algorithms like EndRE [14] may be more suitable for these protocols.

5.3 Limitation of Analysis

A limitation of our analysis is that Istio's BookInfo application is not designed to have a realistic distribution of data. We had hoped to test on a more realistic benchmarking system like DeathStarBench [21], but we did not have time to integrate our system with it.

5.4 Pitfall Avoided?

In our project proposal, we mentioned a potential pitfall of eliminating deliberate redundancy, such as redundancy introduced to increase resilience/reliability of the service mesh. This turned out to be a non-issue, since our focus lied in eliminating redundancies in payloads, whereas deliberate redundancy tends to be in terms of additional Kubernetes nodes, duplicate servers on standby, etc. – they are on a layer different from our concerns. Careful readers may recall that our HTTP cache algorithm prevented access to different versions of the same microservice given a valid cache. This behavior is okay, by definition of "valid cache"; we can easily configure the implementation to distinguish between responses from different versions of the microservice if this behavior is undesirable. In words, our algorithms do not fundamentally interfere with any deliberate redundancy; such redundancy is either on a separate layer or can be accounted for by properly configuring our algorithms.

5.5 Security Connotations

General-purpose compression algorithms are known to introduce security vulnerabilities when used in conjunction with encryption. If an attacker is able to both observe the amount of data transferred through the network and modify a portion of a compressed and encrypted piece of that data, then it is possible to extract secrets by using the length of the data as a hint at the amount of redundancy. [23] This was used in the well-publicized CRIME and BREACH attacks [28, 22, 26]. Workarounds for this generally include either separating secret data from the rest of the surrounding environment and compressing it separately [16], increasing the granularity of compressed data (as in HTTP/2's HPACK algorithm [24]; also used in [20]), or injecting randomness into the algorithm to make it harder for an attacker to measure redundancy [25]. The latter approach could be provided automatically for microservice traffic, since large-scale services handle traffic from large numbers of users at once; if it is hard for an attacker to distinguish their own traffic from other users' traffic, then traffic from other users could serve as the randomness. However, we have not verified this assumption, and the amount of effective randomness can vary based on the application and the usage.

6 Conclusion

In this paper we surveyed different existing network Redundancy Elimination efforts and explained how they inspired our approaches and implementations. We notice that some RE techniques can benefit (or fail to benefit) the microservice framework similarly to how they interact with general network traffic (e.g., the Application-layer RE algorithms), while other RE techniques take advantage of patterns unique to microservice architectures (our Transport-layer RE algorithm).

We then discussed in detail our implementations of microservice RE algorithms. Our results suggest that redundancy elimination at the transport layer can reduce the amount of traffic flowing in a service mesh by up to 92.9% by taking advantage of redundancies unique to the traffic patterns in microservices. While there is a need for more analysis of these approaches in terms of their performance in more realistic systems and in terms of their security implications, we believe this is a promising field of research that deserves more attention than it currently receives.

7 References

[1] Cache filter — envoy 1.29.0-dev-acc54c documentation. `https://www.envoyproxy.io/docs/envoy/latest/start/sandboxes/cache`.

[2] Compression in http - mdn web docs. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Compression`. Accessed: 2023-12-04.

[3] Istio / architecture. `https://istio.io/latest/docs/ops/deployment/architecture/`.

[4] Istio / BookInfo application. `https://istio.io/latest/docs/examples/bookinfo/`.

[5] Istio / envoy filter. `https://istio.io/latest/docs/reference/config/networking/envoy-filter/`.

[6] The Istio service mesh. `https://istio.io`.

[7] Lua: global table to store state between threads. `https://github.com/envoyproxy/envoy/issues/4027#issue-346800276`.

[8] Lua libraries in istio's envoyfilter. `https://www.atomiccommits.io/lua-libraries-in-istios-envoyfilter`.

[9] Lua — envoy 1.29.0-dev-cad728 documentation. `https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/lua_filter`.

[10] Microservices adoption in 2020. `https://www.oreilly.com/radar/microservices-adoption-in-2020/`.

[11] Pods | kubernetes. `https://kubernetes.io/docs/concepts/workloads/pods/`. Accessed: 2023-12-04.

[12] Rochet2/lualzw: A relatively fast lzw compression algorithm in pure lua. `https://github.com/Rochet2/lualzw/blob/master/lualzw.lua`.

[13] Wasm — envoy 1.29.0-dev-6d9a6e documentation. `https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/wasm_filter`.

[14] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. EndRE: An End-System redundancy elimination service for enterprises. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, San Jose, CA, Apr. 2010. USENIX Association.

[15] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[16] J. Alupotha, S. Prasadi, J. Alawatugoda, R. Ragel, and M. Fawsan. Implementing a proven-secure and cost-effective countermeasure against the compression ratio info-leak mass exploitation (crime) attack. In *2017 IEEE International Conference on Industrial and Information Systems (ICIIS)*, pages 1–6, 2017.

[17] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet caches on routers: The implications of universal redundant traffic elimination. *SIGCOMM Comput. Commun. Rev.*, 38(4):219–230, aug 2008.

[18] A. Anand, V. Sekar, and A. Akella. Smartre: An architecture for coordinated network-wide redundancy elimination. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 87–98, New York, NY, USA, 2009. Association for Computing Machinery.

[19] M. Bishop. HTTP/3. RFC 9114, June 2022.

[20] J. Fan, C. Guan, K. Ren, and C. Qiao. Middlebox-based packet-level redundancy elimination over encrypted network traffic. *IEEE/ACM Transactions on Networking*, 26(4):1742–1753, 2018.

[21] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[22] D. Goodin. Crack in Internet's foundation of trust allows HTTPS session hijacking. `https://arstechnica.com/information-technology/2012/09/crime-hijacks-https-sessions/`, September 2012.

[23] J. Kelsey. Compression and information leakage of plaintext. In *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers*, volume 2365 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 2002.

[24] V. Krasnov. HPACK: the silent killer (feature) of HTTP/2. `https://blog.cloudflare.com/hpack-the-silent-killer-feature-of-http-2/`, November 2016.

[25] R. Palacios, A. F. Fernández-Portillo, E. F. Sánchez-Úbeda, and P. García-De-Zúñiga. Htb: A very effective method to protect web servers against breach attack to https. *IEEE Access*, 10:40381–40390,

2022.

[26] A. Prado, N. Harris, and Y. Gluck. SSL, gone in 30 seconds: A BREACH beyond CRIME. `https://media.blackhat.com/us-13/US-13-Prado-SSL-Gone-in-30-seconds-A-BREACH-beyond-CRIME-Slides.pdf`, August 2013.

[27] M. O. Rabin. Fingerprinting by random polynomials. *Technical report*, 1981.

[28] J. Rizzo and T. Duong. The CRIME attack. `https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit`, September 2012.

[29] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. *SIGCOMM Comput. Commun. Rev.*, 30(4):87–95, aug 2000.

[30] L. Yu, H. Shen, K. Sapra, L. Ye, and Z. Cai. Core: Cooperative end-to-end traffic redundancy elimination for reducing cloud bandwidth cost. *IEEE Transactions on Parallel and Distributed Systems*, 28(2):446–461, 2017.