EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Coding Theory and Cryptology Group

Secure Dynamic Setup for MPyC Sessions to Support (Ad Hoc) Multiparty Computation

Preparation phase report

Emil Nikolov

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

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

December 5, 2022

Abstract

Abstract

Contents

Chapter 1

Introduction

This report will present the results of the preparation phase of the master thesis project titled Secure Dynamic Setup for MPyC Sessions to Support (Ad Hoc) Multiparty Computation. 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)[**devreedeAssortedAlgorithmsProtocols2020**] 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. Yaos Millionaires Problem[**yaoProtocolsSecureComputations1982**] is one famous example in which a number of millionaires want to know who is richer without revealing their net worths. Other practical applications include electronic voting, auctions or even machine learning[**knottCrypTenSecureMultiParty20** where one partys private data can be used as an input for another partys private machine learning model.

The basic operating procedure is that each party splits its secret input into shares using a scheme such as Shamir's Secret Sharing (SSS) [**shamirHowShareSecret1979**] and sends one to each of the rest. A protocol involving multiple communication rounds and further reshares 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**[**mpycSource**; **mpycHome**] - an opensource MPC Python framework developed primarily at TU Eindhoven, but our results should be applicable to others as well.

We broadly categorize the potential users of the MPyC framework as 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 InterfaceInfrastructure as Code (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)[**cerfProtocolPacketNetwork1974**] 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 others 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 partys 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 countrys government or it can be self-issued. Depending on the method, a 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 others 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 dont 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 messages original sender and be certain that nobody other than themselves can read it. Solutions that dont 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 Phase I 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 Phase I 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[**biasHistoryPetsVs2016**], 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 the 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 are need to work with 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 management process:

- provisioning - Terraform[**tfDocs**], Cloud Formation[**cfDocs**], etc.
- deployment automation - Ansible[**ansibleDocs**], Puppet[**puppetDocs**], Chef[**chefDocs**], etc.
- container orchestration - Docker Swarm[**dockerDocs**], Kubernetes[**kubeDocs**], 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 the 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[**nixopsSource**; **nixopsDocs**] - official tool
- Colmena [**colmenaSource**; **colmenaDocs**]
- morph [**morphSource**]
- deploy-rs [**deployrsSource**]

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.

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[**openVPNDocs**] 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[**tincDocs**], Tailscale[**tailscaleDocs**] and Nebula[**nebulaDocs**] utilize direct P2P links between the clients for the data traffic. Depending on the chosen mesh VPN solution, either one of the hosts that has a fixed public IP address or a separate centralized service is needed to help with the initial coordination of the P2P links. As mentioned in the problem description section, in a typical home network, only the router has a public IP address

This is done by a combination of NAT traversal[**tailscaleHowNATTraversal**] techniques and relaying.

In a typical home network, there is a router that serves as the interface between the LANs and the rest of the Internet.

2.3 Execution

2.4 Secrets

TODOs:

- talk about NAT
- talk about the differences between Tinc/Tailscale and Nebula and why we chose Tailscale for the initial version
- talk about Decentralized Identifiers (DIDs)
- talk about blockchain-like solutions
- pssh
- secrets

2.5 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, blockchain based solutions and combinations of the above.

Chapter 3

Design

##

3.1 Summary

We have designed an extensible setup for running experiments with various MPyC networking scenarios including combinations of cloud and physical machines.

The test setup is currently using DigitalOcean as a cloud provider because they offer free credits to students but the setup can be adapted to other cloud providers.

The cloud provisioning is defined declaratively using Terraform. The setup allows to define a set of machines across the regions supported by DigitalOcean, e.g. Amsterdam, New York City, San Francisco, Singapore and others.

The machines run NixOS - a declarative and highly reproducible Linux distribution.

Chapter 4

Implementation

Talk about the specifics of the implementation

Show snippets from the terraform spec

Show snippets from the Nix spec

body
body
body

$$x^2 + y^2 = 1$$

```
packages.colmena = {
    meta = {
        nixpkgs = pkgs;
    };
    defaults = mkDigitalOceanImage {
        inherit pkgs lib modulesPath;
        extraPackages = [ mpyc-demo ];
    };
} // builtins.fromJSON (builtins.readFile ./hosts.json);

output "node" {
    value = local.nodes_expanded
    }
    output "node-hostnames" {
    value = { for i, node in local.nodes_expanded : node.name => merge(node, { hostname =
    }

    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 "file" {
    #   content     = tailscale_tailnet_key.keys.key
    #   destination = "/root/keys/tailscale"
    # }

    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
        ]
    }
}
```

asd
asd

Chapter 5

Conclusions

5.1 Planning