EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Coding Theory and Cryptology Group

Secure Sessions for Ad Hoc Multiparty Computation in MPyC

Preparation phase report

Emil Nikolov

Id nr: 0972305
emil.e.nikolov@gmail.com

Supervisor : Dr. ir. L.A.M. (Berry) Schoenmakers

December 12, 2022

Abstract

The field of Secure Multiparty Computation provides methods for jointly computing functions without revealing their private inputs from multple parties. This master thesis assignment focuses on the MPyC framework for MPC and explores various approaches for connecting the parties via the internet. A technical survey was performed in the preparation phase to identify viable techniques and tools to achieve that. Furthermore a test environment dubbed E^3 was developed to support the exploration process that will take place during the implementation phase of the assignment. It is composed of a combination of physical and virtual machines that are able to execute a multiparty computation together using MPyC. It employs several declarative Infrastructure as Code tools to automate the deployment process and make it reproducible. Specifically, Terraform is used for provisioning NixOS virtual machines on the DigitalOcean cloud provider and Colmena is used for remotely deploying software to them. The reference implementation described in this report uses the Tailscale mesh VPN for connectivity, and a number of additional implementations are planned for the next phase of the project.

Contents

List of Abbreviations

E^3 Extensible Evaluation Environment. 3, 4, 5, 6, 8, 9, 11, 14, 19, 21, 22

ACL Access Control List. 8

API Application Programming Interface. 3

CA Certificate Authority. 21

CGNAT Carrier-Grade NAT. 7

CLI Command Line Interface. 1, 20

DDOS Distributed Denial of Service. 9

DERP Designated Encrypted Relay for Packets. 8

DID Decentralized Identifier. 9, 21

DNS Domain Name System. 2

GUI Graphical User Interface. 1

IaC Infrastructure as Code. 2, 5, 21

IP Internet Protocol. 2, 3, 4, 7

IPFS InterPlanetary File System. 22

ISP Internet Service Provider. 2

IT Information Technology. 2

LAN Local Area Network. 2, 7

MPC Secure Multi-party Computation. 1, 2, 3, 21

NAT Network Address Translation. 2, 7, 22

P2P Peer to Peer. 2, 7, 22

SSH Secure Shell Protocol. 21

SSI Self-Sovereign Identity. 9, 21

SSS Shamir's Secret Sharing. 1

STUN Session Traversal Utilities for NAT. 7

TCP Transmission Control Protocol. 2, 7

TLS Transport Layer Security. 7

TOR The Onion Router. 22

UDP User Datagram Protocol. 7

VM Virtual Machine. 4, 5, 6, 11, 14, 21

VPN Virtual Private Network. 7, 22

List of Figures

Chapter 1

Introduction

This report will present the results of the preparation phase of the master thesis project titled "Secure Sessions for Ad Hoc Multiparty Computation in MPyC". The goal of this phase is to gain sufficient insight into the topic, perform some preliminary tasks and propose a plan with well defined goals for the implementation phase of the project.

1.1 Background

Secure Multi-party Computation (MPC) is a set of techniques and protocols for computing a function over the secret inputs of multiple parties without revealing their values, but only the final result. A good overview can be found on Wikipedia[Wik22]. Yao's Millionaires' Problem[Yao82] is one famous example in which a number of millionaires want to know who is richer without revealing their net worths. Other practical applications[LK15] include electronic voting, auctions or even machine learning[Kno+22] where one party's private data can be used as an input for another party's private machine learning model.

The general process is that each party uses a scheme like Shamir's Secret Sharing (SSS) [Sha79] to split its secret input into shares and sends one to each of the other parties. A protocol involving multiple communication rounds and further re-shares of intermediate secret results is used by the parties so that each of them can compute the final result from the shares it has received.

A number of MPC frameworks have been developed for various programming languages and security models. As part of this project we will focus our efforts on **MPyC**[Sch22a; Sch22b] - an opensource MPC Python framework developed primarily at TU Eindhoven, but our results should be applicable to others as well.

To help us determine the types of solutions we need to consider, we can group the potential users of the MPyC framework into three broad categories: casual users, power users and enterprises.

We define **casual users** as people who are used to Windows or Mac, prefer software installers to package managers, Graphical User Interface (GUI) programs rather than Command Line Interface (CLI) based ones and do not feel comfortable with manually modifying their systems or using scripts.

We define **power users** as users who manage a number of personal physical machines and may have some familiarity with Linux, terminals and shell scripting. They are assumed to be able to execute the necessary steps to setup a machine given a guide.

For our purposes we define **enterprises** as companies with operations departments that manage their IT infrastructure and optimize for scale. They usually have large numbers of Linux based servers which are a combination of physical, virtual and container based. Those can be deployed either in the cloud or on premise in an automated way using Infrastructure as Code (IaC) tools.

1.2 Problem description

MPyC supports Transmission Control Protocol (TCP) connections from the Internet Protocol (IP) suite between the MPC participants but it does not currently provide a service discovery mechanism. Before performing a joint computation, all parties must know and be able to reach each other's TCP endpoints - either via a local IP address on a Local Area Network (LAN), or via a public IP address or the Domain Name System (DNS) on the internet. This is not likely to pose a problem for most enterprise users, who are usually already exposing some public services, e.g. their website. However, most casual users who are not Information Technology (IT) experts typically do not own a domain name nor know how to configure a publicly accessible server. Due to the limited supply of addresses supported by IPv4 and the slow adoption of IPv6, most Internet Service Providers (ISPs) do not allocate a public address for each machine in the home networks of their residential customers. Usually only their router has a temporary public address and it utilizes techniques such as Network Address Translation (NAT) in order to enable the other local machines to initiate remote internet connections. However, connections to them cannot be initiated from outside the LAN without manually configuring port forwarding in the router to send the appropriate traffic to the intended machine. This poses some challenges and limits the usability of MPyC in every day scenarios due to the inherently Peer to Peer (P2P) nature of the involved communications.

1.3 Research questions

Based on the problem description above, we formulate the following central research question:

How can MPyC be extended to enable casual users, power users and enterprises with limited prior knowledge of each other to discover each other and perform a secure multiparty computation under diverse networking conditions?

We further identify the following sub-questions:

- *What deployment strategies should be supported in order to accommodate the potential users of MPyC programs?*

Depending on the technical background of a user and their typical computing usage, they might have different expectations for how to execute their part of the joint multiparty computation, e.g. enterprises might expect support for automation tools, while casual users could expect simplicity. There should be some safety (not necessarily security) mechanism for detecting and preventing mistakes where the users are accidentally executing incompatible MPyC programs/versions.

- *What are the most suitable approaches for a party to obtain an identity and prove it to other parties for the purposes of MPC?*

A party's digital identity is a persistent mechanism that allows others to provably track it across digital interactions with the party. An identity can either be issued by a digital authority, e.g. an organization like google, or a country's government or it can be self-issued.

Depending on the method, an identity verification can involve demonstrating a cryptographic proof of ownership of a public key, or separate communication with the digital authority.

- *What mechanisms can be used by the parties to initially get in contact and discover each other's identities?*

Different approaches should be considered based on the types of users and their prior relationships. Some examples could be for companies to publicly post their identities on their websites, end-users who know each other could use social media or group chats and if they do not know each other, they could use an (anonymous) matchmaking service.

- *How can the parties establish communication channels with each other based on the chosen identity solutions under diverse networking conditions?*

As previously mentioned, some parties could be on a home network and not have a public IP, which may require considering approaches that use a mediator.

- *How can the parties communicate securely as part of the MPC execution? To what extent can the parties' privacy be preserved? How efficiently can this be achieved?*

In order for the execution of an MPC protocol to be secure, it is important for the parties to be able to cryptographically verify the identity of a message's original sender and be certain that nobody other than themselves can read it. Solutions that do not reveal physically identifying information such as IP addresses are also interesting to consider. The performance overhead of the security mechanisms should be evaluated.

1.4 Preparation phase scope

During the implementation phase, we will answer the posed research questions of the project after evaluating various connectivity approaches for MPyC. The scope of the preparation phase will cover a technical survey to identify some of the tools that can be used and the development of an Extensible Evaluation Environment (E^3) that will support the evaluation process in the next phase. E^3 must enable fearless experimentation with different implementations of the connectivity layer. E^3 will focus on being reproducible by enterprise and power users while keeping it representative of real world scenarios involving casual users as well.

Below, we formulate our requirements for E^3 and group them in terms of several important characteristics:

- Complexity
  - simple - given the limited time of the preparation phase, E^3 must focus on simplicity, e.g. the easiest to implement connectivity approach should be chosen
  - extensible - E^3 must allow for switching the building blocks during the next phase of the project, e.g. it should be easy to experiment with different connectivity approaches in order to measure and compare their characteristics
- Source code
  - open-source - the source code of the resulting implementation of E^3 must be available in a public repository, e.g. on Github.com
  - no plaintext secrets such as Application Programming Interface (API) keys and passwords should be present in the public repository, but others should be able to easily provide their own secrets in order to use E^3 in their own environment.
- Deployment
  - cross region - the machines should be provisioned in multiple geographical regions in order to be able to observe the effects of varying latency on the system

  - cross platform - in a real world scenario the machines will be controlled by different parties that run various operating systems, hardware architectures and deployed using different tools, e.g. Party A might be an enterprise that uses containers, while Party B is a power user running a few Virtual Machines (VMs) and Party C could be using an ARM-based raspberry pi
  - automated - appropriate tools should be chosen to allow automatically deploying and destroying the runtime environment without manual intervention other than running a minimal set of commands
  - reproducible - it should be easy for others to reproduce the test setup in their own environment
  - disposable - E^3 should be easy to destroy and quickly recreate from scratch at any time; as in the famous DevOps analogy[Bia16], it should be based on machines that are like cattle rather than pets
- Connectivity
  - identity - it must be possible for the machines to communicate based on a long-lived identity rather than a potentially temporary IP address.
  - secure - a message sent by a party must be readable only by its intended targets.
  - authenticated - a party must be able to determine which party a message was sent by
  - private - no more information than strictly necessary should be revealed about a party. Depending on the method of communication, it may be necessary to choose a tradeoff or introduce a tuning parameter between performance and privacy.

Chapter 2

Technical Survey

In this chapter we will perform a high level survey of the available tools and approaches that could be used for E^3 and select ones that fit our requirements. In the next chapter we will go more in depth and cover the implementation details using those tools.

Since we need to design for a heterogeneous runtime environment, we need to choose building blocks that are compatible with as many scenarios as possible while also keeping the complexity low.

2.1 Deployment

Our primary users are enterprises and power users. Enterprises can employ a variety of IaC tools in their infrastructure management process:

- provisioning - Terraform[Has22], Cloud Formation[AWS22], etc.
- deployment automation - Ansible[Red22], Puppet[Pup22], Chef[Che22], etc.
- container orchestration - Docker Swarm[Doc22], Kubernetes[Kub22], etc.

According to our definitions, power users typically use physical machines while enterprises can use both virtual machines and container orchestration tools. Based on our requirements for E^3 we need a cross region deployment. VMs can be automatically provisioned across different regions in the cloud using IaC tools. Once provisioned, a VM is usually managed via an automation tool that executes a set of deployment steps over SSH. Those deployment steps can be adapted to physical machines so that power users can make use of them.

Kubernetes is used for dynamically scaling a large number of long-running processes across a cluster of VMs within the same geographic region. Enterprises may wish to run MPyC programs in a Kubernetes cluster and might benefit from an example of doing so. But for the purposes of E^3, it does not provide sufficient benefits compared to using VMs directly, while it adds complexity in terms of deploying multiple clusters across regions and adding a cross cluster communication mechanism.

Based on this analysis, we choose to base E^3 on a combination of VMs deployed in the cloud and a set of personal devices owned by the authors - a Linux laptop, a Windows desktop with Windows Subsystem for Linux, and an ARM Raspberry Pi 2 to serve as an example of both enterprises and individual power users.

IaC tools use specifications that are either imperative or declarative. **Imperative** specifications describe the steps needed to be executed for the infrastructure to reach the desired state, while **declarative** specifications describe the desired final state and let the tool worry

about how to get there. Imperative tools are more likely to suffer from *configuration drift* - the infrastructure state might become out of sync with the specification due to either manual changes or left-over state from previously applied specifications. On the other hand, if something is removed from a declarative specification, when it gets applied, the corresponding resources will also be removed from the infrastructure. In addition, declarative tools are idempotent - applying the same specification multiple times in a row does not change the state. Therefore in order to achieve high reproducibility, we will prefer declarative tools to imperative ones where possible.

On Linux, software is usually installed via package managers. Most of the popular Linux distributions such as Ubuntu, Debian, Fedora, Arch use package managers that only support automating this process via imperative shell scripts rather than a declarative specification. Additionally those do not offer an easy way of specifying and locking the required software versions to a known good configuration that can be reproduced in the future. NixOS on the other hand is based on the declarative Nix package manager which does support version locking via its **flakes** feature. This is why we choose to use the NixOS operating system for our VMs.

Most of the popular application deployment tools such as Ansible, Chef and Puppet are either imperative or have limited support for declarative specifications. Fortunately, the NixOS ecosystem, offers a number of deployment tools that can apply a declarative specification on remote hosts:

- NixOps[Nix22a; Nix22b] - official tool
- Colmena [Li22a; Li22b]
- morph [Bib22]
- deploy-rs [Ser22]

NixOps is the official deployment tool for NixOS but it was being redesigned at the time of writing. The new version was not complete yet and lacked documentation, while the old one was no longer being supported and depended on a deprecated version of Python. The rest of the tools were still actively maintained. Colmena was the best fit for our use case because it supported both flakes and parallel deploys, while morph lacked support for flakes and deploy-rs could not deploy to multiple hosts in parallel.

The declarative tools that were considered for provisioning were Terraform, Pulumi and Cloud Formation. Cloud Formation only works on AWS which would prevent us from using it with other cloud providers. Pulumi is a newer and less proven tool compared to Terraform, which the authors had more experience with. Therefore our choice was to use Terraform.

We decided to use DigitalOcean as a cloud provider because they are supported by Terraform and offered free credits for educational use.

2.2 Connectivity

There are a number of approaches for communication between our host machines. During the preparation phase of the project we will perform a high level exploration of our options and summarize them. For E^3 we will initially use the simplest to implement one. During the implementation phase of the project we will go more in depth and implement more approaches and analyze how they compare to each other in practice.

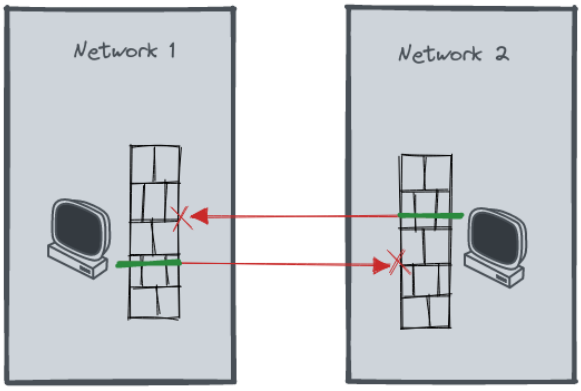

Figure 2.1: Two parties behind separate NATs

2.2.1 Virtual Private Networks (VPNs)

Virtual Private Networks (VPNs) are commonly used for securely connecting machines from different LANs. They provide software emulation of a network device on the operating system level and allow other software to transparently use the functionality of the IP suite without requiring extra changes. Traditional VPNs such as OpenVPN[Ope22] use a centralized service that all (encrypted) client communications must pass through. This introduces a single point of failure and a potential bottleneck that might negatively impact the performance of the multi-party computations due to their P2P nature.

On the other hand, mesh VPNs such as Tinc[Sli22], Tailscale[Tai] and Nebula[Def22] utilize direct P2P links between the clients for the data traffic. Authentication, authorization and traffic encryption are performed using certificates based on public key cryptography. As we mentioned in the introduction chapter, the devices in a typical home network can only initiate connections to public endpoints (via NAT) but cannot be discovered from outside their LAN. This poses a challenge when two parties who want to communicate via a direct link are both behind separate NATs 2.1 and neither can be contacted by the other one first. Mesh VPNs solve this issue via NAT traversal techniques such as User Datagram Protocol (UDP) hole punching based on concepts from Session Traversal Utilities for NAT (STUN). The machines of each party can contact a public STUN server 2.2, which will note what IP addresses the connections come from and inform the parties. Since the parties initiated the connection to the STUN server, their routers will keep a mapping between their local IP addresses and the port that was allocated for the connection in order to be able to forward the incoming traffic. Those "holes" in the NATs were originally intended for the STUN server, but mesh VPNs use the stateless "fire and forget" UDP protocol for their internal communication, which does not require nor provides a mechanism for the NATs to verify who sent a UDP packet. With most NATs, this is enough to be able to (ab)use the "punched holes" for the purpose of P2P traffic from other parties. Mesh VPNs implement the stateful TCP and Transport Layer Security (TLS) protocols on top of UDP and expose an regular network interface to the other programs, keeping them shielded from the underlying complexities. Other NAT implementations such as Symmetric NAT and Carrier-Grade NATs (CGNATs) can be more difficult to "punch through" due to their more complex port mapping strategies. In those cases, establishing P2P connections might involve guess work or even fail and require falling

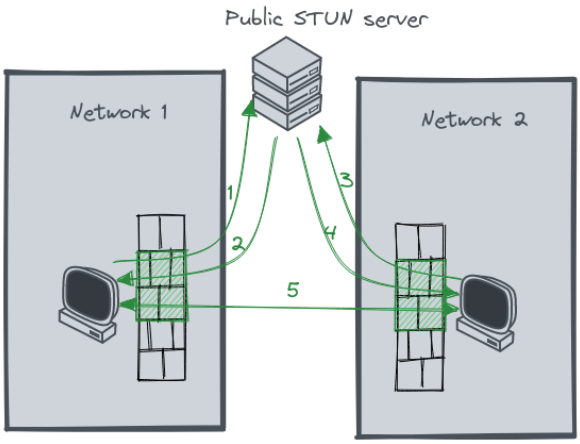

Figure 2.2: NAT traversal via STUN

back to routing the (encrypted) traffic via another party or service.

Now that we have a general understanding of how mesh VPNs work, let us see how Tinc, Tailscale and Nebula compare. All three are open-source, with the exception of Tailscale's coordination service which handles the peer discovery and identity management. Headscale [Fon22] is a community driven open-source alternative for that component. Tinc is the oldest of the three but has a relatively small community. It is mainly developed by a single author and appears to be more academic than industry motivated. Nebula and Tailscale are both business driven. Tailscale was started by a number of high profile ex-googlers and is the most end-user focused of the three, providing a service that allows people to sign up using a variety of identity providers including google, microsoft, github and others. They also provide an Admin console that allows a user to easily add their personal devices to a network or share them with others. It also has support for automation tools like Terraform for creating authorization keys and managing an Access Control List (ACL) based firewall. Nebula was originally developed at the instant messaging company Slack to create overlay networks for their cross region cloud infrastructure, but the authors later started a new company and are currently developing a user-centric platform similar to Tailscale's. Nebula is more customizable than Tailscale and since it is completely open-source it can be adapted to different use cases, but it is also more involved to set up. A certificate authority needs to be configured for issuing the identities of the participating hosts. Furthermore, publicly accessible coordination servers need to be deployed to facilitate the host discovery. Tailscale employs a distributed relay network of Designated Encrypted Relay for Packets (DERP) servers, while Nebula can be configured to route via one of the other peers in the VPN.

We decided to use Tailscale for the initial implementation of E^3 because it has all the necessary features to support networked MPC while also being the easiest one to set up as it does not require any extra services to be deployed.

We will now briefly mention some additional approaches we looked into that may be a good starting point for the next phase of the project.

2.2.2 Decentralized Identifiers (DIDs) and DIDComm

Self-Sovereign Identity (SSI) is a way of managing a digital identity that emphasises an individual's ownerhip and control over their personal data. In contrast with more traditional methods, SSI does not rely on a third party such as a government or an organization to issue identities - people issue their own identities, usually in the form of an asymmetric key-pair. Decentralized Identifiers (DIDs)[Spo+22] are a form of SSI that recently reached W3C Recommendation Status and DIDComm[Fou22] is a set of communication protocols based on DIDs. Its main design goals are to be private, secure, decentralized, transport agnostic and routable. Its primary concerns are with the message formats, cryptograhic algorithms and processes that enable identity owners to find each other and interact digitally based on their identities rather than TCP concepts like IP addresses. One thing we noticed was that the initial version of DIDComm does not support stateful sessions and therefore all messages need to be both encrypted with the recepient's public key and signed by the sender's private key. This will likely cause performance issues in the MPC setting because it usually involves a large number of small messages containing the secret shares of the parties. In order for DIDComm to be usable for MPyC we would likely have to implement a TLS-like protocol on top of it that supports sessions.

2.2.3 The Onion Router (TOR)

Two machines need to know each other's IP addresses in order to be able to interact via the internet. An IP address can reveal details about a person's location or be used to launch a Distributed Denial of Service (DDOS) attack against them. Additionally, nearby attackers could be listening to their traffic and tracking their internet behaviour, which may be undesirable depending on a person's privacy requirements. TOR uses a network of relays to obfuscate the communication route between two parties. The original sender prepares a multi-layered message, where each layer is encrypted for a specific relayer. When one of them receives a message, they only know who was the previous link and after decrypting their part of the message, they know the next link. They do not know who was the original sender and who is the final destination. The privacy comes at the cost of performance. Additionally, TOR has the concept of Onion Services, which receive an address under the .onion pseudo top level domain and correspond to a public key (e.g. vww6ybal4bd7szmgncyruucpgfkqahzddi37ktceo3ah7ngmcopnpyyd.onion). It allows two way privacy preserving communications. A concept similar to TOR can be optionally incorporated in MPyC for the cases when privacy is essential. It is interesting to measure the performance impact on MPyC computations when routed via a TOR-like relay network.

2.3 Summary

In this chapter we compared a number of potential building blocks for E^3 and made some choices informed by our requirements. Specifically, we will use Terraform for provisioning Virtual Machines running NixOS on DigitalOcean and Colmena for deploying to them. Our initial implementation will use Tailscale as the connectivity layer due to its ease of use. In the next phase of the project, we plan to explore solutions based on Nebula, DIDComm, TOR and combinations of the above.

Chapter 3

Implementation details

In this chapter we will cover the implementation of E^3, which can be found on Github[1]. It is a fork of MPyC[2] that adds deployment capabilities. We will now discuss the more critical parts of the implementation.

3.1 Reproducible development of MPyC

As previously discussed, the VMs of E^3 run the NixOS distribution of Linux, which is based on the declarative package manager Nix. One of its benefits is that it can also be used to declaratively manage the dependencies of a software project via its development shells feature. Normally such a project would have to explain in its readme how to install and configure all of the extra tools that are needed for working with it. On the other hand, with Nix, we can run the command `nix develop` in a directory containing a declarative specification in a flake.nix file. This will open a temporary shell environment and install the specified versions of the dependencies. Exiting the shell will uninstall them. This process makes it easy to work on projects that require conflicting versions of packages. To achieve this, nix does the following:

- it places each build result under `/nix/store/`, in a sub-directory prefixed by the hash of its inputs, e.g. `/nix/store/2ispfz80kmwrsvwndxkxs56irn86h43p-bash-5.1-p16/`
- nix opens a new shell with a modified `PATH` environment variable that includes the nix store path that contains the new package.

There are tools like nix-direnv that take dev shells a step further by automatically loading/unloading the specified dependencies when entering/leaving a directory that contains a dev shell specification.

[1] https://github.com/e-nikolov/mpyc
[2] https://github.com/lschoe/mpyc

The entrypoint for Nix in our MPyC fork is the flake.nix[3] file. A simplified version can be seen below:

```
{
  description = "MPyC flake";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-unstable";
  };

  outputs = inputs@{ self, nixpkgs, ... }:
    let
      # import the derivation of mpyc and all of its python dependencies
      mpyc-demo = (import ./nix/mpyc-demo.nix { inherit pkgs; dir = ./.; });

      pkgs = import nixpkgs {
        system = "x86_64-linux";
      };
    in
    {
      devShell.x86_64-linux = pkgs.mkShell {
        shellHook = ''
          export PYTHONPATH=./ # enable editable mode for mpyc
        '';

        nativeBuildInputs = [
          pkgs.curl
          pkgs.jq
          pkgs.colmena
          pkgs.pssh
          (pkgs.terraform.withPlugins
            (tp: [
              tp.digitalocean
              tp.null
              tp.external
              tp.tailscale
              tp.random
            ]))
          mpyc-demo
        ];
      };
    };
}
```

The flake is functionally pure in the sense that all external inputs are explicitly declared in the inputs section and their hashes are kept in a `flake.lock` file. In our example, the only input is `nixpkgs` - a community managed repository containing the nix build expressions for more than 80 000 packages. When a nix command uses the file for the first time, the latest revision of the nixos-unstable branch of the git repository will be fetched and its contents will be hashed and placed in the flake.lock. Further executions will reuse the revision from the lock file and verify that the resulting hash matches the original one. The lock file can be updated via the `nix flake update` command. The output section contains the `devShell.x86_64-linux` attribute which declares the packages required to work with the project. Specifically, it needs

[3] https://github.com/e-nikolov/mpyc/blob/master/flake.nix

the nix packages for curl, jq, colmena, pssh and terraform with a number of plugins. Finally it also builds the `mpyc-demo` package which contains python and all python dependencies needed by MPyC. Its specification is imported from the `mpyc-demo.nix` file:

```
{ pkgs, dir }:
(pkgs.poetry2nix.mkPoetryEnv {
    python = pkgs.python3;
    projectDir = dir;

    extraPackages = (ps: [
      (pkgs.python3Packages.buildPythonPackage
        {
          name = "mpyc";
          src = dir;
        })
    ]);

    overrides = pkgs.poetry2nix.overrides.withDefaults (
      self: super: {
        gmpy2 = pkgs.python3Packages.gmpy2;
      }
    );
  })
```

MPyC uses the python specific dependency management tool poetry[Poe22] and the poetry2nix package dynamically generates nix expressions from its configuration in `pyproject.toml`

```
[tool.poetry]
name = "mpyc"
version = "0.8.8"
description = "MPyC for Multiparty Computation in Python"
authors = ["Berry Schoenmakers <berry@win.tue.nl>"]
readme = "README.md"
packages = [{include = "demos"}]

[tool.poetry.dependencies]
python = "^3.10"
qrcode = "^7.3.1"
numpy = "^1.23.4"
gmpy2 = "^2.1.2"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

There was an issue with the gmpy2 library when building it via poetry2nix, but fortunately we could override it with the version already present in nixpkgs.

Using this setup we can now go to the root directory of MPyC and run the `nix develop` command, which will automatically download, build and install all of our dependencies in a temporary shell. We are then ready to make changes to MPyC or locally run the demos, e.g. via `python ./demos/secretsanta.py`.

3.2 Building a NixOS image for DigitalOcean

As previously discussed we will deploy E^3 on DigitalOcean droplets - their name for VMs. They do not provide an official NixOS image, but we can build our own using nix.

```
## flake.nix

{
  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-unstable";
  };

  outputs = inputs@{ self, nixpkgs, ... }:
    let
      mpyc-demo = (import ./nix/mpyc-demo.nix { inherit pkgs; dir = ./.; });

      pkgs = import nixpkgs {
        system = "x86_64-linux";
      };

      digitalOceanConfig = import ./nix/digitalocean/image.nix {
        inherit pkgs;
        extraPackages = [ mpyc-demo ];
      };
    in
    {
        packages.digitalOceanImage = (pkgs.nixos digitalOceanConfig).digitalOceanImage;
    };
}
```

```
## nix/digitalocean/image.nix

{ pkgs, extraPackages ? [ ], ... }:
{
  imports = [ "${pkgs.path}/nixos/modules/virtualisation/digital-ocean-image.nix" ];
  system.stateVersion = "22.11";

  environment.systemPackages = with pkgs; [
    jq
  ] ++ extraPackages;

  services.tailscale.enable = true;

  networking.firewall = {
    enable = true;
    checkReversePath = "loose";
    trustedInterfaces = [ "tailscale0" ];
  };
}
```

The image is based on the default version provided by nixpkgs and adds some extra packages and configurations. It:

- enables the tailscale service so that we can easily configure them to join the same tailscale network;

- configures the firewall to allow tailscale traffic;
- includes the `mpyc-demo` package we made earlier for the development shell.

The image will likely be built only once, but it is still useful to have even an outdated version of the demo baked into it as it helps us avoid having to build all of the python dependencies while deploying later.

Running `nix build .#packages.digitalOceanImage` creates a `.qcow2.gz` formatted image file that can be imported into DigitalOcean.

3.3 Building a NixOS image for RaspberryPi

We can build a NixOS image for a RaspberryPi4 with a similar nix expression:

```
{
  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-unstable";
  };

  outputs = inputs@{ self, nixpkgs, ... }:
    let
      mpyc-demo = (import ./nix/mpyc-demo.nix { inherit pkgs; dir = ./.; });

      pkgs = import nixpkgs {
        system = "aarch64-linux";
      };
    in
    {
        packages.raspberryPi4Image = (pkgs.nixos ({ config, ... }: {
            system.stateVersion = "22.11";
            imports = [
              ("${pkgs.path}/nixos/modules/installer/sd-card/sd-image-aarch64-installer.
              ↪  nix")
            ];

            environment.systemPackages = [
                mpyc-demo
            ];
        })).sdImage;
    };
}
```

The build command has to be executed on an ARM64 based processor in order to succeed. This can be achieved either via emulation with qemu binfmt or via a virtual machine. When already running NixOS as a host, all that is required is to add `extra-platforms = aarch64-linux` to the `/etc/nixos/nix.conf` file.

3.4 Provisioning via Terraform

We will provision DigitalOcean droplets with Terraform from the VM image created earlier. We need to provide it with a DigitalOcean authorization key in the `DIGITALOCEAN_TOKEN` environment variable so it can use their API on our behalf.

Importing the image

The snippet below handles the upload of our NixOS image.

```
variable "nixos-image-path" {
  type    = string
  default = "../../bin/image/nixos.qcow2.gz"
}

resource "digitalocean_spaces_bucket" "tf-state" {
  name   = "mpyc-tf-state"
  region = "ams3"

  lifecycle {
    prevent_destroy = true
  }
}

resource "digitalocean_spaces_bucket_object" "nixos-image" {
  region = digitalocean_spaces_bucket.tf-state.region
  bucket = digitalocean_spaces_bucket.tf-state.name
  key    = basename(var.nixos-image-path)
  source = var.nixos-image-path
  acl    = "public-read"
  etag   = filemd5(var.nixos-image-path)
}

resource "digitalocean_custom_image" "nixos-image" {
  name    = "nixos-22.11"
  url     = "https://${digitalocean_spaces_bucket.tf-state.bucket_domain_name}/${digital
  ↪  ocean_spaces_bucket_object.nixos-image.key}"
  regions = local.all_regions
  tags    = ["nixos"]

  lifecycle {
    replace_triggered_by = [
      digitalocean_spaces_bucket_object.nixos-image
    ]
  }
}
```

The DigitalOcean API only supports importing images from a URL, so we first need to upload the image to a publicly accessible location. For that purpose, the snippet above first provisions a Bucket within Spaces - DigitalOcean's cloud storage solution and uploads the image there. After that, the `digitalocean_custom_image` will import the image from the URL generated by the bucket.

Generating hostnames

The snippet below starts with a specification of how many machines per region we want to have and transforms it into a list of descriptive ids (e.g. `mpyc-demo-ams3-0-15e53f39`) that will be used as host names so we can easily distinguish the machines when they start communicating with each other.

```
locals {
  node_definitions = var.DESTROY_NODES != "" ? [
    { region = "ams3", num = 0 },
    { region = "sfo3", num = 0 },
    { region = "nyc3", num = 0 },
    { region = "sgp1", num = 0 },
    ] : [
    { region = "ams3", num = 3 },
    { region = "sfo3", num = 1 },
    { region = "nyc3", num = 1 },
    { region = "sgp1", num = 1 },
  ]

  nodes_expanded = flatten([
    for node in local.node_definitions : [
      for i in range(node.num) :
      merge(node, {
        name = "mpyc-demo--${node.region}-${i}"
      })
    ]
  ])

  nodes = {
    for node in local.nodes_expanded :
    node.name => merge(node, {
      hostname = "${node.name}-${random_id.mpyc-node-hostname[node.name].hex}"
    })
  }
}
```

Provisioning the hosts

The snippet below provisions the droplets in the specified regions and then using Tailscale's terraform provider creates auth keys for the machines, copies them to the machines and configures them to join the tailscale network. When the droplets are being destroyed, the provisioner will remove the nodes from the network.

```

resource "digitalocean_droplet" "mpyc-node" {
  for_each = local.nodes

  image    = digitalocean_custom_image.nixos-image.id
  name     = each.value.hostname
  region   = each.value.region
  size     = "s-1vcpu-1gb"
  ssh_keys = [for key in digitalocean_ssh_key.ssh-keys : key.fingerprint]

  connection {
    type = "ssh"
    user = "root"
    host = self.ipv4_address
  }

  provisioner "remote-exec" {
    inline = [
```

```
      "mkdir -p /var/keys/",
      "echo ${tailscale_tailnet_key.keys.key} > /var/keys/tailscale",
      "tailscale up --auth-key file:/var/keys/tailscale"
    ]
  }

  provisioner "remote-exec" {
    when = destroy
    inline = [
      "tailscale logout"
    ]
  }

  lifecycle {
    replace_triggered_by = [
      tailscale_tailnet_key.keys
    ]
  }
}

resource "tailscale_tailnet_key" "keys" {
  reusable      = true
  ephemeral     = true
  preauthorized = true
}
```

Interfacing with other tools

This snippet outputs the hostnames of the provisioned droplets so they can be picked up by other tools. For example Colmena will read them from a json file so it can deploy further software changes to them.

```


output "hosts-colmena" {
  value = { for node in local.nodes : node.hostname => {} }
}

output "hosts-pssh" {
  value = join("", [for node in local.nodes : "root@${node.hostname}\n"])
}
```

3.5 Colmena deployment

Whenever we need to update the NixOS configuration of our VMs, we could rebuild the image and re-provision them, but this would be slow. Instead we use Colmena to deploy and apply the new configuration to all existing VMs. It uses the same `digitalOceanConfig` attribute we created for the NixOS image:

```
{
  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-unstable";
  };
```

```
5
6     outputs = inputs@{ self, nixpkgs, ... }:
7       let
8         mpyc-demo = (import ./nix/mpyc-demo.nix { inherit pkgs; dir = ./.; });
9
10        pkgs = import nixpkgs {
11          system = "x86_64-linux";
12        };
13
14        digitalOceanConfig = import ./nix/digitalocean/image.nix {
15          inherit pkgs;
16          extraPackages = [ mpyc-demo ];
17        };
18      in
19      {
20        packages.colmena = {
21          meta = {
22            nixpkgs = pkgs;
23          };
24          defaults = digitalOceanConfig;
25        } // builtins.fromJSON (builtins.readFile ./hosts.json);
26      };
27  }
```

This allows us to quickly make changes to the NixOS configuration, deploy it via Colmena and once we are satisfied with it, we can choose to rebuild the image so that new machines get provisioned with all of our changes baked in.

3.6 Runtime execution

We have identified the tools we will use to deploy E^3 and how the host machines will communicate. What remains is to determine how to run a joint computation on the hosts. When running such an experiment, it would be desirable to be able to iterate quickly. Colmena can be used to deploy a new version of the whole operating system, but it would be unnecessary to rebuild all dependencies every time we want to run a command. Therefore we decided to use two additional tools:

- *prsync* - a variant of the popular *rsync* utility that can additively sync the contents of a directory to multiple remote hosts
- *pssh* - a tool for executing an ssh command on many hosts in parallel

An example execution looks like this:

```
1  prsync -h hosts.pssh -zarv -p 4 ./ /root/mpyc
2  pssh -h hosts.pssh -iv -o ./logs/$t "cd /root/mpyc && ./prun.sh"
```

It loads the hostnames from the `hosts.pssh` file that was previously generated by terraform and syncs the current state of the mpyc directory. The second line will execute the `prun.sh` script on each host.

An example of a possible `prun.sh` script:

```
1  #!/bin/sh
2
3  MAX_PARTIES=600
```

```
4  hosts="./hosts.pssh"
5  port=11599
6
7  i=0
8  MY_PID=-1
9
10 args=""
11 while IFS= read -r line
12 do
13     if [ $i -ge $MAX_PARTIES ]
14     then
15         break
16     fi
17     if [ -z "$line" ]
18     then
19         break
20     fi
21
22     host=${line#"root@"}
23
24     if [ "$host" = "$HOSTNAME" ]
25     then
26         MY_PID=$i
27     fi
28     ((i = i + 1))
29
30     args+=" -P $host:$port"
31 done < "$hosts"
32
33 if [ $MY_PID = -1 ]
34 then
35     echo Only $i parties are allowed. $HOSTNAME will not participate in this MPC session
36 else
37
38 cmd="python ./demos/secretsanta.py 3 --log-level debug \
39     -I ${MY_PID} \
40     ${args}"
41
42 echo $cmd
43 $cmd
44
45 fi
```

Each host runs the same script that builds the arguments for the `secretsanta.py` demo of MPyC. Each party determines its own Party ID based on the index at which its hostname appears in the `hosts.pssh` file.

3.7 Secrets

The implementation expects that secrets are supplied as environment variables with the specific mechanism being left up to the user. We are currently keeping the secret values in the 1Password manager and use its CLI to populate the environment variables at runtime, e.g. via `op run - make deploy`.

Chapter 4

Conclusions

In this report we presented the results of the preparation phase for the master thesis assignment "Secure Sessions for Ad Hoc Multiparty Computation in MPyC". We developed an Extensible Evaluation Environment (E^3) for the purpose of creating ad hoc networks of host machines that perform Secure Multi-party Computations (MPCs) in hybrid scenarios involving both cloud and physical machines. E^3 makes extensive use of declarative IaC tools in order to achieve highly reproducible deployments in an automated way. We provided a reference implementation that makes use of the Tailscale mesh VPN that creates a network of RaspberryPis and cloud VMs on DigitalOcean. The cloud provisioning is defined declaratively using Terraform and allows to define a set of host machines across the regions supported by DigitalOcean (e.g. Amsterdam, New York City, etc) and automatically add them to a shared Tailscale network. The machines run NixOS - a declarative and highly reproducible Linux distribution while Colmena is used to declaratively manage the software installed on them via Secure Shell Protocol (SSH). The tools `prsync` and `pssh` are used to run MPyC demos in parallel on the deployed hosts.

Implementation phase planning

During the next phase of the thesis assignment, we plan to implement various connectivity approaches for E^3's host machines and analyse their suitability for MPyC.

The following is a list of high level tasks that we plan to carry out as part of the implementation phase:

- replace the proprietary Tailscale coordination service from our reference implementation of E^3 with the open-source self-hosted alternative Headscale[Fon22]
- develop a network overlay for E^3 based on the Nebula mesh VPN. Nebula only provides a way to manually perform the initial setup, so our implementation should add a way to automatically:
  - allocate virtual IP addresses for the hosts
  - generate identity certificates using the Nebula Certificate Authority (CA)
  - distribute the certificates among the hosts
- develop network overlays for E^3 that incorporate parts of the mesh VPN implementations but with alternative identity management approaches:
  - using a CA that is managed jointly using MPC
  - using a form of SSI such as DIDs
- implement a network overlay for E^3 based on DIDComm

- explore options for enhancing the DIDComm implementation to:
  - support sessions - the DIDComm protocol is currently stateless and uses a new asymmetric key for each message, which negatively impacts performance
  - employ NAT traversal techniques similar to mesh VPNs
- implement a privacy mechanism for E^3 based on The Onion Router (TOR) in order to prevent leaking sensitive information like which parties are communicating with each other and their IP addresses
- investigate if we can apply ideas from the P2P implementations in other software like the Ethereum[Ethow; Woo22] blockchain and the InterPlanetary File System (IPFS) [IPF]
- analyse and compare all of the above implementations in terms of:
  - security
  - performance
  - ease of use
  - privacy
- compare E^3 to other work related to deploying MPC such as the Carbyne stack[Gmb22]

Bibliography

[AWS22] AWS. *AWS CloudFormation Documentation*. 2022. URL: https://docs.aws.amazon.com/cloudformation (visited on 11/28/2022) (cit. on p. 5).

[Bia16] Randy Bias. *The History of Pets vs Cattle and How to Use the Analogy Properly*. Cloudscaling. Sept. 29, 2016. URL: http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/ (visited on 12/05/2022) (cit. on p. 4).

[Bib22] Dansk BiblioteksCenter. *Morph*. DBC, Nov. 30, 2022. URL: https://github.com/DBCDK/morph (visited on 12/04/2022) (cit. on p. 6).

[Che22] Chef. *Chef Documentation*. 2022. URL: https://docs.chef.io/ (visited on 11/28/2022) (cit. on p. 5).

[Def22] Defined. *Nebula: Open Source Overlay Networking | Nebula Docs*. 2022. URL: https://docs.defined.net/docs/ (visited on 12/01/2022) (cit. on p. 7).

[Doc22] Docker. *Swarm Mode Overview*. Docker Documentation. Nov. 25, 2022. URL: https://docs.docker.com/engine/swarm/ (visited on 11/28/2022) (cit. on p. 5).

[Ethow] Ethereum. *Ethereum Development Documentation | Ethereum.Org*. now. URL: https://ethereum.org/en/developers/docs/ (visited on 12/10/2022) (cit. on p. 22).

[Fon22] Juan Font. *Juanfont/Headscale*. Dec. 6, 2022. URL: https://github.com/juanfont/headscale (visited on 12/06/2022) (cit. on pp. 8, 21).

[Fou22] Digital Identity Foundation. *DIDComm Messaging Specification v2 Editor's Draft*. 2022. URL: https://identity.foundation/didcomm-messaging/spec/ (visited on 12/06/2022) (cit. on p. 9).

[Gmb22] Robert Bosch GmbH. *Carbyne Stack*. Carbyne Stack, Dec. 5, 2022. URL: https://github.com/carbynestack/carbynestack (visited on 12/10/2022) (cit. on p. 22).

[Has22] Hashicorp. *Terraform Documentation*. Terraform | HashiCorp Developer. 2022. URL: https://developer.hashicorp.com/terraform (visited on 11/28/2022) (cit. on p. 5).

[IPF] IPFS. *IPFS Documentation | IPFS Docs*. URL: https://docs.ipfs.tech/ (visited on 12/10/2022) (cit. on p. 22).

[Kno+22] Brian Knott et al. *CrypTen: Secure Multi-Party Computation Meets Machine Learning*. Sept. 15, 2022. DOI: 10.48550/arXiv.2109.00984. arXiv: 2109.00984 [cs]. URL: http://arxiv.org/abs/2109.00984 (visited on 12/04/2022) (cit. on p. 1).

[Kub22] Kubernetes. *Kubernetes Documentation*. Kubernetes. 2022. URL: https://kubernetes.io/docs/home/ (visited on 11/28/2022) (cit. on p. 5).

[Li22a] Zhaofeng Li. *Colmena*. Dec. 4, 2022. URL: https://github.com/zhaofengli/colmena (visited on 12/04/2022) (cit. on p. 6).

[Li22b] Zhaofeng Li. *Reference - Colmena (Unstable)*. 2022. URL: https://colmena.cli.rs/unstable/reference/ (visited on 12/04/2022) (cit. on p. 6).

[LK15] Peeter Laud and Liina Kamm, eds. *Applications of Secure Multiparty Computation*. Cryptology and Information Security Series volume 13. Amsterdam, Netherlands: Ios Press, 2015. 253 pp. ISBN: 978-1-61499-531-9 (cit. on p. 1).

[Nix22a] NixOps. *NixOps*. Nix/Nixpkgs/NixOS, Dec. 3, 2022. URL: https://github.com/NixOS/nixops (visited on 12/04/2022) (cit. on p. 6).

[Nix22b] NixOps. *NixOps User's Guide*. 2022. URL: https://hydra.nixos.org/build/115931128/download/1/manual/manual.html (visited on 12/04/2022) (cit. on p. 6).

[Ope22] OpenVPN. *Community Resources*. OpenVPN. 2022. URL: https://openvpn.net/community-resources/ (visited on 11/30/2022) (cit. on p. 7).

[Poe22] Poetry. *Introduction | Documentation | Poetry - Python Dependency Management and Packaging Made Easy*. Dec. 7, 2022. URL: https://python-poetry.org/docs/ (visited on 12/07/2022) (cit. on p. 13).

[Pup22] Puppet. *Docs | Puppet*. 2022. URL: https://puppet.com/docs/ (visited on 11/28/2022) (cit. on p. 5).

[Red22] Redhat. *Ansible Documentation*. 2022. URL: https://docs.ansible.com/ (visited on 11/28/2022) (cit. on p. 5).

[Sch22a] Berry Schoenmakers. *MPyC Homepage*. MPyC. 2022. URL: https://win.tue.nl/~berry/mpyc (visited on 11/29/2022) (cit. on p. 1).

[Sch22b] Berry Schoenmakers. *MPyC Multiparty Computation in Python*. Nov. 28, 2022. URL: https://github.com/lschoe/mpyc (visited on 11/29/2022) (cit. on p. 1).

[Ser22] Serokell. *Serokell/Deploy-Rs*. Serokell, Dec. 2, 2022. URL: https://github.com/serokell/deploy-rs (visited on 12/04/2022) (cit. on p. 6).

[Sha79] Adi Shamir. "How to Share a Secret". In: *Communications of the ACM* 22.11 (Nov. 1, 1979), pp. 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: https://doi.org/10.1145/359168.359176 (visited on 11/29/2022) (cit. on p. 1).

[Sli22] Guus Sliepen. *Tinc Docs*. Nov. 30, 2022. URL: https://www.tinc-vpn.org/docs/ (visited on 11/30/2022) (cit. on p. 7).

[Spo+22] Manu Sporny et al. *Decentralized Identifiers (DIDs) v1.0*. July 19, 2022. URL: https://www.w3.org/TR/did-core/ (visited on 12/06/2022) (cit. on p. 9).

[Tai] Tailscale. *Tailscale*. Tailscale. URL: https://tailscale.com/kb/ (visited on 11/30/2022) (cit. on p. 7).

[Wik22] Wikipedia. *Secure Multi-Party Computation*. In: *Wikipedia*. Dec. 6, 2022. URL: https://en.wikipedia.org/w/index.php?title=Secure_multi-party_computation&oldid=1125968812 (visited on 12/12/2022) (cit. on p. 1).

[Woo22] Dr Gavin Wood. "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER". In: (Oct. 24, 2022), p. 41 (cit. on p. 22).

[Yao82] Andrew C. Yao. "Protocols for Secure Computations". In: 23rd Annual Symposium on Foundations of Computer Science (Sfcs 1982). IEEE Computer Society, Nov. 1, 1982, pp. 160–164. DOI: 10.1109/SFCS.1982.88. URL: https://www.computer.org/csdl/proceedings-article/focs/1982/542800160/12OmNyUnEJP (visited on 12/04/2022) (cit. on p. 1).