Karl Koch, BSc

# b4M */bæm/*

# A Benchmark Framework for SCALE-MAMBA with Applications to MPC-Friendly PRFs and Beyond

**Master's Thesis**

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to
**Graz University of Technology**

**Supervisors**
Dr.techn. Dragoș Rotaru
Univ.-Prof. Dipl.-Ing. Dr.techn. Christian Rechberger

Institute of Applied Information Processing and Communications (IAIK)
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Stefan Mangard

Faculty of Computer Science and Biomedical Engineering

Graz, December 2020

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

This LaTeX template is based on Karl Voit's template [1] and can be found online:
https://github.com/LosFuzzys/LaTeX-KOMA-template

[1] https://github.com/novoid/LaTeX-KOMA-template

## Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

| | |
|---|---|
| Date | Signature |

# Abstract

Nowadays data becomes more and more prevalent. With the prevalence of data storage in the cloud, privacy becomes more and more important too. Usually users do not want that everyone can access their data, and, moreover, they would like to control who has access. However, in certain cases users have to share their data in order to have a specific functionality; like computing something together with other parties. Such scenarios are called multi-party computation. Multi-party computation with a focus on security & privacy is called secure multi-party computation (MPC).

In today's world, pseudo-random functions (PRFs) are an important building block. Given an input and key, a secure PRF provides an output which is indistinguishable from an output of a truly random function. One such use case for PRFs is in the context of privacy-preserving searchable symmetric encryption. And the challenge of key exchange in this use case can be solved by leveraging PRF evaluations in MPC. As described by [Gra+16], PRFs which allow to perform evaluations in MPC efficiently, are so-called *MPC-friendly PRFs*. This new requirement for PRFs heralds the dawn of a new era: *MPC-friendly cipher design*.

Normally, new ciphers are created analytically based on their requirements. Then, when a new cipher has been created, the cipher has to be benchmarked in the envisioned environments. This benchmarking allows to (1) evaluate the general performance of the cipher and (2) compare the cipher with other ciphers in this area, based on sound metrics; such as the runtime of the execution. Though, the benchmarking of several MPC programs, like several PRFs, considering MPC-related metrics simulated in different environments with various settings, has to be done manually and is a cumbersome process; up until now ☺.

The main contributions of this thesis are fourfold: (1) Creation of the benchmark framework *Benchmarking for MPC* (b4M) for the MPC-engine SCALE-MAMBA; (2) Identification of relevant benchmark metrics (i.e. requirements), for (a) MPC programs in general and also (b) specifically for *MPC-friendly PRFs*; (3) Identificaton of interesting and (potentially) relevant benchmark settings and questions; (4) Benchmarking and evaluation of the two use cases (a) 3-players with (basically) no network restrictions (local area network (LAN)), and (b) 3-players which are connected within a limited network (wide area network (WAN)). Moreover, with respect to future work, we identified interesting paths to follow.

# Kurzfassung

Heutzutage ist die Generierung, Speicherung und Verarbeitung von Daten immer weiter verbreitet. Mit der Verbreitung der Datenspeicherung in der Cloud wird die Privatsphäre immer wichtiger. Normalerweise möchten Benutzer nicht, dass jede/r auf ihre Daten zugreifen kann, und sie möchten außerdem steuern wer Zugriff hat. In bestimmten Fällen müssen Benutzer jedoch ihre Daten freigeben, um eine bestimmte Funktionalität zu erhalten; wie etwas zusammen mit anderen Personen zu berechnen. Solche Szenarien werden als *Multi-Party Computation* bezeichnet. Multi-Party Computation mit Schwerpunkt auf Sicherheit und Datenschutz wird weiters als *Secure Multi-Party-Computation* (MPC) bezeichnet.

In der heutigen Welt sind Pseudo-Zufalls-Funktionen (PRFs) ein wichtiger Baustein. Bei der Eingabe von Daten und einem Schlüssel liefert eine sichere PRF eine Ausgabe, die nicht von der Ausgabe einer wirklichen Zufalls-Funktion zu unterscheiden ist. Ein solcher Anwendungsfall für PRFs ist die privatsphärenschützende Suche von verschlüsselten Daten via symmetrischer Verschlüsselung. Die Herausforderung des Schlüsselaustauschs in diesem Anwendungsfall, kann durch die Nutzung von PRF-Berechnungen in MPC gelöst werden. Wie von [Gra+16] beschrieben, sind PRFs, mit denen Auswertungen in MPC effizient durchgeführt werden können, sogenannte *MPC-freundliche PRFs*. Diese neue Anforderung an PRFs läutet den Beginn einer neuen Ära ein: *MPC-freundliches Chiffrendesign*.

Normalerweise werden neue Chiffren basierend auf ihren Anforderungen analytisch erstellt. Wenn dann eine neue Chiffre kreiert wurde, wird sie in den vorgesehenen Umgebungen getestet. Dieses Testen ermöglicht es, (1) die allgemeine Leistung der Chiffre zu bewerten und (2) sie mit anderen Chiffren in diesem Bereich basierend auf fundierten Metriken zu vergleichen; wie etwa die Laufzeit der Ausführung. Das Testen mehrerer MPC-Programme, wie verschiedene PRFs, unter Berücksichtigung von Metriken bezogen auf

MPC, die in verschiedenen Umgebungen mit unterschiedlichen Einstellungen simuliert werden, muss jedoch manuell durchgeführt werden und ist ein mühsamer Prozess; bis jetzt ☺.

Die Hauptbeiträge dieser Arbeit beziehen sich auf vier Bereiche: (1) Erstellung des Test-Frameworks *Benchmarking für MPC* (b4M) für das MPC-Framework SCALE-MAMBA; (2) Identifizierung relevanter Test-Metriken, die sogenannten Anforderungen, für (a) MPC-Programme im Allgemeinen und (b) speziell für *MPC-freundliche PRFs*; (3) Identifizierung interessanter und (potenziell) relevanter Test-Einstellungen und -Fragen; (4) Testen und Bewerten der beiden Anwendungsfälle (a) 3-Parteien mit (grundsätzlich) keinen Netzwerkeinschränkungen (lokales Netzwerk (LAN)) und (b) 3-Parteien, die innerhalb eines begrenzten Netzwerks verbunden sind (Weitverkehrsnetzwerk (WAN)). Darüber hinaus haben wir im Hinblick auf weiterführende Arbeit interessante Pfade identifiziert.

# Contents

# List of Acronyms

**AES** Advanced Encryption Standard

**ASX** access structure . . . . . 70

**AVX** advanced vector extensions . . . . . 128

**b4M** Benchmarking for MPC . . . . . 130

**BFN** balanced Feistel network . . . . . 24

**CRF** contracting round function . . . . . 23

**ERF** expanding round function . . . . . 127

**FHE** fully homomorphic encryption . . . . . 53

**GCD** greatest common divisor . . . . . 21

**GMiMC** Generalized Feistel MiMC . . . . . 127

**GUI** graphical user interface . . . . . 128

**HMiMC** HADES MiMC . . . . . 127

**IAIK** Institute of Applied Information Processing and Communications . . 67

**IoT** internet of things . . . . . 44

List of Acronyms

**KB** kilobytes . . . . . 61

**LAN** local area network . . . . . 130

**Leg** Legendre . . . . . 72

**LowMC** Low Multiplicative Complexity . . . . . 16

**MB** megabytes . . . . . 62

**MDS** maximum distance separable . . . . . 29

**MiMC** Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity . . . . . 72

**MPC** secure multi-party computation . . . . . 130

**MRF** multi-rotate round function . . . . . 25

**ms** milliseconds . . . . . 60

**NGF** Nyberg's generalized-feistel-network round function . . . . . 24

**NR** Naor-Reingold . . . . . 16

**P-SPN** partial substitution-permutation network . . . . . 28

**PRF** pseudo-random function . . . . . 130

**PRG** pseudo-random generator . . . . . 35

**SCALE** SCALE-MAMBA . . . . . 130

**SIMD** Single Instruction, Multiple Data . . . . . 127

**SNARKs** Succinct Non-interactive Arguments of Knowledge . . . . . 17

# List of Figures

# List of Tables

# List of Protocols

# Chapter 0

# Mulțumesc

First and foremost I want to thank my supervisors **Christian** and **Dragoș** for their invaluable support, patience, feedback, kindness, and belief. As well as **Denise**, **my Mom**, and all the other **invaluable family members and friends** for their unwavering support where they encouraged me to continue striving towards the completion of this thesis during my hardest times.

Furthermore I want to thank:

**Nigel Smart** for the opportunity to do parts of this thesis at the research centre COSIC in Leuven, Belgium.

**The respecitve authors** of the pseudo-random functions (PRFs) Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity (MiMC), Legendre (Leg), Generalized Feistel MiMC (GMiMC), and HADES MiMC (HMiMC) for providing the MAMBA code for this thesis' benchmarks.

**Philip and Sharif**, (former) the Institute of Applied Information Processing and Communications (IAIK)'s sysadmins, for installing the two dedicated secure multi-party computation (MPC)-friendly networks on IAIK's cluster, which allowed benchmarks in a local area network (LAN) and simulated wide area network (WAN) environment.

**Amit, Tadesse, Pedro, and Elisson** for their invaluable support in Leuven.

**Karl-Christian Posch (aka. KC)**, **Daniel Hein**, **Johannes Winter**, **Peter Teufl**, and all the other great lecturers and researchers at IAIK for showing me the exciting world of IT Security.

**All Fuzzys** for the great time in the *FuzzyLab* et al. during CTFing, brainstorming, oxcafe talks, etc.

**"Party"** (as in multi-*party* computation ☺) songs listened to during the creation of this thesis; include, but are not limited to: Black Eyed Peas - **Don't stop the party**. As well as **Eminem's Not Afraid** (although please note that this one is not a "party" song).

# Chapter 1

# Introduction

**Data prevalence & cloud storage.** Nowadays data becomes more and more prevalent. For instance, many people use their smartphones to track activities, such as sports, food, or general tasks. This tracking is, however, not limited to smartphones. Activities can also be tracked with smartwatches, smartglasses, or just by entering it on a laptop or the more "static" Personal Computer (PC). What all of these devices have in common is their access to storage on another entity's computer via the internet; this "internet storage" is referred to as the cloud. Many people use the cloud for data storage, as it enables convenient use cases. One such use case is that data is accessible on all the users' devices, not on only one.

**Privacy & (secure) multi-party computation.** With the prevalence of data storage in the cloud, privacy becomes more and more important too. Usually users do not want that everyone can access their data, and, moreover, they would like to control who has access. However, in certain cases users have to share their data in order to have a specific functionality; like sharing data with friends or computing something together with other parties. One example for such a joint computation is to calculate the average age in an audience. Everyone has to give data, the age, and someone is computing the average. Such scenarios are called multi-party computation. In terms of privacy, people are usually interested to keep their input private; like the person's age in the mentioned example of the audience's average age. Multi-party computation with a focus on security & privacy is called *secure multi-party computation*.

**(Secure) multi-party computation based on a TTP.** To realize (secure) multi-party computation in the naive way, one could think of a trusted third party (TTP). This TTP receives the input of all participating parties, and responds with the result. Figure 1.1 illustrates the appraoch using a TTP with three parties. This works fine as long as the TTP (1) behaves honestly and (2) no data leaks, e.g., due to a hack. In January 2018 a group of researchers publicly announced the critical vulnerabilities *Meltdown* and *Spectre*; with which it is possible to steal data from other applications, in certain circumstances even when they are operating in a cloud environment [Gra20; Lip+18; Koc+19]. Thus, the solution with the TTP works, in principle, but might not always achieve the goal of security & privacy of the parties' data. Can we do better than that? Yes, we can.

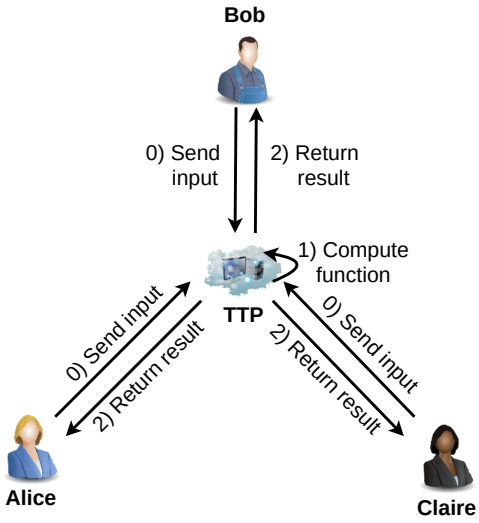


Figure 1.1: (Secure) multi-party computation based on a TTP with three parties.

**Secure multi-party computation based on secret sharing.** With concepts such as secret sharing, one can achieve secure multi-party computation while having better *guarantees* to ensure data privacy. In a secret-sharing scheme, each party splits its secret value into several parts. These parts are shared with the participating parties. Then, each party computes a sum of their parts and shares this with the other parties. Now, each party can compute the overall

sum of the secret values without knowing the specific values. This example shows the case for secure addition, without relying on a TTP. Figure 1.2 illustrates the approach using secret sharing with three parties. Though, if parties cheat and share more than their partial sums, it is possible to recover the secrets of the individual parties. There exist different attack scenarios, as well as different countermeasures, which are shown later on.

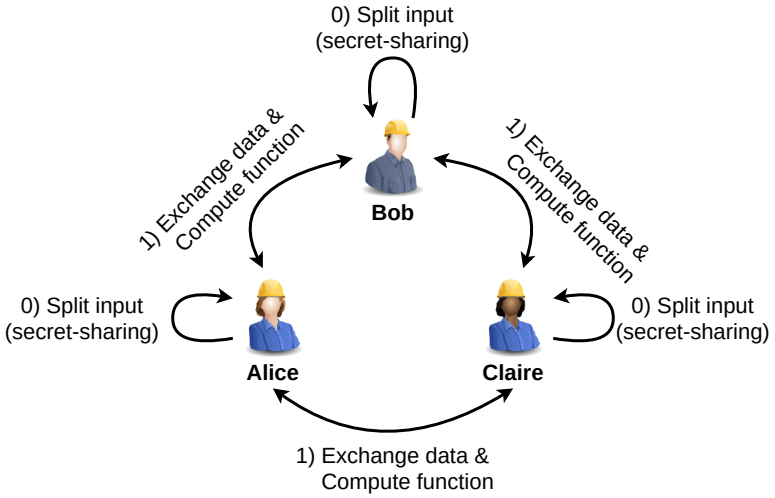


Figure 1.2: Secure multi-party computation based on secret sharing with three parties.

Throughout the rest of this thesis, when we say secure multi-party computation (MPC), we refer to approaches where no TTP is needed and parties compute the function collaboratively, such as the mentioned secret-sharing-based approach.

**Applications for secure addition.** Already secure addition on its own enables useful applications. For instance, in the context of "simple" voting, where people vote for *yes* or *no*. A *yes* could be represented as a 1, and a *no* could be represented as a 0. Now, the computed sum represents the amount of votes for *yes*, without revealing who voted for *yes* or *no*. However, one could do "logical" attacks by analyzing the sum. This reveals, e.g., how many people voted for *yes*. On the other side, this is probably intended as part of the protocol, as people usually want to know how many people voted for *yes*, and how many

voted for *no*. And if there are, e.g., only two parties, one gets to know, of course, the input of the other party.

Another context for secure addition is computing the average. One could think of a presentation where the speaker would like to announce the average age of the audience, as was shown before. Usually, not everyone likes to publicly announce their age. With the approach of secure addition, as described above, each party can privately compute the sum of all people in the audience, and then divides the result by the amount of people. This approach can also be applied to, e.g, compute the average salary of employees in a certain field among several companies.

**Programs evaluated in MPC.** By leveraging secure addition, among other things, we are able to evaluate "normal" programs in MPC. For instance, with this approach we can write a program which outputs a secret-shared value. Now, $c$ parties would need to get corrupted or collude in order to get the plaintext value of the result. Whereas, depending on the underlying MPC protocol, $c$ can either be some of the participating parties, or even all.

**Pseudo-random functions & their evaluation in MPC.** In today's world, pseudo-random functions (PRFs) are an important building block [Bon+18]. Given an input and key, a secure PRF provides an output which is indistinguishable from an output of a truly random function [Gol01]. For instance, when we access a website in a secure way, thus using tansport layer security (TLS) (the *s* in *https*), the content is (normally) encrypted using a symmetric block cipher [DA99]. Currently, one of the most common block ciphers is the Advanced Encryption Standard (AES) [DR98]. And block ciphers like AES can also be seen as PRFs.

Another use case for PRFs is in the context of privacy-preserving searchable symmetric encryption (SSE) [SWP00; Goh03; Cur+06]. In the (currently) typical way, a client uploads along with some encrypted data, also an encrypted index for these data items to a server. Now, when a client wants to search on the encrypted data on the server, essentially four steps are necesary:

0. as a note, the client used key $k$ to create the encrypted index;

1. the client locally encrypts the keyword $kw$, using a PRF such as AES with a key $k$: $kw_e = Enc_k^{AES}(kw)$;
2. then, the client sends the encrypted keyword $kw_e$ to the server;
3. now, the server uses the encrypted keyword $kw_e$ to search on the encrypted index for matching data items;
4. finally, the server responds with the matching data items.

This way, clients can search on their encrypted data items while the server does not learn the corresponding keywords.

Though, there is one drawback with this SSE approach. When a client, say Dave, shares his data items on the server with another client, say Emma, Emma also needs key $k$ in order to search remotely on Dave's encrypted data items. Thus, there is the challenge of secure key exchange; and the challenge grows linearly with the number of "shared clients". Figure 1.3 illustrates the typical SSE approach.

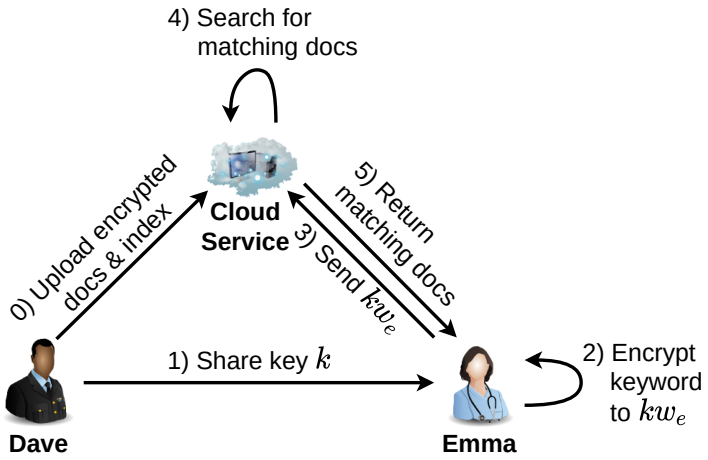


Figure 1.3: The typical multi-user SSE approach, where the data owner (Dave) needs to share a key with other users (e.g. Emma).

One solution to this challenge is provided by leveraging MPC. Specifically, PRF evaluations in MPC, and it works as follows:

1. Dave uploads some encrypted data items with an encrypted index, using key $k$;

2. Dave splits $k$ into $n$ parts onto $n$ MPC nodes;
3. Dave shares the encrypted data items with Emma. And now, Emma wants to search on Dave's encrypted data for keyword $kw$;
4. Emma splits $kw$ into $n$ parts and sends each part to the corresponding MPC node;
5. the MPC nodes compute the encrypted keyword $kw_e$, and each node returns $kw_{e_i}$ to Emma;
6. finally, Emma can reconstruct the different parts to the encrypted keyword $kw_e$, and as in the typical way, Bob can use $kw_e$ to search on Dave's encrypted data.

In the "MPC way", the challenge of secure key exchange is shifted to trusting MPC nodes. And there exist strong MPC protocols which guarantee security even when all nodes bar one are malicious. However, this challenge does not grow linearly with the number of "shared clients". Furthermore, as "shared clients" do not get the key $k$, revocation of access might be easier; for instance, it might not be necessary to re-encrypt the index with another key $k'$. Figure 1.4 illustrates the multi-user SSE approach using MPC.

Also, please note that we do not address the actual decryption of shared data items. Though, as with the encryption of the keyword $kw$, MPC can also be leveraged to compute the decryption of data items without knowing $k$. Moreover, this kind of distributed PRF evaluation can be interesting for many applications. Actually, whenever a PRF is evaluated and $k$ needs to be shared. For instance, to search for items in an encrypted database, which reflects the previously mentioned application of searching on encrypted data items. Or even for a single user to have a secure (secret-shared) backup of $k$.

More general, in the MPC setting the input, the key, or the output of a PRF evaluation might be secret-shared, or even all of them. As described by [Gra+16], PRFs which allow to perform evaluations in MPC efficiently, are so-called *MPC-friendly PRFs*. This new requirement for PRFs heralds the dawn of a new era: *MPC-friendly cipher design*.

**The challenge: ease of evaluations.** Normally, new ciphers are created analytically based on their requirements. These requirements comprise (usually) efficiency and security. Then, when a new cipher has been created, the cipher

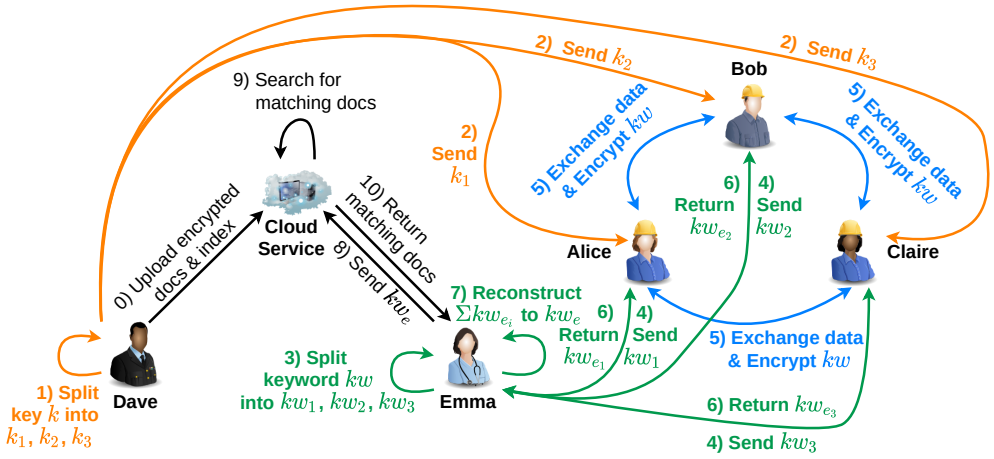


Figure 1.4: The multi-user SSE approach leveraging MPC, where the data owner (Dave) splits the key among $n$ MPC nodes. In this example $n = 3$. Other users which got access to the data (e.g. Emma), can request an encrypted keyword from the MPC nodes. Please note that the nodes share only parts of the data; so if *enough* nodes are honest, they do not get to know the plain data. How many *enough* is, depends on the underlying protocol.

has to be benchmarked in the envisioned environments. This benchmarking allows to (1) evaluate the general performance of the cipher and (2) compare the cipher with other ciphers in this area, based on sound metrics; such as the runtime of the execution.

[Gra+16] benchmarked five PRFs which were evaluated in MPC. AES is very efficient for plaintext data. However, in the MPC setting, AES does not perform so well; especially when compared to other PRFs, like Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity (MiMC) or Legendre (Leg). In their paper, [Gra+16] compared the different PRFs in a local area network (LAN) and a wide area network (WAN) using two players, one underlying MPC protocol, and some parallel execution. Now, it would be interesting to see the performance of the PRFs also for, e.g., three players, as well as different protocols and settings; like comparing the different parallel executions. Though, this benchmarking of different settings usually works like this:

1. Manually...
   a) ...compile the program (one PRF) with the right settings;
   b) ...execute the program in the right environment, like two players;
   c) ...parse the program-execution's output;
   d) ...collect the data of the different metrics;
   e) ...visualize the results by, e.g., creating a graph.
2. Repeat this manual process for all combinations of PRFs, environments, and settings.

Another challenge in the area of *MPC-friendly cipher design* is to get a clear overview of the different metrics and potential environments for programs evaluated in MPC. These environments reflect use cases in potential real-life scenarios.

Thus, the benchmarking of several MPC programs, like several PRFs, considering MPC-related metrics simulated in different environments with various settings, is a cumbersome process; up until now ☺.

## 1.1 Goals & Contributions of this Thesis

Solving the identified challenges of the previous section is the primary goal of this thesis. Hence, we aim to (1) ease the cumbersome process of benchmarking different settings and (2) shed more light on MPC-related metrics and (potentially) relevant settings. Next to this primary goal, we also aim to showcase the results in a restricted environment with a subset of the identified settings.

By achieving these goals, our main contributions of this thesis are fourfold:

1. Creation of the benchmark framework *Benchmarking for MPC (b4M)* for the MPC-engine SCALE-MAMBA (SCALE);
2. Identification of relevant benchmark metrics (i.e. requirements), for
   a) programs evaluated in MPC in general and also
   b) specifically for *MPC-friendly PRFs*;
3. Identificaton of interesting and (potentially) relevant benchmark settings and questions;

4. Benchmarking and evaluation of the two use cases
   a) 3-players with (basically) no network restrictions (LAN), and
   b) 3-players which are connected within a limited network (WAN).

## 1.2 Outline

The story of this thesis continues as follows:

- In **Chapter 2**, *Secure Multi-Party Computation (MPC)*, we **give** some background information and further references to MPC.
- In **Chapter 3**, *Benchmarking PRFs for MPC*, we **present** the selected PRFs for showcasing the benchmark framework, **identify** relevant MPC-related metrics (i.e. requirements) and benchmark dimensions, **show** our approach for tackling the benchmarks for this thesis, and, finally, we **introduce** our created benchmark framework *Benchmarking for MPC (b4M)*.
- In **Chapter 4**, *Performance Evaluation & Recommendations*, we **describe** our benchmark environment, **give** a comparison with the benchmarks of the paper by [Gra+16], and further **showcase** b4M with two 3-player use cases; one in a local area network (LAN), and one in a wide area network (WAN).
- In **Chapter 5**, *Future Work*, we **outline** potential future work in the area of (1) the (identified) benchmark metrics, dimensions, and questions, (2) the benchmark framework b4M, and (3) PRF evaluations in MPC.
- In **Chapter 6**, *Conclusion*, we **conclude** this thesis.

# Chapter 2

# Secure Multi-Party Computation (MPC)

This chapter briefly mentions essential building blocks for secure multi-party computation (MPC).

## 2.1 Definitions & Notations

Throughout this thesis, we note the following definitions and notations:

- "$\xleftarrow{\$}$" means to get a value uniformly at random.
- Secret-shared variables are written with square brackets; for instance, $[s]$ denotes the secret-shared representation of the variable $s$.
- "XORing" in this thesis means to simply add two elements in $\mathbb{F}_p$.
- Programs evaluated in MPC, thus, e.g., the MAMBA bytecode for SCALE-MAMBA (SCALE), are throughout the thesis simplified called *MPC programs*. Please see Section 2.6 for further details on SCALE's approach.
- MPC nodes participating in the distributed computation are called *players*.

## 2.2 Secret-Sharing-based MPC

As MPC based on Shamir's secret sharing is one of the most common MPC techniques, in this focus we mainly focus on this approach. There exist different variants of it; for instance Linear Secret Sharing, Additive Secret Sharing, Replicated Maurer, or Replicated Reduced.

## 2.3 Security Models

In terms of security, we mainly distinguish between active and passive security models in MPC.

**Active Security.** In an active security model, we assume that corrupted parties actively disturb the protocol. One way of disturbing is to send malformed input to another party. Another way of disruption is to abort the message and send nothing at all to other parties.

**Passive Security.** In a passive security model, we assume that corrupted parties behave like in the *semi-honest* model. In the *semi-honest* model no party disturbs the protocol, they try to get as much information as possible from the data they receive anyways.

**Player corruptions.** There exist different ways how various player corruptions are reflected in the description of MPC protocols. For instance, $t$-threshold, $Q2$, $Q3$, or *fully-malicious*. $t$-threshold describes the security for an MPC protocol of up to $t$ corruptions.

The security models of MPC are further described in Section 3.2.4.

## 2.4 Protocols

There exist different approaches, so-called protocols, to tackle MPC. In this thesis, we use the Smart-Pastro-Damgård-Zakarias (SPDZ) protocol.

## 2.5 Circuits

MPC programs can either be evaluated in an arithmetic, garbled, or MArBled (mixed arithmetic and boolean) circuit.

## 2.6 MPC Engines

There exist several MPC engines, http://www.multipartycomputation.com/mpc-software and https://github.com/rdragos/awesome-mpc give a comprehensive overview of the different engines and their properties. In this thesis, we use the MPC-engine SCALE-MAMBA (SCALE).

# Chapter 3

# Benchmarking PRFs for MPC

This chapter first describes our selected pseudo-random functions (PRFs) for benchmarking and the requirements on the execution of a secure-multi-party-computation (MPC) program. Then, based on the requirements, we show our *benchmark dimensions* and our benchmarking plan. Finally, we describe our benchmarking framework Benchmarking for MPC (b4M).

## 3.1 Selection of PRFs

This section describes our selected PRFs for benchmarking. First, we describe the criteria and process of selecting the PRFs for this thesis. Then, we describe for each PRF (1) briefly the background, (2) the used approach, (3) potential variants, if they exist, (4) suggested security parameters, and (5) how the PRF is evaluated using MPC.

### 3.1.1 Choosing PRFs

**The baseline.** As the paper by [Gra+16] serves as starting point for this thesis, the used PRFs in this paper also serve as starting point for the selection of PRFs. The main focus of the paper is to benchmark primarily ciphers operating on $\mathbb{F}_p$ natively, for large p. Such ciphers are well suited for arithmetic circuits, and are more costly to realize with garbled circuits [KPR18].

When MPC started to become practical, many people used the Advanced Encryption Standard (AES) [DR98] to perform benchmarks of PRF evaluations

in MPC. Though, it was not designed for use cases of, e.g., a low multiplicative depth, and thus is not a natural choice for MPC. The PRF Low Multiplicative Complexity (LowMC) [Alb+15], on the other hand, was designed for, e.g., a low multiplicative depth (hence the name LowMC ☺). But both AES as well as LowMC operate natively on $\mathbb{F}_{2^8}$ and $\mathbb{F}_2$ respectively. Because of these facts, [Gra+16] used AES and LowMC primarily only as reference comparison to the, back then, state of the art of PRFs for MPC. Thus, we do not include AES and LowMC in the benchmarks of this thesis.

Furthermore, [Gra+16] used the elliptic-curve variant of Naor-Reingold (NR) [NR97], and SCALE-MAMBA (SCALE) does not support elliptic curves out of the box. Hence, we do not include NR either. Therefore, our baseline PRFs are: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity (MiMC) [Alb+16] and Legendre (Leg) [Dam88; Gra+16].

**The successors.** At the time of writing this thesis, MiMC has two successors: Generalized Feistel MiMC (GMiMC) [Alb+19b] and HADES MiMC (HMiMC) [Gra+20]. As these succesors further improved in the area of $\mathbb{F}_p$, for large p, we include them for the benchmarks in this thesis.

**Further PRFs.** Apart from the mentioned PRFs, there exist of course more. For instance, the *encoded-input* PRF by [Bon+18], or Rescue [Aly+19]. Though, for showcasing b4M and giving example evaluations, we limit ourselves to the aforementioned PRFs.

Thus, our list of selected MPC-friendly PRFs for benchmarks of this thesis comprises:

- MiMC,
- GMiMC,
- HMiMC, and
- Leg.

### 3.1.2 Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity (MiMC)

**Background.** LowMC was published at EuroCrypt 2015 and was one of the first ciphers which was specifically designed for areas where a low multiplicative depth in the computation circuit is important. Such areas are, e.g., MPC and Succinct Non-interactive Arguments of Knowledge (SNARKs). LowMC was able to outperform the competitors at that time. However, there were still little to no ciphers which natively support operations on $\mathbb{F}_p$, for large primes $p$. Though, this property is especially relevant for MPC when computing in an arithmetic circuit. This need led to the initial idea of MiMC [Alb+16] (Asiacrypt 2016); thus, somehow it can be seen as a kind of successor of LowMC (there is also the similarity in the name ☺).

**Approach.** MiMC provides three cryptographic primitives: (1) a block cipher (PRF), (2) a permutation, and (3) a (permutation-based) hash function. The primitives operate either in $\mathbb{F}_p$, where $p$ is a prime, or in $\mathbb{F}_{2^n}$. In this thesis we focus on the block-cipher primitive operating in $\mathbb{F}_p$.

In order to achieve the low multiplicative depth for large $\mathbb{F}_p$, MiMC takes up a (relatively old) design idea by Knudsen-Nyberg (from 1995); namely to use $F(x) = x^3$ as round function [NK95]. This round function has the advantage that it does not need too many multiplications, as in contrast to using, e.g., an Sbox. One condition for the exponent, in this case 3, and the prime $p$ is: $gcd(3, p-1) = 1$. This condition ensures that we get a full permuation, which is required for the security guarantees of the PRF.

MiMC takes as input (plaintext) an element in $\mathbb{F}_p$. The key and round constants, as well as the output (ciphertext), are in $\mathbb{F}_p$ too. In each round, first, the current state, key, and a random round constant get XORed; and second, the cubing round function is applied to update the current state. The state is initially the plaintext and the first round constant is 0; whereas the other round constants $c_i$ are chosen uniformly at random ($c_i \xleftarrow{\$} \mathbb{F}_p$). When the defined number of rounds have been performed, the key is XORed to the current state one more time to give us the ciphertext. One encryption of MiMC

looks then as follows:

$$\begin{aligned} y_0 &= (x \oplus k \oplus c_0)^3 \\ y_i &= (y_{i-1} \oplus k \oplus c_i)^3 \\ & \quad i = 1, \ldots, r-1 \\ y \text{ (= the output)} &= y_{r-1} \oplus k \end{aligned} \tag{3.1}$$

whereas the XORing of the constant $c_0$ is grayed out because the value of $c_0$ is 0, and $r$ denotes the number of rounds. Figure 3.1 represents the encryption process.

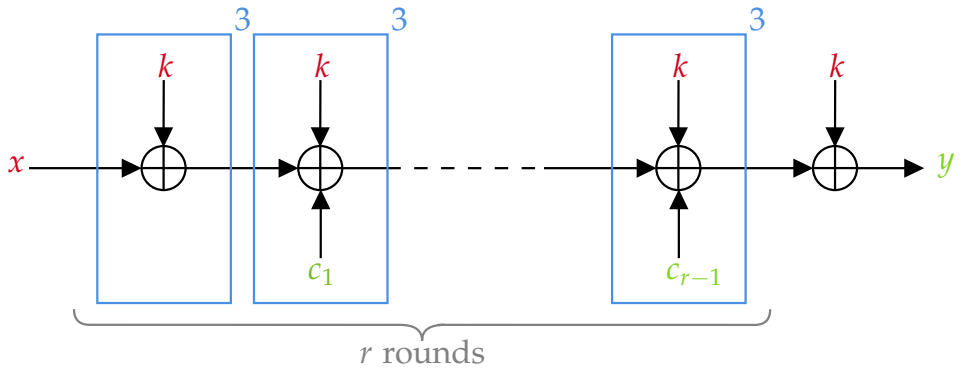


Figure 3.1: One encryption of MiMC. The first round has no constant because $c_0$ is defined to be 0.

**Variants of MiMC.** Besides the aforementioned approach of MiMC, there exist also other, slightly modified, variants. One of these approaches is to use a **Feistel network**. When we use a Feistel network, in each round two plaintext blocks get processed. Due to this "double processing" of plaintext blocks, the number of rounds need to be doubled too. The encryption process works basically the same, only the round function is different:

$$x_L || x_R \rightarrow (x_L \oplus k \oplus c_i)^3 \oplus x_R || x_L \tag{3.2}$$

Figure 3.2 represents the encryption process of MiMC-Feistel. As we have for each round one more XOR and the number of rounds need to be doubled,

the encryption process should be slightly slower than the regular variant. However, when we need to perform a decryption, MiMC-Feistel is likely faster because for MiMC-Regular we need to compute multiplicative inverses. The multiplicative inverse of $x^3$ is slower because they need more computation steps than the "simple" cubing:

$$x = y^{\frac{2^{n+1}-1}{3}} \tag{3.3}$$

For more details on computing the inverse, please refer to the original MiMC paper [Alb+16].

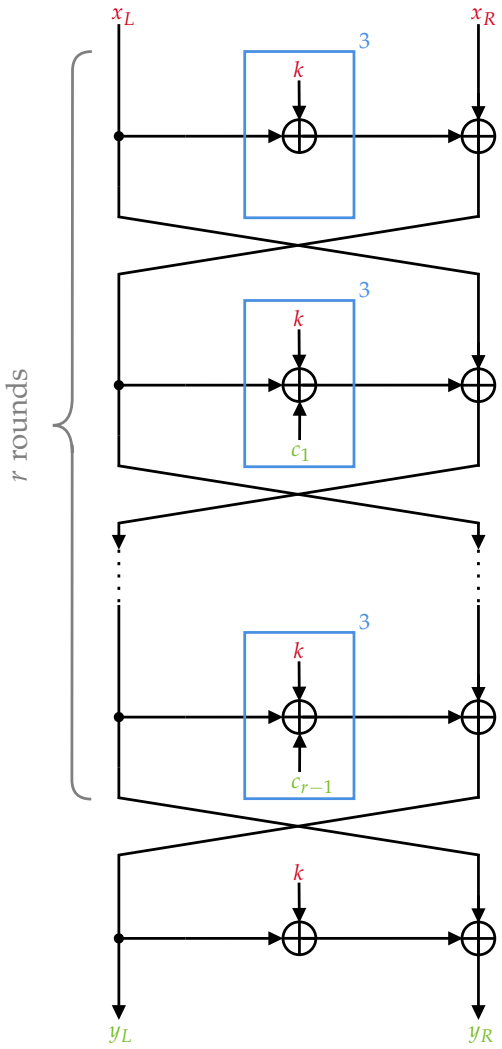


Figure 3.2: One encryption of MiMC-Feistel. The first round has no constant because $c_0$ is defined to be 0.

Another variation is to use an **extended key size**. This extension does not use the same key for each round, but introduces a larger one; for instance, $\epsilon$-times larger. Then, in the individual rounds, the keys get chosen iteratively over

these $\epsilon$ keys. One round with an extended key size looks like this:

$$F_i(x) = (x \oplus k_{i \bmod \epsilon} \oplus c_i)^3 \tag{3.4}$$

Since these additional keys only add one more modulo operation and a lookup (e.g., in a table) for each XOR with the key, the extended key size should not significantly increase the runtime. On the other hand, the extended key size makes certain algebraic attacks more difficult; such as the greatest-common-divisor (GCD) attack.

A further approach is to use **different round functions**:

$$F_i(x) = (x \oplus k \oplus c_i)^d \tag{3.5}$$

Whereas [Alb+16] were limiting themselves to specific values of $d$. $d$ has to be either $(2^t - 1)$ or $(2^t + 1)$, and $t$ has to be a positive integer. However, they also state that for $\mathbb{F}_p$, these kind of exponents do not offer an advantage over $d = 3$. Furthermore, as with $d = 3$, the condition $gcd(d, p - 1) = 1$ has to be fulfilled to guarantee full permutation. This condition holds for MiMC-Regular, but not for MiMC-Feistel. For further details on the choice of the exponent $d$, please refer to the original MiMC paper [Alb+16].

**Security parameters.** In the context of MPC, the selection of appropriate security parameters is a twofold process. First, we choose the parameters for MPC; such as the prime size or amount of shares. Second, based on the choice of Step One, we choose the parameters for the PRFs.

A common prime size for MPC programs is 128-bit.For MiMC, we have to choose the number of rounds $r$ accordingly. We do this based on the analysis by [Gra+16] (the baseline paper). If there is no restriction on the amount of encryptions for an attacker: $r = \lceil log_3(p) \rceil$. Otherwise, if we can restrict the amount of encryptions $n$ for an attacker, such that $n < p$:

$$r = max\{\lceil log_3(n) \rceil, \lceil log_3(p) - 2log_3(log_3(p)) \rceil\} \tag{3.6}$$

Since the introduction of MiMC, its security has been further evaluated. For instance, [Eic+20] successfully attacked all rounds of the $\mathbb{F}_{2^n}$-variant of MiMC. As a consequence, the number of rounds for the $\mathbb{F}_{2^n}$-variant needs to be increased. However, please note that for the variant used in this thesis, the $\mathbb{F}_p$-variant of MiMC, the security is still intact (so far).

**Main applications.** The main intended application of MiMC was for SNARKs [Ben+13]; by using MiMC as hash function. This is interesting, e.g., for the digital privacy-preserving currency Zerocash [Ben+14]. Another interesting area for this new kind of cipher design is Scalable Transparent ARgument of Knowledge (STARK) [Ben+18]. From August 2019 until March 2020, StarkWare [Sta20a] ran the *STARK-friendly hash challenge* [Sta20b; BGL20] to find a suitable (new) cipher meeting the requirement of being *STARK-friendly*. MiMC's successors, GMiMC and HMiMC, were two of the three candidate families for this new *STARK-friendly* hash primitive. Moreover, as shown by [Gra+16], MiMC is also a competitive block cipher in the area of MPC.

**MiMC for MPC.** With regard to actually implementing the PRF, [Gra+16] showed that there is more than one way; at least for secret-sharing-based MPC. One cubing ($x^3$) can be realized either in the basic, or "naive", way, or in the "cube" way.

In the basic way, we first compute the square: $x^2 \leftarrow x \cdot x$. Then, we compute the cube: $x^3 \leftarrow x^2 \cdot x$. Thus, we have one squaring and one multiplication per round, which gives us for $r$ rounds: $3 \times r$ *openings in* $2 \times r$ *rounds of communication*.

In the "cube" way, we first precompute for each round the following 3-tuple: $([t], [t^2], [t^3])$, whereas $t \xleftarrow{\$} \mathbb{F}_p$. Then, in the actual round, to compute the cube of the input $x$, we follow a two-step approach:

1. Compute and open an intermediate value $tmp$, which serves as hidden representation of $x$:
   $tmp = open(x - t)$
2. Compute the secret-shared representation of $x^3$:
   $[x^3] = (3 \times tmp \times [t^2]) + (3 \times tmp^2 \times [t]) + (tmp^3 + [t^3])$,
   which can be computed essentially linearly in the Smart-Pastro-Damgård-Zakarias protocol (SPDZ).

With the "cube" way in SPDZ, we need for the precomputation $2 \times r$ openings in $r$ rounds of communication, when we take a 2-tuple $([t], [t^2])$ and compute: $[t^3] \leftarrow [t] \times [t^2]$. For the actual round, we essentially only need 1 opening. Thus, when we do not consider the negligible communication cost during the

actual round, we end up in total with $3 \times r$ *openings in r rounds of communication*. Hence, *r rounds fewer communication than in the basic way*.

### 3.1.3 Generalized Feistel MiMC (GMiMC)

**Background.** Due to the success of MiMC, further similar approaches with respect to Feistel networks have been tried out. This follow-up work of MiMC-Feistel, led to the birth of GMiMC [Alb+19b] (ESORICS 2019).

**Approach.** The standard version of GMiMC uses the same cubing round function as MiMC, $F(x) = x^3$, but different Feistel structures. Due to these Feistel structures the (approximately) doubling of the number of rounds $r$ is not required anymore; $r$ is much smaller. And because of these fewer rounds GMiMC is even more competitive than MiMC.

**Variants of GMiMC.** The different variants of GMiMC are mostly reflected in the different Feistel structures. Specifically, [Alb+19b] introduced four Feistel structures for GMiMC. Other possibilities of variations lie in the area of using different key schedules or different exponents for the round functions.

One of the four Feistel structures is **a contracting round function (CRF)**. GMiMC-CRF makes use of an unbalanced Feistel network (UFN), and can be written as:

$$\begin{gathered}(X_{t-1}, X_{t-2}, \ldots, X_1, X_0) \\ \rightarrow \\ (X_{t-2}, X_{t-3}, \ldots, X_0, X_{t-1} + F(X_{t-2}, \ldots, X_0)\end{gathered} \tag{3.7}$$

with the key-dependent round function in the $i$-th round defined as:

$$F(X_{t-2}, \ldots, X_0) = \left(\sum_{j=0}^{t-2} X_j + k_i + c_i\right)^3 \tag{3.8}$$

Figure 3.3 represents one round of GMiMC-CRF.

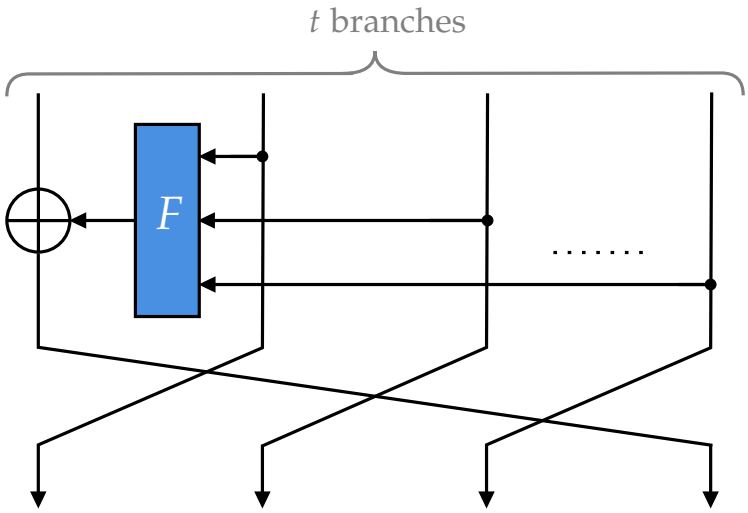


Figure 3.3: One round of GMiMC with a contracting round function (CRF) in an unbalanced Feistel network (UFN) (GMiMC-CRF), using $t$ branches.

Another variant is using **an expanding round function (ERF)**, which is also embedded in a UFN. GMiMC-ERF can be written as:

$$\begin{gathered}(X_{t-1}, X_{t-2}, \ldots, X_1, X_0) \\ \rightarrow \\ (X_{t-2} + F_i(X_{t-1}), X_{t-3} + F_i(X_{t-1}), \ldots, X_0 + F_i(X_{t-1}), X_{t-1})\end{gathered} \tag{3.9}$$

with the key-dependend round function in the $i$-th round defined as:

$$F(X_{t-1}) = (X_{t-1} + k_i + c_i)^3 \tag{3.10}$$

Figure 3.4 represents one round of GMiMC-ERF.

GMiMC with **a Nyberg's generalized-feistel-network round function (NGF)** embedded in a balanced Feistel network (BFN) is the third variant. GMiMC-NGF can be written as:

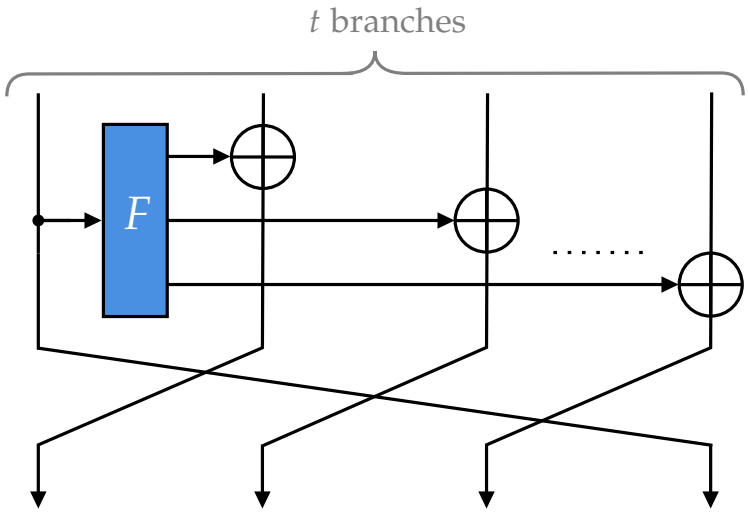


Figure 3.4: One round of GMiMC with an expanding round function (ERF) in an unbalanced Feistel network (UFN) (GMiMC-ERF), using $t$ branches.

$$\begin{gathered}(X_{t-1}, X_{t-2}, \ldots, X_1, X_0) \\ \rightarrow \\ (X_{t-2} + F_0(X_{t-1}), X_{t-3}, ..., X_0 + F_{(t/2)-1}(X_1), X_{t-1})\end{gathered} \tag{3.11}$$

with the key-dependend round function in the $i$-th round defined as:

$$F_j(X) = (X + k_{j+i\cdot\frac{t}{2}} + c_{j+i\cdot\frac{t}{2}})^3 \tag{3.12}$$

Figure 3.5 represents one round of GMiMC-NGF.

The fourth variant of a Feistel structure for GMiMC is using **a multi-rotate round function (MRF)**, embedded in a BFN. GMiMC-MRF can be written as:

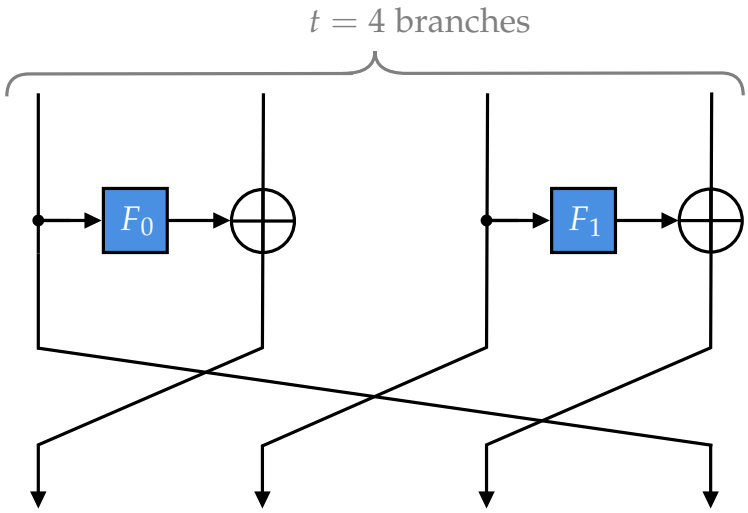


Figure 3.5: One round of GMiMC with a Nyberg's generalized-feistel-network round function (NGF) in a balanced Feistel network (BFN) (GMiMC-NGF), using $t = 4$ branches.

$$\begin{gathered}(X_{t-1}, X_{t-2}, ..., X_1, X_0) \\ \rightarrow \\ (X_{\frac{t}{2}-1} + F_{0-s \bmod (\frac{t}{2})}(X_{t-1-fx(s,0)}), X_{\frac{t}{2}-2} + F_{1-s \bmod (\frac{t}{2})}(X_{t-1-fx(s,1)}), \\ ..., X_0 + F_{\frac{t}{2}-1-s \bmod (\frac{t}{2})}(X_{t-1-fx(s,\frac{t}{2}-1)}), X_{t-1}, ..., X_{\frac{t}{2}+1}, X_{\frac{t}{2}})\end{gathered} \tag{3.13}$$

with the key-dependend round function in the $i$-th round defined as:

$$F_j(X) = (X + k_{j+i\cdot\frac{t}{2}} + c_{j+i\cdot\frac{t}{2}})^3 \tag{3.14}$$

Figure 3.6 represents one round of GMiMC-MRF with a branch size $t$ of 4.

In terms of the **different key schedules** and **different exponents** for the round function, it is the same as for GMiMC's predecessor MiMC. When using a univariate key schedule (always the same key for each branch and round), the PRF is less secure, than using a multivariate key schedule (different keys for the branches and rounds). However, for MPC this should not matter, as the different keys can be precomputed. When we use a different exponent

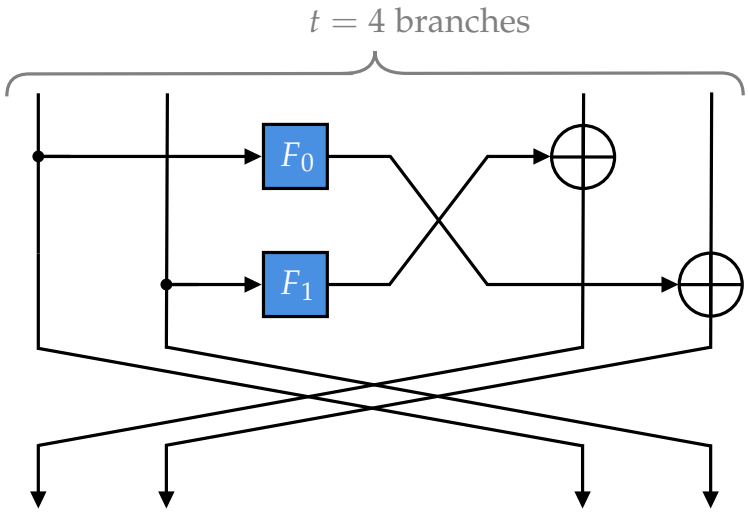


Figure 3.6: One round of GMiMC with a multi-rotate round function (MRF) in a balanced Feistel network (BFN) (GMiMC-MRF), using $t = 4$ branches.

than 3, and following the restriction that the exponent needs to be odd and in the form of $2^a - 1$, for some integer $a$, we do not get any advantage or disadvantage. Thus, we follow the approach of the authors of GMiMC and use the exponent 3.

**Security parameters.** Same as for MiMC, we first choose a prime size, and then an according number of rounds $r$. Based on the security analysis of [Alb+19b], $r$ should be the maximum number of rounds of a given set of considered attacks. These attacks are in the area of statistical and algebraic attacks. Moreover, [Bey+20] further analyzed the security of GMiMC(-ERF) and also HMiMC, and revealed new attacks against those two primitives. For GMiMC(-ERF) the authors found a zero-sum distinguisher for the full permutation. Thus, for practical security, the original minimum number of rounds for GMiMC should be updated. For the details of the specific attacks and their formula for calculating the minimun number of rounds, as well as the whole security analysis, we refer to the corresponding papers.

**Main applications.** As with its predecessor MiMC, GMiMC is a competitive PRF for SNARKs and MPC. In the area of STARK, GMiMC is one of the three candidate families for the *STARK-friendly hash challenge* [Sta20b; BGL20]. Furthermore, GMiMC is also a competitive hash function in the area of post-quantum-secure signature schemes. In this area, GMiMC offers the benefit of small signature sizes.

[Alb+19b] concluded that GMiMC-ERF is the most competitive variant in the area of MPC. Thus, for our benchmarks in this thesis we will only consider GMiMC-ERF.

**GMiMC-ERF for MPC.** The function evaluation, $x \rightarrow x^3$, is the same as for MiMC. Further, $t-1$ branches get "XORed" with $x^3$, and this is a simple addition in $\mathbb{F}_p$. Therefore, the complexity from the MPC point of view, *openings and rounds of communication per round* $r$, are for GMiMC-ERF the same as for MiMC. Hence, the main advantage of GMiMC-ERF over MiMC is the faster processing of more than one branch $t$.

### 3.1.4 HADES MiMC (HMiMC)

**Background.** After the success of MiMC and GMiMC in the area of MPC, the following question came up: *Are there also other (relatively old) design ideas which could be benefical in the area of MPC?* This question led to the successor of MiMC and GMiMC, HADES MiMC (HMiMC) [Gra+20] (Eurocrypt 2020).

**Approach.** HMiMC is combining two worlds: substitution-permutation networks (SPNs) and partial substitution-permutation networks (P-SPNs). In the beginning and end of one encryption process, there are $r_f$ rounds of SPNs (i.e., full S-box layers). In between these SPN rounds, there are $r_p$ rounds of P-SPNs (i.e., a mixture of S-box layers and identity mappings). One round consists of three steps:

1. add the round key,
2. apply the substitution, with ...
    a) ... full S-box layers for SPN rounds ($r_f$), and

   b) ... partial S-box layers combined with identity mappings for P-SPN rounds ($r_p$), and finally
3. multiply the whole state with a matrix $M$.

The $i$-th round key is produced by adding the key $k$ with the $i$-th round constant. In the MPC case, each branch uses the same round key.

Applying the substitution (S-box layers) means that the state gets cubed. Hence, this round layer is the same concept as with MiMC and GMiMC. And as with HMiMC's predecessors, to achieve a full permutation, we have to choose an appropriate prime $p$, such that $p \neq 1 \bmod 3$. For branches with no S-box layer in a P-SPN round, the identity mapping is applied. Applying the identity mapping means that such a branch is forwarded unchanged to the next round layer.

$M$ is a maximum distance separable (MDS) matrix, which guarantees that each branch depends on each other branch in each round. The whole state, so all $t$ branches, gets multiplied with the $t \times t$ MDS matrix $M$. The result of the multiplication is the input state for the next round. In the last round, the multiplication with $M$ is omitted.

After the last round, the key, with a random round constant, gets added one more time to produce the ciphertext. Figure 3.7 represents one round of HMiMC's SPN. Figure 3.8 represents one round of HMiMC's P-SPN. Figure 3.9 represents the last round of HMiMC.

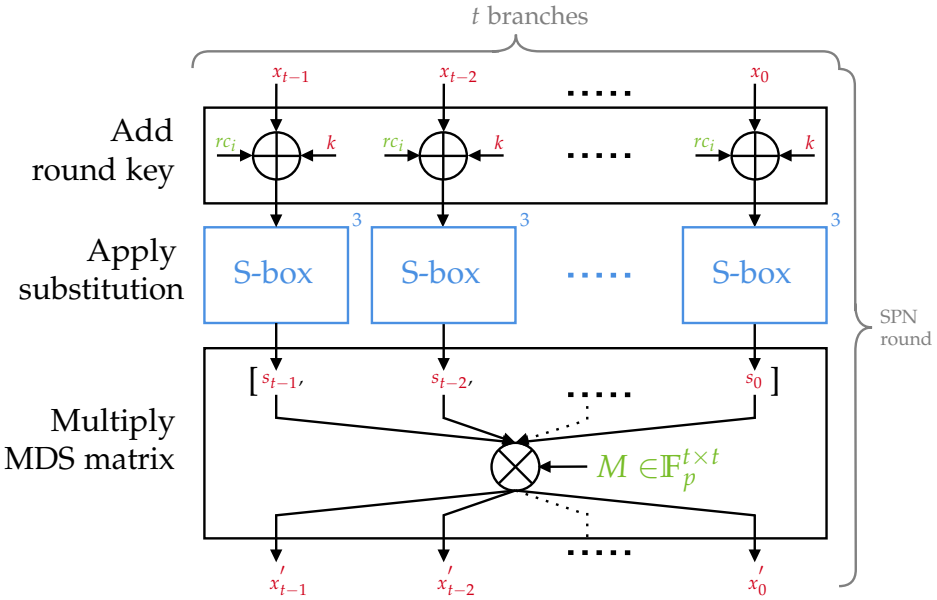


Figure 3.7: One SPN round of HMiMC (with full S-box layers). When adding the $i$-th round key, also an $i$-th round constant is added.

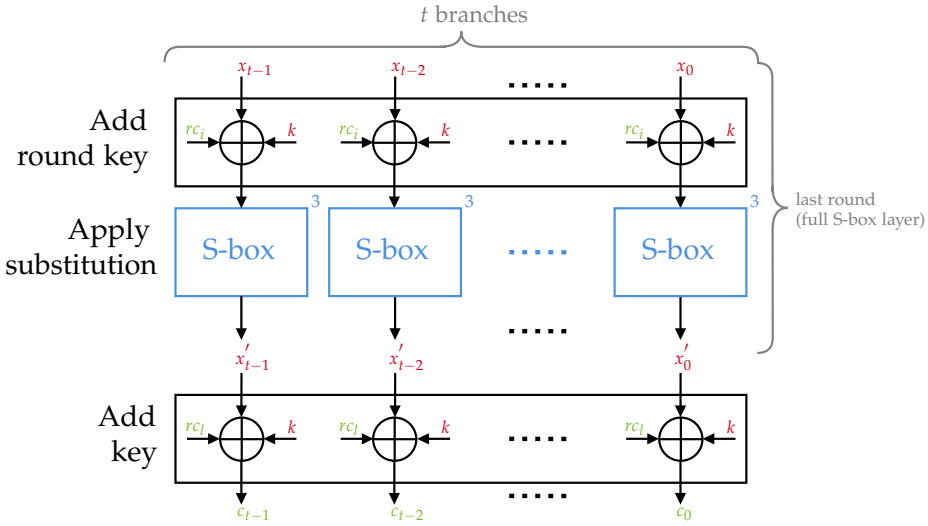


Figure 3.9: The last round of HMiMC. Please note that the last round of HMiMC will always be an SPN round. In the last round, the state does not get multiplied with the MDS matrix $M$. Instead, the output of the substitution layer is the input for the last operation: adding the round key one more time. As with the other rounds, when adding the $i$-th round key, also an $i$-th round constant is added. Also in the last operation a random constant gets added ($rc_l$).

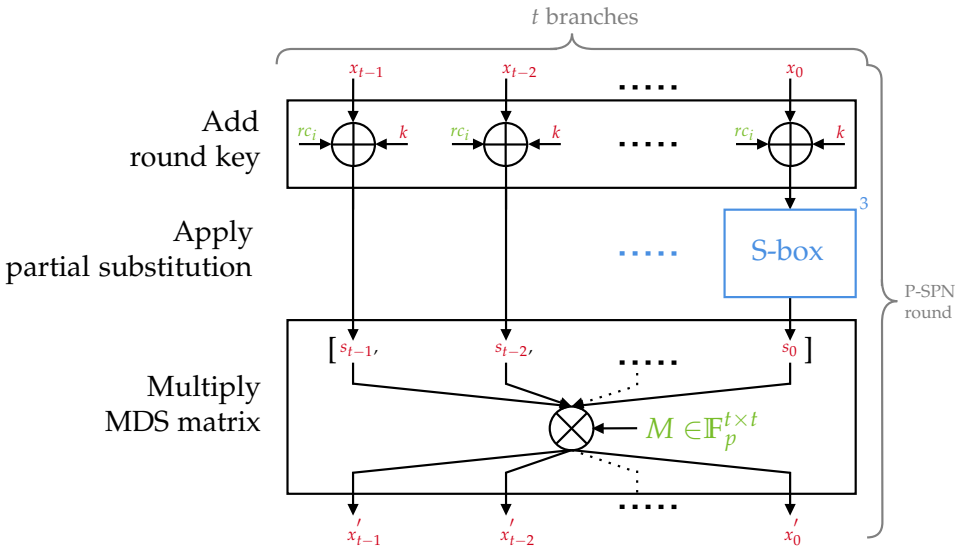


Figure 3.8: One P-SPN round of HMiMC (with partial S-box layers). In the rounds with P-SPN, the amount of S-boxes is variable; thus, it could even be that such a round has only one S-box. The other branches apply the identity mapping; meaning that the state gets forwarded unchanged to the next round layer. As with the SPN round, when adding the $i$-th round key, also an $i$-th round constant is added.

**Variants of HMiMC.** HMiMC is very parameterizable, and thus there are many variants possible. One parameter to tune the PRF is the **number of rounds** $r_f$ (start and end layer) and $r_p$ (middle layer). Though, for $r_f$, there have to be at least $r_{f_{stat}}$ rounds of full S-box layers in the beginning and end. These $r_{f_{stat}}$ rounds are required to guarantee security against certain statistical attacks. Besides that, the other rounds of $r_f$, let us call them $r'_f$, as well as $r_p$ can be chosen arbitrarily in a defined range. This defined range is given by the number of rounds $r$. Let us denote $r'$ as: $r' = r - 2 \cdot r_{f_{stat}}$; the number of rounds which can be freely chosen as $r'_f$ or $r_p$. The condition on these rounds is: $r'_f + r_p = r'$, for $r'_f, r_p \geq 0$; this means that $r'_f$ or $r_p$ could even be 0. Thus, all in all we get for the total number of rounds $r$ the following:

$$r = 2 \cdot r_{f_{stat}} + r'_f + r_p \tag{3.15}$$

whereas $r_{f_{stat}}$ is fixed, and $r'_f$ as well as $r_p$ can be chosen arbitrarily ($\geq 0$ though). Figure 3.10 represents one encryption of HMiMC; thus also the

concept of the different number of rounds $r_{f_{stat}}$, $r'_f$, and $r_p$.

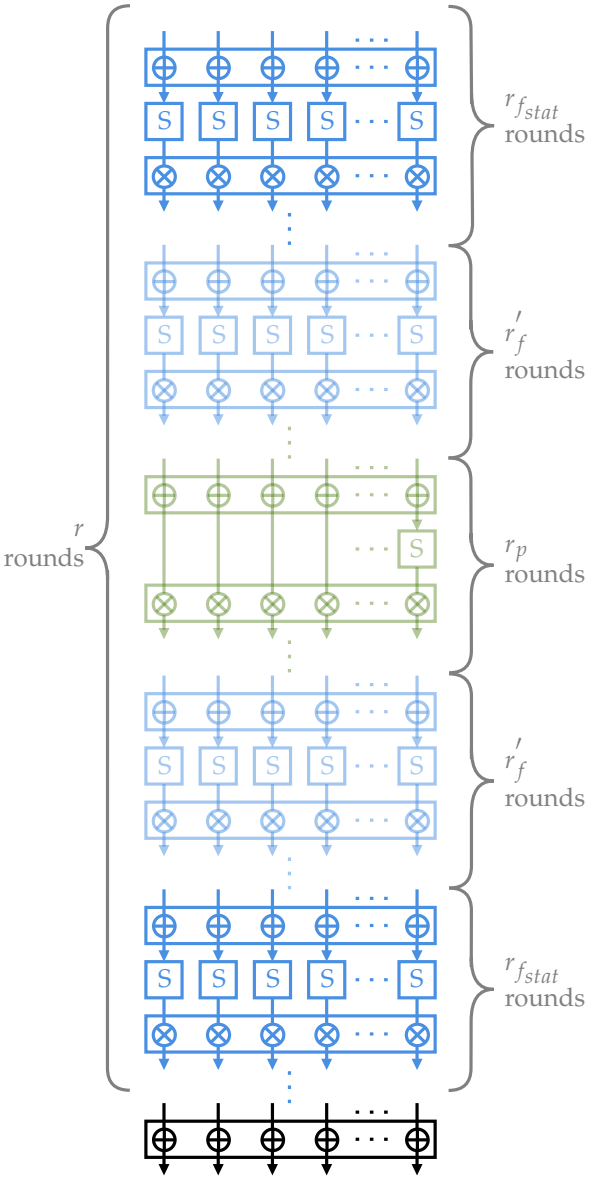


Figure 3.10: One encryption of HMiMC. The $r_{f_{stat}}$ rounds, which are necessary in the beginning and end of the encryption process, are shown in normal blue. The variable rounds, $r'_f$ and $r_p$, are shown in lighter blue and lighter green respectively. In the last round, the multiplication with the matrix $M$ is omitted; for reasons of simplicity and space we do not show this in this figure. After processing $r$ rounds, the state gets added one more time with the key $k$ and a random constant.

Another parameter is the **amount of S-box layers** $s$ in one round of a P-SPN: $1 \leq s < t$, whereas $t$ represents the branch size (amount of messages processed in one encryption). Thus, there has to be at least one S-box layer in a P-SPN round. Furthermore, we can also change the **degree of the S-box's function**. Though, as with HMiMC's predecessors MiMC and GMiMC, different degrees than 3 do not offer a significant advantage in terms of performance.

The **MDS matrix** $M$ is parameterizable too. The designers of HMiMC, [Gra+20], state that $M$ can be chosen arbitrarily. However, [KR20] concluded that according to certain values of $M$, the security of HMiMC either increases or decreases. In any case, different values of $M$ should not have a significant impact on the performance of the execution, as all operations are in $\mathbb{F}_p$.

The last parameter is the **key schedule**. HMiMC differentiates two (use) cases: (1) the normal, or "generic", case, and (2) the MPC case. For the MPC case, as explained above, each branch has the same key added with an $i$-th round constant. For the normal case, first, each branch has an own key $k_b$; second, the round keys get multiplied with a matrix $M_k$. Hence, every branch has its own key value and every branch depends on every other branch. $M_k$ can either be equivalent to $M$ or has different values.

**Security parameters.** As with MiMC and GMiMC, also with HMiMC we first define the prime size (in the MPC case). Then, we choose an according number of rounds $r$, so that HMiMC is secure against all identified attacks by the designers [Gra+20]. For the MPC case, less rounds are required than for the normal case. These fewer rounds are possible because an adversary is restricted to use a maximum of $\sqrt{p}$ data in the MPC case. However, as mentioned for GMiMC, [Bey+20] revealed new attacks against GMiMC and also HMiMC. For HMiMC, the authors propose a pre-image attack on certain linear layers. Though, the pre-image attack is independent of the number of rounds. Thus, for this attack the linear layer has to be chosen properly. For the details of the specific attacks and their formula for calculating the minimun number of rounds, as well as the whole security analysis, we refer to the corresponding papers.

**Main applications.** One of the main intended use cases of HMiMC is in the area of MPC. For MPC HMiMC is a competitive PRF, and outperforms its predecessors MiMC and GMiMC, as well as Leg, in the throughput metric (throughput is going to be explained later on in Subsection 3.2.1. Only when the branch size $t$ is $\geq 16$, GMiMC-ERF achieves a better throughput. The designers of HMiMC relate this "fallback" for $t \geq 16$ to the computation of the, then bigger, $t \times t$ MDS matrix $M$.

Furthermore, as HMiMC aims to minimize the multiplicative depth (non-linear operations), the PRF could also be competitive in the area of SNARKs. In the area of STARK, HMiMC is one of the three candidate families for the *STARK-friendly hash challenge* [Sta20b; BGL20].

**HMiMC for MPC.** The "XORing" of the key and the constants to the branches can be performed linearly and locally. For the S-box layer, as for its predecessor MiMC, we have the function evaluation of $x \rightarrow x^3$. Thus, we can either realize the S-box layer in the basic or in the "cube" way. The multiplication of the state with the MDS matrix $M$ represents a summation of multiplications of a secret-shared value (state elements) with a public scalar (matrix element). For the summation it is the same as for the "XORing" of, e.g., the key. The multiplications do not require communication either, but require some more computation effort.

Hence, as with MiMC, the main bottleneck in the MPC case for HMiMC is the function evaluation in the S-box layer. In addition, HMiMC also has the multiplication with $M$. However, as with GMiMC, HMiMC can process more than one branch $t$ per encryption. This processing of more than one branch per encryption, allows GMiMC-ERF and HMiMC to outperform MiMC; such as in the throughput metric.

### 3.1.5 Legendre (Leg)

**Background.** The underlying concept of Leg is already a relatively old idea. The basic concept had been already introduced in the mathematical community at around the 1930s. For practical cryptography, Damgård was the first who introduced Leg with a key as a pseudo-random generator (PRG)

in 1988 [Dam88]. Since then many years passed. And just recently, due to the practical relevance of MPC, Leg got introduced as MPC-friendly PRF by [Gra+16] (CCS 2016).

**Approach.** Leg makes use of Fermat's little theoreom. Given a prime $p$ and a positive integer $x$, Leg outputs either 1 or $-1$ [Kho19]:

$$Leg(x) = x^{\frac{p-1}{2}} \bmod p \tag{3.16}$$

If $x$ is a square of some integer, say $x = y^2$, then Fermat's little theorem applies [HS08]:

$$\begin{gathered} y^{\frac{2\times(p-1)}{2}} \bmod p = \\ y^{p-1} \bmod p = \\ 1 \end{gathered} \tag{3.17}$$

If $x$ is not a square of some integer, but $> 0$, Leg outputs $-1$:

$$\begin{aligned} x^{\frac{p-1}{2}} &= -1 && (\mathrm{mod}\, p) \\ x^{\frac{p-1}{2}} &= p-1 && (\mathrm{mod}\, p) \\ x^{p-1} &= (p-1)^2 && (\mathrm{mod}\, p) \\ 1 &= p^2 - 2\times p + 1 && (\mathrm{mod}\, p) \\ 1 &= 1 && (\mathrm{mod}\, p) \\ & && \square \end{aligned} \tag{3.18}$$

The only difference occurs when the input $x$ is 0; then also the output of Leg is 0.

Thus, the essential security assumption of Leg's PRG is based on the fact, that the "quadratic-value" representation of $x$ ($= y^2$) is uniformly distributed. [Dam88] conducted experiments for the outcome of Leg and concluded that this "quadratic-value" distribution, mod the prime $p$, occurs uniformly in terms of the statistics of the experiment.

[Dam88] introduced a keyed version of this basic Leg approach; denoted as $Leg^k(x)$. This keyed version makes use of a key $k$, when given the input $x$:

$$Leg^k(x) = (x+k)^{\frac{p-1}{2}} \bmod p \tag{3.19}$$

Hence, now the sum of $x+k$ needs to be the square of some integer $y \pmod p$ to get 1 as output. [Gra+16] called this the *"Shifted-Legendre-Symbol Problem"*. Leg is a special case where the exponent ($e$) is of the form $\frac{(p-1)}{2}$. More generally, [Vero8] called these kinds of mathematical problems, where we add the key to the input and apply the exponent $e$, the *"Hidden-Root Problem"*.

Though, given the input $x \in \mathbb{F}_p$, so far we basically only get 1 bit as output (1 or $-1$; given that the input $\neq 0$). Thus, we have the PRG as described by [Dam88]. In order to encrypt $x$, to get a proper PRF, we need to apply the basic approach of Leg more often.

When using Leg as PRF for MPC, [Gra+16] first, shifted the output of Leg to be either in 1, $\frac{(p+1)}{2}$, or 0:

$$\frac{Leg^k(x)+1}{2} \bmod p \tag{3.20}$$

So if the output of Leg is 1, it is mapped again to 1 $\left(\frac{1+1}{2}\right)$. If the ouput is $-1$, it is mapped to 0 $\left(\frac{-1+1}{2}\right)$. If the output is 0, we get $\frac{1}{2}$; which we can map to $\frac{(p+1)}{2}$ due to the nature of being in the modulo group $p$.

Now, to get a PRF, we have not only one $\mathbb{F}_p$ as key, but a matrix of $\mathbb{F}_p$ elements. We consider the case where we have only one $\mathbb{F}_p$ element as input; thus a branch size of one. In case we need to apply the PRF for a branch size $> 1$, we can, e.g, encrypt each branch separately, or apply the CBC mode of operation [KL14], as also stated by [Gra+16].

**Variants of Leg.** Instead of using a single key, we can also use $d$ keys: $Leg^{k^0,\ldots,k^{d-1}}(x)$. With these $d$ keys, the output of Leg is computed as:

$$(k_0 + k_1 \times x + k_2 \times x^2 + \ldots + k_{d-1} \times x^{d-1} + x^d)^{\frac{p-1}{2}} \bmod p \tag{3.21}$$

[Beu+20] also showed that, based on Leg, one can also generalize it to the Jacobi-Symbol PRF or the $r$-th Power-Residue-Symbol PRF. Though, they show in their work, that neither the Jacobi Symbol, nor the $r$-th Power Residue Symbol offer a practical benefit for MPC over Leg.

**Security parameters.** Since Leg needs only one "round" per encrypted bit, the essentially only parameter to tune the PRF is the key. For instance, to have $d > 1$ keys per encryption, as shown in the previous paragraph. Another approach would be to use for each encrypted bit an own key; or even combine the two approaches (several $d > 1$ keys per encrypted bit). These approaches, of larger and more keys respectively, make cryptanalysis more difficult.

**Main applications.** As shown by [Gra+16], Leg was not considered as a PRF until recently (2016). In the area of MPC it is a promising cipher, especially when the network becomes a bottleneck (as we will show later in Section 4.3). And just more recently (2020), [BS20] introduced Leg as a basis for a promising post-quantum signature scheme; called *LegRoast*. Furthermore, currently Ethereum considers Leg as a potential PRF for their next blockchain version, the Ethereum 2.0 blockchain [Eth20; dan20; Beu+20].

**Leg for MPC.** In order to use Leg for MPC, we can either implement it in a way that the output is public or secret-shared. As with the previous PRFs, we will use the secret-shared-output variant, using the protocol as described by [Gra+16]; shown by Protocol 3.1.

With this approach we need to produce one random square, one random bit, and two random Beaver triples in the preprocessing phase. Please note that the public quadratic non-residue can be chosen in the very beginning, or even beforehand, and can be re-used. Then, during the online phase, we need to multiply two times two secret-shared values and have one opening. Thus, overall for the computation of an $n$-bit output, for the preprocessing we need $n$ random squares, $n$ random bits, and $2 \times n$ Beaver triples. For the actual computation we need $2 \times n$ multiplications and $n$ openings in $3 \times n$ rounds of communication.

| | | | |
|---|---|---|---|
| 1 : | $pqnr \pmod p$ | $\leftarrow$ | public quadratic non-residue |
| | | | // is chosen and can be reused |
| 2 : | $[s^2]$ | $\xleftarrow{\$}$ | random square |
| 3 : | $[b]$ | $\xleftarrow{\$}$ | random bit |
| 4 : | $[mask]$ | $\leftarrow$ | $[s^2] \times ([b] + pqnr \times (1 - [b]))$ |
| | | | // if $b = 0 \rightarrow [s^2] \times pqnr$ |
| | | | // if $b = 1 \rightarrow [s^2]$ |
| 5 : | $hidden$ | $\leftarrow$ | **open**$([mask] \times ([input] + [key])$ |
| 6 : | **return** $[output]$ | $\leftarrow$ | $\frac{(Leg_p(hidden) \times (2[b] - 1)) + 1}{2}$ |

Protocol 3.1: Given the secret-shared input and key, this protocol shows the computation of Leg to obtain a secret-shared output.

In contrast to the approaches of MiMC, GMiMC, and HMiMC, which compute $r$ times a specific round function, Leg's computation is one single computation per bit; like only one round per bit. And as computation-wise the bits do not depend on each other, they can be computed in parallel. Thus, theoretically the individual rounds of communication per bit can be batched altogehter, which would lead overall to just three rounds of communication per PRF encryption. This parallelization of computation and communication, is an advantage of Leg over the other PRFs.

## 3.2 Requirements on MPC Programs

When running an MPC program, we are usually aiming to run it as efficiently and secure as possible. To run it most efficiently and secure, we need to find a suitable (1) protocol and (2) underlying cipher, at least for the PRF evaluations; but based on which metrics? The relevant performance metrics are runtime, network data, and memory consumption. As the fastest MPC program does not help us much in terms of privacy, when it is insecure, the security model

is a complementary requirement. This section explains the different metrics for MPCs programs in the order mentioned before.

### 3.2.1 Runtime

For runtime, the leading questions are:

- *How long does it take to run the MPC program?*
  → Overall Runtime ($s$ or $ms$)
- *How long does it take to compute one PRF encryption?*
  → Latency ($ms/op$)
- *How many PRFs can be computed in one second?*
  → Throughput ($ops/s$)

In terms of execution of programs, runtime is the "classic" one, as it is interesting for basically all programs and algorithms. Usually, when measuring the timing of a program's execution, it is only interesting how long the overall runtime is. In some cases it might also be interesting to get the runtime for certain sub-parts of the program. In terms of PRFs for MPC programs, the runtime metric itself is split into three sub-categories: overall runtime, latency, and throughput.

#### Overall Runtime

To get the overall runtime of PRF encryptions, we simply start a timer right before the execution and stop the timer when the program is finished. However, the overall runtime of all protcols in SCALE can again be split. Therefore, in addition to the overall runtime, we can also measure the runtime of SPDZ's preprocessing and online phase.

The preprocessing phase is independent of the program, but does the heavier computation (triple generation). The online phase depends on the program and on the preprocessing-phase's triples, but the computation is not so heavy. As the two phases differ in their dependencies and amount of computation, taking separate timings for the preprocessing and online phase allows to

optimize separately. Using this approach, we can take a deeper look into the execution process and optimize in a more fine-grained way.

Thus, for the overall runtime it is interesting to measure the timings for the execution of the

- whole program,
- preprocessing phase, and
- online phase.

The runtime's unit for these measurements is usually given in seconds: $s$. If a program is very short or the execution is very fast, way below a second, the unit can also be given in milliseconds: $ms$.

#### Latency

In terms of PRF encryptions in an MPC program, besides the overall runtime of the whole program or a specific phase, it is also interesting how long it takes to perform one PRF encryption; this is referred to as the latency. The unit of latency is time (usually milliseconds) per operation (PRF encryption in our case): $ms/op$.

#### Throughput

Another interesting metric for PRF encryptions in MPC, is to see how many PRF encryptions can be performed in one second; this is referred to as throughput. The unit of throughput is operations (PRF encryptions in our case) per a given time frame (usually one second): $ops/s$.

### 3.2.2 Communication

For communication, the leading questions are

- *How often do the players communicate with each other during the run of an MPC program?*
  → Rounds of communication (#)

- *How much data is sent and received of each player during the run of an MPC program?*
  → Network Data (*MB* or *GB*)

In order to compute the desired function in an MPC program, the players need to exchange data. (1) How often such a data exchange occurs (rounds of communication), and (2) how much data is transferred (network data), depends on the used access structure and PRF. This dependency makes also the communication between players an interesting metric for MPC programs.

### Rounds of Communication

For the data exchange during an MPC-program's run, the current state of the art consists of two approaches: Either the data is exchanged in a synchronous manner, or in an asynchronous manner [BZL20]. In an asynchronous manner there is no real limit to the time of the data exchange, but the security guarantees are usually a bit lower. In a synchronous manner the time of the data exchange is limitied, thus it must not exceed a certain time limit (like 500*ms*), but the security guarantees are usually higher than for the asynchronous approach. In this thesis we consider the synchronous approach.

This dependency on the availability for the synchronous approach means that certain steps in the MPC protocol can only be finished, when the players received enough data from the other players *fast enough*; especially for protocols where data from each other player has to be received. Hence, in real-life executions in a synchronous manner, fewer rounds of communication means a higher probability of execution success; as the failure, due to the fact that a player is not available for some time, is less likely. This probability of failure increases the larger the amount of participating parties is, and the more data is exchanged, specifically in slow or unstable network environments.

The unit for this occurring of data exchange is the amount of rounds of communication: #.

#### Network Data

As during the execution of an MPC program data is exchanged between the players, we want to see how much data is exchanged for our chosen PRFs in combination with the different access structures (ASXs). In a LAN-like network environment, the data exchange is probably not an issue. But in a WAN-like network environment the data exchange could be a bottleneck, especially if a lot of data is exchanged.

Furthermore, in terms of network data we differ between data sent and data received. Because for certain protocols one player might send more than other players. In total, when summing up the sent & received data, each player should get the same amount. However, it is still interesting to see, e.g., how big the difference of data sent between the players is. The approach of seeing this difference allows to optimize in the direction of uniform data distribution; so that not one player unnecessarily blocks the execution because the player needs to send much more data than the other players.

The unit for data received and sent typically depends on the amount of data, but is usually given in: $MB$ or $GB$.

### 3.2.3 Memory Consumption

For memory consumption, the leading questions are:

- *How much memory do the different runtime threads consume during the run of an MPC program?*
  $\rightarrow$ Thread Memory ($MB$ or $GB$)
- *How much memory does each player consume during the run of an MPC program?*
  $\rightarrow$ Player Memory ($MB$ or $GB$)
- *How much memory does the whole run of an MPC program consume?*
  $\rightarrow$ Execution Memory ($MB$ or $GB$)

Next to the runtime of an MPC program and the communication between players, there is also a third metric, the memory consumption. Given "normal" conditions, a relatively modern computer or server and just a few players

($\leq 5$), memory consumption should not be an issue. Though, for the following scenarios memory consumption could become a bottleneck or even an execution issue itself:

1. A memory-restricted environment; such as running MPC programs on internet-of-things (IoT) devices.
   Would resource-restricted IoT devices, such as a Rasperry Pi 4 [Ras20], be able to run MPC programs?
2. An MPC protocol with many players, such as $\geq 10$ players.
   How does the execution scale in terms of memory?
3. For local testing, as probably not everyone has access to a server or a dedicated cluster. In addition, sometimes one might just want to quickly test an MPC program on the local machine.
   Which MPC programs can be run on a relatively modern personal computer in terms of memory consumption?

**Memory-Consumption Aspects.** With memory consumption, we mean the random access memory of a computer, the RAM. As we are interested in Thread Memory, we will measure the RAM usage of each runtime thread individually. This way, we see which runtime threads need the most RAM. Then, when summing up the Thread Memory of each runtime thread, we compute the Player Memory. Player Memory is especially interesting for real-life deployments in restricted-memory environments. Finally, when summing up the Player Memory of each player, we compute the Execution Memory. Execution Memory is specifically relevant for testing of MPC programs on a local machine, usually a single computer. The different levels of memory-consumption aspects allow for a more fine-grained analysis and optimization of MPC programs, protocols, and engines. Depending on the amount of RAM usage, the unit of the different memory aspects is usually: *MB* or *GB*.

### 3.2.4 Security Model

As we differ between the way (1) how a malicious player acts and (2) how players get corrupted [Lin20], for the security model, the essential questions are:

- Malicious-player behaviour:
  - *How secure is the MPC-program's execution against a pasively malicious player?*
    → Passive-player security
  - *How secure is the MPC-program's execution against an actively malicious player?*
    → Active-player security
- Malicious-player corruption:
  - *How secure is the MPC-program's execution against static player corruptions?*
    → Static player corruption
  - *How secure is the MPC-program's execution against dynamic player corruptions?*
    → Dynamic player corruption
- With respect to adversaries from outside, like an eavesdropper in the network; i.e., not malicious players:
  - *How secure is the MPC-program's execution against an adversary from outside (non-player adversaries)?*
    → Outside Security

We note that, e.g, [Lin20; Lin16; Canoo] further split the active-player security and dynamic player corruption. Though, the essential difference is between passively and actively malicious players, as well as static and dynamic player corruptions respectively. Moreover, [Canoo] introduced a general approach to model the security of arbritary cryptographic protocols. And [Lin16] describes a simulation technique to proof the security of, e.g., MPC in different adversary scenarios.

Furthermore, we also note that, to the best of our knowledge, the term *Outside Security* is not common in the literature. Hence, the term is introduced in this thesis. *Outside Security* mainly relates to Network Security; and in the network aspect, this can be easily omitted using, e.g., tansport layer security (TLS) and certificate pinning. Another aspect is an intruder, or hacker, in one of the player's system. The intruder could, for example, try to read the processed data in RAM.

**MPC-Protocol Selection.** As the runtime, communication, and memory consumption are metrics for the performance of an MPC program, so is the selected security model (scenario) the metric for the program's security level. For an MPC program, the three security models to consider are: (1) Outside Security, (2) Semi-Honest-Player Security, and (3) Malicious-Player Security. Usually, when we select an MPC protocol for our MPC program, we do not want to leak the input of the data, and sometimes also not the output. To guarantee the data privacy for all three scenarios, we need to select an appropriate MPC protocol.

**Priority of Requirements.** The metric of the desired security level is not measureable, like the performance metrics. However, usually it is more important to guarantee the privacy of the relevant data than having a slightly faster program execution. Though, when the execution of an MPC program is way too slow, thus not practical, the best security does not help, as the program is probably not chosen in real-life scenarios. Due to this practical relevance in real-life scenarios, measuring the performance is important too.

Tackling this security-performance tradeoff is one of the thesis' goals. We try to choose an appropriate security level, and find the right settings for practical performance. These settings are our so-called Benchmark Dimensions (Section 3.3), and the overall approach of tackling this security-performance tradeoff is the starting point of Our Benchmarking Plan (Section 3.4).

## 3.3 Benchmark Dimensions

**Variable settings as benchmark dimensions.** After the identification of the requirements on the security level, we look for settings that meet the performance requirements of the MPC-program's execution. These settings are tunable variables. We call these variable settings dimensions, as each variable setting spans a dimension for the benchmarking of an MPC-program's execution. The more dimensions we have, the more complex the analysis of finding the right settings for certain requirements on an MPC program is. The reduction of this benchmarking complexity, is one of the primary goals of the thesis.

**Two types of dimensions.** When we look depper into the benchmark dimensions, we can distinguish two types of dimensions. Firstly, the chosen environment (env); and secondly, settings for the actual MPC program (prog). The env dimensions are basically selected by the chosen environment, such as the network bandwidth and latency, or maybe also the amount of players. The prog dimensions can tweak the actual program, such as parallel computation of PRF evaluations.

Why do we also look at the env dimensions? One the one hand, simply because it is still interesting to see the performance of the MPC-program's execution in the envisioned environment. On the other hand, they are still variable settings for the execution of an MPC program, and the benchmarking of the envisioned environment might give valuable feedback and changes the environment. For instance, if an execution with six players would be envisoned, but after the benchmarking one sees that this setting consumes too much memory (RAM) for the players' devices; this way, the amount of players might be reduced to, e.g., three. Again, it boils down to the aforementioned security-performance tradeoff.

This section explains our different benchmark dimensions for the MPC-friendly PRFs in this thesis, in the following order:

- Env Dimensions
  - Network
  - Amount of players
  - Access structure (ASX)
- Prog Dimensions
  - Amount of evaluations
  - Amount of parallel encryptions (batch size)
  - Single Instruction, Multiple Data (SIMD)
  - Amount of messages (branch size)

### 3.3.1 Env Dimensions

Env dimensions are basically chosen by the target environment. Though, these dimensions are still variables for the execution of an MPC program. This

sub-section describes the env dimensions network, amount of players, and access structure (ASX).

**Network.** The network dimension determines the bandwidth and latency of the players' network data. We differ between the environments of a local-area network (LAN) and a wide-area network (WAN). For LAN, the typical bandwidth is $\sim 1 Gbit/s$ with a latency of $\sim 0.5 ms$. For WAN, e.g., [Gra+16] chose the MPC-benchmark setting to have a bandwidth of $\sim 50 Mbit/s$ with a latency of $\sim 100 ms$. Though, it is of course possible to distinguish between different levels of WAN environments; thus simulating, e.g., a latency of $\sim 200 ms$.

**Influence of network dimension.** As the network has an impact on the bandwidth and latency of the data exchange, this dimension influences the runtime of the MPC-program's execution. The other performance metrics, communication and memory consumption, should not be influenced. Given an MPC program, this influence introduces the following benchmark question:

- Does a change in the network have a linear impact on the runtime?

**Amount of players.** The amount of players determines the number of participating parties. The more players participate in an MPC-program's execution, the more input is needed from other players. Depending on the chosen ASX, a player needs input from all other players, or only from some of them. The dimension for the amount of players is probably defined by the envisoned environment's use case. However, for some use cases the amount of players might be free to choose in a range, such as between three and seven players.

**Influence of amount-of-players dimension.** The more players participate in an execution, usually the more communication needs to be done. Due to the fact of more communication, this dimension also influences the runtime and memory consumption. Thus, this influence leads to the following benchmark question:

- Does an increase of the amount of players
  linearly increase the three performance metrics?

**Access structure (ASX).** When we execute an MPC program, we need to choose an MPC protocol, the so-called ASX. Usually, each ASX can guarantee the envisioned security level, given the right settings. These settings are, for instance, the amount of honest players or the modulus-prime's bit size.

**Influence of access-structure (ASX) dimension.** Depending on the choice of the ASX, more or less communication is needed. Due to this impact, the ASX might also influence the runtime and memory consumption. Therefore, the ASX dimension introduces the following benchmark question:

- Given an envisioned security level,
  how does the ASX influence the three performance metrics?

### 3.3.2 Prog Dimensions

Prog dimensions define the various settings for the actual MPC program. We selected four relevant dimensions for PRF evaluations. This sub-section describes the chosen prog dimensions: amount of evaluations, amount of parallel encryptions (batch size), amount of messages (branch size), and single instruction, multiple data (SIMD).

We note that although especially the branch size and batch size dimensions mainly correspond to benchmarking PRFs, the others are also more general performance metrics which should guide the user to select a suitable protocol. And for programs which are able to batch, e.g., encryptions or have several branches to evaluate, thus very similar to PRF evaluations, even the branch size and batch size can be relevant benchmark dimensions. Hence, the described prog dimensions are not limited to PRF evaluations.

**Amount of evaluations.** For benchmarking our PRFs, we need to ensure a runtime which is high enough, so that we have reliable measurements. Usually a runtime above five minutes is a good lower bound. To achieve this lower bound on the runtime, we use the amount-of-encryptions dimension. This dimension determines how often the PRF is encrypting a test value.

**Influence of amount-of-evaluations dimension.** The more test values are encrypted, the higher the runtime will be. But it normally also influences the communication and memory consumption. This collateral influence leads to the following benchmark question:

- Does the amount of encryptions have a linear impact on communication and memory consumption?

**Batch size.** When performing PRF encryptions, we can choose how many encryptions should be performed in parallel. This setting of parallel encryptions is our batch-size dimension. The main goal of this dimension is to have a higher throughput of PRF encryptions; thus having a better runtime.

**Influence of batch-size dimension.** More encryptions in parallel should lead to a better runtime. However, in addition, a higher batch size might result in a higher memory consumption. The communication metric, in turn, should not be much influenced, as in total the same amount of encryptions are performed. These influences lead to the following benchmark questions:

- How well does the batch size improve the runtime?
- Does a runtime improvement via the batch size linearly increase the memory consumption?
- When changing the batch size, does the communication metric stay the same?

**Single Instruction, Multiple Data (SIMD).** With SIMD, we can apply the same operation on several bytes simultaneously. Thus, SIMD is very similar to the batch-size dimension. But SIMD itself is not restricted to the parallel

computation of PRF evaluations. In turn, this also depends on the actual MPC program, if SIMD can be applied at all.

**Influence of SIMD dimension.** As SIMD allows us to skip some byte-code instrunctions, it should decrease the runtime. Though, due to the simulatenous byte operations, it might increase the (peak) memory consumption. For the communication metric, SIMD should have no influence. Thus, this expected influence introduces the following benchmark questions:

- How well does SIMD improve the runtime?
- Does a runtime improvement via SIMD linearly increase the memory consumption?
- When using SIMD, does the communication metric stay the same?

**Branch size.** For each PRF evaluation we can specify how many messages should be encrypted; this is the branch-size dimension. As we are mainly interested in the performance of PRF evaluations with one message, usually this dimension just has the value one. However, GMiMC, for instance, needs a branch size of two as it uses a Feistel network. Moreover, certain PRFs might outperform other PRFs when the branch size has a certain number; like we have seen for GMiMC-ERF, which outperforms HMiMC when using a branch size of 16.

**Influence of branch-size dimension.** A branch size of $> 1$ means that more than one encryption is computed per PRF evaluation. Thus, for the same amount of evaluations, more encryptions are performed. Due to more encryptions, different branch sizes should affect all three MPC-benchmark metrics, which leads to the following benchmark question:

- Does an increase of the branch size linearly increase the runtime, communication, and memory consumption?

## 3.4 Our Benchmarking Plan

**Starting point.** In the previous sections we have chosen specific PRFs to benchmark, ascertained requirements on an MPC-program's execution, identified relevant benchmark dimensions, and derived interesting benchmark questions. These requirements, dimensions and benchmark questions serve as the starting point for our benchmarking plan. We want to answer some of the questions, in order to give recommendations on the choice of a good PRF-dimensions combination with a specific requirement in mind. This section describes a plan to find the suitable PRF-dimensions combination for the envisioned use case.

**Overall goal.** The overall goal of the thesis' benchmarking approach is to (1) identify relevant benchmark requirements, and (2) based on given requirements, find the best PRF-dimensions combination which also fulfill the required security level. To achieve this overall goal, we split it into two sub-goals. First, we do a sanity check of the benchmarking framework. The sanity check is done by performing the same experiments as in our baseline paper [Gra+16]. Second, we aim to give recommendations with a set of requirements in mind. The plan to achieve these sub-goals is described in the mentioned order in the following sub-sections.

### 3.4.1 General Benchmarking Approach

The general benchmarking approach, to give recommendations with a set of requirements in mind, looks as follows. Firstly, we gather relevant benchmark data; and secondly, we analyze this data. For the gathering of the benchmark data, we run once a set of defined combinations of dimensions; in order to get benchmark data of the required variable settings. Then we are going to extract the relevant results of these tests, and perform an analysis based on our requirements.

**Data gathering.** In terms of gathering relevant benchmarking data, the paper by [Gra+16] gives us the setting for the sanity check. For the discovery of recommendations for real-life scenarios, we can run once a set of combinations of dimensions. Then, based on the specific requirement, we zoom in and analyze.

**Amount of players.** For the sanity check we use the same approach as in the "baseline" paper by [Gra+16]; hence performing benchmarks with two players. For the recommendations with a set of requirements in mind we extend to three players. The extension to three players has two reasons: first, to check if one more player has a linar or non-linear impact on the performance; second, the already existing ASXs within SCALE support an honest-majority setting only starting with three players.

**Access structures (ASXs).** For the benchmarks, we use the ASXs which already exist in SCALE. And all of these ASXs are in the malicious-adversary setting. For two players, our sanity check, we use the corresponding ASX: dishonest majority based on fully homomorphic encryption (FHE) with 128-bit security (***#18***).

For three players, there exist five ASXs for honest majority and two ASXs for dishonest majority. The honest-majority ASXs are *(3,1)*; thus from three players maximum one player can be malicious to guarantee security. And, the honest-majority ASXs are based on linear secret sharing. The two dishonest-majority ASXs are based on FHE and support a 32-bit and an 128-bit prime respectively. For our benchmarks, we use 128-bit (primes) security. Hence, all in all we have six three-players ASXs to select for our benchmarks:

- Honest majority based on linear secret sharing
  - *Shamir (3,1)* (**#1**)
  - *Replicated (3,1) Maurer* (**#2**)
  - *Replicated (3,1) Reduced* (**#3**)
  - *Q2 Shamir (3,1)* (**#4**)
  - *Q2 Replicated (3,1)* (**#5**)
- Dishonest majority based on FHE

– *FHE Based 3 party 128-bit prime* (***#19***)

The italic names in the bullet-point list are the same as in SCALE's repository [KU 20]. The information about the pre-defined ASXs can be found in the file `README.txt` within the folder `Auto-Test-Data`. Furthermore, each of the seven mentioned ASXs (one 2-player and six 3-player), got a hashtag with a number; like ***#18***. These numbered hashtags are used within the description of the benchmarks and refer to the corresponding ASXs.

**Data analysis.** The result of the data analysis will be plotted graphs, to illustrate the benchmarks. After the creation of the graphs, these results will be presented, evaluated, and discussed in Chapter 4.

### 3.4.2 Sanity Check

The paper by [Gra+16], which initiated the idea for the thesis, serves as sanity check for our benchmarking framework. Although the experiments are roughly four years old, and the MPC engines probably improved since then, we still expect similar results. The aim of the paper was to compare the PRFs AES, LowMC, MiMC, Leg, and NR with a focus on the runtime metric, thus latency and throughput. As mentioned in Section 3.1, we benchmark only MiMC and Leg from the PRFs of the baseline paper.

**Dimensions.** In the paper by [Gra+16] an ASX for malicious security with two players and a branch size of one was used. The amount of PRF evaluations was set to at least 1,000; in order to get high enough runtimes right away ($\geq$ five minutes), we set the amount of evaluations to 1,000,000. Furthermore, the authors evaluated the PRFs using the preprocessing and online phase, within a local-area-network (LAN) and a simulated wide-area-network (WAN) environment. To get better runtimes, the PRFs were also tested with different batch sizes.

**Experiment setup.** Based on the given dimensions, we can state specific tests for our experiment. Table 3.1 shows the experiment setup for the sanity check. Now, we run the tests by executing all combinations of these dimensions.

| Dimension | Value(s) |
|---|---|
| PRFs | MiMC, Leg |
| #Evaluations | 1,000,000 |
| Batch size | 1,2,4,8,...,4096 |
| Branch size | 1 |
| #Players | 2 |
| Network | LAN, WAN |
| Preprocessing | Real |
| ASX | #18 |

Table 3.1: The set of dimensions of the experiment setup for our sanity check. The bluely highlighted dimensions are the variables for the PRFs MiMC and Leg in this experiment.

### 3.4.3 Recommendations with a Set of Requirements in Mind

**Experiment setup for data gathering.** Table 3.2 shows the set of test values for each dimension for our selected PRFs. The first row shows the different PRFs we are going to benchmark and evaluate. In the middle (three) rows we have the program (prog) dimensions. In the last (four) rows we have the environment (env) dimensions. The bluely-highlighted dimensions are the variables in this experiment: #Evaluations, Batch size, and Branch size (prog), as well as Network, Preprocessing, and ASX (env). Thus, the only fixed dimension is the amount of players: #Players (env). #Evaluations is dynamically chosen so that we have a runtime of $\geq$ five minutes, and therefore more reliable benchmarks. Also, the Branch size is only for the PRF MiMC a variable. Leg is evaluated with a Branch size of only one. Whereas GMiMC and HMiMC are evaluted with a Branch size of only two respectively. Please note that both GMiMC and HMiMC can only be evaluted with a Branch Size of $\geq$ two. This limitation on GMiMC and HMiMC is due to the fact, that they are leveraging a Feistel network.

Please note further, that in general more values are possible for the different

dimensions. Since the main focus of the thesis is to make benchmarking easier, we showcase the applicability with a limited set.

For each combination of PRF and ASX, which fulfill a set security level, we benchmark each possible combination of different values for our identified dimensions.

| **Dimension** | **Value(s)** |
|---|---|
| PRFs | MiMC, GMiMC, HMiMC, Leg |
| #Evaluations | * (large enough) |
| Batch size | 1,2,4,*,512 |
| Branch size | 1,2[+] |
| #Players | 3 |
| Network | LAN, WAN |
| Preprocessing | Real, Fake |
| ASX | #1,#2,#3,#4,#5,#19 |

Table 3.2: The set of dimensions of the experiment setup for our recommendations in real-life scenarios. The bluely highlighted dimensions are the variables for the PRFs MiMC, GMiMC, HMiMC, and Leg in this experiment. [+] The branch size is only a variable for MiMC. Leg is evaluated only with a branch size of 1. GMiMC and HMiMC are evaluated only with a branch size of 2.

**Data-analysis approach for recommendations in real-life scenarios.** For the sake of simplicity with regards to showcasing the benchmarking framework b4M, we will analyze the gathered data for two use cases:

*Sweet spot of the batch size to achieve best throughput and latency in a...*

1. ...*LAN environment*
2. ...*limited network (WAN)*

## 3.5 Benchmarking for MPC (b4M) in SCALE-MAMBA (SCALE)

This section describes the design and implementation of our benchmarking framework b4M. Next to the framework, also the necessary modifications

in SCALE are described. First, the design of b4M is shown; and second, the concrete implementation of the framework and modifications in SCALE are described.

### 3.5.1 Design

**Minor modifications in SCALE.** The vision of b4M is to make it as independent as possible from SCALE. We want to have the usual MPC framework, SCALE, and beside that b4M. Thus, b4M should need as little modifications as possible from SCALE. The most important modification is the additional benchmark output.

When the execution of the MPC program is finished and the user wants to benchmark the execution, the additional benchmark data is outputted. Then we have to parse the benchmark data somehow and extract the relevant information in b4M.

**Parsing of benchmark data.** To make parsing easier, we are going to combine HTML and JSON in the benchmarking-relevant output. Each part of a benchmarking-relevant output is wrapped inside an HTML element, like <b3m4>. Inside the HTML element, the actual content is given in a JSON-compatible form. Figure 3.11 illustrates an example of an output containing a benchmarking-relevant part.

**Updating benchmark data.** This approach has the advantage that if we want to update SCALE's benchmarking output, we only need to add or update the JSON data inside the HTML elements of the execution's output. The extraction and further processing of the (updated) benchmarking data is, then, handled by b4M.

**Pipeline of Benchmarking for MPC (b4M).** b4M should not only parse, extract, and process the benchmarking-relevant data of SCALE's execution output, it should also start the benchmark in the first place. The whole pipeline looks as follows:

```
...
Exiting aBit production thread
<b3m4>
{"player":1,
  "thread":2,
  "netdata":{
    "sent":{"bytes":159438386,"MB":159.44},
    "received":{"bytes":159438412,"MB":159.44}
  }
}
</b3m4>
Total Time (with thread locking) = 17.8034 seconds
Produced a total of 5040000 triples
...
```

Figure 3.11: An example of SCALE's execution output containing benchmarking-relevant information. Benchmarking-relevant information is wrapped in the blue HTML element <b3m4>. Inside the HTML element, the actual benchmarking data is outputted in a JSON-compatible form.

1. **analyze** what to benchmark based on a config file, thus define goals, like which PRFs and network settings (LAN vs. WAN);
2. **execute** benchmark for benchmark, until every chosen combination is run;
3. **parse** each run and extract the relevant information; and
4. **view** the results.

b4M's pipeline is going to be implemented in different modules. The modules are split into diffrent main modules and helper modules. The main modules are: **Main**, **Analyze**, **Execute**, **Parse**, and **View**. The helper modules are: **Utils** and **Config**. Each module has a different focus within the pipeline. Figure 3.12 illustrates the modules of b4M and their position within the pipeline.

### 3.5.2 Implementation

**Modifications in SCALE-MAMBA (SCALE).** Before starting with b4M, SCALE had already output some information of an execution of an MPC protocol. In order to get the relevant benchmark information in the right

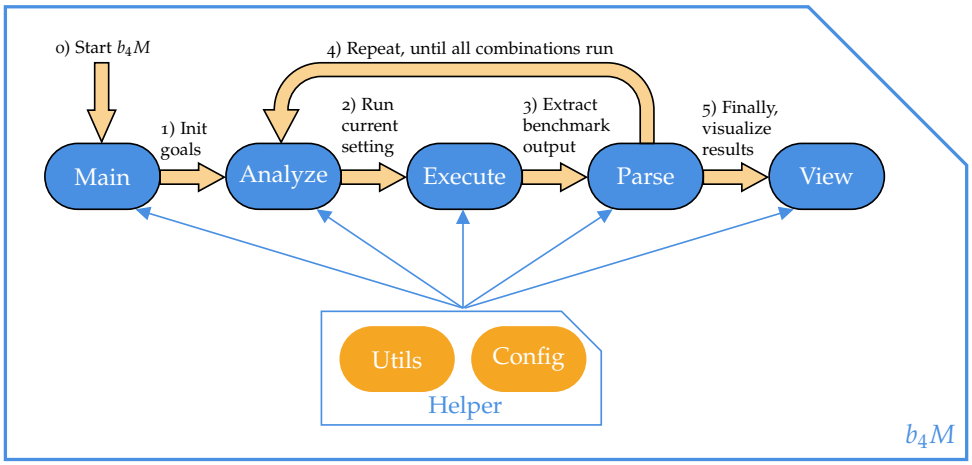


Figure 3.12: The different modules of b4M and their position within the pipeline. The blue modules are the main modules. The orange modules are the helper modules.

format, such as memory consumption and network traffic, we have to modify some parts of the source code. The modification includes only simple things.

Additionally, to even get the benchmark data, we have to tell SCALE to output it. This "telling" is implemented with compilation flags in SCALE. Now, to get benchmark output, the added compilation flags have to be set in the first place.

**Runtime measurements.** Taking measurements of the runtime was already possible before. To measure the runtime we have to call the $start\_timer(i)$ function for the start, and the $stop\_timer(i)$ function for the end of the timing. $i$ represents an integer value for the timer. With different integers for the timer, we can take different runtime measurements for different locations in the MAMBA code. Figure 3.13 shows a runtime measurement of a cipher initialization and of a message encryption in a MAMBA code.

Though, in order to get the runtime output in the required *HTML-JSON* format, we adapted the output in SCALE's `Machine::stop_timer` function, which is located in `src/Online/Machine`. Figure 3.14 shows the output of two timers after the MPC-program's execution. In the JSON output, we have first

```
...
start_timer(1)
cipher = init_regular(num_branches)
stop_timer(1)
...
start_timer(2)
priv_res = cipher.encrypt(message)
stop_timer(2)
...
```

Figure 3.13: Measuring the runtime of a cipher initialization and message encryption in the MAMBA code. The timers have the index 1 and 2, for the cipher initialization and message encryption respectively.

the timer's index, and then the runtime for the measured area in seconds and milliseconds (ms).

```
...
<b3m4>
{"timer":1,
  "time":{"seconds":179.018120,"ms":179018.1199}
}
</b3m4>
<b3m4>
{"timer":2,
  "time":{"seconds":0.005361,"ms":5.3614}
}
</b3m4>
...
```

Figure 3.14: Example benchmark output of the runtime for two different areas in the MAMBA code; indicated by the timers' indexes 1 and 2. The runtime is provided in the JSON object *time*, in the units seconds and milliseconds (ms). As in all benchmark-relevant outputs, the JSON-formatted runtime measurements are wrapped in the HTML elements <b3m4>.

**Communication measurements.** For communication, we added two counters; the counting of the sent and received bytes for each player (network data). Interestingly, the development of b4M and corresponding enhancement of benchmark measurements, led to a further advancement. Because the core

developers of SCALE added then also two more counters for the communication; the counting of the amount of broadcast as well as peer-to-peer messages (round data). These counters for network data and round data are added for each player in `src/System/Player`; specifically, in the functions:

- `Player::send_all`,
- `Player::send_to_player`, and
- `Player::receive_from_player`.

Furthermore, did we add there the function `Player::print_network_data`, which prints the relevant network data and round data in the required *HTML-JSON* format. `Player::print_network_data` is then called in the function `Main_Func`, which is located in `src/System/RunTime`, when the program is finished. As `Main_Func` is called for each player's execution thread, the network data and round data are given for each thread separetely. Thus, in b4M we have to combine the network data and round data of the different threads for each player.

To activate the benchmark output for communication, we introduced a benchmark flag for SCALE's config file, `CONFIG`. `CONFIG` needs to be renamed or copied to `CONFIG.mine` in order to work. Figure 3.15 displays the activated communication compilation flag. Now, when we compile SCALE with this flag and run an MPC program, we get the benchmark output for communication. Figure 3.16 shows the output of player 0 for thread 2 and 4. In the JSON output, we have first the player number and thread number, and then the network data, given in bytes and kilobytes (KB), as well as round data, given in amount of messages.

**Memory-consumption measurements.** For memory consumption, we add the measurement of the peak RAM usage for each execution process. The maximum resident set size gives us the the peak RAM usage of the calling process [Wik20e; Wik20b]. This measurement is taken when the MPC program is finished, and located in `src/System/RunTime`. We added there the function `Print_Memory_Info`.
`Print_Memory_Info` prints the maximum resident set size in the required *HTML-JSON* format.

```
...
# Benchmark flags
#   BENCH_COMMUNICATION = Enable benchmark output of
#                         communication. This includes
#                         sent & received bytes,
#                         rounds of communication, and
#                         amount of peer-to-peer as well as
#                         broadcast messages.
...
FLAGS = -DBENCH_COMMUNICATION
...
```

Figure 3.15: SCALE's compilation flag for the benchmark output of communication, which is located in the file `CONFIG` and `CONFIG.mine` respectively.

To activate the benchmark output of memory consumption, we added also for this case a compilation flag in SCALE's config file; `CONFIG` and `CONFIG.mine` respectively. Figure 3.17 displays the activated memory-consumption compilation flag. The different compilation flags can be combined; thus, even all of them could be activated at the same time. Figure 3.18 shows the activated flags for debug information, communication, memory consumption, and deterministic computation.

Now, when we compile SCALE with the memory-consumption flag and run an MPC program, we get the memory-consumption benchmark output. Figure 3.19 shows the memory-consumption of player 0 for thread 2 and 4. In the JSON output, we have first the player number, thread number, and process number, and then the maximum resident set size, given in kilobytes (KB) and megabytes (MB).

**Main Modules of Benchmarking for MPC (b4M).** b4M consists of five main modules: **Main**, **Analyze**, **Execute**, **Parse**, and **View**. **Main** is the entry point of b4M and does the startup procedures as well as cleanup after the benchmarking. In the startup procedures **Main** checks if the environment is properly set up, such as if required folders and files are present. The cleanup removes temporarily created files which are not needed anymore.

**Analyze** first checks the current goal, which is set in the helper module Config.

```
...
<b3m4>
{"player":0,
  "thread":2,
  "netdata":{
    "sent":{"bytes":8423594,"MB":8.42},
    "received":{"bytes":17827569,"MB":17.83}
  },
  "roundsdata":{
    "broadcast":6738,
    "p-to-p":71
  }
}
</b3m4>
...
<b3m4>
{"player":0,
  "thread":4,
  "netdata":{
    "sent":{"bytes":812,"MB":0.00},
    "received":{"bytes":812,"MB":0.00}
  },
  "roundsdata":{
    "broadcast":4,
    "p-to-p":2
  }
}
</b3m4>
...
```

Figure 3.16: Example benchmark output of the communication for the execution threads 2 and 4 of player 0. Sent and received bytes are provided in the JSON object *netdata*, in the units bytes and megabytes (MB). The amount of broadcast and peer-to-peer messages are provided in the JSON object *roundsdata*. As in all benchmark-relevant outputs, the JSON-formatted communication measurements are wrapped in the HTML elements <b3m4>.

Then, based on the current goal **Analyze** prepares the set of combinations of benchmark dimensions to be executed. After the preparation of the set of combinations, each combination gets forwarded to the main module Execute. Finally, when all combinations were executed, **Analyze** calls the main module View.

```
...
# Benchmark flags
# ...
#   BENCH_MEMORY  = Enable benchmarking of memory consumption.
#                   This includes the peak RAM usage of each
#                   player's execution thread.
#
...
FLAGS = -DBENCH_MEMORY
...
```

Figure 3.17: SCALE's compilation flag for the benchmark output of memory consumption, which is located in the file `CONFIG` and `CONFIG.mine` respectively.

```
...
FLAGS = -DDEBUG -DDETERMINISTIC -DBENCH_MEMORY -DBENCH_COMMU...
...
```

Figure 3.18: Combination of compilation flags, which is located in the file `CONFIG` and `CONFIG.mine` respectively. The activated flags here are for debug information, deterministic computation, memory-consumption benchmark output, and communciation benchmark output. SCALE's compilation flags can be arbitrarily combined. Even all flags could be activated at the same time.

**Execute** performs a benchmark of a given set of combinations of benchmark dimensions. To perform the benchmark, this main module does basically four things:

1. First, **Execute** prepares everything that is needed for the execution; such as loading the access structure or setting the IP addresses for the participating nodes in the corresponding network environment.
2. Second, **Execute** runs the MPC program with the given settings.
3. Third, after the run of the MPC program, **Execute** calls the main module Parse, to parse the output of the MPC-program's execution. Depending on the settings in the helper module Config, the execution-and-parse procedure might be repeated until the number of runs per benchmark has been reached.
4. And fourth, **Execute** calls again the main module Parse, to combine the parsed output of each benchmarks' run, as well as tracks the progress of

```
...
<b3m4>
{"player":0,
  "thread":2,
  "process":1,
  "memory":{
    "max_rss":{"KB":7301972,"MB":7301.97}
  }
}
</b3m4>
...
<b3m4>
{"player":0,
  "thread":4,
  "process":1,
  "memory":{
    "max_rss":{"KB":7301972,"MB":7301.97}
  }
}
</b3m4>
...
```

Figure 3.19: Example benchmark output of the memory consumption for the execution threads 2 and 4 of player 0. The maximum resident set size is provided in the JSON object *max_rss*, in the units kilobytes (KB) and megabytes (MB). As in all benchmark-relevant outputs, the JSON-formatted memory-consumption measurements are wrapped in the HTML elements <b3m4>.

the current benchmark. The tracking of the progress helps, when a run fails. Because then, the benchmarks which have already been succesfully performed can be ignored, and the benchmarking continues with the previously failed run.

**Parse** provides mainly two utilities:

1. First, the parsing of an MPC-program run's output; which includes, e.g., the parsing of runtime, communication, and memory consumption. What can be parsed exactly depends on the setting; if SCALE was compiled with the benchmark flags and a timing was taken in the MAMBA code respectively.
2. And second, **Parse** provides the combining of the different runs of a

benchmark. In this combination of benchmark runs the average runtime, network data, and memory consumption is calculated.

**View** illustrates the benchmark results. For the illustration, **View** provides support for graphs.Which graph should be created can be defined in the helper module Config.

**Helper Modules of b4M.** In addition to the five main modules, b4M also consists of two helper modules: **Config** and **Utils**. **Config** provides all essential settings for the preparation, execution, and parsing of runs, as well as for illustrating the benchmark results.

**Utils**, as the name already tells, provides helper utility functions and classes. These functions include, e.g., getting of parsed JSON data or checking if a file exists. Besides the helper functions, **Utils** also defines the class *Benchmark*, which is used throughout the whole benchmarking pipeline (from Analyze to View).

**Programming language & usage.** b4M is implemented in Python 3. Each module of b4M is a single file. The entry point for starting the benchmarking is the main module **Main**. In order to configure the benchmarking, the helper module **Config** is used. Thus, first, the desired settings are applied in **Config**; and second, the benchmarking is started using **Main**.

b4M can be started either by using Python, `python3 b3m4_main.py`, or by executing the main module **Main** directly, `./b3m4_main.py`. However, to have all required Python3 libraries available, the libraries in the file `requirements.txt` need to be installed first. At the moment of writing the thesis, the *external* Python3 libraries, which need to be installed manually usually, are:

- *lxml* [Dev20b], for parsing the HTML elements in the execution's output.
- *tqdm* [Dev20a], for showing the benchmarking's progress on the command line.
- *pandas* [Tea20], for preparing the benchmarking's JSON data for plotting, by converting it into a `pandas DataFrame`.
- *seaborn* [Was20], for plotting the benchmarkings's outcome as graphs.

# Chapter 4

# Performance Evaluation & Recommendations

This chapter (1) describes how the different benchmarks were performed, (2) shows the results of the benchmarks, and (3) gives recommendations based on the different secnarios. These three steps are baked into the two main points of our benchmarking plan: the sanity check and recommendations with a set of requirements in mind. Though, step (3) of this chapter essentially applies to the recommendations for real-life scenarios.

## 4.1 Benchmark Environment

**Executing machine.** The benchmarks were taken on an *Intel(R) Xeon(R) CPU E5-4669 v4 @ 2.20GHz* having 88 threads with (basically) 192$GB$ of RAM. However, for the actual secure-multi-party-computation (MPC) benchmarks we limited the amount of RAM and threads per player. Each player got assigned 30$GB$ of RAM and 10 threads.

**Network environment.** For simulating the two different network conditions, we used a local area network (LAN) and set up a wide area network (WAN) environment respectively. The LAN network provides a bandwith of $\sim$1$Gbit/s$ with a latency of $\sim$0.05$ms$. The WAN network provides a bandwith of $\sim$50$Mbit/s$ with a latency of $\sim$100$ms$. And please note that the two dedicated MPC-friendly networks were installed by the Institute of Applied Information Processing and Communications (IAIK)'s sysadmins.

**Benchmark target.** We use the same benchmark target as stated in the "baseline" paper by [Gra+16]. Thus, only pseudo-random function (PRF) encryptions of one $\mathbb{F}_p$ element are considered and benchmarked.

## 4.2 Sanity Check

This section describes how we assure that our benchmarking results are in a reasonable range. For this we do two things. First, getting the measurements of the three main components of the preprocessing phase: triples, squares, and bits. Second, running the same experiment as in the paper by [Gra+16], which was the baseline for this thesis, and doing a comparison. For the comparison we check on the one hand, if it seems reasonable with respect to the preprocessing measurements, and on the other hand, if the measurements are similar to those of the paper by [Gra+16].

### 4.2.1 Preprocessing Measurements

**Taking benchmarks.** In order to take the benchmarks for the preprocessing phase, we write three MAMBA programs; one for each main component of the preprocessing:

- `triple-demo.mpc`

```
import sys
params = [int(_) for _ in sys.argv[3:]]
n = params[0] # number of total triples to be produced
n_parallel = params[1] # number of triples processed in
                       # parallel

start_timer(2)
@for_range(n / n_parallel)
def _(i):
    a = sint.get_random_triple(size=n_parallel)
stop_timer(2)
```

- `square-demo.mpc`

```
import sys
params = [int(_) for _ in sys.argv[3:]]
n = params[0] # number of total squares to be produced
n_parallel = params[1] # number of squares processed in
                       # parallel

start_timer(2)
@for_range(n / n_parallel)
def _(i):
    a = sint.get_random_square(size=n_parallel)
stop_timer(2)
```

- `bit-demo.mpc`

```
import sys
params = [int(_) for _ in sys.argv[3:]]
n = params[0] # number of total bits to be produced
n_parallel = params[1] # number of bits processed in
                       # parallel

start_timer(2)
@for_range(n / n_parallel)
def _(i):
    a = sint.get_random_bit(size=n_parallel)
stop_timer(2)
```

As SCALE-MAMBA (SCALE) usually produces more triples, squares, and bits than needed, to have leftovers for a potential need in the future, we also add the execution parameter `max`. With the parameter `max` it is possible to restrict the amount of produced triples, squares, and bits. Thus, when measuring, e.g., triples, we restrict the production of squares and bits to really have the measurements for the produced triples. Though, as a value of 0 would mean infinite production, SCALE will produce at least one batch of triples, squares, and bits; that is why we set the production of the other components to 1. To have a long-enough runtime and way more, e.g., triples than squares and bits, we produce 1,000,000 triples. Given this setting, these are then our values for the execution parameter `max`:

- triples: `-max 1000000,1,1`,
- squares: `-max 1,1000000,1`, and
- bits: `-max 1,1,1000000`.

With respect to our identified benchmark dimensions (in the previous chapter), Figure 4.1 shows the different settings of this benchmark.

| Dimension | Value(s) |
|---|---|
| PRF | - |
| #Triples/Squares/Bits | 1,000,000 |
| Batch size | 1,2,4,8...,16384 |
| #Evaluations | - |
| Branch size | - |
| #Players | 2 |
| Network | LAN, WAN |
| Preprocessing | Real |
| Access structure (ASX) | #18 |

Table 4.1: The set of dimensions of the experiment setup for the preprocessing measurements as part of our sanity check. The bluely highlighted dimensions are the variables in this experiment. The grayly highlighted dimensons are variables which are not used in this experiment; they are added for reasons of comparison.

**Benchmark results & evaluation of thereof.** Figure 4.1 illustrates the throughput of the experiment. Figure 4.2 and 4.3 show the latency and runtime respectively. Please note that due to the nature of computing the latency, the graph is identical to the runtime's one, it just has different numbers. As expected, due to the slower network in the wide-area-network (WAN) setting, SCALE reaches more throughput in the local-area-network (LAN) setting. Triples have in both network settins the slowest throughput. Interestingly, squares have a relatively much higher throughput in the LAN setting than in the WAN setting, when compared to bits, for instance. Also interestingly, bits have almost the same throughput as triples.

For latency and runtime, the differences are not that significant. In the LAN setting, the latency between triples ($\sim 0.8ms$) and squares ($\sim 0.7ms$) differs only $\sim 0.1ms$ In the WAN setting, the latency between triples ($\sim 1.9ms$) and sqares ($\sim 1.6ms$) differs a bit more, but still only $\sim 0.3ms$. In the paper by [Gra+16], the latency of triples and bits is identical. In our preprocessing measurements, the latency of triples and bits is almost identical; in the LAN setting yes, in the WAN setting they differ $\sim 0.2ms$.

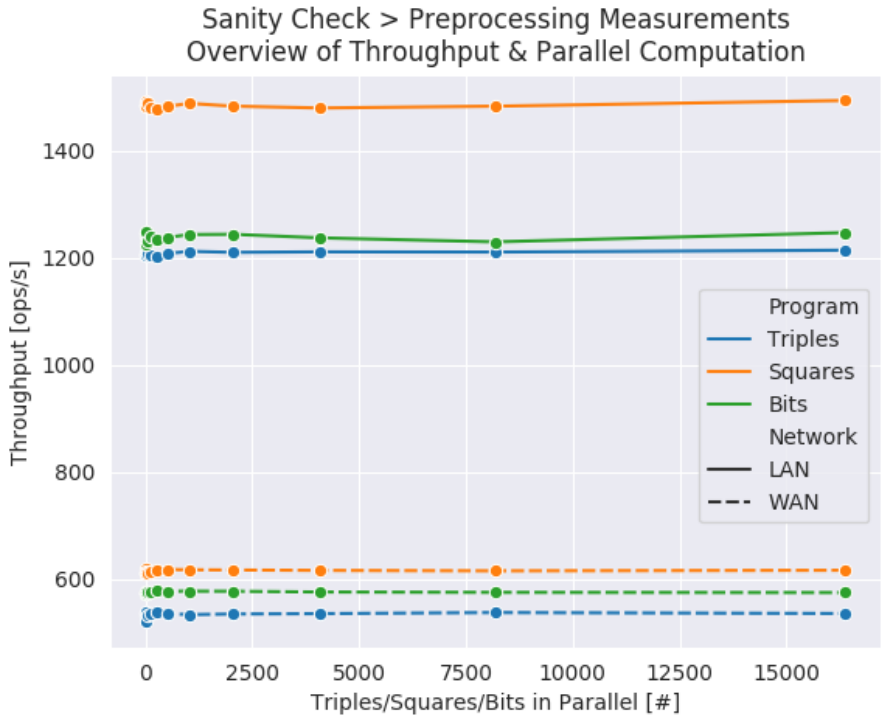


Figure 4.1: Resulting throughput of our preprocessing measurements, as part of our sanity check, with parallel computation.

### 4.2.2 PRF Evaluations

**Taking benchmarks.** To take benchmarks as in the paper by [Gra+16], we use a MAMBA program for each PRF to benchmark:

- `MiMC-Cube-128.mpc`
- `Legendre-128.mpc`

Please note that the MAMBA code of `MiMC-Cube-128.mpc` and `Legendre-128.mpc` was provided by the authors of [Gra+16].

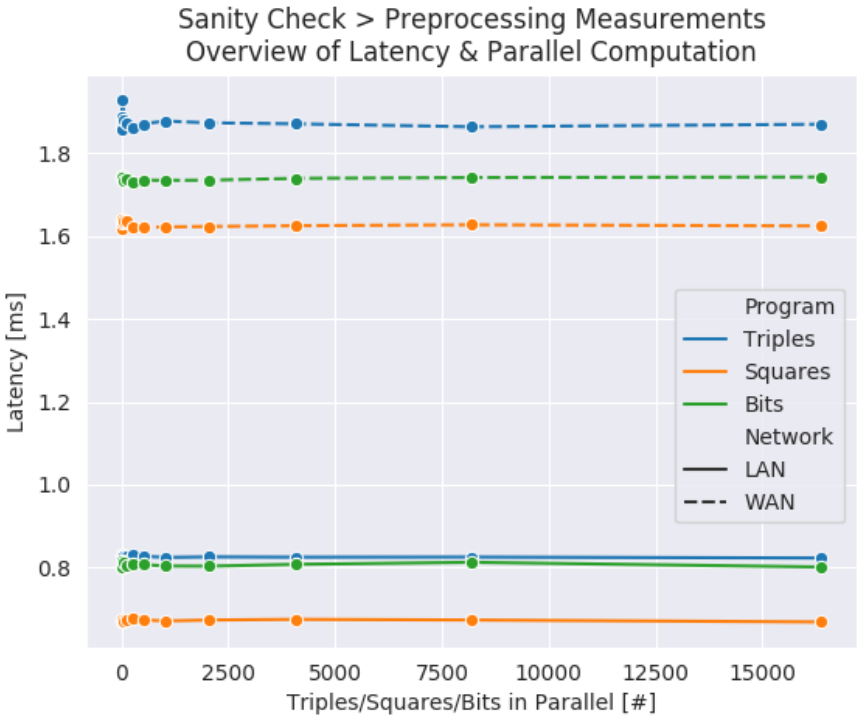


Figure 4.2: Resulting latency of our preprocessing measurements, as part of our sanity check, with parallel computation.

**Benchmark results & evaluation of thereof.** Figure 4.4 shows the resulting throughput of our benchmarks for the comparison with our "baseline" paper. Figures 4.5 and 4.6 show the resulting latency and runtime respectively.

As we measured the preprocessing and online phase together, and [Gra+16] measured the two phases separately, we cannot do a simple one-to-one comparison. Thus we approximate by comparing with their preprocessing phase. Our throughput for Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity (MiMC) is in the LAN setting significantly slower, but basically the same in the WAN setting (LAN: $\sim 5\ ops/s$ (ours) vs. $\sim 33\ ops/s$; WAN: $\sim 1.5\ ops/s$ (ours) vs. $\sim 1.6\ ops/s$). Our throughput for Legendre (Leg), on the other side, is not too different for LAN and even better for WAN (LAN: $\sim 4.5\ ops/s$ (ours) vs. $\sim 9\ ops/s$; WAN: $\sim 1.5\ ops/s$ (ours)

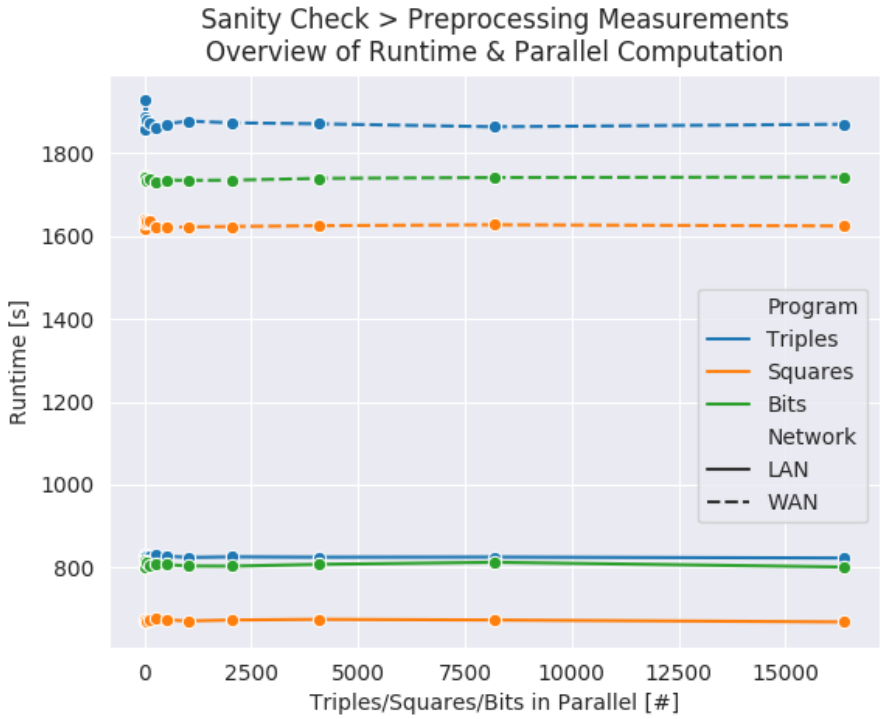


Figure 4.3: Resulting runtime of our preprocessing measurements, as part of our sanity check, with parallel computation.

vs. $\sim 0.5\ ops/s$). The latency, however, is only shown for the online phase in our "baseline" paper.

## 4.3 Recommendations with a Set of Requirements in Mind

**Taking benchmarks.** In order to showcase Benchmarking for MPC (b4M) for the LAN use case, we benchmarked our selected PRFs with two different branch sizes: one and two. With a branch size of one, we benchmarked MiMC and Leg. With a branch size of two, we benchmarked MiMC, Generalized

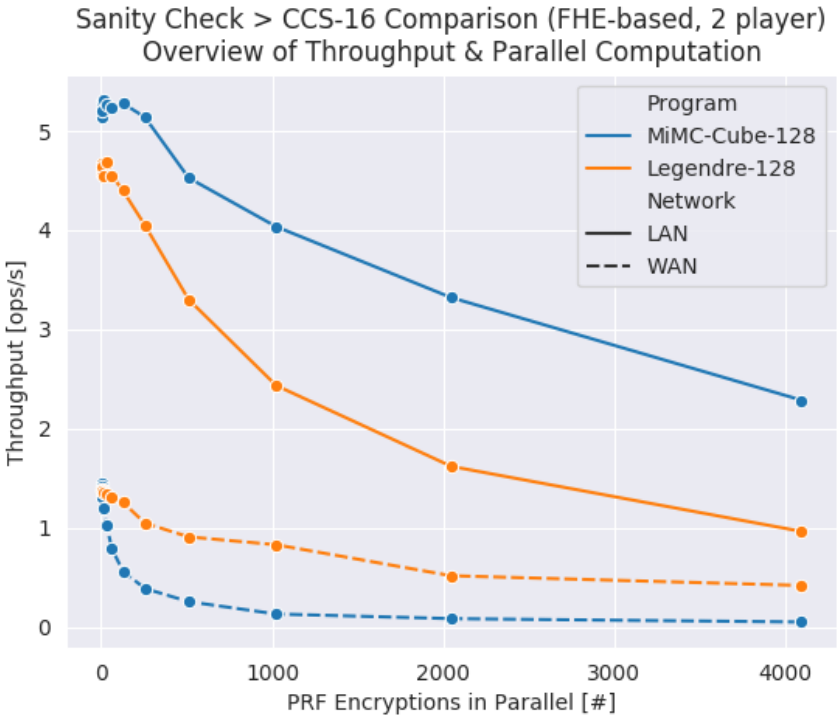


Figure 4.4: Resulting throughput of our comparison measurements with the paper by [Gra+16], as part of our sanity check, with parallel computation.

Feistel MiMC (GMiMC)-expanding round function (ERF), and HADES MiMC (HMiMC). To demonstrate the difference between the two phases of the Smart-Pastro-Damgård-Zakarias (SPDZ) protocol, each setting got benchmarked with (1) only the online phase, and (2) the preprocessing and the online phase.

Primarily, for both LAN and WAN, taking benchmarks with the different ASXs was taken with a batch size of 1, 2, 4, and 512. As the batch sizes in between could be interesting too, we also show this case for our selected dishonest-majority PRF with only the only phase. We chose the maximum batch size of 512 due to runtime issues for some settings with a higher batch size.

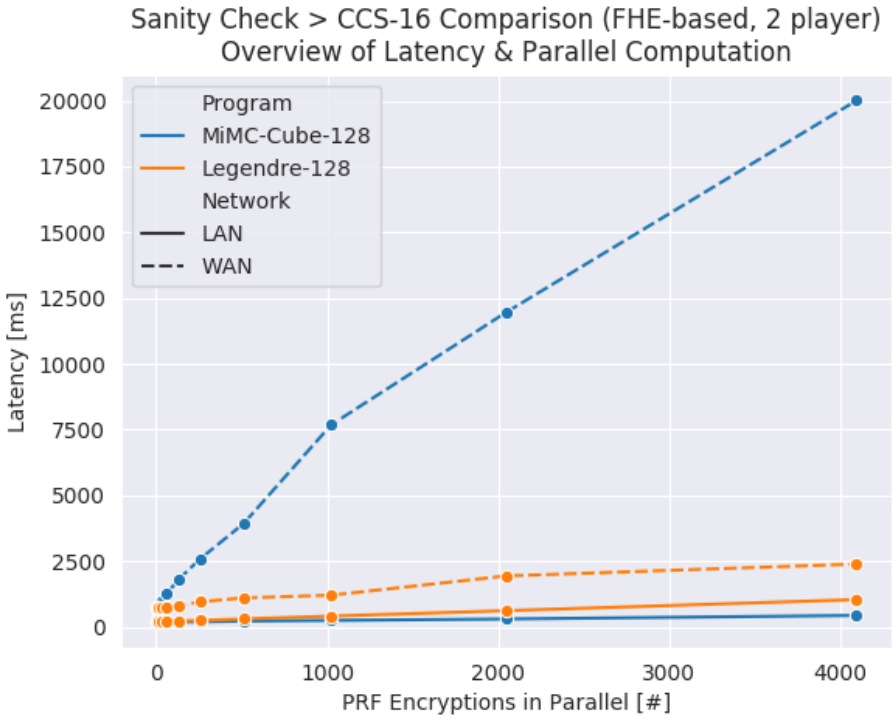


Figure 4.5: Resulting latency of our comparison measurements with the paper by [Gra+16], as part of our sanity check, with parallel computation.

In the graphs' legend, the programs are shown in different colors and the ASXs are shown with different kinds of dashed lines. Our dishonest-majority ASX is shown as *FHE-based**. Our honest-majority ASXs are shown as stated in Section 4.1, just without *(3,1)*.

### 4.3.1 Focus on Runtime Metrics in a LAN Network

**Benchmark results of 3-players LAN.** The following figures show the results of our 3-players LAN benchmarks:

- Branch size = 1
  - Preprocessing and online phase

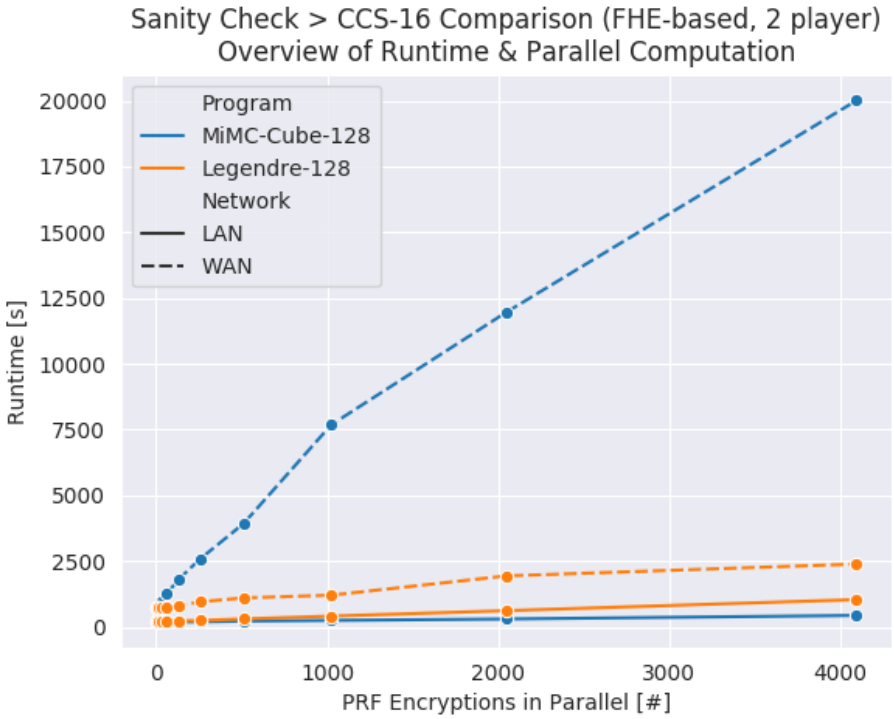


Figure 4.6: Resulting runtime of our comparison measurements with the paper by [Gra+16], as part of our sanity check, with parallel computation.

    - Honest-majority and dishonest-majority ASXs
        - Latency: Figure 4.7
        - Throughput: Figure 4.8
    - Honest-majority ASXs
        - Latency: Figure 4.9
        - Throughput: Figure 4.10
    - Dishonest-majority ASX
        - Latency: Figure 4.11
        - Throughput: Figure 4.12
- Only online phase
    - Honest-majority and dishonest-majority ASXs

        - Latency: Figure 4.13
        - Throughput: Figure 4.14
      - Honest-majority ASXs
        - Latency: Figure 4.15
        - Throughput: Figure 4.16
      - Dishonest-majority ASX
        - Latency: Figure 4.17
        - Throughput: Figure 4.18
- Branch size = 2
  - Preprocessing and online phase
    - Honest-majority and dishonest-majority ASXs
      - Latency: Figure 4.19
      - Throughput: Figure 4.20
    - Honest-majority ASXs
      - Latency: Figure 4.21
      - Throughput: Figure 4.22
    - Dishonest-majority ASX
      - Latency: Figure 4.23
      - Throughput: Figure 4.24
  - Only online phase
    - Honest-majority and dishonest-majority ASXs
      - Latency: Figure 4.25
      - Throughput: Figure 4.26
    - Honest-majority ASXs
      - Latency: Figure 4.27
      - Throughput: Figure 4.28
    - Dishonest-majority ASX
      - Latency: Figure 4.29
      - Throughput: Figure 4.30

**Evaluation of branch size = 1.** As expected, the ASX for dishonest majority has a significantly worse throughput and latency than the honest-majority ASXs for the preprocessing and online phase. However, when considering

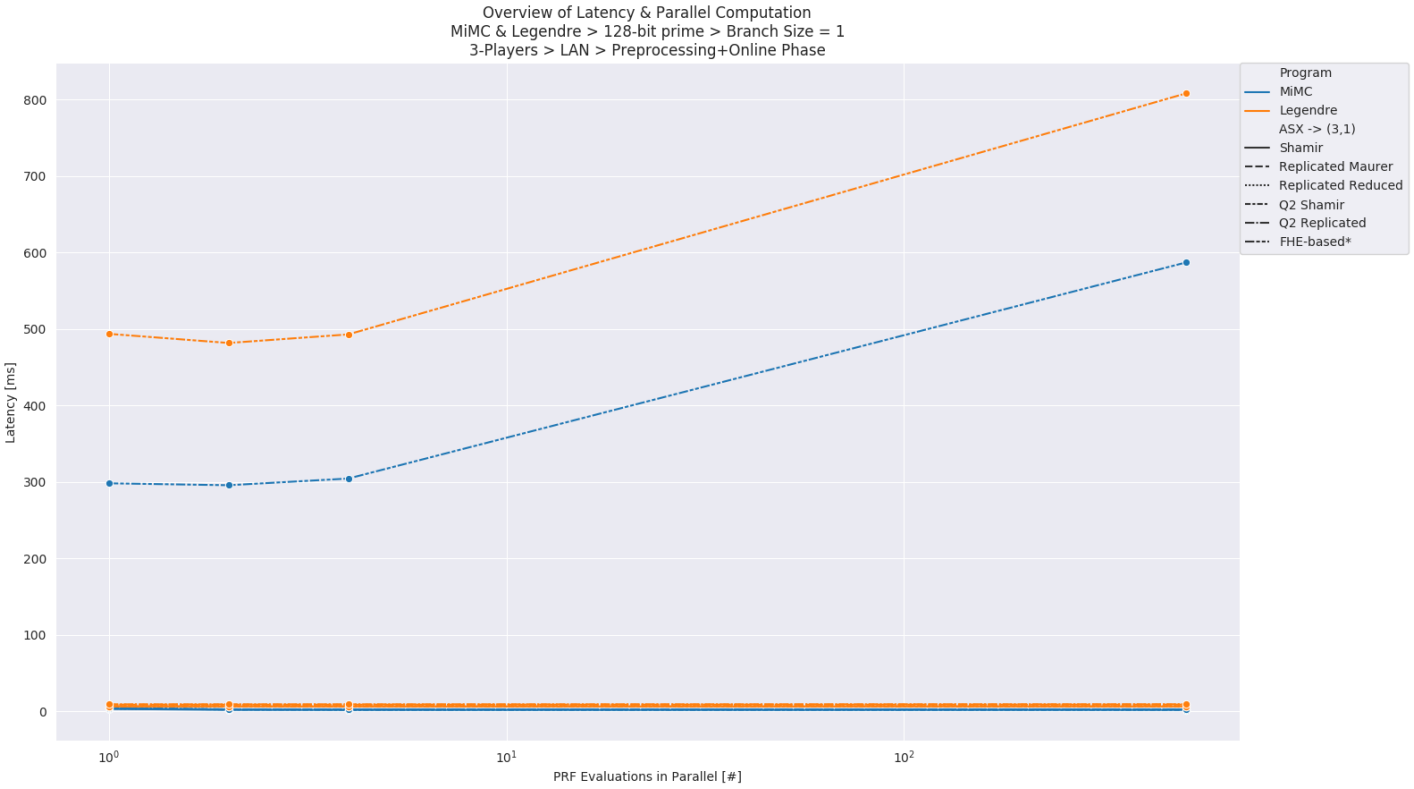


Figure 4.7: The latency of the 3-players-LAN use case for MiMC and Leg using a branch size of one with the preprocessing and online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

only the online phase, the dishonest-majority ASX outperforms the other ASXs.

In terms of PRFs, MiMC outperforms Leg for all throughput and latency measurements when the batch size is 512, especially when considering the preprocessing and online phase.

Interestingly, MiMC benefits in all scenarios of a higher batch size, especially when it equals 512, except for the case of the dishonest-majority ASX when considering preprocessing and online phase. For this one case the latency and throughput becomes even worse with a batch size of $> 2$, for both MiMC and

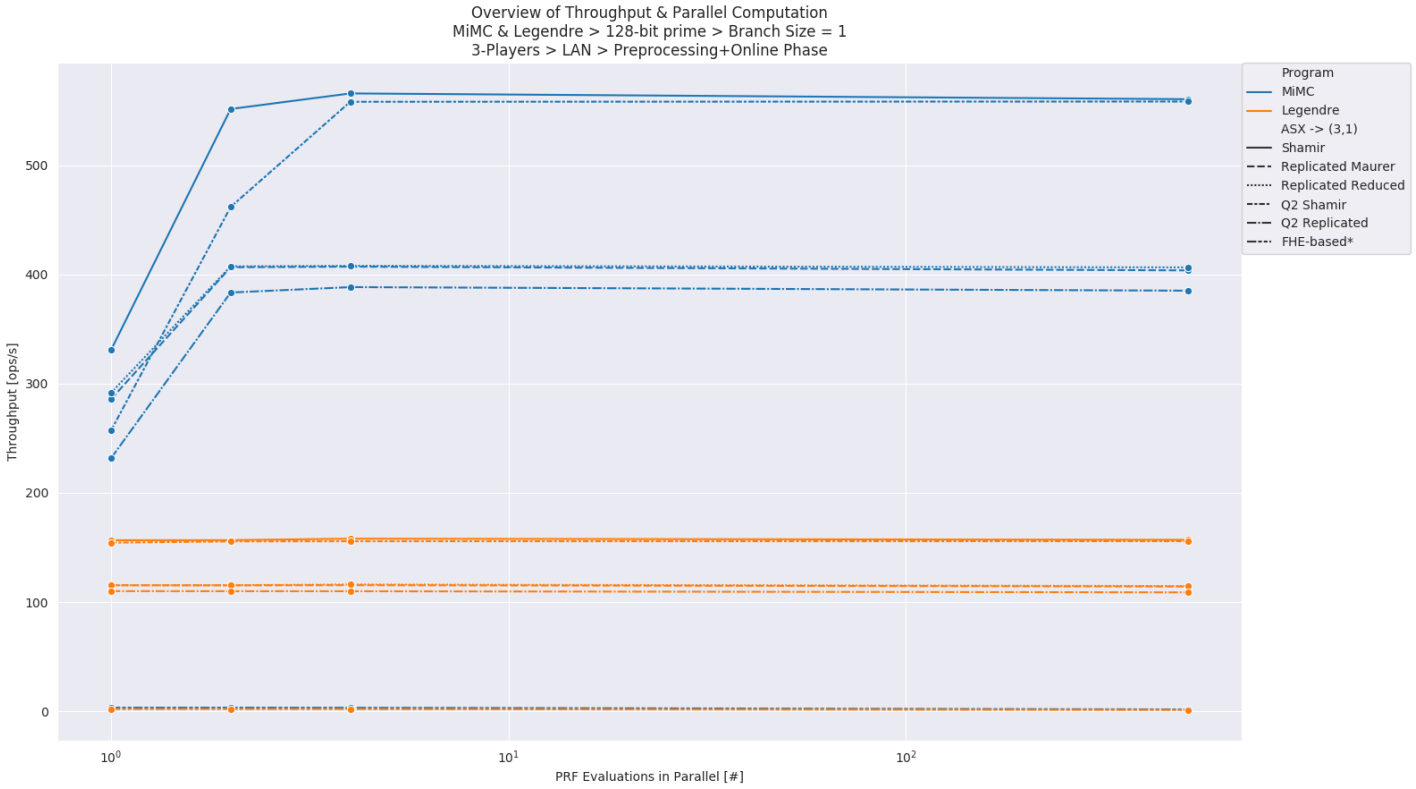


Figure 4.8: The throughput of the 3-players-LAN use case for MiMC and Leg using a branch size of one with the preprocessing and online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

Leg. Except for this one case, in general, Leg does not benefit too much from a higher batch size.

**Evaluation of branch size = 2.** For the evalutions of a branch size of 2, it is interesting that HMiMC outperforms MiMC and GMiMC, but not in all ASX combinations. For instance, for the dishonest-majority ASXs with the preprocessing and online phase, and a batch size of $\geq 4$, MiMC with the Shamir-based ASXs is slightly better than HMiMC with the Replicated-based ASXs, for both throughput and latency. And in general, the Shamir-based

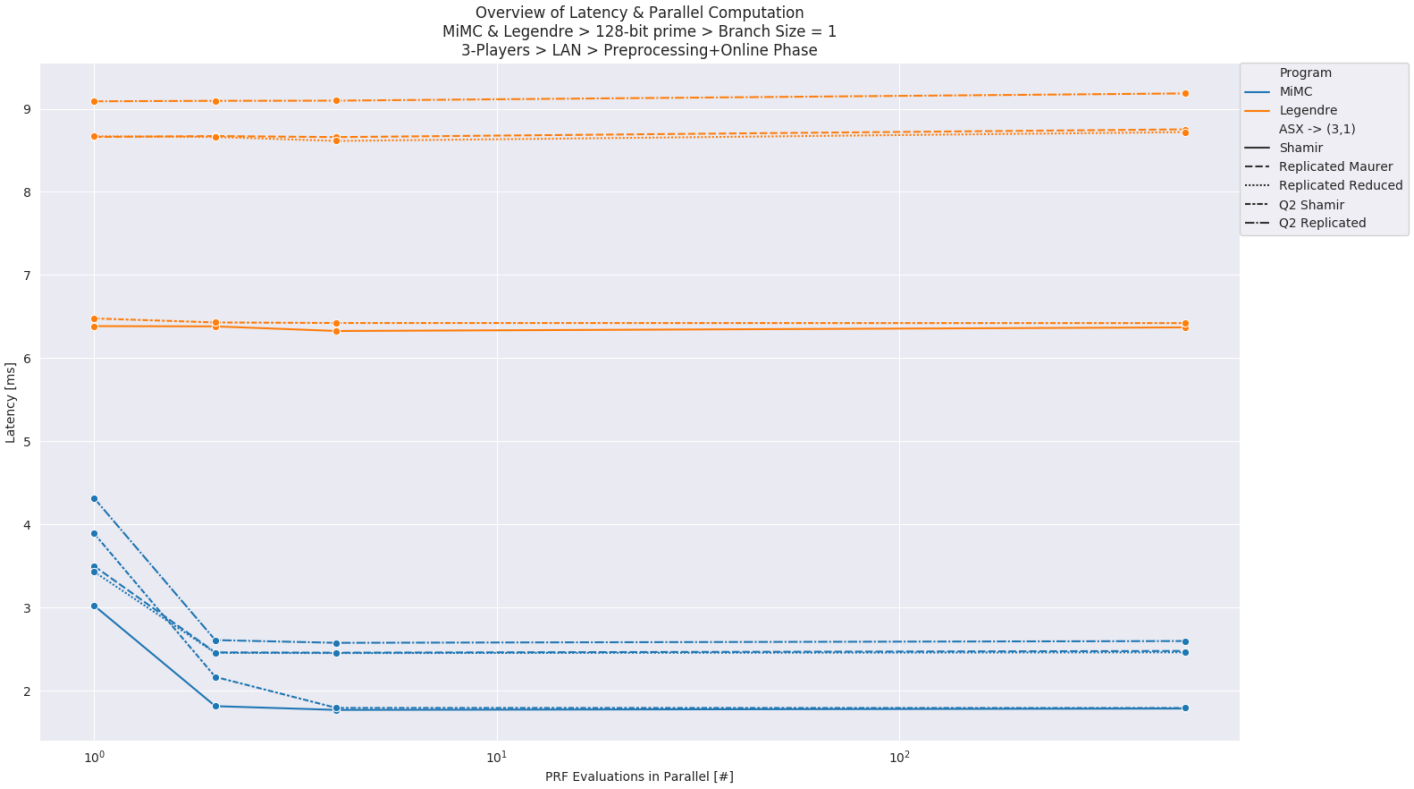


Figure 4.9: The latency of the 3-players-LAN use case for MiMC and Leg using a branch size of one with the preprocessing and online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

ASXs achieve a significantly higher throughput and latency than the other dishonest-majority ASXs.

### 4.3.2 Focus on Runtime Metrics in a WAN Network

**Benchmark results of 3-players WAN.** The following figures show the results of our 3-players LAN benchmarks:

- Branch size = 1

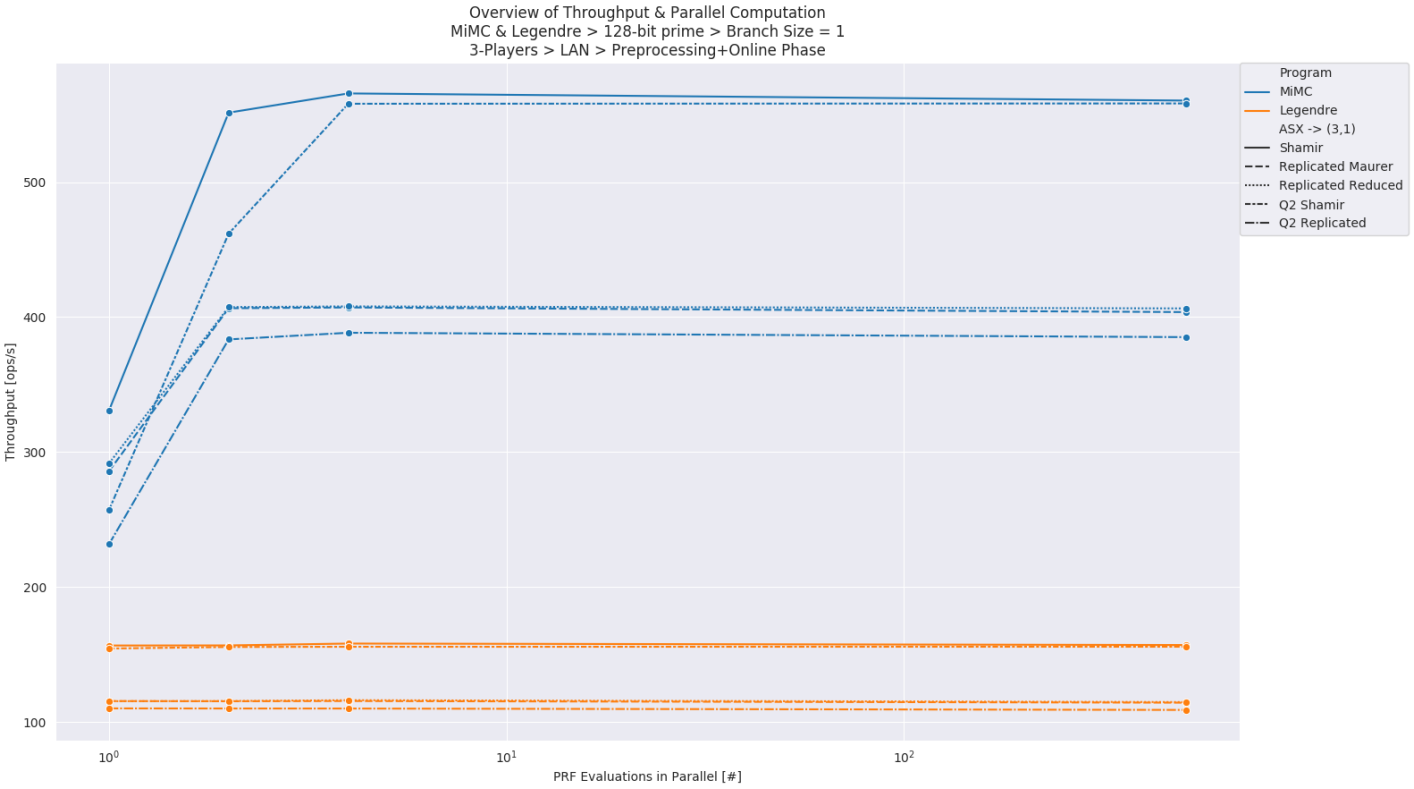


Figure 4.10: The throughput of the 3-players-LAN use case for MiMC and Leg using a branch size of one with the preprocessing and online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

- Preprocessing and online phase
  - Honest-majority and dishonest-majority ASXs
    - Latency: Figure 4.31
    - Throughput: Figure 4.32
  - Honest-majority ASXs
    - Latency: Figure 4.33
    - Throughput: Figure 4.34
  - Dishonest-majority ASX
    - Latency: Figure 4.35

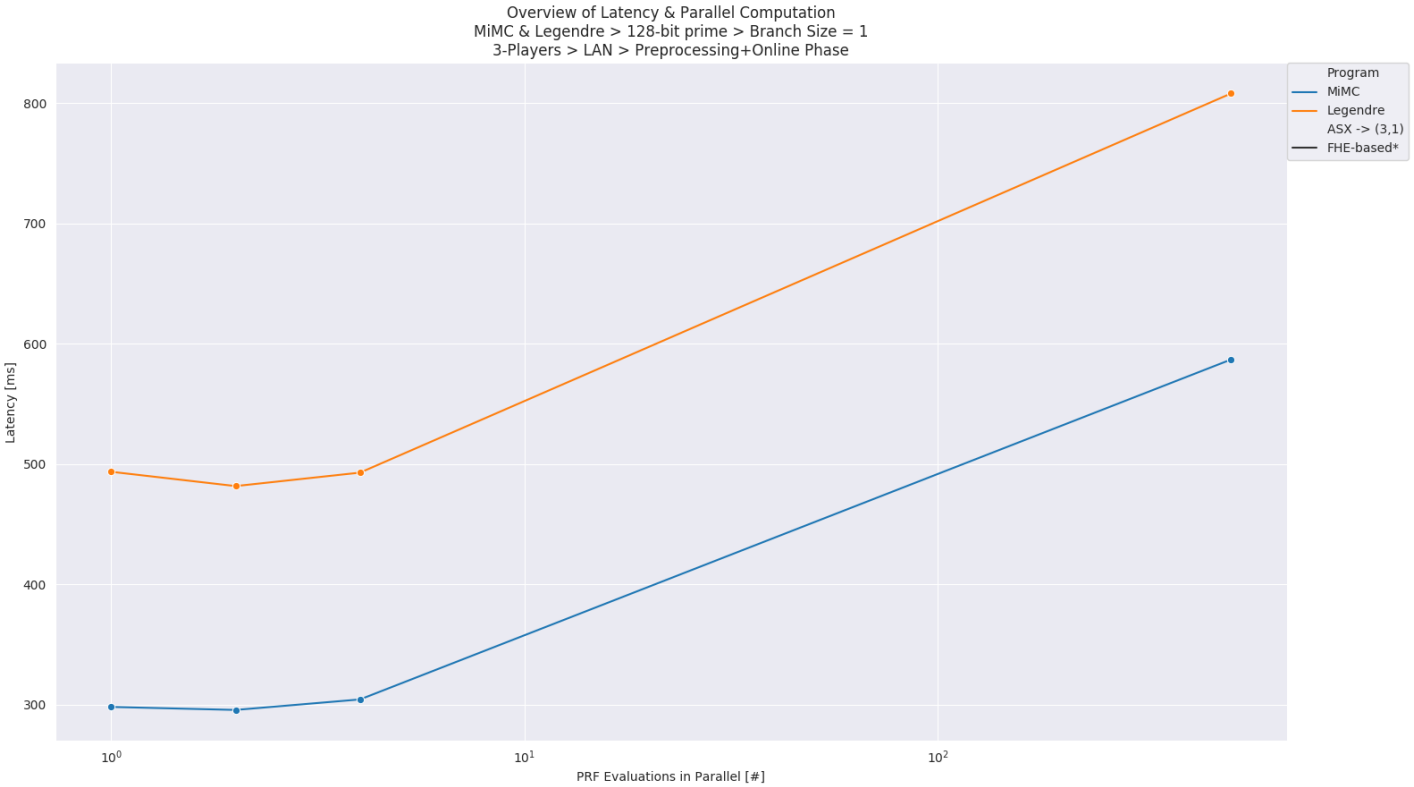


Figure 4.11: The latency of the 3-players-LAN use case for MiMC and Leg using a branch size of one with the preprocessing and online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4, and 512.

    - Throughput: Figure 4.36
- Only online phase
    - Honest-majority and dishonest-majority ASXs
        - Latency: Figure 4.37
        - Throughput: Figure 4.38
    - Honest-majority ASXs
        - Latency: Figure 4.39
        - Throughput: Figure 4.40
    - Dishonest-majority ASX

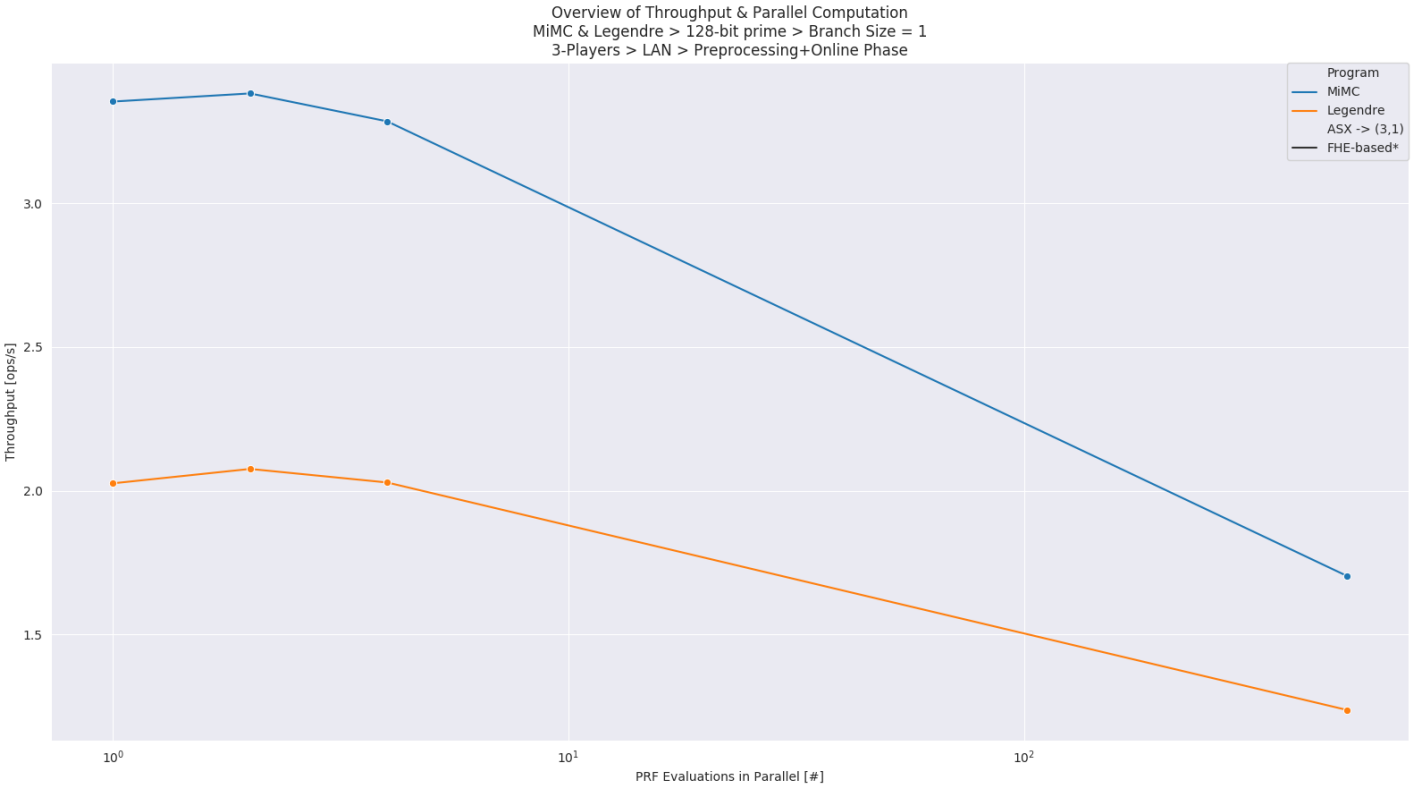


Figure 4.12: The throughput of the 3-players-LAN use case for MiMC and Leg using a branch size of one with the preprocessing and online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4, and 512.

        - Latency: Figure 4.41
        - Throughput: Figure 4.42
- Branch size = 2
    - Preprocessing and online phase
        - Honest-majority and dishonest-majority ASXs
            - Latency: Figure 4.43
            - Throughput: Figure 4.44
        - Honest-majority ASXs
            - Latency: Figure 4.45

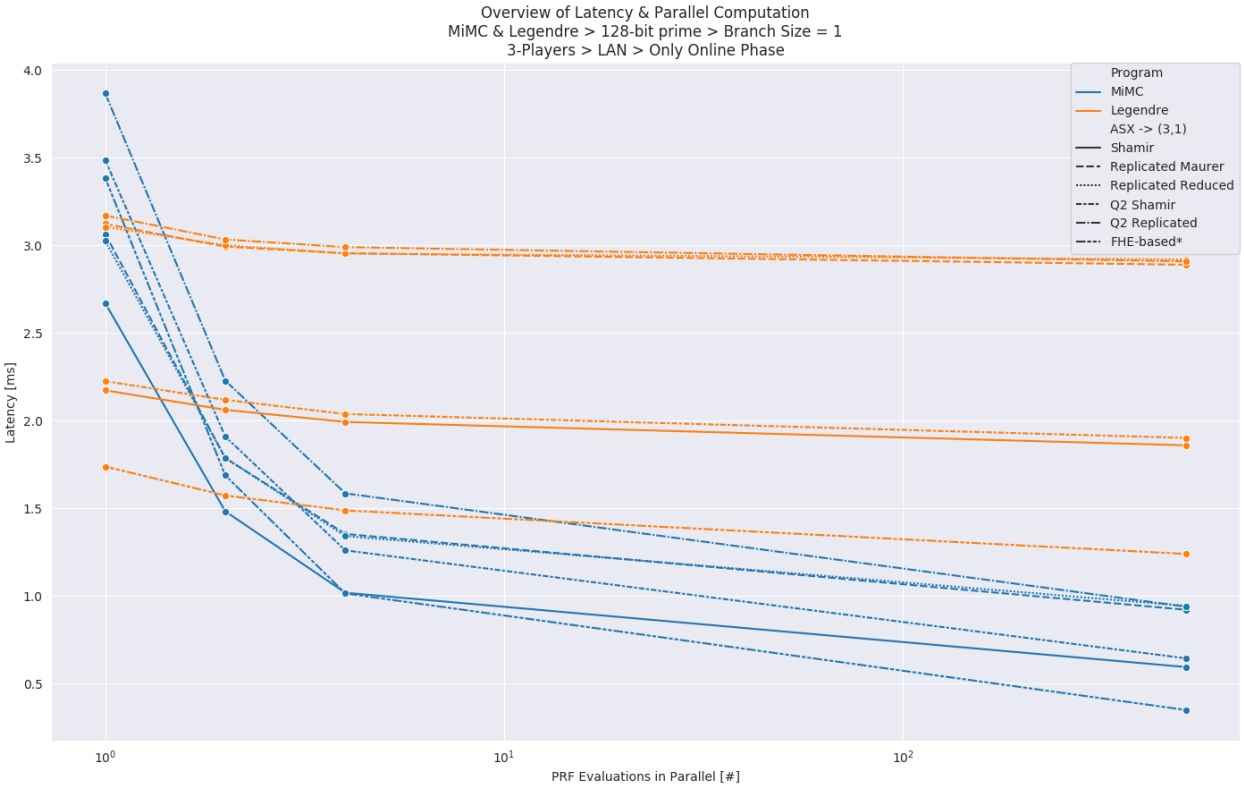


Figure 4.13: The latency of the 3-players-LAN use case for MiMC and Leg using a branch size of one with only the online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

        - Throughput: Figure 4.46
    - Dishonest-majority ASX
        - Latency: Figure 4.47
        - Throughput: Figure 4.48
- Only online phase
    - Honest-majority and dishonest-majority ASXs
        - Latency: Figure 4.49
        - Throughput: Figure 4.50
    - Honest-majority ASXs

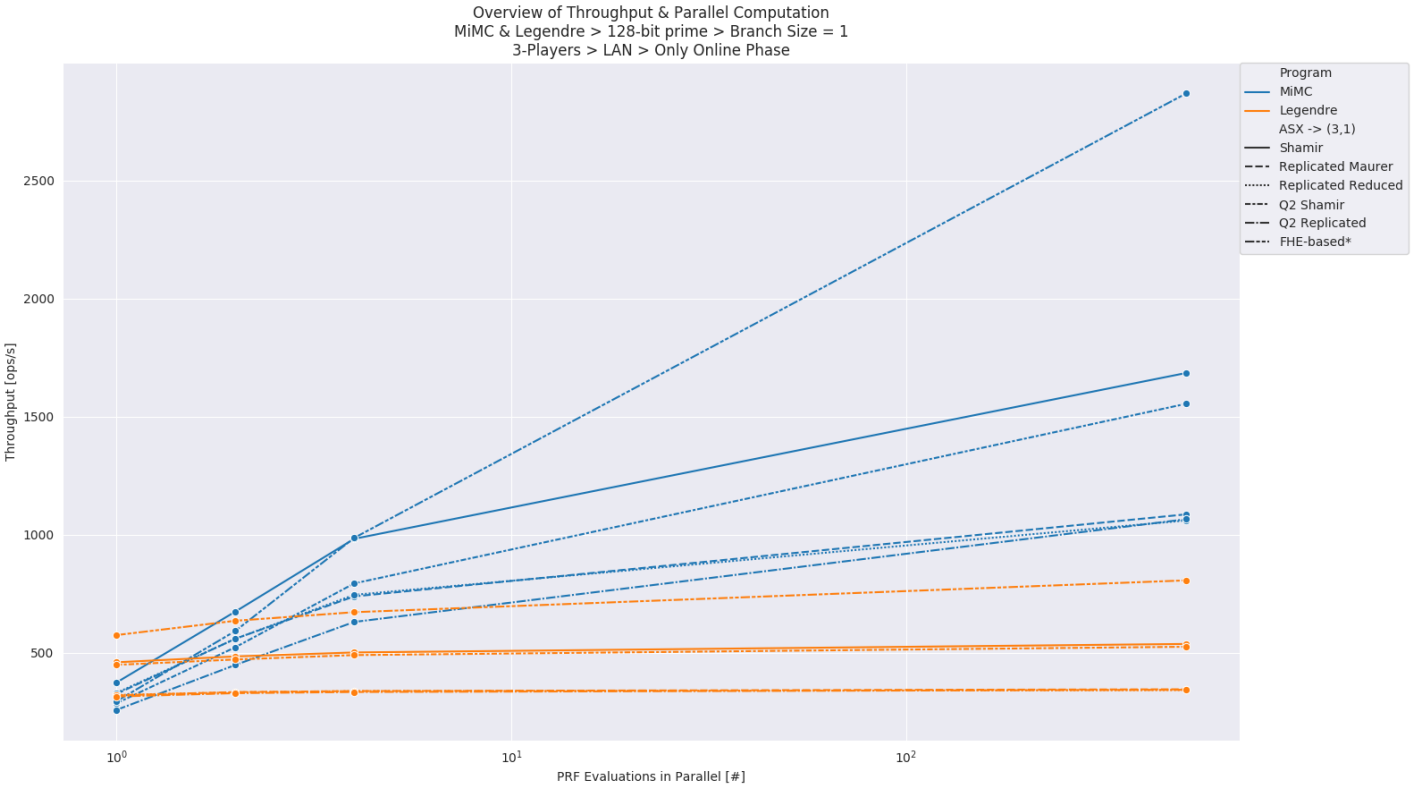


Figure 4.14: The throughput of the 3-players-LAN use case for MiMC and Leg using a branch size of one with only the online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

- Latency: Figure 4.51
- Throughput: Figure 4.52

* Dishonest-majority ASX
  - Latency: Figure 4.53
  - Throughput: Figure 4.54

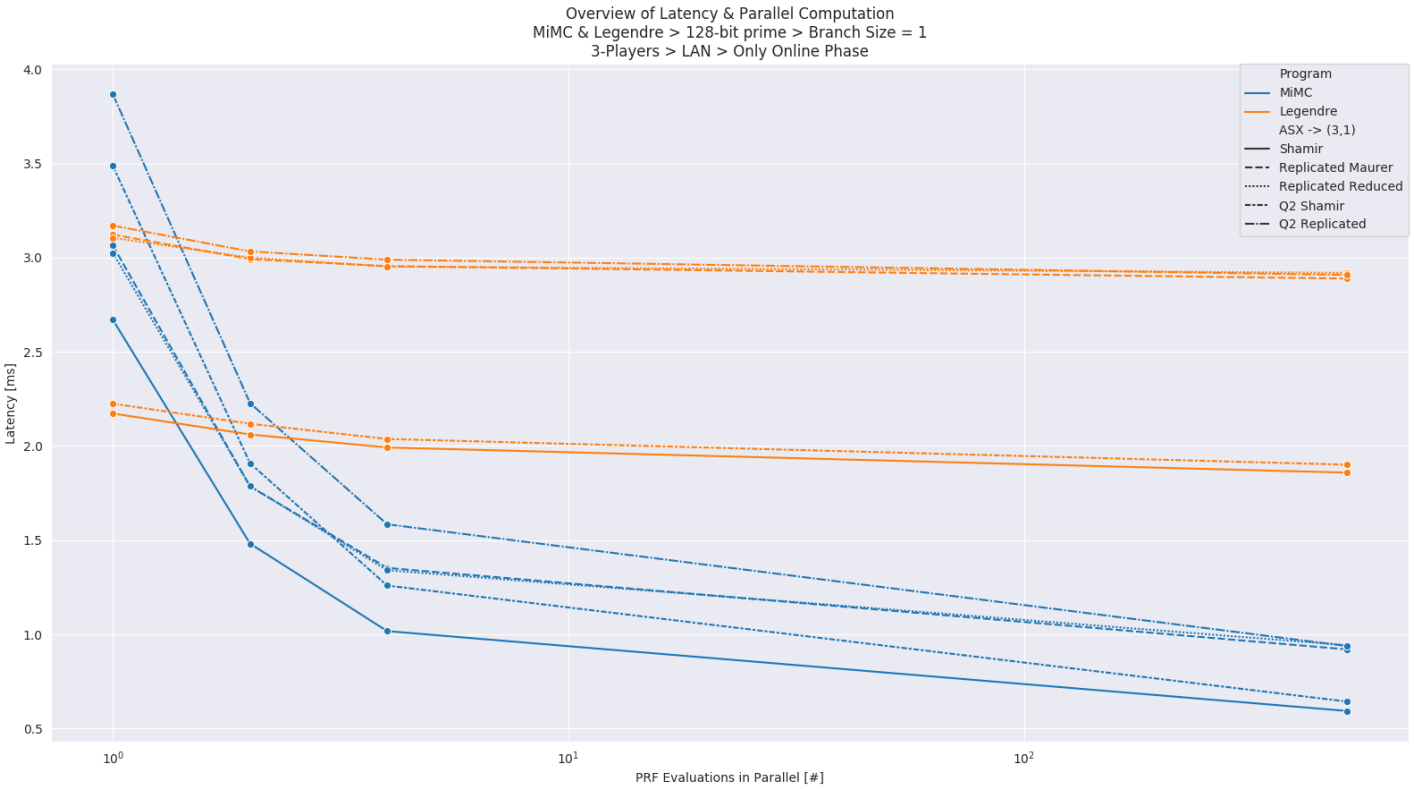


Figure 4.15: The latency of the 3-players-LAN use case for MiMC and Leg using a branch size of one with only the online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

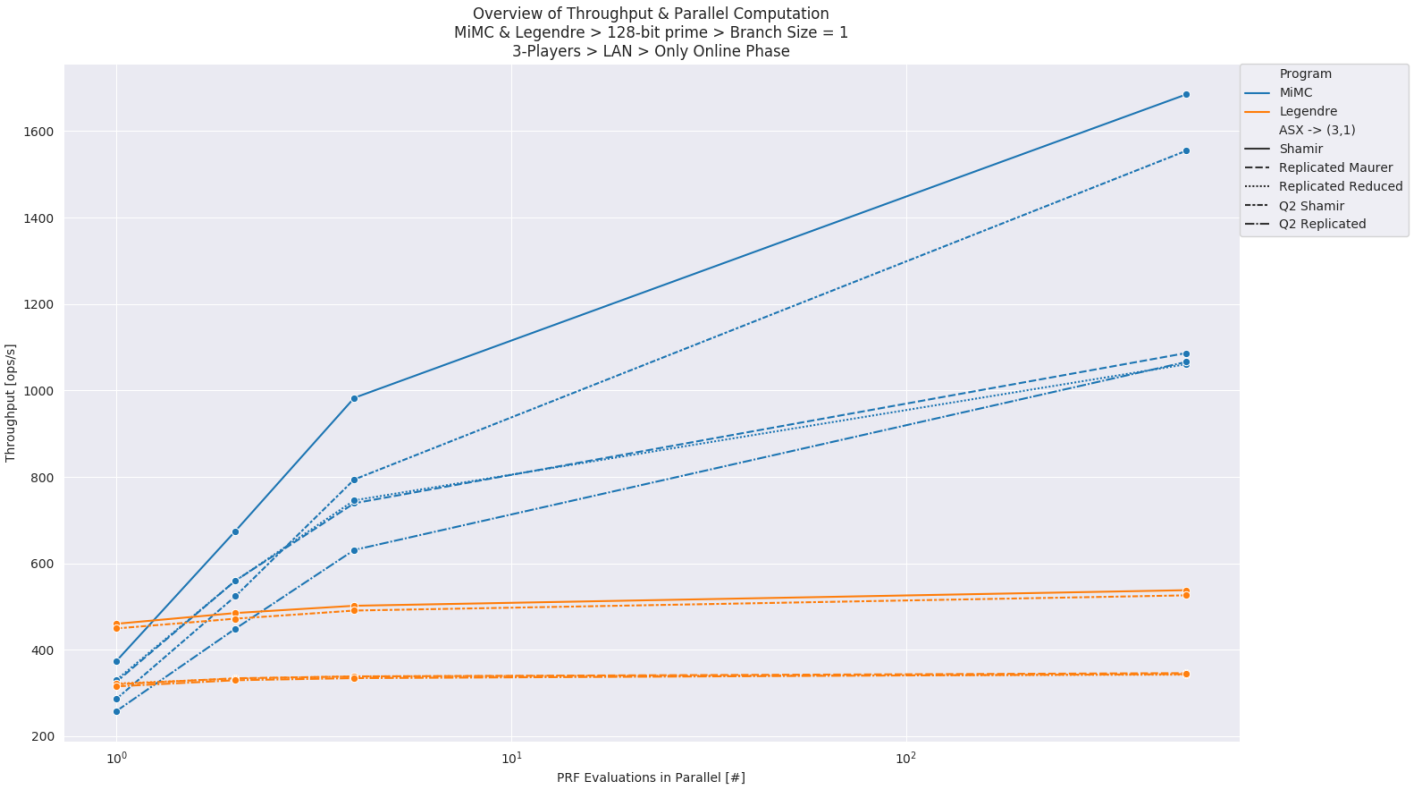


Figure 4.16: The throughput of the 3-players-LAN use case for MiMC and Leg using a branch size of one with only the online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

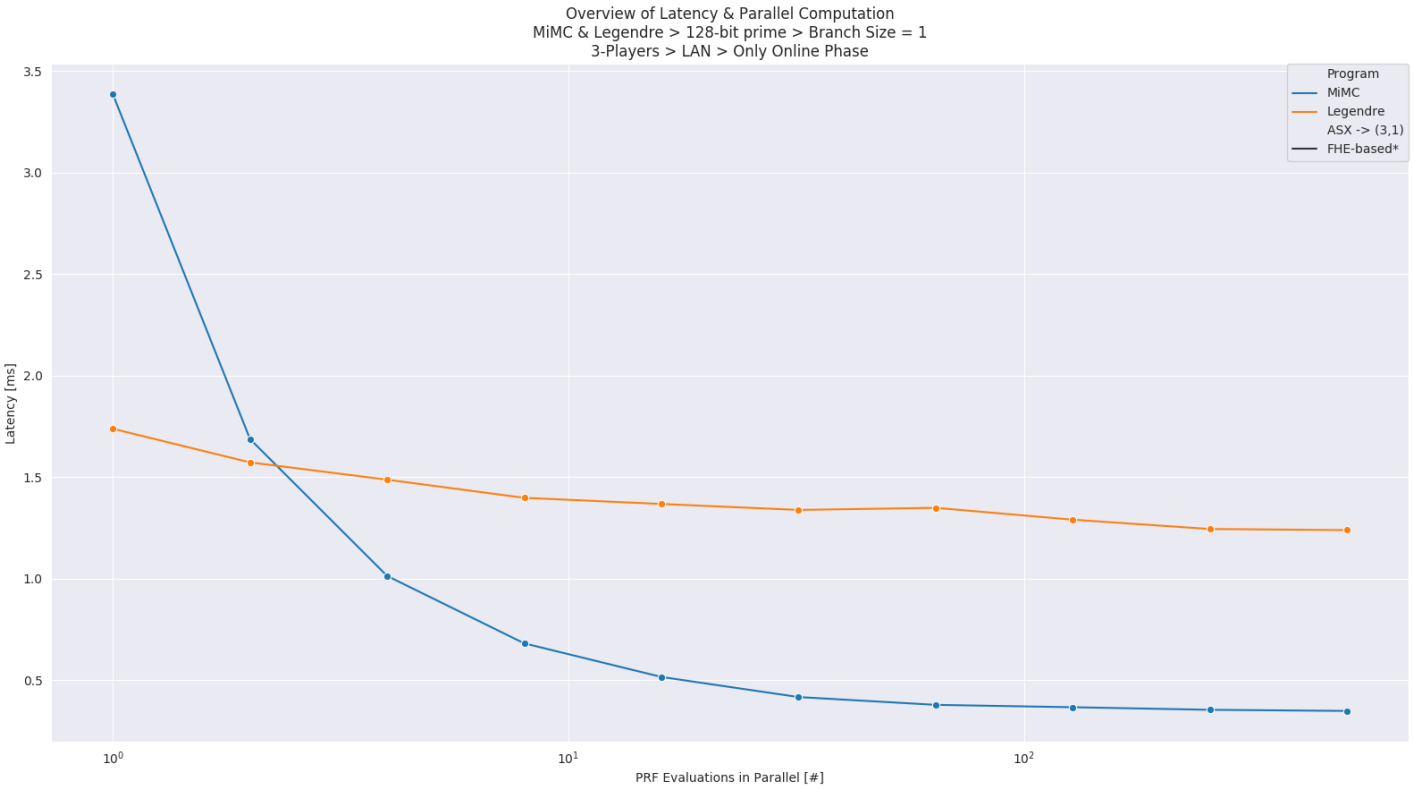


Figure 4.17: The latency of the 3-players-LAN use case for MiMC and Leg using a branch size of one with only the online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4,8,16,32,64,128,256, and 512.

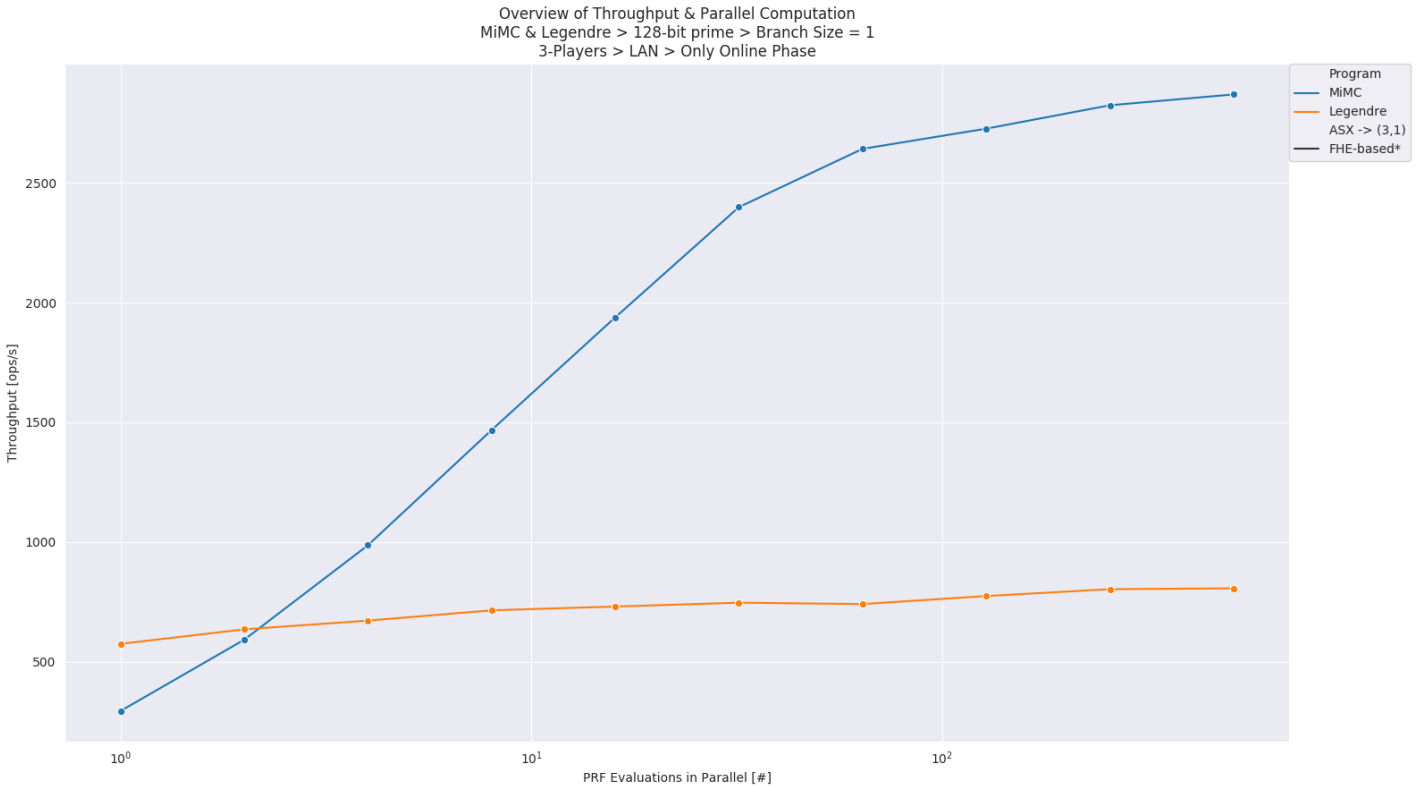


Figure 4.18: The throughput of the 3-players-LAN use case for MiMC and Leg using a branch size of one with only the online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4,8,16,32,64,128,256, and 512.

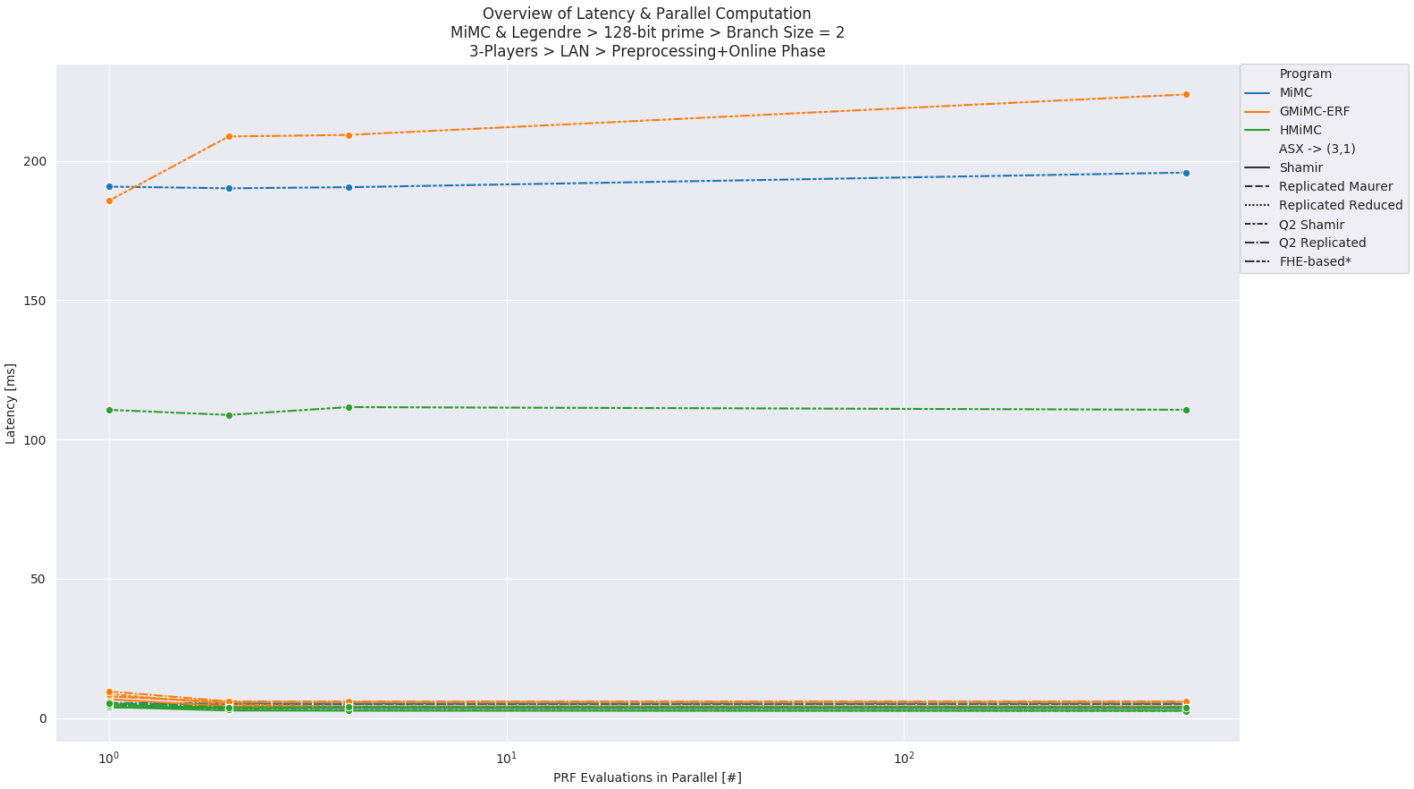


Figure 4.19: The latency of the 3-players-LAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with the preprocessing and online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

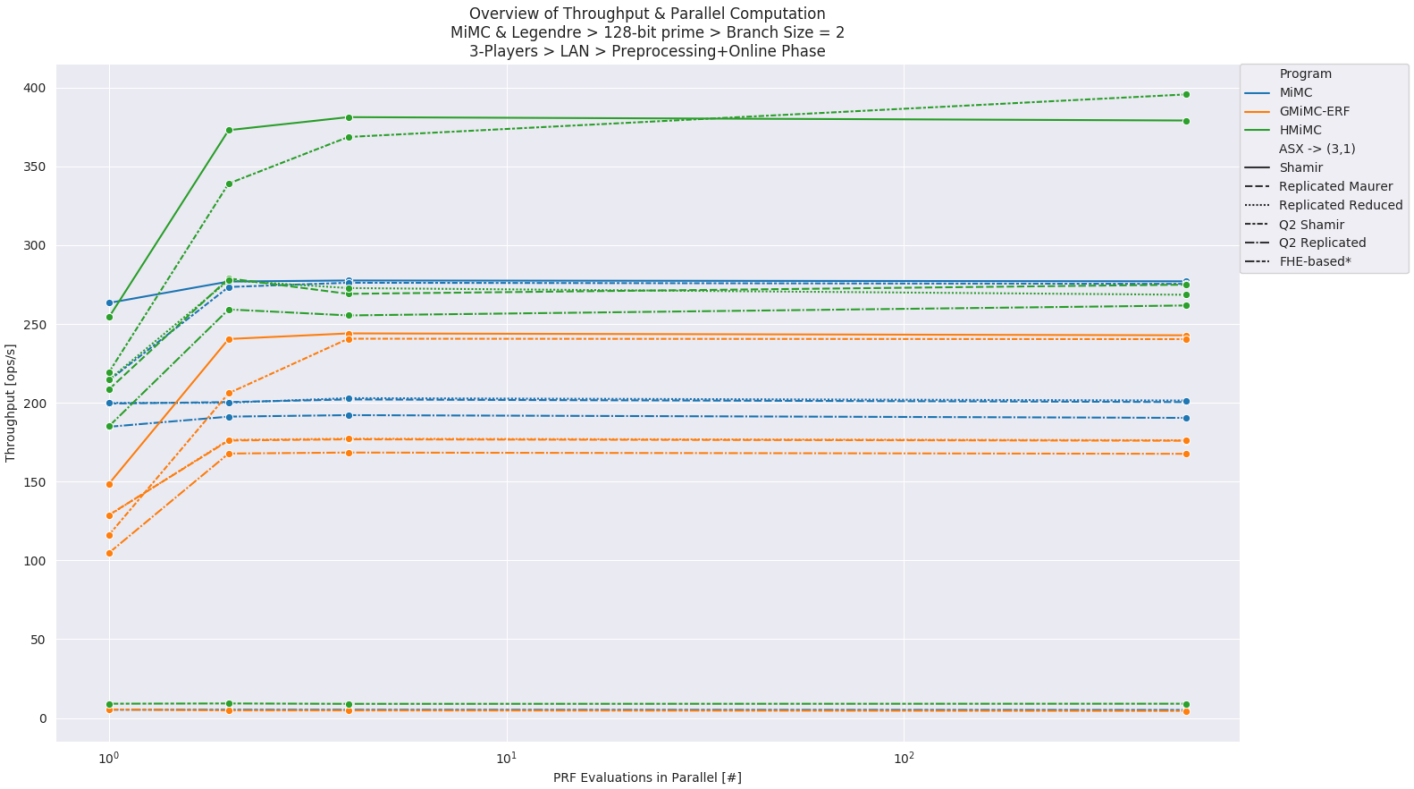


Figure 4.20: The throughput of the 3-players-LAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with the preprocessing and online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

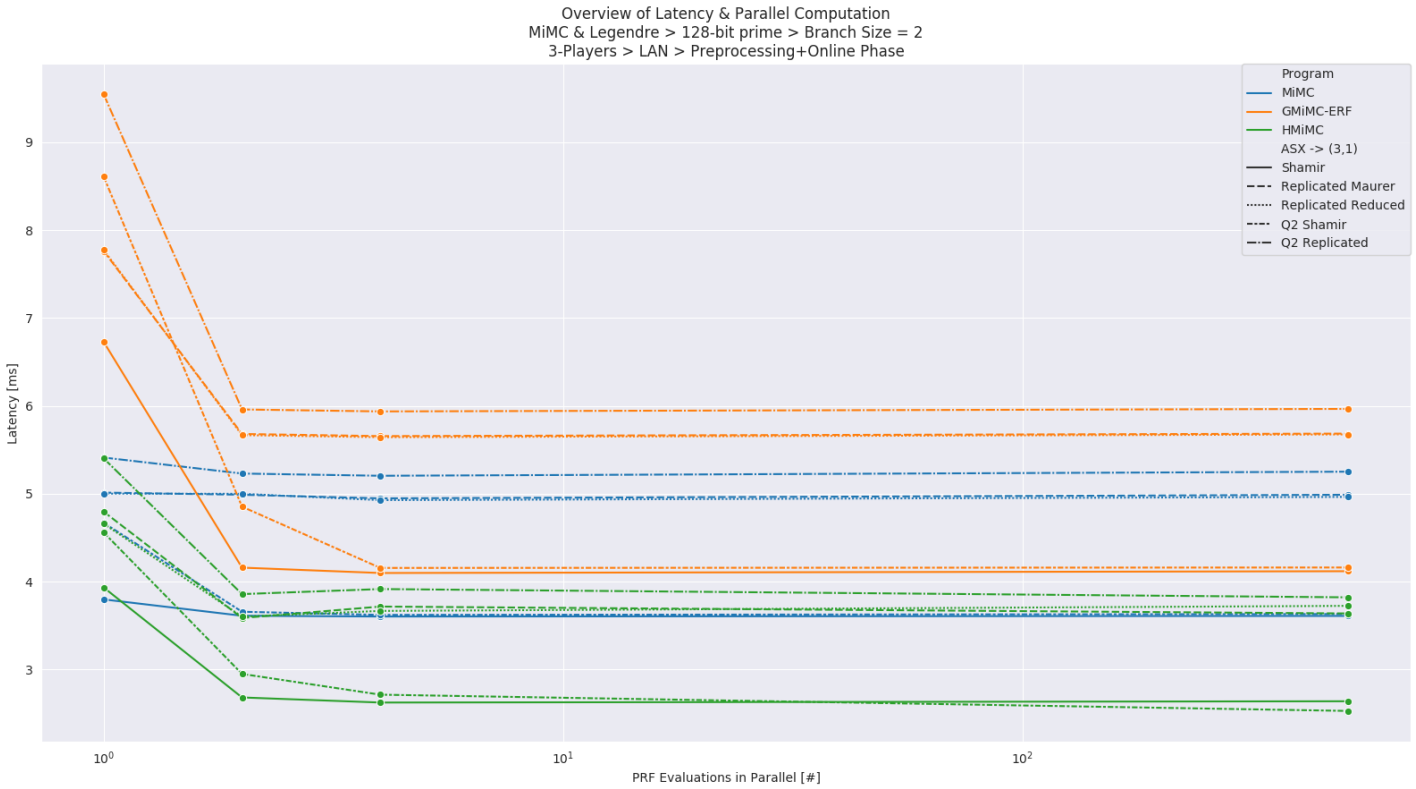


Figure 4.21: The latency of the 3-players-LAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with the preprocessing and online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

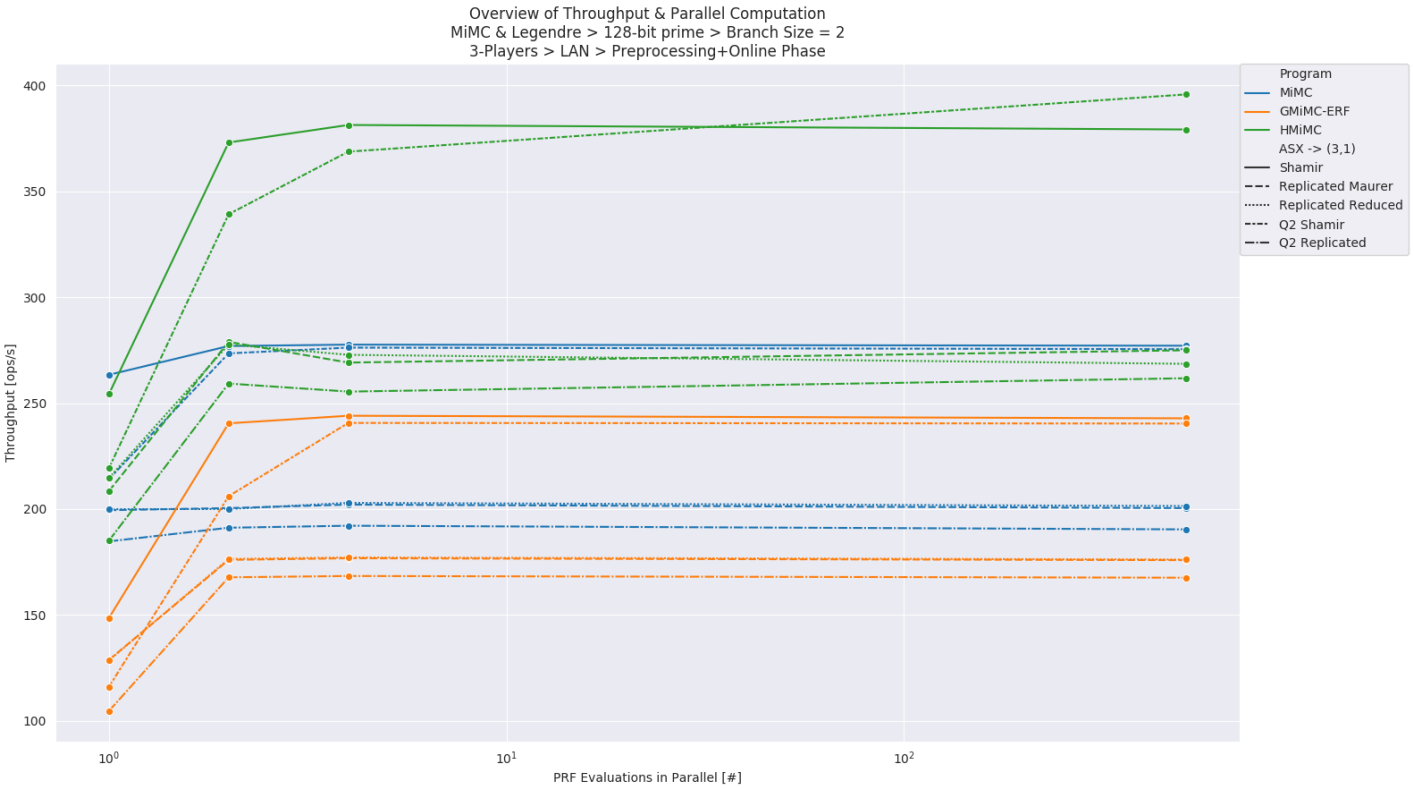


Figure 4.22: The throughput of the 3-players-LAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with the preprocessing and online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

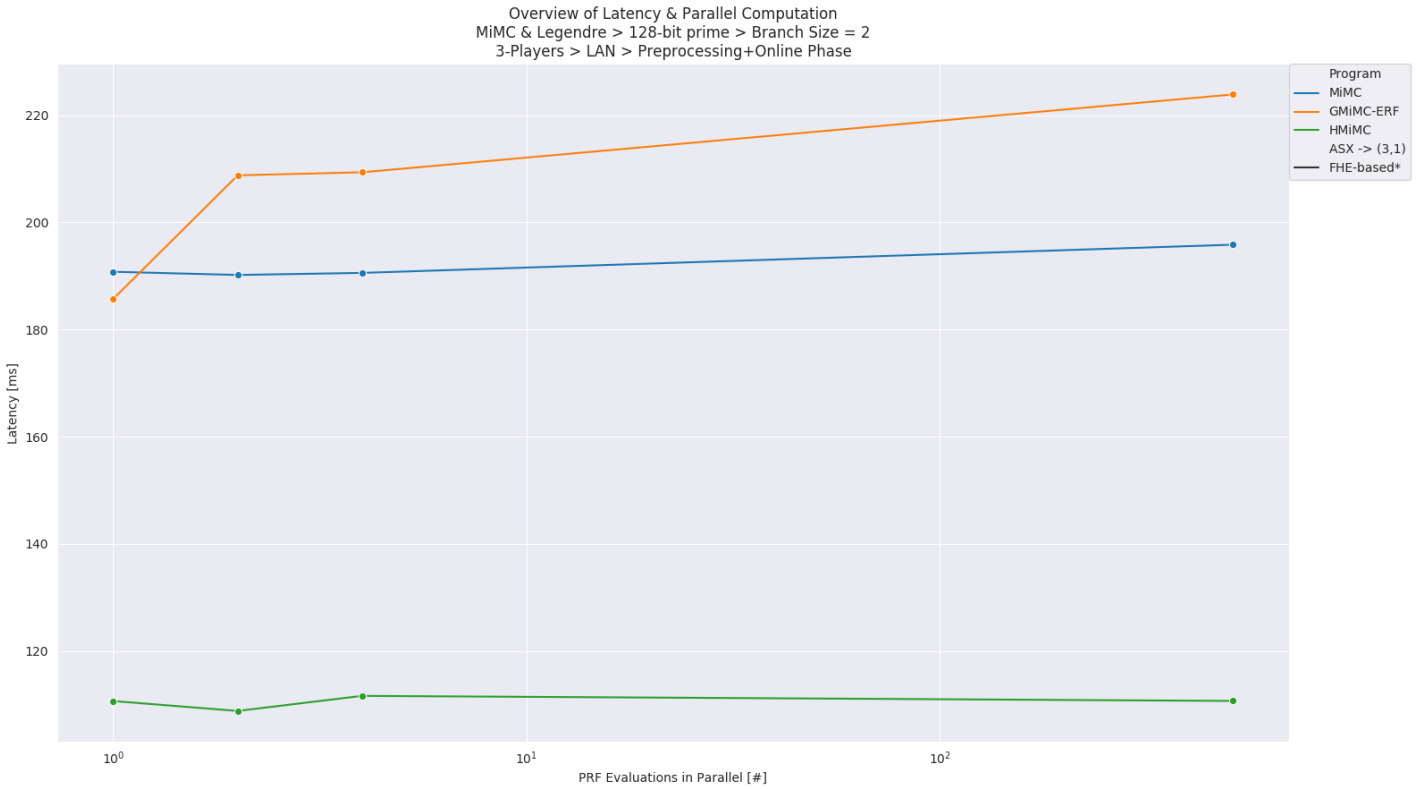


Figure 4.23: The latency of the 3-players-LAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with the preprocessing and online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4, and 512.

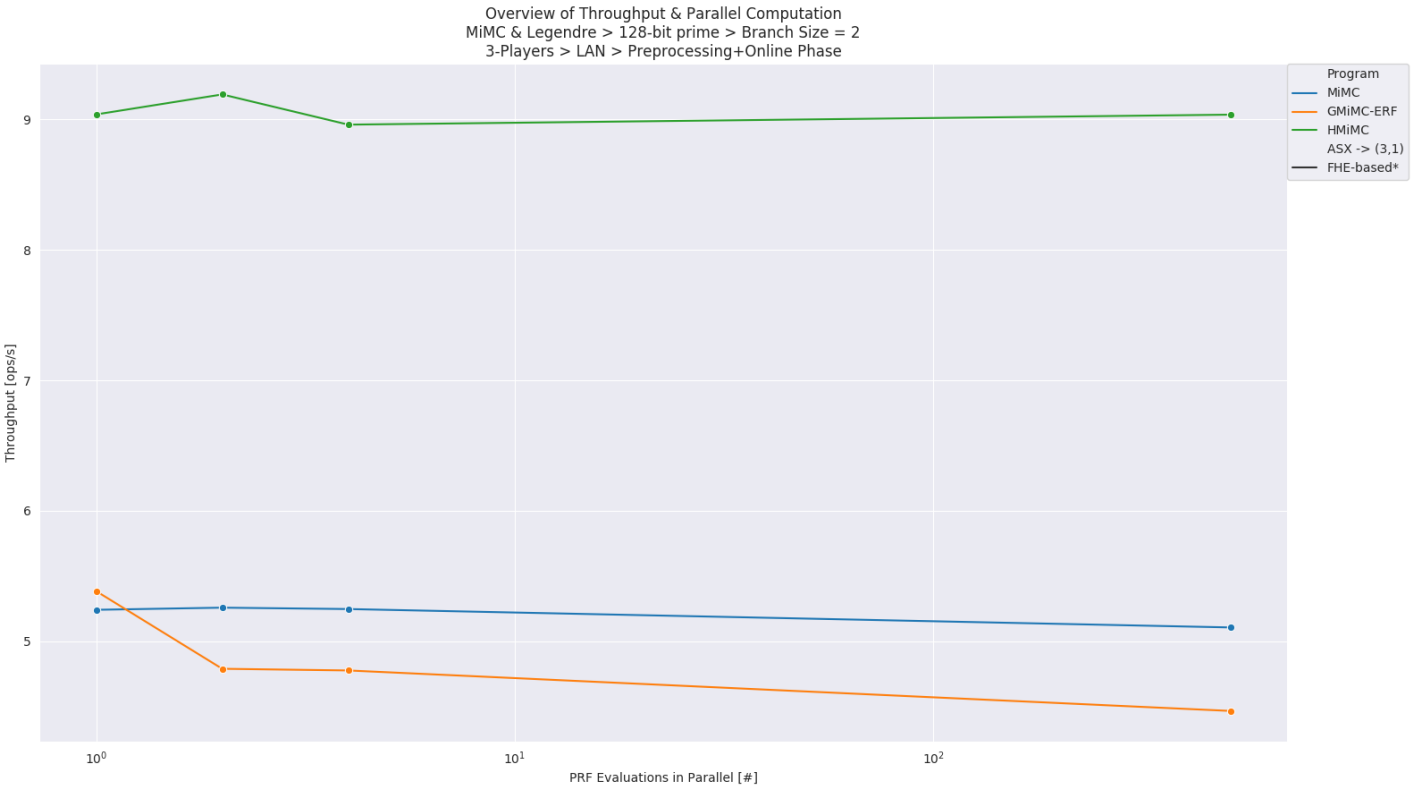


Figure 4.24: The throughput of the 3-players-LAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with the preprocessing and online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4, and 512.

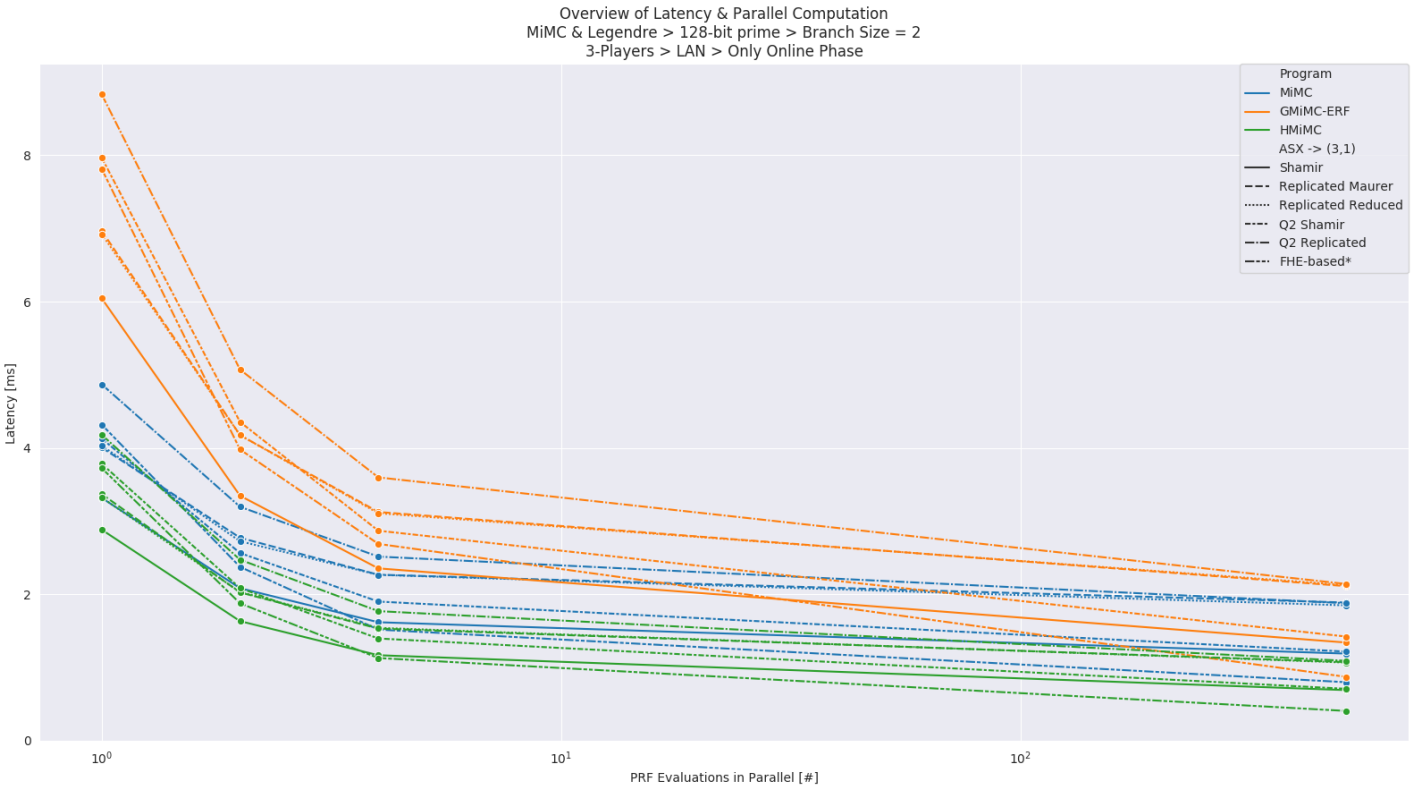


Figure 4.25: The latency of the 3-players-LAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with only the online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

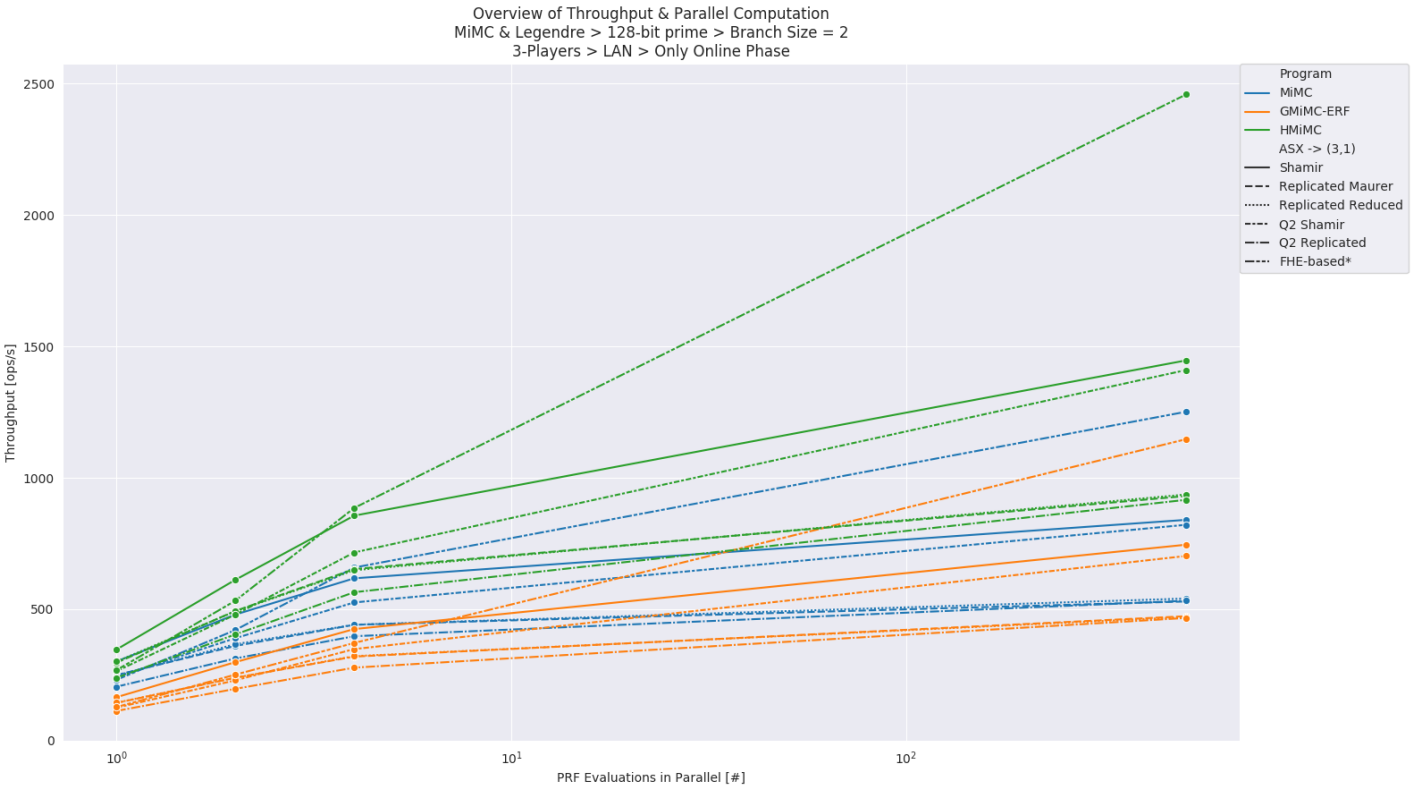


Figure 4.26: The throughput of the 3-players-LAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with only the online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

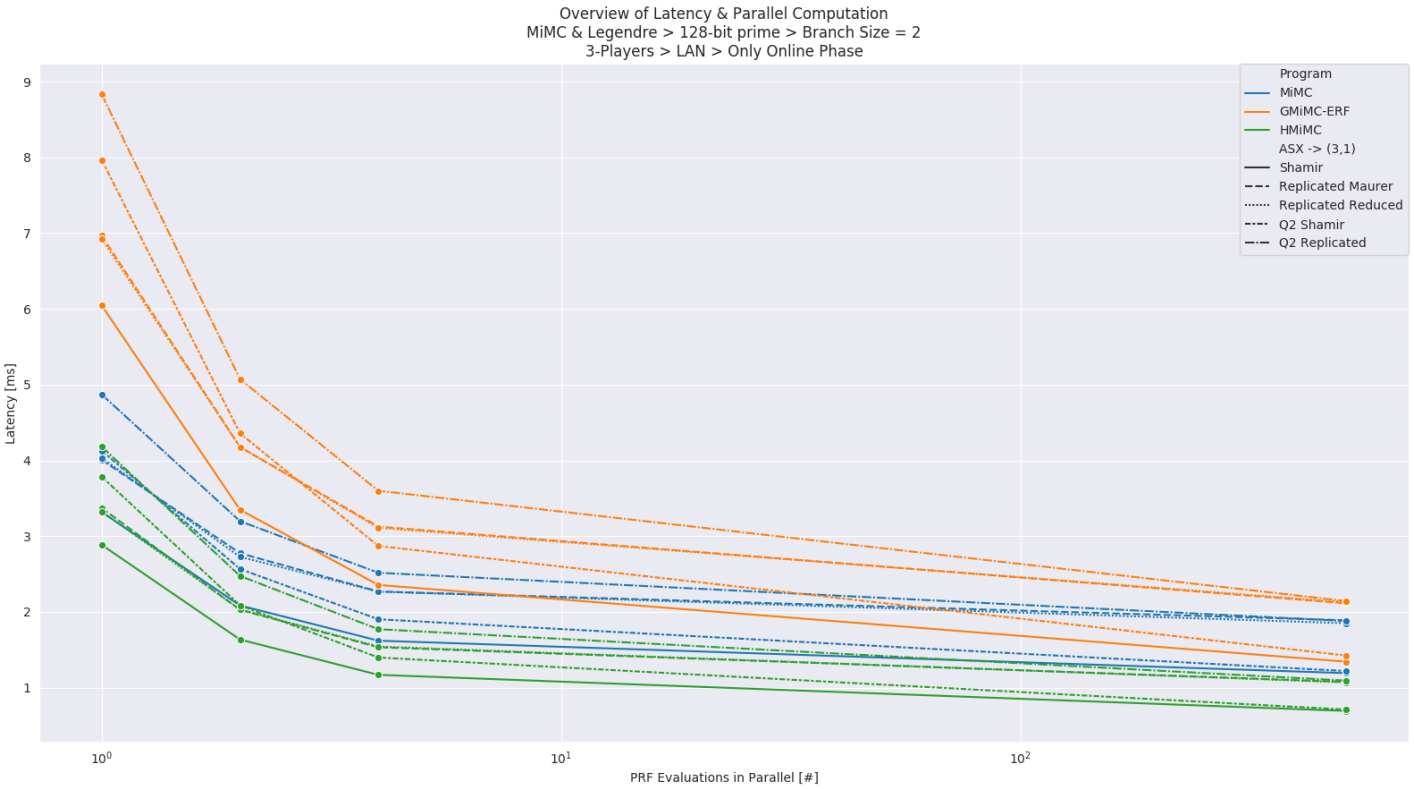


Figure 4.27: The latency of the 3-players-LAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with only the online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

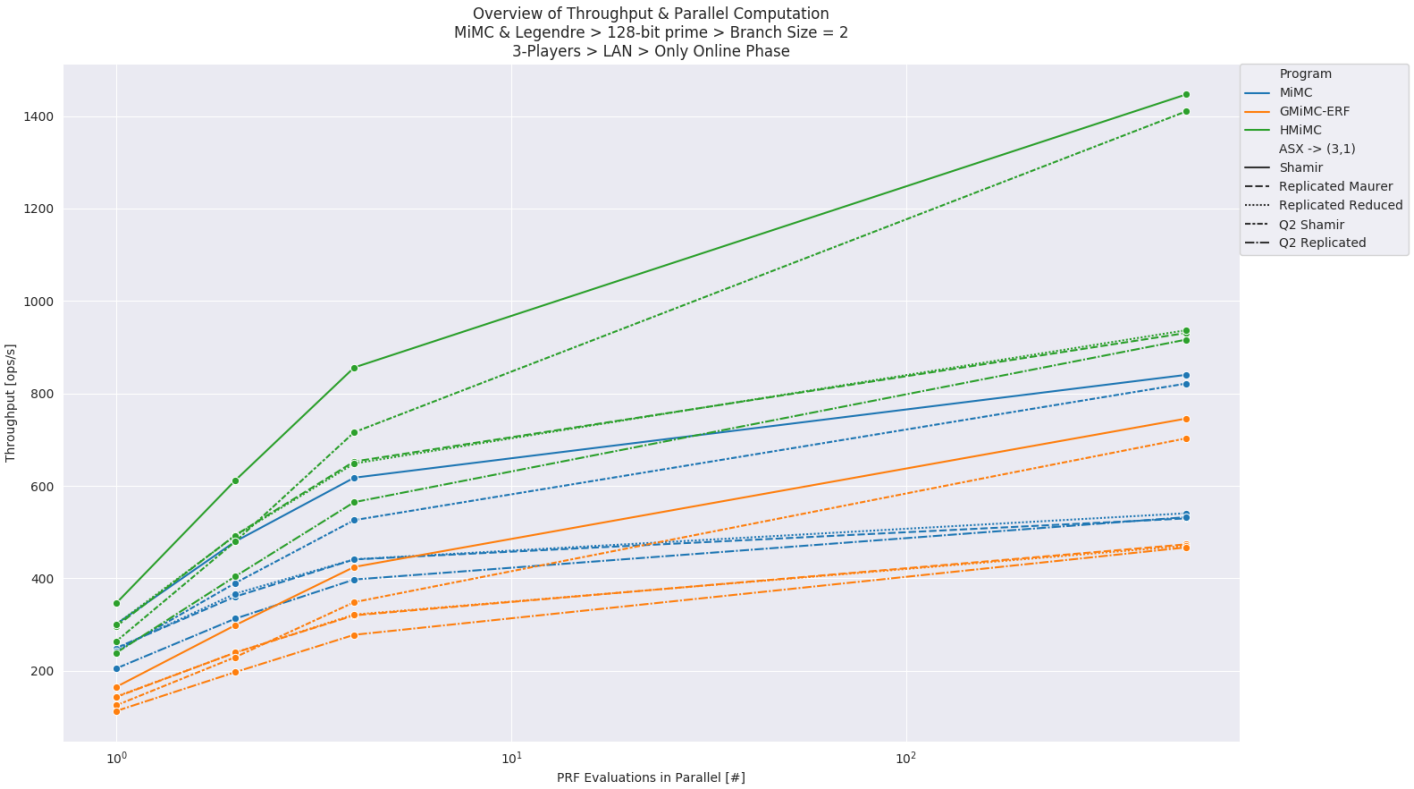


Figure 4.28: The throughput of the 3-players-LAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with only the online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

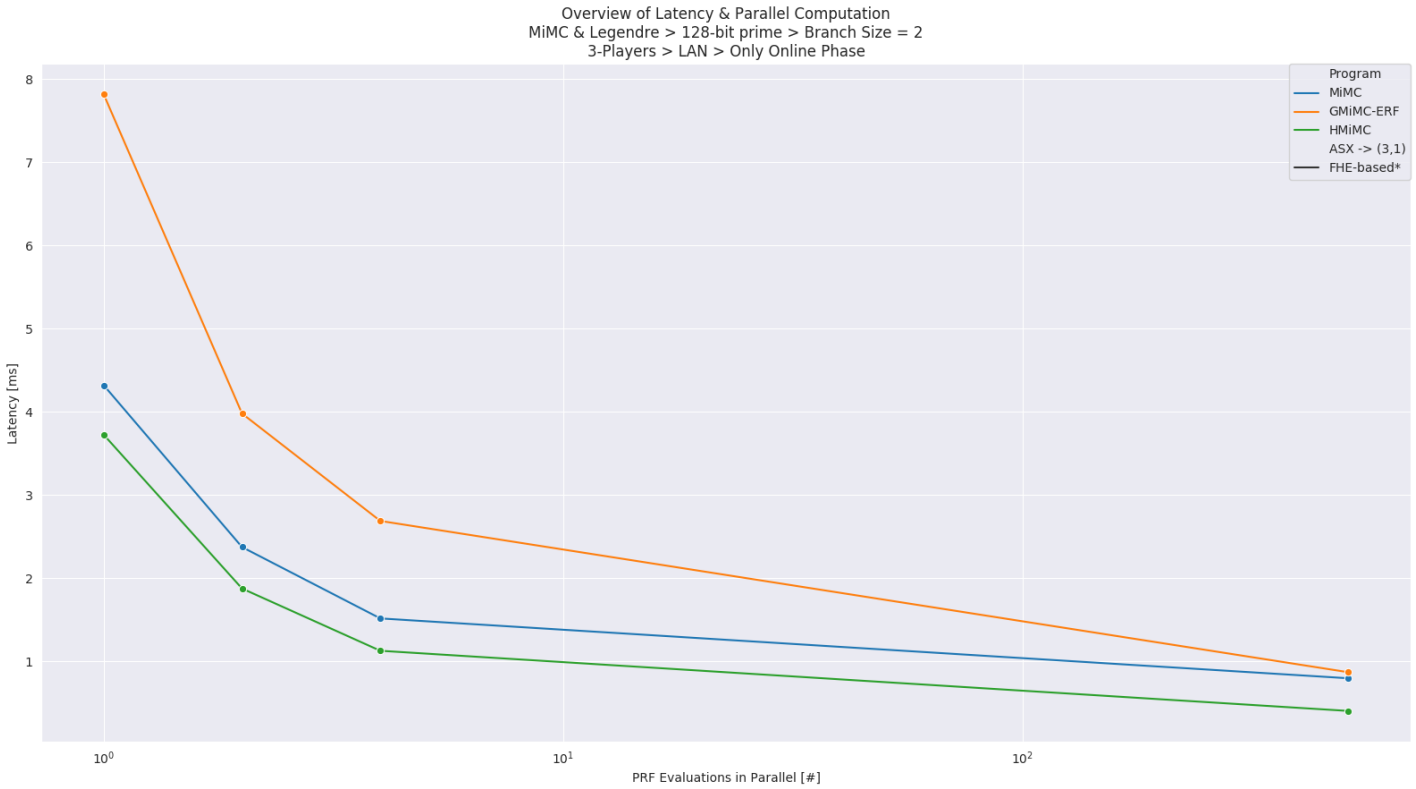


Figure 4.29: The latency of the 3-players-LAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with only the online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4, and 512.

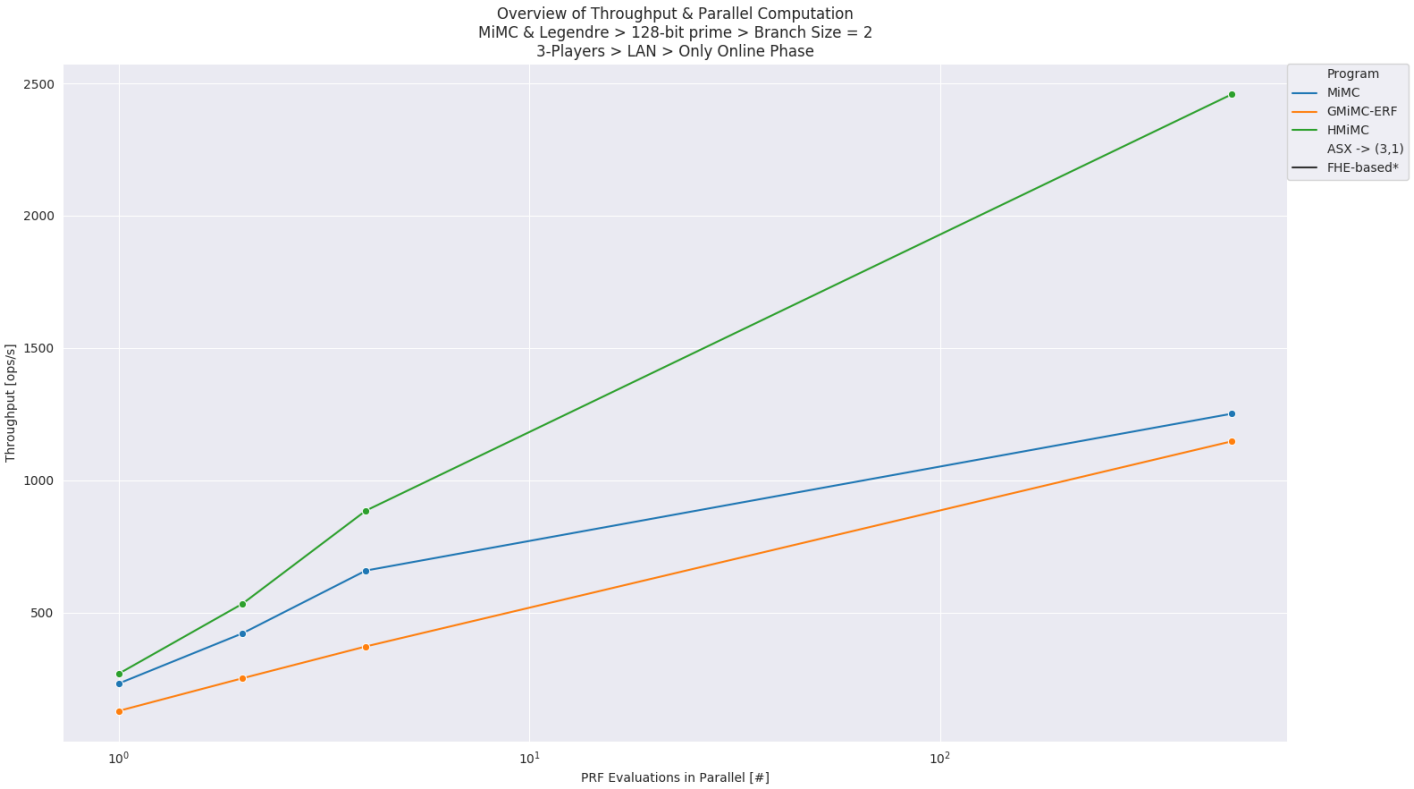


Figure 4.30: The throughput of the 3-players-LAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with only the online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4, and 512.

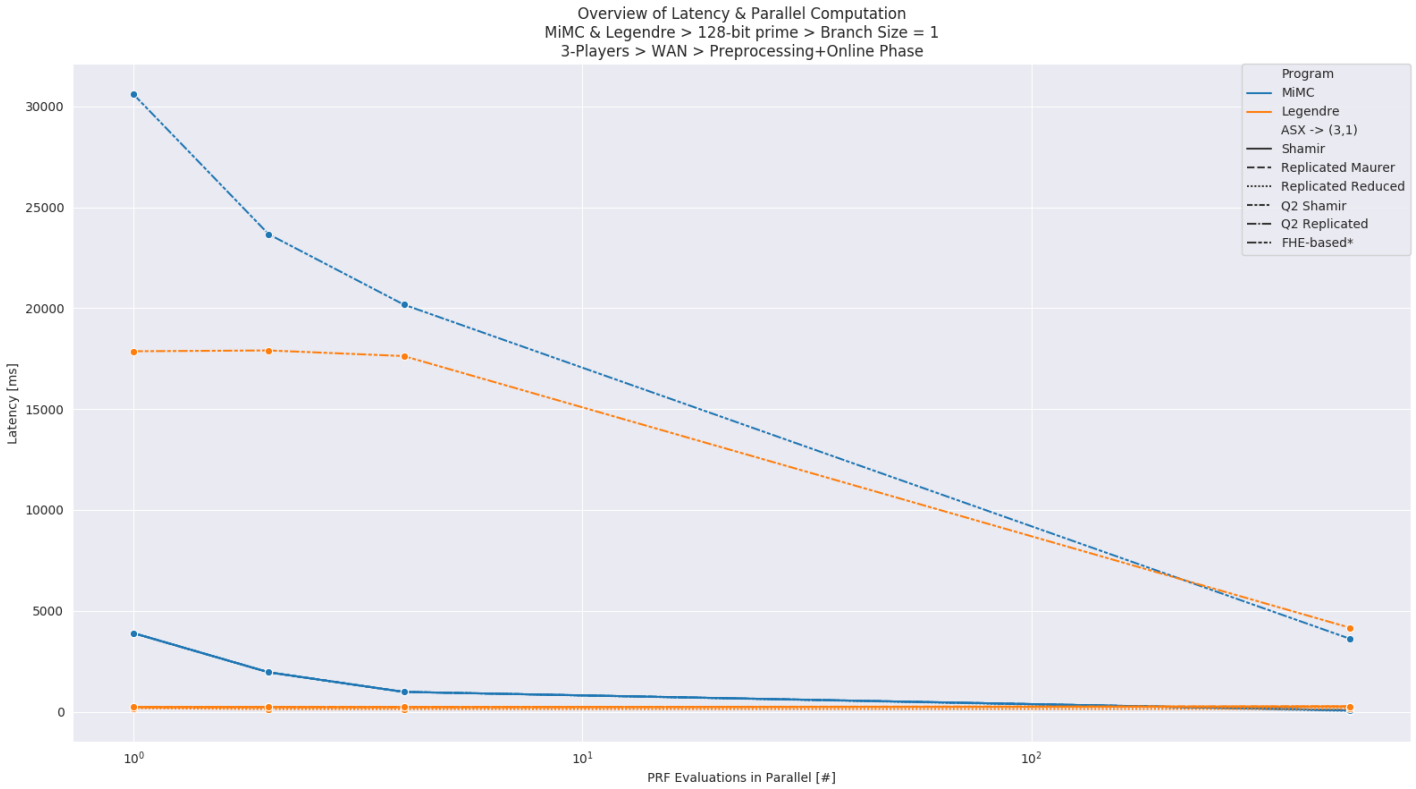


Figure 4.31: The latency of the 3-players-WAN use case for MiMC and Leg using a branch size of one with the preprocessing and online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

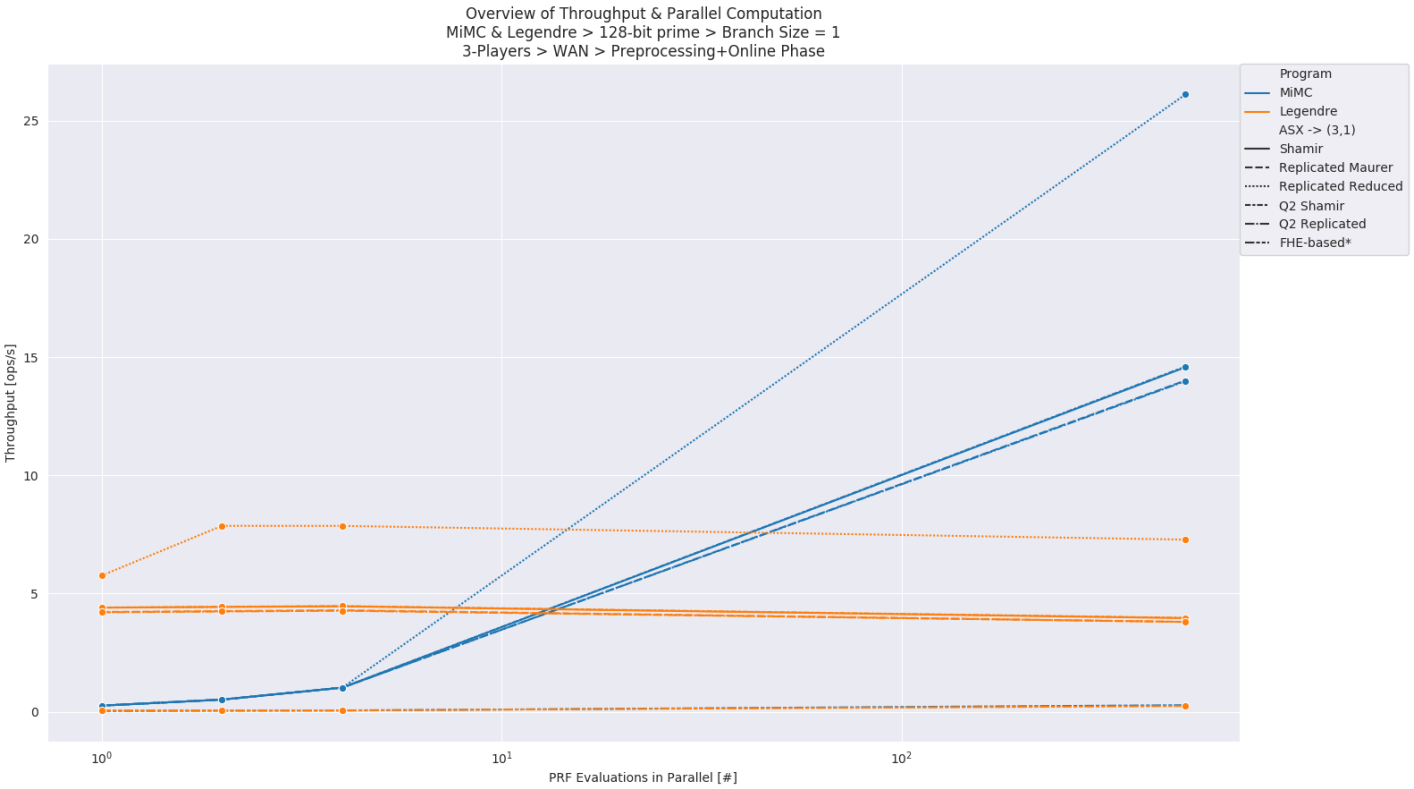


Figure 4.32: The throughput of the 3-players-WAN use case for MiMC and Leg using a branch size of one with the preprocessing and online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

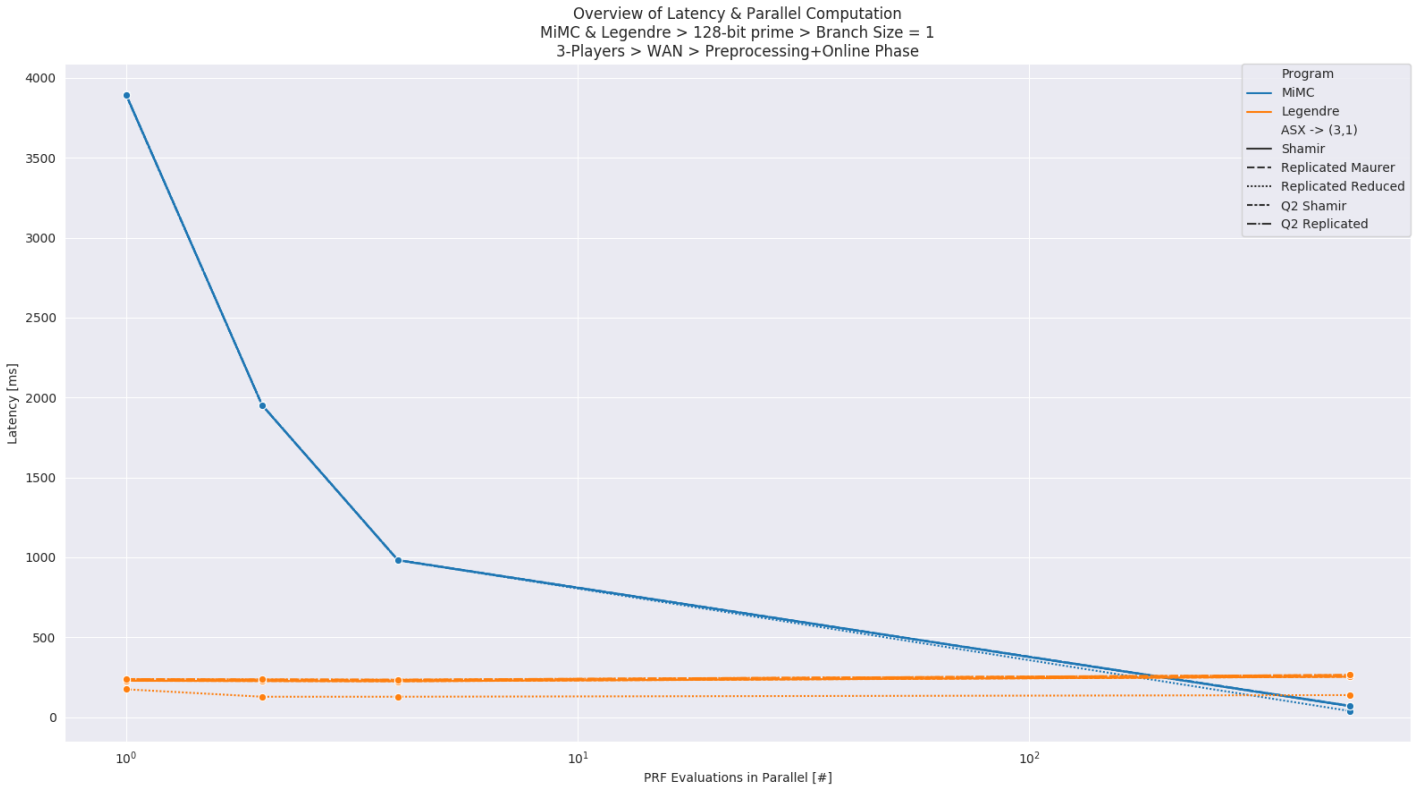


Figure 4.33: The latency of the 3-players-WAN use case for MiMC and Leg using a branch size of one with the preprocessing and online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

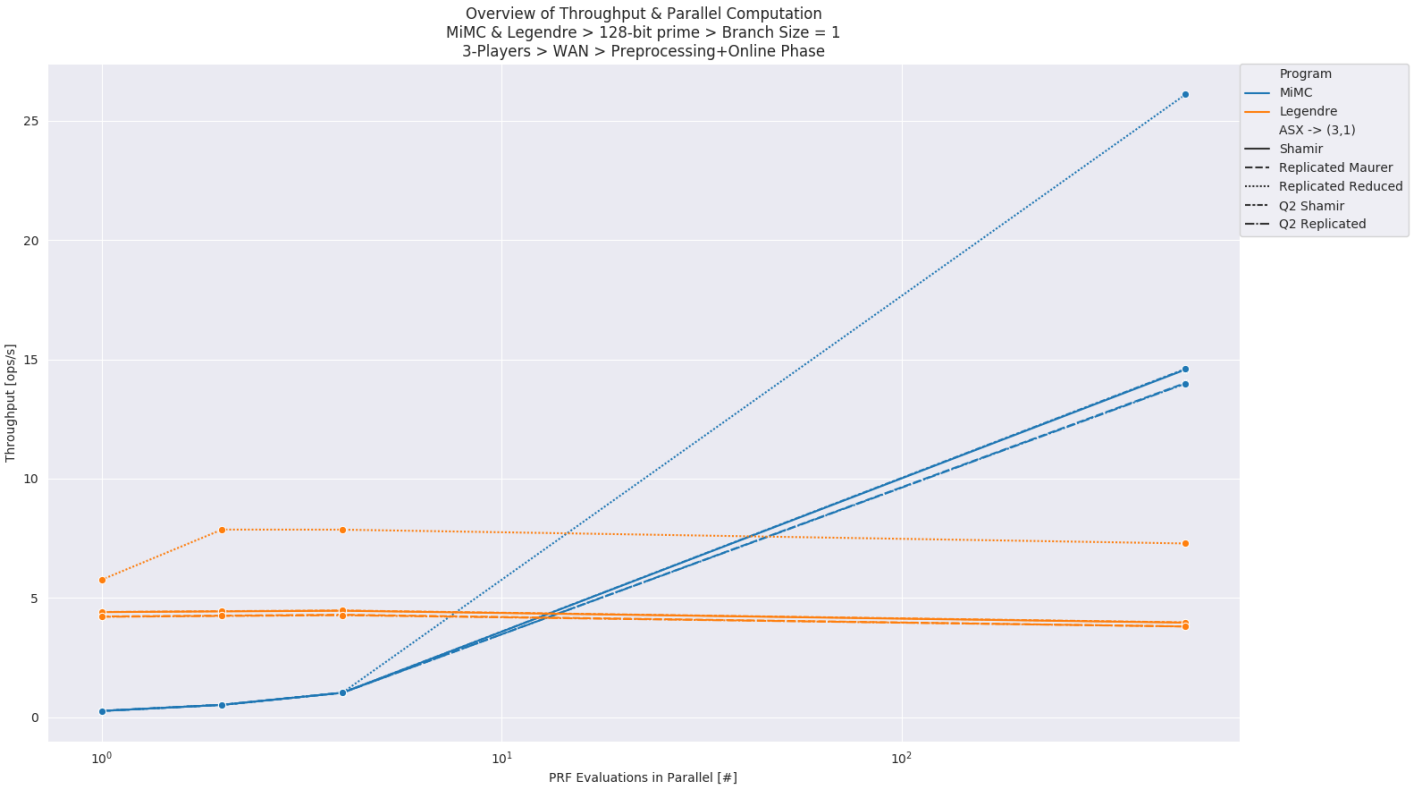


Figure 4.34: The throughput of the 3-players-WAN use case for MiMC and Leg using a branch size of one with the preprocessing and online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

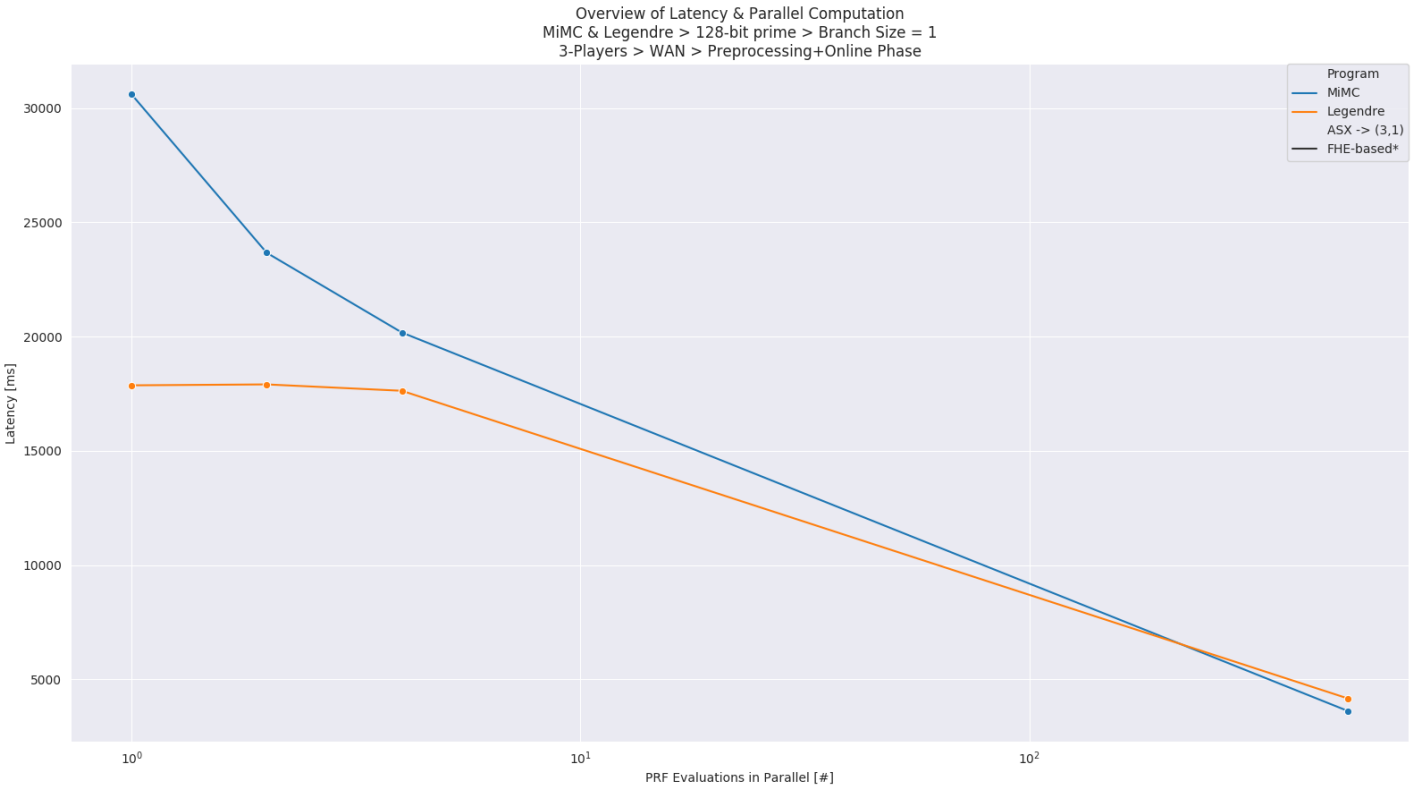


Figure 4.35: The latency of the 3-players-WAN use case for MiMC and Leg using a branch size of one with the preprocessing and online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4, and 512.

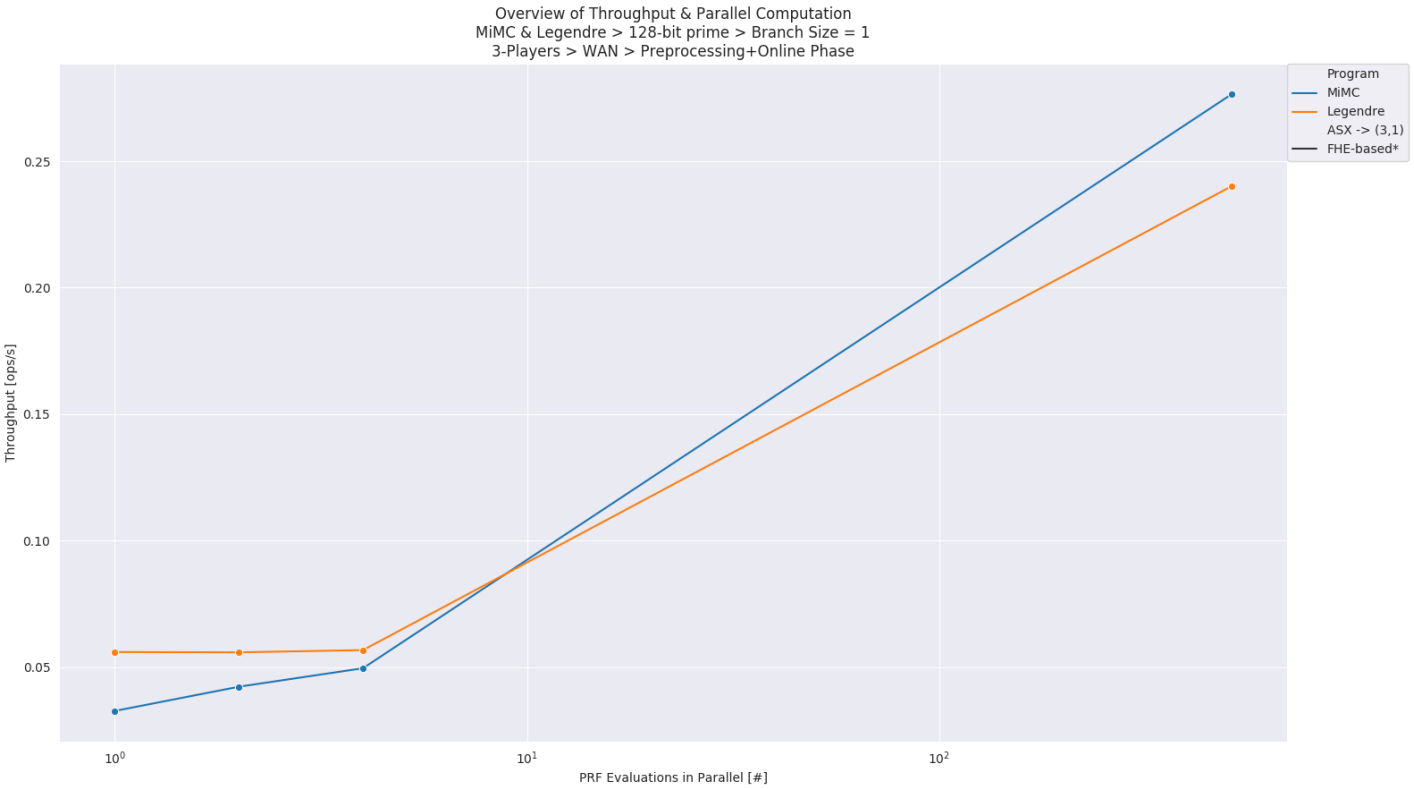


Figure 4.36: The throughput of the 3-players-WAN use case for MiMC and Leg using a branch size of one with the preprocessing and online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4, and 512.

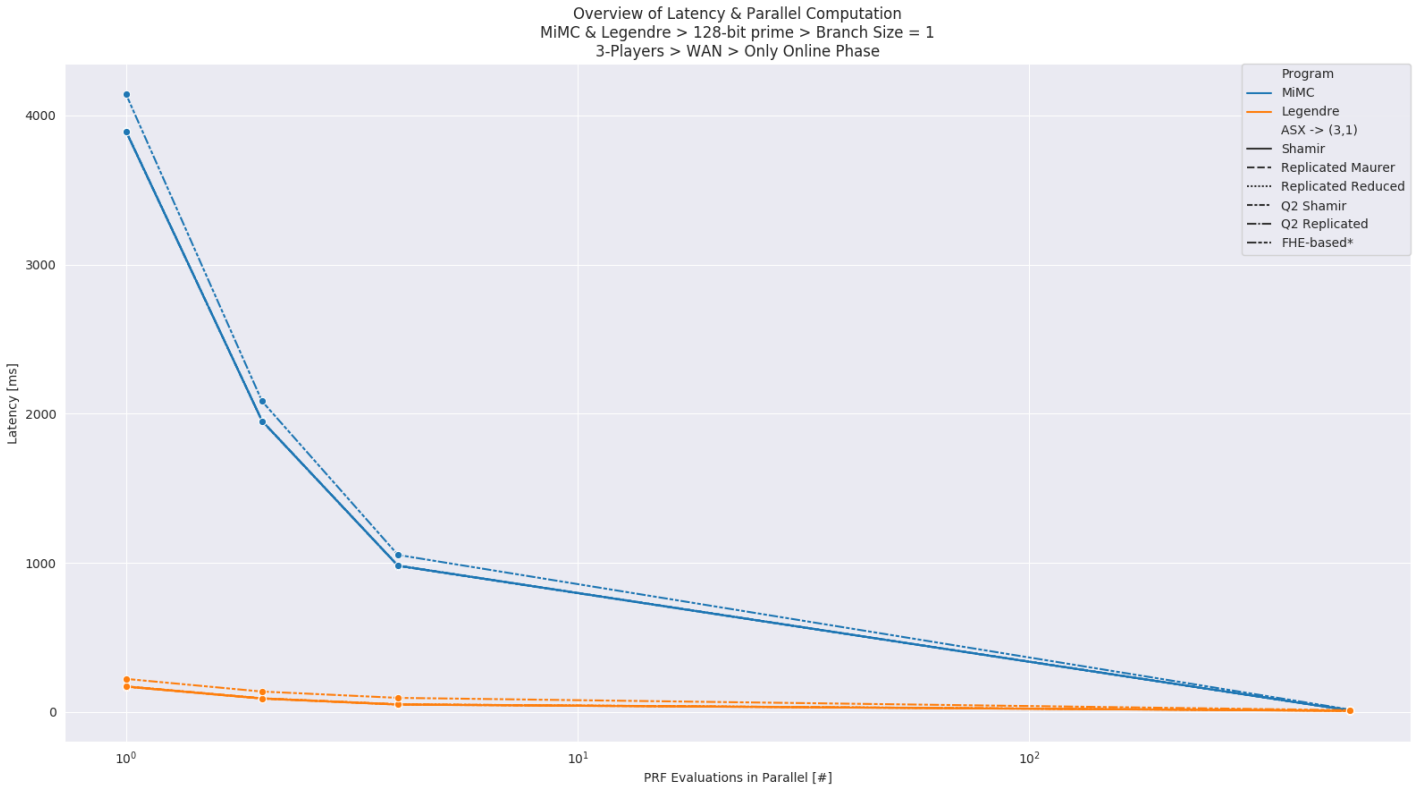


Figure 4.37: The latency of the 3-players-WAN use case for MiMC and Leg using a branch size of one with only the online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

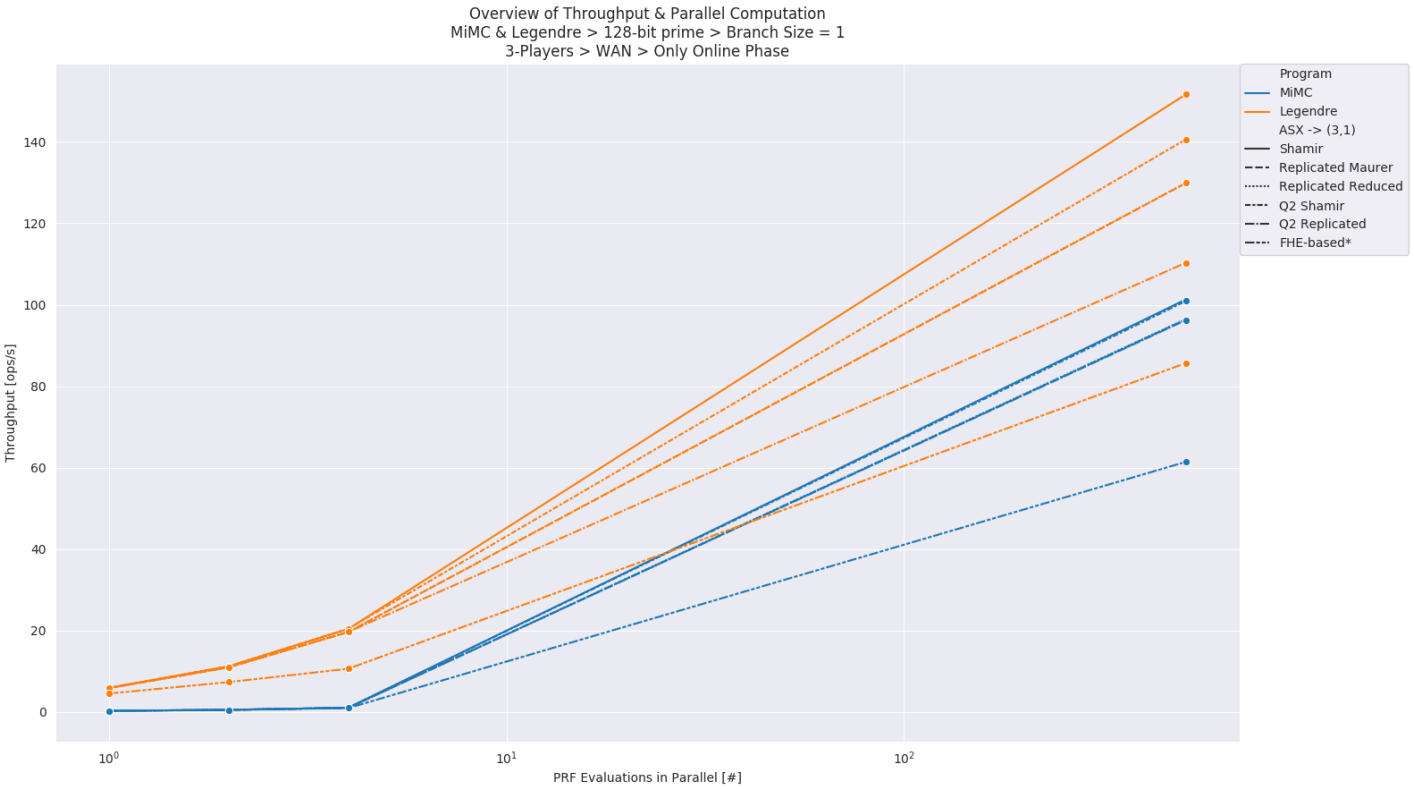


Figure 4.38: The throughput of the 3-players-WAN use case for MiMC and Leg using a branch size of one with only the online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

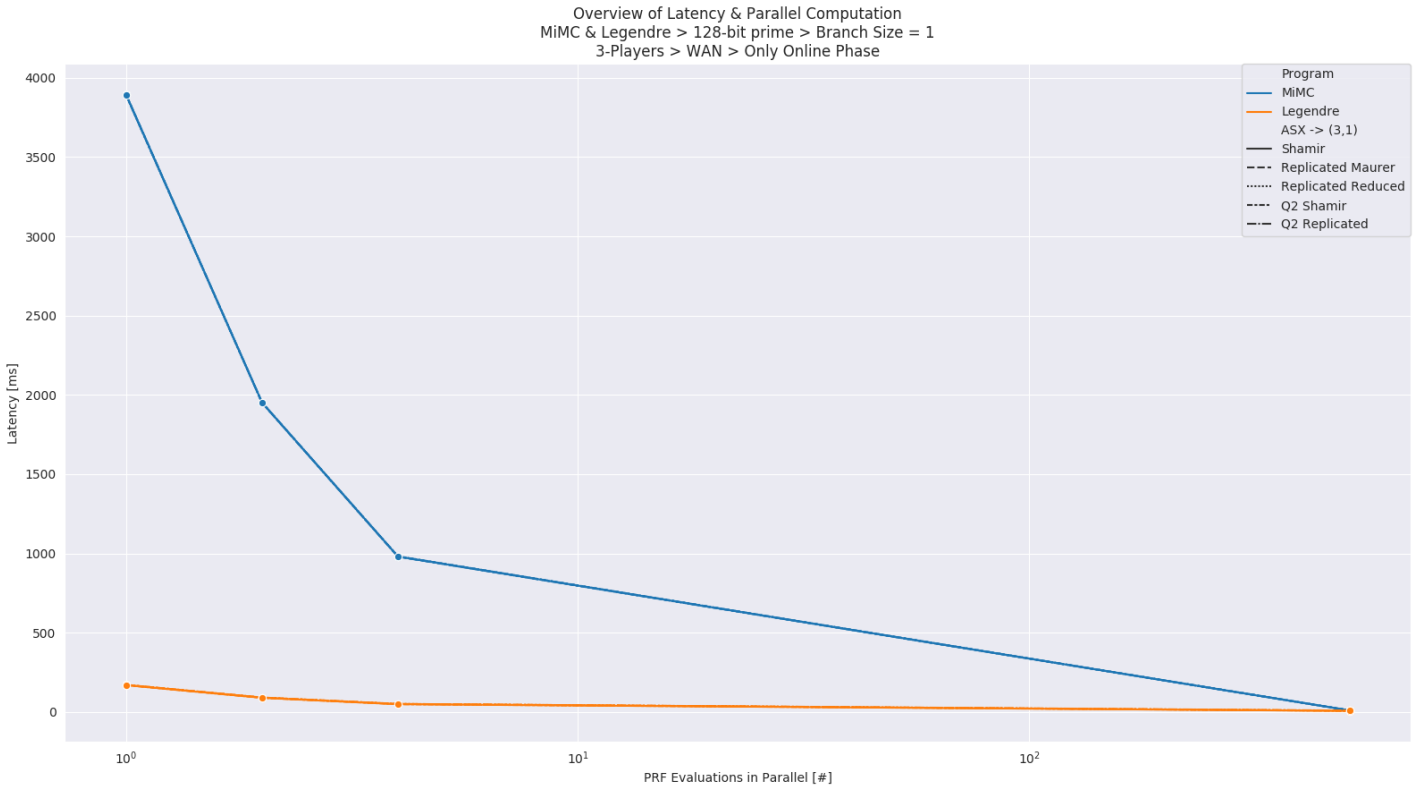


Figure 4.39: The latency of the 3-players-WAN use case for MiMC and Leg using a branch size of one with only the online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

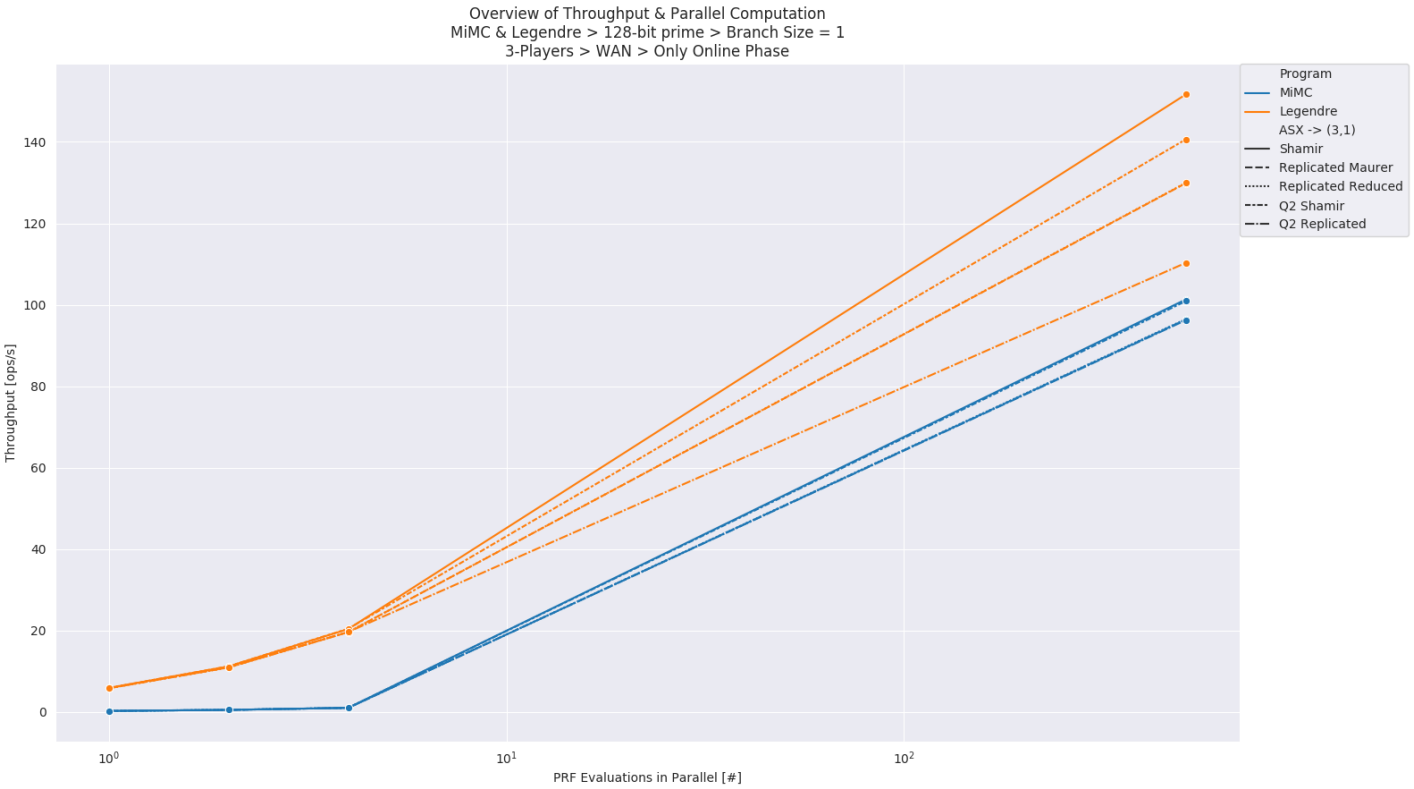


Figure 4.40: The throughput of the 3-players-WAN use case for MiMC and Leg using a branch size of one with only the online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

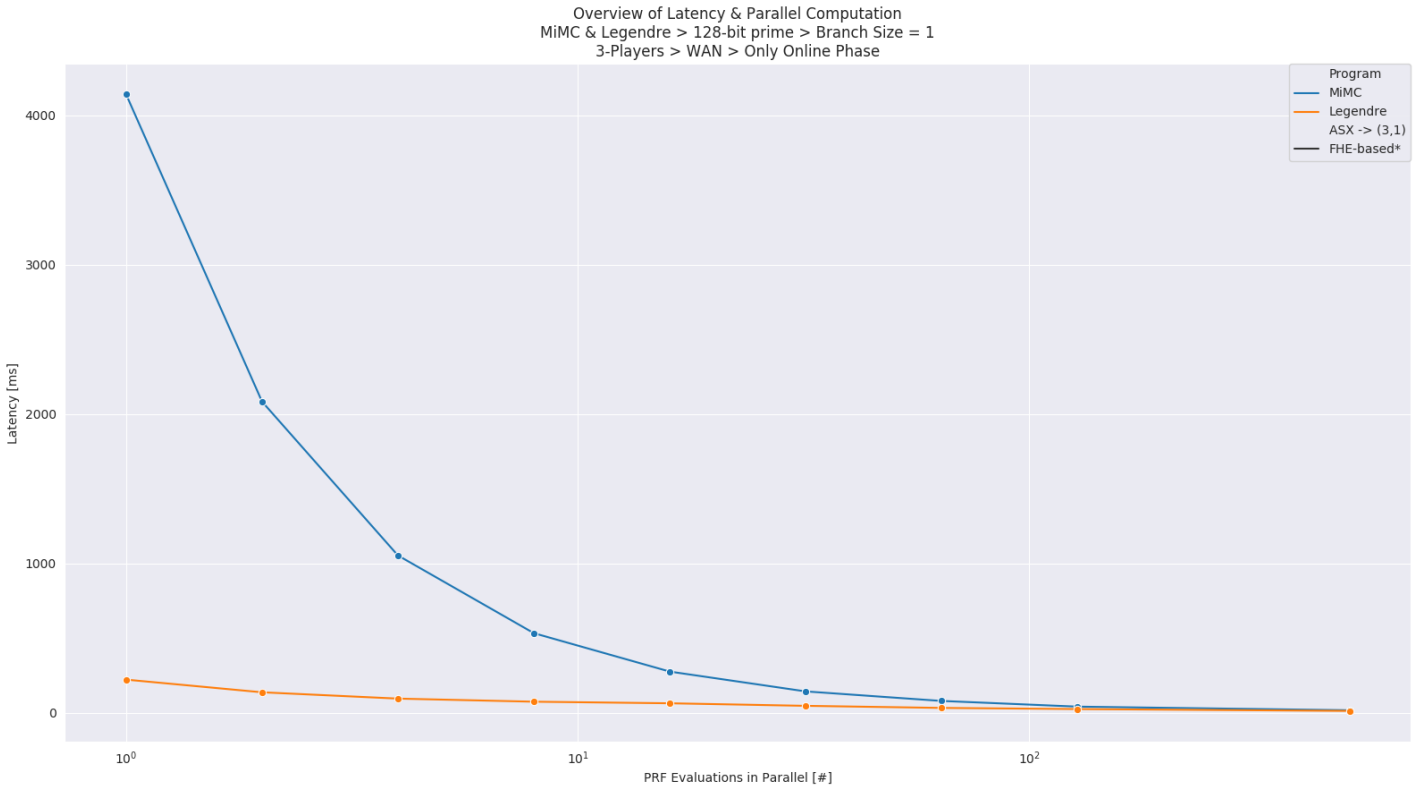


Figure 4.41: The latency of the 3-players-WAN use case for MiMC and Leg using a branch size of one with only the online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4,8,16,32,64,128,256, and 512.

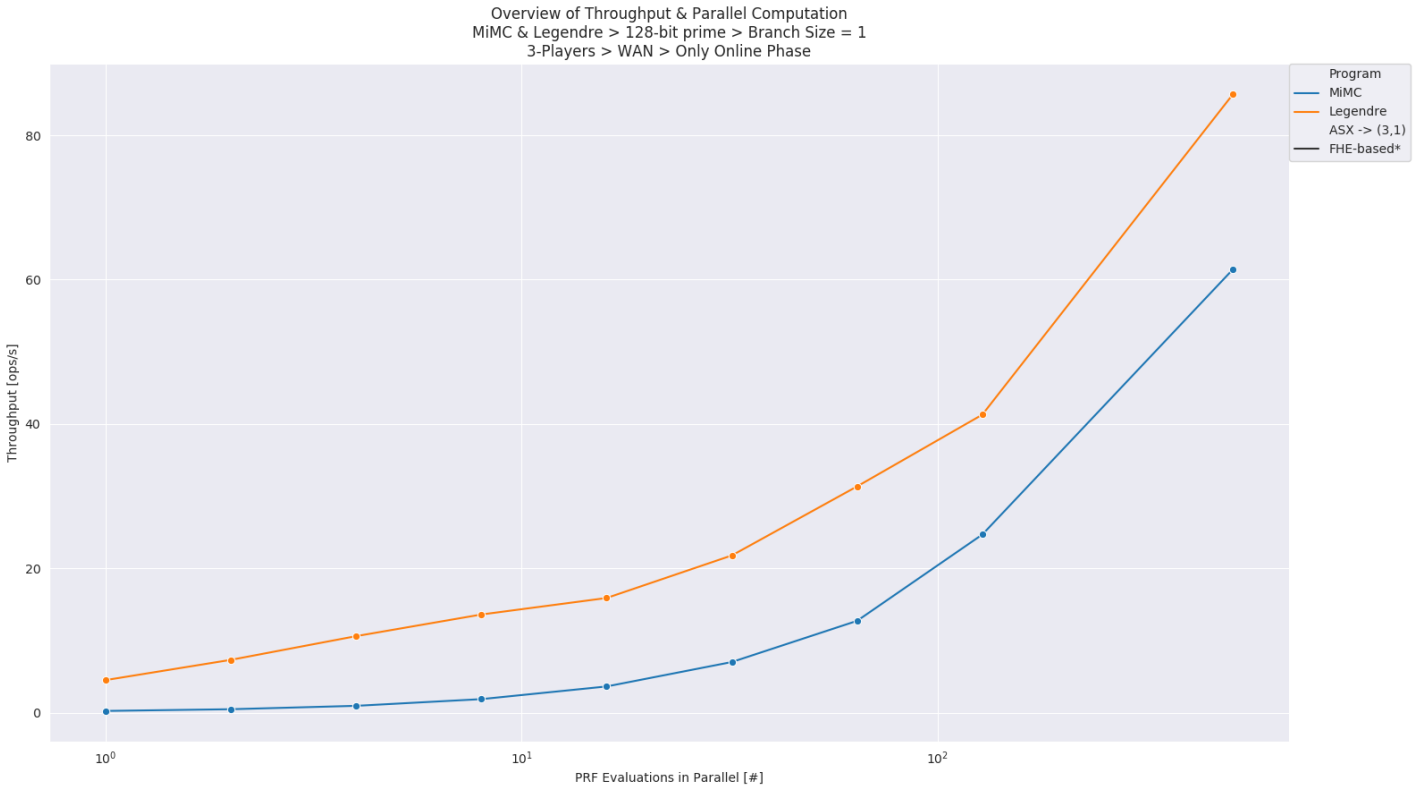


Figure 4.42: The throughput of the 3-players-WAN use case for MiMC and Leg using a branch size of one with only the online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4,8,16,32,64,128,256, and 512.

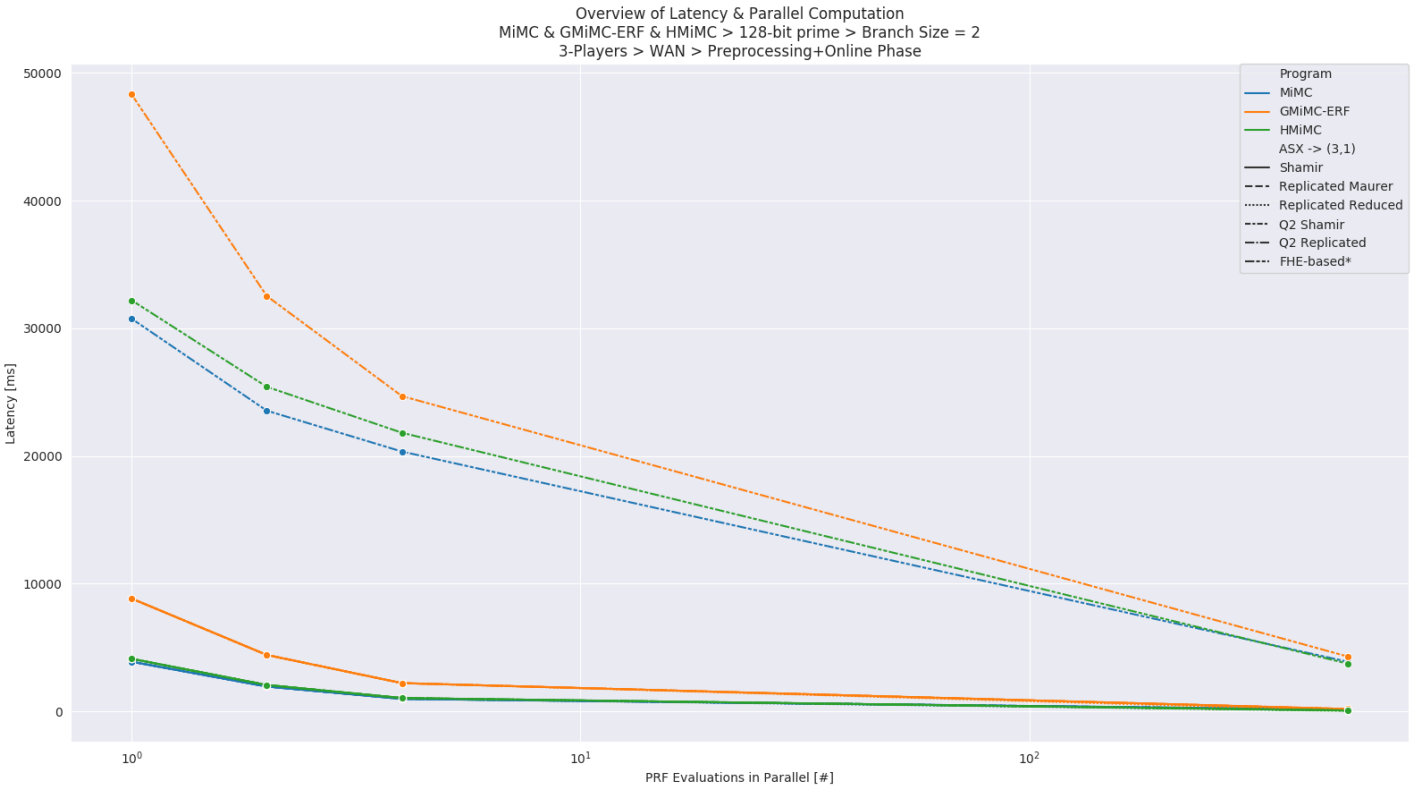


Figure 4.43: The latency of the 3-players-WAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with the preprocessing and online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

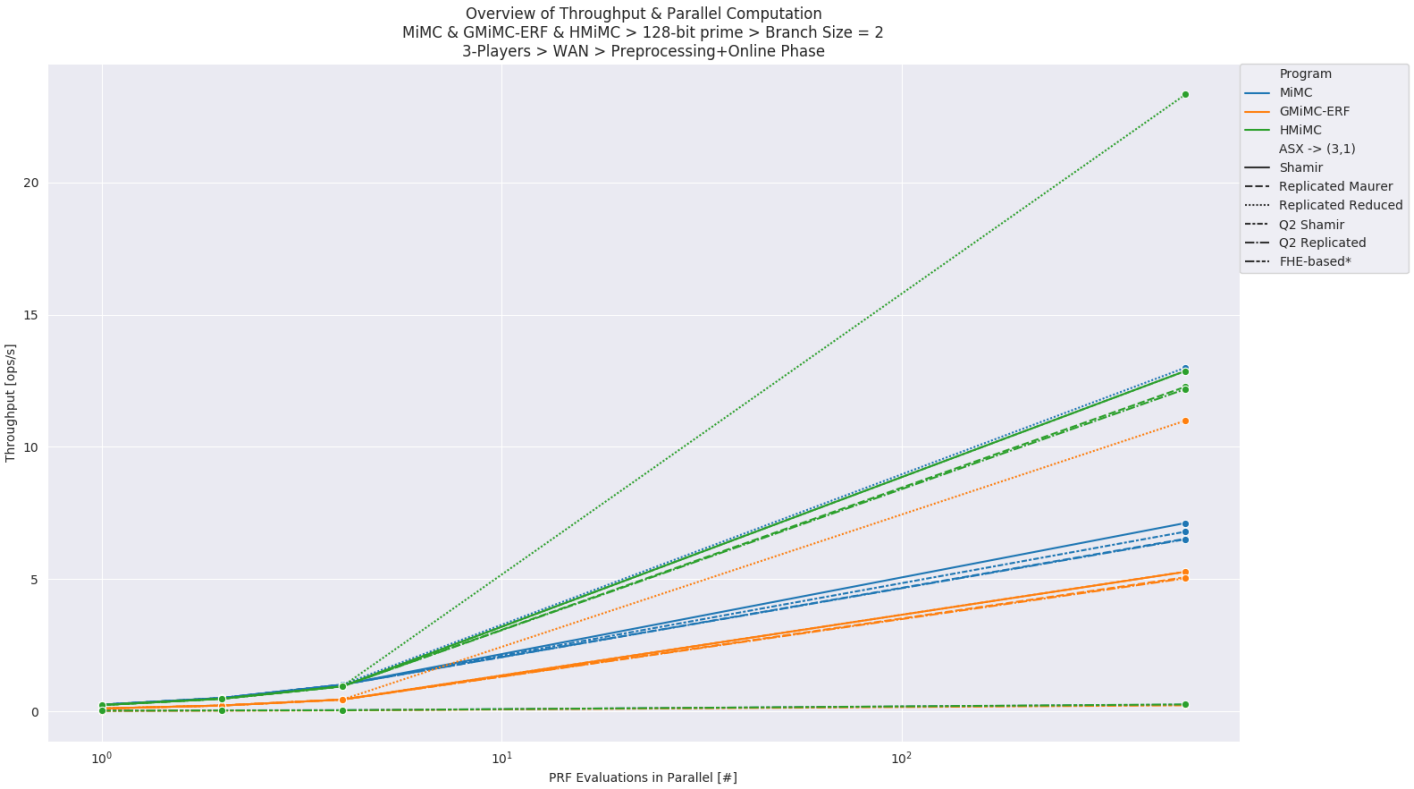


Figure 4.44: The throughput of the 3-players-WAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with the preprocessing and online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

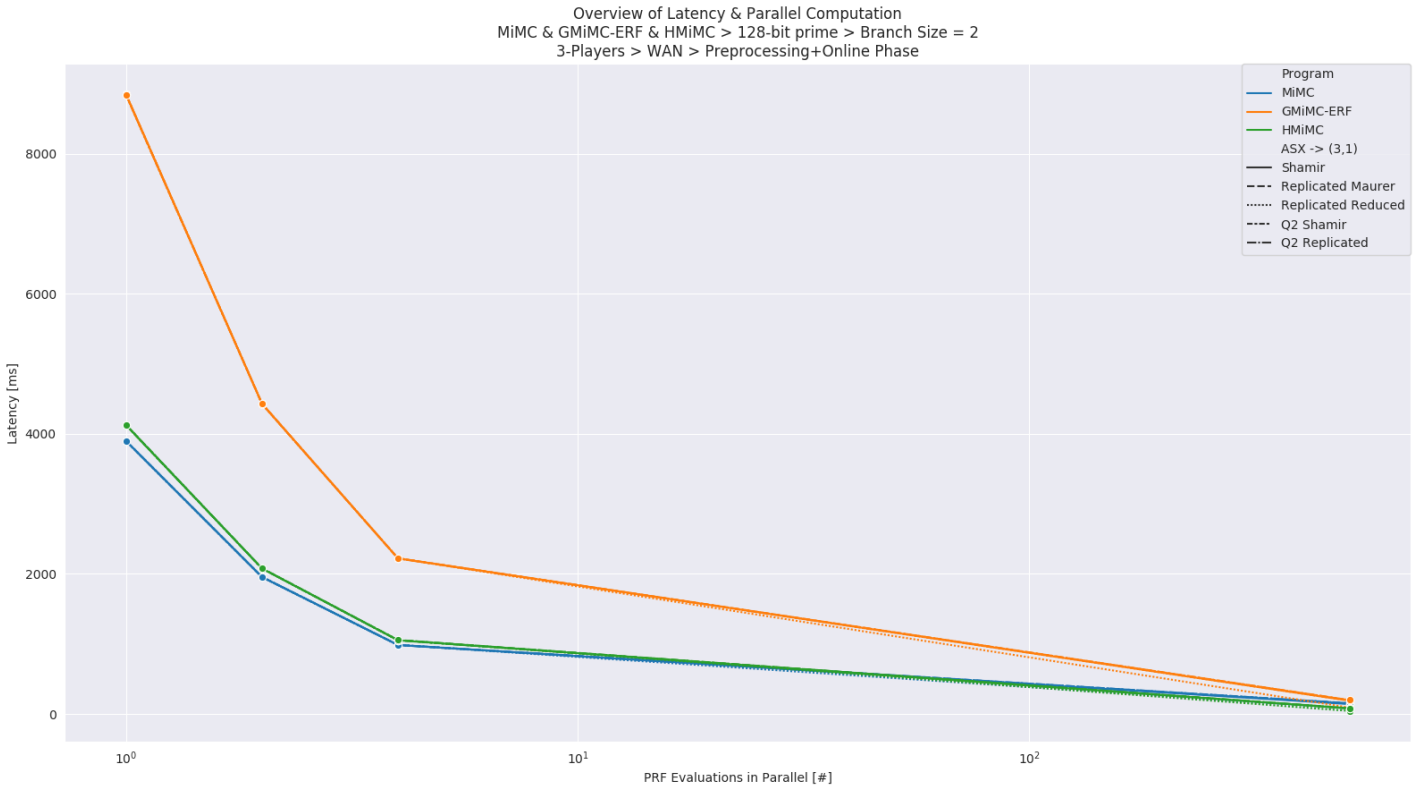


Figure 4.45: The latency of the 3-players-WAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with the preprocessing and online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

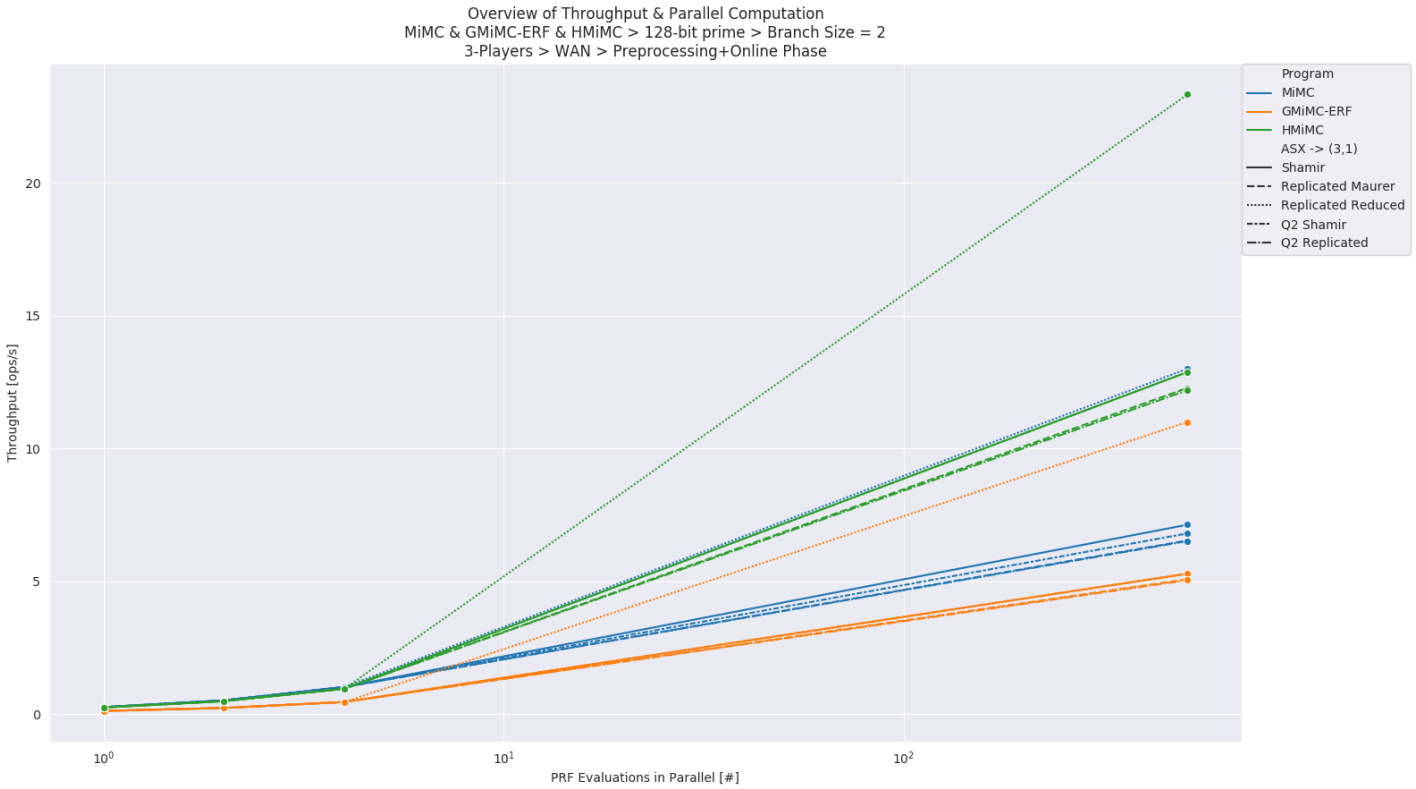


Figure 4.46: The throughput of the 3-players-WAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with the preprocessing and online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

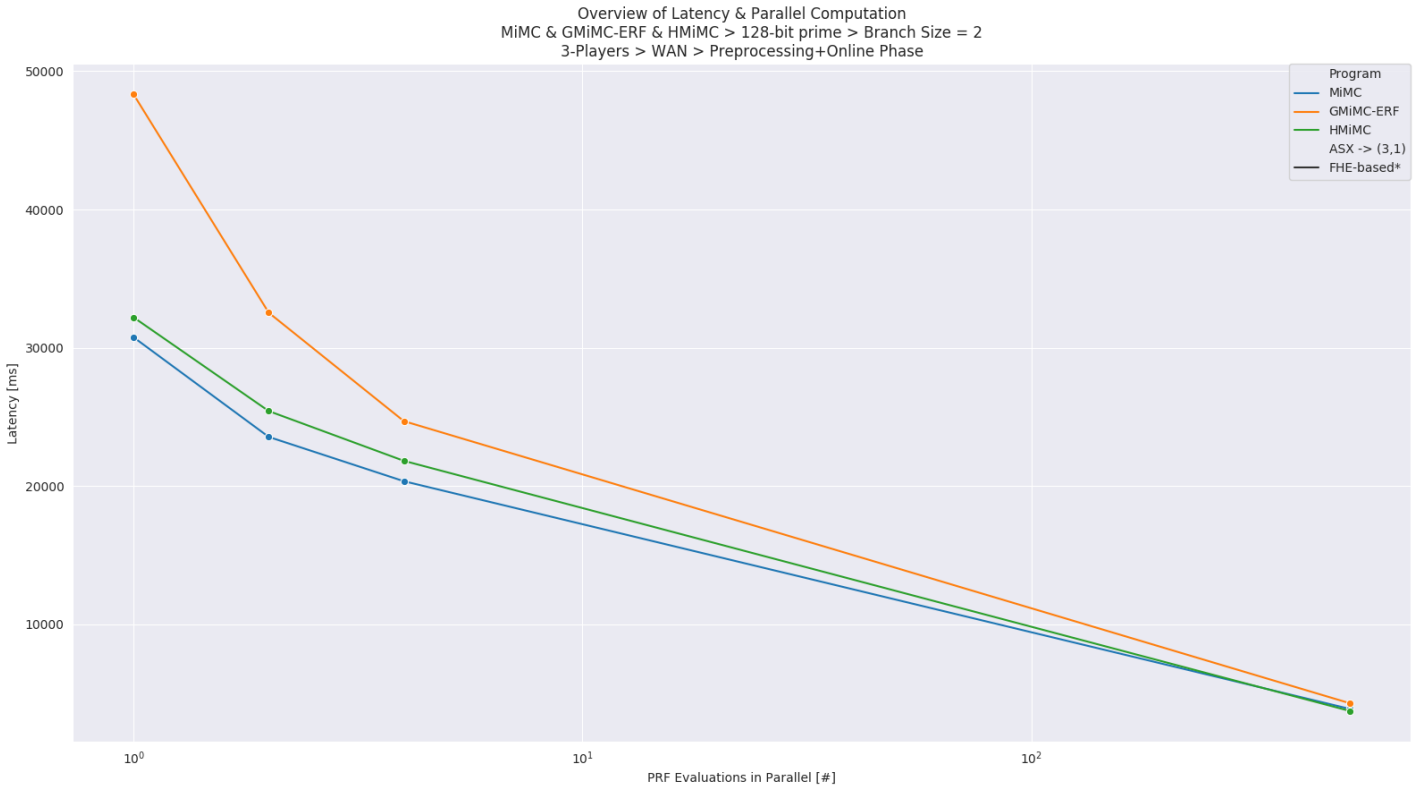


Figure 4.47: The latency of the 3-players-WAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with the preprocessing and online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4, and 512.

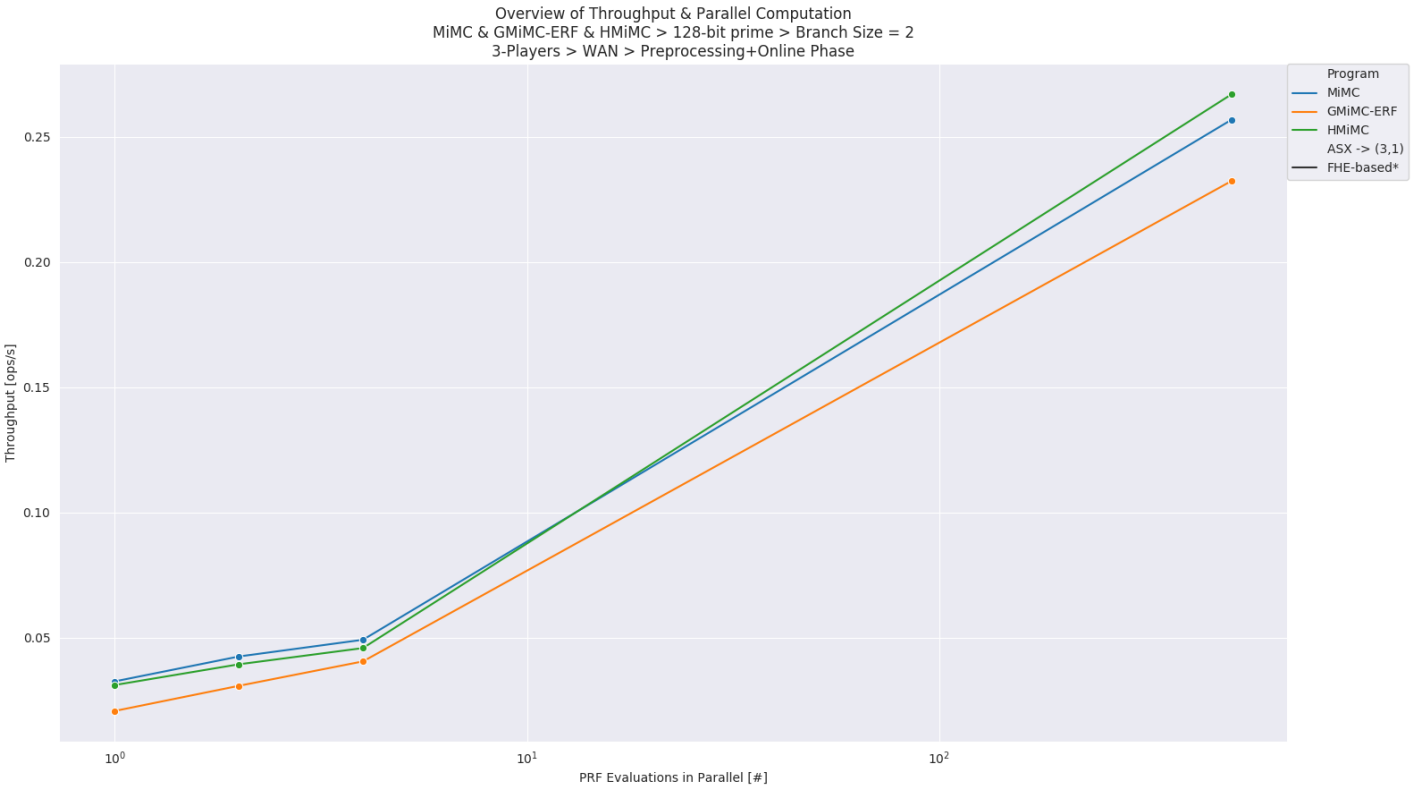


Figure 4.48: The throughput of the 3-players-WAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with the preprocessing and online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4, and 512.

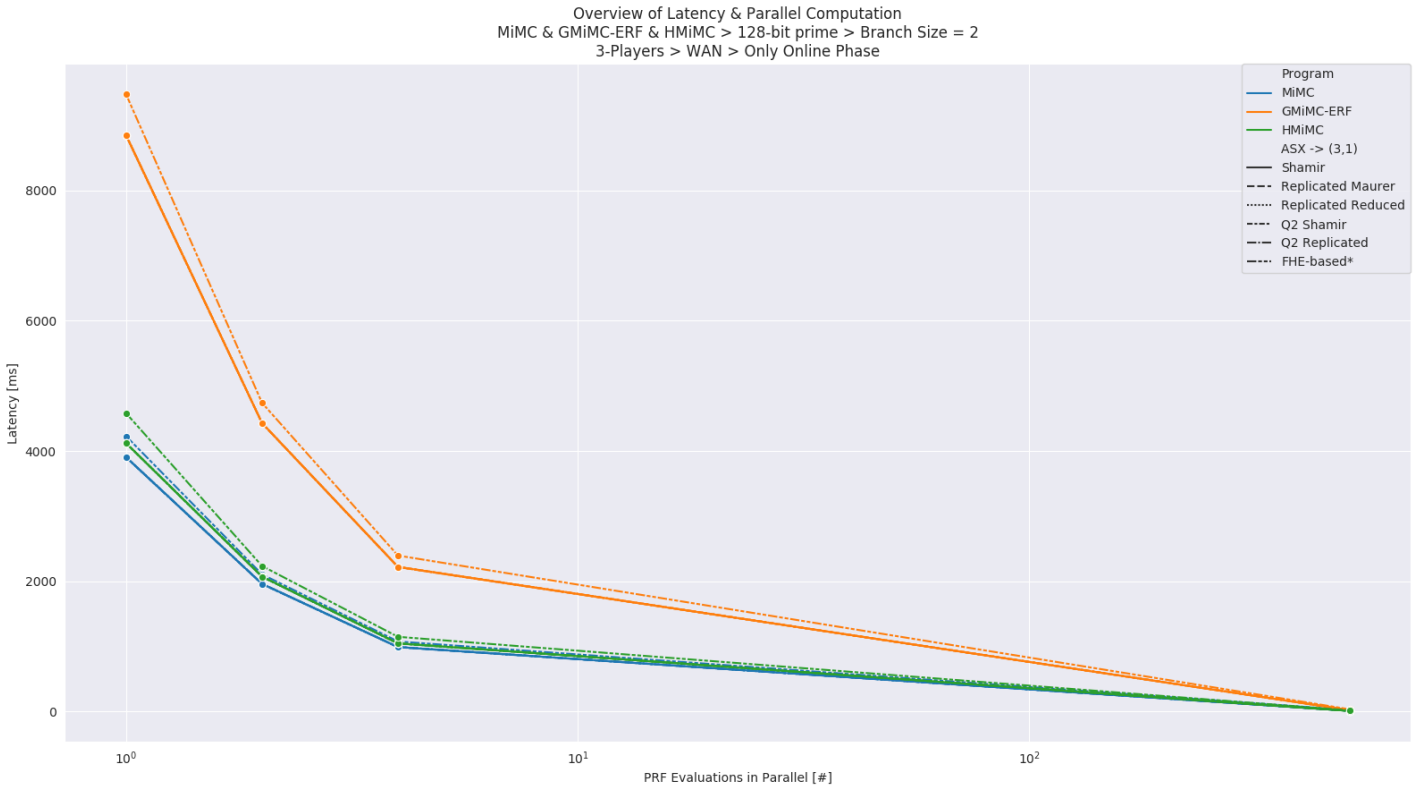


Figure 4.49: The latency of the 3-players-WAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with only the online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

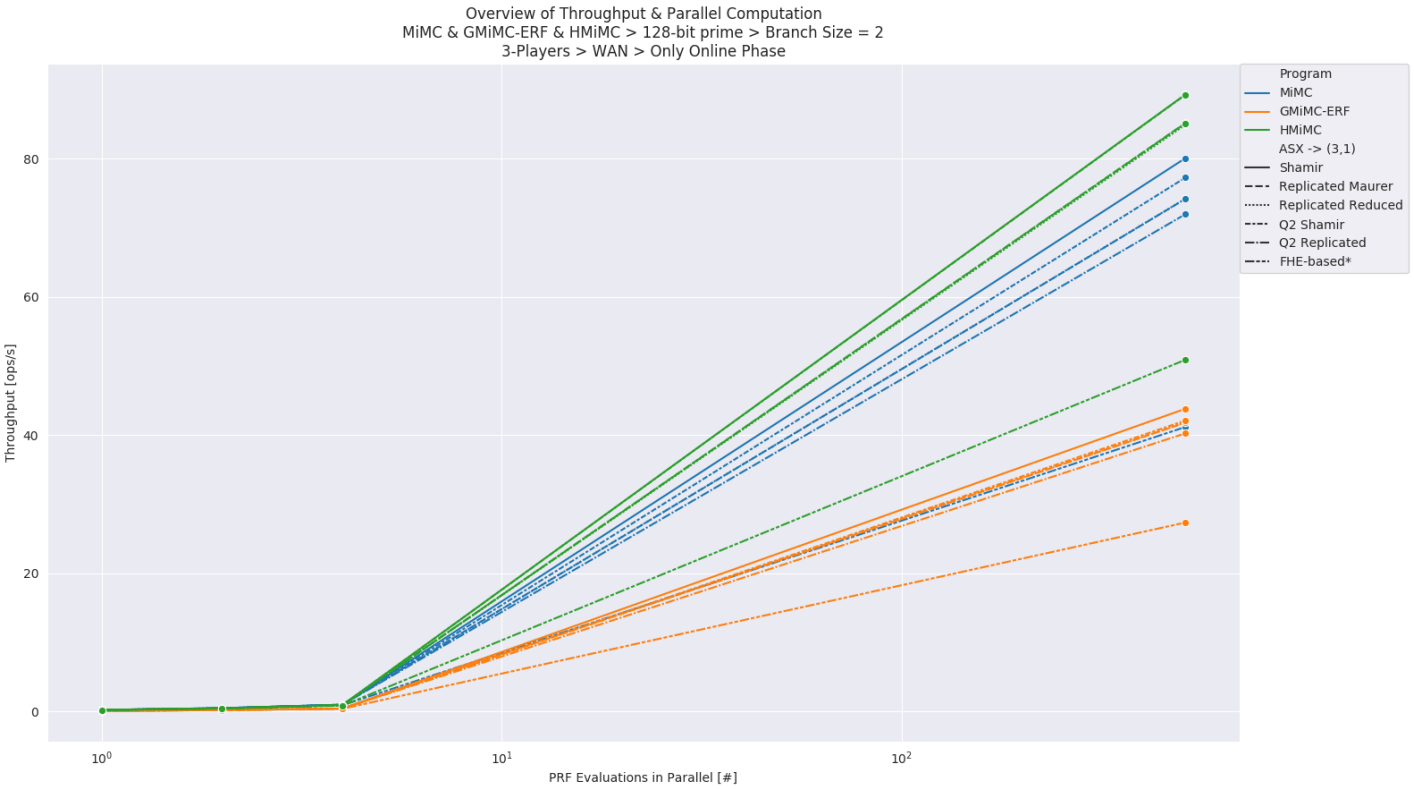


Figure 4.50: The throughput of the 3-players-WAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with only the online phase, all selected ASXs, and a batch size of 1,2,4, and 512.

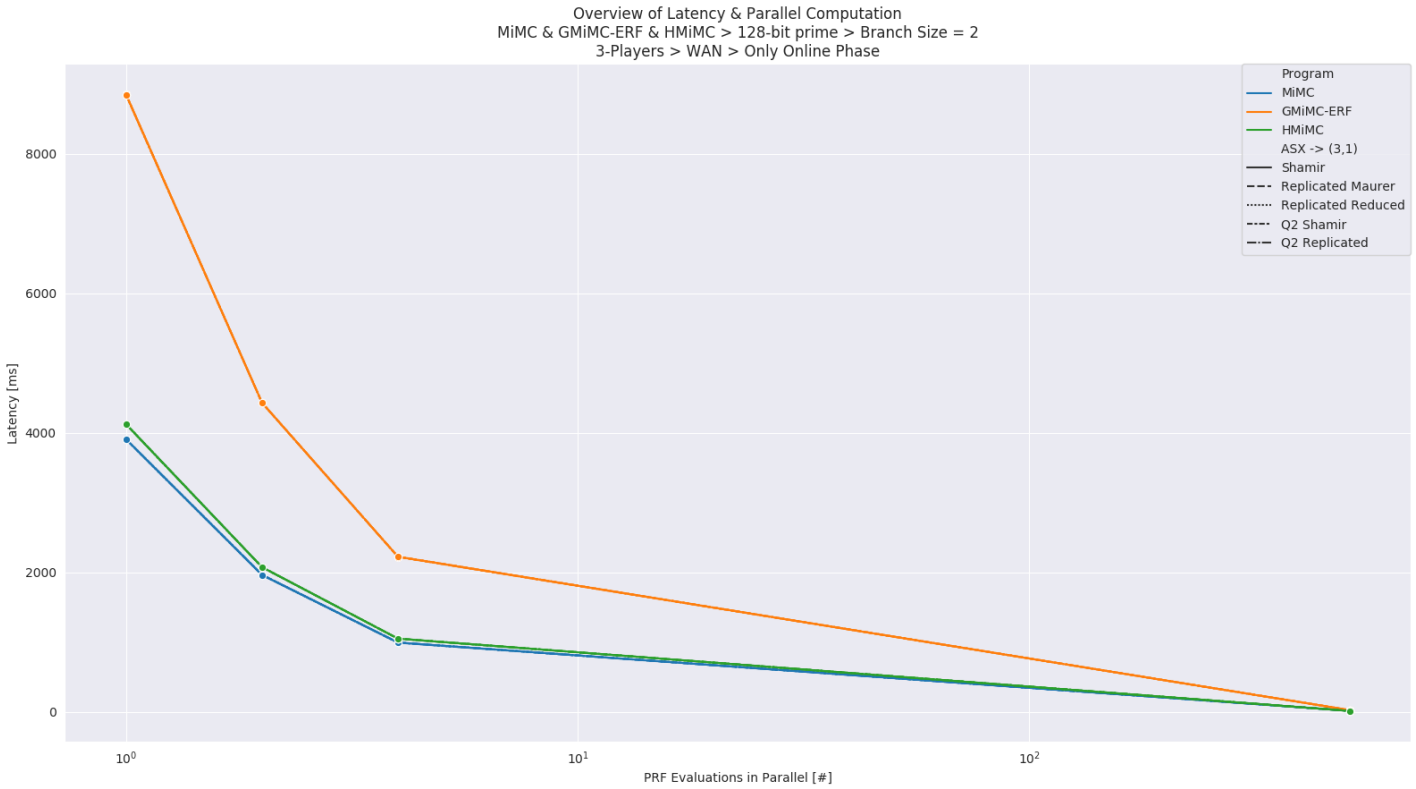


Figure 4.51: The latency of the 3-players-WAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with only the online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

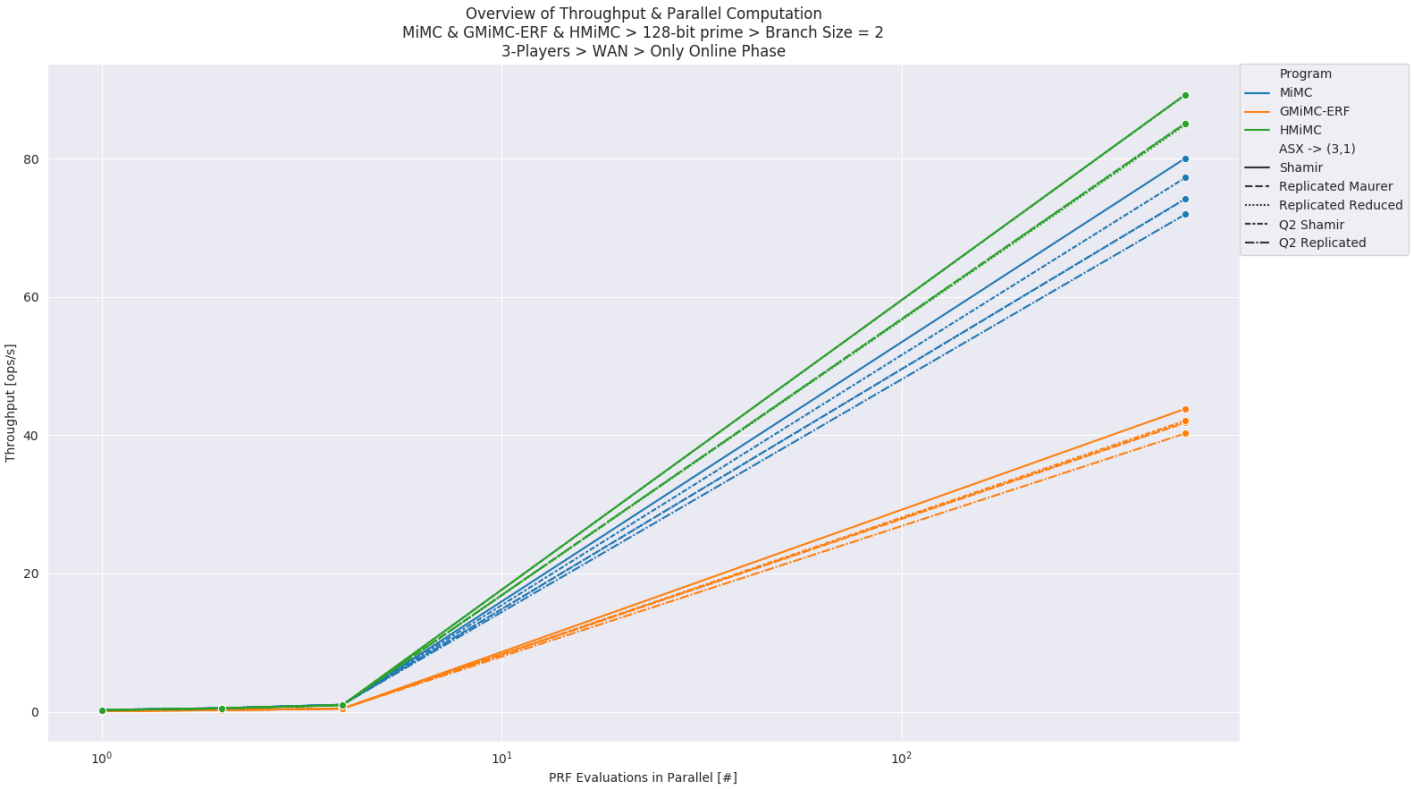


Figure 4.52: The throughput of the 3-players-WAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with only the online phase, our selected honest-majority ASXs, and a batch size of 1,2,4, and 512.

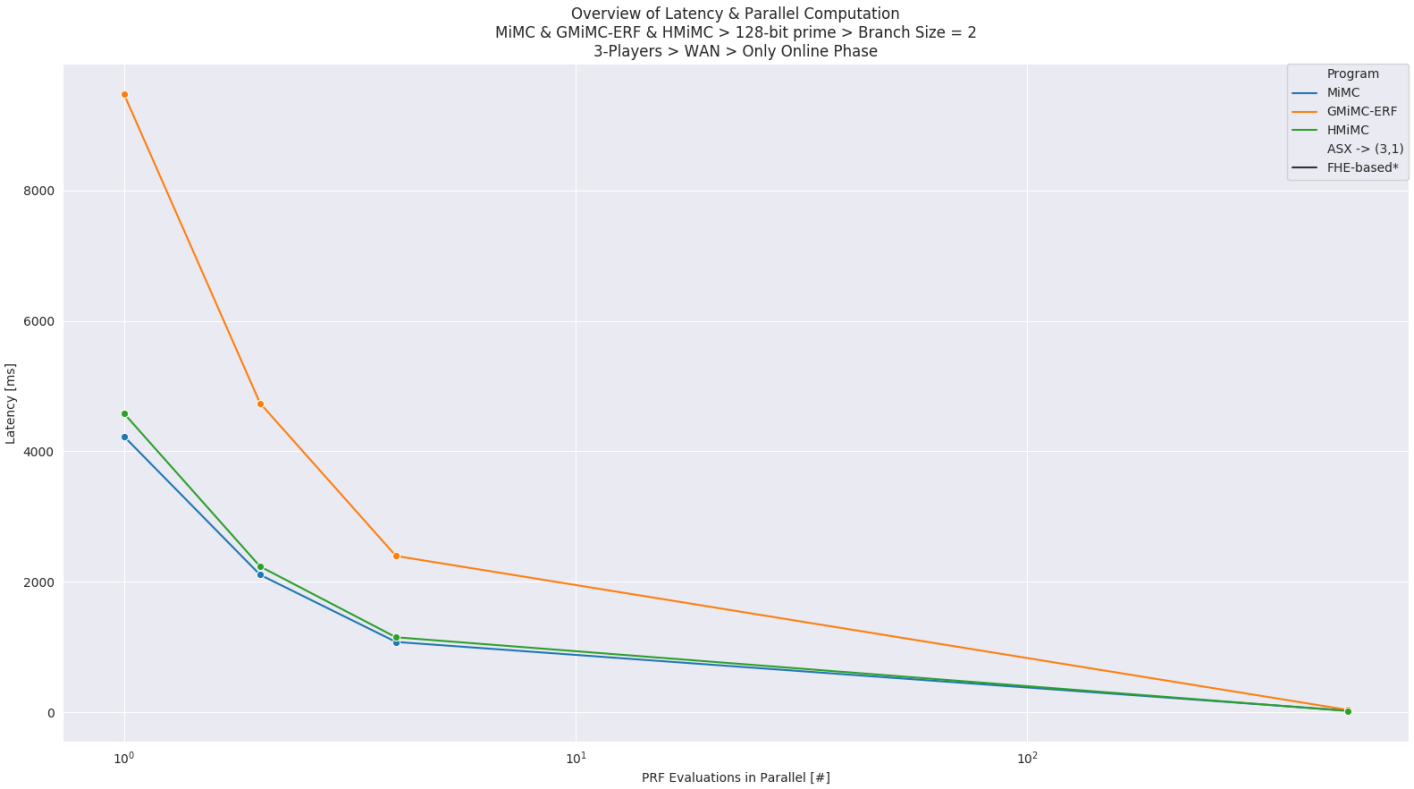


Figure 4.53: The latency of the 3-players-WAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with only the online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4, and 512.

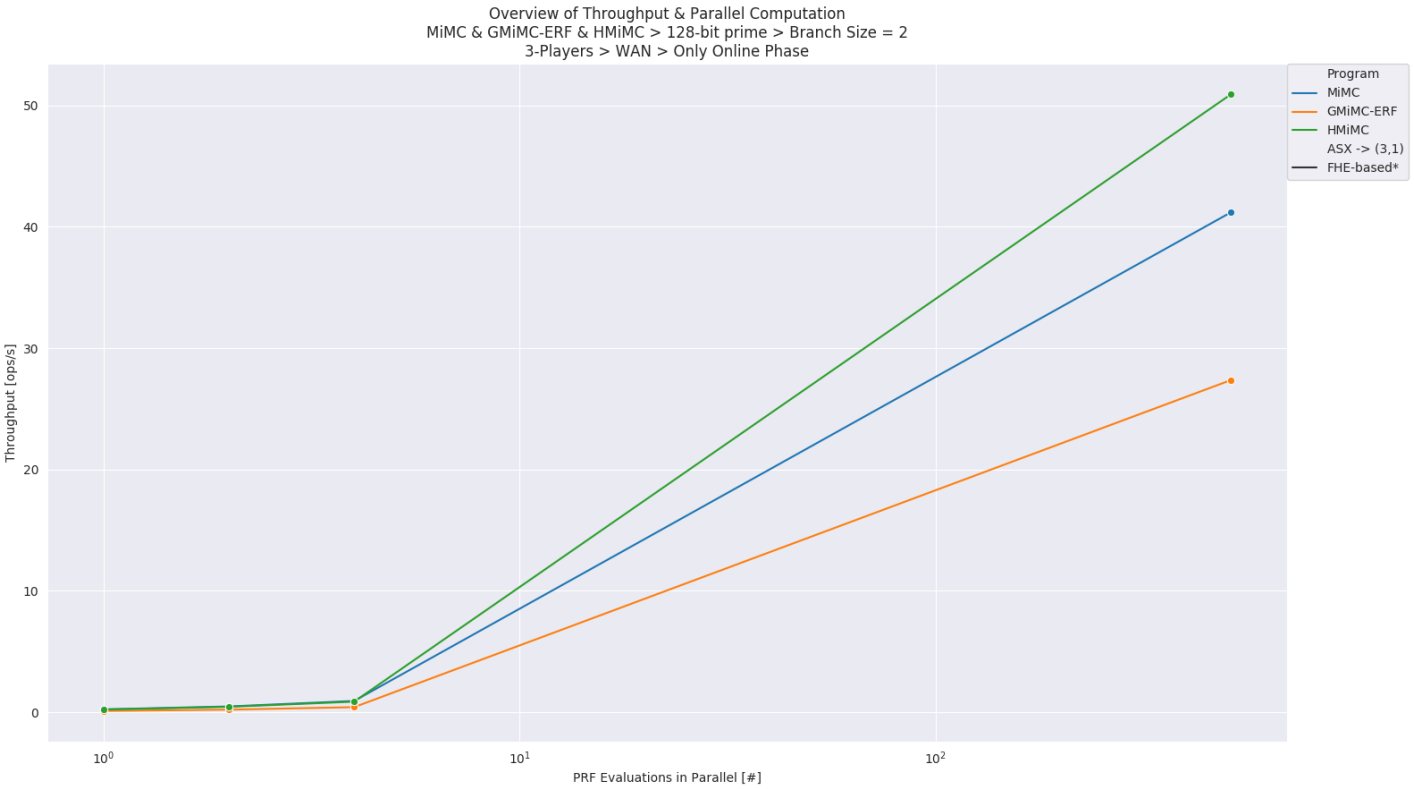


Figure 4.54: The throughput of the 3-players-WAN use case for MiMC, GMiMC-ERF, and HMiMC using a branch size of two with only the online phase, our selected dishonest-majority ASX, and a batch size of 1,2,4, and 512.

# Chapter 5

# Future Work

This chapter shows and briefly discusses potential future work. First, we show potential follow-up work of our identified benchmark (1) metrics, (2), dimensions, and (3) questions. Then, we show interesting future work in the area of (4) the framework Benchmarking for MPC (b4M) and (5) the evaluation of pseudo-random functions (PRFs) in secure multi-party computation (MPC), specifically the decryption process.

## 5.1 Improvement of Benchmark Metrics

**Runtime.** In terms of the runtime, it would be interesting to distinguish the measurement of the overall runtime in a more fine-grained way, by, e.g., splitting the cipher initialization and the actual PRF evaluation. In addition to showing the benchmark metrics for (1) only the online phase and (2) the preprocessing and online phase, showing (3) only the preprocessing phase could be interesting too.

**Communication & Memory consumption.** Graphs and evaluations of the communication and memory-consumption metric are not covered in this thesis, and are not yet part of b4M. When we also consider these two metrics, we are able to benchmark, compare, and evaluate in a more holistic way. Thus, in order to push the comparison of MPC-friendly PRFs to the next level, adding support for the benchmark metrics communicaiton and memory consumption is a must.

## 5.2 Expansion of Benchmark Dimensions

**Further PRFs.** We leave the benchmarking of further secure-multi-party-computation (MPC)-friendly PRFs as future work. For instance, benchmarking also the other variants of Generalized Feistel MiMC (GMiMC) with b4M. Furthermore, other PRFs in the area of $\mathbb{F}_p$ are Jarvis [AD18] and its successor Rescue [Aly+19], e.g. Although, Jarvis was broken recently [Alb+19a] (Asiacrypt 2019).

Moreover, evaluating PRFs which operate natively in $\mathbb{F}_p$ using garbled circuits might be interesting to benchmark as well. Such benchmarks would enable to compare evaluations using arithmetic circuits with recent developments in garbled circuits [BMR16]. Furthermore, the benchmarking of PRFs which do not natively operate in $\mathbb{F}_p$ might also be interesting. Though, as PRFs are (simply) programs to be evaluated with MPC, one basically just has to provide the corresponding program. Of course, the MPC engine has to support the underlying MPC protocol.

**Spanning benchmark dimensions.** In this thesis we split the benchmark dimensions into environment (env) and program (prog) dimensions. As we benchmarked only a specific setting, such as three parties (=env) and a branch size of one and two (=prog), evaluating the runtime, communication, and memory consumption for a more complete setting is an interesting future work. For instance, Single Instruction, Multiple Data (SIMD) is not covered in the benchmarks of this thesis. For future benchmarks it would be interesting to take also possibilities of SIMDs in MPC programs into account. Further examples are benchmarking with a larger number of parties, using different wide area network (WAN) settings, or larger branch sizes; to see if the impact is linear. Such as the impact of a branch size of 16, where GMiMC-expanding round function (ERF) outperforms HADES MiMC (HMiMC); and maybe one finds a setting where HMiMC is still better by tweaking, e.g., batch sizes?

**Exploring benchmark dimensions.** Introducing further benchmark dimensions might lead to interesting discoveries too. One such dimension, which is not covered in this thesis, is the machine specs (=env). As we have seen that

HMiMC's bottleneck for large branch sizes ($\sim \geq 16$) is the local computation, maybe with a better machine the performance can be further improved? Apart from the machine specs, maybe also the computer architecture or computer processor respectively, more specifically some concepts of it, can make an impact too; such as the SIMD operation, which requires support for advanced vector extensions (AVX). And AVX seems, currently, to be only supported by Intel and AMD processors [Wik20c; Wik20a; Che20].

## 5.3 Evaluation of Benchmark Questions

During the process of identifying benchmark dimensions for MPC programs, we also discovered interesting benchmark questions. And especially the env dimensions are relevant for MPC programs in general. Hence, evaluating these benchmark questions would be an interesting follow-up work.

**Tackling the evaluation.** Given the fact, that when we focus on one dimension, there are many possibilities for the other dimensions. For example, when we look at the question *Does a change in the network has a linear impact on the runtime?*, we have to fix all the dimensions besides the network to a specific value. And there are many possible values for the other dimensions. Thus, getting reliable answers for all these questions is complex. An approach to tackle this complexity could be to consider first only a few options for the other dimensions, like setting them to a specific use case. And then, adding complexity along they way.

## 5.4 Enhancement of Benchmarking for MPC (b4M)

**On GUIs and engines.** The whole process of benchmarking using b4M works via editing Python files, followed by the execution in the terminal or a Python interpreter respectively. To simplify the configuration of benchmarks and execution of thereof, the enhancement of b4M would be interesting future work. This enhancement could improve, e.g., two aspects. Firstly, instead of editing configurations in Python files, a dedicated graphical user interface

(GUI) can be introduced. Secondly, to have a broader view of MPC executions, b4M can be expanded to benchmark programs using even diffrent MPC engines; like MP-SPDZ [Kel20] or Fresco [Ale20] in addition. This approach of comparing the execution with different MPC engines, could lead to interesting discoveries of advantages and disadvantages (trade-offs) among the different engines.

## 5.5 Consideration of Decryption

**Inverting encryption.** In this thesis, we only considered benchmarking of the encryption process of the PRFs. There might be use cases where one does not only need the encryption of a PRF in MPC, but the decryption too. For such use cases it would be interesting to also consider the decryption process of the PRFs evaluated in MPC. As this (simply) requires to add the functionality in the corresponding MPC programs, the evaluation using b4M is straightforward.

Some PRFs, as, e.g., MiMC note that decryption is much less efficient than encryption. Thus, the authors of MiMC recommend to use the cipher in a context where only encryption is needed. Furthermore, in the "plain" Legendre PRF, decryption is not even possible. To also enable decryption for Legendre, we would need to put it in, e.g., a Feistel Network or a different mode of operation, such as the counter mode [Wik20d; DH79].

# Chapter 6

# Conclusion

The main contributions of this thesis are fourfold:

1. Creation of the benchmark framework *Benchmarking for MPC (b4M)* for the secure multi-party computation (MPC)-engine SCALE-MAMBA (SCALE);
2. Identification of relevant benchmark metrics (i.e. requirements), for
   a) MPC programs in general and also
   b) specifically for *MPC-friendly pseudo-random functions (PRFs)*;
3. Identificaton of interesting and (potentially) relevant benchmark settings and questions;
4. Benchmarking and evaluation of the two use cases
   a) 3-players with (basically) no network restrictions (local area network (LAN)), and
   b) 3-players which are connected within a limited network (wide area network (WAN)).

Furthermore, with respect to future work, we identified interesting paths to follow.

# Bibliography


[AD18] T. Ashur and S. Dhooghe. "MARVELlous: a STARK-Friendly Family of Cryptographic Primitives." In: *IACR Cryptol. ePrint Arch.* 2018 (2018), p. 1098. URL: https://eprint.iacr.org/2018/1098 (cit. on p. 127).

[Alb+15] M. R. Albrecht et al. "Ciphers for MPC and FHE." In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*. Ed. by E. Oswald and M. Fischlin. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 430–454. DOI: 10.1007/978-3-662-46800-5_17. URL: https://doi.org/10.1007/978-3-662-46800-5_17 (cit. on p. 16).

[Alb+16] M. R. Albrecht et al. "MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity." In: *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*. Ed. by J. H. Cheon and T. Takagi. Vol. 10031. Lecture Notes in Computer Science. 2016, pp. 191–219. DOI: 10.1007/978-3-662-53887-6_7. URL: https://doi.org/10.1007/978-3-662-53887-6_7 (cit. on pp. 16, 17, 19, 21).

[Alb+19a] M. R. Albrecht et al. "Algebraic Cryptanalysis of STARK-Friendly Designs: Application to MARVELlous and MiMC." In: *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part III*. Ed. by S. D. Galbraith and S. Moriai. Vol. 11923. Lecture Notes in Computer Science. Springer, 2019, pp. 371–397. DOI: 10.1007/978-3-030-

34618-8_13. URL: https://doi.org/10.1007/978-3-030-34618-8_13 (cit. on p. 127).

[Alb+19b] M. R. Albrecht et al. "Feistel Structures for MPC, and More." In: *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*. Ed. by K. Sako, S. Schneider, and P. Y. A. Ryan. Vol. 11736. Lecture Notes in Computer Science. Springer, 2019, pp. 151–171. DOI: 10.1007/978-3-030-29962-0_8. URL: https://doi.org/10.1007/978-3-030-29962-0_8 (cit. on pp. 16, 23, 27, 28).

[Ale20] Alexandra Institute. *FRESCO - A FRamework for Efficient Secure COmputation*. Accessed on October 2020. URL: https://github.com/aicis/fresco (cit. on p. 129).

[Aly+19] A. Aly et al. "Efficient Symmetric Primitives for Advanced Cryptographic Protocols (A Marvellous Contribution)." In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 426. URL: https://eprint.iacr.org/2019/426 (cit. on pp. 16, 127).

[Ben+13] E. Ben-Sasson et al. "SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge." In: *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*. Ed. by R. Canetti and J. A. Garay. Vol. 8043. Lecture Notes in Computer Science. Springer, 2013, pp. 90–108. DOI: 10.1007/978-3-642-40084-1_6. URL: https://doi.org/10.1007/978-3-642-40084-1_6 (cit. on p. 22).

[Ben+14] E. Ben-Sasson et al. "Zerocash: Decentralized Anonymous Payments from Bitcoin." In: *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, 2014, pp. 459–474. DOI: 10.1109/SP.2014.36. URL: https://doi.org/10.1109/SP.2014.36 (cit. on p. 22).

[Ben+18] E. Ben-Sasson et al. "Scalable, transparent, and post-quantum secure computational integrity." In: *IACR Cryptol. ePrint Arch.* 2018 (2018), p. 46. URL: http://eprint.iacr.org/2018/046 (cit. on p. 22).

[Beu+20] W. Beullens et al. "Cryptanalysis of the Legendre PRF and Generalizations." In: *IACR Trans. Symmetric Cryptol.* 2020.1 (2020), pp. 313–330. DOI: 10.13154/tosc.v2020.i1.313-330. URL: https://doi.org/10.13154/tosc.v2020.i1.313-330 (cit. on p. 38).

[Bey+20] T. Beyne et al. "Out of Oddity - New Cryptanalytic Techniques Against Symmetric Primitives Optimized for Integrity Proof Systems." In: *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*. Ed. by D. Micciancio and T. Ristenpart. Vol. 12172. Lecture Notes in Computer Science. Springer, 2020, pp. 299–328. DOI: 10.1007/978-3-030-56877-1_11. URL: https://doi.org/10.1007/978-3-030-56877-1_11 (cit. on pp. 27, 34).

[BGL20] E. Ben-Sasson, L. Goldberg, and D. Levit. "STARK Friendly Hash - Survey and Recommendation." In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 948. URL: https://eprint.iacr.org/2020/948 (cit. on pp. 22, 28, 35).

[BMR16] M. Ball, T. Malkin, and M. Rosulek. "Garbling Gadgets for Boolean and Arithmetic Circuits." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by E. R. Weippl et al. ACM, 2016, pp. 565–577. DOI: 10.1145/2976749.2978410. URL: https://doi.org/10.1145/2976749.2978410 (cit. on p. 127).

[Bon+18] D. Boneh et al. "Exploring Crypto Dark Matter: - New Simple PRF Candidates and Their Applications." In: *Theory of Cryptography - 16th International Conference, TCC 2018, Panaji, India, November 11-14, 2018, Proceedings, Part II*. Ed. by A. Beimel and S. Dziembowski. Vol. 11240. Lecture Notes in Computer Science. Springer, 2018, pp. 699–729. DOI: 10.1007/978-3-030-03810-6_25. URL: https://doi.org/10.1007/978-3-030-03810-6_25 (cit. on pp. 6, 16).

[BS20] W. Beullens and C. D. de Saint Guilhem. "LegRoast: Efficient Post-quantum Signatures from the Legendre PRF." In: *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020, Paris, France, April 15-17, 2020, Proceedings*. Ed. by J. Ding and J. Tillich.

Vol. 12100. Lecture Notes in Computer Science. Springer, 2020, pp. 130–150. DOI: 10.1007/978-3-030-44223-1_8. URL: https://doi.org/10.1007/978-3-030-44223-1_8 (cit. on p. 38).

[BZL20] E. Blum, C. L. Zhang, and J. Loss. "Always Have a Backup Plan: Fully Secure Synchronous MPC with Asynchronous Fallback." In: *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*. Ed. by D. Micciancio and T. Ristenpart. Vol. 12171. Lecture Notes in Computer Science. Springer, 2020, pp. 707–731. DOI: 10.1007/978-3-030-56880-1_25. URL: https://doi.org/10.1007/978-3-030-56880-1_25 (cit. on p. 42).

[Can00] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols." In: *IACR Cryptol. ePrint Arch.* 2000 (2000), p. 67. URL: http://eprint.iacr.org/2000/067 (cit. on p. 45).

[Che20] Chess Programming Wiki. *AVX*. Accessed on October 2020. URL: https://www.chessprogramming.org/AVX (cit. on p. 128).

[Cur+06] R. Curtmola et al. "Searchable symmetric encryption: improved definitions and efficient constructions." In: *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*. Ed. by A. Juels, R. N. Wright, and S. D. C. di Vimercati. ACM, 2006, pp. 79–88. DOI: 10.1145/1180405.1180417. URL: https://doi.org/10.1145/1180405.1180417 (cit. on p. 6).

[DA99] T. Dierks and C. Allen. "The TLS Protocol Version 1.0." In: *RFC* 2246 (1999), pp. 1–80. DOI: 10.17487/RFC2246. URL: https://doi.org/10.17487/RFC2246 (cit. on p. 6).

[Dam88] I. Damgård. "On the Randomness of Legendre and Jacobi Sequences." In: *Advances in Cryptology - CRYPTO '88, 8th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1988, Proceedings*. Ed. by S. Goldwasser. Vol. 403. Lecture Notes in Computer Science. Springer, 1988, pp. 163–172. DOI: 10.1007/0-387-34799-2_13. URL: https://doi.org/10.1007/0-387-34799-2_13 (cit. on pp. 16, 36, 37).

[dan20] dankrad .at. ethereum .dot. org. *Legendre pseudo-random function*. Accessed on September 2020. URL: https://legendreprf.org/ (cit. on p. 38).

[Dev20a] tqdm Developers. *Fast, Extensible Progress Meter*. Accessed on May 2020. URL: https://pypi.org/project/tqdm/ (cit. on p. 66).

[Dev20b] lxml Development Team. *Powerful and Pythonic XML processing library combining libxml2/libxslt with the ElementTree API*. Accessed on May 2020. URL: https://pypi.org/project/lxml/ (cit. on p. 66).

[DH79] W. Diffie and M. E. Hellman. "Privacy and authentication: An introduction to cryptography." In: *Proceedings of the IEEE* 67.3 (1979), pp. 397–427. DOI: 10.1109/PROC.1979.11256 (cit. on p. 129).

[DR98] J. Daemen and V. Rijmen. "The Block Cipher Rijndael." In: *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings*. Ed. by J. Quisquater and B. Schneier. Vol. 1820. Lecture Notes in Computer Science. Springer, 1998, pp. 277–284. DOI: 10.1007/10721064_26. URL: https://doi.org/10.1007/10721064_26 (cit. on pp. 6, 15).

[Eic+20] M. Eichlseder et al. "An Algebraic Attack on Ciphers with Low-Degree Round Functions: Application to Full MiMC." In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 182. URL: https://eprint.iacr.org/2020/182 (cit. on p. 21).

[Eth20] Ethresearch - dankrad. *Using the Legendre symbol as a PRF for the Proof of Custody*. Accessed on September 2020. URL: https://ethresear.ch/t/using-the-legendre-symbol-as-a-prf-for-the-proof-of-custody/5169 (cit. on p. 38).

[Goh03] E. Goh. "Secure Indexes." In: *IACR Cryptol. ePrint Arch.* 2003 (2003), p. 216. URL: https://eprint.iacr.org/2003/216 (cit. on p. 6).

[Gol01] O. Goldreich. *The Foundations of Cryptography - Volume 1: Basic Techniques*. Cambridge University Press, 2001. ISBN: 0-521-79172-3. DOI: 10.1017/CBO9780511546891. URL: http://www.wisdom.weizmann.ac.il/%5C%7Eoded/foc-vol1.html (cit. on p. 6).

[Gra+16] L. Grassi et al. "MPC-Friendly Symmetric Key Primitives." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by E. R. Weippl et al. ACM, 2016, pp. 430–443. DOI: 10.1145/2976749.2978332. URL: https://doi.org/10.1145/2976749.2978332 (cit. on pp. iv, vi, 8, 9, 11, 15, 16, 21, 22, 36–38, 48, 52–54, 68, 70–72, 74–76).

[Gra+20] L. Grassi et al. "On a Generalization of Substitution-Permutation Networks: The HADES Design Strategy." In: *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II*. Ed. by A. Canteaut and Y. Ishai. Vol. 12106. Lecture Notes in Computer Science. Springer, 2020, pp. 674–704. DOI: 10.1007/978-3-030-45724-2_23. URL: https://doi.org/10.1007/978-3-030-45724-2_23 (cit. on pp. 16, 28, 34).

[Gra20] Graz University of Technology. *Meltdown and Spectre - Vulnerabilities in modern computers leak passwords and sensitive data*. Accessed on October 2020. URL: https://meltdownattack.com/ (cit. on p. 4).

[HS08] G. Herman and M. Soltys. "A polytime proof of correctness of the Rabin-Miller algorithm from Fermat's little theorem." In: *CoRR* abs/0811.3959 (2008). arXiv: 0811.3959. URL: http://arxiv.org/abs/0811.3959 (cit. on p. 36).

[Kel20] M. Keller. "MP-SPDZ: A Versatile Framework for Multi-Party Computation." In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 521. URL: https://eprint.iacr.org/2020/521 (cit. on p. 129).

[Kho19] D. Khovratovich. "Key recovery attacks on the Legendre PRFs within the birthday bound." In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 862. URL: https://eprint.iacr.org/2019/862 (cit. on p. 36).

[KL14] J. Katz and Y. Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014. ISBN: 9781466570269. URL: https://www.crcpress.com/Introduction-to-Modern-Cryptography-Second-Edition/Katz-Lindell/p/book/9781466570269 (cit. on p. 37).

[Koc+19] P. Kocher et al. "Spectre Attacks: Exploiting Speculative Execution." In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002. URL: https://doi.org/10.1109/SP.2019.00002 (cit. on p. 4).

[KPR18] M. Keller, V. Pastro, and D. Rotaru. "Overdrive: Making SPDZ Great Again." In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*. Ed. by J. B. Nielsen and V. Rijmen. Vol. 10822. Lecture Notes in Computer Science. Springer, 2018, pp. 158–189. DOI: 10.1007/978-3-319-78372-7_6. URL: https://doi.org/10.1007/978-3-319-78372-7_6 (cit. on p. 15).

[KR20] N. Keller and A. Rosemarin. "Mind the Middle Layer: The HADES Design Strategy Revisited." In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 179. URL: https://eprint.iacr.org/2020/179 (cit. on p. 34).

[KU 20] KU Leuven - Nigel Smart et al. *Repository for the SCALE-MAMBA MPC system*. Accessed on September 2020. URL: https://github.com/KULeuven-COSIC/SCALE-MAMBA (cit. on p. 54).

[Lin16] Y. Lindell. "How To Simulate It - A Tutorial on the Simulation Proof Technique." In: *IACR Cryptol. ePrint Arch.* 2016 (2016), p. 46. URL: http://eprint.iacr.org/2016/046 (cit. on p. 45).

[Lin20] Y. Lindell. "Secure Multiparty Computation (MPC)." In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 300. URL: https://eprint.iacr.org/2020/300 (cit. on pp. 44, 45).

[Lip+18] M. Lipp et al. "Meltdown: Reading Kernel Memory from User Space." In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by W. Enck and A. P. Felt. USENIX Association, 2018, pp. 973–990. URL:

https://www.usenix.org/conference/usenixsecurity18/presentation/lipp (cit. on p. 4).

[NK95] K. Nyberg and L. R. Knudsen. "Provable Security Against a Differential Attack." In: *J. Cryptology* 8.1 (1995), pp. 27–37. DOI: 10.1007/BF00204800. URL: https://doi.org/10.1007/BF00204800 (cit. on p. 17).

[NR97] M. Naor and O. Reingold. "Number-theoretic Constructions of Efficient Pseudo-random Functions." In: *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*. IEEE Computer Society, 1997, pp. 458–467. DOI: 10.1109/SFCS.1997.646134. URL: https://doi.org/10.1109/SFCS.1997.646134 (cit. on p. 16).

[Ras20] Raspberry Pi Foundation. *Raspberry Pi 4 Tech Specs*. Accessed on September 2020. URL: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/ (cit. on p. 44).

[Sta20a] StarkWare Industries Ltd. *About*. Accessed on October 2020. URL: https://starkware.co/about-us/ (cit. on p. 22).

[Sta20b] StarkWare Industries Ltd. *STARK-Friendly Hash Challenge*. Accessed on October 2020. URL: https://starkware.co/developers-community/hash-challenge/ (cit. on pp. 22, 28, 35).

[SWP00] D. X. Song, D. A. Wagner, and A. Perrig. "Practical Techniques for Searches on Encrypted Data." In: *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*. IEEE Computer Society, 2000, pp. 44–55. DOI: 10.1109/SECPRI.2000.848445. URL: https://doi.org/10.1109/SECPRI.2000.848445 (cit. on p. 6).

[Tea20] P. D. Team. *Powerful data structures for data analysis, time series, and statistics*. Accessed on May 2020. URL: https://pypi.org/project/pandas/ (cit. on p. 66).

[Ver08] F. Vercauteren. "The Hidden Root Problem." In: *Pairing-Based Cryptography - Pairing 2008, Second International Conference, Egham, UK, September 1-3, 2008. Proceedings*. Ed. by S. D. Galbraith and K. G. Paterson. Vol. 5209. Lecture Notes in Computer Science.

Springer, 2008, pp. 89–99. DOI: 10.1007/978-3-540-85538-5_6. URL: https://doi.org/10.1007/978-3-540-85538-5_6 (cit. on p. 37).

[Was20] M. Waskom. *seaborn: statistical data visualization*. Accessed on May 2020. URL: https://pypi.org/project/seaborn/ (cit. on p. 66).

[Wik20a] WikiChip. *Advanced Vector Extensions 512 (AVX-512) - x86*. Accessed on October 2020. URL: https://en.wikichip.org/wiki/x86/avx-512 (cit. on p. 128).

[Wik20b] WikiChip. *Resident Set Size (RSS)*. Accessed on September 2020. URL: https://en.wikichip.org/wiki/resident_set_size (cit. on p. 61).

[Wik20c] Wikipedia. *Advanced Vector Extensions*. Accessed on October 2020. URL: https://en.wikipedia.org/wiki/Advanced_Vector_Extensions (cit. on p. 128).

[Wik20d] Wikipedia. *Block cipher mode of operation*. Accessed on December 2020. URL: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Counter_(CTR) (cit. on p. 129).

[Wik20e] Wikipedia. *Resident set size*. Accessed on September 2020. URL: https://en.wikipedia.org/wiki/Resident_set_size (cit. on p. 61).