Co-Designing Cryptographic Systems with Resource-Constrained Hardware

By

Jean-Luc Watson

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Prabal Dutta, Co-chair
Associate Professor Raluca Ada Popa, Co-chair
Assistant Professor Natacha Crooks
Assistant Professor Amit Levy

Summer 2024

Co-Designing Cryptographic Systems with Resource-Constrained Hardware

Abstract

Co-Designing Cryptographic Systems with Resource-Constrained Hardware

by

Jean-Luc Watson

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Prabal Dutta, Co-chair

Associate Professor Raluca Ada Popa, Co-chair

Sensitive private information is increasingly processed on relatively public networks and systems. Location data is constantly gathered from users' mobile phones and correlated with their activity, measuring both who they interact with in the real world, and their online behavior. Analytics workloads operate over the private data associated with millions of customers, which could include Amazon purchase histories, location trace data, or web browsing information derived from third-party cookies. Machine learning workloads are consuming every source of public (and non-public) data available through the Internet, as companies like OpenAI and Google compete to create the best and most accurate large language models. While cryptographic techniques can protect sensitive information, performant deployments are still out of reach because they assume better system capabilities than currently exist. As a result, in many cases where user privacy is at risk, the cost of protecting it directly trades off with commercial viability. For example, consumers likely would not tolerate a mobile app that privately interacted with nearby Internet of Things (IoT) devices but drained their battery, or a private ChatGPT version that required over a minute to respond to every question.

As a result, reducing the divide between the theoretical capabilities of advanced cryptographic primitives and what we can hope to reasonably compute in today's cryptographic *systems* is critical to supporting user data privacy moving forward. Thankfully, not all is lost. A new generation of heterogeneous hardware is becoming commonplace, from millions of embedded sensors and consumer mobile platforms that are more energy-efficient and computationally powerful than ever before, to server-class graphics processing units (GPUs) with hundreds of cores for general-purpose computing. Critically, however, mobile platforms and GPUs exhibit significantly limited energy and memory availability, respectively, so cryptographic systems we develop must take these into consideration to achieve practicality.

In this dissertation, I present my work on adapting cryptography for hardware resource constraints in order to achieve both performance and privacy. I will cover several systems we developed: Nebula, a protocol for embedded sensors and mobile phones to retrieve data from anywhere without leaking user participation; Piranha, a platform to accelerate multiparty computation-based ML training to privately incorporate sensitive datasets into a collaboratively-computed model; and finally, a systems approach to increase both the scale and throughput of zero-knowledge proving to improve the performance of private identity systems, blockchains, and verifiable computation. In each of these cases, considering the compute, energy, and memory constraints of specialized hardware allows us to recast expensive cryptographic problems into practically-efficient systems.

To my family, who always believed they would read this some day.

Contents

List of Figures

List of Tables

Co-Authored Material

Parts of this dissertation are based on previously published material co-authored with others, as follows.

- Chapter 2 is based on the following publication [226]:
  Jean-Luc Watson, Tess Despres, Alvin Tan, Shishir Patil, Prabal Dutta, and Raluca Ada Popa. "Nebula: A Privacy-First Platform for Data Backhaul". In: *IEEE Symposium on Security and Privacy.* IEEE, 2024.

- Chapter 3 is based on the following publication [225]:
  Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. "Piranha: A GPU Platform for Secure Computation". In: *USENIX Security.* USENIX, 2022.

Acknowledgments

I could not have even considered going to grad school without a huge amount of support from my friends and family. Maman and Papa, you helped me understand that grad school was a great option to shoot for, and as a result, it was something I had in my mind from the beginning. Thank you for not bugging me too much about when my thesis would be done (it's done!) and for accompanying me over the phone on many drives to the grocery store over the years. Guy, Pascal, and Andre, I know I can always count on you for advice (pets and relationships, plants, and mythology, respectively) and I remember fondly coming back from each semester to a home full of energy.

When I started at Berkeley, I had no idea that would also find a new family as well. Samyu, thank you from the bottom of my heart for always believing in me, listening to me, and supporting me. I have never been happier. To Anu, Lakshman, and Jamun, I've never felt more welcome – thank you for making the Bay Area feel like a second home.

I might not have made it here without the mentorship and support of many people throughout the years. Tommy, I remember well when you asked me if I was interested in a science project that led to my very first research experience, driving up from high school to campus every other day to run our experiments. I can't thank Dr. Anderson enough for taking us under her wing, helping us develop our experimental protocols, and letting us take our first steps with the scientific method. My undergraduate days wouldn't have been the same without my draw group – Justin, roommate extraordinare, Thomas, Ashwin, Andrew, Varun, Carol, Krysten, Joseph – I still have the tshirt we made for move-in day. Finally, I would not have made the jump from undergraduate to graduate school without Phil, who let me into the wonderful world of embedded systems and wireless communication.

Prabal and Raluca, thank you for supporting me, even when I wanted to explore problems that none of us had much experience in. Thanks to you, my research process, from formulating problems to evaluating and writing, has improved so much in my time at Berkeley. You enabled me to succeed, even when I didn't think I could make it, and I feel that I have achieved the goal when I set started grad school – to learn how to lead and develop a research team. Thank you to my other committee members, Natacha and Amit, for your wonderful feedback and interest in supporting my research.

I want to thank my wonderful collaborators who made research more fun and interesting, who consistently challenged me to defend my ideas and ultimately made this research better. Saharsh, Ryan, and Sherry, you made incredible contributions to our live update work, even after I threw you into the embedded systems deep-end to learn a wholly new field. Sameer, your constant mentorship and availability to walk through problems made Piranha into the successful project that it became. And Zoë, thank you for bridging the gap between the theoreticians and the systems-builders, at least in a small way. My approach to problems wouldn't be the same without intensive discussions with my unofficial collaborators: Sam, Yuncong, Branden, Josh, Will, Meghan, and Neal.

Lab11 is the singular reason I came to Berkeley in the first place. You have all been incredibly welcoming and supportive, and formed the most insightful, caring group of researchers I have ever seen over the past six years. Pat, Noah, Branden, Josh, Thomas, Matt, Rohit, Andreas, and Bernard, thank you for creating a lab environment that I was eager to stay even longer than needed each night. I'll never forget the long discussions about my nascent research questions, your excitement and desire to build cool systems, and your advice over an ill-fated Sake tasting or beer down at Jupiter. Neal and Will, I couldn't have asked for better friends. Thank you both for the shadow advising, helping me fix my bike derailleur, which works flawlessly to this day, and waiting in line with me outside of Berkeley Bowl. We will play more games soon. Oh and Aspen's Slack account, whoever they are, has pretty good taste in television. To the squad – Shishir, Tess, and Alvin – you've been amazing collaborators and friends, and I'm super proud of the breadth of our work together and the good times we had in 545W. Our trip to France was a highlight of grad school (apologies to Tess for our last-minute dash to the airport) and someday we have to actually make it to Amsterdam. Finally, to the new Lab11 members: Guangyu, Paul, and those who follow: welcome! I hope you'll have as good a time here as I did.

The PhD wouldn't have been complete without my friends from RISE (for the new kids, that's Sky). Emma, Vivian, Micah, Conor, Laura, Shadaj, Justin, Mayank, Deevesh, Darya, and Sam, our plan to revitalize a group culture at the inaugural ski retreat paid massive dividends – I enjoyed the many Friday afternoons hanging out on the terrace and hope the chairs will still be there when I visit. Thank you for your insightful feedback and support, even when I was going on about embedded systems, and of course, the project logos from Vivian and Samyu. Daniel, we took the lab by storm in the most successful unionization campaign ever seen in Berkeley EECS. It wouldn't have been possible without you leading the way. Thanks to Meghan for getting me involved in the first place, and Garrett, Tanzil, Tarini, Greg, and Kavitha for landing the plane.

No two people deserve more credit for dealing with my antics than Ed and Daniel (different Daniel this time), my erstwhile roommates during my time at Berkeley. Ed, I can't express how nice it was to move to grad school already knowing someone who was going through the same process. I'm a bit nostalgic of dodging raccoons on the way into our apartment and collaborating on some of the first real cooking I ever did. Daniel, I appreciated your support even as I was starting up the Cafe Bustelo at 10pm another late night. It was an honor surviving the pandemic together.

I am so grateful to everyone else for their friendship, camaraderie, and help throughout the PhD... so many in fact that they are hard to list without my committee complaining that I'm inflating the page count. So for you, I have mangled something Snoop Dogg once said:

> I want to thank you for believing in me, I want to thank you for helping me do all this hard work. I wanna thank you for having no days off. I wanna thank you for never quitting on me. I wanna thank you all for always being givers and trying to give more than you all receive. I wanna thank you for trying to do more right than wrong. I wanna thank you for being you at all times, y'all are bad ************s.

* * *

Funding and support for this dissertation came primarily from you (probably), the American taxpayer, including a National Defense Science & Engineering Graduate Fellowship, NSF CISE Expeditions Award CCF-1730628, NSF CAREER 1943347, the U.S. Department of Energy's Office of Energy Efficiency and Renewable Energy (EERE) under the award number DE-EE0008220, an Okawa Foundation Research Grant, the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Support was also provided by gifts from the Sloan Foundation, Alibaba, Amazon Web Services, AMD, Ant Group, Anyscale Ericsson, Facebook, Futurewei, Google, IBM, Intel, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Nvidia, Samsung SDS, Scotiabank, Splunk, Uber, and VMware.

Chapter 1

Introduction

The location and scale of data processing has shifted dramatically in the past two decades, with local applications on a personal computer giving way to massive cloud computing applications serving tens of millions of users. With respect to user data, these systems are often a double-edged sword. Machine learning (ML) models can increasingly recommend, rank, answer, or chat like a trusted confidant, yet user responses are recorded and refed into the model to improve its performance. Internet-based messaging applications have exploded in popularity, but even if message contents are encrypted, a user's social network is analyzed for interests and connections to better target advertising. Collecting sensor data on a massive scale might tie one's home WiFi network to the location of various embedded devices. And as computation is increasingly delegated to a wide array of cloud vendors or third-party platforms, leaks of personal data are now commonplace.

1.1 Computing in Public With Private Data

In all of the above cases, aggregating a user's private data with that of others and computing over it in a "public" network, such as a particular product's cloud deployment, or sharing it between different organizations, is critical to a company's success. As a result, significant research effort has led to the development of a new class of cryptographic primitives, aimed at facilitating these operations while limiting direct access to private information.

Multiparty computation (MPC) allows mutually-distrusting parties to collaborate on a computation while leaking nothing about their individual inputs other than what is exposed by the final, joint answer. MPC has been used to enable private auctions without revealing individual bids [59], allow banks to combine financial records for fraud tracking without violating financial privacy laws [193], aggregate medical data across different hospitals [208], and perform ML inference and training with input data fragmented between many different stakeholders [223, 221, 173].

Similarly, various techniques have been developed for *metadata-hiding communication*, to hide information leakage from contact patterns. These approaches can prevent, for example, a messaging service from learning that I only text my brother every 6 months, or that a particular government employee is a whistleblower in contact with the New York Times [78]. Metadata-hiding systems might have participants send messages to seemingly random digital mailbox addresses [51], shuffle messages through a hidden pipeline or crowd-sourced cluster of intermediaries [132], or inject cover traffic to obscure real communication patterns [213].

Finally, one can convince another party of the *truth* of a statement (e.g. "I am old enough to vote (18+)") without revealing the information underlying the statement ("I am 23 years old") using *zero-knowledge proofs* (ZKPs). This cryptographic capability is critical in supporting user privacy even in cases that require personal data, such as in an anonymous online voting system that must also prevent someone from casting a ballot twice, or in verifying that a particular computation was performed correctly without redoing the entire computation from scratch. Some ZKPs can be verified quickly, requiring just a few hundred milliseconds, and with minimal information, ranging from a few hundred to thousand bytes, without learning any of the dependent private information, and regardless of the complexity of the proven statement. These proofs underlie private cryptocurrency [194], delegated computation [30], and even allow users to use anonymous credentials [190].

However, each of these approaches makes a significant tradeoff between privacy and performance. Simply put, they are currently much too expensive compared to their non-private alternatives. To put this in context, *plaintext* inference using VGG-16, a common convolutional neural network for image processing, requires only 0.8 ms [209], whereas Falcon [223], an MPC-based 3-party protocol for ML inference, requires 0.79 and 1.27 *seconds* to perform the same inference when parties communicate over a LAN or a WAN, respectively – a 987 to 1588× overhead. Similarly, while modern RISC processors operate at GHz frequencies, generating a zero-knowledge proof that the computation has been correctly executed is orders of magnitude slower (although growing faster), verifying on the order of hundreds of cycles per second [152]. And while platforms like WhatsApp support upwards of 2 billion monthly active users [40], leading systems to provide metadata-hiding communication can only process approximately 50 messages per second [78].

1.2 Opportunities to Design for Resource-Constrained Hardware

A new generation of heterogeneous hardware platforms is emerging in parallel to new cryptographic primitives, offering an opportunity to close the gap between plaintext and secure performance. These platforms are at the same time more powerful, in terms of sheer numbers of devices or accelerator cores, and more limited, in that they are more specialized and trade efficiency in certain operations (e.g. more cores and parallelism for computation) for restricted flexibility (e.g. less memory on-device, or a limited programming model that is inefficient for branching logic). In this dissertation, we will focus on two platform classes with particularly widespread availability: mobile devices and graphics processing unit (GPU) accelerators.

Mobile and embedded devices have benefited from vastly increased processing capabilities and decreasing power utilization. For example, a Nordic nRF52832 Bluetooth Low Energy (BLE) radio and 64 MHz microprocessor in 2016 had a peak current draw of 152 μA/MHz [167], whereas a 2024 Nordic nRF5340 operates a *128 MHz* core at only 61 μA/MHz [168], yielding overall more efficient and powerful sensing platforms. Processors from companies targeting wearable devices, such as Ambiq, offer even better power efficiency at just 4 μA/MHz on an Apollo4 Lite SoC [12]. Memory availability has a also increased over the same time period, from the nRF52832's 64 kB to the nRF5340's 1.5 MB of RAM. In consumer mobile electronics, the resources are even greater: a 2024 iPhone 15 has 6 GB of RAM [85] and two 3.5 GHz high-performance cores in addition to four 2 GHz power-efficient cores [13]. To compound these advances, the number of smartphones in the US has increased dramatically. In 2011, only 35% of Americans owned a smartphone; now, over 97% of the population use one [159].

GPUs have also seen vastly improved performance, shifting from graphics processing workloads to general purpose GPU (GP-GPU) workloads with increased interest from blockchain users (who in Bitcoin, for example, have leveraged GPUs to parallelize hashing to mine new blocks [204]) and from the ML industry, who use vast arrays of GPUs to train Large Language Models (LLMs) [199]. A Tesla K10 GPU in 2012 could support approximately 20 GFLOPs per watt, whereas an RTX 3090 GPU, released in 2020, yields upwards of 80 GFLOPS per watt [96].

The new hardware landscape opens up a new design space in which we can seek to develop performant cryptographic systems. If expensive computation for metadata-hiding can be distributed over many mobile platforms, each much more capable than before, we could hope to increase the scalability of private communication. Likewise, the repetitive but highly-parallelizable core operations within MPC and ZK proving protocols might benefit from the high core counts in modern GPU architectures.

However, there is no free lunch when utilizing new hardware, as each platform comes with limitations to balance their advantages. Mobile platforms may have more power-efficient and capable microprocessors, but they still operate on limited energy reserves that must be carefully managed. In particular, while CPU cycles may be energy-efficient on an embedded sensor, radio communication is generally much more intensive, so balancing local processing with data transmission is a persistent problem. In GPUs, while computation speed is increasing, overall application performance is often bottlenecked by limited communication bandwidth with the host CPU and on-device memory availability. Where machine learning models might have models consuming hundreds of GBs (GPT-4 requires almost 7 TB to store its parameters [38]), even a server-class Nvidia A100 GPU only has 80 GB of on-device memory [170]. Using GPU programming models such as CUDA yield a more restrictive set of applications, which must be properly parameterized to take full advantage of the hardware.

These restrictions, however limiting, are also the source of new system design opportunities. In particular, we stand to significantly improve the performance of cryptographic applications if we can understand which parts of the computation can be fit onto mobile or

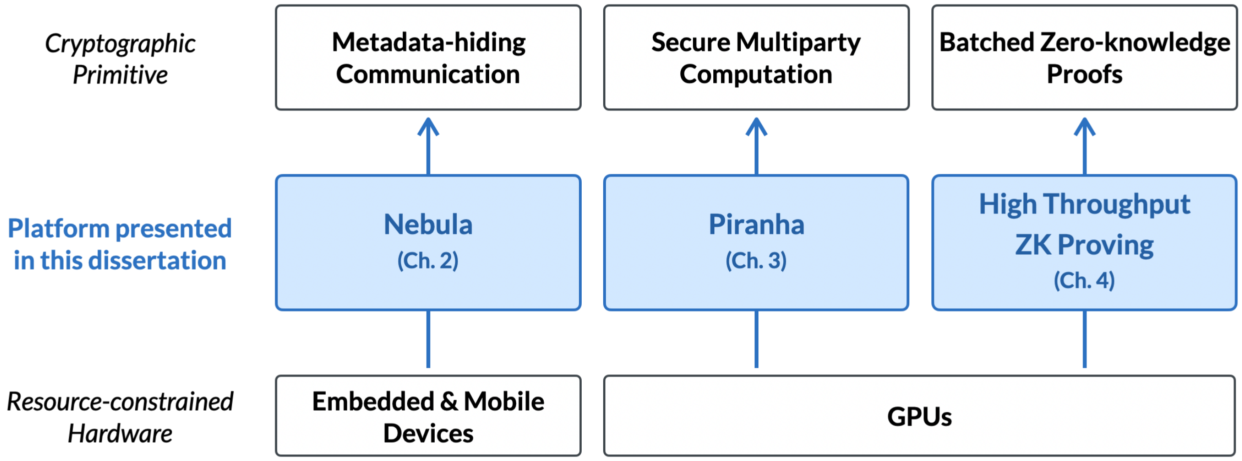

Cryptographic Primitive
Metadata-hiding Communication
Secure Multiparty Computation
Batched Zero-knowledge Proofs
Platform presented in this dissertation
Nebula (Ch. 2)
Piranha (Ch. 3)
High Throughput ZK Proving (Ch. 4)
Resource-constrained Hardware
Embedded & Mobile Devices
GPUs

Figure 1.1: This dissertation's contributions (middle, blue). In each of the technical chapters, we discuss the design of a platform that restructures an advanced cryptographic primitive (above) to execute efficiently on a constrained hardware platform (below).

accelerator hardware while working within their limits. In secure applications, the specific input data (messages, training examples, proof witnesses) may change, but the computational pattern remains the same, as input-dependent processing might leak the contents of the messages or training examples. Thus, in many cases we know the exact sequences of operations a-priori, and can work to better support them on the hardware we have. Where cryptographic primitives currently have no obvious division between existing CPUs and new hardware platforms, we can reformulate them to better suit the environment in which they will be running. Finally, given new heterogeneous hardware systems, we have the ability to reshape overall system architectures to distribute computation to mobile platforms, or focus heavyweight problems towards the cloud where accelerators are available. Therefore, we present our thesis statement, which we will support throughout the rest of the dissertation.

1.3 Thesis Statement

Real-world applications can feasibly leverage advanced cryptographic primitives without completely sacrificing performance by restructuring the primitives to move protocol execution to resource-constrained devices.

1.4 Roadmap for This Dissertation

This dissertation explores in depth how we can build performant systems for complex cryptography by leveraging the computational benefits of specialized hardware, while working within their constraints. We validate our thesis by building three general-purpose platforms, each supporting a different cryptographic primitive, which are detailed in Figure 1.1. Each platform enables a wide array of different applications that all use this core primitive.

Chapter 2 details our work in creating a new metadata-hiding architecture for data backhaul that can run on mobile and embedded devices. In turn, allowing third-party devices to privately expand network connectivity can support countless sensing applications, from distributed scientific data collection (e.g. temperature, air pollution) to package tracking or infrastructure use monitoring, anywhere on Earth, without requiring bespoke infrastructure. Prototype implementations for our sensor, gateway, and cloud-based components can be found at `https://github.com/lab11/nebula`.

Chapter 3 presents our platform for secure MPC that leverages GPU acceleration. We use it to support three linear secret-sharing protocols and privately train convolutional neural networks that would have been infeasible to securely train otherwise, requiring weeks of computation. Piranha is also designed to allow future protocol improvements to benefit from GPU acceleration, and each protocol can be extended to execute desired MPC application. We open-sourced our platform at `https://github.com/ucbrise/piranha`.

We discuss our work on enabling large single- and batch-instance zero-knowledge proofs on the GPU in Chapter 4. We target a core bottleneck in zero-knowledge proofs, multiscalar multiplication (MSM), improve latency for large single proof execution on the GPU, and allow larger MSM input sizes that leverage both CPU and GPU memory. Our platform can thus accelerate zero-knowledge attestation of more complex circuits. We also discuss an approach for improving the runtime of *batch* MSM evaluation by leveraging addition sequences, targeting high-throughput scenarios where the same proof is continually evaluated with different inputs (e.g. validating blockchain transactions).

Finally, this dissertation concludes in Chapter 5 by summarizing the lessons we learned in developing each platform, as they can be applied to adapting new cryptographic primitives to evolving device capabilities or designing new hardware support for efficient, private, systems. In particular, we find that cryptography can most effectively utilize hardware accelerators if it primarily relies on supported data types (e.g. “small” 32- and 64-bit integer fields) and computation (common symmetric and asymmetric cryptography), supplemented by more expensive operations as needed. Counter-intuitively, we show that cryptographers should prioritize low memory overheads even when it results in more computation, all the better to fit larger problem sizes onto larger device. On the other hand, we note that hardware developers could better support cryptography by adding hardware support for common building blocks like elliptic curve cryptography and increasing memory availability to support larger problem sizes.

Chapter 2

Hiding Metadata in Mobile Data Backhaul Networks

2.1 Introduction

Imagine being able to deploy a small, battery-powered device nearly anywhere on earth that humans frequent and having it send data to the cloud, running large-scale networks for wildfire monitoring, smart farming [215], search and rescue [105], censorship circumvention [182, 176], or asset tracking [81, 104] – all without setting up a physical gateway, looking up WiFi credentials, or acquiring a cellular SIM card. *Backhauling* (i.e. retrieving) data from widely-dispersed sensors is one of the greatest bottlenecks to deploying the long tail of small, embedded, and power-constrained IoT devices in nearly any setting.

Unfortunately, decoupling device deployment from the network configuration needed to transmit, or backhaul, sensor data to the cloud remains a tricky challenge, but the success of Tile and AirTag offers hope. They have shown that mobile phones can crowd-source worldwide local network coverage to, for example, find lost items. We argue that the same network design can be leveraged in an embedded sensing context.

As an example, consider deploying sensors on urban bike paths to gather usage data. Such data is crucial to inform investments in future infrastructure [145]. Rather than setting up a dedicated infrastructure to retrieve sensor results, Figure 2.1 shows the parties involved in a proxy-based approach to data backhaul, inspired by Tile's architecture. The *sensor*, with a low-power BLE radio, gathers usage data. When a *mule* (e.g. a mobile device) is within range, it connects to the sensor and collects a data payload. When the mule returns to network coverage (e.g. cellular or WiFi), the payload is uploaded to an *application server*. Throughout, a *platform provider* manages the backhaul network. While other technologies exist that could transmit data from the sensors deployed in the wide area to the cloud, they suffer from some combination of deployment hassle, high cost, and energy limits.

An ad-hoc data backhaul platform would remove the need for manual data retrieval, application-specific networking infrastructure, or costly and power-hungry wide-area con-

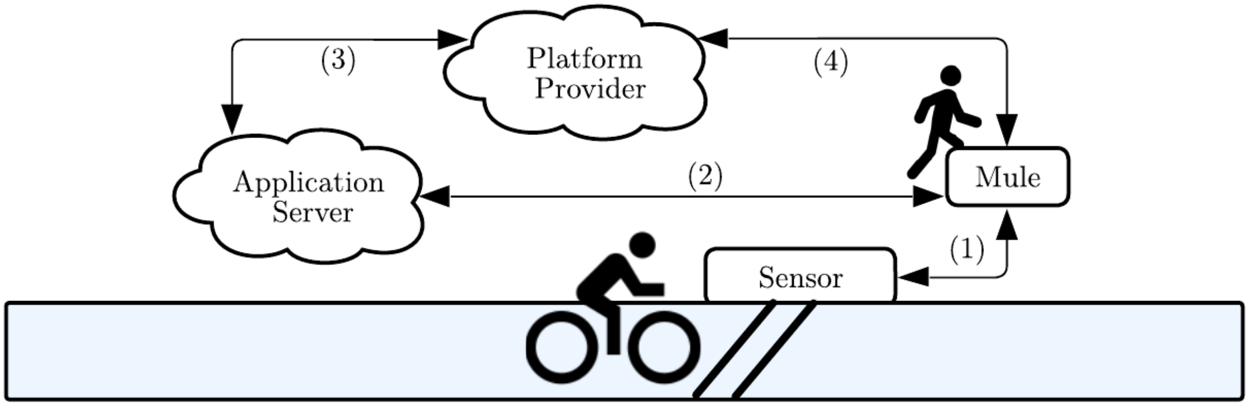


Figure 2.1: Data backhaul participants in an urban infrastructure application. Data *mules* (e.g. people with phones) pass *sensors* and collect data (1) using a local low-power wireless protocol (e.g. Bluetooth). When in cellular or WiFi range, they upload the data to an end *application server* (2). A *platform provider* administers the network by charging application servers (3), and paying mules (4).

nectivity. Recognizing the benefits of backhaul systems, several bespoke solutions exist, but with limitations. Apple's FindMy network [81] privately reports device location information to device owners, but this system is extremely application-specific. Sidewalk [9] is a static backhaul system operating on Amazon-manufactured hardware. Sidewalk, as a large-scale system, enables new sensor capabilities and extends the range of many existing devices. For example, Tile is a participant in the Sidewalk network and can retrieve tag locations through Sidewalk-enabled gateways. Third-parties can also certify and add devices [227]. However, because it is highly-centralized, Sidewalk collects device identifiers at scale which must be deleted to avoid potentially tracking passers-by [68]. Over time, a centralized provider (in this case, Amazon) is able to collect vast amounts of time-location data, tied to device identifiers, about the people who participate on the network. Privacy leakage in such a ubiquitous network is a major concern: location data about users has been shown to leak harmful information including medical conditions, religious affiliations, social relationships and location traces [73, 20, 65, 68]. Moreover, none of these systems offer a financial incentive mechanism.

As the scope of backhaul networks grows to encompass more mule devices, thus expanding network reach and capability, we urgently require a ground-up solution to privacy leakage that is practically deployable. We therefore explore the question:
How do we architect a backhaul system that minimizes the purview of the central platform provider, thereby preserving mule privacy from the provider, while enabling an incentivized, scalable data backhaul network?

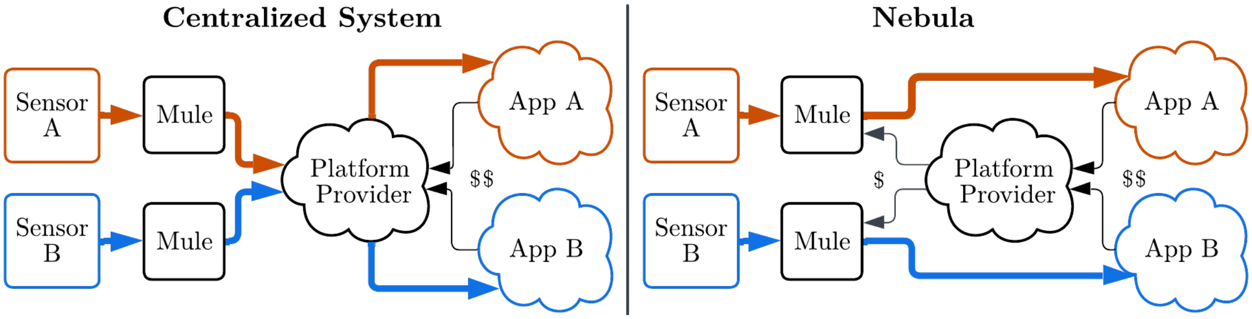


Figure 2.2: Nebula, in comparison to centralized designs such as Sidewalk [9], takes a decentralized approach. Mules route data directly to application servers, improving location privacy.

2.1.1 Nebula

In response to this question, we introduce Nebula, a privacy-first platform for general-purpose data backhaul. In contrast to prior backhaul efforts, we prevent mules from revealing their location to the provider, allow the provider to charge applications, compensate mules for system usage, and handle authentication and spam prevention in the absence of end-to-end connectivity. Nebula avoids giving the platform provider wide and deep visibility by employing a decentralized architecture, as shown in Figure 2.2. At the same time, applications in Nebula can still benefit from the ease of deployment and management that the platform provider offers: the platform provider recruits and manages a network of mules, and handles the associated payments.

Participants in Nebula upload sensor data directly to individual application servers, and interact with the provider entirely *asynchronously* from any data upload, which prevents the provider from observing backhauled data. A decentralized approach allows the platform provider to retain useful properties while eliminating a global view of identity, payment, and location. Removing the platform provider from the data path addresses our primary concern of passive observation of all mule upload behavior, but it also complicates several critical network management functions, which we discuss below. We show that Nebula (a) provides a significantly stronger notion of participant privacy while (b) placing minimal additional burdens on energy- and compute-constrained mobile devices.

2.1.2 Enabling Payment with Pre-purchased Tokens

When a provider routes data payloads from mules to application servers, like in Sidewalk [9], billing is simple: charge applications based on forwarded payloads and, if desired, reward mules. The challenge is to maintain this payment functionality while preserving mule privacy. A key insight underlying Nebula is that this process need not require payload-by-payload accounting: applications can pre-purchase tokens and distribute them to mules in exchange for data. Importantly, when redeemed, these tokens should not be linked to the party who purchased them, as this would give the provider information about the applications a particular mule was interacting with. Instead, mules in Nebula redeem these tokens on a fixed schedule, while the tokens themselves are indistinguishable from each other, avoiding additional information leakage.

We instantiate our design with PrivacyPass [64], as their token scheme is practically efficient and currently deployed at Internet-scale to automate Internet challenges like CAPTCHAs. PrivacyPass is a building block and by itself not sufficient to provide privacy-preserving backhaul. The original construction assumes that participants will directly exchange tokens with a central server in exchange for resources [64], while Nebula's decentralized setting is significantly different, in that the application servers initially receiving tokens will pass them off out-of-band to various mules who will then attempt to redeem them. Preventing misuse becomes more challenging, requiring a novel protocol that wraps PrivacyPass (Section 2.5).

2.1.3 Preventing Spam with Mule-Based Authentication

Since mules may backhaul payloads for sensors in areas that lack direct network connectivity, such as in elevators [80], on farmland [215], in parking garages [171], or on hiking trails [154], our system cannot rely on contemporaneous sensor-to-cloud connections for authentication. Instead, we assign persistent identities to sensors, which enable mules to verify sensor validity before accepting data payloads. This conserves mule resources by ignoring unauthenticated or misbehaving sensors. We establish DTLS sessions over BLE connections when mules encounter sensors, and demonstrate low session establishment costs in Section 2.10.

2.1.4 Using a Complaint Process to Handle Misbehavior

In Nebula, mules owned by third-parties directly interact with application servers. This sets up a conflict of interest, in that mules might attempt to abuse the system to gain more compensation without uploading valid payloads, while application servers might try to use mules' uploads without fairly compensating the mules. To address these issues, we present a payload delivery protocol in Section 2.5.4 that, in the case of incomplete delivery, allows mules to submit anonymous complaints of misbehavior to the platform provider.

Prototype Implementation

We implement our protocol design with BLE-enabled sensors (Nordic nRF52840-based) and BLE/WiFi-enabled mules (Espressif ESP32), and evaluate performance in deployment scenarios using real-world BLE data to estimate mule-sensor interactions (Section 2.10.3). Given the wide range of expected deployment environments, we develop an analytical model of energy and memory consumption Section 2.8. On average, our results show that sensors are able to upload data at 2.8 kB/s while drawing 40.3 mW (Section 2.10.1). Based on our measurements, we estimate that a smartphone mule can backhaul 1,000 data payloads every day while only consuming 5% battery each day and 3 MB of storage total (Section 2.10.3). We deployed application servers and a provider capable of producing and redeeming over 445,000 tokens per second (Section 2.10.4).

2.2 Background and Related Work

Low-Power, Wide-Area Networks (LPWANs) are designed to retrieve data for distributed sensor networks, and can be categorized into *licensed spectrum* cellular LPWANs and *unlicensed band* LPWANs. The most well-known and widely-used licensed band standards are NB-IoT and LTE-M. Among LPWANs operating in the unlicensed bands, the most commonly-used protocol is LoRaWAN, an open standard managed by the LoRa Alliance [143] that utilizes the 915 MHz ISM band for communication. LPWANs, while low in power, sacrifice throughput for range. WiFi, on the other hand, has high throughput but is also relatively high power. BLE does not have the range of LPWANs or the throughput of WiFi, but offers the lowest-power and lowest-cost solution [144, 141, 166, 79, 1]. Furthermore, the cost of consumer-focused cellular data plans in the United States has been falling (from $4.64/GB in 2018 to $2.75/GB in 2023) while the cost of cellular IoT remains high [205]. As a result, we argue that the ubiquity of relatively powerful mobile devices (with BLE, cellular and WiFi) have created an environment ripe for low-power, limited-range backhaul from distributed sensors.

2.2.1 Limitations of Current Backhaul Deployments

Helium

Helium is currently the worlds largest LoRaWAN network, with gateways blanketing most European and US urban areas [102]. It crowdsources participants to deploy stationary gateways, mine Helium cryptocurrency (HNT) based on the data they backhaul, and send payloads to Helium routers which forward the data to application servers [95]. Helium uses a Proof-of-Coverage (PoC) algorithm which requires miners to prove that they are providing wireless coverage to a specific region [95]. While Helium lowers the barrier to entry for sensor deployments, it is still limited: many rural areas lack coverage [112], malicious location spoofing is possible [162], and the unlicensed nature of LoRaWAN means that the upload capacity is limited [87]. From a privacy perspective, traffic is logged on a public blockchain where it can be attributed to a particular application [112]. Nebula takes an alternative approach to Helium, performing backhaul opportunistically.

FindMy

In contrast to Helium, Apple's FindMy [81] network represents a vertically-integrated, proprietary backhaul network focused on a single application: location tracking. At frequent intervals, FindMy devices advertize a rotating key to nearby FindMy-enabled Apple devices. The receiving devices use this key to encrypt and upload their own GPS location to a database. The owner of a device can then query for a position report [91] without directly allowing Apple or the other FindMy devices to learn the transmitted position report. Recent work has shown how to build third-party devices that can generate FindMy-compatible keys [101] to piggy-back on the network, and how to transmit a small stream (i.e. tens of bytes per second) of arbitrary information encoded into advertised keys [22]. However, FindMy fundamentally asks sensors to only provide key material, relying on the mules to construct the location report. When transitioning to a general-purpose backhaul network, third-party mules will upload sensor payloads with metadata (i.e. timestamp, destination application) that could leak personal information. In Nebula, a central privacy goal is to limit the information a backhaul platform provider can gain.

Sidewalk

Amazon's Sidewalk network is currently the closest deployed example of a general-purpose backhaul network, enabling Amazon-owned hardware (e.g. Ring doorbells and Alexa smart speakers) to act as Sidewalk Gateways for devices with low-power BLE and LoRA (i.e. 900 MHz) wireless radios [9, 42]. The network has a centralized architecture where endpoint devices connect to a central Sidewalk Network Server (SNS) through Gateways. The SNS then routes data to the appropriate end-destination application server. While this architecture simplifies tasks such as charging for network use and managing access control, it also means that sensors and mules must authenticate directly with the SNS as every connection is made [9]. The routing metadata provided to the SNS includes information on persistent endpoint and gateway identifiers, transmission times, and desired destination application servers. Importantly, this metadata reveals which gateways an endpoint sensor visits, which provides the SNS a centralized view of every device's last-reported location. Nebula, in contrast, ensures routing metadata is not exposed to the central platform provider.

2.2.2 Related Work

In addition to Helium, Find My and Sidewalk, many smaller-scale wireless networks inform this work. ZebraNet [117], an early wireless network, placed devices on Zebras to track their location. Large scale habitat monitoring offers interesting challenges, such an intermittent connectivity, not seen in end-to-end connected systems [148]. These early attempts, along with others [94, 183, 52], inspired many other deployments [162, 134, 228, 216, 4]. Public WiFi hotspots, including those deployed on trash cans, have been shown to leak information about the people who connect to them [211, 6]. Finally, Google deployed interaction based services with the Physical Web and Eddystone [224, 115], but ran into challenges with spam prevention.

Space or balloon-based sensor networks can also extend connectivity to sensor systems and search and rescue [202, 155, 212], but require expensive infrastructure to scale. Recent public health events have inspired BLE exposure notifications from Apple and Google [14], which maintain participant privacy through the use of exposure keys designed to restrict encounter records to individual user devices. In smart homes, energy data can reveal private information, making data processing-based privacy schemes desirable for smart meters [114]. VPriv [180] and PrivStats [181] provide location privacy when computing functions or statistics on user paths. There are many anonymous messaging protocols (e.g. for whistle blowers or activists) [44, 140, 151, 78, 47, 213, 210, 133, 51], as an alternative to Tor. Some prior work has investigated using such centralized cryptographic mechanisms for use in data reporting in opportunistic networks – Cornelius *et al.* use a mixnet to scramble incoming payloads together at a centralized location [50] and a follow-on by Kapadia *et al.* propose a statistical privacy mechanism to blur reports from users in the same geographical area [121]. In contrast, Nebula first focuses on using existing infrastructure to decentralize the backhaul mechanism, while providing stronger privacy from a centralized party, as the provider is not on the data path. Second, we investigate the implications of incentivization in this setting and propose methods to prevent misbehavior.

2.2.3 Unlinkable Tokens

Nebula requires an unlinkable token construction that satisfies two security guarantees: *unlinkability* and *one-more-token security*, as defined in Davidson et al. [64]. A scheme's tokens are unlinkable if, when redeemed to a malicious server, the server cannot link the tokens to the client(s) that generated them. Second, a scheme satisfies the one-more-token guarantee if, even with knowledge of many valid tokens, a malicious client cannot forge more valid tokens.

We use PrivacyPass, a protocol originally deployed to privately replace CAPTCHAs in CDNs, in a black-box manner to provide these unlinkable tokens, although Nebula is not tied to this construction. We provide an overview of the protocol below and refer readers to the PrivacyPass papers for a full treatment of the scheme [64, 129].

At its core, the PrivacyPass protocol uses a verifiable oblivious pseudorandom function (VOPRF) [64]. A VOPRF involves a client with some input value x and a server with a PRF secret key k, and calculates the result $t = \mathrm{PRF}_k(x)$ for the client without revealing anything to the server. Later, this result t can be verified as a valid PRF output using a publicly-revealed commitment Y to the secret key k.

A client acquires new unlinkable tokens by picking random inputs and evaluating a VOPRF over them with a central server. Crucially, the server also provides a batched discrete log equivalence proof (DLEQ) to the client, which proves in zero-knowledge that all tokens were signed by the same secret key (the k committed to by Y). For Nebula, this means that we do not even need to trust the provider to generate tokens correctly, as application servers can efficiently check their correctness. Thus, the provider cannot cause privacy leakage by, for example, using many separate PRF keys. Finally, when a mule presents an unblinded token later to the server, the server can efficiently verify the token using the public commitment to the secret key it used during token generation, without being able to identify the mule.

2.3 System Overview

In this section, we present the Nebula system overview. We begin by highlighting the stakeholders involved, and then discuss the Nebula architecture.

2.3.1 Stakeholders

In Nebula, three distinct parties work together to perform sensor data backhaul: the platform provider, application servers along with their deployed sensors, and mules.

Platform Provider

The platform provider is the coordinating entity, hosting cloud infrastructure and managing payment processes at scale. As a system administrator, the provider registers application servers and mule devices and deploys backhaul software to them. As a payment processor, the platform provider charges application providers and compensates mules based on data upload.

Application Servers

Application servers are entities that wish to deploy sensors and receive data without provisioning their own mule infrastructure. They register with the platform provider and pay based on the volume of data they upload through the mules. Before backhaul begins, they provision their sensors with Nebula-specific credentials (Section 2.5.1). Any one sensor is managed by only a single application server.

Mules

Mules are mobile entities, primarily cell phone users, that have (potentially intermittent) Internet access. They run a Nebula service that detects nearby sensors, collects the sensors' payloads, and delivers them to application servers. They are compensated for their uploads by the platform provider, which incentivizes them to upload data quickly and often. If the compensation is sufficiently large, some mules may choose to act as stationary "routers" around particularly active sensors. In general, smartphones are excellent candidates for mules, as they are mobile, have low-power wireless radios, and frequently connect to the Internet.

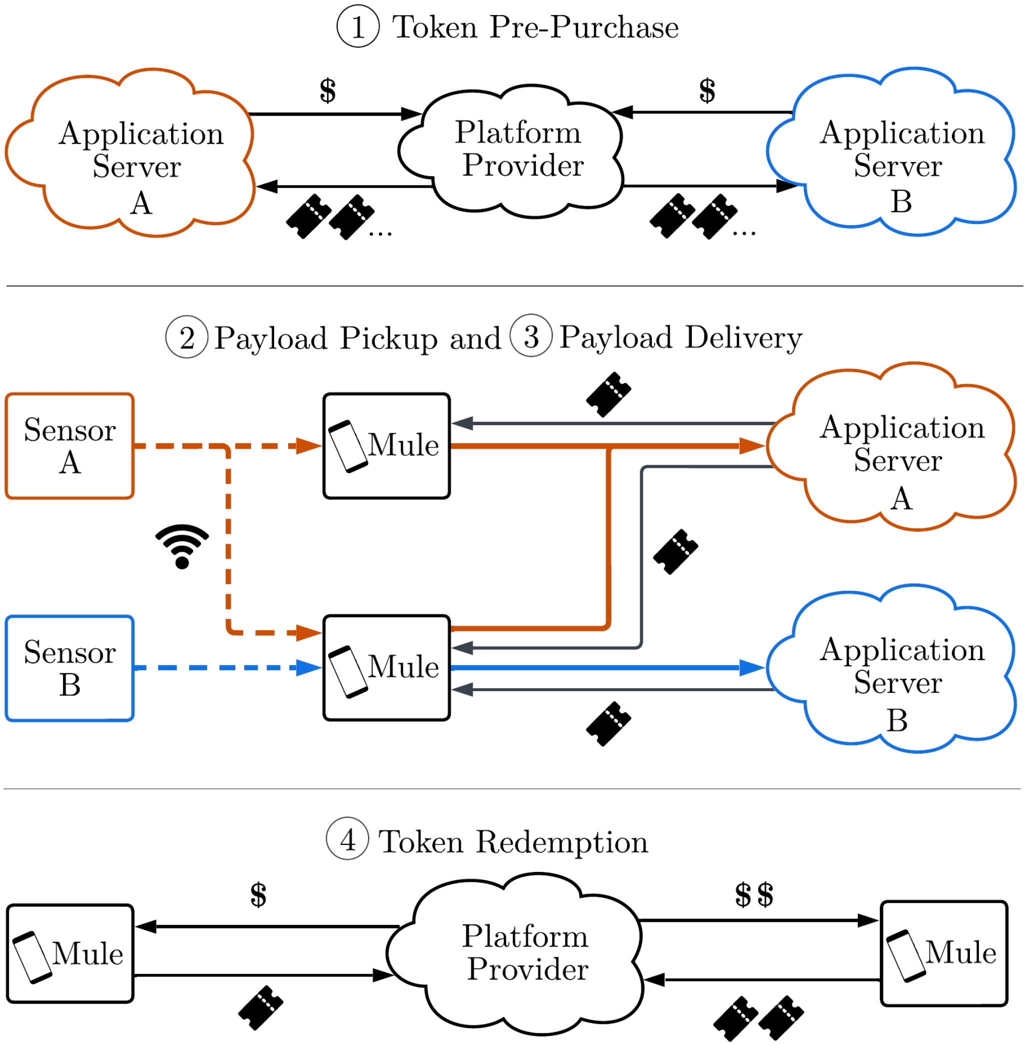


Figure 2.3: Nebula's privacy-preserving data backhaul architecture. Application servers (1) pre-purchase unlinkable tokens. When mules pass by a sensor, they (2) pick up application payloads and (3) deliver them to the relevant application server in exchange for a token. At fixed intervals (e.g. each month), mules (4) redeem tokens with the platform provider in exchange for micro-payments.

2.3.2 System Architecture

Nebula operates over a series of long-running *epochs* (on the scale of a month). Epochs correspond to points at which the platform provider performs key rotation. The epoch length forms a tradeoff between privacy and quick compensation. To backhaul sensor data to the cloud, Nebula has four main phases shown in Figure 2.3 and described in detail in

Section 2.5:

1. Application servers *pre-purchase tokens* from the provider, which can be exchanged for sensor payloads.

2. Mules opportunistically encounter deployed sensors, verify their identity, and securely perform *payload pickup* to gather data to backhaul.

3. Mules *deliver payloads* to the desired application server, in exchange for a token purchased earlier.

4. Once an epoch, mules *redeem tokens* with the platform provider and, if they detect misbehavior, can choose to expose it to the provider.

To protect mules' privacy, Nebula's architecture removes the platform provider from the data path, with payloads backhauled by mules directly to the intended application servers without a centralized routing step. Sensor data payloads are end-to-end encrypted between sensors and the application server ensuring that mules and platform providers do not have visibility into data. When picking up packets, mules authenticate sensor certificates by tracing trust up the certificate chain to the root CA (i.e. Nebula Certificate Authority). Similarly, application servers check for packet validity before accepting data and giving tokens to mules, filtering out potential spam. At the end of the epoch, the mules authenticate with the platform provider and trade tokens for payment. This scheme allows us to achieve our privacy goal (Section 2.4), since the platform provider only sees the number of Privacy-Pass tokens it signs for each application server, and the number of valid unlinkable tokens redeemed by each mule, every epoch.

2.4 Threat Model and Security Guarantees

2.4.1 System Abuse

Nebula's design includes a large system of third-party devices as backhaul mules, and rewards mule participation through micro-payments. Nebula assumes that mules and applications servers may behave maliciously for financial gain or to deplete precious network bandwidth, while the platform provider is honest for the purposes of providing soundness: namely, it will not try to render a mule unpaid, or deny service, as it is financially incentivized to continue running Nebula correctly.

Application servers are financially incentivized to minimize the cost of backhaul, so we consider scenarios in which they attempt to cheat mules, either by refusing to exchange tokens or by giving invalid tokens in return for sensor payloads. Similarly, mules might attempt to extract tokens from the application servers without uploading a payload to maximize profit. In Section 2.5.4 and Section 2.5.6, we detail Nebula's protocol for payload delivery that allows mules to submit anonymous complaints against misbehavior, but prevents mules from gaining unearned tokens. The mules themselves are third-party devices that might attempt to attack system integrity. We assume that a malicious mule can try to spam other mules with information, or impersonate a sensor for the same purpose. Mules can attempt to collude with each other to replay duplicate payloads or manufacture new ones in the hope of getting paid by the system for participating in an upload. At a high level, Nebula's soundness guarantees are that only valid payloads will result in payment and that payment is provided only once per payload. Through a (rate-limited) complaint process, if the mule correctly followed Nebula's protocol, delivered the payload to an application server, and did not receive payment, the mule will either be able to redeem payment from the platform provider or convince the platform provider that an application server is misbehaving. We state these guarantees along with their proof sketches in Section 2.6.

2.4.2 Privacy

Since backhaul deployments could span millions of personal devices, Nebula's goal is *to preserve mule privacy at the provider* in that it reveals only the following information to the provider each epoch (e.g. each month):

1. How many payloads each mule uploaded system-wide,
2. How many prepaid payload deliveries each application server purchased from the provider, and
3. A set of anonymous complaints against application servers for misbehaving.

At the protocol level, Nebula does not reveal mule identifiers to the provider and application servers during their communications; as for non-Nebula-specific network information that can leak identity (e.g. IP addresses), we rely on complementary mechanisms for anonymizing Internet communication, such as Tor [69, 232] or secure messaging/mixnets [187, 44, 140, 78, 47, 213, 51] that allow clients to anonymize their network metadata when connecting to untrusted servers. For the rest of this chapter, we assume that mules can connect to the platform provider and application servers without revealing their identity.

Nebula prevents wide-scale privacy leakage to a service provider by removing the provider's visibility of data payloads containing mule identifiers, destination applications, and encounter timestamps, aggregated over all participants and applications. It is important to note that Nebula's decentralized design does not place additional trust in any application server (AS). Each AS, whether they use a centralized upload service like Sidewalk, or a decentralized service like Nebula, inherently has access to information encoded in their own sensor data payloads, which could include time of upload and sensor location. Thus, ASes may still observe time and location of uploads of their own payloads from the mules as a natural consequence of receiving timely device-specific data from sensors that have been deployed in known locations. In some cases, this can indirectly leak information about mule paths [68]

– so in Nebula, the mules are compensated by the application servers for their work as well as for their exposure, and mules choose which application servers they wish to serve.

Collusion

We provide malicious privacy against a platform provider that can collude with other application servers and mules. We consider this strong threat model in particular because nothing stops the platform provider from deploying their own application servers or mules to interact with the remainder of the Nebula deployment.

A platform provider colluding with a subset of application servers will be able to observe all information seen by any of the colluding parties, but not the state of honest application servers or mules. In particular, the malicious parties will be able to identify the tokens exchanged for any payload (containing time, location, and other application-specific information) uploaded to the colluding application, which they can later link directly to a mule when they redeem the associated token. However, the colluding parties cannot observe actions a mule takes that that do not interact with a corrupted application server, and payloads a mule uploads to an honest server are still unlinkable to the tokens that are eventually redeemed. In the case of a complaint, mules may leak a small additional amount of information as detailed in Section 2.5.6. This threat model is mirrored in the security definition presented below.

2.4.3 Formal Definition

To formally define Nebula's privacy guarantee, we use the simulation paradigm of Secure Multi-Party Computation [39]. We provide context for our definition here, and reserve a more detailed treatment, including an explanation of how our simulation-based definition matches the informal guarantee above, for Section 2.7.

We refer to an experiment called the *real world*, which contains parties running the actual Nebula protocol, and an *ideal world*, which encodes what the adversary $\mathcal{A}$ learns in a privacy-preserving backhaul system. In both worlds, we consider what information leaks when executing all possible sequences X of Nebula operations (i.e. token purchase, payload delivery, token redemption, complaints, and epoch changes). Below, we define what it means for Nebula to be privacy-preserving, where λ is the security parameter.

Definition 1 (Privacy-Preserving Backhaul System)**.** *Let π be the protocol for a backhaul system, providing parties with the API defined in Section 2.5.*

We say that π is privacy-preserving if there exists a non-uniform probabilistic polynomial-time machine $\mathcal{S}$ such that, for every non-uniform probabilistic polynomial-time machine $\mathcal{A}$, and for every valid sequence X:

$$\{\text{REAL}_{\pi,\mathcal{A}(z),X}(1^\lambda)\}_\lambda \overset{c}{\equiv} \{\text{IDEAL}_{\mathcal{S},\mathcal{A}(z),X}(1^\lambda)\}_\lambda$$

where $\overset{c}{\equiv}$ denotes computational indistinguishability.

Theorem 1 (Privacy in Nebula)**.** *Assume a semantically secure encryption scheme, a existentially unforgeable signature scheme, a collision-resistant hash function, and a simulator for Unlinkable Tokens $\mathcal{S}_{UT}$ in Definition 2. Then π_{Nebula} is privacy-preserving as defined in Definition 1.*

We further describe our simulation-based formalism and proof sketches in detail in Section 2.7 following a description of Nebula's core protocol.

2.4.4 Limitations

Nebula does not consider the existence of a number of orthogonal sources of leakage and misbehavior that can be mitigated through existing means. For example, the deployed application sensors may attempt to glean identifying information from any data a mule wirelessly broadcasts (e.g. physical chipset irregularities [88] or unique advertised information[21]). These problems are not unique to Nebula – local wireless device tracking and identification has seen significant recent interest [137, 100, 35], and progress in that area can complement a Nebula deployment.

Given that they are mostly deployed in uncontrolled areas, sensors are likely to face physical compromise. Malicious sensors might attempt to impersonate other sensors or mules, tying up computational and communication resources by constantly advertising new or malformed payloads. Misbehaving sensors can be blocked by their current MAC address, and in extreme cases, a mule can simply turn off their radio until they move to a different physical location. However, since backhaul systems are by nature best-effort, this work does not consider denial of service attacks with limited local impact. Similarly, just like any other deployed internet-connected service, Nebula cloud parties (provider and application servers) could experience DDoS attacks, which can be remedied with complimentary solutions proposed in [120, 26, 142]. In line with *best-effort* backhaul, if mules choose to drop data or never connect to WiFi, application servers could lose some data, but in this case mules will not be paid. To add more reliability, application servers can choose to pay for duplicate data packets.

2.5 Privacy-First Backhaul Protocol

In this section, we outline the Nebula protocol. This includes device provisioning and deployment, token pre-purchase, payload pickup and payload delivery along with epoch updates and complaint management.

2.5.1 Provisioning and Deployment

Application servers are provisioned with a public-private keypair pk_{as}, sk_{as}, a matching certificate for pk_{as}, $c_{pk_{as}}$, signed by the platform provider, and an AES symmetric key k_{comm} preshared with the platform provider. The asymmetric keypair is used to attribute messages

Algorithm 1 Token Purchase

```
Input: Vector of $n_t$ randomly sampled blinded tokens
Output: Vector of signed tokens $T$ s.t. $|T| = n_t$ or $\perp$
Participant(s): Between App Server (AS) and Provider (P)
1: AS pays P to purchase $\mathrm{n}_t$ tokens and P updates its count of tokens purchased by AS in
   the current epoch.
2: AS randomly samples a vector of $\mathrm{n}_t$ values $T'$ and performs $t = \mathsf{PrivacyPass.Sign}(t')$ with
   P using the delivery keypair on each value $t'$ in $T'$.
3: If the signing protocol succeeds, AS obtains a vector of signed tokens $T$ s.t. $|T| = n_t$,
   else $\perp$.
```

to the application server when delivering a payload, and k_{comm} protects the confidentiality and integrity of token commitments made by the application server for the platform provider. The application servers check $c_{pk_{as}}$, and additionally use Certificate Transparency [41] to prevent impersonation. Mules also check these certificates against Certificate Transparency.

Each application server provisions and deploys each of their sensors s with a unique sensor ID id_s, an AES symmetric key k_s, a public-private keypair pk_s, sk_s, and a matching certificate for pk_s, c_{pk_s}, that is signed by the application server.

The sensor ID id_s is used by the application server to identify the source of incoming payloads, and k_s secures end-to-end payload confidentiality and integrity between the sensor and the application server using AES-based authenticated encryption with additional data (AEAD). k_s can be securely derived from id_s and a secret key known only to the application server, such that the application can easily and quickly derive k_s from the sensor ID. In the event that a sensor needs a new key, the application server can assign a new id'_s to the sensor, but must physically redeploy it with the new secret key. pk_s and c_{pk_s} are used by nearby mules to locally authenticate the sensor, establish a secure wireless connection, and verify membership in a specific deployment.

Finally, each epoch, mules are granted a small, fixed number of t_c *complaint tokens*, which can be verified by the platform provider using a different PrivacyPass keypair than normal data upload tokens (they are not interchangeable). These tokens limit the number of complaints an individual mule can lodge against application servers each epoch; above this limit, mules should stop interacting with particularly malicious application servers.

2.5.2 Token Pre-Purchase

Before the mule can upload the sensor data to the application server, the application server must pre-purchase tokens that can be exchanged with mules for delivering sensor payloads as in Algorithm 1. To prevent the platform provider from identifying which application servers each mule sent data to (which may be associated with mule activity or location information), the application server generates a large set of signed tokens by executing the

Algorithm 2 Payload Delivery

```
Input: Application identifier ($\mathrm{id}_{AS}$), data ($d$), payload hash ($P_{hash}$), and signature ($\sigma_{hash}$)
Output: Tokens and complaint record ($t$, $c_t$) or $\perp$
Participant(s): Between Mule (M) and App Server (AS)
1: M creates an anonymous encrypted channel with AS identified by $\mathrm{id}_{AS}$.
2: M sends $P_{hash} = (H(d), \mathrm{id}_s)$ and $\sigma_{hash} = \mathsf{Sign}(P_{hash}, sk_s)$ to AS.
3: AS checks the sensor's id and $\mathsf{Verify}(P_{hash}, \sigma_{hash}, pk_s)$, as well as that there are no duplicate payloads matching $H(d)$, aborting if both checks do not succeed.
4: AS samples a random nonce $r$ and unused token $t$.
5: AS encrypts the token $\hat{t} = \mathsf{Enc}(t, sk_{comm})$ using a token commit secret key $sk_{comm}$ shared by AS and P, then sends pre-delivery payload $P_{pre} = (r, \hat{t}, H(d))$ and signature $\sigma_{pre} = \mathsf{Sign}(P_{pre}, sk_{as})$ to M.
6: M checks $\mathsf{Verify}(P_{pre}, \sigma_{pre}, pk_{as})$ succeeds, or aborts.
7: M sends $d$ to AS.
8: AS verifies $H(d)$ matches the payload hash in $P_{hash}$, or aborts. AS sends the unencrypted token in $P_{token} = (r, t, H(d))$ and $\sigma_{token} = Sign(P_{token}, sk_{as})$ to M.
9: M checks $\mathsf{Verify}(P_{token}, \sigma_{token}, pk_{as})$ succeeds, or aborts.
10: On success, M retains token $t$ and a complaint record $c_t = (P_{pre}, \sigma_{pre}, P_{token}, \sigma_{token})$. If the protocol did not complete, M obtains an empty token and complaint record $c_t = (P_{pre}, \sigma_{pre}, d)$, and adds $c_t$ to its complaint list $C$. If the protocol aborts, the M obtains $\perp$.
```

PrivacyPass signing protocol (PrivacyPass.Sign, Section 2.2.3) with the platform provider, for some monetary cost over an encrypted TLS sessions.

2.5.3 Payload Pickup

When a sensor s desires to upload sensor data, it broadcasts wireless advertisements with a Nebula-specific identifier to nearby devices. Any mule that passes by can then identify and choose to connect to the advertising sensor. To confirm that the sensor is authorized to send data using Nebula, the sensor authenticates to the mule while establishing a TLS session, from which following communication over the wireless connection derives confidentiality and integrity. For example, malicious nodes in the same location cannot inject traffic into the wireless link. Most importantly, the sensor identifies itself to the mule using its certificate c_{pk_s}, which the mule can use to confirm that the sensor belongs to an application server for which it has chosen to backhaul data. The mule does not mutually authenticate with the sensor to prevent directly leaking its identity to the application. If session setup fails, the mule ignores the sensor and closes the wireless connection.

If authentication is successful, the sensor generates data payload and end-to-end encrypts it using authenticated symmetric encryption with a fresh nonce under key k_s to create d. This encryption ensures that only the correct application server can verify and decrypt the sensor

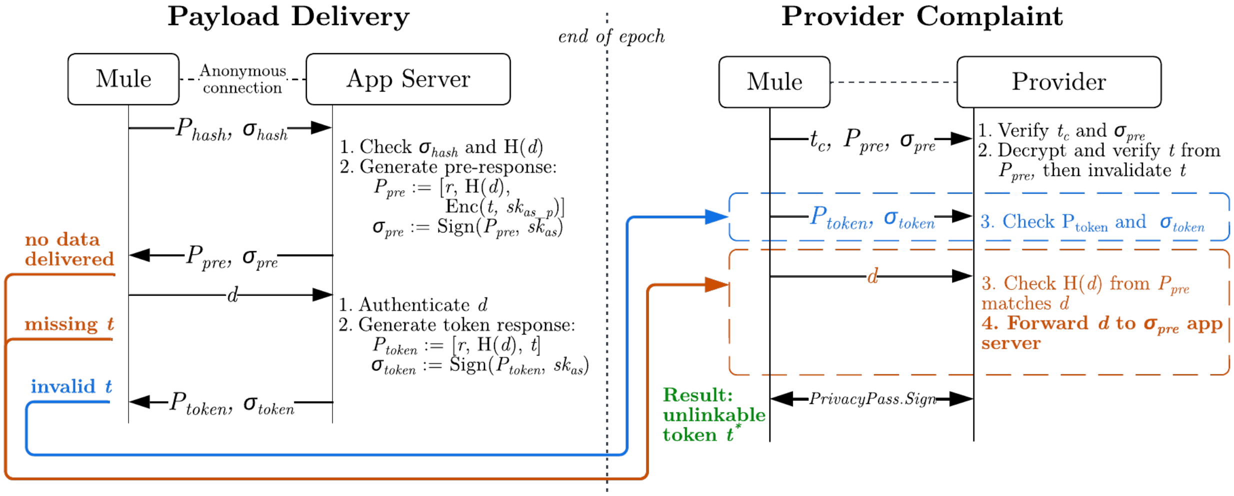

Payload Delivery
end of epoch
Provider Complaint
Mule
Anonymous connection
App Server
P_{hash}, σ_{hash}
1. Check σ_{hash} and H(d)
2. Generate pre-response:
$P_{pre} := [r, \mathrm{H}(d), \mathrm{Enc}(t, sk_{as\_p})]$
$\sigma_{pre} := \mathrm{Sign}(P_{pre}, sk_{as})$
no data delivered
P_{pre}, σ_{pre}
d
1. Authenticate d
2. Generate token response:
$P_{token} := [r, \mathrm{H}(d), t]$
$\sigma_{token} := \mathrm{Sign}(P_{token}, sk_{as})$
missing t
invalid t
P_{token}, σ_{token}
Mule
Provider
t_c, P_{pre}, σ_{pre}
1. Verify t_c and σ_{pre}
2. Decrypt and verify t from P_{pre}, then invalidate t
P_{token}, σ_{token}
3. Check $\mathrm{P_{token}}$ and σ_{token}
d
3. Check H(d) from P_{pre} matches d
4. Forward d to σ_{pre} app server
Result: unlinkable token t^*
PrivacyPass.Sign

Figure 2.4: The Nebula delivery (top) and complaint (bottom) protocols. Using a complaint token t_c, a mule can submit a complaint to the platform provider alleging misbehavior if the exchange is not completed (orange), or if the token is invalid (blue). If valid, the platform provider grants the mule a new token and forwards the missing payload to the application. The mule reveals nothing in the complaint other than that it interacted with the application in that epoch.

data payload, and provides confidentiality and integrity against tampering by other parties that obtain the payload. The sensor also generates a header payload P_{hash}, containing id_s and a hash of the payload H(d), and signs the header with its secret key, yielding signature σ_{hash}. This hash can allow the application server to identify when duplicate payloads are being submitted (Section 2.5.5). The sensor then sends the mule P_{hash}, σ_{hash}, and d. Once a mule has received a complete backhaul payload – P_{hash}, σ_{hash}, and d – it is ready to deliver the payload to the application server.

2.5.4 Payload Delivery

To deliver a payload, the mule forms a TLS connection through an anonymous communication service with the destination application server. Before accepting a payload, the application server verifies the hash of both the payload P_{hash} and signature σ_{hash}. If the payload is valid, the application server picks a token t it will exchange for a valid data d upload. The payload delivery scheme is shown in detail in Figure 2.4 and Algorithm 2.

During the exchange, payloads may be malformed, so the mule and app servers must carefully verify every payload and signature. We also identify three non-trivial cases where misbehavior during the delivery phase could result in an application server not receiving a payload or a mule not receiving a valid token in exchange (Figure 2.4), which require an out-of-band *complaint* process. In particular, (1) a mule may simply decide not to deliver

Algorithm 3 Token Redemption

```
Input: Tokens accumulated $T$
Output: $T_{invalid}$ or $\perp$
Participant(s): Between Mule (M) and Provider (P)
M authenticates with P and sends token list $T$
For each $t$ in $T$, P checks that (1) PrivacyPass.Verify($t$) succeeds and (2) $t$ is not yet in P's redeemed token list. On success, P adds $t$ to its list of M's redeemed tokens. On failure, P adds $t$ to invalid token list $T_{invalid}$. If a duplicate, P adds $t$ to the original redeemer's $T_{dup}$ list.
P sends $T_{invalid}$ to M, or $\perp$ if the verification aborts.
For each invalid token, M adds the corresponding complaint record retained from payload delivery (Algorithm 2) to its complaint list $C$.
```

payload d after receiving P_{pre}, (2) an application server might not respond with any token t, or (3) it might respond with an invalid token. We show in Section 2.5.6 that a separate complaint process can force a mule to upload missing payloads in the case of (1) and confirm application server misbehavior to the platform provider in the case of (2) or (3).

2.5.5 Token Redemption

Once per epoch, each mule forms a secure TLS session with the platform provider, authenticates itself, and redeems its accumulated tokens. The platform provider verifies each token has been correctly signed using PrivacyPass.Verify and indicates any invalid tokens to the mule. The server must then verify that the valid tokens are not duplicates of redeemed tokens, in order to prevent replaying tokens for economic gain. As immediate remuneration is not critical, Nebula allows the platform provider to check for replays in the background, and notifies mules at the end of the epoch if any of their tokens have seen duplicate submissions. Finally, the provider compensates mules out-of-band for the number of valid, unredeemed tokens they submit.

In order to prevent the database of already-submitted tokens from getting too large, we flush old token reuse databases from previous epochs when the platform provider rotates keys each epoch. The Nebula platform provider retains the token database and key material from the most recent previous epoch so that old tokens can still be redeemed by mules across one epoch boundary, but rejects any older tokens. Application servers are responsible for only distributing tokens corresponding to the current epoch, otherwise, mules may submit complaints to the platform provider after having receiving invalid tokens.

2.5.6 Complaining About Misbehavior

We address the significant latitude for issues during payload delivery by allowing mules to register complaints with the platform provider, once per epoch, with the hope of recovering a

Algorithm 4 Complaint

Input: A single complaint token (t_c) and record (c)
Output: A valid token for redemption t^* or $\perp$
Participant(s): Mules execute on the start of a new epoch

```
M creates an anonymous encrypted channel with P and sends complaint token $t_c$ and complaint record $c_t$ generated by payload delivery (Algorithm 2).
P checks that $\mathsf{PrivacyPass.Verify}(t_c)$ succeeds with the current epoch complaint keypair, $t_c$ is not yet in P's used complaint token list, $\mathsf{Verify}(P_{pre}, \sigma_{pre}, pk_{as})$ succeeds using the $\sigma_{pre}$ in $c$, and $\mathsf{PrivacyPass.Verify}(t)$ succeeds with the current epoch delivery keypair, for the decrypted token $t = \mathsf{Dec}(\hat{t}, sk_{comm})$
On failure, P sends $\perp$ to M. On success, P adds $t_c$ to its used complaint token list and adds $t$ to its redeemed token list to prevent duplicate redemptions.
If $c$ contains a $P_{token}$ payload, M sends $P_{token}$ and $\sigma_{token}$ to P, who checks that $\mathsf{Verify}(P_{token}, \sigma_{token}, pk_{as})$ succeeds and the token in $P_{token}$ is actually invalid or a duplicate, and aborts if not. Sends $\perp$ if a mule has already complained about this token as a duplicate.
Otherwise, M sends $d$ to P, who verifies that $H(d)$ matches the hash in $P_{pre}$ and forwards $d$ to the AS that signed $\sigma_{pre}$ if successful and aborts if not.
M samples a random value $t^{*'}$ and performs $t^* = \mathsf{PrivacyPass.Sign}(t^{*'})$ with P using the next epoch delivery keypair.
If the signing protocol succeeds, M obtains a token $t^*$ redeemable in the new epoch, else $\perp$.
```

token lost to misbehavior. Note that to do so, mules must leak a small amount of information – they interacted with the application server they are accusing at some point in this epoch and the size of the payload – privacy-conscious mules retain the ability to never avail themselves of the complaint process. For the platform provider, complaints against specific applications can be fed into a rating system that can identify misbehavior over time and allow the provider to take action. To complain, the mule uploads one of its limited complaint tokens and information about the delivery interaction for the platform provider to evaluate at the start of an epoch as described in Algorithm 5. Tokens provided by honest ASes will not collide, but dishonest ASes that collude to reuse tokens will be identified as malicious during complaint process. The bottom section of Figure 2.4 and Algorithm 4 detail the complaint process.

2.6 Formal Soundness Guarantees

Given the specifications of each phase, we prove below a group of closely-related properties related to the soundness of the core Nebula protocol: payload delivery and complaints.

Algorithm 5 New Epoch

```
Input: Complaint count $n_c$, prior epoch $T_{dup}$ token list
Output: Complaint tokens and valid tokens ($T_c$, $T^*$) or $\perp$
Participant(s): Mules execute on the start of a new epoch
1: Each M marks the next epoch active, randomly samples a vector of n$_c$ values $T'_c$ and
   performs $t_c = \mathsf{PrivacyPass.Sign}(t'_c)$ with P using the next epoch complaint keypair on each
   value $t'_c$ in $T'_c$.
2: If the signing protocol succeeds, each M obtains a complaint token list $T_c$ s.t. $|T_c| = n_c$,
   else $\perp$.
3: M adds complaints matching tokens in $T_{dup}$ to $C$.
4: Each M anonymously runs the complaint protocol (Algorithm 4) with each of the complaint
   payloads in its $C$, obtaining a set of new tokens $T^*$ valid in the new epoch, else
   $\perp$.
```

Claim 1 (Nebula valid token exchange)**.** *Assuming an existentially unforgeable signature scheme, in the Nebula system, any mule M that delivers a payload to an honest AS using Algorithm 2 will receive token t only if* $\mathsf{Verify}(P_{hash}, \sigma_{hash}, pk_s)$ *succeeds in Algorithm 2 step 3.*

Proof Sketch. If the check at step 3 fails, the AS aborts and returns $\perp$ instead of a valid token. Soundness follows from the *existential unforgeability* of the signature scheme, because the signature σ_{hash} on P_{hash} could only have been generated by the sensor who knows pk_s. □

Claim 2 (Nebula duplicate payload upload)**.** *Assuming an existentially unforgeable signature scheme, a semantically secure encryption scheme, and a collision-resistant hash function, in the Nebula system, any mule M that delivers a valid payload twice to an honest AS using Algorithm 2 (i.e.* $\mathsf{Verify}(P_{hash}, \sigma_{hash}, pk_s)$ *succeeds in Algorithm 2 step 3) will receive at most one token t.*

Proof Sketch. On the first and second deliveries, if the payload d does not match P_{hash}, M will not receive any tokens per Claim 1. If M can present a valid P_{hash}, σ_{hash}, and d to the AS, on the first delivery, AS can either abort the protocol, yielding no token, or send one valid token t in P_{token}. AS stores the payload hash $H(d)$, such that on the second delivery, the payload hash will match and abort the protocol. This follows from the *deterministic* property of the hash function, and results in at most one token. Note that while an encrypted token is present in P_{pre}, the *semantic security* of the encryption scheme ensures that only AS or P could decrypt the ciphertext to reveal a valid token. M could attempt to file a complaint with the provider using Algorithm 4 in a bid to retrieve another token using the (P_{pre}, σ_{pre}) it received in Algorithm 2 step 5 and either d or (P_{token}, σ_{token}) from Algorithm 2 step 10. In the former case, the provider would simply invalidate the token returned by AS in

Algorithm 4 step 3 before generating the new token, and in the latter case, the provider would abort in step 4 as the original token is still the one valid token. □

Claim 3 (Nebula token guarantee)**.** *Assuming an Unlinkable Token scheme with unlinkability and one-more-token security guarantees, an existentially unforgeable signature scheme, a collision-resistant hash function, and a semantically secure symmetric encryption scheme, in the Nebula system, for any honest mule M, for any application server AS, if M validates P_{pre} in Algorithm 2 step 6, M will either receive a token t that can be successfully redeemed in Algorithm 3 or convince the provider of AS misbehavior.*

Proof Sketch. After receiving a P_{pre} message from the AS with a valid signature σ_{pre}, there are four cases:

Case 1. Upon M sending d to AS, AS does not respond with P_{token} and σ_{token}; the AS is attempting to steal the data without rewarding M with a token. M saves a complaint record as described in Algorithm 2 and, at the end of the epoch, lodges a complaint using Algorithm 4. P verifies that P_{pre} is correct, which relies on the *existential unforgeability* of the signature scheme, and M sends the payload d to P which matches the hash in P_{pre}. Given the *collision-resistant* property of the hash function, this is the payload that the AS originally committed to receiving, so P forwards the payload to AS and signs a new token for M. M can verify that this token was signed with the correct key, following from the Unlinkable Token scheme's *unlinkability* guarantee, resulting in a valid redeemable token t^*.

Case 2. AS responds with P_{token} and σ_{token}, but the token t contained in P_{token} is flagged as invalid by P in Algorithm 3; the AS gave M an invalid token. M saves a complaint record and lodges a complaint at the end of the epoch using Algorithm 4. P verifies σ_{token}, which relies on the *existential unforgeability* of the signature scheme, and that P_{token}'s token cannot be verified using the Unlinkable Token scheme. This implicates AS in providing an invalid token, because AS checks that tokens are correctly signed by P at purchase time, which follows from the *unlinkability* guarantee. P signs a new token for M, which M can likewise verify, resulting in a valid redeemable token t^*.

Case 3. AS responds with a P_{token} and σ_{token} but the token t in P_{token} is a duplicate of a token already redeemed. M and the mule that redeemed t earlier can lodge a complaint at the end of the epoch, with P verifying σ_{token} as above in Case 2. As the AS gave M an already-used token, the different P_{token} payloads in each complaint will contain the same t, convincing P that the signing AS(es) are misbehaving.

Case 4. AS responds with a P_{token} and σ_{token} containing a valid redeemable token t. □

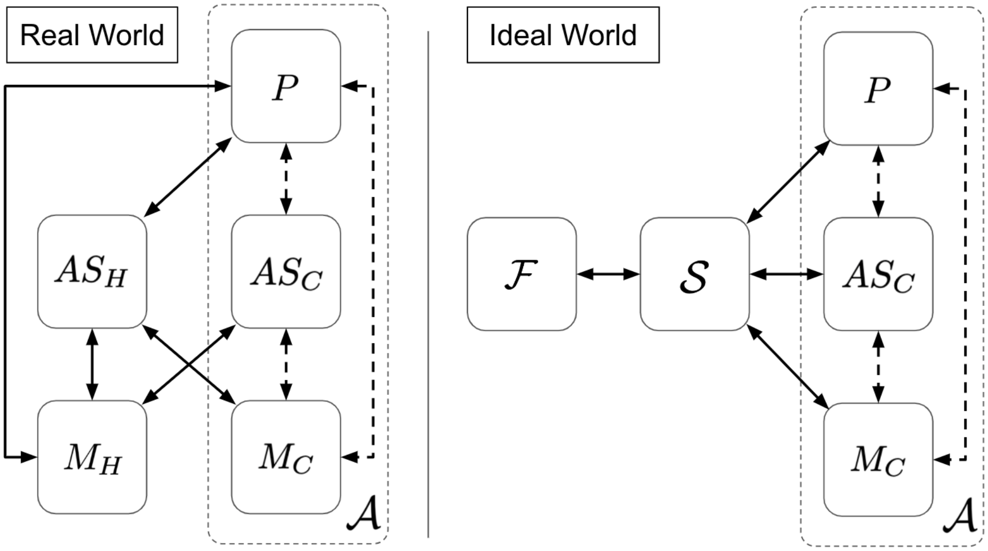


Figure 2.5: Overview of the real and ideal worlds.

2.7 Formal Privacy Guarantee

In this section, we define Nebula's privacy guarantee in the simulation paradigm [39] (note that Nebula's privacy could also be defined and proved in a computational indistinguishability framework depending on the guarantees of the underlying building blocks). We give an overview of the proof and connect it to the informal privacy guarantees in Section 2.4, then present the structure of the simulation and a proof sketch for indistinguishability.

The real and ideal world are depicted in Figure 2.5. In the real world, honest application servers and mules (AS_H and M_H, respectively) interact with a malicious provider, application servers, and mules (P, AS_C, and M_C) controlled by $\mathcal{A}$. In the ideal world, the honest parties are represented by an uncorruptible trusted party $\mathcal{F}$ called an ideal functionality; on every operation, $\mathcal{F}$ provides a simulator party $\mathcal{S}$ with a well-defined subset of information about the operation. This subset defines what information Nebula leaks to $\mathcal{A}$ and provides a clear definition of privacy in our setting. $\mathcal{S}$ then interacts with $\mathcal{A}$ to complete the operation. $\mathcal{S}$, however, cannot perform the operation exactly as an honest party because it does not know all operation inputs, only the subset it was given by $\mathcal{F}$. $\mathcal{S}$ must simulate the operation such that what $\mathcal{A}$ sees is computationally indistinguishable from what it would see in the real world. The existence of $\mathcal{S}$ that can properly simulate the Nebula protocol would show that Nebula reveals no more to $\mathcal{A}$ than what $\mathcal{F}$ gives $\mathcal{S}$ on each operation.

2.7.1 Limitations

For simplicity and to enable us to focus on the core behavior of Nebula, we do not model all aspects of the complex real system, as follows. First, in practice, epochs overlap to avoid losing data from payload pickups near the end of the epoch: we model one epoch at a time in sequence. Second, we don't directly model network connections, including network information that might be leaked when honest parties connect to corrupted participants, because parties can hide this information using complementary systems like Tor Section 2.4.4. We also do not model messages dropping/becoming corrupted during transport. Third, we don't model timing or concurrent operations (although our implementation of Nebula in Section 2.9 handles concurrent redemption), and assume that we process one operation at a time. Fourth, we assume the sensors and payment mechanism are external, as they have little bearing on performing the protocol: sensors simply generate the inputs to the mules, which we model as appearing directly in X, and we assume the provider can charge for executing **purchase_tokens** operations. Fifth, application servers and mules may be malicious, but we restrict $\mathcal{A}$ to static compromise – honest parties do not later become malicious. Finally, our definition here captures the privacy that mules have in Nebula, and not the confidentiality of the payload from the sensors (which is taken care off independently via end-to-end encryption).

2.7.2 Real World

In the real world (Figure 2.5), honest mules M_H and application servers AS_H (the honest parties) interact directly with the mules, application servers, and provider (M_C, AS_C, and P, respectively) corrupted by $\mathcal{A}$ according to the Nebula protocol described in Section 2.5. The honest parties are given a sequence of operations $X = \{x_1, ..., x_n\}$ to execute, where each operation x_i is one of the following operations: **purchase_tokens**(id$_{AS}$, n_t), **deliver**(id$_M$, id$_{AS}$, (d, P_{hash}, σ_{hash})), **redeem**(id$_M$, T), and **new_epoch**(n_c, T_{dup}).

Valid sequences X have a specific structure, which models the flow of the Nebula protocol over the course of each epoch. Specifically, for any X of length n, X begins and ends with a new epoch – x_1 and x_n are always **new_epoch** – and normal operations and new epochs are interleaved in the rest of the sequence – every x_i for $1 < i < n$ is one of **purchase_tokens**, **deliver**, **redeem**, or **new_epoch**.

In each X, the mules and application servers that should execute the operation are identified by id$_M$ and id$_{AS}$, respectively. n_t is the number of payloads an application server should prepay for delivery (e.g. number of tokens an application server should purchase), n_c the number of complaints a mule can make in that epoch, T is a list of delivery tokens a particular mule attempts to redeem with the provider, d is the sensor data to deliver, and T_{dup} is a list of the mule's submitted tokens that saw duplicate submissions.

Operations that do not include an honest party are not included in X because $\mathcal{A}$ already has perfect visibility on the operation, and need not follow the Nebula protocol at all. However, operations that are only visible to honest parties *are* included in X, as this ensures that our proof demonstrates that interactions been honest parties do not affect participant privacy.

2.7.3 Ideal World

In the ideal world (Figure 2.5), we define an ideal functionality $\mathcal{F}$ that captures Nebula's privacy guarantees. $\mathcal{F}$ receives sequence X as input and executes the operations one at a time in the order provided. To perform each individual operation, $\mathcal{F}$ interacts with the simulator $\mathcal{S}$ as specified below, with communication from $\mathcal{F}$ to $\mathcal{S}$ underlined. $\mathcal{S}$, in turn, simulates the Nebula protocol toward $\mathcal{A}$ based on the subset of each call with which it was provided.

For **new_epoch**(n_c, T_{dup}), $\mathcal{F}$ marks the next epoch as active, initializes each honest mule's complaint counter to n_c, and sends **new_epoch**(n_c, T_{dup}) to $\mathcal{S}$.

For **complain**(c) from $\mathcal{S}$, $\mathcal{F}$ finds the mule corresponding to c, and if its complaint counter is greater than 0, sends $\top$ to $\mathcal{S}$ and decrements the counter. Else, $\mathcal{F}$ sends $\bot$.

For **purchase_tokens**(id$_{AS}$, n_t), $\mathcal{F}$ adds n_t to id$_{AS}$'s purchased token counter for this epoch, and sends **purchase_tokens**(id$_{AS}$, n$_t$) to $\mathcal{S}$.

For **deliver**(id$_M$, id$_{AS}$, (d, P_{hash}, σ_{hash})), there are 4 cases:

1. If both id$_M$ corresponds to an honest mule (id$_M$ in M_H) and id$_{AS}$ corresponds to an honest application server (id$_{AS}$ in AS_H), check for a duplicate payload hash $H(d)$ and: decrement id$_{AS}$'s purchased token counter if is greater than 0 and increment id$_M$'s delivered payload counter by 1 and add $H(d)$ to id$_{AS}$'s received payload list, else do nothing. $\mathcal{F}$ does not send anything to $\mathcal{S}$.

2. If id$_M$ corresponds to an honest mule but id$_{AS}$ corresponds to a malicious application server (id$_{AS}$ in AS_C): increment id$_M$'s delivered payload counter by 1 and send $\mathcal{S}$ the operation **deliver**(_, id$_{AS}$, (d, P_{hash}, σ_{hash})), where the "_" notation indicates that the first argument (in this case, id$_M$) is not sent to the simulator.

3. If id$_M$ corresponds to a malicious mule (id$_M$ in M_C) but id$_{AS}$ corresponds to an honest application server, check for a duplicate payload hash $H(d)$ and: decrement id$_{AS}$'s purchased token counter if greater than 0, add $H(d)$ to id$_{AS}$'s received payload list, and send $\mathcal{S}$ $\top$ to denote success, else send $\mathcal{S}$ $\bot$ to denote failure.

4. If id$_M$ corresponds to a malicious mule and id$_{AS}$ is honest but d is missing (only P_{hash} is provided by $\mathcal{S}$), check for a duplicate payload hash $H(d)$ and id$_{AS}$'s purchased token counter is greater than 0 and send $\mathcal{S}$ $\top$ to denote no duplicate or $\bot$ otherwise.

For **redeem**(id_M, T), $\mathcal{F}$ calculates the redemption size n as $\min(|T|$, id_M delivered payload counter), decrements id_M's delivered payload counter by n, and sends $\mathcal{S}$ the operation **redeem**(id_M, n).

Relation to informal properties in Section 2.4.2

$\mathcal{F}$ sends a set of information to $\mathcal{S}$ above that correspond to the informal privacy properties listed in the main text. Specifically, $\mathcal{S}$ sees:

1. Each **purchase_tokens** operation, allowing $\mathcal{A}$ to see how many payload deliveries each AS prepurchases in an epoch,
2. The count of tokens in each **redeem** operation, which leaks to $\mathcal{A}$ how many tokens overall a mule redeems in each epoch, and
3. Each **complain** operation without mule identifiers, leaking to $\mathcal{A}$ each complaint registered against each AS. In addition, $\mathcal{A}$ sees when new epochs begin, duplicate token uploads, the payloads delivered to malicious ASes, and if uploads succeed from malicious mules.

2.7.4 Proof

We prove Theorem 1 by constructing a simulator $\mathcal{S}$ for Nebula that, given the information provided by $\mathcal{F}$ on each operation, interacts with $\mathcal{A}$ so that it cannot distinguish the real world from the ideal world. For readability, we say "$\mathcal{A}$ cannot distinguish the real world from the ideal world" in this section to mean the cryptographic equivalence stated in Definition 1 applied to π_{Nebula}:

$$\exists \mathcal{S} \forall \mathcal{A} \forall X \{\mathrm{REAL}_{\pi_{Nebula},\mathcal{A}(z),X}(1^\lambda)\}_\lambda \overset{c}{\equiv} \{\mathrm{IDEAL}_{\mathcal{S},\mathcal{A}(z),X}(1^\lambda)\}_\lambda$$

.

$\mathcal{S}$ interacts with $\mathcal{A}$ on an operation-by-operation basis, just as honest mules or application servers would in the real world. $\mathcal{F}$ is designed to avoid giving $\mathcal{S}$ mule identifiers when delivering payloads (or complaining about payload delivery), so $\mathcal{S}$'s main role is to act as (1) a mule performing all honest backhaul operations in the system and (2) an honest application server managing malicious mules. Nebula is designed such that the malicious provider, application servers, and mules cannot distinguish this case from individual mules each providing a subset of the backhaul service.

We use a unlinkable token (UT) protocol as a building block in Nebula. Let $\mathcal{S}_{\mathrm{UT}}$ be a simulator for a UT protocol with the *unlinkability* and *one-more-token* security guarantees as described in Section 2.2.3. We instantiate $\mathcal{S}_{\mathrm{UT}}$ with the PrivacyPass protocol [64, 129]; given that Davidson et al. [64] do not formally provide an existing simulator for PrivacyPass, and it is out-of-scope here to create it, we assume a simulator with an interface as defined below.

Definition 2 ((Informal) Simulator for Unlinkable Tokens)**.** *$\mathcal{S}_{UT}$ has the following interface:*

- *$\mathcal{S}_{UT}.\mathsf{KeyGen}()$ simulates generating a new keypair and publishing its public parameters to $\mathcal{A}$*
- *$\mathcal{S}_{UT}.\mathsf{Sign}_k(t')$ simulates signing a given token value t' under some key pair k*
- *$\mathcal{S}_{UT}.\mathsf{Redeem}_k(t)$ simulates redeeming a signed token t under some key pair k*

Description of $\mathcal{S}$

In the setup phase, $\mathcal{A}$ simulates $\mathcal{S}_{\mathrm{UT}}.\mathsf{KeyGen}()$ twice to generate epoch delivery and complaint keys k_c and k_d, and reveals the public params to $\mathcal{S}$. $\mathcal{A}$ generates a separate symmetric key k_i for each id_{AS} in AS_H and sends to $\mathcal{S}$, and $\mathcal{S}$ generates public-private keypairs for each AS in AS_H and a keypair for every sensor s controlled by an honest AS; $\mathcal{A}$ gives the public keypairs for each AS in AS_C to $\mathcal{S}$.

For **start_epoch**, $\mathcal{S}$ simulates $\mathcal{S}_{\mathrm{UT}}.\mathsf{Sign}_{k_c}(t)$ for every t in a random vector of n_c blinded tokens and appends each token list to the epoch master Mule complaint token list. $\mathcal{S}$ adds the complaint records the tokens in T_{dup} to its complain list. Then, for every record c in the complaint list, $\mathcal{S}$ sends **complain**(c) to $\mathcal{F}$. If $\mathcal{F}$ returns $\top$, $\mathcal{S}$ pops a token off of last epoch's master Mule complaint token list and sends it along with c to $\mathcal{A}$. If c contains P_{token}, $\mathcal{S}$ sends $(P_{token}, \sigma_{token})$, else d to $\mathcal{A}$. If $\mathcal{A}$ does not reply with $\bot$, $\mathcal{S}$ simulates $\mathcal{S}_{\mathrm{UT}}.\mathsf{Sign}_{k_d}(t)$ on a new token t with $\mathcal{A}$ and appends it to the new epoch Mule delivery token list.

For **purchase_tokens**, $\mathcal{S}$ simulates $\mathcal{S}_{\mathrm{UT}}.\mathsf{Sign}_{k_d}(t)$ for every t in a random vector of n_t blinded tokens. $\mathcal{S}$ appends the tokens to the epoch master AS token list.

For **deliver**, there are two cases. First, if $\mathcal{S}$ receives the operation from $\mathcal{F}$ to deliver a payload to a malicious id_{AS}: $\mathcal{S}$ sends the AS $(P_{hash}, \sigma_{hash})$, verifies the resulting σ_{pre} using the right sk_{as}, sends d to $\mathcal{A}$, verifies the resulting σ_{token} and saves the output token in the epoch master Mule delivery token list and a complaint record. Second, if $\mathcal{S}$ receives the operation from $\mathcal{A}$ to deliver a payload to an honest id_{AS}: $\mathcal{S}$ checks σ_{hash} using the correct sensor secret key and sends P_{hash} to $\mathcal{F}$ to check for duplicate. On success, $\mathcal{S}$ picks a random r, pops a token t from the master AS token list, and uses its keys to generate P_{pre} and sign σ_{pre} for $\mathcal{A}$. When $\mathcal{A}$ sends d, $\mathcal{S}$ checks that the payload hash matches and generates and sends P_{token} and σ_{token}.

For **redeem**, $\mathcal{S}$ pops n tokens from the epoch's master Mule delivery token list and sends the list to $\mathcal{A}$. On response with the invalid token list, $\mathcal{S}$ adds the matching complaint records to its complaint list.

Proof Sketch for Indistinguishability

In this subsection, we sketch a proof for why $\mathcal{A}$ cannot distinguish the real world from the ideal world for each operation executed by the simulator $\mathcal{S}$.

Proof Sketch. For **start_epoch**, $\mathcal{S}$ uses $\mathcal{S}_{\mathrm{UT}}$ to simulate the complaint token generation. Since this simulation is, by definition, indistinguishable, and $\mathcal{S}$ knows to sign $|M_H|$ vectors of n_c elements at each epoch, $\mathcal{A}$ sees indistinguishable signing requests from the real world. This follows from the computational indistinguishability of the random blinded tokens $\mathcal{S}$ submits to $\mathcal{A}$. When handling complaints, $\mathcal{S}$ interacts with $\mathcal{F}$ to ensure it submits at most n'_c complaints per mule. Since $\mathcal{S}$ only submits complaints from its honest mules about malicious ASes, it can retain the necessary complaint record c from a previous **deliver** operation and perfectly replay the messages to $\mathcal{A}$ to submit the complaint as in the real world. Finally, $\mathcal{S}_{\mathrm{UT}}$ indistinguishably simulates the signing of a new token.

For **purchase_tokens**, $\mathcal{S}$ again uses $\mathcal{S}_{\mathrm{UT}}$ to simulate the delivery token generation. Since this simulation is indistinguishable, and $\mathcal{S}$ knows how many tokens to sign (n_t), it can generate its own random token values, and $\mathcal{A}$ will see an indistinguishable signing request from the real world, following from the computationally indistinguishable random blinded tokens $\mathcal{S}$ submits to $\mathcal{A}$.

For **deliver**, in the first case, $\mathcal{S}$ is given the payload to deliver (d, P_{hash}, σ_{hash}) and so can exactly replicate the payloads to $\mathcal{A}$ for the delivery protocol as if in the real world. In the second case, $\mathcal{S}$ uses $\mathcal{F}$ to check for duplicates, matching exactly the real-world abort behavior in case of duplicate uploads. $\mathcal{S}$ picks a random token indistinguishable from what the honest AS would choose in the real world and takes the first available delivery token to return to $\mathcal{A}$. These signed tokens, following from the unlinkable token scheme's *unlinkability* guarantee [64], are computationally indistinguishable to $\mathcal{A}$, so it doesn't matter which particular token $\mathcal{S}$ chooses from the epoch's master AS token list; thus, P_{token} and σ_{token} are indistinguishable to $\mathcal{A}$.

Finally, for **redeem**, $\mathcal{S}$ knows exactly how many tokens to submit, and, similarly to the case above, the choice of tokens does not matter (any set of n tokens in the epoch's master Mule delivery token list is valid), as each signed token is computationally indistinguishable in $\mathcal{A}$'s view. Thus, $\mathcal{A}$ sees an indistinguishable token list from $\mathcal{S}$. □

2.8 Analytical Model for Energy and Memory Consumption

By nature of intermittent interactions between sensors and mules, the memory and energy consumption of sensors and mules in such a system may vary dramatically across applications and locations. In an effort to characterize the main factors that affect the performance of this system and to make informed sensor design choices, we develop two numerical models – one for a deployed sensor and one for a mule – and explore how memory and energy consumption may vary depending on the deployment. In Section 2.10.3, we use these models to estimate the lifetime of a sensor and consider how many payloads a smartphone mule can upload given energy and memory constraints.

2.8.1 Sensor Model

Low-power sensors provisioned with BLE are often designed to run for years while periodically advertising [111]. In this section, we explore sensor performance in Nebula.

Sensor Configuration

We assume that the sensor runs off of a battery of size $size_{battery}$ and employs some sensing workload $wkld$ (i.e. periodic sensing, event-driven sensing, or long-running sensing). The sensor accumulates data and desires to upload the data at a frequency of f_{upload} (e.g. once a day). When the sensor desires to upload data, it starts broadcasting BLE advertisements at a rate of f_{adv} until it successfully forms a connection with a nearby mule. Once the connection is made, the sensor stops broadcasting BLE advertisements.

External Conditions

The largest source of uncertainty for the sensor is in how long it must broadcast BLE advertisements before forming a connection. The duration of this time can be parameterized by mule arrival frequency $f_{arrival}$, how long each mule stays in the vicinity of the sensor τ_{mule}, how frequently each mule listens for advertisements f_{listen}, and the probability that any given attempt at connecting with a mule transfers all the data $p_{success}$. These values vary greatly depending on the physical location, time of day, or time of year. While we explore some of the range that these values can take on in Figure 2.8 and Section 2.10.3, for the sake of this model, we make the simplifying assumption that these values are constant and the interactions are uniformly distributed.

Sensor Energy Consumption and Lifetime

The average power consumption of the sensor can be broken into two parts: workload power from sensing, and network power from BLE advertisements and the wireless communication Nebula requires. We denote the workload power P_{wkld} and calculate the network power. At any moment in time, we assume there are $f_{arrival} * \tau_{mule}$ number of mules near the sensor. Since each mule listens at a frequency of f_{listen}, the period of time between is

$$1/(f_{arrival} * \tau_{mule} * f_{listen})$$

If it only takes one listening event to successfully connect to an advertising sensor, a sensor has to advertise for

$$1/(2 * f_{arrival} * \tau_{mule} * f_{listen})$$

amount of time on expectation before connecting to a mule. A sensor must connect to an average of $1/p_{success}$ mules before successfully handing off a data payload. Thus, the average networking power is

$$P_{ntwk} = \left(\frac{f_{adv} * E_{adv}}{2 * f_{arrival} * \tau_{mule} * f_{listen}} + E_{conn} \right) * \frac{f_{upload}}{p_{success}}$$

where E_{adv} is the energy consumed per BLE advertisement and E_{conn} is the energy consumed while uploading data to the mule. This gives us a sensor lifetime of

$$T_{sensor} = \frac{size_{battery}}{P_{wkld} + P_{ntwk}}$$

2.8.2 Mule Model

We estimate the energy and memory consumption on a mule for participating in Nebula.

Mule Configuration

We assume that each mule listens for BLE advertisements at a rate of f_{listen} (e.g. 0.1 Hz), uploads the collected data payloads at a rate of f_{upload}, and redeems the tokens at a rate of f_{redeem}. We assume that the mule does this perpetually and uniformly across time.

External Conditions

The largest uncertainty for the mule is how many sensors it will encounter and how much data each sensor will try to backhaul. We parameterize these by expected sensor interaction frequency $f_{interact}$ and expected sensor payload size $size_{payload}$, and assume that these values are constant and the interactions are uniformly distributed. In reality, interactions with sensors may be bursty and payload size variable, however assuming constant values allows us to compute a reasonable estimate.

Mule Energy and Memory Consumption over Time

A mule is expected to periodically do three distinct tasks: collect data from sensors, upload data to servers, and redeem tokens with the platform provider. We consider the *energy consumption* of each of these tasks separately.

$$P_{collect} = f_{listen} * E_{listen} + f_{interact} * E_{interact}$$

where E_{listen} is the energy consumed each time the mule listens for BLE advertisements, and $E_{interact}$ is the energy consumed while receiving data from a sensor. This happens perpetually throughout the day, and so consumes battery at a rate of $\frac{P_{collect}}{size_{battery}}$.

$$P_{upload} = f_{interact} * E_{upload}$$

where E_{upload} is the energy consumed each time the mule uploads a single payload. If data is continually uploaded throughout the day, this would consume battery at a rate of $\frac{P_{upload}}{size_{battery}}$. If the upload is batched instead, each time the mule uploads consumes

$$E^{batched}_{upload} = \frac{P_{upload}}{f_{upload}} = \frac{f_{interact}}{f_{upload}} * E_{upload}$$

consuming $\frac{size_{battery}}{E^{batched}_{upload}}$ of the battery. As the upload schedule is flexible, the uploads could be delayed and batched to correspond with mule recharging patterns.

$$P_{redeem} = f_{interact} * E_{redeem}$$

where E_{redeem} is the energy consumed each time the mule redeems a single token with the platform provider. If tokens are redeemed perpetually throughout the day, this would consume battery at a rate of $\frac{P_{redeem}}{size_{battery}}$. If the redemption is batched instead, each time the mule redeems consumes

$$E^{batched}_{redeem} = \frac{P_{redeem}}{f_{redeem}} = \frac{f_{interact}}{f_{redeem}} * E_{redeem}$$

amount of energy, amounting to $\frac{size_{battery}}{E^{batched}_{redeem}}$ amount of the battery. Since the redemption frequency should be very low in order to obfuscate mule timing information from the platform provider, this would preferably be delayed and also scheduled to correspond to mule recharging patterns.

Mule *memory consumption* consists of data payloads picked up from sensors that haven't yet been uploaded and tokens received from the servers that haven't yet been redeemed. We assume that the mule deletes sensor payloads after uploading them to the servers, and tokens after redeeming them with the platform provider. Suppose that a mule starts with zero memory consumption at time $t = 0$. The expected memory consumption of a mule over time is

$$M_{mule}(t) = f_{interact} * \left[size_{payload} * \left(t \mod \frac{1}{f_{upload}} \right) + size_{token} * \left(t \mod \frac{1}{f_{redeem}} \right) \right]$$

where $size_{token}$ is the size of each token. Then the maximum memory consumption is

$$f_{interact} * \left(\frac{size_{payload}}{f_{upload}} + \frac{size_{token}}{f_{redeem}} \right)$$

2.9 Implementation

We describe our Nebula prototype that we implemented to test the overall performance of the backhaul system.

Hardware and Setup

We implement each sensor on an nRF52840 development board, which is provisioned with a 256-bit AES key shared with the sensor's application server, public-private `secp256r1` key pair, and a matching ECDSA certificate signed by the application server. Our prototype mule is a ESP32-WROOM development board. Both the platform provider and the application server are implemented as GCP instances.

Token Pre-Purchase

In order to allow application servers and the platform provider to generate signed unlinkable tokens, we wrap a Rust-based implementation of the PrivacyPass protocol [11, 64] (Section 2.2.3) with Python bindings and instruct application servers to pre-purchase tokens in 100-token chunks when they have exhausted all of their previously-purchased tokens.

Payload Pickup

Our prototype sensors upload data packets over a BLE connection to mules they encounter over a DTLS secure session. We implemented DTLS sessions over BLE because session establishment requires the mule to verify the correctness of the sensor's certificate without internet connectivity, and results in a secure channel. In addition DTLS sessions can be established without requiring the user to input a code or push a button, as is the case in BLE Secure Connections [27].

The sensor advertises that it has data for pickup while the mule board scans for sensors advertising a Nebula BLE service. Once connected, the Nordic-based sensor acts as a BLE *peripheral* and the ESP32 mule acts as a BLE *central*. We structure the Nebula BLE service with two characteristics (e.g. writable and readable attributes). We take advantage of the two characteristics to create read and write "sockets" for `MbedTLS`. As BLE uses notifications to indicate characteristic changes, we carefully manage state indicating whether each party is listening, receiving, or writing in order to synchronize the parties. During the DTLS handshake, the mule verifies that the sensor owns a certificate that has been signed by the Nebula Certificate Authority (CA) hierarchy. Specifically, a successful DTLS session setup ensures that the sensor certificate is signed by its application server, acting as an intermediary CA for the platform provider, who is the root CA. Once verified, the sensor is able to send its payload to the mule. The payload itself is end-to-end encrypted with AES-GCM, using 12-byte nonces and 16-byte authentication tags.

Payload Delivery

Sensor payloads are stored in the ESP32-based mule's memory until the mule comes within WiFi range of a known network. At that point, a TLS session is initialized with the correct application server and the received payloads are uploaded. The application server attempts to perform AES-GCM decryption on the payload using the key derived from the payload's sensor ID and its own secret key (Section 2.3.2) and returns a signed PrivacyPass token if successful.

Token Redemption

Our platform provider is conveniently packaged as a container, making deployment easy. We run the provider on a 128-core, general-purpose GCP virtual machine (`n2-standard-128`). The system utilizes two persistent databases: `token_db`, which checks for duplicate entries in the current epoch's submitted tokens, and `mule_payment_db`, which records mule upload totals. Our front end HTTPS server is hosted using *uvicorn*, a popular open-source solution, ensuring connections for mules and application servers. We configure mules to upload in batches of 700 tokens per request. Upon mule token redemption, we first verify the signature and submit the tokens to `token_db`, an in-memory hash-table. To optimize performance, we divide `token_db` into 16 shards. Backend workers then update `mule_db`, a SQLite database that tracks each mule's owed amount for out-of-band payments. To keep costs low, in our prototype, we hosted `token_db` and `mule_payment_db` on the same node as our HTTPS provider, but can be easily replaced with a hosted in-memory and on-disk databases, to facilitate horizontal scaling.

2.10 Evaluation

We built and evaluated Nebula to demonstrate the feasibility of developing a privacy-preserving, large-scale data backhaul system and answer four key questions:

1. What are the energy costs between a mule and a sensor? What is the overhead of implementing security?
2. What energy costs does a mule incur in delivering sensor data and redeeming tokens?
3. What is the expected energy and memory consumption of running the Nebula service?
4. How well can Nebula's cloud-based provider perform?

2.10.1 Payload Pickup

Sensor to mule data uploads are by nature transient. Therefore, we measured the amount of data that can be transferred based on how long a mule is near (within BLE range) of

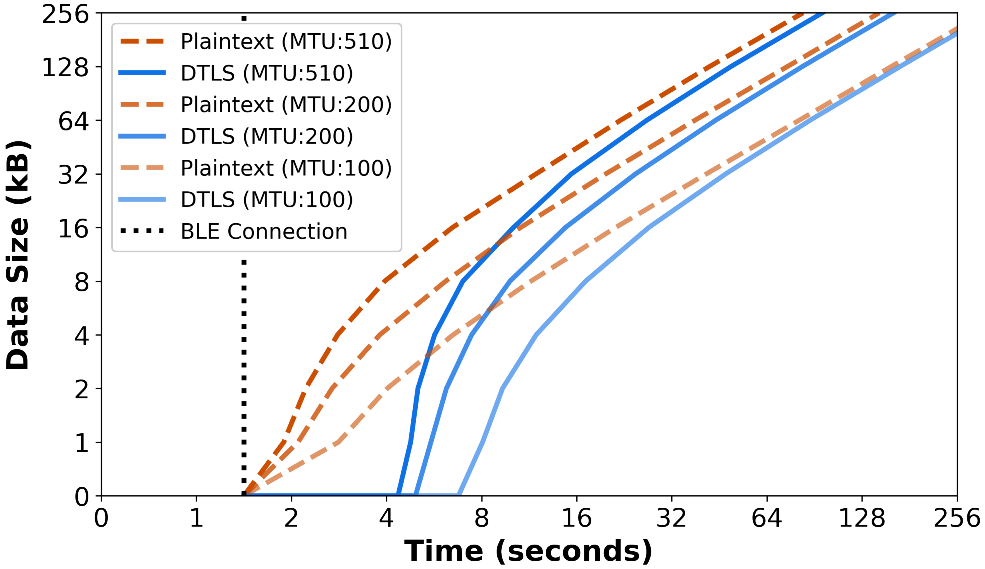


Figure 2.6: The amount of data that can be transferred based on how long a mule is in connection range with different BLE MTU sizes. Handshake time is amortized as mules spend longer in proximity to sensors.

a sensor Figure 2.6. Since transfer rate is highly dependant on BLE implementations, we picked three different BLE maximum transmit units (MTUs) to measure with. BLE has to connect the devices before data can be transferred, this overhead is shown in the black dotted line in Figure 2.6. We conservatively measured the BLE connection time by including the task start and subscription to BLE characteristics. Encrypted DTLS has both the connection startup time (approximately 1.5 seconds) and TLS handshake time (approximately 2.5 to 5 seconds depending on MTU size) as overhead. However, as more data is transferred the handshake overhead of a few seconds is amortized. We expect that the BLE link could be further optimized, which would improve both the handshake time and data upload time.

Additionally, energy usage on the sensor device is important for prolonged battery use. We measured current draw on the nRF52840 while advertising to be 5.7 mA and current while transmitting to be 12.2 mA. The board supply power was 3.3 V, therefore the advertising and transmitting power is 18.81 mW and 40.26 mW, respectively. Figure 2.7 shows the breakdown of energy spent in different phases (connection, handshake, and transmit). All current measurements were taken with a Keithley SMU 2401 source meter.

2.10.2 Payload Delivery

In our architecture, the mule has a larger battery capacity compared with the sensor. However, the overall cost to upload data is still important. We measured the time taken for our ESP32 mule to set up an HTTPS connection with and send HTTPS requests to our application server and our platform provider. We found that the transmission time is largely dominated by the time it takes to set up a connection and the round-trip time of an HTTPS request, more so than the size of each request. In particular, the handshake time is around 2.4 ± 0.3 seconds and each subsequent round trip request took 0.8 ± 0.1 seconds. However, once the size of a request exceeded around 10 kB, we started seeing latency increasing with packet size, which we believe is due to queuing delays from filling up the ESP32's WiFi transmission buffer. We also measure the current draw on the ESP32. We found on the ESP32 development board, as expected, that the WiFi radio draws more power (454 mW) on average compared with the BLE radio (225 mW). Thus, uploading a single packet requires 3.2 seconds and 1.45 J.

Smartphone energy usage

In our evaluation, we implement the mule on a development board in order to isolate the execution of Nebula's protocol from other confounding factors in phone operating systems. In particular, the difficulty of evaluating long-running behavior when backgrounded and issues with OS Bluetooth networking stacks, without OS-level support, require significant engineering "hacks" as described in [126] that are beyond the scope of this work. However, the WiFi and BLE chipsets in modern phones are significantly more power-optimized than that on the ESP32 that we evaluate. To give a comparison point between our implementation and expected energy usage in a smartphone deployment, we profiled WiFi/BLE power usage on a Pixel 7 Pro running Android 13 using the integrated On Device Power Monitor [10]. While uploading data over WiFi to our application server, the smartphone draws 116 mW on average, 25% of the ESP draw. Sensor data transfer over BLE required 50 mW on average, 22% of the ESP required power. As a result, the energy usage analysis in our evaluation represents a conservative overestimate of what would likely occur in a widespread smartphone deployment.

2.10.3 Energy & Memory Usage Estimates

We now apply our energy consumption data from Section 2.10.1 and Section 2.10.2 to an estimate of sensor and mule behavior, using our analytical model from Section 2.8.

Sensors

We consider sensors deployed for the bike counting example presented in Section 2.1. Suppose each sensor counts the number of bicyclists that pass by it every hour and draws around $P_{wkld} = 50\ \mu\text{W}$ doing so [111]. Each sensor desires to upload its data once a week, so $f_{upload} = \frac{1}{604800}$ Hz. Each payload consists of $24 * 7 = 168$ samples, each containing a timestamp and a bicyclist count, yielding payloads on the order of 1 kB.

Next we check if, given the deployed location, nearby mules are likely to stay in the sensor's vicinity long enough to pick up the data packets. We reference Figure 2.6, which

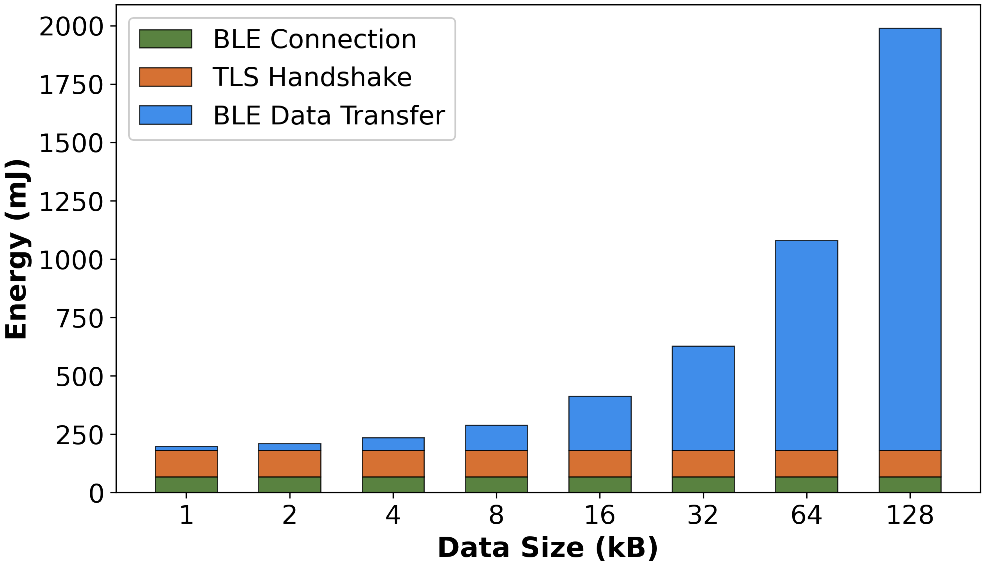


Figure 2.7: The energy used by the nRF52840 for different payload sizes. There is a set amount of energy spent on setup, so the larger the payload the more the energy is amortized.

shows that it would take about 5 seconds to connect to a mule and transfer 1 kB of data. Figure 2.8 suggests that areas with foot traffic (e.g. on campus and in a park) will contain many mules that are in BLE range of a sensor for at least 5 seconds, so we should be able to successfully upload data to a mule. However, to provide a conservative estimate and account for interactions which are shorter than 5 seconds let's suppose that $p_{success} = 0.5$. From these clusters of mule interactions, the expected duration is around $\tau_{mule} = 10$ seconds with a mule arrival rate of $f_{arrival} = 0.01$ Hz for sparser areas, again estimated from Figure 2.8.

Using our BLE advertising power $f_{adv} * E_{adv} = 18.81$ mW and supposing mules listen at a rate of $f_{listen} = 0.1$ Hz, we get

$$P_{ntwk} = \left(\frac{18.81 \times 10^{-3}}{2 * 0.01 * 10 * 0.1} + 0.0169 \right) * \frac{1/604800}{0.5} = 3.2\mu\text{W}$$

which is an order of magnitude lower than the workload power. Given a coin cell battery of $size_{battery} = 2200$ J [201], the sensor would last $T_{sensor} = \frac{2200}{53.2 \times 10^{-6}}$ seconds ≈ 1.3 years. Given two alkaline AA batteries with a total capacity of $size_{battery} = 27000$ J [156], the sensor would last $T_{sensor} = \frac{27000}{53.2 \times 10^{-6}}$ seconds ≈ 16 years.

This is only one simplified example, and meant to demonstrate that there's nothing fundamentally infeasible about deploying an application on Nebula. Different applications would have their own challenges and opportunities. For instance, some sensors may want to advertise perpetually (e.g. for global asset tracking), while others would be able to coordinate backhaul times with collaborating mules (e.g. in smart farming). This section

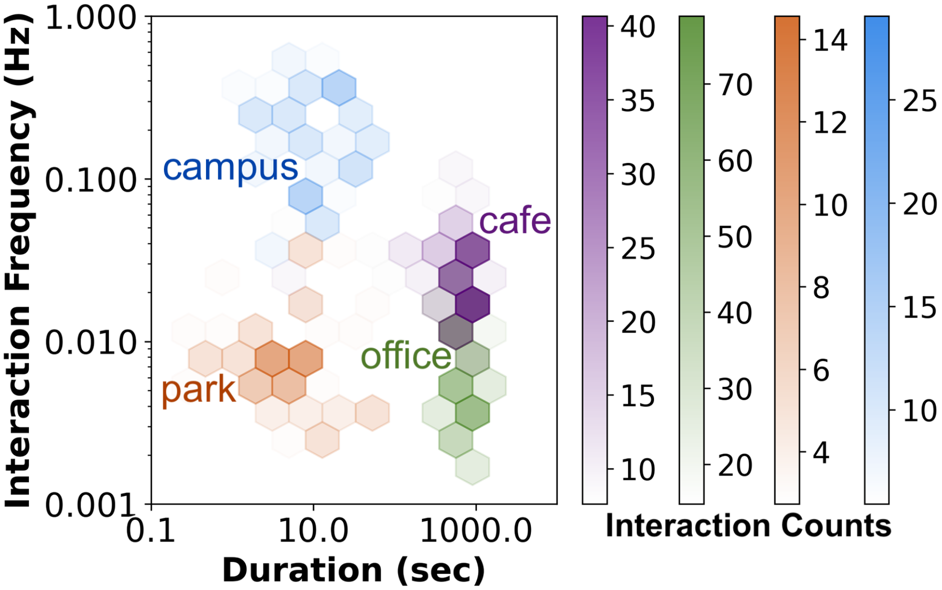


Figure 2.8: Interaction frequency and interaction duration varies depending on the location. We collect BLE advertisements in four representative locations and construct interactions from repeated MAC addresses. For this data collection, we anonymized all MACs with an irreversible hash and received an IRB exemption from our institution review board.

is only a starting point from which a developer can consider how their own sensors would interact with the Nebula system.

Mules

Consider a smartphone user who is interested in participating in Nebula. Suppose they are willing to give up 5% of their battery throughout the course of a day and can spare 5 GB of storage on their phone, as they charge their phone overnight and have a 128 GB device. They usually keep their Bluetooth enabled, so when they join Nebula they do not incur the marginal cost of BLE listening. What is the maximum number of sensors they can interact with per day? In other words, what is the largest $f_{interact}$ possible without violating either their power or memory constraints?

Suppose the smartphone has a battery capacity of 4000 mAh ≈ 54720 J [17, 66]. Then the maximum power draw across a day would be $P_{max} = \frac{0.05*54720}{86400} = 31.67$ mW. Thus,

$$31.67\ mW \geq P_{collect} + P_{upload} = f_{interact} * (E_{interact} + E_{upload})$$

so

$$f_{interact} \leq \frac{31.67 \times 10^{-3}}{E_{interact} + E_{upload}} = \frac{31.67 \times 10^{-3}}{0.0945 + 1.45 * 2} = 10.6\text{mHz}$$

so power constrains each mule to 915 sensor payloads per day.

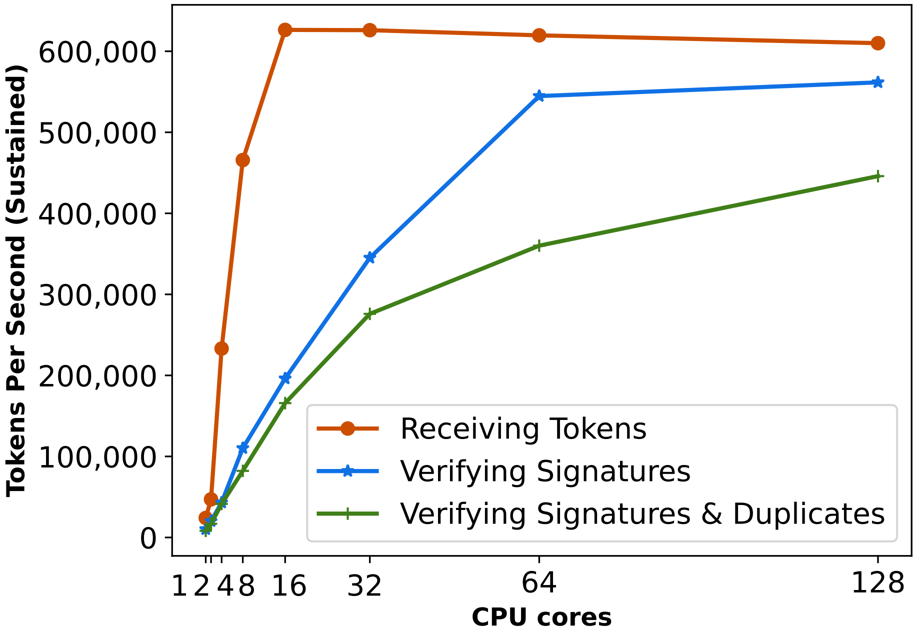


Figure 2.9: Number of tokens redeemed per second. With 128 cores, we can verify 445,900 tokens, including filtering for duplicates, per second, or over 250 million tokens per dollar.

Now let's consider the memory constraint. Suppose that each payload is around 1 kB of data. Then the spare 5 GB can fit 5 million payloads. If the mule stores all these payloads and uploads them once a month, they can pick up $5,000,000/30 = 1666,666$ 1 kB payloads every day without running out of storage. On the other hand, if they upload each packet immediately after receiving it, they do not incur any memory costs for storing sensor data and can hold up to 50 million tokens, which are each 100 bytes long, before redeeming them at the end of the month. This would loosen their memory constraint to upper bound at $1,666,666$ arbitrarily-sized payloads every day.

It is clear that the energy constraint is far more restrictive than the memory constraint. Over the course of a month, this mule would upload at most $915 * 30 = 27450$ data payloads due to energy constraints, resulting in about 3 MB of storage used on the phone itself (assuming relatively frequent payload uploads). During the token redemption process, if the mule redeemed the tokens using a single HTTPS connection and uploading requests containing 100 tokens, it would take $2.4 + 275 * 0.8 = 223$ seconds $= 3.7$ minutes to redeem all the tokens. This redemption would consume $223 * 0.454 = 101$ J $\approx 0.18\%$ of the phone's battery.

2.10.4 Redemption

We discuss the performance of the Nebula platform provider when processing incoming tokens. When a mule redeems a set of tokens, the platform provider is responsible for verifying that the tokens are correctly signed and ensuring there no duplicate tokens were submitted, in order to determine how much to pay each mule. Our implementation showcases an efficient and cost-effective system, processing more than 445,000 tokens per second using a single node.

In Figure 2.9, we plot how the system scales with an increasing number of cores. We measure token processing rates under three different scenarios: (1) simply *receiving tokens* as fast at the HTTPS server can accept connections, (2) *verifying the signature* for each token, and (3) both verifying token signatures and ensuring that *duplicate* tokens are not redeemed twice. As CPU cores increase, we see substantial improvements in all three rates. Starting at around 16 cores, the rate at which our uvicorn HTTPS server can accept new connections becomes the the limiting factor. On a single core, the platform can receive $24,080$ tokens/sec, verify the blind signatures at $10,150$ tokens/sec, and check $8,190$ tokens/sec for duplicates. This scales up to 16 cores, to $626,360$, $196,140$, $165,830$ tokens/sec, respectively. With a full 128 cores, Nebula can fully process $445,900$ tokens per second. The GCP `n2-standard-128` instance on which we implemented our provider costs \$6.2 an hour (\$1.5 with Spot pricing). At $445,900$ tokens per second, Nebula can process 258.90 million tokens per dollar (1.06 billion tokens per dollar with Spot). Finally, mules are able to efficiently lodge complaints with the platform provider in response to misbehavior; our implementation requires 178 ms to validate a complaint based on an invalid token and 247 ms to validate a complaint involving an incomplete delivery.

Our implementation of the Nebula platform provider demonstrates high performance at low cost, while verifying token signatures and checking for duplicates. This allows for a system design that enables privacy-focused data backhaul that can accommodate a large number of participants and sensors with minimal constraints.

2.11 Opportunities for Future Work

In re-envisioning how an opportunistic network can operate to preserve user privacy, Nebula focuses on defining and evaluating the core architecture and protocols. However, a significant amount of future work is needed to bring a Nebula-type system into broad circulation, where mobile phones owned by random passers-by can interact with sensor data needing a “lift” to their destination.

Deployment At Scale

Section 2.10 evaluates the individual pieces of a backhaul system (e.g. data pickup and delivery, token redemption) in a series of microbenchmarks. However, to truly determine the effectiveness of Nebula’s design, a larger, wide-scale deployment is necessary. Future studies can incentivize study participants to carry a Nebula app on their phone, interacting with pre-placed sensors and measuring energy consumption and connectivity. In the end, a larger existing backhaul network like Tile, FindMy, or Sidewalk should integrate Nebula's ideas into their protocols to evaluate their effectiveness on a larger geographic scale across different urban areas.

Bidirectional Data Transfer

Currently, our design only uploads data from sensor to the cloud, without the opportunity for application servers to send new data to their sensors. These could include parameter changes, new ML model versions, or whole updated binary packages for a Device Firmware Update (DFU). Future work could study how to download important data to sensors through passing mobile platforms, in a way that would alleviate user privacy concerns. This is an important avenue for future work, as it must address a seemingly unavoidable contradiction: we would like the network to know when a mule is close enough to a sensor to proxy an update payload, yet we also seek to hide the mule's location from that same network.

Policies for Misbehavior Prevention

In Section 2.5, we describe a way for the Platform Provider to *collect* complaints of misbehavior – additional research can certainly be done on how to *process* those complaints to reduce overall misuse of system resources. What are the best indicators that an application server is deliberately misbehaving, as opposed to innocently executing the protocol but nevertheless being accused of misbehavior by a large group of malicious mules? In our complaint protocol, we remove any financial incentive for complaining – mules will get no extra tokens in addition to the token they are owed, and data transmission to the application server must occur to get a new token – but mules incentivized by other reasons may still seek to abuse the system. Future work in policy-setting for balancing the concerns of applications and mule participants is a must as this protocol goes into wider use.

Power and Wireless Protocols

Finally, there are many open opportunities to improve the speed, power efficiency, and functionality of Nebula's wireless component. We are able to backhaul several kilobytes of data in approximately 5-10 seconds, but future work in speeding DTLS connection time (or a faster, more data-efficient secure connection protocol) could enable faster vehicles like buses, bikes, and cars to pick up data in shorter time windows. Similarly, the increasing availability of ultra-wideband (UWB) radios in mobile phones could help both sensors and mules localize each other with high fidelity – knowledge which could allow them to focus on making connections with higher success rate and ignoring devices on the periphery of their range.

2.12 Summary

How can we massively extend network connectivity to the edge while protecting the privacy of the participants from a centralized platform? In this chapter, we described and evaluated Nebula, a decentralized architecture for privacy-preserving, general-purpose data backhaul. We experimentally showed that our system incurs low energy and computational overheads, and developed an analytical model to estimate real-world performance. Using Nebula, a smartphone anywhere in the world could backhaul almost a thousand data payloads a day consuming only 5% of its battery and 3MB of storage, without revealing its location to a central network server. For embedded sensing applications, our architecture vastly expands the scope of potential deployments while reducing the deployment cost.

Nebula demonstrates the power of protocol decentralization across many independent mules, restricting the relatively more costly token generation process to cloud-based parties with easy access to large pools of computational resources. At the same time, we leverage each individual mule's ability to verify payload pickup, delivery, and managing complaint submission to provide metadata-hiding privacy from a centralized provider.

Up to this point, we have not addressed the privacy implications of processing the data being transmitted and aggregated by the backhaul network. In the next chapter, we tackle the design of a cryptographic system for private ML training that could securely combine sensitive sensor data from many different sources without leaking them to the other participants. For this, we move from energy-constrained mobile platforms to *memory*-constrained GPU accelerators.

Chapter 3

Accelerating Multi-party Computation for ML Training using GPUs

3.1 Introduction

Applications like machine learning (ML) have enjoyed tremendous success in automating tasks such as biometric authentication, personalized ad recommendation, or detecting fraudulent financial transactions [31, 174, 175]. However, these models come at a significant privacy cost, as the data underlying them can be highly sensitive, ranging from medical data to online behavior and financial records. This has incentivized the development of privacy-preserving approaches to ML [233, 86, 75].

Secure Multi-Party Computation (SMC/MPC) has emerged as a promising tool for privacy-preserving computation [233, 90, 23]. MPC enables a group of entities to perform a joint computation without revealing their inputs to the computation. Thus, when data is sensitive, MPC can enable a the group of entities to generate insights from this data (such as training ML models or performing inference) without ever disclosing the data in plaintext to the other parties involved. MPC has shown tremendous progress in the past few years, making significant algorithmic improvements [33, 32, 160, 157] as well as robust, efficient, and versatile implementations [122, 189, 7, 53]. However, despite these advances, the overhead of MPC remains prohibitive when considering large computations. For instance, secure training of large machine learning models is over 4 orders of magnitude slower than plaintext training [223].

In the plaintext setting, large ML inference and training tasks are made practical by the use of GPUs – many-core hardware accelerators that support highly-parallelizable workloads. Individual operations, or kernels, are tiled across the many GPU processor threads to minimize execution time over large input data. For example, the use of GPUs can improve the training times of commonly used ML models by $10-30\times$ [198], making them an essen-

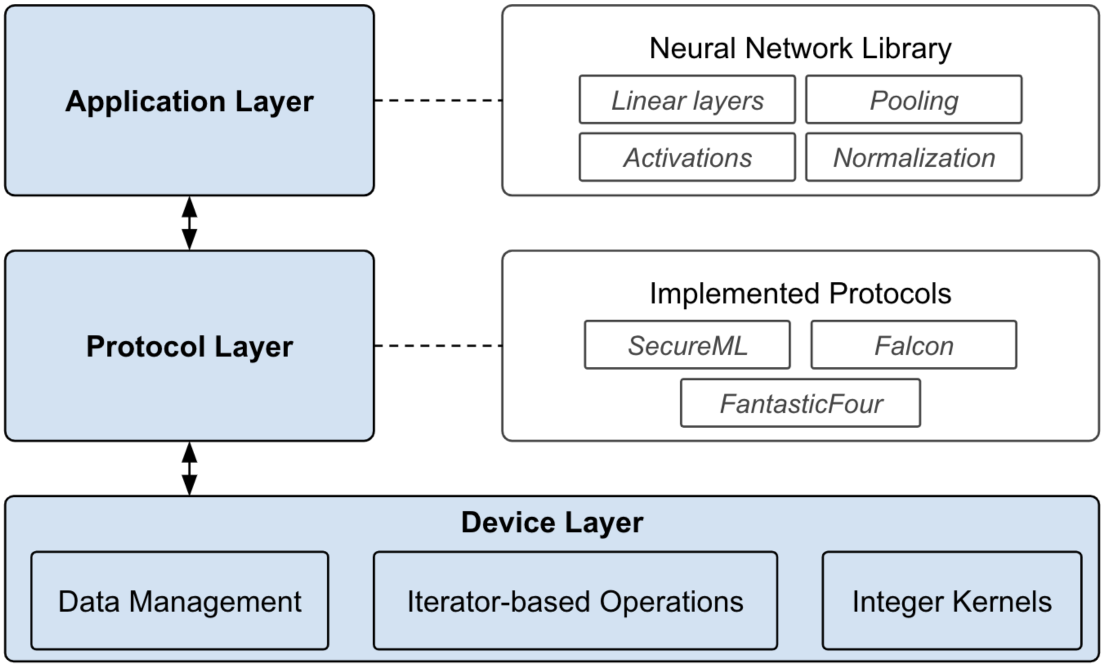


Figure 3.1: Piranha's three-layer architecture in blue, with components implemented on top in white. On the device layer, we contribute low-level GPU kernels accelerating local, integer-based data shares. At the protocol layer, we implement functionality for three different linear secret-sharing (LSSS) MPC protocols at the protocol layer: SecureML [161] (2-party), Falcon [223] (3-party), and FantasticFour [57] (4-party). At Piranha's application layer, we provide a protocol-agnostic neural network library that can be executed by any of the protocols. Piranha is modular in that it can support additional components beyond what we provide.

tial tool in today's ML infrastructure. A few recent works [157, 203, 107, 82] have utilized GPUs to accelerate MPC computation. However, their GPU usage is limited to accelerating individual operations or a specific MPC protocol. Delphi [157], for example, only accelerates convolution operations before continuing computation on the CPU, while CryptGPU [203] designs a specific 3-party protocol for its application. As a result, any new protocol must re-implement the same basic GPU support, a difficult task in general. Requiring MPC developers to develop domain-specific knowledge of GPU task scheduling and memory hierarchy to implement efficient kernels raises the barrier to entry and impedes the development of practical MPC-based systems. This raises a natural question: can secure MPC generally leverage GPU acceleration?

3.1.1 Challenges and Insights

Supporting efficient secure computation on the GPU faces a few core problems. Plain-text ML computation is straightforward and can be done directly in floating point with reasonable memory constraints, while the equivalent multi-party computation can be accomplished using any number of different protocols, operating over integer types, with significantly higher available memory requirements. We design Piranha to address these challenges while providing a general-purpose platform for MPC development, with support for linear secret-sharing schemes (LSSS), encompassing a large (and growing) number of state-of-the-art protocols for secure computation [43, 172, 184, 57, 221, 223, 149].

Challenge: Protocol-independent acceleration.

As even simple multi-party operations such as multiplications may be computed using a wide variety of approaches based on the protocol used, how can a platform efficiently provide acceleration support to each of them? While entire MPC computations are quite different, they are almost always decomposed into individual operations over local data shares mixed with communication between the parties to obtain the final result. Thus, accelerating local operations over local shares can yield significant performance benefits while remaining entirely protocol-independent. Piranha uses vector shares as the basic unit of computation over individual values, as it ensures that any protocol or application implemented using them will inherently take advantage of the GPU's parallelism. With a shared abstraction for local data, Piranha can transparently manage data transmission and memory allocation, keeping data on the GPU for the entirety of the computation while minimizing data transfer from the CPU.

Challenge: Enabling integer-based GPU computation.

Data representation is an important consideration for secure computation libraries. There is a tension between supporting high-precision real values required by applications such as NN training (e.g. float datatypes) and structured algebraic properties required by the secret sharing schemes (e.g. int datatypes). State-of-the-art secure computation libraries and frameworks resolve this tension by using fixed-point datatypes, encoding real values with a fixed precision into a large integral datatype (typically 64-bits). Unfortunately, GPUs primarily focus on accelerating floating-point computation with extremely efficient kernel implementations, targeting plaintext graphics and ML workloads. This has resulted in a dearth of GPU kernels for large bitsize (32- and 64-bit) integer computations [54]. We argue that the lack of integer kernels in existing GPU libraries significantly hampers simple acceleration for MPC protocols; Piranha explicitly provides for integer-based shares and matching GPU integer kernels to accelerate common operations.

Challenge: Supporting large MPC problems in limited GPU memory.

While modern CPUs boast terabytes of RAM for computation, present-day GPUs are constrained to a severely limited pool of available memory – 12 or 16 GB for commodity models. This is a salient issue for MPC, where protocols often maintain duplicated copies of data in separate secret shares, leading to a multiplicative increase in memory requirements. When paired with ML model parameters whose footprint can range in the gigabytes, even in plaintext, Piranha must make as efficient use of its limited device memory as possible. This can directly impact overall performance: in ML training, memory availability limits the total batch size – i.e. the number of data points processed in parallel – that can be supported on a single GPU. To address this problem, we primarily support in-place operations for local shares, performing additional memory allocation only when a protocol explicitly requests it. While applications like privacy-preserving ML training will always require a baseline allocation, encouraging protocols to reuse existing buffers minimizes temporary peaks in total memory usage. Second, MPC protocols may exhibit non-standard memory access patterns incompatible with the integer kernels available. Naively copying data into the desired layout before performing the computation unnecessarily limits the problem sizes we can support, so to efficiently parallelize some operations, Piranha's insight is that memory-efficient computation can be achieved with views over GPU memory, allowing for in-place computation. In particular, this approach precludes the need to manually modify GPU data layouts, avoiding any temporary memory allocation or data transfer overhead that the computation would normally require.

3.1.2 A Platform for MPC Acceleration

Piranha addresses these issues with a modular, three-layer GPU-based framework for secure computation (Figure 3.1). Piranha contributes a three-layer architecture: (1) a *device layer* that can independently accelerate secret-sharing protocols by providing integer-based kernels absent in current general-purpose GPU libraries, (2) a modular *protocol layer* that allows developers to maximize utility of limited GPU memory with in-place computation and iterator-based support for non-standard memory access patterns, and (3) an *application layer* that allows applications to remain completely agnostic to the underlying protocols they use (Section 3.3).

Piranha allows the MPC community to easily leverage the benefits of a GPU without requiring GPU expertise. We demonstrate the practical use of Piranha in implementing three different LSSS protocols for secure neural network training – the 2-party SecureML [161], 3-party Falcon [223], and 4-party Fantastic Four [57] protocols. Piranha does not propose a new secure multi-party protocol, rather, we focus on demonstrating how the platform accelerates existing protocols. We plug these protocols into a high-level neural network library to provide GPU-assisted private training and inference of ML models.

Compared to state-of-the-art CPU implementation [63] of computational building blocks such as matrix-multiplication, convolutions, and comparisons, Piranha improves runtime by

2 to 3 orders of magnitude. Thus, Piranha makes a big step forward towards practical MPC training. For example, prior work such as Falcon estimates that training a realistic neural network like VGG16 using its 3-party protocol would require 14 days [223]. In comparison, Piranha can perform the same training process in 33 hours, a 10× improvement.

One would expect that since Piranha accelerates general LSSS-based MPC, Piranha would thus be slower than a system like CryptGPU [203] that is tailored for a *specific* MPC protocol. We show that in fact, we achieve generality while demonstrating a 2-12× improvement in runtime and supporting up to a 4× greater problem size on the same GPU hardware. CryptGPU [203] only demonstrates full end-to-end training on simple networks such as AlexNet [131], while only micro-benchmarking single-layer training passes for larger networks like VGG16 [200] which has twice as many parameters. In contrast, for the first time, Piranha demonstrates the feasibility of training a *realistic* neural network like VGG [200], *end-to-end*, using MPC *in a little over one day.*

3.2 Background and Related Work

In recent years, a number of new frameworks have been proposed for privacy-preserving approaches to machine learning. While most frameworks demonstrate a CPU-only implementation, there are a few works that explore GPU assisted computation. The two earliest works by Husted *et. al.* [107] and Frederiksen and Nielsen [82] explore the use of GPUs for improving secure computation using garbled circuits and OT extensions. Delphi [157] uses GPUs to improve the performance of linear components of the computation. In a more recent work, CryptGPU [203] building on top of the CrypTen framework [53] uses GPUs for the entire computation. Recently, GForce [163] shows the benefits of GPU acceleration for secure inference. In a somewhat related effort, cuHE [56] and PixelVault [214] use GPUs for homomorphic encryption, securing keys, and encryption operations. Visor [179] has looked at using GPUs for secure computation over enclaves, while Slalom [207] investigates NN inference on trusted hardware.

A number of general purpose frameworks have improved the practical performance of MPC. In the dishonest majority setting, a number of works [124, 45, 77, 8, 191, 123] improve the performance of the original SPDZ protocols [60, 62]. Helen [243] proposes a system to train a linear model in a dishonest majority setting. Poseidon [195] explores the use of MPC techniques for federated learning in a similar corruption model. A lot more frameworks propose new specialized protocols and implementations in the semi-honest and honest majority adversarial settings. Recent 2-party computation frameworks include [161, 185, 139, 157, 118, 188, 173, 119] that typically look at protocols in the semi-honest setting. A number of frameworks explore a 3-party setup with an honest majority corruption. This includes [43, 172, 160, 218, 203, 128, 57]. Similarly, 4-party computation frameworks include [128, 57, 184, 37]. Other proposed frameworks include [242, 178]. An entire line of work improves the performance of garbled circuit based approaches to secure computation. Recent advances include as well as silent OT extension protocols such as [33, 34, 196, 231].

Finally, our platform can be used to implement efficient protocols for other applications such as sorting networks [48], ORAMs [89, 220], and differential privacy [75, 222].

A number of libraries with varying infrastructures are open sourced. MP-SPDZ and SCALE-MAMBA [122, 7] implement a number of protocols, including most of the dishonest majority protocols. CrypTen [53] implements a few protocols over PyTorch. Other popular libraries providing a number of useful secure computation tools include [67, 189]. There also exist open-source libraries for privacy-preserving machine learning such as Rosetta and PySyft [2, 3], but no open source library that enables general secure computation applications to benefit from the use of GPUs or the development of new accelerated protocols. Piranha can not only fill this gap, but reduce the performance gap between plaintext and privacy-preserving computation.

3.3 System Architecture

Piranha contributes three distinct, modular layers that provide a separation of concerns for GPU-accelerated secure computation (Figure 3.1): a *device* layer that abstracts GPU-specific code from MPC developers; a *protocol* layer that implements different MPC protocols, their secret-sharing schemes, and adversarial models; and an *application* layer that uses these protocols in an agnostic manner for high-level computation.

The *device layer* consists of two components. First, it provides an abstraction of a GPU-based integer vector, which represents a locally-held share of a vector whose values are secret-shared among multiple parties. These shares live on the GPU throughout the computation, minimizing data transfer overhead. Communication is handled in a protocol-independent manner: when necessary, the device layer copies a share to the CPU before transmitting it over the network. Second, the device layer maintains a set integer kernels that implement commonly-needed functionality (e.g. elementwise addition or matrix multiplication) over local share vectors. We discuss how MPC operations are accelerated in Section 3.4.

The *protocol layer* allows MPC developers to compose operations on local shares into a full protocol, benefiting from GPU acceleration without developing expert knowledge or re-implementing GPU support from scratch. Applications rely on each protocol to provide an interface in the form of a secret-shared vector and a set of functionalities that can operate on them. Alongside individual protocol definitions, we implement protocol functionality under the Arithmetic Black Box Model that can be used to supplement any of the specific protocols, demonstrating the benefit of Piranha's modular structure. In addition, MPC protocols can require intricate computation that cannot be foreseen at the device layer; Section 3.5 details how iterator-based views over local shares on the GPU can be used to enable these operations while remaining within the GPU's limited memory constraints.

Finally, at the *application layer*, computation can focus on solving domain-specific challenges such as secure neural network training, without a dependency on any specific protocol. The functionality set provided by each protocol determines which applications can use a given protocol without requiring modification.

To put Piranha in context, imagine implementing a simple privacy-preserving neural network layer. Its core logic (e.g. updating layer parameters during forward and backward passes) remains untouched at the application layer. Instead of using plaintext vectors, however, the layer makes use of a vector secret-shared by an implementation at the protocol layer, and operates on these secret shares using the corresponding protocol functionality, for example, a privacy-preserving matrix multiplication. In turn, the protocol decomposes its multiplication into a series of local matrix multiplications, which are accelerated by a protocol-independent integer kernel in Piranha's device layer.

Threat model

Piranha assumes that parties participating in a protocol execution operate in separate trust domains, using their dedicated GPUs (e.g. in a cloud provider of choice). A GPU in Piranha communicates with another parties' GPU through their associated CPUs and across a normal Internet connection. As such, Piranha can be used in both LAN and WAN environments. Due to Piranha acting as a platform for existing MPC protocols, parties executing an application with Piranha inherit the security guarantees of the underlying MPC protocol. For example, a protocol with semi-honest security retains those guarantees while being executed by Piranha. We implement and evaluate three such semi-honest protocols on top of Piranha in Section 3.7.

3.4 Device Layer for Accelerating Local Operations

Effectively and easily interfacing with the GPU is a major barrier to MPC developers who wish to accelerate their protocols, but lack experience in programming optimized GPU kernels. Thus, a flexible abstraction is needed to support a wide array of MPC protocols while minimizing any domain-specific knowledge required. In this section, we discuss how Piranha addresses two primary challenges in providing extensible GPU support for MPC protocols: managing vectorized GPU data and supporting acceleration for integer-based computation.

3.4.1 Data management on the GPU

Piranha provides access to GPU memory through a single data abstraction we call a DeviceData buffer. A key property that DeviceDatas maintain is that their data resides only on the GPU; no buffers are maintained in CPU memory to avoid data transfer overhead when computing with GPU-based kernels. In the context of MPC protocols, these buffers often logically correspond to local copies of a secret share. A DeviceData can be templated by integral C++ data types such as uint32_t or uint64_t. Share *vectors*, not individual share values, are the basic unit of computation in Piranha, and so the abstraction is functionally equivalent to a std :: vector class, except that the data remains on-device. Listing 1, lines 2-4 show a few examples of how DeviceData vectors can be initialized.

Listing 1 Sample DeviceData usage demonstrating its key capabilities: transparently accelerating element-wise operations (lines 7-8), using Piranha-implemented integer kernels for computation such as matrix multiplication (line 11), communicating share contents with other parties (lines 14-15), and using iterators to define views of existing data without performing a data copy (lines 18-20).

```
1  // Device share initialization
2  DeviceData<uint32_t> a = {1, 2, 3, 4, 5, 6};
3  DeviceData<uint32_t> b = {1, 0, 1};
4  DeviceData<uint32_t> c(2);
5
6  // Vectorized element-wise operations
7  a += 10;
8  a *= 2;
9
10 // GEMM call: a (2x3) * b (3x1) -> c (2x1)
11 c = gpu::gemm(a, b, 2, 1, 3);
12
13 // Communication with party id 1
14 a.send(1);
15 a.join();
16
17 // Even (offset=0) or odd (offset=1) values
18 DeviceData<uint32_t> d(
19   stride(c,2).begin()+offset,
20   stride(c,2).end()
21 );
```

Element-wise operations over collections of secret-shared values are common in secure computation. As a result, they are prime targets to accelerate in parallel, enabling the GPU to naturally improve protocol performance. As an added benefit, by using vectorized DeviceData shares, developers at the protocol layer inherently parallelize their protocol implementation. Piranha's device interface supports a variety of local operations on individual share vectors; as a simple example, lines 7 and 8 of Listing 1 perform an accelerated element-wise scalar addition and multiplication, with each value modified in parallel by a different GPU kernel thread.

A primary insight Piranha makes is that, independent of the specific protocol, MPC functionalities over secret-shared data decompose into a common set of local arithmetic operations. It is this narrow waist that the device layer targets for acceleration in a way that can benefit every MPC protocol. Consider a widely used primitive, secure matrix multiplication, that decomposes into simple matrix multiplications and additions over local data in a protocol-agnostic way. To this end, Piranha provides integer kernels for performing general matrix multiplication (GEMM) over the DeviceData class, which we use to build secure matrix multiplication protocols (cf. Section 3.5 for an example). An individual GEMM call is shown in Listing 1, line 11. In Section 3.7, we evaluate how these kernels improve the performance of secure matrix multiplication by up to 200× over a CPU-based implementation.

A Note on Communication

Currently, support for direct GPU-GPU communication over the network is nascent and not widely available. Thus, in Piranha, communication between GPUs is bridged via the CPU, incurring a data copy overhead for each round of communication. Given that GPU-CPU data transfer speeds are significantly faster than communication over the network, this overhead is not significant in the applications we consider. We manage communication by abstracting this complexity away from MPC developers by providing simple data transmission functions. A sample communication round to a different machine is shown at Listing 1, lines 14 and 15. In the background, Piranha copies the values in DeviceData a to a temporary CPU buffer, and transmits it over the network. The protocol execution can then wait until the buffer has been successfully sent by calling join() to synchronize protocol execution.

3.4.2 Iterator-based operations

Another key design criteria for Piranha's device share abstraction is memory efficiency. While CPU-based protocols have enjoyed "effectively" unlimited memory availability, realistic GPU-based MPC computation is restricted to commercially-available GPUs that generally have around 16 GBs of memory. Given the increase in memory consumption required by secret-shared protocols, the result of inefficient memory usage is to unnecessarily limit application problem sizes. Furthermore, the overhead of data allocation, particularly for vectors of large sizes, forms a significant portion of the total overhead of using GPUs. To address this issue, we seek to avoid any redundant temporary data allocation used to transform data into a specific layout for kernel execution. We achieve this using an iterator-based abstraction in our DeviceData class, as follows.

Piranha's iterators allow the developer to traverse data vectors in a program-defined order, applying operations over a "view" of GPU memory decoupled from the actual physical data layout. For instance, a common operation requires pairwise operation over elements of a vector (cf Section 3.5 for details), i.e., operations over $\mathsf{vec}[2\mathsf{i}], \mathsf{vec}[2\mathsf{i}+1]$ for a given vector vec and over all indices i. A naïve approach would either require copying the odd and even components of the vector or to allocate new memory for storing the result. Our iterator-based approach allows us to define odd and even views over the same vector that effectively allow the GPU to interpret the memory with a stride of 2. This abstraction enables memory efficient code design by allowing us to view a given memory allocation in different ways. Hence, this approach encourages limited additional memory allocation – performing in-place element-wise operations as well as storing the computation result in existing memory.

Lines 17-21 of Listing 1 demonstrate this concept. The two DeviceData vectors even and odd hold a view of all the values in c at a stride of 2, or put otherwise, skipping every other value (odd starts at index 1). Note that this is simply a "view", i.e., even, odd operate on the same physical memory held by the original DeviceData c. Creating this view for every other indexed value allows a pairwise computation to be performed *with no additional memory allocation required.*

3.4.3 Integer kernels

The MPC protocols we implement in Section 3.5.4 operate on additive secret sharing over 32- or 64-bit ranges. As discussed in Section 3.1, there is a lack of kernel implementations for these data types [54], because prior work has focused on improving the performance for floating point data types. Some integer kernels are implemented for 8-bit matrix multiplications into 32-bit accumulators, for example, but the lack of support for larger integer types can be attributed to concerns of overflow in the product. Thus, there are two ways to benefit from GPUs for large bit-width integer types.

The first is to decompose large integers into multiple values of smaller width, such as 16 bits, representing the original value $x = x_3 2^{48} + x_2 2^{32} + x_1 2^{16} + x_0$. Computation can then be performed over each 16-bit sub-value x_3, x_2, x_1, x_0 by embedding them into 64-bit floating point types. Note that a large slack is required, as the result of multiplying matrices of 16-bit values will often exceed 32 bits in size and floating point computation does not have the same modular overflow as for integers. The problem with this approach is that it requires multiple individual floating point kernel calls over 16-bit values to compute one 32- or 64-bit integer result.

The second approach, which we take, is to directly implement kernels over integer data types. Piranha directly adds support for full-size integer matrix multiplication and convolution kernels at the device layer. We use the general-purpose templated matrix multiplication and convolution kernels in CUTLASS [55] to support 32- and 64-bit integer types.

While we cannot use existing, highly optimized floating-point GPU kernels such as those provided by cuBLAS [54], there are two benefits to our approach:

- Piranha's modular structure allows independent improvement of kernels, and thus future hardware support for large integer operations on GPUs can be easily integrated and benefit all pre-existing protocols, and
- the ability to directly compute integer results in a single call to a GPU kernel yields a better performance overall than multiple calls to a more efficient floating point kernel.

We demonstrate these gains in Section 3.7.

3.5 Protocol Layer for Linear Secret-Sharing Schemes

Piranha provides a framework for implementing various MPC protocols leveraging the benefits of GPU acceleration. We first describe how we use Piranha's DeviceData class to implement MPC protocols, then highlight how complex protocols can be parallelized in a memory-efficient manner, and finally, how Piranha allows for functionality reuse between protocols.

3.5.1 MPC protocol implementation

Any protocol implemented in Piranha specifies two things: the secret sharing base, including the adversarial model, and operations over this secret sharing base. For example, suppose a MPC developer seeks to implement a 3-party protocol using replicated secret sharing for an honest majority of semi-honest corruptions (for instance [15, 223]). In this setting, a secret value x is composed of 3 shares $x \equiv x_0 + x_1 + x_2$, where each party holds only 2 of the 3 shares. Thus, the class for such a protocol will contain two DeviceData objects, one per share. Simple operations such as additions can be specified component-wise, leveraging the underlying GPU layer as shown in Listing 1.

To multiply two secret matrices x, y, if the first party holds shares (x_0, x_1), and (y_0, y_1), the output can be computed by regrouping the terms of the product as [15]:

$$\begin{aligned} x \cdot y &= (x_0 + x_1 + x_2) \cdot (y_0 + y_1 + y_2) \\ &= (x_0 \cdot y_0 + x_0 \cdot y_1 + x_1 \cdot y_0) + (...) + (...) \end{aligned} \tag{3.1}$$

Thus the computation can be split such that the first term can be computed locally by the first party (and similarly for the other parties). Leveraging the device layer for each individual local GEMM computation (cf. Listing 1 line 11), the overall secure matrix multiplication protocol can be easily implemented as shown in Listing 2. This example shows the ease of implementing various MPC functionalities in Piranha's protocol layer by building over the local functionality at the device level. In Section 3.7, we directly evaluate the performance benefit of this implementation against a similar CPU-based protocol for secure matrix multiplication.

Randomness generation

We assume that the parties maintain secure point-to-point communication channels and share pairwise AES keys to generate common randomness. Recent works have looked at efficiently generating randomness on the GPUs [16, 138, 230]. While CPU cores outperform GPU cores for smaller amount of randomness, it becomes desirable to generate randomness using GPUs for large scale random number requirements [16]. Such random number generation can be easily added to the protocol layer in Piranha.

Listing 2 A replicated secret-sharing protocol class (3-party setting) implemented at the Piranha protocol layer. The protocol specifies the secret-sharing base: each party has two local DeviceData shares templated by type T. The matmul functionality is performed for this class by implementing a secure matrix multiplication based on Eq. 3.1.

```
// Replicated secret sharing class
class RSS<T> {
    DeviceData<T> shareA, shareB;
}

void RSS<T>::matmul(RSS<T> a, RSS<T> b,
                    RSS<T> c, ...) {
    DeviceData<T> localC;

    localC += gpu::gemm(a.shareA, b.shareA, ...);
    localC += gpu::gemm(a.shareA, b.shareB, ...);
    localC += gpu::gemm(a.shareB, b.shareA, ...);
    // Reshare and truncate localC to c
}
```

3.5.2 Memory-efficient protocols

Section 3.4 demonstrates an iterator-based implementation for DeviceData buffers. In this section, we showcase how this abstraction can be used to perform efficient in-place memory computations. As an example, we consider a CarryOut protocol, that securely computes the carry bit for binary addition i.e., given the bitwise sharing $(a_{k-1}, \cdots, a_0)$ and $(b_{k-1}, ..., b_0)$ of two k-bit vales a, b, the goal is to compute the carry bit at the MSB c_k. This primitive forms the backbone of nearly every state-of-the-art comparison protocol [149, 160, 77, 61]. In the case of the neural network library we discuss in Section 3.6, comparisons enable standard activation functions and pooling operations including ReLU and Maxpool.

The computation proceeds in $\log_2 k$ rounds by emulating a simple carry-lookahead adder [197]. As part of the computation, at round $i \in \{1, 2, \cdots, \log_2 k\}$, the CarryOut computes the AND between adjacent propagating bits, i.e., $p'_j = p_{2j} \wedge p_{2j+1}$ where p_j are propagation bits at round i and p'_j are the propagation bits for the next round. At the end of $\log_2 k$ rounds, the final bit is the result of CarryOut.

A naïve implementation of the above will suffer from two major inefficiencies. First, bitwise expansion requires that each secret-shared bit be stored separately, increasing the memory footprint on the GPU. Second, using contiguous allocations to separate pairwise bits results in non-trivial overhead from additional memory use and data copies. Figure 3.2a shows 3 rounds of this CarryOut operation implementation where the propagating p bits are combined. Unfortunately, due to the vectorized nature of data computation on the GPU, half of p must be copied at each step to a different memory allocation before the next round can

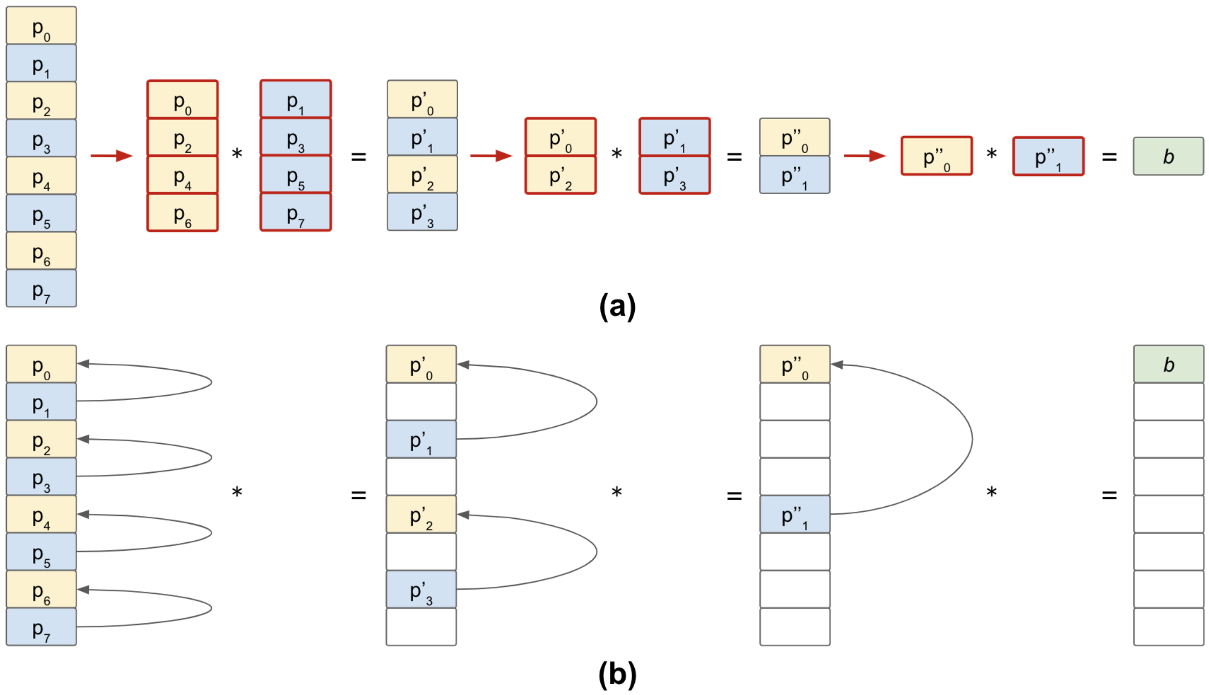


Figure 3.2: Comparison of a memory-inefficient naive carryout implementation Figure 3.2a and our iterator-based in-place computation Figure 3.2b. In the former approach, new memory allocations and data copy – highlighted in red – are done to split pairwise elements into contiguous vectors for parallel GPU processing. The ability to define iterators and execute kernels over non-contiguous memory allows Piranha to avoid any additional memory allocation.

be evaluated (red-outlined in Figure 3.2a). During one execution of this particular CarryOut implementation, $\log(n)$ additional data copies are performed.

In contrast, Piranha uses iterator-based views to allow access to non-contiguous data elements in strides. Figure 3.2b demonstrates an memory-optimized version of CarryOut leveraging this ability. For each round, the protocol defines two iterators, one for every even element (yellow values), and one for every odd element (blue values), and uses those as the basis for executing a kernel computing the next values of the propagation bit. The iterators are input to a pairwise comparison kernel that would otherwise expect data marshalled into a specific contiguous layout, allowing for efficient computation entirely without data movement.

Furthermore, we can reuse the first iterator to store the results in the original allocated buffer, resulting in no additional data copies or memory allocation. Since the entire bitwise vector is allocated until the end of the protocol, we continue to use (increasingly less of) it to store intermediate results until the final carry bit is calculated. Finally, templating allows the bit-vectors to use smaller datatypes (say uint8_t) compared to the datatypes used in the secure computation (say uint64_t), thus minimizing the memory footprint they require on the GPU.

3.5.3 Reusable protocol components

The structure of Piranha supports reusing protocol implementations, so that protocols can build on other implementations in a number of ways. For instance, a new protocol for secure comparison that operates in the same setting as another implemented protocol in Piranha can focus solely on implementing the secure comparison functionality and inherit the rest from the existing share type in Piranha. This also helps in maintaining the compatibility at the application layer.

Another reusable component of Piranha is the implementation of share agnostic functionalities. For example, this includes protocols that have been proven secure in the arithmetic black box model $\mathcal{F}_{\mathsf{ABB}}$ [58]. Such protocols are specified agnostic to the specific adversarial model and remain the same as long as the basic operations are performed securely in the specific adversarial model. A number of cryptographic primitives and functionalities are proven secure in this model [135, 77, 149]. Piranha allows such methods to be generically implemented once at the protocol level, alongside protocol-specific functionality, and can then be inherited by any other implementation. As two examples, we implement a state-of-the-art comparison protocol by Makri *et. al.* [149] and a protocol for approximate square-root and inverse computation based on [223].

Secure comparison

We use secure comparison as an example of implementing a method in the arithmetic black box model. The comparison protocol uses edaBits [77] as preprocessing material to efficiently compute a comparison of secret values. An edaBit is a secret sharing of a random value and the bit decomposition of the same value as boolean shares i.e.,

$$\mathsf{edaBit} : [r]_M\,, [r_0]_2\,, [r_1]_2\,, \cdots, [r_m]_2 \text{ where } r \overset{\$}{\leftarrow} \mathbb{Z}_M \tag{3.2}$$

where $m + 1 = \log_2 M$. The protocol for generating this can be found in [77]. The problem of secure comparison over arithmetic secret-sharing can then be converted to a secure comparison over boolean secret-sharing using the edaBit. The latter can then be implemented efficiently using bitwise operations such as CarryOut [197]. Details of this operation are presented in Section 3.5.2.

Approximate computations

The privacy-preserving neural network application we implement requires a pair of specific protocols for the normalization layers: secure integer division and secure computation of a square root. MPC protocols for these primitives typically require approximate computation using Newton's methods. We write a generic functionality based on the protocols from [223, 192] where we find the nearest power of two for each input value and then evaluate a fixed-point Taylor series polynomial approximation. We use a simple Python script to compute polynomials of a given degree that approximate each target function, in this case, sqrt and inverse. These functionalities are then implemented and used across different protocols. Specifically, Piranha uses the following approximations:

$$\begin{aligned} \mathsf{sqrt}(x) &= 0.424 + 0.584(x) \\ 1/x &= 4.245 - 5.857(x) + 2.630(x^2) \end{aligned} \tag{3.3}$$

These approximations achieve an L1 error of 0.00676 and 0.02029, respectively, for x between 0.5 and 1.

3.5.4 MPC protocols

We implement three different MPC protocols to demonstrate Piranha's generality at the protocol layer: a 2-party implementation based on SecureML [161], a 3-party implementation built upon Falcon [223], and a 4-party protocol [57]. We briefly describe each of these protocols below, and prefix them with "P-" to indicate they are implementations accelerated by Piranha.

Two-party protocol (P-SecureML)

In 2017, Mohassel and Zhang [161] proposed a 2-party (and a trusted third party variant) protocol for privacy-preserving machine learning, using a 2-out-of-2 arithmetic secret sharing as the basis for its functionality. The linear layers are computed using Beaver triples and the non-linear layers are evaluated with garbled circuits. In our implementation, we replace the expensive GC-based evaluation of ReLUs with a more recent and efficient comparison protocol using edaBits [149, 77].

Three-party protocol (P-Falcon)

We build a 3-party protocol using the work of Wagh *et. al.* [223]. It uses a 2-out-of-3 replicated secret-sharing as the basis for its functionality. The linear layers are performed using local multiplications and resharing, a technique used in many other 3PC frameworks [15, 83, 160]. The non-linear layers are computed using a specialized comparison protocol building upon [221]. Once again, we replace the comparison protocol using the more efficient work by Makri *et. al.* [149].

Four-party protocol (P-FantasticFour)

Our 4-party implementation follows the work of Dalskov *et. al.* [57]. It uses 3-out-of-4 replicated secret sharing: linear layers are performed using a generalization of the replicated secret sharing approach, thus using a combination of local multiplications and resharing (known as joint message passing and INP in the work and similar to [128]). For comparison (probabilistic truncation), the protocol uses a combination of [76] and [128].

3.6 Application Layer for Secure Training and Inference

Our final layer of abstraction is the neural network layer. This interface is guided by the types of the deep learning architectures we wish to support. Currently, Piranha implements protocol-agnostic versions of the following layers in full generality [5, 18]:

1. Linear layers: Convolution and fully-connected layers
2. Pooling operations: Maxpool and averagepool
3. Activation functions: ReLU
4. Normalization: Layer normalization

Layers use the popular Kaiming weight initialization [98]. Any neural network architecture that is composed of these layers can be run using Piranha. This covers a large class of popular networks used in computer vision - from simple multi-layer perceptrons like SecureML [161] to more complex convolutional neural networks such as AlexNet [131] and VGG16 [200]. In our evaluation in Section 3.7, we compare Piranha to the networks used in prior works [203, 223].

3.6.1 Interfacing the neural network library

As discussed in Section 3.6, we focus on the secure evaluation of neural network models as our target application. To support the neural network library over multiple MPC protocols, we require each MPC protocol to implement a common set of functionalities. Once this set is implemented, the protocol can support training and inference over any neural network architecture constructed with the supported layers. This required set of MPC functionalities is given in Table 3.1.

Listing 3 shows a simplified look at the forward pass of a fully connected layer. The functionality simply takes a batch of inputs, multiplies them with the layer weights and adds the layer's bias to compute the activations. The forward pass implementation is protocol-agnostic in that it can be templated with any given Share type (e.g. from Listing 2's RSS share) and requires only that the required functionality matmul be implemented by that protocol.

matmul(...)	Matrix multiplication of two matrices.
convolution(...)	Convolution of two tensors.
maxpool(...)	Compute the maximum of set of values.
truncate(...)	Truncate i.e., divide shares by power of 2.
reconstruct(...)	Opening of secret shares.
selectShare(...)	Select one out of two shares given a boolean secret shared value.
comparison(...)	Compare two shares.
sqrt(...)	Compute an approximate square root.
inverse(...)	Compute an approximate fixed-point inverse.

Table 3.1: Functionalities required by the NN training application, implemented by each class in Piranha's protocol layer.

3.6.2 Secure training of neural networks

Training neural networks, especially larger and deeper networks presents a number of challenges. In order to demonstrate learning, we face three major challenges:

1. Back propagation gradients are frequently much smaller than the remaining activations and must be preserved by the finite precision available in fixed-point integers.

2. The quality of the gradients can also significantly affect the training process. Ensuring that the final layer gradient computation is accurate has a significant impact on how well the network trains. Inaccuracies are compounded by linear layers, which yield approximate values due to each multiplication performed with finite precision arithmetic.

3. Closely related to the previous issue is the stability of the final layer gradients. As the network trains, the magnitudes of the final layer activations grow in size. Softmax [74] computations to generate the needed gradients (which involve an exponentiation) can quickly exceed the size of the data type, yielding an overflow and destabilizing the learning process.

We showcase in Section 3.7 that privately training neural networks is indeed possible for large networks with over 100 million parameters. We use fixed-point arithmetic to encode real numbers for neural network experiments. For private inference, we observe that the neural network can be run over 32-bit data-types with a fixed-point precision of 13 bits. However, for private training, to retain the gradients with sufficient precision, we use 64-bit data types with 20 or more bits of fixed-point precision, with deeper network depths

Listing 3 Protocol-agnostic implementation of a fully-connected neural network layer. Any protocol class, such as the RSS class in Listing 2, that implements the desired matmul functionality can be used to compute the forward pass.

```
// Fully connected layer forward pass
template<typename Share>
void FCLayer<Share>forward(Share input) {
    matmul(input, this->weights,
            this->activations, ...);
    this->activations += this->bias;
}
```

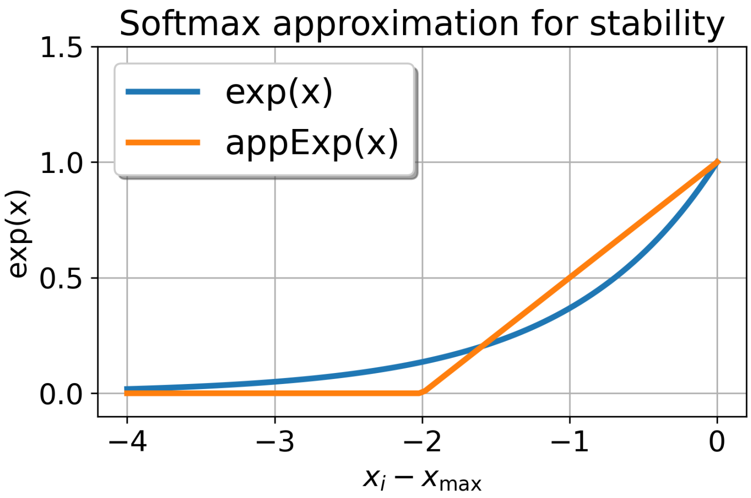


Figure 3.3: Our new approximate computation of last layer gradients that stabilize the learning process.

requiring higher precision (Section 3.7.5). Finally, to address latter challenges, we propose a new gradient computation function. Our gradient computation has two main advantages: it is more stable to large activations, and it is MPC-friendly. The first is achieved because we approximate the exponential with a function that does not increase the magnitude of the secret-shared values. The second is achieved by using only comparisons, which significantly reduces the round complexity of the computation.

Gradient Computations

In order to compute the gradients for the backward propagation [99], we apply a softmax coupled with the cross-entropy loss function. Suppose the output of the last layer is $\mathbf{x} = (x_0, \cdots, x_9)$, and $\mathbf{y} = (y_0, \cdots, y_9)$ is a one hot encoding of the true label, then the loss function (per image) is given by:

$$\ell = -\sum_i y_i \log p_i \quad \text{where} \quad p_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{3.4}$$

The gradient is then given by:

$$\nabla_i = \frac{\partial \ell}{\partial x_i} = p_i - y_i \tag{3.5}$$

While there are a few different ways to compute this gradient [125], they do not solve the challenges mentioned above, which are critical when training is performed on larger networks and datasets. Note that the softmax function remains the same if the logits p_i are computed using the activations $x_i - x_{\mathsf{max}}$ where $x_{\mathsf{max}} = \max(x_1, \cdots, x_k)$ if k is the number of classes. In other words,

$$p_i = \frac{e^{x_i - x_{\mathsf{max}}}}{\sum_{j=1}^{k} e^{x_j - x_{\mathsf{max}}}} \tag{3.6}$$

We propose a new function computation to approximate the above computation (Eq. 3.6):

$$p_i \approx \mathsf{appExp}(x_i - x_{\mathsf{max}}) / \sum_{j=1}^{k} \mathsf{appExp}(x_j - x_{\mathsf{max}}) \tag{3.7}$$

where $\mathsf{appExp}(\cdot)$ is the approximate exponential function as shown in Fig. 3.3. We compute the inverse in plaintext using a functionality similar to Falcon. To preserve the long tail of the exponential, we add a small bias of 10^{-3} to each component of $\mathsf{appExp}(\cdot)$. Note that this function is (1) relatively easy to compute within MPC, and (2) preserves (i.e., does not increase) the magnitude of the activations. These factors make the gradient computations using this function stable from the machine learning perspective.

3.7 Evaluation

In our evaluation, we answer the following questions:

1. *In comparison to state-of-the-art, CPU-based prior work, how well does* Piranha *accelerate the same computation tasks?* (Section 3.7.2)

2. *Can* Piranha *be used to successfully and securely train large neural networks (e.g. over 100 million parameters) in a reasonable amount of time?* (Section 3.7.4 and Section 3.7.5)

3. *What are* Piranha*'s computation and communication costs in LAN and WAN environments?* (Section 3.7.6)

4. *How well does* Piranha *manage constrained GPU memory and how well does its memory-conscious design improve scalability at the application layer?* (Section 3.7.7)

5. *How does the runtime performance of privacy-preserving inference and training, supported by* Piranha*'s protocol-agnostic acceleration, compare with prior work on targeted protocols?* (Section 3.7.8)

3.7.1 Evaluation set-up

We run our experiments over similar hardware and networking environments as prior works [161, 223, 57]. For CPU-based implementations, we use Azure F32s_v2 instances with Intel Xeon Platinum 8272CL @ 3.4GHz processors and 64 GB of RAM. Networked experiments are executed in a LAN setting with a bandwidth of 10 Gbps and ping time of 0.2 ms. GPU-based experiments are run on Azure NC6s_v3 instances with 6-core Intel Xeon E5-2690 v4 CPUs with 112 GB RAM and Nvidia Tesla V100 GPUs with 16 GB RAM.

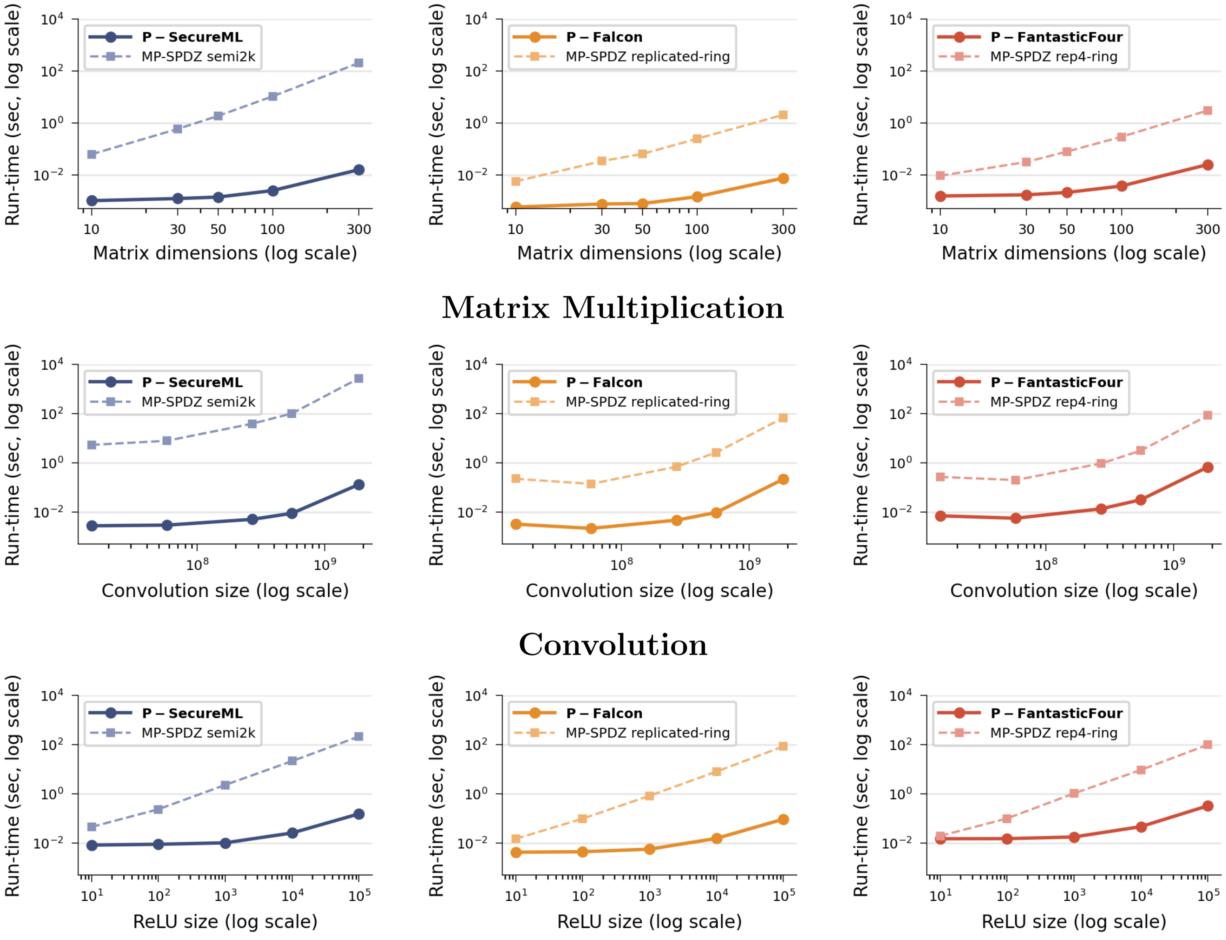
We add matrix multiplication and convolution kernels for large integer types by building on CUTLASS [55], at commit 0f10563, to which we add support for 32- and 64-bit integer matrix multiplication and convolution. We use the default tiling parameters, while element-wise kernels are parallelized using Thrust [169].

Baseline

As a baseline, we compare against protocol implementations from MP-SPDZ [122, 63] at commit e6dbb4. MP-SPDZ is a state-of-the-art open-source secure computation platform with over 34 protocols and represents a CPU-based analog to Piranha. For each MPC protocol that we implement, we choose a state-of-the-art protocol implemented by MP-SPDZ in the same setting: individual operations are benchmarked in Section 3.7.2 with the 2-party semi2k, 3-party replicated-ring, and 4-party rep4-ring protocols. Each of these implementations operate on a single CPU core. We focus evaluating Piranha's performance in the data-dependent "online" phase, as offline generation of data-independent components such as Beaver triples [161] or edaBits [77] can be easily parallelized independently from a particular computation.

Models and Datasets

We evaluate our high-level neural network library with four neural network architectures: SecureML [161], a simple 3-layer network, and LeNet [136], a 5-layer convolutional network, over MNIST [158], and AlexNet [131], an 8-layer convolutional network, and VGG16 [200], a 16-layer convolutional network, over the CIFAR10 dataset [130]. While Piranha fully supports the use of maxpool layers in these architectures, as in CryptGPU [203], we substitute them with averagepool layers to maintain comparative accuracy. Notably, averaging operations are significantly less expensive than max operations in each Piranha-accelerated protocol, as summation requires only a locally-computed linear combination of secret shares while oblivious comparison incurs a logarithmic number of communication rounds among the parties.



Matrix Multiplication

Convolution

ReLU

Figure 3.4: The figures benchmark secure protocols for matrix multiplication, convolutions, and ReLU across 2-, 3-, and 4-party protocols for various sizes of these computations. Piranha consistently improves the run-time of these computations, with improvements as large as 2-4 orders of magnitude for larger computation sizes.

3.7.2 Comparison vs. CPU Implementations

In this section, we compare the performance of Piranha with state-of-the-art CPU-based protocols over a set of MPC workloads. For each protocol discussed in Section 3.5.4, we execute individual operations commonly used by a secure neural network application – matrix multiplications, convolutions, and ReLU comparisons – and compare against the same operations computed using MP-SPDZ [63] with protocols in the same setting, as described in Section 3.7.1. In general, our results find that Piranha's acceleration can improve performance by *2-3 orders of magnitude* for these important MPC functionalities. Figure 3.4 summarizes the results for each these operations as a function of various problem sizes.

We evaluate matrix multiplication performance by multiplying two $N \times N$ matrices for logarithmically-increasing values of N. Considering small matrices of dimension $N = 10$, where platform overhead such as data transfer to the GPU is most likely to have an outsized impact on overall performance, we find that using Piranha results in a performance benefit of 6 to 60× in the four- or two-party settings, respectively. Likewise, as the problem size increases, so does the impact of GPU acceleration on runtime. For the largest matrix multiplication benchmarks with $N = 300$, Piranha's 3- and 4-party protocols improve on the CPU-based MP-SPDZ implementations by 2 orders of magnitude, while P-SecureML shows a 4 order of magnitude improvement over MP-SPDZ's semi2k implementation.

For the convolutions, we benchmark problems in order of increasing complexity. Each convolution layer is parameterized by a $[iw, c_{\mathsf{in}}, c_{\mathsf{out}}, f]$ tuple, where iw is the input image dimension, c_{in} and c_{out} are the number of input and output channels, respectively, and f is the filter size. We use the total number of multiplications as a proxy for layer complexity (the complexity of the resulting unrolled matrix multiplication). The specific convolutions we compute are listed in Figure 3.4, ranging in complexity from 1.47×10^7 to 1.86×10^9 multiplications. Similar to the matrix multiplication benchmarks, Piranha shows a significant improvement in performance, performing on average 175 and 73× better in the 3- and 4-party setting, respectively. Piranha is much faster than the MP-SPDZ 2-party semi2k implementation, achieving a speed up of 3 orders of magnitude, on average.

Finally, ReLU operations are benchmarked over N-element vectors of logarithmically increasing size. For small vectors of $N = 10$ vectors, Piranha improves on each CPU-based protocol by between 1.3 and 5.5×, again seeing modest gains due to overhead dominating the relatively simple computation. For large vector sizes, we show extensive gains by applying GPU acceleration. Figure 3.4 shows between a 300 and 1380× speedup across MPC protocols over large ReLU inputs, completing 90 second CPU-based operations in less than a second.

3.7.3 Comparison with Floating Point Kernels

We mention in Section 3.4 the tradeoff in performance when computing directly over integer buffers on the GPU, as opposed to decomposing large bit-width values into smaller chunks for use in floating point-based kernels. In Table 3.3, we compare the 32- and 64-bit kernels that Piranha uses, implemented with CUTLASS [55], against state-of-the-art 32- and 64-bit

Network (Dataset)	Protocol	Time (min)	Comm. (GB)	Accuracy	
				Train (%)	Test (%)
SecureML (MNIST)	P-SecureML	12.99	49.55	97.37	96.56
	P-Falcon	7.51	22.84	97.37	96.56
	P-FantasticFour	23.39	33.01	97.37	96.56
LeNet (MNIST)	P-SecureML	87.55	683.18	96.78	96.80
	P-Falcon	71.56	485.90	96.88	97.10
	P-FantasticFour	219.20	676.13	96.88	97.11
AlexNet (CIFAR10)	P-SecureML	156.01	740.50	40.74	40.47
	P-Falcon	110.66	382.18	40.59	40.71
	P-FantasticFour	296.57	533.74	40.97	40.14
VGG16 (CIFAR10)	P-SecureML	3822.84	35454.91	55.02	54.35
	P-Falcon	1979.92	17235.35	55.13	54.26
	P-FantasticFour	7697.54	29106.24	55.02	54.35

Table 3.2: Time and communication costs for completing 10 training iterations over four neural network architectures, for each of Piranha's MPC protocol implementations. We are the first work to demonstrate end-to-end secure training of VGG16, a network with over 100 million parameters.

Kernel		Time (ms)			
Library	Datatype	784x9x20	1024x27x64	784x147x64	10000x1000x10000
cuBLAS	float-32	0.014	4.16	4.45	54.19
Piranha	float-32	0.981	4.51	4.56	65.16
Piranha	int-32	3.61	4.38	4.52	78.35
cuBLAS	float-64	4.58	6.37	4.70	126.5
Piranha	float-64	4.60	5.92	4.69	114.95
Piranha	int-64	4.76	4.66	4.90	2482.17

Table 3.3: Runtime for matrix multiplication kernels used in Piranha vs. the cuBLAS implementation for different sizes.

floating point kernels from cuBLAS [54]. While we can directly compare floating point performance between the systems, cuBLAS does not support large integer matrix multiplication, so we only present Piranha-based results for comparison.

We benchmark the runtimes for the matrix multiplication kernels used in Piranha vs. the cuBLAS implementation on various sizes of matrices in Table 3.3. We observer that Piranha kernels, when executed with floating point datatypes result in comparable overhead to cuBLAS implementations. However, executing 32-bit integer multiplications is much more expensive in Piranha compared to the floating point case. 64-bit integer multiplications are relatively comparable to cuBLAS 64-bit floating point, but at very large matrix sizes, there is a significant difference between the two. This is likely due to the fact that 64-bit integer operations are emulated using 32-bit integer instructions that target the GPU integer cores used.

3.7.4 Secure Training of Neural Networks

No prior work has successfully trained, within secure computation, a network such as VGG16, which over CIFAR10 has over *100 million learnable parameters*. While existing work has estimated the time to train such a network, the training times are prohibitively large – over 14 days [223] to complete 10 training epochs. This work is the first to securely train such a neural network, in less than a day and a half: our results are detailed in Table 3.2.

We train each network with each protocol Piranha currently supports for 10 epochs with 128-image batches. For each training run, we report the total training time and per-party communication. Every training pass used the MPC-friendly softmax replacement we propose in Section 3.6; over every network architecture we evaluate, our approximation remains stable and allows the networks to train successfully. To ensure that even small gradients can backpropogate through each networks and train a useful model, we vary the level of fixed-point precision: we train the shallow SecureML with 20 bits of fixed-point precision, LeNet and AlexNet with 23 bits, and VGG16 with 26 bits of precision. We further discuss how fixed-point precision impacts model accuracy in Section 3.7.5.

On a small dataset like MNIST, Table 3.2 shows that Piranha's neural network training library can quickly train SecureML and LeNet, achieving greater than 96% test accuracy in no more than 2 hours with P-Falcon and P-SecureML, compared to approximately 97% and 98% accuracy, respectively, when trained in plaintext. For larger networks, the cost of privacy-preserving matrix multiplication dominates the overall runtime [223]. This explains why P-FantasticFour generally takes 2 to 3× longer for the same training pass, because the 4-party protocol requires 7 local matrix multiplication operations for every privacy-preserving matrix multiplication, compared to only 3 local multiplications for the 3-party P-Falcon implementation.

On the larger CIFAR10 dataset, training times increase significantly but remain feasible. Over AlexNet, all protocols can successfully complete their training runs in under 5 hours, achieving 40% test accuracy over that time. We observed a 59% accuracy when training the same model in plaintext (note that an untrained network/random guessing achieves a 10%

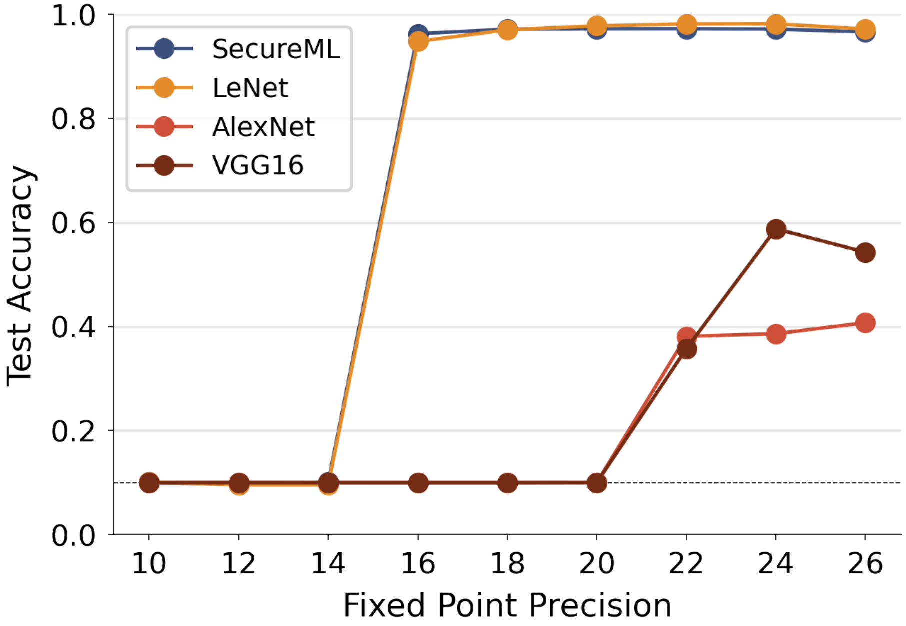


Figure 3.5: Test accuracy as the fixed-point precision increase for each network architecture, after 10 training epochs using P-Falcon. The dashed line indicates the baseline accuracy when randomly guessing. Sharp increases in training accuracy indicate that the model now has enough precision to fully backpropagate gradients.

accuracy given that there are 10 classes). When considering VGG16, the largest network Piranha trains over, training times are considerable: P-SecureML and P-FantasticFour require 2 and 5 days, respectively, to complete. Importantly, however, we can complete 3-party VGG16 training in only 33 hours with 54% test accuracy (compare to 67% test accuracy in plaintext on the same model), which prior work estimated to take *14 days* but did not actually execute the training [223].

These training times are only possible due to two main factors. First, improved computation times (through the use of GPU-accelerated kernels) reduces the overhead of matrix multiplication and convolution, whose costs grow super-quadratically with their dimensions, and are a significant part of the total runtime. The second is the ability to train over large batch sizes. Large batch sizes improve the efficiency of the stochastic gradient descent algorithm, and runtime scales better with batch sizes. Thus, a batch size of 128 has a lower run-time than computing over two 64-image batches.

3.7.5 Impact of Fixed-point Precision

For deeper networks, we observe that gradients reaching the initial layers routinely approach 2^{-20}, and are further reduced by the current learning rate. If the fixed-point precision

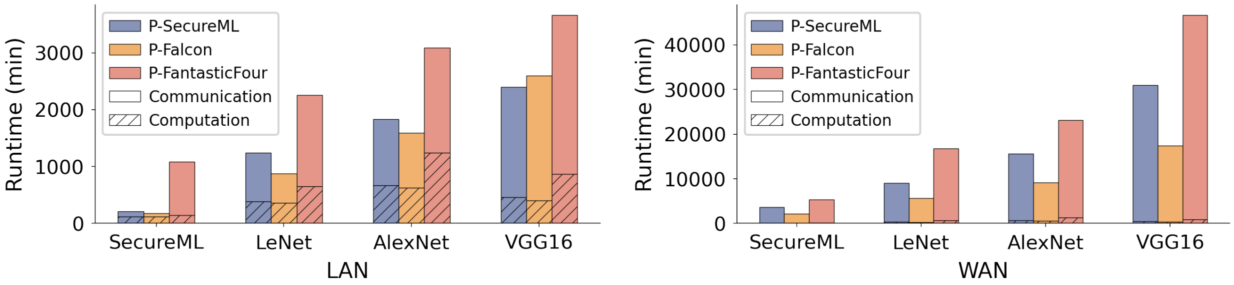
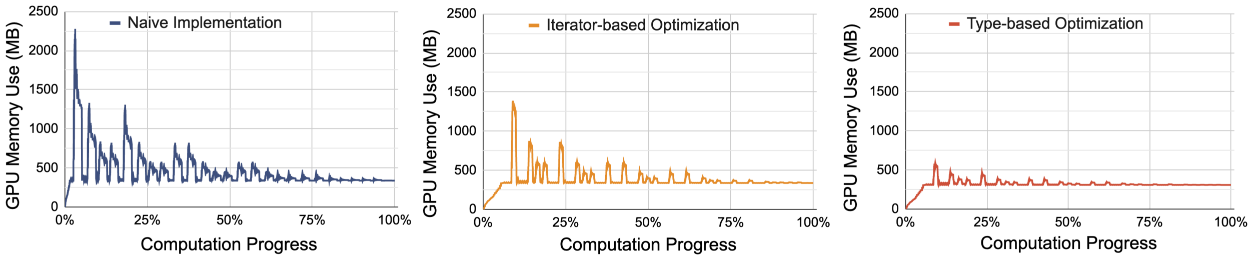


Figure 3.6: Computation and communication overhead for private training iterations in LAN and WAN settings. Piranha significantly accelerates local computation on a GPU, resulting in communication costs dominating overall runtime as latency between parties and network size increases.

used by the network is not selected carefully, parameter update gradients will approach the minimum value Piranha can represent, yielding imprecise results and barring the model from training correctly. Figure 3.5 quantifies what precision is necessary to train each network, showing the final test accuracy after 10-epoch P-Falcon training runs at increasing amounts of precision from 10 to 26 bits. There is a clear distinction between precisions at which each network fails to train and those that allow the networks to do better than random guessing. While 13 bits of precision are sufficient for private inference, even SecureML cannot begin to train until more than 14 bits of precision are used. For the deeper networks, AlexNet and VGG16, which see very small gradients by the end of backpropagation, a higher precision (at least 22 and 24 bits, respectively) is needed. More importantly, these results indicate that computation over 64-bit integers is not just desirable for secure training of large networks but in fact, necessary. In addition, as the size and depth of MPC-trained models increases, the amount of fixed-point precision necessary will likely grow as well, or the use of adaptive fixed-point computation may be necessary.

3.7.6 Computation and Communication Cost

We measure Piranha's ratio of computation time (time spent performing GPU-accelerated local computation) to network overhead (time spent waiting for other parties) in Figure 3.6 for both LAN and WAN settings. In the LAN setting, all parties executed on GPUs in the same datacenter, with approximately 1.5 ms of observed latency, while in the WAN setting, we run the parties in datacenters in different geographic locations with 60ms of latency in between. When the network is fast, so is the end-to-end runtime: Piranha completes training iterations over each network architecture in ∼3 seconds or less over LAN but takes up to 40 seconds over WAN to perform a 4-party training iteration for VGG. We note that the raw time spent on local computation is the same in both settings, but the computation-communication ratio is very different. We observed that parties in the LAN setting spent



(a) (b) (c)

Figure 3.7: Memory footprint over a VGG16 forward pass. Each point is a snapshot of the total GPU memory allocation (in MB) at each memory operation (allocation or de-allocation). Figure 3.7a corresponds to a naive GPU implementation, Figure 3.7b measures the footprint after iterator-based optimizations, and Figure 3.7c after efficiently sizing bit-containing data structures.

between 15% and 60% of the time on compute (on Secure ML and VGG16, respectively), while in comparison, parties in the WAN setting never spent more than 6% of their time on computation. Piranha inherits its communication behavior from the protocol that it is accelerating, and so it does not fundamentally alter the network overhead that would be observed. It is likely that future protocols performing increased computation in favor of minimizing communication [219, 217] would see a large benefit from executing on Piranha in a WAN setting.

3.7.7 Memory Efficiency

Commodity GPUs, including those we use to evaluate Piranha, are commonly constrained to 16GB of memory. We evaluate how effectively Piranha manages this memory constraint by tracking peak memory usage over training passes. When all other parameters are the same (protocol, computational task, and GPU hardware), prior work can only execute over batch sizes of 32 [203]. This section shows how careful memory management directly translates to executing neural network training over significantly larger batch sizes on a single GPU than has been previously possible.

We illustrate the benefits of two main memory-based modifications we discussed in Section 3.5.2 in reducing our memory footprint. We consider three Piranha versions: (1) a naive computation approach with large uniform data types and minimal in-place computation, (2) an iterator-based implementation that seeks to avoid memory allocation when at all possible, and (3) a version that correctly sizes data types to minimize wasted memory (e.g. in the case of secret-shared bits). For each version, Figure 3.7 tracks on-GPU memory usage for P-Falcon, updated after every (de)allocation, during a VGG16 forward pass with an input batch size of 4. We also measure the maximum VGG16 batch size that Piranha can support with on-GPU memory, and total runtime and peak memory footprint with a batch size of

Network (Dataset)	k	Memory usage for Private Training (MB)		
		P-SecureML	P-Falcon	P-FantasticFour
SecureML (MNIST)	1	319	325	331
	64	321	327	335
	128	325	331	339
LeNet (MNIST)	1	437	461	481
	64	535	577	651
	128	661	749	897
AlexNet (CIFAR10)	1	507	603	675
	64	531	649	743
	128	585	689	805
VGG16 (CIFAR10)	1	629	847	1027
	64	3017	3927	5481
	128	5505	7207	10197

Table 3.4: The maximum memory usage of a secure training pass (forward and backward pass) for various MPC protocols and network architectures. Piranha's memory efficient design enables running large networks such as VGG16 with a batch size of 128 where prior works have been limited to 32 [203].

32 to compare between versions. Peak memory footprint indicates the amount of temporary allocations necessary at runtime, which can significantly strain the GPU's available memory and preclude larger batch sizes.

Figure 3.7a shows the memory allocation trace for the naive P-Falcon implementation described in Figure 3.2(a), which requires a significant amount of data allocation while executing ReLU comparisons, where secret-shared values are expanded into bitwise format. Driven by the initial network layers with larger inputs, the peak GPU memory load is 2.28 GB, a 7× increase over the allocation required for the network itself (345 MB). The total number of memory operations is high: almost 16,000 such allocations and frees are performed over the course of the computation. During a 32 batch size run, this approach can complete a training iteration in 27 seconds with a peak memory footprint of 14.9 GB, or 93% of available GPU memory.

Figure 3.7b shows the results of an improved iterator-based implementation that operates over views of already-allocated shares, without incurring additional memory load. In-place computation yields significant memory savings: for batches of 4 images, the iterator-based Piranha version requires only 1.38 GB at its peak compared to the base implementation of Figure 3.7a. The number of GPU memory operations also drops, resulting in almost 4× less allocations and frees during the network's inference pass. However, even with these optimizations, the measured peak memory usage of over 1 GB in Figure 3.7b would not support training runs over 128-image batches. Similar to the naive implementation, the maximum batch size the iterator-based version can train with is 32, but only incurs a maximum memory footprint of 8.9 GB, an approximately 60% improvement. Execution time increases slightly to 35 seconds per pass, which we suspect is due to the inherent cost of non-contiguous and indirect memory access.

In Figure 3.7c, we evaluate the impact of sizing memory appropriately for data at the protocol layer. In the previous versions analyzed above, the bitwise expansion used in our ReLU comparison protocol remained a major source of memory blowup, as bit values were each stored into a full 64-bit values. Modifying Piranha protocols to closely match the size of allocated values with their logical sizes significantly cuts the peak memory usage in Figure 3.7c by a factor of 2, to 581 MB, or only 250 MB above the baseline model memory requirements. This has an outsized effect on training execution time, as smaller data types require less communication overall: with this change, Piranha can support P-Falcon-based training iterations with a batch size of 256 in just 7.6 seconds, with a maximum memory footprint of 1.8 GB.

Finally, Table 3.4 shows peak GPU memory usage for Piranha over all networks as it performs training passes using the protocols we implemented on Piranha. For SecureML in particular, the baseline memory used by the network parameters dominates any temporary memory requirements, as the peak memory use only grows by 6 MB between runs over batches of 1 image and 128 images. As expected, P-FantasticFour exhibits larger increases in peak memory use as batch size increases, due to the increased number of local shares it must maintain for each secret-shared value, proportionally increasing memory load.

3.7.8 Comparison with Prior Work

Finally, we compare the runtime and communication overhead of Piranha relative to state-of-the-art protocols for neural network training: a CPU-based implementation, Falcon [223], and a GPU-based implementation, CryptGPU [203]. Both protocols are fixed to a 3-party setting, while Piranha is designed to support a general class of LSSS protocols. In this section, we compare the performance of existing protocols with Piranha's equivalent 3-party P-Falcon implementation, to evaluate whether the generality of Piranha's design comes at a performance cost.

We benchmark the run-time for a *single* training and inference pass over 3 different networks – LeNet, AlexNet, and VGG16. While we can support batch sizes of up to 128 on each of these networks, we scale down our computation to provide an apples to apples comparison with prior work. The results are presented in Table 3.5.

For private inference, where the forward passes use a single input image (batch size of 1), the computation is not large enough to fully benefit from GPU acceleration. Table 3.5 shows that Piranha achieves comparable performance to the CPU-based Falcon for private inference over small networks, but over the much larger VGG16 architecture, Piranha already yields a 3× performance improvement.

GPU acceleration has a much stronger impact on private training iterations, where the computation sizes are much larger due to the increased batch size and the addition of a

	Model (Dataset)	Private Inference			Private Training		
		Falcon	CryptGPU	**P-Falcon**	Falcon	CryptGPU	**P-Falcon**
Time (s)	LeNet (MNIST)	0.038	0.380	**0.031**	14.9	2.21	**0.888**
	AlexNet (CIFAR10)	0.110	0.910	**0.131**	62.37	2.910	**1.419**
	VGG16 (CIFAR10)	1.440	2.140	**0.469**	360.83	12.140	**7.473**
Comm. (GB)	LeNet (MNIST)	2.29	3	**2.492**	0.346	1.14	**0.417**
	AlexNet (CIFAR10)	4.02	2.43	**1.960**	0.621	1.37	**0.581**
	VGG16 (CIFAR10)	40.05	56.2	**88.39**	1.78	7.55	**4.261**

Table 3.5: We compare the run-times for private training and inference of various network architectures with prior state-of-the-art works over CPU and GPU. Falcon and CryptGPU values are sourced from [203] Table I. Private inference uses batch size of 1, training uses 128 for LeNet, AlexNet and 32 for VGG16. For smaller computations (private inference), Piranha provides comparable performance to CPU-based protocols. However, for larger computations (private training), Piranha shows consistent improvement between $16 - 48\times$, a factor that improves with scale.

backward pass over the network. Even on the smallest architecture, LeNet, Piranha performs training iterations 16× faster by leveraging a GPU, while on the larger architectures we benchmark, we show between a 44-48× speedup.

In addition to evaluating the benefits of GPU acceleration, Table 3.5 also quantifies whether Piranha incurs additional overhead from supporting multiple protocol implementations, compared to tools that integrate a specific MPC protocol end-to-end like CryptGPU [53]'s 3-party implementation. Considering private inference, Piranha is significantly faster, showing approximately 12, 7, and 4× speedup on each of LeNet, AlexNet, and VGG16, respectively. We also show a performance advantage in computing training iterations, with performance gains ranging from approximately 2.5× on LeNet to 1.6× on VGG16. We attribute these constant improvements to a few factors. First, Piranha's direct use of 64-bit integer kernels avoids the repeated 16-bit floating point multiplications that CryptGPU incurs. We do this at the cost of using less powerful GPU integer cores and kernel implementations that must be emulated with 32-bit integer instructions. Second, even though Piranha supports many different protocol implementations, Table 3.5 shows that the negligible overhead of our approach can yield the same or better performance than single-protocol designs. Third, some portion of these performance difference may be attributable to different programming environments – Piranha is implemented in C++ while CryptGPU is implemented over PyTorch.

3.8 Future and Subsequent Work

3.8.1 Opportunities for Future Work

We show in Section 3.7 that GPUs provide much-needed performance acceleration for secure computation. Piranha's modular platform structure means that functional enhancements made at any layer of the platform – from future performance improvements in the GPU kernels to additional MPC protocols or new privacy-preserving applications – can immediately benefit other system components.

Device Layer

The device layer separates protocols from the GPU interface. Thus, acceleration of local operations, optimizations, or entirely different methods of performing integer- and fixed point-based calculations can be independently developed. Even in its current state, Piranha's integer kernels are slower than their floating-point equivalents implemented by popular libraries like cuBLAS [54], as they can take advantage of features like tensor cores that focus exclusively on floating-point. Future efforts can focus on supporting better kernels, enabling multi-GPU usage, and supporting custom accelerators on platforms such as FPGAs [72].

Protocol Layer

Piranha can be used for development of newer multi-party protocols, expanding support for different number of parties, innovative protocols, and adversarial models. As noted in Section 3.1, we focus on LSSS protocols in a semi-honest security model, and the protocols we implement operate over 32- and 64-bit integer rings, such that the existing hardware support for modular arithmetic simplifies computational overhead. However, support for other protocol types can be expanded, in supporting field operations, accelerating garbled circuit evaluation [233], or adding homomorphic encryption support [56] to enable dishonest-majority protocols.

Application Layer

We showcase the use of Piranha for making meaningful progress on private neural networks training. Piranha's modular approach provides a rich environment for innovation in MPC-friendly neural network design, such as private training of newer architectures like residual networks, transformers, or LSTMs. While we only evaluate Piranha over a neural network training application, the platform allows development of arbitrary, protocol-agnostic secure computation. Future work can focus on demonstrating the ability of the platform to support applications in other areas, such as oblivious sorting or oblivious RAMs.

3.8.2 Subsequent Work after Piranha

Research in secure ML training and inference has exploded in recent years [238, 150, 164], as more and more personal data is being used to train larger and larger models. Since Piranha's publication, critical advances in several key areas have been made to move secure MPC-based machine learning even closer to practical deployment.

Improvements in Communication

Looking forward, one of the key takeaways from 3.7.6 was that in terms of training on a local- or wide-area network, Piranha is computing as fast as it can, yet it cannot change the communication overhead inherent in the protocols it supports. Meteor [70] proposed better 3-party protocols for the primitives we use – addition, multiplication, sign-bit extraction for ReLU – with lower communication costs, increasing performance. Going beyond the LSSS regime, Orca [113] and Pika [219] leverage Function Secret-Sharing to *significantly* decrease the communication cost between parties, as shares of the extremely large matrices of weights and activations need not be sent back-and-forth between parties. Instead, much smaller shares of a function to be evaluated can be distributed among participants. Compared to Piranha, Orca evaluates NN training between 8 and 100 times faster, which pushes secure learning significantly closer to plaintext.

Specialization for Large Language Models

Since Piranha, machine learning has discovered the benefits and scale of Large Language Models (LLMs) and significant effort has been put towards enabling secure evaluation of LLMs using personal data. CipherGPT [103] proposed new optimizations based on the specific transformer architecture used in LLMs, including a secure protocol for evaluating the GELU activation function required. PUMA [71] implemented secure protocols for, among others, embeddings, LayerNorm, and GELU, showing inference of 1 token using Llama-7b in 5 minutes, while Sigma [93] goes even further, using an FSS-based approach to evaluate Llama2 in less than a minute.

3.9 Summary

In this chapter, we described Piranha, our platform for GPU-accelerated MPC protocol development. Piranha contributes three modular components: a device layer that manages protocol memory on the GPU and accelerates MPC-specific integer operations, a protocol layer where memory-efficient in-place operations can be leveraged to fit the constrained GPU environment, and an application layer for privacy-preserving computation on any underlying protocol. Piranha's modular structure provides wide applicability for other projects to use GPU acceleration without requiring expert knowledge. To demonstrate that Piranha as a general-purpose platform provides significant improvements in run-time through GPU-based acceleration, we implement 3 different MPC protocols for secure training of neural networks on top of Piranha, resulting in a 16-48× performance improvement over CPU-based implementations. Finally, using Piranha, we are able to securely train a realistic neural network end-to-end, with over 100 million parameters, in a little over a day.

This chapter has demonstrated that many heavyweight cryptographic primitives like MPC require a significant amount of computation, which many less capable clients (e.g. mobile devices or personal laptops) might find infeasible. In such cases, one would ideally outsource the actual computation to a third party with more computational resources, but there is often no guarantee that the computation was performed correctly. In the next chapter, we describe and evaluate system improvements for zero-knowledge proving on the GPU, which would allow a compute provider to prove it had correctly generated the desired output without forcing a client to check the computation step-by-step.

Chapter 4

Towards High-Throughput Zero-Knowledge Proving on GPUs

4.1 Introduction

Zero-knowledge (ZK) proofs are incredibly useful for verifying that private computation was done correctly, impacting areas from blockchains (verifying that a transaction is valid [194] to anonymous ID systems (verifying that someone has the right to vote [241] or access a particular resource [190]), maliciously-secure cryptographic protocols [110], or even nuclear disarmament [177].

Unfortunately, while verifying proofs tends to be very simple and fast for clients, *generating* a proof is much more time-consuming, both in absolute terms and relative to the amount of plaintext computation that is being proven. As an example, we implemented a proof using the popular Gnark [29] library for ZKPs, attesting to 128 correct EcDSA signatures in the Plonk proof system. The proof required about 21 seconds of online time to generate, whereas simply verifying each signature if accessible directly would conservatively require less than a millisecond [36].

Proving in ZK-SNARKs (Succinct Non-Interactive Arguments of Knowledge) [194], the most well-known flavor of zero-knowledge proving in which a verifier can check a proof it receives without interaction with the prover, is bottlenecked by several high-impact, expensive operations. Chief among them is the Multi Scalar Multiplication (MSM), used to commit to large polynomials during the proving process. In addition, the Number Theoretic Transform (NTT) is used to convert elliptic curve polynomials from their coefficient to point-based forms, and back again. By far the most execution time is spent in the MSM, contributing up to 70-85% of the total runtime [146].

This clear and critical bottleneck in the proving process makes MSM acceleration a key target for acceleration on specialized hardware, including FPGAs [229, 239, 186] and GPUs [244, 106, 147, 165]. Given that zero-knowledge proofs enable private cryptocurrency systems [194] that do not leak the participants or amount associated with each transaction, allow powerful computers to efficiently process blockchain transactions offline while checking correctness, enable embedded devices to access firmware updates without revealing their identity, or support anonymous voting systems [236], there are significant industry-based efforts to improve the practical efficiency of MSM and end-to-end proving performance, yielding competitions like as the Zprize contest [245] where prizes worth thousands of dollars are awarded to teams that can achieve performance benefits over the currently-known state-of-the-art.

Critically, in SNARKs, the prover does not have to communicate with the verifier to generate the proof, so it can put whatever computational resources it has available to the task (this is markedly different from the acceleration environment for Piranha in Chapter 3, in which the GPUs from different parties had to communicate during the protocol). In addition, MSMs are large in size (proofs typically consist of millions of constraints or more). They are also high in number, with many different proofs of the same structure being computed at the same time. For example, a single ZK proof might seek to attest to the state of an entire microarchitecture during an execution cycle, or hundreds of individual proofs might be generated to attest that various blockchain transactions were generated correctly.

In this chapter, we tackle two related problems. For single, very large MSMs, how can we leverage GPUs in a more effective way to decrease proof generation latency? This is critical for large proofs, such as those involved in "rollups" of zero-knowledge proofs, whose constraints (and thus proving effort) increases with the number of inputs. And, for many proofs, how can we increase throughput by batching MSM computations together? In this setting, systems such as blockchains that must verify a stream of smaller proofs for e.g. private transactions as they are submitted by clients.

4.1.1 Challenge: Decreasing Latency for Large MSM Problems

Handling large MSM problems is a question of immense vertical scale. As the computations we need to prove get more complex, so do MSMs. In ZPrize [245], a ZKP acceleration challenge in 2023, contestants were challenged with accelerating MSM computation for input sizes of up to $N = 2^{26}$, or 67 million inputs, equivalent to over 8 GB of data. Given limited GPU memory, increasing the size of MSMs we can support by 4, 8, or 16 times will not be as easy as allocating a larger buffer, as the data will simply not fit on a single device at one time.

Thus, the systems challenge we tackle is determining how to efficiently expand MSM memory use into CPU memory without incurring overhead or increasing the proof latency. In Section 4.3, we discuss our chunking-based approach to large MSM computation, in which we note that the amount of memory required for temporary state remains constant independent of the input size. Once this state is allocated at the beginning of the computation, we divide the input into a set of independent chunks that are streamed to the GPU in sequence while computation over the previous chunk is ongoing. We show in Section 4.5 that we can successfully chunk large MSMs ($N > 2^{26}$) and schedule their data transfer such that computation on the GPU is not interrupted, maximizing utilization.

4.1.2 Challenge: Increasing Throughput for MSM Computation

Likewise, supporting large batches of MSMs is a problem with large horizontal scale. Given a workload of tens or hundreds of proofs at a time, we must find a way to increase overall throughput by efficiently processing *batches* of MSM problems. Unfortunately, the common approach to MSM evaluation, Pippenger's algorithm (see Section 4.2), requires a significant amount of additional memory allocation per-proof, consuming much of what is available on-device. At the same time, each individual proof shares a common structure, in that they all reference the same elliptic curve points as inputs to the MSM, with only the scalar multipliers depending on the specific proof input.

The primary challenge in this setting is determining a strategy to reuse work amongst the batch execution, given known properties of each MSM. Since we need to effectively multiply the same base point by a large set of different psuedo-random scalars, we can leverage an efficient addition sequence [49, 234] to calculate each intermediary target multiplication as a step in the process towards computing the final target multiplication. Our insight is that, unlike prior work in which addition chains are generated at compile-time for known exponents, we generate addition chains at *runtime* for scalar sequences that are not known a-priori. In this setting, we prioritize achieving relatively short chains over perfectly optimal ones, generated with minimal processing overhead. Finally, we demonstrate a split CPU-GPU design where instructions are generated for each addition sequences on the host before being executed on the accelerator; we demonstrate in Section 4.5 that leveraging addition sequences can make a significant impact to batch MSM processing compared to the baseline.

4.2 Background and Related Work

4.2.1 Zero-knowledge Proofs

Zero-knowledge proof systems generally share the same objective: a *prover* seeks to convince a *verifier* of the truth of a particular statement, without revealing the information that makes it true. More specifically, the prover seeks to generate a valid proof π using a secret witness w and public input x, such that the verifier can check π using x but without needing w.

Zero-knowledge protocols must be *complete*, in that an honest verifier will always be able to verify a π from an honest prover; *sound*, in that an honest verifier will never be fooled by an incorrect π from a dishonest prover; and eponymously, *zero-knowledge*, in that nothing other than the correctness of π is leaked to a malicious verifier. *Interactive* proof systems require some rounds of communication back-and-forth between the prover and verifier, while *non-interactive* proofs allow a verifier to check π immediately once it's been received. *zk-SNARKs* (Zero-Knowledge Succinct Non-interactive Arguments of Knowledge) are non-interactive proofs that are also very short and quick to verify: several hundred bytes no matter the complexity of the operation being proved. zk-SNARKs have a setup phase where a Common Reference String (CRS) is generated from some source of randomness with publicly-known parameters used by the prover and the verifier. Importantly, if the setup is insecure and the randomness underlying the CRS is known, a malicious party can forge proofs under the CRS that will fool even honest verifiers.

There are two commonly-used zk-SNARK proof systems: Groth16 [92] and Plonk [84]. Both proof systems decompose arbitrary proof statements into arithmetic circuits consisting of addition and multiplication gates, then encoding circuit values for a particular input into polynomial equations over elliptic curve points that can be quickly verified. Groth16 requires a circuit-specific trusted setup, so each proof application must generate their own CRS and ensure that the randomness used in that process remains secure from malicious parties. On the other hand, Plonk provides a universal setup, with any number of different proofs sharing the same CRS (this setup is usually performed under strict scrutiny and using other tools like multi-party computation [127]). The convenience of Plonk comes at the cost of slightly increased proof sizes and additional proving time required to generate valid proofs.

4.2.2 Multi-scalar Multiplication

An extremely common operation in the generation of a ZK proof is multiplying elliptic curve points in the CRS by various input-dependent scalar values and obtaining their sum. Given n points P_i in the CRS and a set of n scalars s_i, the multi-scalar multiplication (MSM) result P is

$$P = \sum_{i=1}^{N} s_i P_i$$

This seemingly simple operation consumes almost 80% of the total proving time using Groth16 [146]. Elliptic curve scalar multiplication is very expensive, requiring repeated point addition and doubling. In addition, the size of each MSM may be very large – for example, PipeZK [237] evaluates over MSMs ranging from $N = 2^{14}$ to 2^{20}, over a million elements.

4.2.3 Pippenger's Algorithm

Given the bottleneck that MSMs represent to efficient ZK proving, numerous efforts have been made to reduce its computational cost. Most current efficient MSM implementations today use Pippenger's algorithm [25], which aims to reduce the overall number of scalar multiplications required, at the cost of additional additions.

In Pippenger's algorithm, each B-bit scalar is split into c-bit windows. For each window, the associated elliptic curve points are accumulated into one of 2^c buckets based on window value. Accumulated bucket points, rather than individual input points, are multiplied by their bucket value, then summed and adjusted for window position to achieve the final result. For example, using the BLS12-381 curve, with $B = 256$ bit scalars, one might split each scalar into 16 16-bit windows but could pick some other c. As the window size c increases, so does the number of buckets needed and and required multiplications, but small c will increase the count of bucket sets and number of times each point must be added into a bucket. It can also decrease parallelism, as a limited number of buckets can be aggregated in parallel, compared to wider c.

4.2.4 Addition Sequences

For a target set of integers $T = (t_1, t_2, \cdots t_n)$, an addition sequence $S = (s_1, s_2, \cdots s_m)$ such that every t in T is also in S, and for every $1 < i \leq m$, $s_i = s_j + s_k, j, k < i$ [49]. For example, if $T = (5, 8, 10)$, a valid addition sequence S for T would be $(1, 2, 3, 5, 8, 10)$.

An addition chain is very similar to an addition sequence, but in this case, there is only one target value. In fact, generating addition *sequences* is commonly used as a subroutine towards achieving an addition *chain*, where the desired target is decomposed into an intermediary sequence for which an addition sequence is found [24]. Addition chains and sequences are well-known in other cryptographic areas: significant manual effort has been expended on finding the minimal-length addition chain for elliptic curve field inversion [206], where the same exponent is constantly evaluated at runtime and so offline optimization of the addition program yields large benefit.

4.2.5 Related Work on GPU Acceleration for Zero-knowledge

Given the excitement surrounding zero-knowledge proofs in both industry and academia and the massive bottleneck that MSMs represent for proof processing, it's no surprise that a significant work has recently been done to support acceleration of the core ZK proving primitives.

Zhu *et al.* [244] this year proposed a dynamic method to perform pre-processing for Pippenger's algorithm, based on the hardware configuration being used, leveraging the tradeoff between storage space and runtime reduction. Similarly, Huang, Zheng, and Zhu [106] investigate using zero-copy memory using pinned CPU memory to improve memory throughput in ZK proving (likewise, our implementations in this paper also leverage pinned memory to speed memory transfer time significantly). Other research groups have delved into better supporting core operations deep in the GPU hardware, verifying proper cache access patterns and grouping tasks of a similar size [147], or parallelizing the basic elliptic curve modular multiplication that lies at the core of MSM computation [165].

From a systems angle, Ji *et al.* have developed ways to leverage multiple GPUs for Pippenger acceleration, including implementing register-efficient elliptic curve kernels more suited to GPU architectures and leveraging tensor cores [116]. With parallel device execution comes the need to study synchronizing and load-balancing proving workloads [46]. In addition, better abstraction layers for ZK application developers are being developed, hiding implementation details over multiple hardware platforms, like GPUs and FPGAs [109].

Finally, many systems are being built for FPGAs (Field Programmable Gate Arrays), rather than GPUs. While less flexible and more resistant to easy application development, FPGAs offer performance closer to that of an application-specific integrated circuit (ASIC). Xavier *et al.* [229] developed PipeMSM, an FPGA-based efficient modular multiplication technique designed to quickly move data through the device, upon which BSTMSM improves in terms of latency using a different circuit approach [239]. Similarly, Zhao *et al.* [240] propose a pipelined FPGA acceleration implementation for NTT computation. Hardcaml MSM [186] performs precomputation while accelerating MSM computation and leverages CPU resources as well. Finally, PipeZK proposes an actual ASIC implementation for end-to-end ZK proving [237].

4.3 Unbounded-Size MSM Evaluation with Memory Pipelining

In this section, we describe our system design to ameliorate the impact of limited GPU memory on our ability to compute over large MSM problem sizes. While GPU memory availability is always a concern, it is much more acute when computing large MSMs. Each BLS12-381 base element requires 128 bytes of memory, while scalars occupy another 32 bytes. Allocating temporary memory (e.g. Pippenger buckets), also reduces the supportable problem sizes. Thus, the question is, how can we expand MSM execution to problem sizes that may not fit into memory at all?

4.3.1 Large MSMs using Unified Memory

A natural approach is to somehow integrate a (relatively) larger amount of CPU-based host memory with GPU device memory to store larger amounts of the MSM. We can very easily and cleanly use NVIDIA Unified Memory [97] with our GPU MSM-based application by changing `cudaMalloc` calls in the code setup to `cudaMallocManaged` invocations. This allows us to store our MSM inputs split across all the GPU and CPU memory we can access, increasing our input size appropriately.

Figure 4.1 shows evaluation times for MSMs using an unmodified Yrrid MSM GPU implementation [235] and one modified to use managed memory allocation and the GPU's unified memory features. As an MSM is additive, we can also imagine a scenario in which we would take a large MSM and compute entirely separate MSM results for smaller chunks, aggregating at the end. To measure the overhead in processing time that this would incur, we test the unmodified and unified memory implementations on MSMs split into 1 to 4 chunks.

Overall, we can see that unified memory significantly increases our ability to process large MSMs – the unmodified library runs out of memory at $N = 2^{26}$, while unified memory can support 8 times larger MSM sizes at $N = 2^{29}$. However, this comes at a small overhead cost compared to the device memory-only version of the library. Overhead is introduced by the CUDA driver when it has to page memory from the CPU to the GPU or vice versa, stalling MSM computation and increasing end-to-end latency. Finally, we observe that simply chunking an MSM into several smaller MSM instances is not an effective memory strategy in that it increases the runtime significantly. For $N = 2^{28}$, an MSM chunked into 4 segments

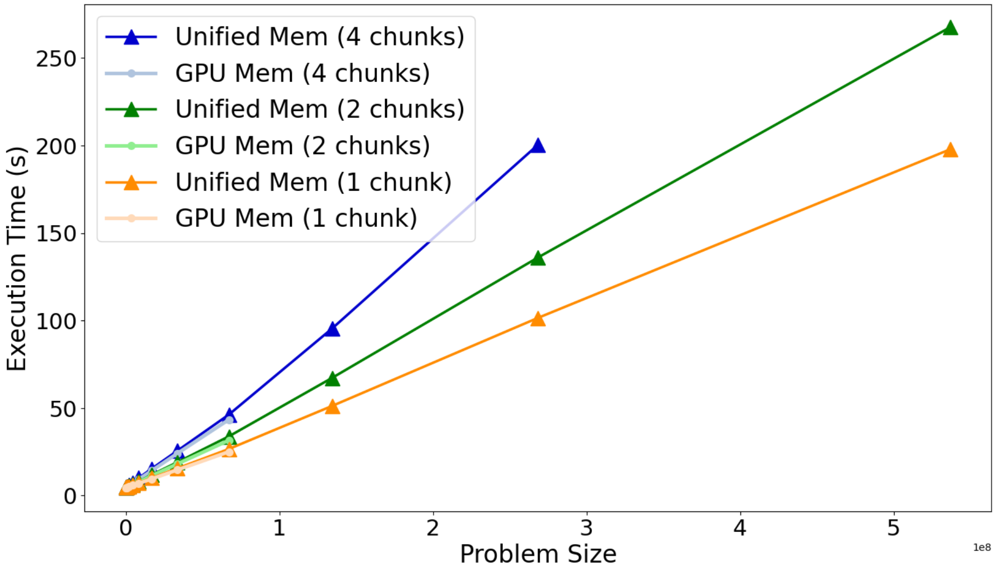


Figure 4.1: MSM execution time as problem size increases, based on a sequential chunking strategy. For a given MSM size N, we can choose to split the MSM evaluation into a number of independent evaluations (chunks). Using only GPU memory, we quickly run out of memory. Alternatively, unified memory allows MSMs to scale to much higher size, but at a small overhead compared to using solely device memory.

requires 200 seconds to complete evaluation, while a 1-chunk MSM completes in only 101 seconds.

4.3.2 Chunking MSM Execution to Hide Memory Loads

Rather than rely on the CUDA runtime to detect page faults and perform memory transfers when a particular portion of MSM data is needed on the GPU, we argue that the structure of Pippenger's algorithm allows us to schedule memory loads in a way that completely overlaps with computation. On Nvidia GPUs, one memory copy in either direction (host CPU → device or device → host CPU) can occur concurrently over the PCIe bus with kernel execution.

Figure 4.2 shows a high-level overview of the window-based Pippenger algorithm operation, with two primary phases, as described in Section 4.2. In the first, we accumulate points into a fixed number of buckets based on scalar bit windows, and in the second, these buckets are multiplied to get the final result.

Importantly, we note that no matter the size N of the input MSM, the number of buckets will always remain the same. As long as the buckets can fit in GPU memory, we hypothesize that we can split the input into a smaller series of chunks, not unlike what we did when testing unified memory above. However, this time, we will stream the chunks to the GPU

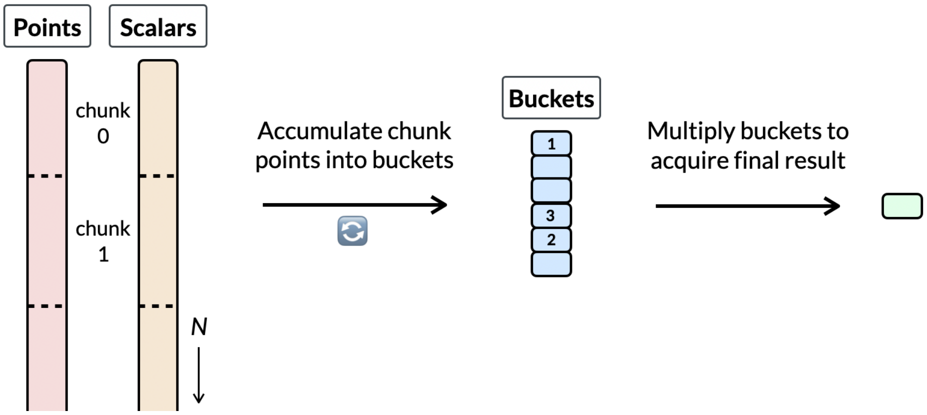

Points
Scalars
chunk 0
chunk 1
N
Accumulate chunk points into buckets
Buckets
1
3
2
Multiply buckets to acquire final result

Figure 4.2: High-level Pippenger operation with chunking strategy. We stream chunks of input point/scalar pairs into the GPU for bucket aggregation, then multiply buckets once after all chunks have been processed.

for insertion in the same set of buckets, thus avoiding much of the overhead of entirely separate MSM evaluations. In addition, we expect that streaming chunks will improve overall execution time because the accumulation kernel can continue processing as more chunks are loaded into GPU memory.

We implement this chunking strategy by modifying the ICICLE GPU ZK acceleration library [108]. The core bucket accumulation kernel and data transfer logic operate on two, separate streams with some synchronization between rounds. Input data loading and computation operates on a two-segment memory buffer, where even-numbered blocks load data into input segment 0 and odd-numbered block into segment 1. If all data for the next chunk is loaded before computation on the previous chunk completes, the data stream waits for the current block to complete before replacing its input data with the next expected chunk from the CPU. In practice, the bottleneck operation is bucket accumulation, so data loads have plenty of time to complete; an example of a load-and-compute timeline for our MSM can be found in Figure 4.5.

We evaluate the effect of chunking MSMs on overall latency and our ability to evaluate MSMs that cannot fit into the GPU memory limit in Section 4.5.

4.4 Accelerated Batch MSM Evaluation with Addition Chains

Batch MSM computation typically takes as input a single set of N base points, based on the proof system's CRS, and b sets of N exponent scalar values, the proceeds to evaluate b parallel Pippenger instances, one for each set of scalars. This has the notable benefit of minimizing the memory footprint of the base points across all MSMs in the batch. However,

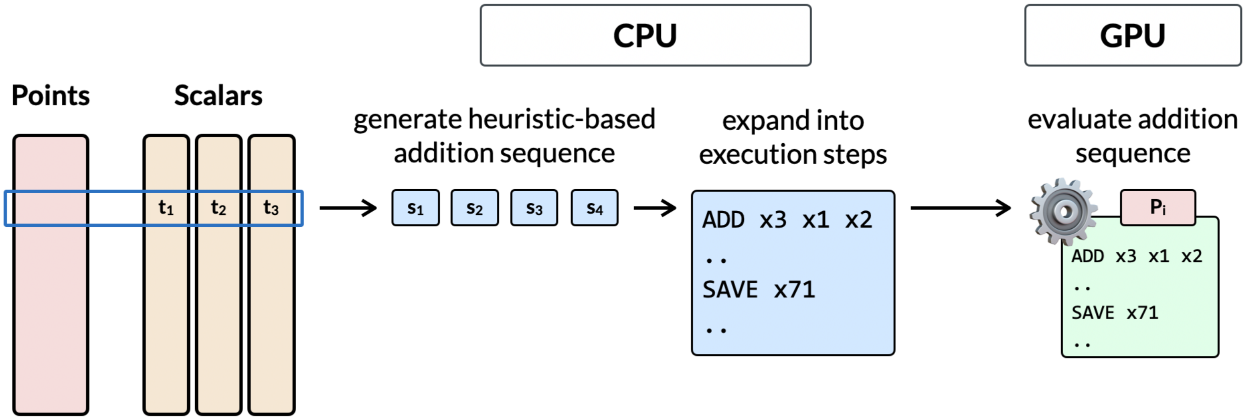

CPU
GPU
Points
Scalars
t₁
t₂
t₃
generate heuristic-based addition sequence
s₁
s₂
s₃
s₄
expand into execution steps
ADD x3 x1 x2
..
SAVE x71
..
evaluate addition sequence
Pᵢ
ADD x3 x1 x2
..
SAVE x71
..

Figure 4.3: CPU-GPU hybrid system to generate and evaluate addition sequences. Scalars from a batch of MSM inputs form targets for sequence generation (CPU), which are converted to program instructions that are interpreted by a an execution kernel to multiply the associated base point (GPU).

for large batches, this appears wasteful, as the Pippenger instances will eventually multiply the same base b times in some bucket accumulation step.

We propose that there is a more efficient way to generate the scalar multiplication results for many instances of the same base than Pippenger, using addition sequences (see Section 4.2). Naive double-and-add, in comparison to Pippenger, of the bases is not efficient for single MSMs. But in a batch setting, addition sequences could yield much shorter paths to achieve the correct result, amortized by the number of additional multiplications we happen to compute on the way to the end of the sequence. We show in Section 4.5.4 and Section 4.5.5 that this approach yields better performance over large batch sizes than our baseline.

We implemented a C++-based addition sequence generator and program generator for CPU based on the `addchain` library [153], and a program evaluation kernel in CUDA for the GPU. First, we take horizontal slices through our MSM inputs (e.g. $s_{i,0}$ for i in $[0, b)$, then $s_{i,1}$, etc.) as individual target values for sequence generation. Figure 4.3 shows each stage of the system, from sequence generation on the CPU to GPU-accelerated program execution.

We search for a sequence using the Delta Largest heuristic proposed by Bos and Coster [28]. Over a sorted sequence of target scalars, the heuristic continually adds the largest scalar to the output sequence, working backwards. At each step, the heuristic sorts the difference (or delta) to the next-largest target back into the target list. We use the `std::set` typically backed by a red-black tree to ensure logarithmic insertions in sorted order.

Once the sequence is generated, we convert it into a “program” of primitive operations, mirroring [153]: Add, Double, Shift (repeated doubling), and Copy, which saves an intermediate addition value in a specified output array index. We then iterate through the sequence looking for two prior computed values that sum to the desired result for that element in the sequence. Repeated scans can be quite slow, scaling poorly with the size of the list. However, given that our sequence is in sorted order and generated using the Delta Largest heuristic, we know that every element in the sequence is the sum of the previous element and some prior element, likely quite close to the current element. As a result, scanning backward from the desired element is quite quick for our use case – we evaluate how fast sequence generation is in Section 4.5.5.

Next, we perform register allocation for temporary values by scanning through the operation list and keeping a map of occupied memory state, then convert the program into 32-bit op codes: an 8-bit operation type, 8-bit destination register, 8-bit source register, and an optional 8-bit second source register. The opcodes, as well as the total number of temporary registers required, are passed to the GPU for execution.

Finally, each GPU thread allocates a local memory buffer on its stack to store temporary results, decodes the instructions, and executes them step by step over the input elliptic curve point. Each time it encounters a Copy instruction, the current point value in the specified register is copied to an output buffer that aggregates each thread's result. Finally, a second kernel aggregates every resulting point for a particular MSM and returns the output to the user. As program generation occurs on the CPU and execution is parallelized on the GPU, a new set of programs to execute might be ready once the kernels exit, and the process can continue as new batches arrive.

4.5 Evaluation

In this section, we evaluate the main concepts from the previous sections – chunked execution for large MSMs and batch evaluation for high-throughput scenarios – in order to answer 4 key evaluation questions:

1. What is the overhead of chunked MSM evaluation, compared to a baseline implementation? (Section 4.5.2)
2. Can chunked evaluation effectively remove the GPU's limit on MSM problem size? (Section 4.5.3)
3. How do addition sequence length and execution times scale as MSM batch size increases? (Section 4.5.4)
4. How does overall MSM evaluation time scale as batch size increases, in comparison to using a Pippenger algorithm? (Section 4.5.5)

4.5.1 Implementation

To answer these questions, we implemented two methods for MSM computation in CUDA/C++: a modified version of the open-source GPU-based library for zero-knowledge acceleration,

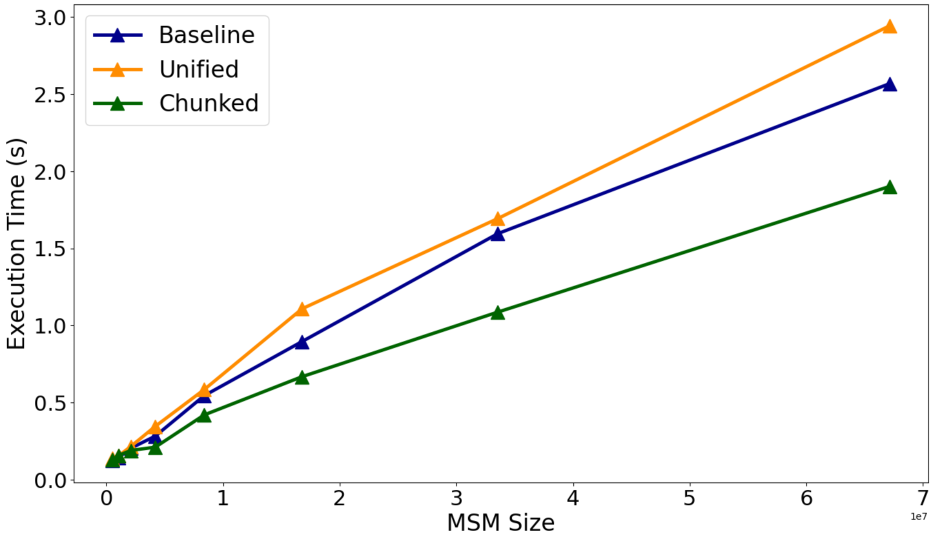


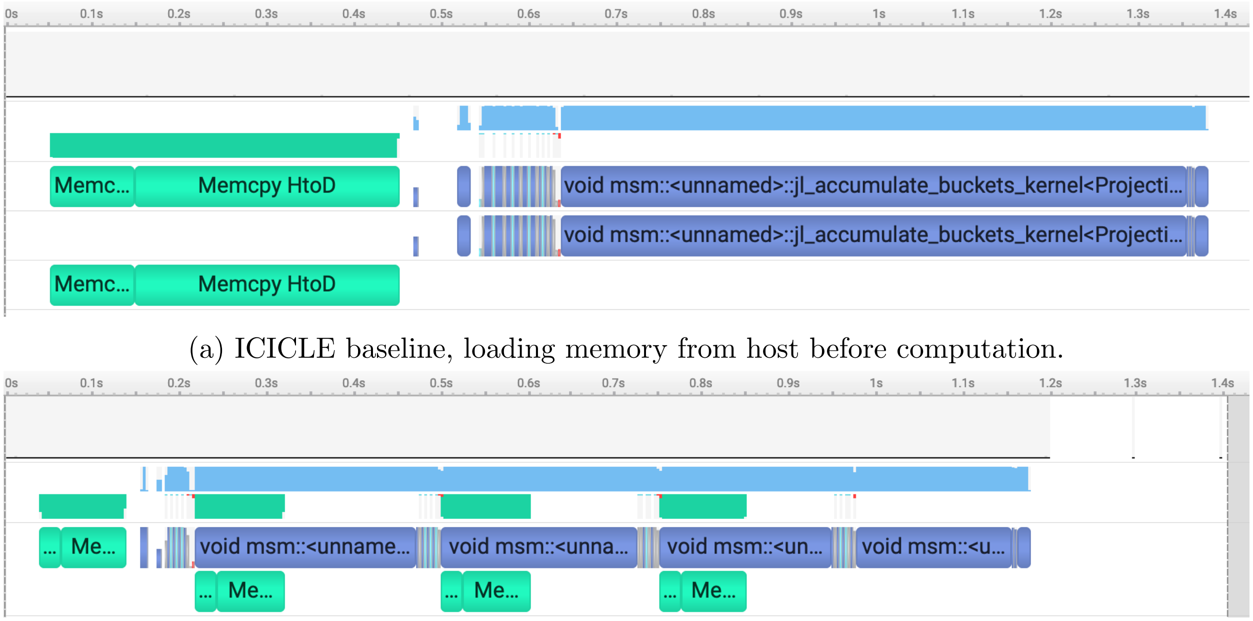
Figure 4.4: MSM execution time as input size increases, for a baseline Pippenger implementation, a slightly-modified implementation loading MSM inputs on both CPU and GPU using Unified Memory, and our *chunked* implementation overlapping memory loads and MSM computation. At large MSM sizes, chunking avoids an overhead over the baseline entirely, showing 25% improvement in runtime performance.

ICICLE [108], forked at commit `36e288c`, and a custom addition sequence generation and GPU execution kernel, based on the `addchain` [153] Golang library. We benchmark both our ICICLE-based implementation with chunking and our addition chain implementation against an unmodified baseline ICICLE version.

We evaluate both systems using a Nvidia V100 GPU with 32GB RAM. On the host side, we use a 80 2.20 GHz Intel(R) Xeon(R) CPU E5-2698 v4 core machine. We evaluate MSMs using randomly-generated elliptic curve points in the BLS12-381 curve and 256-bit exponent scalars. In our experiments, we load random MSM inputs from disk before evaluation, reporting online computation time. Addition sequence generation is parallelized across CPU cores, with each thread generating a sequence for one set of target scalar, while *evaluation* of those sequences is performed on the GPU. As a result, both stages can be pipelined to improve proof throughput.

4.5.2 Chunked MSM Overhead

As we saw in Section 4.3.1, naively attempting to increase available GPU memory for larger MSM problem sizes incurs a small but non-trivial amount of overhead compared to MSM problems entirely present in GPU memory. In Figure 4.4, we evaluate the overhead of MSM chunking on overall MSM execution time as MSM sizes increase from 2^{19} to 2^{26} elements, with chunking in green. We compare our chunked runtimes to an unmodified ICICLE baseline



(a) ICICLE baseline, loading memory from host before computation.

(b) Chunked MSM evaluation, with memory loads in 4 chunks while computation is occuring.

Figure 4.5: $N = 2^{25}$ MSM execution timeline, comparing baseline and chunked versions. Green boxes indicate memory copies to/from the GPU using CPU pinned memory, while blue boxes indicate kernel execution. When chunking, the MSM kernels can execute while data loads, reducing end-to-end computation latency.

in blue and that same baseline modified to use unified memory for input MSM points and scalars in orange, which we discussed in Figure 4.4. We chunk input MSMs into 4 chunks for smaller sizes, 8 chunks for 2^{25}, and 16 chunks for 2^{26}-element MSMs.

As opposed to the use of unified memory for larger MSMs, in which we saw a 5-23% overhead associated with paging individual memory accesses back-and-forth between host CPU and device GPU, chunked evaluation does not yield an overhead on MSM computation at large MSM sizes. This is due to the fact that we are overlapping data transfer with computation, thus hiding data access latency. To see this in more detail, we profiled the baseline and chunked MSM executions, which can be seen in Figure 4.5. In the baseline, no MSM kernel is running for more than 0.4s while the desired bases and scalars are copied to the device; this is greater than 25% of the total runtime.

Alternatively, in our chunked implementation, we take only about 0.15s to load the first quarter of the MSM problem and begin computing the Pippenger kernel (each blue `void msm::` block). Meanwhile, the GPU can perform another pair of memory transfers (each green `Memcpy` block) in parallel to kernel execution, and thus immediately begin processing the next chunk when the first kernel has finished. Note that since bucket accumulation (i.e. elliptic curve addition) is our compute bottleneck, we have plenty of time to load each

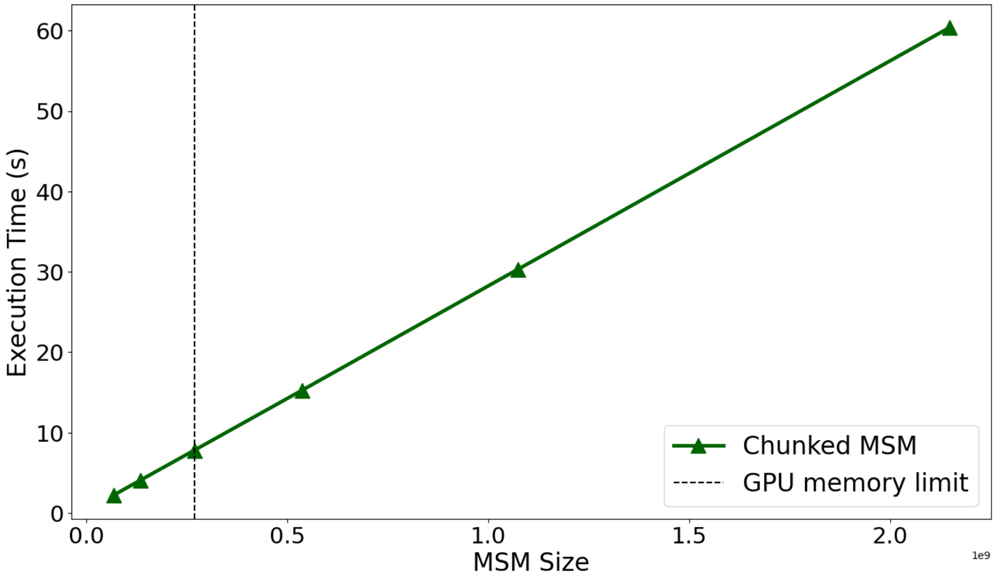


Figure 4.6: MSM evaluation times (shown here for $N = 2^{26} \rightarrow 2^{31}$) remain linear as problem size increases past GPU memory capacity.

subsequent memory chunk as it is needed.

As a result, our chunked approach performs better than the ICICLE baseline at large MSM sizes, with a 25% improvement in runtime – from 2.6s to 1.9s – when using chunking to hide memory access latency at $N = 2^{26}$. On the other hand, at small MSM sizes, the overhead of initiating repeated memory transfers is less efficient than a single copy of all the bases and scalars. When $N = 2^{19}$, or approximately 500,000 elements, chunking exhibits a 4% increase in execution time over the baseline. Thus, for small inputs, chunking is not necessary, while for large inputs, chunking can have a noticeable effect on runtime. Below, we evaluate the limits on problem sizes that we can compute using this chunking technique.

4.5.3 Chunking Huge MSM Problems

The Pippenger algorithm can already combine many base points in an MSM into a single bucket, thus requiring a static amount of bucket memory regardless of problem size. Given that we also segment the MSM and load data to the GPU in fixed-size chunks, it is possible to now compute over MSM problems of unbounded size. In Figure 4.6, we show the evaluation time for problem sizes far surpassing the amount of memory on our 32 GB GPU. We fix our chunk size at $C = 2^{24}$ elements, or approximately 1.26 GB of base points and scalars, and queue enough chunks through the GPU such that we achieve the desired MSM size.

Even without counting the buckets and other memory allocations required by the Pippenger algorithm, our 32 GB of on-device memory could only store approximately 2^{28} (96-byte base point, 32-byte scalar) pairs. With chunking, we can leverage the GPU to calculate,

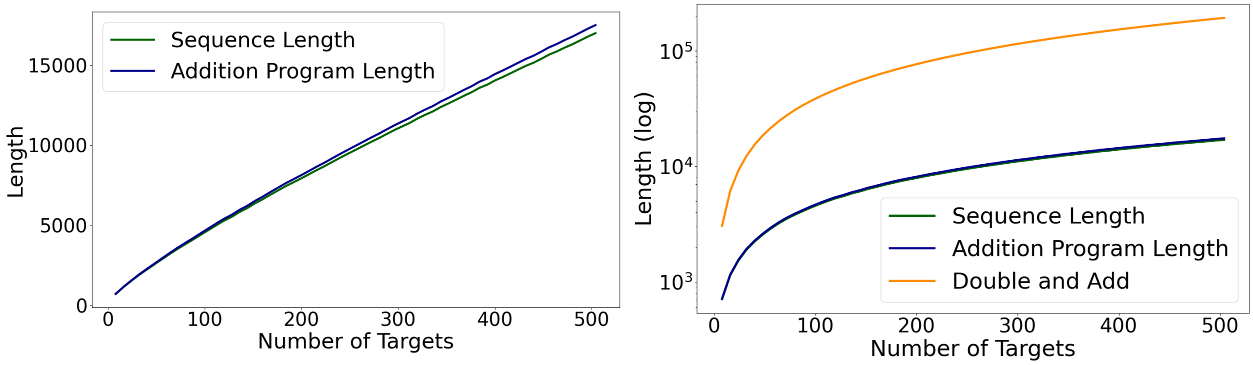


Figure 4.7: Addition sequence length and program length as the number of target scalars increases. The number of program instructions generated closely matches the number of scalars in the addition sequence. On the right, we compare the sequence length to the number of operations in a naive double-and-add strategy on a log axis for each of the bases in the batch.

for example, a 2^{31} size MSM, 8 times larger than that limit, in 60.4 seconds. Note that, as MSM sizes grow larger, we might expect the overall runtime to see sublinear growth, as the number of expensive elliptic curve multiplications stays fixed with the number of buckets. However, we see linear growth in MSM execution time, as the sheer number of base points that need to be added into the Pippenger buckets outweighs the savings in exponentiation cost. This linear operation dominates the overall runtime.

4.5.4 Addition Sequence Generation

Figure 4.7 shows the growth of the addition sequence programs we generate, in instructions, as a function of the number of scalar targets we are attempting to compute (equivalent to the batch size). From [234, 49], we expect the addition sequence lengths to grow linearly with the number of targets r, proportional to $\log N + cr \log N / \log \log N$, where c is some constant and N is the maximum exponent value.

For MSMs with pseudorandom scalars, the maximum exponent value will generally land between 2^{255} and $2^{256} - 1$. In the left part of Figure 4.7, we see the length of the addition sequences increases to approximately 17,000 operations as the batch size reaches 512. Since we generally convert addition operations directly to addition instructions in program generation, only adding a instructions to save certain intermediary results, the addition program length closely matches sequence length, with some increasing overhead as the number of targets increases.

The length of this addition sequence is much lower than the number of additions required to double-and-add each base separately to the desired exponent value, shown to the right: the same scalar multiplications implemented using double-and-add would require almost

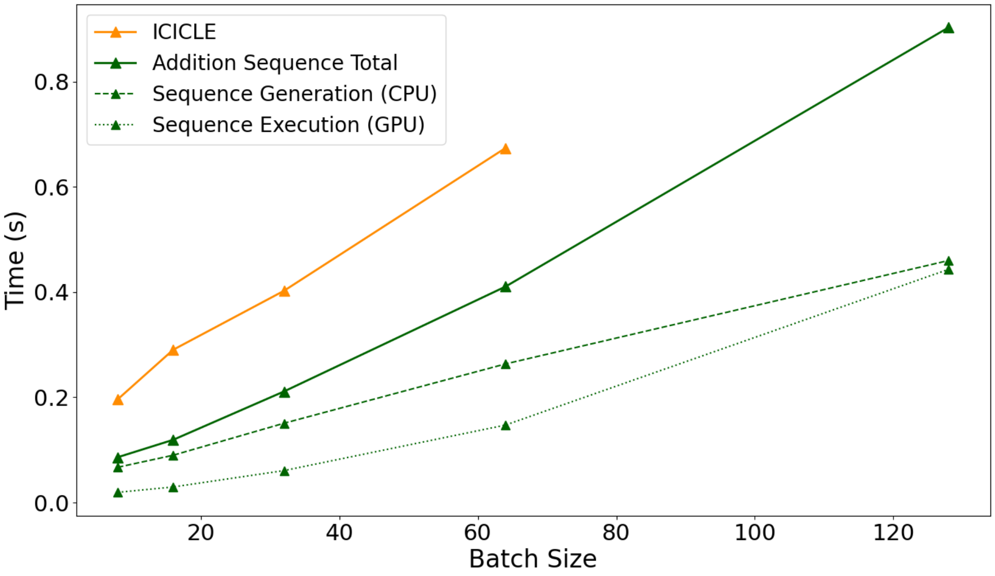


Figure 4.8: MSM evaluation times using addition sequences are consistently lower over a range of input batch sizes of small $N = 100$ proofs. We show the two components of overall sequence evaluation time – program generation on the CPU and program execution on the GPU – separately as well. These components can be pipelined. Note that for batch sizes larger than 64, ICICLE cannot execute due to lack of available memory.

195,000 addition operations. Thus, there is a significant benefit to batching multiplications over the same base with an addition sequence. In the next section, we compare our evaluation technique against the Pippenger algorithm, a better solution than naively multiplying each point by its respective scalar.

4.5.5 Batch MSM Evaluation with Addition Sequences

In Figure 4.8, we compare the runtime of our ICICLE baseline to execution of addition sequences over batches of $N = 100$ small MSM problems, ranging from 8 to 128. In each case, using addition sequences results in better performance compared to the Pippenger algorithm, with improvements ranging from 56% when batch size is 8 to 40% at a larger batch size of 64. Note that our addition sequence-based implementation can support batches of 128 MSMs, while ICICLE fails due to lack of available memory; our implementation does not require a large allocation for a bucket set associated with each individual problem.

A core reason for this performance difference is the amount of computation required by Pippenger for many small MSMs. Each MSM base point is accumulated into a large, per-MSM set of buckets. For BLS12-381, scalars are commonly chunked into 16-bit windows, yielding 16 sets of 2^{16}-entry buckets. At small N, most buckets are empty, with the remaining points evenly distributed throughout. As a result, when the buckets are aggregated on a per-MSM basis, they are effectively performing many 16-bit double-and-add multiplications, which do not see any benefit from batch execution. In contrast, using addition sequences allows us to directly re-use intermediate state from a single multiplication to generate many individual results. It is also important to note that, regardless of MSM size, ICICLE allocates all necessary buckets on the GPU even if they will not be used in the computation, leading to constraints on batch size. Addition sequences allow us to bypass any needed bucket allocation, supporting larger batch sizes: we show that we can compute a batch of 128 MSMs, where ICICLE would otherwise run out of memory and fail to compute.

Finally, Figure 4.8 shows the relative costs of generating the addition sequences and executing the additions based on the resulting program. Importantly, we are able to leverage our CPU cores to generate sequences – a branch-heavy operation that requires significant amount of dynamic memory allocation – while the GPU can focus on sequence execution, given the instructions on which additions to perform and where to copy the intermediate results. Thus, we can leverage pipeline parallelism to improve end-to-end throughput by pre-processing addition sequences while GPU execution is ongoing. We discuss some interesting directions to leverage and improve this ability below.

4.6 Opportunities for Future Work

CPU Parallelism

As batch size increases, so does sequence generation time. Since searching for previous addition results to compose the current value dominates this runtime, additional parallelization could benefit pre-GPU work. We implement addition sequence generation using a C++ thread pool, but overhead from spawning processing threads affects overall runtime. Migrating the addition sequence generation to a process-based parallelism strategy like MPI would increase generation-phase performance. A bright area for additional effort could be in leveraging a second GPU to perform sequence generation, or a relatively cheap distributed cluster to enhance processing power. In addition, parallelizing sequence generation using *different* heuristics could yield better-quality programs that are widen the gap between our approach and Pippenger-based implementations.

Improving Addition Programs

Currently, our prototype simply emits Add instructions when generating the sequence programs, eschewing any additional passes. Future work could easily add additional passes to detect and emit Double and Shift (repeated doubling) instructions, which would make overall computation more efficient in those cases. Optimization passes could also help minimize the number of temporaries required by detecting redundant operations. Finally, more analysis of the addition programs could expose opportunities for in-program parallelism. Currently, programs are expected to be executed serially, but if desired multiplication targets generate addition instructions that branch off from the others, opportunities exist to leverage instruction level parallelism in computing between multiple threads sharing the same program on the GPU.

GPU Parallelism

As mentioned above, future work should investigate moving addition sequence generation to another GPU in order to speed up processing. Several challenges, however, remain. GPU kernels generating sequences must track and modify dynamically-growing data structures, like a binary search tree or similar datastructure underlying the ordered set structure necessary for the Delta Largest heuristic. Threads will also need to manage a map to track temporary variable use in register allocation. These kernels will likely be branch-heavy, reducing overall computational throughput. However, the core operation is extremely parallelizable across every slice of the MSM batch, so leveraging the higher core count on the GPU could help overall throughput regardless.

Integrating Precomputation

Many addition programs may benefit from precomputing multiplications at certain offsets for each base – this precomputation strategy is one that many Pippenger algorithm implementations use to reduce total bucket allocations for each MSM instance. For example, if it is known that the addition sequence will have access to $2^1 P_i, 2^2 P_i, 2^4 P_i, \cdots$, it may allow for faster addition programs overall, composing existing precomputed values instead of generating them from scratch each time. Since addition sequences lengths are scaled by the size of the maximum element in the sequence, "short-circuiting" the sequence by starting within a power of the final result may lead to more efficient programs overall.

4.7 Summary

In this chapter, we presented two complementary approaches to restructuring MSM computation for zero-knowledge proving for execution on GPUs, in a way that does not sacrifice performance. In particular, we first describe a chunking method for reducing end-to-end latency of large MSM calculations that streams input data to the GPU such that it overlaps with computation, supporting overall MSM sizes larger than could feasibly sit in device memory. We do so without incurring the cost of unified memory, and show a modest 25% improvement in processing speed compared to the baseline. Second, we describe a way to leverage addition sequences at runtime to make batch MSM aggregation more efficient, bypassing Pippenger's algorithm completely. We show 40-56% improvement on batches of MSMs, and our prototype leverages both the CPU cores available and the GPU, which enables better pipelining in a high-throughput environment.

Chapter 5

Conclusion

Each of the three systems presented in this dissertation (Nebula, Piranha, and scalable ZK proving) seeks to leverage an advanced cryptographic primitive – metadata-hiding communication, multi-party computation, and zero-knowledge proofs – by restructuring its computation to leverage the capabilities of resource-constrained mobile devices or GPUs.

Nebula (Chapter 2) shows that it is possible for data backhaul networks to return useful data payloads to application servers through third-parties while vastly reducing mule metadata leakage to the central platform provider. We did so by restructuring our data backhaul architecture to place core functionality for payment and system abuse prevention on a *mobile* platform, instead of centralizing that operation in the provider. Blindly-signed tokens allow mules to prove their participation later without revealing their individual upload patterns, and our cryptographic complaint protocol can leverage transcripts of individual deliveries to prove misbehavior, revealing the minimum information necessary only in the case that something goes awry.

While implementing Piranha (Chapter 3), we made the prospect of large-scale private ML training significantly more likely, improving the performance of LSSS-based MPC protocols by restructuring operations over local secret shares so that they could easily be executed on the GPU. In particular, we enable protocol developers to use integer-based matrix multiplication and convolution kernels, rather than incurring the overhead of using existing floating point implementations. Piranha also ensures that the high memory cost of performing bitwise operations (e.g. during secure comparisons) is moderated, allowing larger models to be trained using limited GPU memory. These systems-level changes to adapt to GPU limitations were critical in enabling end-to-end training.

Finally, when considering zero-knowledge proving in a batched scenario (Chapter 4), we argue for restructuring MSM computation into parallel evaluation of many addition chains, one per MSM element across each of a batch's inputs. The resulting, easily parallelized process is a natural fit for a GPU's core computing capability and can minimize allocations otherwise required for many independent MSM calculations.

While the work presented in this dissertation significantly improves our ability to build cryptographic systems, it has certainly not eliminated the wide gulf that remains between plaintext and privacy-first systems. When compared to plaintext computation, primitives such as MPC incur runtimes that are slower by orders of magnitude. To conclude, we highlight some key approaches that future cryptographers and hardware designers alike can take to further lower the cost of privacy.

5.1 Designing Cryptography for New Hardware

A critical modification we made in both Nebula and Piranha was to **map desired cryptographic operations to hardware-supported data types and computation**. In Nebula's case, we consciously placed the most complex operation, applying blind signatures to generate PrivacyPass tokens, to the central provider. During normal operation, application servers (also cloud-based, relatively unconstrained systems) support the other half of the PrivacyPass signing process, generating random elliptic curve points and verifying the DLEQs returned by the provider. Only infrequently, during a complaint, is a mobile phone asked to participate in token generation, and Nebula's embedded sensors never interface with the token accounting system. As a result, we only require the most constrained low-power devices to have access to symmetric and asymmetric cryptographic accelerators, which embedded platforms commonly possess. In Piranha, we note that LSSS protocols using arithmetic secret sharing can naturally operate over local 32- and 64-bit integer shares, for which GPUs already have significant processing power – what is critical is to expose that capability to protocol developers. On the other hand, applications like zero-knowledge proving rely on elliptic curve operations that do not naturally align to what GPUs "normally" compute, and so are rather limited in the speedups that can easily be achieved (e.g. limited register file size reduces the number of threads that can run concurrently when accessing many EC points).

Notwithstanding any future improvements to hardware accelerators, cryptographers should prioritize *simplifying* their constructions to use small fields and existing symmetric and asymmetric primitives, perhaps at the cost of increasing the number of computational steps required to achieve a desired result.

A second design approach that seems highly beneficial, based on our experiences with accelerating MPC and ZK proving, is to **prioritize low memory overheads over computation**. In Piranha, the primary limiting factor for model training was temporary memory allocation during forward and backward passes, caused by memory blowup from binary shares during comparisons. In a WAN setting, the size of the shares being transmitted between parties meant that overall training progress was bottlenecked by communication, not computation. Similarly, the GPU memory reserved for MSM inputs in our baseline limited the problem size we could tackle. In a batch setting, repeated allocation of separate Pippenger bucket sets for each MSM exhausted GPU device memory and limited the batch size we could process. Focusing on memory use, by switching to memory chains to avoid bucket allocation or supporting use of in-place memory iterators to avoid temporary allocation, significantly improved the scale of the problems we were able to tackle.

Given the speed of existing hardware, applied cryptographers should consider whether recomputing locally to reduce memory usage, such as rematerializing activations when needed during ML training, for example, can improve overall performance.

5.2 Designing Hardware for New Cryptography

Expanding the scope and flexibility of current cryptographic accelerators, and in particular, **hardware support for elliptic curve (EC) operations** would be hugely beneficial in supporting new privacy-preserving protocols. Some elliptic curve capability already exists, in that many chips can perform the ECDSA operations that underlie TLS session establishment. However, support only exists for signing and verifying particular messages given a specific public or private key, and does not expose any ability to add or multiply raw elliptic curve points. All MSM implementations that use GPUs today implement expensive EC addition and multiplication routines in software, reducing the computations to bare operations over very large 256- or 381-bit integers that were never intended as first-class citizens.

Similar capabilities are missing from commercial CPUs. Adding efficient support for larger fields and elliptic curves could easily widen the scope of the work in this dissertation and beyond – Piranha could target MPC protocols beyond those that use LSSS, while ZK proving could maximize device usage on the GPU and increase parallelization that is currently limited by the clunky mechanics of moving large EC points around in memory.

Finally, since each of the projects in this dissertation found the overriding constraint to be memory utilization, new hardware generations can support much larger cryptographic systems by simply **increasing memory availability**. In particular, while GPUs currently boast a significant amount of global memory, up to 80 GB on NVIDIA A100 GPUs, the register file size per streaming multiprocessor that threads can leverage during computation is at most 256 kB [170], which has not increased over the previous Volta generation of NVIDIA GPUs. While there is certainly a legitimate cost and other constraints to adding more register file space, expanded memory availability would make expensive operations over large fields easier to parallelize. The increasing availability of fast global memory when multiple GPUs are connected together is also a great target for supporting much larger ML model or ZK proof sizes.

Computation is shifting away from "a computer on every desk and in every home" [19] and towards centralized systems on the cloud. Privacy-enabling cryptographic solutions have lagged behind, but by leveraging increasingly-powerful specialized hardware, we can hope to close that gap. Given recent interest in security technologies from enclaves to MPC and ZKP, it seems likely that this trend will continue. In this dissertation, we showed that building performant cryptographic systems can be accomplished with two techniques. First, *concentrating* computation on hardware accelerators along with careful memory management, such as in Piranha and ZK proof acceleration, can accelerate heavy-weight operations like private neural network training passes or multi-scalar multiplication. Second, optimistically *decentralizing* control of a cryptographic protocol among a vast number of low-compute devices, with a mechanism to rectify misbehavior can support a large-scale private distributed system like data backhaul in Nebula, all without forming a bottleneck at a single centralized operation. Together, they show that cryptographic systems at scale are possible, both in the number of participants and amount of required computation.

Bibliography

[1] *3GPP*. https://www.3gpp.org/. 2023.

[2] *A Privacy-Preserving Framework Based on TensorFlow*. https://github.com/LatticeX-Foundation/Rosetta/.

[3] *A Python library for secure and private Deep Learning*. https://github.com/OpenMined/PySyft.

[4] Joshua Adkins et al. "Applications on the signpost platform for city-scale sensing: demo abstract". In: *International Symposium on Information Processing in Sensor Networks*. 2018.

[5] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. "Understanding of a convolutional neural network". In: *2017 International Conference on Engineering and Technology (ICET)*. 2017.

[6] Suzan Ali et al. "On privacy risks of public WiFi captive portals". In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. 2019.

[7] Abdelrahaman Aly et al. *SCALE-MAMBA v1.2: Documentation*. https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf. 2018.

[8] Abdelrahaman Aly et al. "Zaphod: Efficiently Combining LSSS and Garbled Circuits in SCALE". In: *ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2019.

[9] *Amazon Sidewalk Privacy and Security Whitepaper*. https://m.media-amazon.com/images/G/01/sidewalk/final_privacy_security_whitepaper.pdf. 2023.

[10] *Android Studio — Power Profiler*. https://developer.android.com/studio/profile/power-profiler. 2023.

[11] *Anonymous Tokens: Efficient Anonymous Tokens with Private Metadata Bit*. https://github.com/mmaker/anonymous-tokens. 2023.

[12] *Apollo4 Lite SoC Datasheet*. https://ambiq.com/wp-content/uploads/2023/03/Apollo4-Lite-Datasheet.pdf. 2023.

[13] *Apple A16 Bionic*. https://nanoreview.net/en/soc/apple-a16-bionic. 2024.

[14] Apple/Google. *Exposure Notification - Bluetooth Specification.* https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ExposureNotification-BluetoothSpecificationv1.2.pdf. 2020.

[15] Toshinori Araki et al. "High-throughput semi-honest secure three-party computation with an honest majority". In: *ACM Conference on Computer and Communications Security (CCS).* 2016.

[16] Tair Askar et al. "Evaluation of Pseudo-Random Number Generation on GPU Cards". In: *Computation* (2021).

[17] Android Authority. *You told us: This is the battery capacity of most of your smartphones right now.* https://www.androidauthority.com/smartphone-battery-size-poll-results-1221015/. 2021.

[18] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[19] Hannah Bae. "Bill Gates' 40th anniversary email: goal was a computer on every desk". In: *CNN Money* (2015).

[20] Benjamin Baron and Mirco Musolesi. "Where you go matters: a study on the privacy implications of continuous location tracking". In: *ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* (2020).

[21] Johannes K. Becker, David Li, and David Starobinski. "Tracking Anonymized Bluetooth Devices". In: *PET* (2019).

[22] Alex Bellon, Alex Yen, and Pat Pannuto. "TagAlong: Free, Wide-Area Data-Muling and Services". In: *Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications.* 2023.

[23] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. "Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract)". In: *ACM Symposium on Theory of Computing (STOC).* 1988.

[24] François Bergeron et al. "Addition chains using continued fractions". In: *Journal of Algorithms* (1989).

[25] Daniel J Bernstein. "Pippenger's exponentiation algorithm". In: *Preprint. Available from http://cr. yp.to/papers.html* (2002).

[26] Ketan Bhardwaj, Joaquin Chung Miranda, and Ada Gavrilovska. "Towards IoT-DDoS prevention using edge computing". In: *{USENIX} Workshop on Hot Topics in Edge Computing.* 2018.

[27] *BLE Security.* https://www.bluetooth.com/bluetooth-resources/le-security-study-guide/. 2023.

[28] Jurjen Bos and Matthijs Coster. "Addition chain heuristics". In: *Conference on the Theory and Application of Cryptology.* 1989.

[29] Gautam Botrel et al. *ConsenSys/gnark: v0.9.0*. 2023. URL: https://doi.org/10.5281/zenodo.5819104.

[30] Sean Bowe et al. "Zexe: Enabling decentralized private computation". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020.

[31] Kevin W Bowyer, Karen Hollingsworth, and Patrick J Flynn. "Image understanding for iris biometrics: A survey". In: *Computer vision and image understanding* (2008).

[32] Elette Boyle, Niv Gilboa, and Yuval Ishai. "Function secret sharing". In: *Advances in Cryptology—EUROCRYPT*. 2015.

[33] Elette Boyle et al. "Efficient pseudorandom correlation generators: Silent OT extension and more". In: *Advances in Cryptology—CRYPTO*. 2019.

[34] Elette Boyle et al. "Efficient two-round OT extension and silent non-interactive secure computation". In: *ACM Conference on Computer and Communications Security (CCS)*. 2019.

[35] Jimmy Briggs and Christine Geeng. "BLE-Doubt: Smartphone-Based Detection of Malicious Bluetooth Trackers". In: *2022 IEEE Security and Privacy Workshops (SPW)* (2022).

[36] William J Buchanan. *Digital Signature Benchmark*. https://asecuritysite.com/openssl/openssl3_b2. 2024.

[37] Megha Byali et al. "FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning". In: *Privacy Enhancing Technologies Symposium (PETS)*. 2020.

[38] *Calculating the Size of GPT Models in Gigabytes: A Simple Guide*. https://raiday.ai/blog/generative-ai/calculate-gpt-model-size-in-gb/. 2024.

[39] Ran Canetti. "Universally composable security: A new paradigm for cryptographic protocols". In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. 2001.

[40] Laura Ceci. *WhatsApp - Statistics & Facts*. https://www.statista.com/topics/2018/whatsapp/. 2024.

[41] *Certificate Transparency*. https://certificate.transparency.dev/. Last visited on December, 2023.

[42] Jon Chase. *Amazon sidewalk will share your internet with strangers. it's not as scary as it sounds.* https://www.nytimes.com/wirecutter/blog/amazon-sidewalk-review/. 2021.

[43] Harsh Chaudhari et al. "Astra: High throughput 3pc over rings with application to secure prediction". In: *ACM SIGSAC Conference on Cloud Computing Security Workshop*. 2019.

[44] David Chaum. "The Dining Cryptographers problem: Unconditional sender and recipient untraceability". In: *Journal of Cryptology* (1988).

[45] Hao Chen et al. "Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning". In: *Advances in Cryptology—ASIACRYPT*. 2020.

[46] Yutian Chen et al. "Load-Balanced Parallel Implementation on GPUs for Multi-Scalar Multiplication Algorithm". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2024).

[47] Raymond Cheng et al. "Talek: Private group messaging with hidden access patterns". In: *Annual Computer Security Applications Conference*. 2020.

[48] Rezaul Alam Chowdhury et al. "Oblivious algorithms for multicores and networks of processors". In: *Journal of Parallel and Distributed Computing* (2013).

[49] Henri Cohen et al. *Handbook of elliptic and hyperelliptic curve cryptography*. 2005.

[50] Cory Cornelius et al. "Anonysense: privacy-aware people-centric sensing". In: *Proceedings of the 6th international conference on Mobile systems, applications, and services*. 2008.

[51] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. "Riposte: An anonymous messaging system handling millions of users". In: *2015 IEEE Symposium on Security and Privacy*. 2015.

[52] M. Scott Corson and Joseph P. Macker. "Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations". In: *RFC* (1999).

[53] *CrypTen: Privacy-Preserving Machine Learning built on PyTorch*. https://github.com/facebookresearch/CrypTen. 2019.

[54] *cuBLAS*. https://developer.nvidia.com/cublas.

[55] *CUTLASS: CUDA Templates for Linear Algebra Subroutines*.

[56] Wei Dai and Berk Sunar. "cuHE: A homomorphic encryption accelerator library". In: *International Conference on Cryptography and Information Security in the Balkans*. 2015.

[57] Anders Dalskov, Daniel Escudero, and Marcel Keller. "Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security". In: *USENIX Security Symposium (USENIX)*. 2021.

[58] Ivan Damgård and Jesper Buus Nielsen. "Universally composable efficient multiparty computation from threshold homomorphic encryption". In: *Advances in Cryptology—CRYPTO*. 2003.

[59] Ivan Damgård et al. "Asynchronous multiparty computation: Theory and implementation". In: *International workshop on public key cryptography*. 2009.

[60] Ivan Damgård et al. "Multiparty Computation from Somewhat Homomorphic Encryption". In: *Annual Cryptology Conference*. 2012.

[61] Ivan Damgård et al. "New primitives for actively-secure MPC over rings with applications to private machine learning". In: *IEEE Symposium on Security and Privacy (S&P)*. 2019.

[62] Ivan Damgård et al. "Practical Covertly Secure MPC for Dishonest Majority–or: Breaking the SPDZ Limits". In: *European Symposium on Research in Computer Security*. 2013.

[63] Data61. *MP-SPDZ: Versatile Framework for Multi-party Computation*. https://github.com/data61/MP-SPDZ. 2019.

[64] Alex Davidson et al. "Privacy Pass: Bypassing Internet Challenges Anonymously." In: *Proc. Priv. Enhancing Technol.* (2018).

[65] Yves-Alexandre De Montjoye et al. "Unique in the crowd: The privacy bounds of human mobility". In: *Scientific reports* (2013).

[66] Deloitte. *Smartphone batteries: better but no breakthrough*. https://www2.deloitte.com/content/dam/Deloitte/global/Documents/Technology-Media-Telecommunications/gx-tmt-pred15-smartphone-batteries.pdf. 2015.

[67] Daniel Demmler, Thomas Schneider, and Michael Zohner. "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation". In: *Symposium on Network and Distributed System Security (NDSS)*. 2015.

[68] Tess Despres et al. "Where the sidewalk ends: privacy of opportunistic backhaul". In: *Proceedings of the 15th European Workshop on Systems Security*. 2022.

[69] Roger Dingledine, Nick Mathewson, and Paul Syverson. *Tor: The second-generation onion router*. Tech. rep. Naval Research Lab Washington DC, 2004.

[70] Ye Dong et al. "Meteor: improved secure 3-party neural network inference with reducing online communication costs". In: *Proceedings of the ACM Web Conference 2023*. 2023.

[71] Ye Dong et al. "Puma: Secure inference of llama-7b in five minutes". In: *arXiv preprint arXiv:2307.12533* (2023).

[72] Yong Dou et al. "64-bit floating-point FPGA matrix multiplication". In: *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. 2005.

[73] Matt Duckham and Lars Kulik. "Location privacy and location-aware computing". In: *Dynamic and mobile GIS*. 2006.

[74] Rob A Dunne and Norm A Campbell. "On the pairing of the softmax activation and cross-entropy penalty functions and the derivation of the softmax activation function". In: *Proc. 8th Aust. Conf. on the Neural Networks, Melbourne*. 1997.

[75] Cynthia Dwork, Aaron Roth, et al. "The algorithmic foundations of differential privacy". In: *Foundations and Trends in Theoretical Computer Science*. 2014.

[76] Daniel Escudero, Anders Dalskov, and Marcel Keller. “Secure evaluation of quantized neural networks”. In: *Privacy Enhancing Technologies Symposium (PETS)*. 2020.

[77] Daniel Escudero et al. “Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits”. In: *Advances in Cryptology—CRYPTO*. 2020.

[78] Saba Eskandarian et al. “Express: Lowering the cost of metadata-hiding communication with cryptographic privacy”. In: *USENIX Security*. 2021.

[79] *ESP-IDK programming guide*. https://docs.espressif.com/. 2023.

[80] Honghui Fan, Hongjin Zhu, and Dongming Yuan. “People counting in elevator car based on computer vision”. In: *IOP Conference Series: Earth and Environmental Science*. 2019.

[81] *Find My Network*. https://developer.apple.com/find-my/. 2023.

[82] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, and Jesper Buus Nielsen. “Faster Maliciously Secure Two-Party Computation Using the GPU”. In: *Conference on Security and Cryptography for Networks*. 2014.

[83] Jun Furukawa et al. “High-throughput secure three-party computation for malicious adversaries and an honest majority”. In: *Advances in Cryptology—EUROCRYPT*. 2017.

[84] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. “Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge”. In: *Cryptology ePrint Archive* (2019).

[85] Kimberly Gedeon. *iPhone 15 vs. iPhone 15 Pro: What are the differences?* https://mashable.com/article/iphone-15-vs-iphone-15-pro. 2024.

[86] Craig Gentry. “A fully homomorphic encryption scheme”. crypto.stanford.edu/craig. PhD thesis. 2009.

[87] Branden Ghena et al. “Challenge: Unlicensed LPWANs Are Not Yet the Path to Ubiquitous Connectivity”. In: *The 25th Annual International Conference on Mobile Computing and Networking* (2019).

[88] Hadi Givehchian et al. “Evaluating Physical-Layer BLE Location Tracking Attacks on Mobile Devices”. In: *2022 IEEE Symposium on Security and Privacy (SP)* (2022).

[89] Oded Goldreich. “Towards a theory of software protection and simulation by oblivious RAMs”. In: *ACM Symposium on Theory of Computing (STOC)*. 1987.

[90] Oded Goldreich, Silvio Micali, and Avi Wigderson. “How to play any mental game or a completeness theorem for protocols with honest majority”. In: *ACM Symposium on Theory of Computing (STOC)*. 1987.

[91] Andy Greenberg. *The Clever Cryptography Behind Apple’s ‘Find My’ Feature*. https://www.wired.com/story/apple-find-my-cryptography-bluetooth/. 2019.

[92] Jens Groth. “On the size of pairing-based non-interactive arguments”. In: *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. 2016.

[93] Kanav Gupta et al. “Sigma: Secure gpt inference with function secret sharing”. In: *Cryptology ePrint Archive* (2023).

[94] Jaap C. Haartsen. “Bluetooth-ad-hoc networking in an uncoordinated environment”. In: *IEEE International Conference on Acoustics, Speech, and Signal Processing* (2001).

[95] Amir Haleem et al. “Helium: A Decentralized Wireless Network”. In: (2018).

[96] *Hardware for Deep Learning — Part 3*. https://blog.inten.to/hardware-for-deep-learning-part-3-gpu-8906c1644664. 2024. (Visited on 05/07/2024).

[97] Tim Harris. *Unified Memory for CUDA Beginners*. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/. 2017.

[98] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *IEEE international conference on computer vision*. 2015.

[99] Robert Hecht-Nielsen. “Theory of the backpropagation neural network”. In: *Neural networks for perception*. 1992.

[100] A. Heinrich, Niklas Bittner, and Matthias Hollick. “AirGuard - Protecting Android Users from Stalking Attacks by Apple Find My Devices”. In: *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (2022).

[101] Alexander Heinrich, Milan Stute, and Matthias Hollick. “OpenHaystack: a framework for tracking personal bluetooth devices via Apple’s massive find my network”. In: *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 2021.

[102] *Helium Explorer*. https://explorer.helium.com/. 2023.

[103] Xiaoyang Hou et al. “Ciphergpt: Secure two-party gpt inference”. In: *Cryptology ePrint Archive* (2023).

[104] *How Tile Works*. www.thetileapp.com/en-us/how-it-works. 2023.

[105] Jyh-How Huang, Saqib Amjad, and Shivakant Mishra. “CenWits: a sensor-based loosely coupled search and rescue system using witnesses”. In: *ACM Intl. Conf. on Embedded Networked Sensor Systems*. 2005.

[106] Ying Huang, Xiaoying Zheng, and Yongxin Zhu. “Optimized CPU–GPU collaborative acceleration of zero-knowledge proof for confidential transactions”. In: *Journal of Systems Architecture* (2023).

[107] Nathaniel Husted et al. “GPU and CPU Parallelization of Honest-but-Curious Secure Two-Party Computation”. In: *Annual Computer Security Applications Conference (ACSAC)*. 2013.

[108] *ICICLE: a GPU Library for Zero-Knowledge Acceleration*. https://github.com/ingonyama-zk/icicle. 2024.

[109] Karthik Inbasekar, Yuval Shekel, and Michael Asa. “ICICLE v2: Polynomial API for Coding ZK Provers to Run on Specialized Hardware”. In: *Cryptology ePrint Archive* (2024).

[110] Yuval Ishai et al. “Cryptography from anonymity”. In: *IEEE Symposium on Foundations of Computer Science (FOCS)*. 2006.

[111] Neal Jackson, Joshua Adkins, and Prabal Dutta. “Capacity over Capacitance for Reliable Energy Harvesting Sensors”. In: *The 18th International Conference on Information Processing in Sensor Networks*. 2019.

[112] Dhananjay Jagtap et al. “Federated infrastructure: usage, patterns, and insights from ”the people’s network””. In: *ACM Internet Measurement Conference* (2021).

[113] Neha Jawalkar et al. “Orca: FSS-based Secure Training and Inference with GPUs”. In: *Cryptology ePrint Archive* (2023).

[114] Marek Jawurek, Florian Kerschbaum, and George Danezis. “Privacy Technologies for Smart Grids - A Survey of Options”. In: 2012. URL: https://api.semanticscholar.org/CorpusID:15815464.

[115] Scott Jenson et al. “Building an On-ramp for the Internet of Things”. In: *Proceedings of the 2015 Workshop on IoT challenges in Mobile and Industrial Systems* (2015).

[116] Zhuoran Ji et al. “Accelerating Multi-Scalar Multiplication for Efficient Zero Knowledge Proofs with Multi-GPU Systems”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2024.

[117] Philo Juang et al. “Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with ZebraNet”. In: *ASPLOS X*. 2002.

[118] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. “GAZELLE: A Low Latency Framework for Secure Neural Network Inference”. In: *USENIX Security Symposium (USENIX)*. 2018.

[119] Renuga Kanagavelu et al. “Two-Phase Multi-Party Computation Enabled Privacy-Preserving Federated Learning”. In: *IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2020.

[120] Srikanth Kandula et al. “Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds”. In: *PNSDI*. 2005.

[121] Apu Kapadia et al. “AnonySense: Opportunistic and privacy-preserving context collection”. In: *Pervasive Computing: 6th International Conference*. 2008.

[122] Marcel Keller. “MP-SPDZ: A Versatile Framework for Multi-Party Computation”. In: *ACM Conference on Computer and Communications Security (CCS)*. 2020.

[123] Marcel Keller, Emmanuela Orsini, and Peter Scholl. "MASCOT: Faster malicious arithmetic secure computation with oblivious transfer". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016.

[124] Marcel Keller, Valerio Pastro, and Dragos Rotaru. "Overdrive: making SPDZ great again". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 2018.

[125] Marcel Keller and Ke Sun. *Effectiveness of MPC-friendly Softmax Replacement*. https://arxiv.org/pdf/2011.11202.pdf. 2020.

[126] Noah Klugman et al. "Experience: Android resists liberation from its primary use case". In: *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. 2018.

[127] Markulf Kohlweiss et al. "Snarky ceremonies". In: *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III 27*. 2021.

[128] Nishat Koti et al. "SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning". In: *USENIX Security Symposium (USENIX)*. 2021.

[129] Ben Kreuter et al. "Anonymous tokens with private metadata bit". In: *CRYPTO*. 2020.

[130] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. *The CIFAR-10 Dataset*. https://www.cs.toronto.edu/~kriz/cifar.html. 2014.

[131] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2012.

[132] Albert Kwon et al. "Atom: Horizontally scaling strong anonymity". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017.

[133] Albert Hyukjae Kwon et al. "Riffle: An efficient communication system with strong anonymity". In: (2016).

[134] Shuyue Lan et al. "AdaSens: Adaptive Environment Monitoring by Coordinating Intermittently-Powered Sensors". In: *Asia and South Pacific Design Automation Conference* (2022).

[135] Peeter Laud et al. "Specifying Sharemind's arithmetic black box". In: *ACM workshop on Language support for privacy-enhancing technologies*. 2013.

[136] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* (1998).

[137] Brent Ledvina et al. *Detecting Unwanted Location Trackers*. Tech. rep. 2023. URL: https://datatracker.ietf.org/doc/draft-detecting-unwanted-location-trackers/00/.

[138] David Thomas Lee Howes. *Chapter 37. Efficient Random Number Generation and Application Using CUDA*. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-37-efficient-random-number-generation-and-application.

[139] Ryan Lehmkuhl et al. "Muse: Secure Inference Resilient to Malicious Clients". In: *USENIX Security Symposium (USENIX)*. 2021.

[140] Matthew Lentz et al. "SDDR: Light-Weight, Secure Mobile Encounters". In: *USENIX Security Symposium*. 2014.

[141] Martin Lesund. *LTE-M vs NB-IoT Field Test*. https://devzone.nordicsemi.com/nordic/nordic-blog/b/blog/posts/ltem-vs-nbiot-field-test-how-distance-affects
-power-consumption. 2022.

[142] Xin Liu, Xiaowei Yang, and Yanbin Lu. "To filter or to authorize: Network-layer DoS defense against multimillion-node botnets". In: *ACM SIGCOMM*. 2008.

[143] *LoRa Alliance Site*. www.lora-alliance.org. 2023.

[144] *LoRa README*. https://lora.readthedocs.io/en/latest/. 2023.

[145] Hugh Louch et al. *Innovation in bicycle and pedestrian counts*. altago.com/wp-content/uploads/Innovative-Ped-and-Bike-Counts-White-Paper-Alta.pdf. 2016.

[146] Tao Lu et al. "Cuzk: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on gpus". In: *Cryptology ePrint Archive* (2022).

[147] Weiliang Ma et al. "Gzkp: A gpu accelerated zero-knowledge proof system". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2023.

[148] Alan M. Mainwaring et al. "Wireless sensor networks for habitat monitoring". In: *ACM International Conference on Wireless Sensor Networks and Applications*. 2002.

[149] Eleftheria Makri et al. "Rabbit: Efficient Comparison for Secure Multi-Party Computation". In: *Financial Cryptography and Data Security (FC)*. 2021.

[150] Zoltán Ádám Mann et al. "Towards practical secure neural network inference: the journey so far and the road ahead". In: *ACM Computing Surveys* (2023).

[151] Justin Manweiler, Ryan Scudellari, and Landon P. Cox. "SMILE: encounter-based trust for mobile social services". In: *ACM Conference on Computer and Communications Security*. 2009.

[152] Daniel Marin et al. "Nexus 1.0: Enabling Verifiable Computation". In: (2024).

[153] Michael B. McLoughlin. *addchain: Cryptographic Addition Chain Generation in Go*. Repository https://github.com/mmcloughlin/addchain. 2021.

[154] Matthias Meyer et al. "Event-triggered natural hazard monitoring with convolutional neural networks on the edge". In: *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*. 2019.

[155] François Michel et al. "A first look at starlink performance". In: *Proceedings of the 22nd ACM Internet Measurement Conference* (2022).

[156] Microbattery. *Battery Bios: Everything You Need to Know About the AA Battery*. https://www.microbattery.com/blog/post/battery-bios:-everything-you-need-to-know-about-the-aa-battery/.

[157] Pratyush Mishra et al. "Delphi: A Cryptographic Inference Service for Neural Networks". In: *USENIX Security Symposium (USENIX)*. 2020.

[158] *MNIST database*. http://yann.lecun.com/exdb/mnist/. Accessed: 2017-09-24.

[159] *Mobile Fact Sheet*. https://www.pewresearch.org/internet/fact-sheet/mobile/. 2024.

[160] Payman Mohassel and Peter Rindal. "ABY^3: A mixed protocol framework for machine learning". In: *ACM Conference on Computer and Communications Security (CCS)*. 2018.

[161] Payman Mohassel and Yupeng Zhang. "SecureML: A System for Scalable Privacy-Preserving Machine Learning". In: *IEEE Symposium on Security and Privacy (S&P)*. 2017.

[162] Arslan Musaddiq et al. "Internet of Things for Wetland Conservation using Helium Network: Experience and Analysis". In: *International Conference on the Internet of Things* (2022).

[163] Lucien KL Ng and Sherman SM Chow. "GForce: GPU-Friendly Oblivious and Rapid Neural Network Inference". In: *USENIX Security Symposium (USENIX)*. 2021.

[164] Lucien KL Ng and Sherman SM Chow. "SoK: cryptographic neural-network computation". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023.

[165] Ning Ni and Yongxin Zhu. "Enabling zero knowledge proof by accelerating zk-SNARK kernels on GPU". In: *Journal of Parallel and Distributed Computing* (2023).

[166] *Nordic Infocenter*. https://infocenter.nordicsemi.com/. 2023.

[167] *nRF52832 Product Specification v1.0*. https://docs.nordicsemi.com/bundle/nRF52832-PS/resource/nRF52832_PS_v1.0.pdf. 2016.

[168] *nRF5340 Product Specification v1.5*. https://docs-be.nordicsemi.com/bundle/ps_nrf5340/attach/nRF5340_PS_v1.5.pdf. 2024.

[169] Nvidia. *Thrust, the CUDA C++ template library*. https://docs.nvidia.com/cuda/thrust/index.html.

[170] *NVIDIA A100 Tensor Core GPU*. https://www.nvidia.com/en-us/data-center/a100/. 2024.

[171] Manjusha Patil and Vasant N Bhonge. "Wireless sensor network and RFID for smart parking system". In: *International Journal of Emerging Technology and Advanced Engineering* (2013).

[172] Arpita Patra and Ajith Suresh. "BLAZE: Blazing Fast Privacy-Preserving Machine Learning". In: *Symposium on Network and Distributed System Security (NDSS)*. 2020.

[173] Arpita Patra et al. "ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation". In: *USENIX Security Symposium (USENIX)*. 2021.

[174] Claudia Perlich et al. "Machine learning for targeted display advertising: Transfer learning in action". In: *Machine learning* (2014).

[175] Johan Perols. "Financial statement fraud detection: An analysis of statistical and machine learning algorithms". In: *Auditing: A Journal of Practice & Theory* (2011).

[176] Neil Perry et al. "Strong Anonymity for Mesh Messaging". In: *ArXiv abs/2207.04145* (2022).

[177] Sébastien Philippe, Boaz Barak, and Alexander Glaser. "Designing protocols for nuclear warhead verification". In: *Proc. 56th Annual INMM Meeting*. 2015.

[178] Rishabh Poddar et al. "Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics". In: *USENIX Security Symposium (USENIX)*. 2021.

[179] Rishabh Poddar et al. "Visor: Privacy-preserving video analytics as a cloud service". In: *USENIX Security Symposium (USENIX)*. 2020.

[180] Raluca Ada Popa, Hari Balakrishnan, and Andrew J. Blumberg. "VPriv: protecting privacy in location-based vehicular services". In: *USENIX Security Symposium*. 2009.

[181] Raluca Ada Popa et al. "Privacy and accountability for location-based aggregate statistics". In: *ACM Conference on Computer and Communications Security*. 2011.

[182] Amogh Pradeep et al. "Moby: A Blackout-Resistant Anonymity Network for Mobile Devices". In: *Proc. Priv. Enhancing Technol.* (2022).

[183] Jan M. Rabaey et al. "PicoRadio Supports Ad Hoc Ultra-Low Power Wireless Networking". In: *Computer* (2000).

[184] Rahul Rachuri and Ajith Suresh. "Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning". In: *Symposium on Network and Distributed System Security (NDSS)*. 2019.

[185] Deevashwer Rathee et al. "CrypTFlow2: Practical 2-Party Secure Inference". In: *ACM Conference on Computer and Communications Security (CCS)*. 2020.

[186] Andy Ray et al. "Hardcaml MSM: A High-Performance Split CPU-FPGA Multi-Scalar Multiplication Engine". In: *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 2024.

[187] Marc Rennhard and Bernhard Plattner. "Introducing morphmix: Peer-to-peer based anonymous internet usage with collusion detection". In: *Proceedings of the 2002 ACM Workshop on Privacy in the Electronic Society*. 2002.

[188] M. Sadegh Riazi et al. "Chameleon: A hybrid secure computation framework for machine learning applications". In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2018.

[189] Peter Rindal. *libOTe: an efficient, portable, and easy to use Oblivious Transfer Library*. https://github.com/osu-crypto/libOTe.

[190] Michael Rosenberg et al. "zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023.

[191] Dragos Rotaru and Tim Wood. "Marbled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security". In: *International Conference on Cryptology in India*. 2019.

[192] Théo Ryffel et al. "ARIANN: Low-Interaction Privacy-Preserving Deep Learning via Function Secret Sharing". In: *Privacy Enhancing Technologies Symposium (PETS)*. 2022.

[193] Alex Sangers et al. "Secure multiparty PageRank algorithm for collaborative fraud detection". In: *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*. 2019.

[194] Eli Ben Sasson et al. "Zerocash: Decentralized anonymous payments from bitcoin". In: *2014 IEEE symposium on security and privacy*. 2014.

[195] Sinem Sav et al. "POSEIDON: Privacy-Preserving Federated Neural Network Learning". In: *Symposium on Network and Distributed System Security (NDSS)*. 2021.

[196] Phillipp Schoppmann et al. "Distributed vector-OLE: improved constructions and implementation". In: *ACM Conference on Computer and Communications Security (CCS)*. 2019.

[197] *Secure Supply Chain Management*. https://faui1-files.cs.fau.de/filepool/publications/octavian_securescm/SecureSCM-D.9.2.pdf. 2009.

[198] Shaohuai Shi et al. "Benchmarking state-of-the-art deep learning software tools". In: *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*. 2016.

[199] Mohammad Shoeybi et al. "Megatron-lm: Training multi-billion parameter language models using model parallelism". In: *arXiv preprint arXiv:1909.08053* (2019).

[200] Karen Simonyan and Andrew Zisserman. *Very deep convolutional networks for large-scale image recognition*. https://arxiv.org/abs/1409.1556. 2014.

[201] Sparkfun. *Coin Cell Battery - 20mm (CR2032)*. https://www.sparkfun.com/products/338. 2023.

[202] *Swarm Starlink*. https://swarm.space/. 2023.

[203] Sijun Tan et al. "CryptGPU: Fast Privacy-Preserving Machine Learning on the GPU". In: *IEEE Symposium on Security and Privacy (S&P)*. 2021.

[204] Michael Bedford Taylor. "The evolution of bitcoin hardware". In: *Computer* (2017).

[205] Petroc Taylor. *Average cellular data price per gigabyte*. https://www.statista.com/statistics/994913/average-cellular-data-price-per-gigabyte-in-the-us/. 2023.

[206] *The Most Efficient Known Addition Chains for Field Element and Scalar Inversion for the Most Popular and Most Unpopular Elliptic Curves*. https://briansmith.org/ecc-inversion-addition-chains-01. 2017.

[207] Florian Tramer and Dan Boneh. "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware". In: *arXiv preprint arXiv:1806.03287* (2018).

[208] Raylin Tso et al. "Privacy-preserving data communication through secure multi-party computation in healthcare sensor cloud". In: *Journal of Signal Processing Systems* (2017).

[209] Jack Turner et al. "Characterising across-stack optimisations for deep convolutional neural networks". In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 2018.

[210] Nirvan Tyagi et al. "Stadium: A distributed metadata-private messaging system". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017.

[211] "U.K. bars trash cans from tracking people with Wi-Fi". In: *CBS News* (2013).

[212] Frank C. Uyeda et al. "SDN in the stratosphere: loon's aerospace mesh network". In: *ACM SIGCOMM* (2022).

[213] Jelle Van Den Hooff et al. "Vuvuzela: Scalable private messaging resistant to traffic analysis". In: *SOSP*. 2015.

[214] Giorgos Vasiliadis et al. "PixelVault: Using GPUs for Securing Cryptographic Operations". In: *ACM Conference on Computer and Communications Security (CCS)*. 2014.

[215] Deepak Vasisht et al. "FarmBeats: An IoT Platform for Data-Driven Agriculture". In: *Symposium on Networked Systems Design and Implementation*. 2017.

[216] V. Venkataramanan et al. "Forest Fire Detection and Temperature Monitoring Alert using IoT and Machine Learning Algorithm". In: *International Conference on Smart Systems and Inventive Technology* (2023).

[217] Sameer Wagh. "BarnOwl: Secure Comparisons using Silent Pseudorandom Correlation Generators". In: *Tech Report*. 2022.

[218] Sameer Wagh. "New Directions in Efficient Privacy Preserving Machine Learning". PhD thesis. Princeton University, 2020.

[219] Sameer Wagh. “Pika: Secure Computation using Function Secret Sharing over Rings”. In: *Privacy Enhancing Technologies Symposium (PETS)*. 2022.

[220] Sameer Wagh, Paul Cuff, and Prateek Mittal. “Differentially private oblivious RAM”. In: *Privacy Enhancing Technologies Symposium (PETS)*. 2018.

[221] Sameer Wagh, Divya Gupta, and Nishanth Chandran. “SecureNN: 3-Party Secure Computation for Neural Network Training”. In: *Privacy Enhancing Technologies Symposium (PETS)*. 2019.

[222] Sameer Wagh et al. “DP-Cryptography: marrying differential privacy and cryptography in emerging applications”. In: *Communications of the ACM*. 2020.

[223] Sameer Wagh et al. “FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning”. In: *Privacy Enhancing Technologies Symposium (PETS)*. 2021.

[224] Roy Want, Bill N. Schilit, and Scott Jenson. “Enabling the Internet of Things”. In: *Computer* (2015).

[225] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. “Piranha: A {GPU} platform for secure computation”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022.

[226] Jean-Luc Watson et al. “Nebula: A Privacy-First Platform for Data Backhaul”. In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024.

[227] *Welcome to nRF Connect SDK - Amazon Sidewalk*. https://github.com/nrfconnect/sdk-sidewalk. 2023.

[228] Timm A. Wild et al. “A multi-species evaluation of digital wildlife monitoring using the Sigfox IoT network”. In: *Animal Biotelemetry* (2023).

[229] Charles F Xavier. “Pipemsm: Hardware acceleration for multi-scalar multiplication”. In: *Cryptology ePrint Archive* (2022).

[230] Takeshi Yamanouchi. *Chapter 36. AES Encryption and Decryption on the GPU*. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-36-aes-encryption-and-decryption-gpu.

[231] Kang Yang et al. “Ferret: Fast Extension for coRRElated oT with small communication”. In: *ACM Conference on Computer and Communications Security (CCS)*. 2020.

[232] Lei Yang and Fengjun Li. “mTor: A multipath Tor routing beyond bandwidth throttling”. In: *2015 IEEE Conference on Communications and Network Security (CNS)*. 2015.

[233] Andrew Yao. “Protocols for Secure Computations”. In: *Foundations of Computer Science (FOCS)*. 1982.

[234] Andrew Chi-Chih Yao. “On the evaluation of powers”. In: *SIAM Journal on computing* (1976).

[235] *Z-Prize MSM on the GPU*. https://github.com/yrrid/combined-msm-gpu. 2023.

[236] *Zero-Knowledge Proof: Applications and Use Cases*. https://chain.link/education-hub/zero-knowledge-proof-use-cases. 2023.

[237] Ye Zhang et al. “Pipezk: Accelerating zero-knowledge proof with a pipelined architecture”. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021.

[238] Yifei Zhang et al. “A survey of trustworthy federated learning with perspectives on security, robustness and privacy”. In: *Companion Proceedings of the ACM Web Conference 2023*, pp. 1167–1176.

[239] Baoze Zhao et al. “BSTMSM: A High-Performance FPGA-based Multi-Scalar Multiplication Hardware Accelerator”. In: *2023 International Conference on Field Programmable Technology (ICFPT)*. 2023.

[240] Haixu Zhao et al. “Hardware acceleration of number theoretic transform for zk-SNARK”. In: *Engineering Reports* (2022).

[241] Zhichao Zhao and T-H Hubert Chan. “How to vote privately using bitcoin”. In: *Information and Communications Security: 17th International Conference, ICICS 2015, Beijing, China, December 9–11, 2015, Revised Selected Papers 17*. 2016.

[242] Wenting Zheng et al. “Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning”. In: *USENIX Security Symposium (USENIX)*. 2021.

[243] Wenting Zheng et al. “Helen: Maliciously secure coopetitive learning for linear models”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2019.

[244] Xudong Zhu et al. “Elastic MSM: A Fast, Elastic and Modular Preprocessing Technique for Multi-Scalar Multiplication Algorithm on GPUs”. In: *Cryptology ePrint Archive* (2024).

[245] *Zprize 2023*. https://www.zprize.io/. 2023.