Hermes
A Practical Multi-Device SPARTA

Surendra Jammishetti
CSE 108C
sjammish@ucsc.edu

I. Introduction

With the threat of a global adversary looming over online communication, it's becoming a more pressing concern to have secure, reliable, messaging services. We came up with E2E encryption to protect the contents of our messages, but it's not enough to protect against a global adversary. E2E encryption doesn't hide the metadata of our conversations, as the adversary can still reconstruct who is talking to whom, and when. As former NSA Chief Gen. Michael Hayden said, "The U.S. government kills people based on metadata" [1], necessitating the need for metadata-private systems.

Groove [2], an existing system, uses mix-nets and public providers to offer a metadata-private solution but has many pitfalls. It is a synchronous system, requiring and limited to one message per round. Additionally, the latencies are in the order of epoch times, with a complex architecture due to the underlying mixnets to route messages.

The SPARTA [3] construction offers a metadata-private, anonymous communication system, and the first part of my project will detail the implementation of SPARTA-LL. Then, taking inspiration from Groove, I wanted to add multi-device functionality to SPARTA. Additionally, I've been able to get my SPARTA implementation running inside an AWS Nitro Enclave!

Repository Link: https://github.com/Suri312006/Hermes

A. Threat Model

We inherit the threat model presented by the original SPARTA paper. The adversary is a global active attacker with the following capabilities.

- control / modify all network links
- participate in the protocol
- observe traffic for an arbitrary amount of time
- can breach everything on the server excluding the enclave code

II. Base Sparta

For the core implementation of SPARTA, I followed the pseudocode provided in the paper [3]. Since an oram is required for all of SPARTA's internal data structures, I used Facebook's implementation.

A. *Facebook Oram discussion*

The Facebook PathORAM implementation requires two things to be cryptographically secure; oram clients are running in a secure enclave architecture, that also provides memory encryption as they do not encrypt on write themselves. Additionally, they implement the oblivious position map and stash from Oblix [4]. Like Oblix, it's a pure rust implementation and is recursive in nature. It recurses down until the size of the oram is small enough to fit inside of a Linear Time ORAM, which maintains obliviousness by reading/writing over each memory location per access. I'm unsure why they chose to have a constant time ORAM ($O(N)$) for the base case instead of a PathORAM ($O(\log(N))$). It would be interesting to benchmark the difference between these two base cases.

B. *Personal Implementations*

I implemented the other data structures and operations myself, using Facebook PathORAM as the underlying oram data structure.

1) Oblivious Select

   I ended up making a function that would take in a conditional, and two integers, where it would execute in constant time without branching and would return the first integer if the condition was true, and the second integer if it was false. I used oblivious select in send/fetch implementation, using it to select between two pointers obliviously.

2) Oblivious Multi-Queue

   The oblivious multi-queue is baked into the send and fetch operations, as their pseudocode constructs this multi-queue by reading the queue location from the user store and then en/de queueing when necessary.

3) Oblivious Map

   The oblivious map used for the user store is a custom construction that's not as efficient as state-of-the-art. It boasts a time complexity of $O(N\log(N))$, as it iterates through every element in the allocated oram. It's akin to the LinearTime Oram in the Facebook oram library. I implemented it this way because it's simple and trying to go for an Oblix

[4] or Wang [5] style OMAP would take way too long to get correct, especially as the only person working on this project.

C. Technical Details

The core of the Sparta server is a GRPC server that binds to a virtual socket inside of an AWS Nitro Enclave. As the enclave implementation was only done in the past week, there are a few holes that need to be cleaned up for it to be a deployable instance of SPARTA.

1) TLS

   Amazon has support for enclave traffic to be encrypted via TLS from inside the enclave, ensuring that any client and SPARTA have a secure channel for communication. It would require moving the core logic and state out of the GRPC server and instead wrap it with an HTTP server, and then proxy the traffic via NGINX as described here.

Currently, there is a custom, insecure, GRPC proxy, called Trojan that wraps around the SPARTA vsock and enables the exchange of traffic between SPARTA and clients. I originally chose GRPC is the protocol as it is highly efficient with serialization and deserialization and generally has better DX compared to HTTP.

III. Multi-Device Extension

I propose a multi-device extension to SPARTA, via a trusted proxy on the client side. This trusted proxy would be responsible for fetching said clients' messages from SPARTA on a configurable interval, and making a copy of each real message, as well as the number of dummy messages that each device would have received. Then the client's devices would fetch the messages they need, whenever they please from this trusted proxy. This would reduce the bandwidth requirement for a mobile device / other low-power devices to zero when not in use.

The key idea is that the proxy would send the messages stored for that specific device, as well as the number of dummy messages that would have been received for that device. We need to send these dummy messages since the threat model describes an adversary who can monitor all network links. If we assume that the proxy doesn't send any dummy messages and the adversary observers a client device pinging the proxy, which then returns a small datagram, the adversary can infer that the client hasn't received much traffic between the last time they fetched compared to now. This leakage would break the the notion of traffic analysis resistance as the adversary now knows precisely how much real traffic there is per client, simply by observing the proxy's response to the client fetch requests.

This notion of the proxy sending the client real messages, and then dummy messages to fill in the volume of messages that have been fetched, can be seen as an extension of deferred retrieval between the device and the proxy.

A. *Implementation details*

The proxy is implemented as a GRPC server that internally has a client connected to Trojan. It spawns a thread to fetch from SPARTA with the granularity specified by cli arguments, and then the server takes care of responding to client requests.

Additionally, the proxy has an authentication layer to ensure that any requests made are coming from verified devices, whose public keys have been stored on the proxy ahead of time. The proxy then verifies the request's signature with its stored public key, and will only process the request if the verification succeeds. I chose this form of authentication as it is reminiscent of SSH and was also relatively easy to implement. The authentication layer can be changed out for any other scheme as well.

Sends can be sent from a device or through a proxy, but it is currently implemented such that any sends from a client device are routed through the proxy, mostly for simplicity.

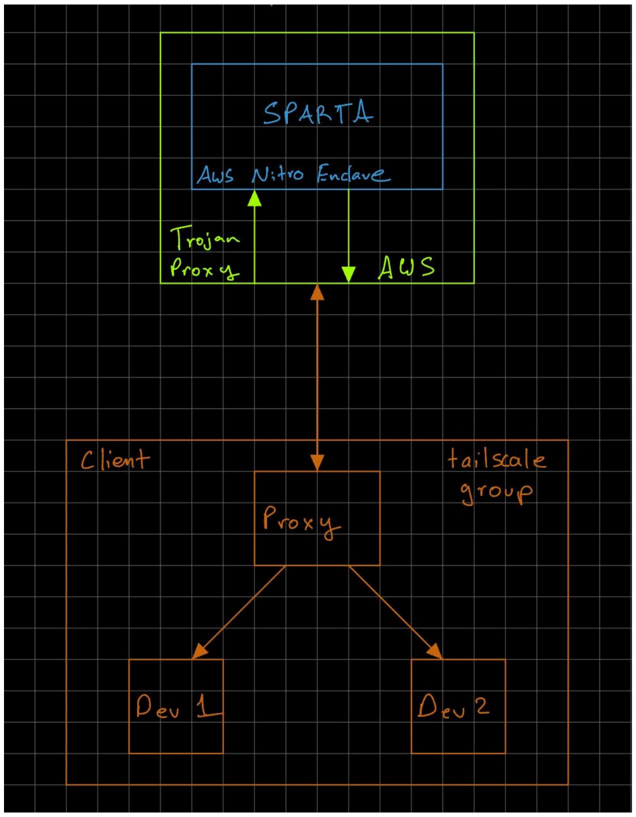

Fig. 1: Architecture diagram of Multi-Device SPARTA

IV. Experiments and Results

All experiments were run on an AWS m5.xlarge with the ami-04acda42f3629e02b image ID for the base os image. The user store was set to a size of 256 for all tests.

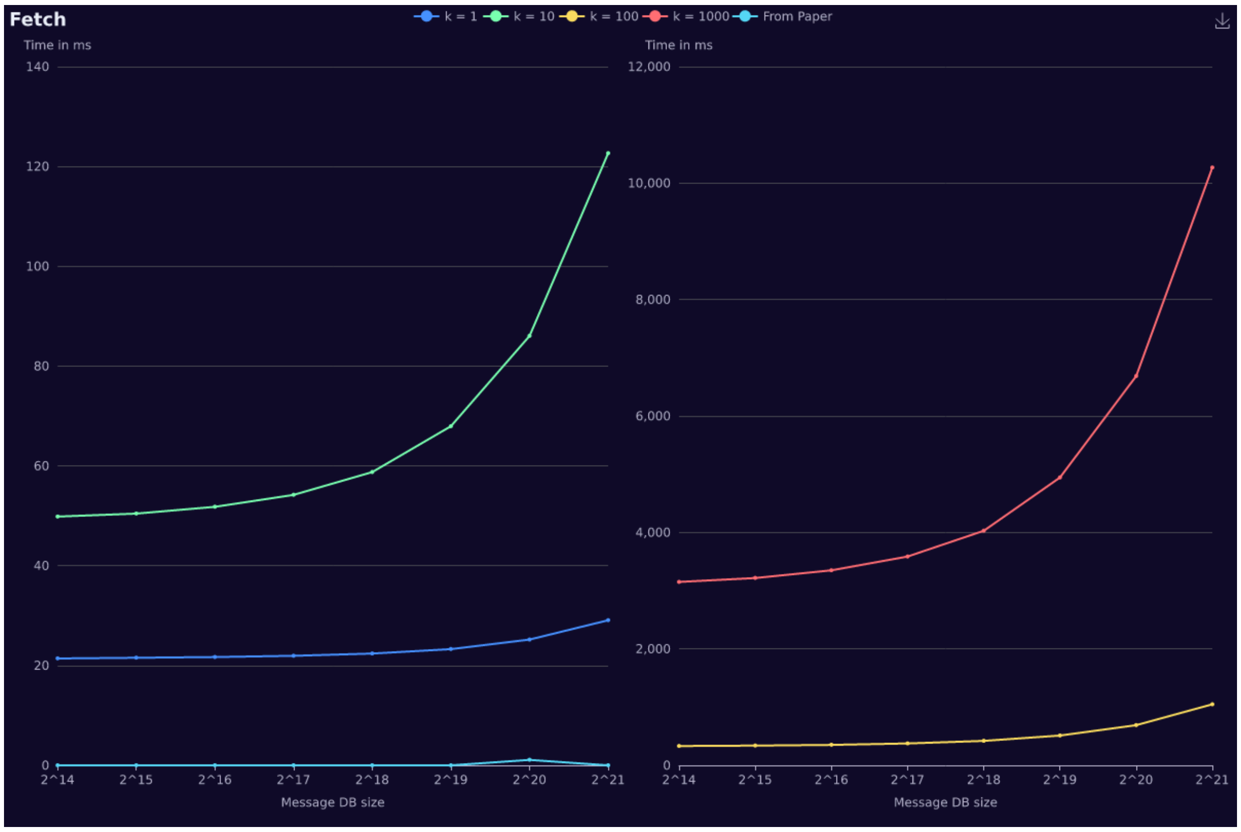

Fig. 2: Fetch benchmark with varying database sizes.
Note the blue line is data from SPARTA [3]

It's observable that my implementation is 20 times slower than the implementation done in the SPARTA [3], which could be due to a varying amount of factors. The enclave is allocated with only 2 vcpu's with 4 GB of RAM and runs inside of a sandboxed environment via the Nitro hypervisor. A more powerful rig could increase the performance of my implementation. Additionally, the LinearTime Oram used in the Facebook oram crate could increase the number of raw memory accesses compared to the implementation in the paper, further slowing down my implementation. Lastly, my server requires a proxy to get data outside of the enclave, which can add latency.

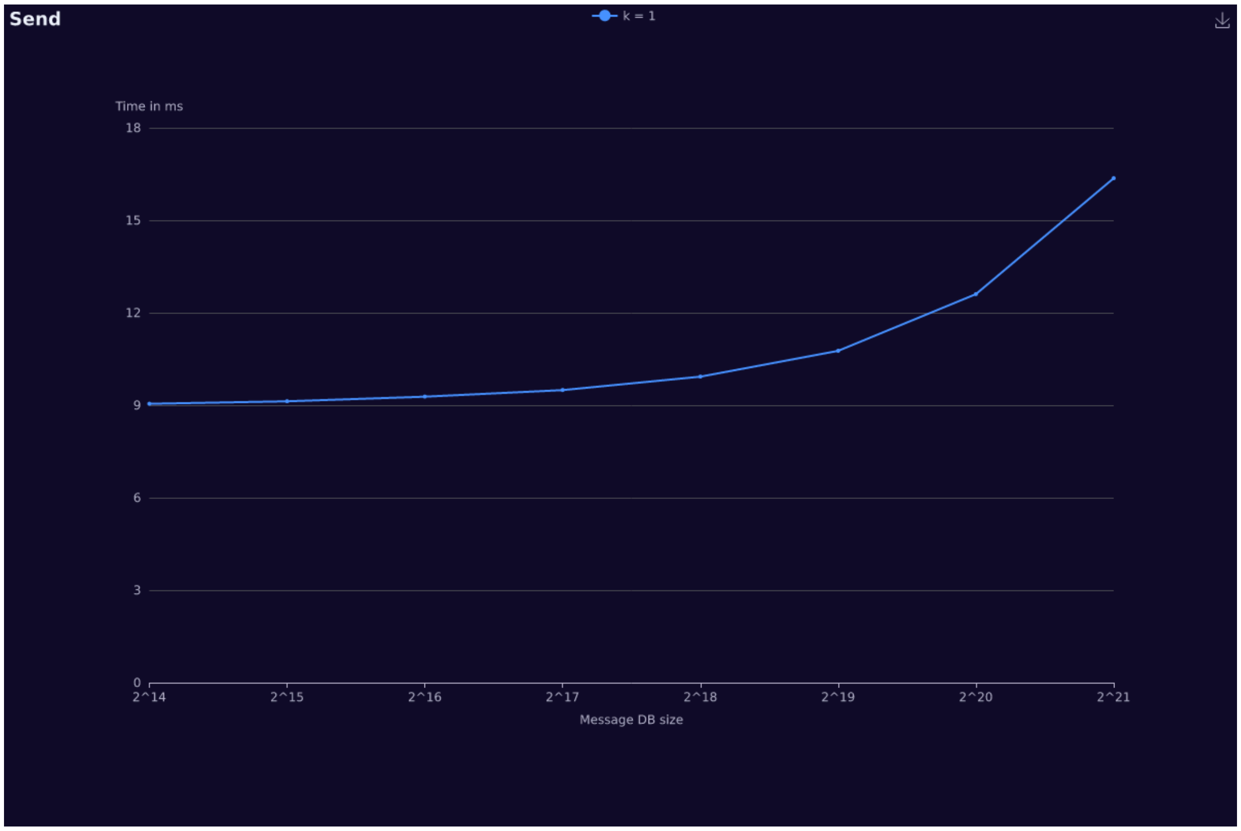

Fig. 3: Send benchmark with varying database sizes

V. Conclusion and Future Work

The multi-device proxy for SPARTA makes meta-data private communication one step closer to practicality, allowing users to have multiple devices. However, there is still a lot to make this project a deployable instance of SPARTA. As mentioned earlier, TLS needs to be integrated into the core SPARTA server, End 2 End encryption protocols need to be built into the client-side programs, and there is currently no address book to hold a user's contacts.

References

[1] L. Ferran, "Ex-NSA Chief: "We Kill People Based on Metadata." [Online]. Available: https://abcnews.go.com/blogs/headlines/2014/05/ex-nsa-chief-we-kill-people-based-on-metadata

[2] L. Barman, M. Kol, D. Lazar, Y. Gilad, and N. Zeldovich, "Groove: Flexible Metadata-Private Messaging," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 735–750. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/barman

[3] K. Fredrickson, I. Demertzis, J. P. Hughes, and D. D. Long, "Sparta: Practical Anonymity with Long-Term Resistance to Traffic Analysis." 2024.

[4] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An Efficient Oblivious Search Index," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 279–296. doi: 10.1109/SP.2018.00045.

[5] X. S. Wang *et al.*, “Oblivious Data Structures,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, in CCS '14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 215–226. doi: 10.1145/2660267.2660314.