THE SMART CONTRACT MODEL OF ASYNCHRONOUS, ROBUST MPC

BY

SHREYAS GANDLUR

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Electrical Engineering
in the College of Engineering of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Adviser:

Professor Andrew Miller

ABSTRACT

Over the past few years, massive cyberattacks have dominated the public imagination. Most Americans have been directly affected by data breaches and an overwhelming majority believe that they are no longer in control of their data. At the same time, there are important applications in which the aggregation of private data is unavoidable. Not surprisingly, there has been considerable interest in developing protocols for secure multi-party computation (MPC), i.e., N parties providing private inputs to jointly compute some function f. Recent developments in MPC have led to leaps in efficiency and MPC is quickly becoming a practical approach for privacy-centric distributed applications.

As MPC applications become deployed, guaranteeing the security of these protocols, even when interacting with other applications, is essential. The standard approach to showing security of MPC protocols under arbitrary composition is through the universal composability (UC) framework. There has been much prior work on this topic; however, most previous work either makes synchronicity assumptions or does not guarantee output delivery in the presence of even one fault. For practical MPC applications, though, dealing with asynchronicity and robustness is essential. In this thesis, we describe two new keywords, eventually and leak, that aid in defining protocols in the asynchronous world and define wrappers that implement these within the UC framework. We then use this novel wrapper to give a UC-realization of a reliable broadcast primitive, by means of Bracha's classic protocol. Finally, we define and prove a realization of what we call the smart contract model of MPC, which serves as a UC-idealization of asynchronous, robust MPC.
Keywords: multiparty computation; cryptographic protocols; universal composability; protocol security

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

A multiparty computation (MPC) protocol allows a set of n parties to compute $y_1 \dots y_n = f(x_1, \dots, x_n)$ for any function f, with the following restrictions: each party P_i learns only their output y_i and each party, and any adversary, learns no other information about inputs and outputs of other parties. The standard way to define security for MPC protocols is by the real/ideal-world paradigm. Recent work has done so through the universal composability (UC) framework [1, 2], which allows for modular analysis of cryptographic protocols. In this setting, security is defined in terms of an environment, $\mathcal{Z}$. An experiment is performed, in which $\mathcal{Z}$ interacts with a real-world and an ideal-world protocol. Whereas the real-world protocol consists of actual communication between parties to perform a joint computation, the ideal-world protocol is comprised of parties providing inputs to and receiving outputs from a trusted third party. Informally, we say a real-world protocol is *secure* if a simulator, $\mathcal{S}$, exists in the ideal world that can cause $\mathcal{Z}$ to be unable to distinguish between the two worlds. Attacks against the real-world protocol will then be limited to only those attacks possible against the ideal-world protocol.

There has been considerable work analyzing the security of protocols in *synchronous* networks, where messages are delivered within a known, finite period of time (see, for example, Katz et al. [3]). Within the MPC context, Damgård and Nielsen [4] defined an arithmetic black box (ABB) functionality for the synchronous world, giving an idealization of MPC as a general-purpose computer. Parties provide inputs to the ABB. The ABB then performs computation securely inside and sends outputs to each party.

Damgård and Nielsen's ABB depends on synchronous rounds, where parties provide inputs in each round. Synchrony assumptions can be quite un-

realistic in the real world. Actual network conditions are inconsistent and delays can have no upper bound. As they capture network unreliability, assuming the existence of only *asynchronous* networks can be more realistic: in such networks, we assume that the communication network is under the control of an adversary that can delay delivery of messages arbitrarily but not infinitely; that is, messages are *eventually* delivered.

The ABB defined in [4] is not sufficient for asynchronous networks: guarantees about parties all being in the same round no longer hold. Waiting for inputs from every party with a timeout, as ABB does, is infeasible: distinguishing between crashed parties and delayed messages isn't a possibility.

Previous works have defined a limited version of asynchronous MPC called *asynchronous secure function evaluation* (ASFE) [5, 6]. However, these functionalities do not extend an ABB to the asynchronous world. For one, ASFE is one-shot in nature, which means it only captures a single execution of the functionality. ASFE is also parameterized by a specific function f; this means that each function f requires a separate functionality. We wish to instead construct a reactive functionality for MPC, in which multiple inputs and outputs are exchanged. This would extend the ABB model of MPC to the asynchronous setting. Because an ABB can maintain an internal state between invocations, this would simplify the process of creating modular constructions of asynchronous MPC protocols: the MPC layer would be abstracted to our functionality.

Before tackling this problem, we first give a new approach to defining asynchronous protocols in UC. There have been previous approaches at representing asynchrony within the universal composability framework. Katz et al. [3] and Coretti et al. [6] model eventual delivery using a polling mechanism. Each party polls each ideal functionality for outputs, decrementing a delay counter. $\mathcal{A}$ causes delay by sending delay to ideal functionalities in unary format. As $\mathcal{A}$ has polynomially-bounded computational power, requiring a unary input limits the total delay on message delivery $\mathcal{A}$ can cause. This provides an indirect approach to eventual delivery: it is a consequence of the polynomial bound on written messages.

We find that this standard approach is unappealing for two reasons: (1) this takes an indirect approach to bounding delay and (2) requiring each functionality to independently account for activations and delays results in complicated protocol designs. Instead, we define a wrapper facilitating *eventual code execution* that resolves both of these issues. We additionally illustrate its use by proving the UC-security of a classical asynchronous reliable broadcast protocol due to Bracha [7].

Using this wrapper and drawing inspiration from smart contracts in cryptocurrencies, we define a version of ABB sufficient for the asynchronous world. We give a realization of this functionality and prove its security in the UC model.

CHAPTER 2

BACKGROUND

2.1 Overview

This chapter provides some necessary background information to understand this work. Section 2.2 provides some background related to multiparty computation and Shamir secret sharing-based MPC. Section 2.3 provides some background related to universal composability and the definition of security in the UC framework.

2.2 Multiparty Computation

We consider the standard MPC setting of n parties $P_1, \ldots, P_n$, where up to $t < \frac{n}{3}$ parties are corrupted by a Byzantine adversary $\mathcal{A}$. We adopt several standard techniques used in Shamir secret-sharing based MPC. We omit these for brevity, but these can be found in [8, Section 2]. Additionally, we use the following notation:

Let p be a prime and $s \in \mathbf{F}_p$ be some secret in the field. Then, $[\![s]\!]_t$ denotes a Shamir secret sharing [9] of threshold t. In particular, let $\phi : \mathbf{F}_p \to \mathbf{F}_p$ be a randomly sampled t-degree polynomial such that $\phi(0) = s$. Then, each P_i gets a share $[\![s]\!]_t^i = \phi(i)$.

2.3 Universal Composability

The universal composability framework of Canetti [1, 2] is based on the real/ideal-world paradigm. Informally, we define security by showing that all attacks in the real world, in which parties communicate according to a

prescribed protocol π, are limited to only those possible in the ideal world, in which parties send inputs to and receive outputs from a trusted third party.

Formally, we adopt the standard Interactive Turing Machine (ITM) treatment of protocol execution (see [1, Section 3]). The execution of the real protocol π consists of the following Interactive Turing Machines (ITMs): the parties, $P_1 \dots P_n$; the environment, $\mathcal{Z}$, which represents everything outside of a protocol execution; and the real-world adversary, $\mathcal{A}$, which describes the behavior of corrupted parties. The execution of the ideal functionality $\mathcal{F}$ consists of the following ITMs: the dummy parties, $\mathcal{D}_1 \dots \mathcal{D}_n$, which act only as channels between $\mathcal{Z}$ and $\mathcal{F}$; the environment, $\mathcal{Z}$; and the ideal-world adversary (or simulator) $\mathcal{S}$, which can only meaningfully communicate with $\mathcal{F}$. In this work, we consider *static* corruptions: before protocol execution begins, the set of parties controlled by the adversary is fixed. In both worlds, $\mathcal{Z}$ provides inputs to and receives outputs from honest parties, and it can communicate with the adversary throughout the protocol.

Protocol execution in the ITM model proceeds through activations. The environment can activate honest parties with input or the adversary with a message. Honest parties compute local computation when activated and are able to activate other ITMs by sending them messages. Corrupted parties, on the other hand, act as pass-through parties for the adversary: the adversary dictates any outgoing messages and incoming messages are routed to it. There is only one ITM active at any time: once ITM A writes a message to the tape of ITM B, A becomes inactive and B becomes activated. If an activation ends without an outgoing message, $\mathcal{Z}$ becomes activated. An execution ends when $\mathcal{Z}$ outputs a single bit.

We now define security in this model. Informally, for any environment $\mathcal{Z}$, we want this output bit to have the same probability distribution in the real and ideal worlds: this would mean that, in the view of the environment, the two worlds are indistinguishable.

More formally, we first consider the execution of the real-world protocol. Let $\mathsf{EXEC}^{\pi,\mathcal{A},\mathcal{Z}}_{\mathsf{REAL}}(\mathsf{z},\tilde{\mathsf{r}})$ constitute the output of $\mathcal{Z}$ in the real world consisting of interactions with the adversary $\mathcal{A}$ and the parties $P_1, \dots, P_n$ running pro-

tocol π. Here, z is the initial input to $\mathcal{Z}$ and $\tilde{\mathsf{r}}$ is the set of random tapes that $\mathcal{Z}$, $\mathcal{A}$, and $P_1, \ldots, P_n$ have access to. We define $\mathsf{EXEC}^{\pi,\mathcal{A},\mathcal{Z}}_{\mathsf{REAL}}(\mathsf{z})$ to be the random variable after choosing $\tilde{\mathsf{r}}$ uniformly at random.

Next, we consider the execution of the ideal-world functionality. Let $\mathsf{EXEC}^{\mathcal{F},\mathcal{S},\mathcal{Z}}_{\mathsf{IDEAL}}(\mathsf{z},\tilde{\mathsf{r}})$ constitute the output of $\mathcal{Z}$ in the ideal world consisting of interactions with the ideal-world adversary (or simulator) S and the dummy parties $\mathcal{D}_1, \ldots, \mathcal{D}_n$ acting as passthroughs between $\mathcal{Z}$ and $\mathcal{F}$. Here, z is the initial input to $\mathcal{Z}$ and $\tilde{\mathsf{r}}$ is the set of random tapes that $\mathcal{Z}$, $\mathcal{S}$, and $\mathcal{F}$ have access to. We define $\mathsf{EXEC}^{\mathcal{F},\mathcal{S},\mathcal{Z}}_{\mathsf{IDEAL}}(\mathsf{z})$ to be the random variable after choosing $\tilde{\mathsf{r}}$ uniformly at random.

Definition 1 (UC-realization)**.** *We say that a protocol* Π *t-securely UC-realizes* $\mathcal{F}$ *in the presence of malicious adversaries, if, for all malicious PPT adversaries* $\mathcal{A}$ *and for all PPT environments* $\mathcal{Z}$*, there exists a PPT adversary* $\mathcal{S}$*, called a simulator, such that*

$$\{\mathsf{EXEC}^{\pi,\mathcal{A},\mathcal{Z}}_{\mathsf{REAL}}(\mathsf{z})\}_{\mathsf{z}\in\{0,1\}^*} \stackrel{c}{\equiv} \{\mathsf{EXEC}^{\mathcal{F},\mathcal{S},\mathcal{Z}}_{\mathsf{IDEAL}}(\mathsf{z})\}_{\mathsf{z}\in\{0,1\}^*}$$

It is often easier to work with the following definition instead.

Definition 2 (UC-realization, dummy adversary)**.** *Let* $\mathcal{D}$ *be the dummy adversary; that is, an ITM that only acts a pass-through between the environment and the parties/protocol. We say that a protocol* Π *t-securely UC-realizes* $\mathcal{F}$ *in the presence of malicious adversaries, if there exists a PPT adversary* $\mathcal{S}$*, called a simulator, such that, for all PPT environments* $\mathcal{Z}$*,*

$$\{\mathsf{EXEC}^{\pi,\mathcal{D},\mathcal{Z}}_{\mathsf{REAL}}(\mathsf{z})\}_{\mathsf{z}\in\{0,1\}^*} \stackrel{c}{\equiv} \{\mathsf{EXEC}^{\mathcal{F},\mathcal{S},\mathcal{Z}}_{\mathsf{IDEAL}}(\mathsf{z})\}_{\mathsf{z}\in\{0,1\}^*}$$

CHAPTER 3

UNIVERSAL COMPOSABILITY WRAPPER

3.1 Overview

In this chapter, we define a new ITM, $\mathcal{W}$, that we call a wrapper. $\mathcal{W}$ will exist in both the real and ideal worlds, and its purpose is increasing the modularity of protocol design. We define two keywords in this chapter: leak and eventually. Section 3.2 discusses the notion of protocol leakage in the UC framework and defines leak. Section 3.3 defines eventually and gives a new approach to eventual code execution. We note that although we define these two keywords below separately, they are part of the same wrapper $\mathcal{W}$. Section 3.4 gives an example of an ideal functionality for atomic broadcast in the polling and wrapper models of eventual code execution. Section 3.5 uses the eventually keyword to define eventual delivery point-to-point channels that can be used in designing asynchronous protocols. Section 3.6 modifies the definitions of UC-security from Section 2.3 to include the wrapper.

3.2 Protocol Leakage and the Leak Keyword

An adversary can gain information during the course of a protocol. A protocol designer will naturally try to capture such adversarial leakage, in order to prove security despite leakage. In the UC framework, the default way for adversaries to gain information from functionalities is via the backdoor message mechanism. However, one downside to backdoor messages is that adversaries are immediately activated. In some situations, this is unwanted. For example, a functionality can contain both adversarial leakage and an eventually codeblock - if one of these calls activated the functionality immediately, the adversary could prevent the other from being called by not

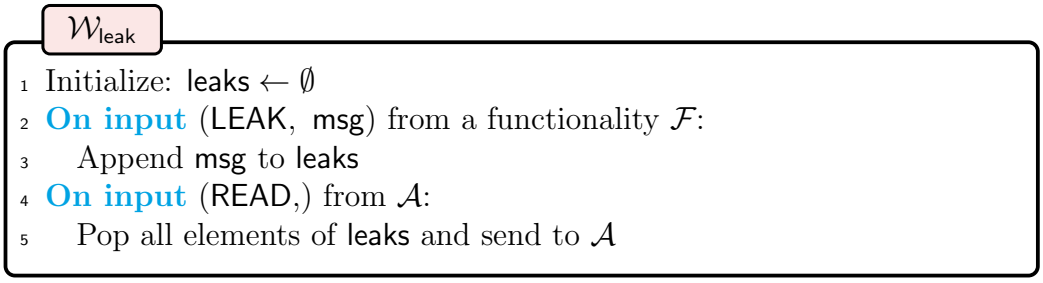

Figure 3.1: Leakage Keyword

returning control. In some cases though, we need the adversary to gain immediate control, as in the advance mechanism defined later in Figure 3.2. We do this by distinguishing a send command, which immediately sends a message to $\mathcal{A}$, and a leak command, which adds messages to a queue to be later retrieved by $\mathcal{A}$. When a functionality calls leak msg, this is translated to a message (LEAK, msg) for the wrapper and handled as defined in Figure 3.1.

3.3 Eventual Code Execution and the Eventually Keyword

We consider an asynchronous network in which the adversary has complete control over message delivery. Katz et al. [3] and Coretti et al. [6] both achieve eventual delivery by requiring $\mathcal{A}$ to provide delays in unary format.

Although this does achieve eventual delivery, it does so in an unsatisfying manner, as a consequence of unary messages. Additionally, these functionalities do not themselves allow for *eventual code execution*, a more general concept in which code, rather than messages, must be executed at the will of the adversary. Instead, in the unary delay/fetch model of eventual code execution, each functionality must independently handle adversarial delays and party fetches, as in the asynchronous Byzantine agreement functionality of Coretti et al. [6].

Eventual code execution is essential to many natural distributed systems tasks in the asynchronous world. One example is atomic broadcast. Defined

in Figure 3.7, the atomic broadcast primitive guarantees that all inputs from a set of fixed parties are appended to a totally ordered distributed log. However, the only guarantee is that inputs must *eventually* enter the log; atomic broadcast does not provide deterministic guarantees on when this happens. Requiring each asynchronous functionality to independently handle fetches and delays results in complicated functionality and protocol structures. This reduces clarity of protocols and modularity of designs. In brief, protocol designers end up needing to focus on ancillary details instead of on the main purpose of a protocol.

We use a different approach and define an eventually keyword that allows for arbitrarily delayed code execution. The most recent [1] version of the UC framework defines computation bounds in terms of a new construct called import. In our wrapper, we directly related delay to import. Contrasting with $\mathcal{A}$ writing unary strings to a functionality's tape, we require $\mathcal{A}$ to provide equal amounts of delay and import to the wrapper. An adversary must therefore "give up" compute power in order to cause delays. Although we do not explicitly deal with this in our proofs later, one benefit of this approach is that UC security proofs must be more explicit in verifying the polynomial run-time of simulators.

We also replace the mechanism of parties sending fetch messages. Instead, we define an eventually wrapper, as in Figure 3.2, in which $\mathcal{Z}$ directly provides advance messages to $\mathcal{W}$. This replaces the polling mechanism in previous approaches with a centralized one with $\mathcal{Z}$. This simplifies functionality design by no longer requiring functionalities themselves to explicitly handle polling. In order to keep this construct meaningful, we only consider those environments that provide sufficiently many advance messages for all code in the wrapper to execute.

We note that the wrapper in Figure 3.2 does not execute code internally; rather, it only stores functions and arguments in a queue. When a function and arguments pair is popped from the queue, W forwards these to the relevant functionality. The functionality itself will then execute the function. This avoids any complexities with properly accounting for import, simplifying run-time analysis. To reduce redundancy, we note that every functionality

using the eventually wrapper implicitly contains the handler defined in Figure 3.3. This handler ensures that functions popped from the eventual queue are executed by functionalities.

$\mathcal{W}_{\text{Eventually}}$

Initialize delay $\leftarrow 0$, runqueue $\leftarrow \emptyset$
On input (EVENTUALLY, func, args, leakmsg) from functionality $\mathcal{F}$:
  Append (func, args, $\mathcal{F}$) to runqueue
  Leak ($\mathcal{F}$, func.name, leakmsg)
  delay $\leftarrow$ delay $+ 1$
  Send (OK,) to $\mathcal{F}$
On input (ADVANCE,), 1 Ⓣ from $\mathcal{Z}$
  If |runqueue| > 0 and delay $== 0$:
    Pop (func, args, $\mathcal{F}$) from runqueue
    Send (func, args) to $\mathcal{F}$
  Else:
    delay $\leftarrow$ max(delay $- 1, 0$)
    Send (ADVANCE,) to $\mathcal{A}$
On input (EXECUTE, idx) from $\mathcal{A}$:
  Remove runqueue[idx] as (func, args, $\mathcal{F}$)
  Send (func, args) to $\mathcal{F}$
On input (DELAY, T), T Ⓣ from $\mathcal{A}$:
  delay $\leftarrow$ delay $+$ T
  Send (OK,) to $\mathcal{A}$

Figure 3.2: Eventual code execution wrapper

On input (func, args) from $\mathcal{W}$:
  Execute func with args

Figure 3.3: Implicit wrapper message handler

In order to aid in clarity of presentation, we often use the notation from Figure 3.4 for the eventually keyword. The anonymous function represented

```
1 Leak leakmsg and Eventually :
2    {codeblock e}
```

Figure 3.4: Simplified notation for eventually

$\mathcal{F}_{\text{Async}}^{P_i,P_j}$

```
1 On input m from P_i:
2    Eventually : send m to P_j
```

Figure 3.5: Asynchronous Network Functionality

by e, along with leakmsg, are passed to $\mathcal{W}$. A functionality can choose to provide no leak message when calling an eventually. In that case, it can omit the leakage part of Figure 3.4 and leakmsg defaults to $\perp$. Although we omit reference to arguments in this notation, any local variables in scope of the codeblock definition will implicitly be passed to the wrapper as arguments.

3.4 Eventual Message Delivery

The UC framework does not include point-to-point channels or network abstractions directly within the model. Instead, functionalities can be designed to adequately capture necessary properties of message communication. Previous approaches to eventual delivery relied on the fetch mechanism (see $\mathcal{F}_{\text{ED-SMT}}$ in [3, Section 3.2] and $\mathcal{F}_{\text{A-SMT}}$ in [6, Section 3]). A major drawback to these functionalities is that they must explicitly handle everything we abstract to $\mathcal{W}$; as such, they become very complicated. Once we define the eventually keyword, however, writing the functionality for an eventual delivery point-to-point channel becomes straightforward, as we see in Figure 3.5.

Throughout this work, we use the shorthand $\mathbf{P_i}$ sends m to $\mathbf{P_j}$ to mean $\mathbf{P_i}$ sends m to $\mathcal{F}_{\text{Async}}^{P_i,P_j}$.

3.5 Atomic Broadcast

Atomic broadcast is a type of message broadcast in which all honest parties must output messages in the same order. This is an important distributed systems primitive, and there has been considerable work building and analyzing such protocols to achieve it. For an example of an asynchronous atomic broadcast protocol, see Miller et al. [10].

Traditionally, atomic broadcast is given a property-based definition (see [10, Definition 1]). We now two UC-style definitions below for functionalities capturing the essence of atomic broadcast: Figure 3.6 gives a definition using the traditional polling mechanism and Figure 3.7 gives a definition using our new model of eventual code execution. We note that these functionalities are not precisely equivalent, but are UC definitions tailored for the mechanisms used.

The atomic broadcast functionality in the fetch model contains considerable fluff: this is necessary to handle delays and capture the properties of atomic broadcast. When using the eventually keyword with the wrapper, however, the functionality delegates all of these tasks to the wrapper and the goal of the functionality is immediately apparent.

$\mathcal{F}_{\text{Atomic}}$

Initialize: $\mathsf{BC} \leftarrow \emptyset$, $\mathsf{inputs} \leftarrow \emptyset$, $\mathsf{D}_1, \ldots, \mathsf{D}_n \leftarrow 0$, $\mathsf{pos}_1, \ldots, \mathsf{pos}_n \leftarrow 0$

On input tx from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:

  Append tx to inputs

  Send tx to $\mathcal{A}$

On input $(\mathsf{FETCH},)$ from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:

  If $D_i > 0$: $D_i \leftarrow D_i - 1$

  Else if $\mathsf{pos}_i < |\mathsf{BC}|$: $\mathsf{pos}_i \leftarrow \mathsf{pos}_i + 1$

  Else if $\mathsf{pos}_i == |\mathsf{BC}|$:

    Pop tx from inputs

    If $\mathsf{tx} \neq \perp$, append tx to BC and set $\mathsf{pos}_i \leftarrow \mathsf{pos}_i + 1$

  Send $\mathsf{BC}[1 : \mathsf{pos}_i]$ to $\mathbf{P_i}$

On input $(\mathsf{DELAY}, \mathbf{P_i}, \mathsf{T})$ from $\mathcal{A}$:

  If T is in unary notation: $\mathsf{D}_i \leftarrow \mathsf{D}_i + \mathsf{T}$

  Send $(\mathsf{OK},)$ to $\mathcal{A}$

On input $(\mathsf{APPEND}, \mathsf{idx})$ from $\mathcal{A}$:

  Remove $\mathsf{inputs}[\mathsf{idx}]$ as tx

  Append tx to BC

  Send $(\mathsf{OK},)$ to $\mathcal{A}$

Figure 3.6: Atomic broadcast using the fetch mechanism

$\mathcal{F}_{\text{Atomic}}$

Initialize: $\mathsf{BC} \leftarrow \emptyset$

On input tx from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:

  Leak (tx, $\mathbf{P_i}$) and **Eventually** :

    Append tx to BC

    $\mathsf{pos} \leftarrow |\mathsf{BC}|$

    For each $\mathbf{P_j}$:

      Leak (pos, $\mathbf{P_j}$) and **Eventually** : send $\mathsf{BC}[1 : \mathsf{pos}]$ to $\mathbf{P_j}$

Figure 3.7: Atomic broadcast using eventually

3.6 UC Security Using the Wrapper

We now give a modified definition of UC-security with the included wrapper.

Consider the execution of the real-world protocol. Let $\mathsf{EXEC}^{\pi,\mathcal{D},\mathcal{W},\mathcal{Z}}_{\mathsf{REAL}}(\mathsf{z},\tilde{\mathsf{r}})$ constitute the output of $\mathcal{Z}$ in the real world consisting of interactions with the dummy adversary adversary $\mathcal{D}$, the wrapper $\mathcal{W}$, the parties $P_1,\ldots,P_n$ running protocol π. Here, z is the initial input to $\mathcal{Z}$ and $\tilde{\mathsf{r}}$ is the set of random tapes that $\mathcal{Z}$ and $P_1,\ldots,P_n$ have access to. We define $\mathsf{EXEC}^{\pi,\mathcal{D},\mathcal{W},\mathcal{Z}}_{\mathsf{REAL}}(\mathsf{z})$ to be the random variable after choosing $\tilde{\mathsf{r}}$ uniformly at random.

Next, consider the execution of the ideal-world functionality. Let $\mathsf{EXEC}^{\mathcal{F},\mathcal{S},\mathcal{W},\mathcal{Z}}_{\mathsf{IDEAL}}(\mathsf{z},\tilde{\mathsf{r}})$ constitute the output of $\mathcal{Z}$ in the ideal world consisting of interactions with the ideal-world adversary (or simulator) S, the wrapper $\mathcal{W}$, and the dummy parties $\mathcal{D}_1,\ldots,\mathcal{D}_n$ acting as passthroughs between $\mathcal{Z}$ and $\mathcal{F}$. Here, z is the initial input to $\mathcal{Z}$ and $\tilde{\mathsf{r}}$ is the set of random tapes that $\mathcal{Z}$, $\mathcal{S}$, and $\mathcal{F}$ have access to. We define $\mathsf{EXEC}^{\mathcal{F},\mathcal{S},\mathcal{W},\mathcal{Z}}_{\mathsf{IDEAL}}(\mathsf{z})$ to be the random variable after choosing $\tilde{\mathsf{r}}$ uniformly at random.

UC-security now consists of the following definition:

Definition 3 (UC-realization, wrapper)**.** *We say that a protocol Π t-securely UC-realizes $\mathcal{F}$ in the presence of malicious adversaries, if there exists a PPT adversary $\mathcal{S}$, called a simulator, such that, for all PPT environments $\mathcal{Z}$,*

$$\{\mathsf{EXEC}^{\pi,\mathcal{D},\mathcal{W},\mathcal{Z}}_{\mathsf{REAL}}(\mathsf{z})\}_{\mathsf{z}\in\{0,1\}^*} \stackrel{c}{\equiv} \{\mathsf{EXEC}^{\mathcal{F},\mathcal{S},\mathcal{W},\mathcal{Z}}_{\mathsf{IDEAL}}(\mathsf{z})\}_{\mathsf{z}\in\{0,1\}^*}$$

Although we do not give a proof, all theorems from Canetti [1] extend to our setting with wrapper $\mathcal{W}$.

CHAPTER 4

ANALYSIS OF BRACHA BROADCAST

4.1 Overview

In this chapter, we demonstrate the use of eventually keyword in UC proofs, by proving the security of Bracha's classical asynchronous reliable broadcast protocol in the UC framework. Section 4.2 gives a traditional definition of asynchronous reliable broadcast. Section 4.3 defines an ideal functionality for asynchronous reliable broadcast. It also gives a representation of Bracha's protocol in the UC framework, and constructs a simulator, ultimating proving the security of Bracha's protocol.

4.2 Property-Based Reliable Broadcast

Reliable broadcast is an important primitive that forms the basis of nearly every asynchronous application. In an instance of reliable broadcast, we consider a fixed set of n parties in which a dealer D sends a message m to all other parties, and parties try to agree on the dealer's input. Traditionally, a single instance of reliable broadcast is required to satisfy the following two properties:

- If the dealer D is honest, then all honest parties will eventually output the same message m
- If the dealer D is dishonest, then either (1) all honest parties eventually output the same message m' or (2) no honest party outputs a message

There is considerable distributed systems literature around asynchronous reliable broadcast. The first reliable broadcast to achieve the optimal $t = n/3$ corruptions threshold is a classical protocol due to Bracha [7]. Security for

this protocol is shown by Bracha by proving the properties given above.

4.3 UC Security of Bracha Broadcast

As an illustration of the eventually keyword, we now give a proof of security for Bracha's broadcast protocol in the UC framework. We first define an ideal functionality for reliable broadcast in Figure 4.1, succinctly capturing the required properties of a reliable broadcast protocol.

$\mathcal{F}_{\mathbf{RBC}}$

1 **On input** tx from dealer **D**:
2 **Leak** (SENT, **D**, tx)
3 For each $\mathbf{P_i}$:
4 **Leak** ($\mathbf{P_i}$, tx) and **Eventually** : send tx to $\mathbf{P_i}$
5 If **D** is corrupt, send (OK,) to **D**

Figure 4.1: Reliable broadcast functionality

We additionally write the Bracha broadcast protocol in the UC framework in Figure 4.2 - note that this protocol is implicitly in the hybrid world consisting of $n(n-1)\mathcal{F}_{async}$ pairwise communication channels.

We use the following two lemmas from [7] to prove UC security for Π_{Bracha}:

Lemma 1 ([7, Lemma 3])**.** *Consider an execution of the protocol* Π_{Bracha} *with up to* $t < \frac{n}{3}$ *corruptions. If an honest party outputs a value* tx*, then every other honest party will eventually output* tx *as well.*

Lemma 2 ([7, Lemma 4])**.** *Consider an execution of the protocol* Π_{Bracha} *with up to* $t < \frac{n}{3}$ *corruptions. If the dealer is an honest party and broadcasts* tx*, then every honest party will eventually output* tx*.*

We prove the security of Π_{Bracha} in a hybrid setting where each party P_i has access to a point-to-point channel $\mathcal{F}_{Async_{i,j}}$ to every other party P_j. It is sufficient to prove security of Π_{Bracha} against the dummy adversary $\mathcal{D}$ [1,

$\Pi_{\textbf{Bracha}}$

Participants: Parties $\{P_1, \ldots P_n\}$, of which one assumes the role of dealer D
define BQ $\leftarrow \lceil \frac{n+t}{2} \rceil \leftarrow$ a byzantine quorum
As Dealer:
  On input tx from $\mathcal{Z}$:
    For each $\mathbf{P_i}$: send (VAL, tx) to $\mathbf{P_i}$
As Recipient:
  On input (VAL, tx) from D or
    (ECHO, tx) from BQ parties or
    (READY, tx) from BQ-t parties:
    For each $\mathbf{P_i}$: send (ECHO, tx) to $\mathbf{P_i}$
  On input (ECHO, tx) from BQ parties or
    (READY, tx) from BQ-t parties:
    For each $\mathbf{P_i}$: send (READY, tx) to $\mathbf{P_i}$
  On input (READY, tx) from BQ parties:
    Output tx

Figure 4.2: Bracha broadcast protocol

Section 4.3.1]. We show this by constructing an ideal world adversary (or simulator) $\mathcal{S}$ such that the environment $\mathcal{Z}$ cannot distinguish its transcript in the real world and the ideal world. Specifically, in our setting, $\mathcal{Z}$ should not be able to distinguish interacting with the parties $P_1 \ldots P_n$, $\mathcal{D}$, and the eventual code execution wrapper $\mathcal{W}$ in the real world and interacting with dummy parties $D_1 \ldots D_n$, $\mathcal{S}$, and $\mathcal{W}$ in the ideal world.

We construct a simulator S_{Bracha} as in Figure 4.3. We must verify that $\mathcal{S}_{\mathsf{Bracha}}$ causes the environment $\mathcal{Z}$ to be unable to distinguish the joint view of the interfaces with the adversary, the wrapper $\mathcal{W}$, and parties in the ideal world versus those in the real world. We prove Lemmas 3 and 4 first, from which Theorem 1 follows.

First, let us state and prove a lemma related to the delay of $\mathcal{W}$ in the ideal world:

Lemma 3. *Consider the ideal world consisting of $\mathcal{Z}$, $\mathcal{S}_{\mathsf{Bracha}}$, the dummy parties $D_1 \ldots D_n$, $\mathcal{W}$, and $\mathcal{F}_{\mathsf{ABB}}$. For every $n > 0$, at the nth activation of*

$\mathcal{Z}$, the runqueue *in $\mathcal{W}$ is empty or the* delay *in $\mathcal{W}$ is non-zero.*

Proof. We prove this by induction.

First, consider the case of $\mathcal{Z}$'s first activation. By definition, runqueue is empty and the lemma holds for $n = 1$.

We now let $n > 0$. We show that, assuming the statement holds for all activations in $[1, n]$, it also holds for activation $n + 1$. On its nth activation, $\mathcal{Z}$ wrote a message to the tape of $\mathcal{S}_{\mathsf{Bracha}}$, dummy party D_i, or $\mathcal{W}$. We show that, in each case, the lemma holds for activation $n + 1$:

- $\mathcal{Z}$ wrote to $\mathcal{S}_{\mathsf{Bracha}}$: This activation does not affect the runqueue or delay of $\mathcal{W}$, and the lemma holds for the $(n + 1)$th activation of $\mathcal{Z}$.

- $\mathcal{Z}$ wrote to D_i: As D_i is a dummy party, the message is routed to the ideal functionality. By definition of $\mathcal{F}_{\mathsf{RBC}}$ and by lines 2-6 of Figure 3.2, runqueue is extended for outputs. However, delay is also incremented and becomes non-zero. Thus the lemma holds for the $(n+1)$th activation of $\mathcal{Z}$.

- $\mathcal{Z}$ wrote to $\mathcal{W}$: The only message $\mathcal{Z}$ can write to $\mathcal{W}$ is ADVANCE. By lines 7-13 of Figure 3.2, since we know that either the runqueue is empty or the delay is non-zero, we know that the delay is decremented, if possible, and $\mathcal{S}_{\mathsf{Bracha}}$ is activated. By lines 4-9 and 15-20 of Figure 4.3, and in particular lines 17-18, a DELAY message is sent to $\mathcal{W}$ if delay becomes 0 with a non-empty runqueue. This increments delay and makes it non-zero.

  We argue that this last step, where $\mathcal{S}$ increments delay, is always possible, even though the wrapper is structured such that infinite delay is impossible. The reason for this is as follows: if runqueue is non-empty, then it means that either (1) an honest party input a value to $\mathcal{F}_{\mathsf{RBC}}$ or

(2) $\mathcal{S}$, in lines 24-27 of Figure 4.3, instructed a corrupt party input in response to an honest output. In case (1), all honest parties in Π_{Bracha} eventually output, as given in Lemma 1. In case (2), all honest parties in Π_{Bracha} eventually output, as given in Lemma 2. Since $\mathcal{S}$ internally emulates Π_{Bracha} and must only cause delay until all simulated honest parties finish outputting, and all simulated honest parties eventually output, delay will not occur indefinitely. $\mathcal{S}$ is thus able to instruct $\mathcal{W}$ to increment delay whenever it becomes 0. Thus, the lemma holds for the $(n+1)$th activation of $\mathcal{Z}$.

In all cases, we have that the lemma holds for the $(n+1)$th activation of $\mathcal{Z}$, and the lemma holds for all n. □

We now show the following lemma, comparing the internal simulation of the real world to the actual real world protocol execution, as well as comparing the transcript of $\mathcal{Z}$ in both worlds.

Lemma 4. *Consider the real world consisting of $\mathcal{Z}$, $\mathcal{D}$, $P_1, \ldots, P_n$, $n(n-1)$ $\mathcal{F}_{Async}$ functionalities, and $\mathcal{W}$. Consider the ideal world consisting of $\mathcal{Z}$, $\mathcal{S}_{\mathsf{Bracha}}$, $D_1, \ldots, D_n$, $\mathcal{F}_{\mathsf{RBC}}$, and $\mathcal{W}$. In particular, in both worlds, we consider the same $\mathcal{Z}$ activated with the same input z and same randomness $\tilde{\mathsf{r}}$. Then, for all $k > 0$, on the kth activation of $\mathcal{Z}$, the following two statements hold:*

i. Either (1) *$k > 1$ and in activation $k-1$, Z wrote a message to D_i/P_i or* (2) *the state of $\mathcal{D}$, $P_1, \ldots, P_n$, the $n(n-1)$ $\mathcal{F}_{Async}$ functionalities, and $\mathcal{W}$ in the real world and within $\mathcal{S}_{\mathsf{Bracha}}$'s simulation in the ideal world are identical.*

ii. Z is activated by the same message in both worlds

Proof. We prove this by mutual induction on the two statements.

First, consider $k = 1$. On $\mathcal{Z}$'s first activation, the simulated and actual real worlds are both in their initialized states, so (i) holds. Additionally, as Z is initialized with the same input z in both worlds, (ii) trivially holds for both worlds. So (i) and (ii) in the lemma hold for $k = 1$.

Now let $k > 1$. We consider $\mathcal{Z}$'s kth activation and assume (i) and (ii) hold for all $k' < k$.

On $\mathcal{Z}$'s $(k-1)$th activation, $\mathcal{Z}$ wrote a message m to the tape of either $\mathcal{W}$, $\mathcal{S}/\mathcal{A}$, or $\mathcal{D}_i/P_i$. By the inductive hypothesis on (ii), we know that the transcript of $\mathcal{Z}$ is the same in both worlds, up to activation $k-1$. As $\mathcal{Z}$'s randomness, $\tilde{\mathsf{r}}$, is fixed, the behavior of $\mathcal{Z}$ is completely deterministic. This means that $\mathcal{Z}$ wrote the same m to the same ITM in both worlds.

We now consider, in turn, each ITM to which $\mathcal{Z}$ could write a message in activation $k-1$, and show that (i) holds for k in each:

- $\mathcal{Z}$ wrote to $\mathcal{W}$ in activation $k-1$: Consider the ideal-world wrapper. We know by Lemma 3 that either runqueue is empty or delay is non-zero. In either case, on an ADVANCE by $\mathcal{Z}$, $\mathcal{W}$ activates $\mathcal{S}_{\mathsf{Bracha}}$ by line 13 of Figure 3.2.

  In the ideal world, we know, by the inductive hypothesis on (i), that one of two cases is true for activation $k-1$:

  - $k > 2$ and Z wrote to D_i/P_i in activation $k-2$. But by lines 4-9 of Figure 4.3, all leaks are fetched and any inputs to $\mathcal{F}_{RBC}$ are sent to simulated real-world parties. This causes the actual and simulated real-worlds to be identical, up to the point of an ADVANCE message. By line 20 of Figure 4.3, $\mathcal{S}_{\mathsf{Bracha}}$ then forwards this message to the simulated wrapper, causing the simulated and actual real-worlds to remain identical when $\mathcal{S}_{\mathsf{Bracha}}$ returns control to $\mathcal{Z}$ for its kth activation. Thus (i) holds for k.

– The states were identical at activation $k-1$. Thus, after receiving the forwarded message from $\mathcal{W}$, by line 21, $\mathcal{S}_{\mathsf{Bracha}}$ forwards this message to the simulated wrapper. This causes the simulated and actual real-worlds to remain identical when $\mathcal{S}_{\mathsf{Bracha}}$ returns control to $\mathcal{Z}$ for its kth activation. Thus (i) holds for k.

- $\mathcal{Z}$ wrote to $\mathcal{S}/\mathcal{A}$ in activation $k-1$: In the ideal world, we know, by the inductive hypothesis on (i), that one of two cases is true at activation $k-1$:

  – $k > 2$ and Z wrote to D_i/P_i in activation $k-2$. By lines 4-9 of Figure 4.3, all leaks are fetched and any inputs to $\mathcal{F}_{RBC}$ are sent to simulated real-world parties. This causes the actual and simulated real-worlds to be identical to the point where the adversary receives a message. By lines 10-14 of Figure 4.3, $\mathcal{S}_{\mathsf{Bracha}}$ causes the simulated real world to have the same state as the actual real world. So when $\mathcal{S}_{\mathsf{Bracha}}$ returns control to $\mathcal{Z}$ for its kth activation, the simulated and actual real worlds contain the same state and (i) holds for k.

  – The states were identical at activation $k-1$. By lines 10-14 of Figure 4.3, $\mathcal{S}_{\mathsf{Bracha}}$ causes the simulated real world to have the same state as the actual real world. Thus, when $\mathcal{S}_{\mathsf{Bracha}}$ returns control to $\mathcal{Z}$ for its kth activation, the simulated and actual real worlds contain the same state and (i) holds for k.

- $\mathcal{Z}$ wrote to D_i/P_i in activation $k-1$: Then (i) holds for k, as the first case is met.

Having shown that (i) holds for k, we now show that (ii) also holds for k. Because (i) holds for k, we know that either (1) $\mathcal{Z}$ wrote to D_i/P_i in activation $k-1$ or (2) the simulated and actual real worlds are identical at activation k. We show that in both cases, $\mathcal{Z}$ is activated by the same message

in activation k:

(1) $\mathcal{Z}$ wrote to D_i/P_i in activation $k-1$: In the real world, $\mathcal{Z}$ writes to P_i. P_i executes lines 4-6 of Figure 4.2 in $\mathcal{F}_{\mathsf{RBC}}$, which are implicit calls to $\mathcal{F}_{\mathsf{Async}}$ functionalities. These, in turn, call $\mathcal{W}$ to append scheduled eventual deliveries. P_i ends its activation with no outgoing message. In the ideal world, $\mathcal{Z}$ writes to D_i. D_i forwards its input to $\mathcal{F}_{\mathsf{RBC}}$, which performs eventually schedules and ends its activation with no outgoing message. In both worlds, as activations end without outgoing messages, $\mathcal{Z}$ is automatically invoked with no incoming message. Thus (ii) holds for k.

(2) The simulated and actual real worlds are identical at activation k: In the actual real world, $\mathcal{Z}$ can receive a message from $\mathcal{D}$ or honest parties. By (i), the simulated real world, $\mathcal{S}_{\mathsf{Bracha}}$ receives the same message from its internal simulation. We consider these cases separately:

- Output from $\mathcal{D}$: By line 21 of Figure 4.3, messages from the simulated dummy adversary are forwarded just as they are in the real world - so (ii) holds for k in the case of outputs from $\mathcal{D}$.

- Output from simulated P_i: Suppose the dealer is honest. Then an honest dummy party D_j will already have provided an input to $\mathcal{F}_{\mathsf{RBC}}$ and $\mathcal{S}_{\mathsf{Bracha}}$ will already have retrieved it earlier by lines 4-9 of Figure 4.3. By Lemma 2, every honest party will eventually output the same value. In particular, P_i will output what was input by D_j and because of the definition of $\mathcal{F}_{\mathsf{RBC}}$, $\mathcal{W}$ will contain this value. By lines 29 and 30 of Figure 4.3, $\mathcal{S}$ instructs $\mathcal{W}$ to deliver this value to the real-world P_i and (ii) holds for k for an honest dealer.

  Suppose the dealer is dishonest. If no honest parties have output a value yet, by lines 24-28, $\mathcal{S}_{\mathsf{Bracha}}$ instructs a corrupt party to input this to $\mathcal{F}_{\mathsf{RBC}}$, and instructs delivery to P_i with lins 29 and

30. If an honest party P_j has already output a value, we know by Lemma 1 that every honest party will eventually output the same value. In particular, we know that P_i output the same value that P_j had output. Because these are the same, $\mathcal{S}_{\mathsf{Bracha}}$ can instruct an output through lines 29 and 30 and (ii) holds for k for a dishonest dealer as well.

Having exhausted all cases, we have shown that (ii) holds for k. Therefore, the lemma holds for all k.

□

In order to show UC security, i.e., that Π_{Bracha} UC-realizes $\mathcal{F}_{\mathsf{RBC}}$, we now show that Lemma 4 implies Theorem 1:

Theorem 1. *Protocol* Π_{Bracha} *securely realizes* $\mathcal{F}_{\mathsf{RBC}}$ *in the* $\mathcal{F}_{\mathsf{Async}}^{n(n-1)}$*-hybrid model, in the presence of a static adversary corrupting up to* t *parties for* $t < \frac{n}{3}$

Proof of Theorem 1. By (ii) of Lemma 4, we have that, for a fixed $\mathcal{Z}$ with fixed input and randomness, $\mathcal{Z}$ receives the same order of activations and messages in both worlds. Since $\mathcal{Z}$ is deterministic once randomness is fixed, this means that the output of $\mathcal{Z}$ will be identical in both worlds. More specifically, we have that, for all $\mathcal{Z}$, for all z, and for all $\tilde{\mathsf{r}}$, $\mathsf{EXEC}_{\mathsf{IDEAL}}^{\mathcal{F},\mathcal{S},\mathcal{W},\mathcal{Z}}(\mathsf{z},\tilde{\mathsf{r}})$ is equal to $\mathsf{EXEC}_{\mathsf{REAL}}^{\pi,\mathcal{D},\mathcal{W},\mathcal{Z}}(\mathsf{z},\tilde{\mathsf{r}})$. But this is the same as saying that the two probability ensembles $\{\mathsf{EXEC}_{\mathsf{REAL}}^{\pi,\mathcal{D},\mathcal{W},\mathcal{Z}}(\mathsf{z})\}_{z\in\{0,1\}^*}$ and $\{\mathsf{EXEC}_{\mathsf{IDEAL}}^{\mathcal{F},\mathcal{S},\mathcal{W},\mathcal{Z}}(\mathsf{z})\}_{z\in\{0,1\}^*}$ are identical, giving UC security by Definition 3. □

$S_{\mathbf{Bracha}}$

Simulate parties $\mathbf{P'_1}, \ldots, \mathbf{P'_n}$ running protocol Π'_{Bracha}, $n(n-1)$ $\mathcal{F}_{async}$ point-to-point channels, dummy adversary $\mathcal{D}'$, and wrapper $\mathcal{W}'$
Init runqueue $\leftarrow \emptyset$, delay $\leftarrow 0$, tx $\leftarrow \perp$

On each activation while tx $== \perp$:
  Send (READ,) to $\mathcal{W}$ and receive leaks
  If there are leaks, because of definition of $\mathcal{F}_{RBC}$:
    Set tx using (SENT, dealer, tx) $\leftarrow$ leaks[0]
    Append leaks[1 : n] to runqueue and set delay $\leftarrow$ delay + n
    Instruct simulated dealer to input m to Π'_{Bracha}
On input (WRITE, $\mathbf{P_i}$, msg) from $\mathcal{Z}$:
  If $\mathbf{P_i}$ is corrupt:
    Send message to $\mathcal{D}'$, instructing $\mathcal{D}'$ to forward msg to $\mathbf{P'_i}$
On input (WRITE, $\mathcal{W}$, msg) from $\mathcal{Z}$:
  Send msg to $\mathcal{W}'$
On input (ADVANCE,) from $\mathcal{W}$:
  delay $\leftarrow$ max(delay $- 1, 0$)
  If delay $= 0$ and runqueue is not empty:
    Send (DELAY, 1), 1 ⓣ to $\mathcal{W}$ and receive (OK,) back from $\mathcal{W}$
    delay $\leftarrow 1$
  Send (ADVANCE,), 1 ⓣ to $\mathcal{W}'$
On input msg from $\mathcal{D}'$:
  Send msg to $\mathcal{Z}$
On input tx′ from $\mathbf{P'_i}$:
  If tx $\neq \perp$:
    tx $\leftarrow$ tx′
    Send message to a corrupt party, directing it to send tx to $\mathcal{F}_{RBC}$
    Send (READ,) to $\mathcal{W}$ and receive leaks
    Append leaks[1 : n] to runqueue
  Pop runqueue[idx] related to delivering msg to $\mathbf{P_i}$
  Send (EXECUTE, idx) to $\mathcal{W}$

Figure 4.3: Simulator for the Bracha broadcast protocol

CHAPTER 5

ASYNCHRONOUS ARITHMETIC BLACK BOX

5.1 Overview

This chapter gives a model of asynchronous, robust MPC in the UC framework. Section 5.2 summarizes previous UC formalisms of MPC. Section 5.3 defines the smart contract model of MPC, giving UC-style definitions of the protocol and ideal functionality. Section 5.4 provides a proof of UC security for our construction.

5.2 Background and Motivation

Universal composability is the standard framework for proving security of MPC protocols. Damgård and Nielsen [4] defines an arithmetic black box (ABB) for synchronous MPC. It defines an ideal functionality for MPC by treating it as a general-purpose computer. Parties can privately provide inputs to the ABB, securely compute any arithmetic function within it, and privately receive outputs from it.

The ABB defined in [4] is not sufficient for asynchronous networks. It proceeds in synchronous rounds, and expects inputs from all parties in each round. However, this is unrealizable in an asynchronous setting as it is impossible to distinguish between crashed parties and messages in transit.

Cohen [5] and Coretti et al. [6] provide idealizations of a weaker form of MPC called *asynchronous secure function evaluation*, in which a functionality only computes a specific function. ASFE is weaker than ABB as an ABB functions as a general-computer with multiple inputs and outputs, accepting

arbitrary computation. Although this is sufficient for simple MPC protocols, this is not a sufficient abstraction for more complicated MPC protocols, such as those in the client-server model of MPC. In such protocols, being able to use ABB as a building block is important: it increases modularity of protocol design and abstracts any complexity related to arbitrary composition.

5.3 Smart Contract Model of MPC

We use a new approach, drawing inspiration from smart contracts. In cryptocurrencies like Ethereum [11], participants collectively build a distributed ledger, called a blockchain, consisting of both transactions and smart contracts. Smart contracts specify a set of rules defining the execution of some function. Transactions can send inputs to smart contracts and cause state transitions to occur. This is very similar to the MPC context. In asynchronous MPC, for example, a common rule is a protocol waiting for $n - t$ inputs to perform a computation. Instead of our MPC protocol dictating specific rules for computation to occur, we instead allow protocol participants to specify any rules directly in contracts and send inputs to those contracts. We define a functionality $\mathcal{F}_{\mathsf{ABB}}$ in Figures 5.1 and 5.2 that represents this. Internally, $\mathcal{F}_{\mathsf{ABB}}$ maintains a totally ordered log of inputs and contracts, and, much like a general-purpose computer, as contracts specify transitions, $\mathcal{F}_{\mathsf{ABB}}$ performs them.

We now comment on a few design choices made in $\mathcal{F}_{\mathsf{ABB}}$:

First, because $\mathcal{F}_{\mathsf{ABB}}$ is a reactive functionality, it informs parties as inputs/contracts are defined and as outputs are produced. However, this cannot be done naively. For example, we cannot define a functionality that simply eventually sends each input and output to each P_i. The reason for this is as follows: such an ideal functionality, through the dummy parties, would only be able to write one message at a time to $\mathcal{Z}$. Since we draw inspiration from cryptocurrencies, the real-world protocol will naturally rely on the atomic broadcast functionality $\mathcal{F}_{\mathsf{Atomic}}$ as given in Figure 3.7. However, by line 9 of Figure 3.7, $\mathcal{F}_{\mathsf{Atomic}}$ can deliver multiple inputs to a party at once. A real-world protocol would accordingly generate multiple messages for $\mathcal{Z}$. $\mathcal{Z}$

$\mathcal{F}_{\mathsf{ABB}}$

Initialize $\mathsf{BC} \leftarrow \emptyset$, $\mathsf{outputs} \leftarrow \emptyset$, $\mathsf{last}_1 \leftarrow 0, \ldots, \mathsf{last}_n \leftarrow 0$, $\mathsf{vals} \leftarrow \emptyset$, $\mathsf{msgs}_1 \leftarrow \emptyset, \ldots, \mathsf{msgs}_n \leftarrow \emptyset$

On initialization: for each $\mathbf{P_i}$, **Eventually:** Execute $\mathsf{deliver}(\mathbf{P_i})$

On input $(\mathsf{INPUT}, x, \mathsf{contract\_id})$ from $\mathbf{P_i}$:

  Leak $(\mathsf{INPUT}, \mathsf{contract\_id}, \mathbf{P_i})$

  Eventually :

    $k \leftarrow$ number of previous inputs by D already included in BC

    $\mathsf{vals}[(\mathbf{P_i}, \mathsf{k}+1)] \leftarrow x$

    Append $(\mathsf{VAR}, (\mathbf{P_i}, k+1), \mathsf{contract\_id}, |\mathsf{BC}|)$ to BC and get pos

    $\mathcal{C} \leftarrow$ contract associated with $\mathsf{contract\_id}$

    Input $((\mathbf{P_i}, \mathsf{k}+1), x)$ to C and for each action in C, perform action

    Execute $\mathsf{GenOutputs}(\mathsf{pos})$

    Leak $\mathsf{OUTPUTS}, \mathsf{outputs}$

    Send $(\mathsf{OK},)$ to $\mathcal{A}$

On input $(\mathsf{CONTRACT}, \mathcal{C})$ from $\mathbf{P_i}$:

  Leak $(\mathsf{CONTRACT}, \mathcal{C}, \mathbf{P_i})$ to $\mathcal{A}$

  Eventually :

    $k \leftarrow$ number of contracts appended to BC

    $\mathsf{contract\_id} \leftarrow k+1$

    Append $(\mathsf{CONTRACT}, \mathcal{C}, \mathsf{contract\_id}, |\mathsf{BC}|)$ to BC and get pos

    Execute $\mathsf{GenOutputs}(\mathsf{pos})$

    Send $(\mathsf{OK},)$ to $\mathcal{A}$

$\mathsf{GenOutputs}(\mathsf{pos})$:

  For all $\mathbf{P_i}$: **Leak** $(\mathbf{P_i}, \mathrm{pos})$ and **Eventually** :

    If $\mathsf{pos} \leq \mathsf{last}_i$: return

    For $(\mathsf{out}, \mathsf{pos}')$ in $\mathsf{outputs}$ where $\mathsf{last}_i < \mathsf{pos}' \leq \mathsf{pos}$:

      Eventually append out to msgs_i

    $\mathsf{last}_i \leftarrow \mathsf{pos}$

    Append $\mathsf{BC}[\mathsf{last}_i + 1 : \mathsf{pos}]$ to msgs_i

  Send $(\mathsf{OK},)$ to $\mathcal{A}$

Deliver$(\mathbf{P_i})$:

  Leak $(\mathbf{P_i})$ and **Eventually** execute Deliver$(\mathbf{P_i})$

  If $|\mathsf{msgs}_i| > 0$: pop all elems of msgs_i and send to P_i

Figure 5.1: Asynchronous arithmetic black box functionality

$\mathcal{F}_{\mathsf{ABB}}$ **(continued)**

Execution of action, given pos and contract $\mathcal{C}$, with label contract_id:
  (type, vars) $\leftarrow \mathcal{C}$
  If type is of LIN_COMB:
    (coeffs $= [a_0, \ldots, a_n]$, in_vals $= [1, x_1, \ldots, x_n]$, sum_label) $\leftarrow$ vars
    (sum_label, coeffs.in_vals) $\rightarrow \mathcal{C}$
  If type is of MULT:
    (factor1_val, factor2_val, prod_label) $\leftarrow$ vars
    (prod_label, factor1_val $*$ factor2_val) $\rightarrow \mathcal{C}$
  If type is of OUTPUT:
    (out_label, out_val) $\leftarrow$ vars
    Append (OUTPUT, (contract_id, out_label), out_val, pos) to outputs
    (OUTPUT, out_label) $\rightarrow \mathcal{C}$

Figure 5.2: Asynchronous ABB action execution

would then trivially distinguish between the ideal and real worlds: a multiple message output would only happen in the real world. It follows that $\mathcal{F}_{\mathsf{ABB}}$ must have a mechanism to deliver multiple messages to each P_i. This would allow ideal-world parties to forward multiple messages at once to $\mathcal{Z}$. As seen in lines 27 and 29 of Figure 5.1, each possible message is eventually appended to a list for each party P_i. Then, per the deliver function defined in lines 31-33, this entire list is eventually output to P_i.

Another aspect we take care to design is the order in which outputs are generated - a party should not receive an output y before receiving definition messages for any input x that y depends on. This keeps outputs from $\mathcal{F}_{\mathsf{ABB}}$ meaningful temporally. $\mathcal{F}_{\mathsf{ABB}}$ ensures this in lines 27 and 29 of the GenOutputs function in Figure 5.1.

Finally, we quickly describe how MPC contracts are defined in our protocol. We want the design of contracts to be as general as possible. We require them to have an interface compatible with $\mathcal{F}_{\mathsf{ABB}}$ in Figure 5.2, but otherwise allow the contracts to act as a black box. Specifically, contracts store variables internally and must output state transitions to be performed by $\mathcal{F}_{\mathsf{ABB}}$ or parties. We require contracts to have the following property: for a given sequence of variable definitions in a contract, the contract must always give

the same sequence of outputs. The reason for this is the following: in the ideal functionality, contracts operate on actual values, whereas in the real functionality, each party will input secret shares to a contract. Contracts therefore should not branch on specific values of variables or use any internal randomness. State transitions that contracts output can be one of three types: linear combinations, multiplications, or outputs. Each behaves as the following:

- Linear combinations: The contract must output coefficients for the linear combination, values stored in the contract that must be linearly combined, and a label into which the sum must be stored. This is seen in line 4 of Figure 5.2.

- Multiplications: The contract must output values of the two factors and a label into which the product should be stored, as in line 7 of Figure 5.2.

- Outputs: The contract must output the label and value for what should be publicly reconstructed, as in line 9 of Figure 5.2. Note that, in this protocol, we only discuss public reconstructions. However, handling private reconstructions only requires a simple change.

We now give a real-world instantiation of $\mathcal{F}_{\mathsf{ABB}}$. We follow the standard online/offline paradigm of MPC protocols. For the offline phase, we define $\mathcal{F}_{\mathsf{Offline}}$ in Figure A.1. $\mathcal{F}_{\mathsf{Offline}}$ represents standard elements used in MPC, generally generated during preprocessing: random sharings used to hide inputs and random triples used for Beaver's trick in secret share multiplication. For the online phase, drawing inspiration from blockchains, we use the functionality for atomic broadcast given in Figure 3.7 as a backbone for how the MPC protocol progresses. Parties send their inputs to $\mathcal{F}_{\mathsf{Atomic}}$ and they will eventually be included in a shared totally ordered log.

We define the online phase protocol, Π_{Online}, in Figure A.2. When parties receive inputs values from $\mathcal{Z}$, these are hidden using sharings from $\mathcal{F}_{\mathsf{Offline}}$

and passed directly into $\mathcal{F}_{\mathsf{Atomic}}$, as in lines 2-4 of Figure A.2. When parties receive contracts to broadcast, these are directly send to $\mathcal{F}_{\mathsf{Atomic}}$ in line 6. As parties receive inputs and contracts from $\mathcal{F}_{\mathsf{Atomic}}$, they are handled using the class ContractHandler, defined in Figures A.3 and A.4. Parties create a new ContractHandler for each contract appended to their log, and they pass inputs and protocol messages directly into them. This is seen in lines 12-25 of Figure A.2. These handlers process these inputs and messages, and perform transitions as defined in contracts.

We now explain how ContractHandler works. As mentioned before, given the same order of inputs, contracts are required to give the same order of state transitions. The reason for that becomes apparent in the definition of ContractHandler. In order to keep all parties coordinated, ContractHandler inputs values one at a time, as seen in lines 2-7 in Figure A.4, and a handler for a contract only performs one transition at a time.

Transitions are resolved inside of these handlers using standard techniques from Shamir secret sharing. Linear combinations are handled as in lines 9-12 of Figure A.4 - secret shares are simply linear combined together according to the defined coefficients. For multiplications, we use random triples from $\mathcal{F}_{\mathsf{Offline}}$ to perform Beaver's trick for multiplication. This is shown in lines 13-29 of Figure A.4. Finally, we perform public reconstructions, as defined in lines 30-40 of Figure A.4, by having all parties send their shares to all other participants and use Robust-Interpolate from [8] to reconstruct outputs. Although our protocol does not rely on batch reconstruction or demonstrate private reconstruction for simplified presentation, the security of Π_{Online}, as shown later, will still hold after modifications to handle those.

5.4 UC Security of Asynchronous ABB

As established previously, it is sufficient to prove security of Π_{Online} against the dummy adversary $\mathcal{D}$ [1, Section 4.3.1]. We construct a simulator $\mathcal{S}_{\mathsf{ABB}}$ in Figures A.5, A.6, and A.7. We must verify that $\mathcal{S}_{\mathsf{ABB}}$ causes the environment $\mathcal{Z}$ to not be able to distinguish its interactions in the real world from

those of the ideal world. Specifically, in our setting, $\mathcal{Z}$ should not be able to distinguish its interactions with the parties $P_1 \dots P_n$, $\mathcal{D}$, and the eventual code execution wrapper $\mathcal{W}$ in the real world from those interactions in the ideal world with dummy parties $D_1 \dots D_n$, $\mathcal{S}_{\mathsf{ABB}}$, and $\mathcal{W}$.

Before proving UC security, we prove a series of useful lemmas. First, note that the only randomness involved in Π_{Online} is in the hiding of private inputs and the generation of secret shares. Therefore, we first prove a lemma related to the indistinguishability of secret shares and public reconstructions generated in the real world to those generated similar to how $\mathcal{S}_{\mathsf{ABB}}$ generates them.

Consider an execution of the real world consisting of $\mathcal{Z}$, $\mathcal{D}$, parties $P_1, \dots, P_n$ running the protocol Π_{Online}, $n(n-1)$ $\mathcal{F}_{\mathsf{Async}}$ functionalities, $\mathcal{F}_{\mathsf{Atomic}}$, and $\mathcal{F}_{\mathsf{Offline}}$. Let inputs be the set of input values that $\mathcal{Z}$ provides parties in Π_{Online} - for honest parties, explicitly, and for corrupt parties, implicitly by directing them to send a message to $\mathcal{F}_{\mathsf{Atomic}}$. Let $\mathsf{hidden\_inputs}$ be the set of hidden inputs broadcast in $\mathcal{F}_{\mathsf{Atomic}}$ on line 6 of Figure A.2. Let $\mathsf{offline\_shares}$ be the set of shares that adversarial parties have access to from $\mathcal{F}_{\mathsf{Offline}}$. Let $\mathsf{factor\_polys}$ be the set of polynomials that adversarial parties receive for hidden factors in Beaver multiplication, from lines 19-24 in Figure A.4. Let $\mathsf{output\_shares}$ be the set of shares that adversarial parties receive from Π_{Online} in line 33 and 34 in Figure A.4. Let $\mathsf{outputs}$ be the set of outputs that parties generated from Π_{Online} and forwarded to $\mathcal{Z}$.

Now we construct sets of secret shares that similar to those generated in $\mathcal{S}_{\mathsf{ABB}}$. Let $\mathsf{hidden\_inputs}'$ be a set of length $|\mathsf{hidden\_inputs}|$ consisting of random values in $\mathbf{F_p}$. Let $\mathsf{offline\_shares}'$ be a set of length $|\mathsf{offline\_shares}|$ consisting of random values in $\mathbf{F_p}$. Let $\mathsf{factor\_polys}'$ be a set of length $|\mathsf{factor\_polys}|$ consisting of random t-degree polynomials in $\mathbf{F_p}$. Now we prove the following lemma:

Lemma 5. *Consider the sets* $\mathsf{hidden\_inputs}'$, $\mathsf{offline\_shares}'$, $\mathsf{factor\_polys}'$, *and* $\mathsf{outputs}$ *generated as described above. Then, it is always possible to create a set* $\mathsf{output\_shares}'$ *consistent with* $\mathsf{hidden\_inputs}'$, $\mathsf{offline\_shares}'$, $\mathsf{factor\_polys}'$,

and outputs. *We also have that*

$$(\mathsf{inputs}, \mathsf{hidden\_inputs}, \mathsf{offline\_shares}, \mathsf{factor\_polys}, \mathsf{output\_shares}, \mathsf{outputs})$$

is indistinguishable from

$$(\mathsf{inputs}, \mathsf{hidden\_inputs}', \mathsf{offline\_shares}', \mathsf{factor\_polys}', \mathsf{output\_shares}', \mathsf{outputs}).$$

Proof. First, we note a few properties of the real-world protocol. By line 19 of Figure A.2, secret sharings used to hide inputs are used only once, and by lines 10-15 of Figure A.1, $\mathcal{F}_{\mathsf{Offline}}$ guarantees that Beaver triples used in multiplications are only used once. This means that each piece of randomness in the protocol is only used once. Additionally, we note that contracts are all processed independently, as a new handler is created for every contract in line 15 of Figure A.2. This means that, without loss of generality, we can assume that all inputs are to the same contract.

Consider the number of points the adversary and environment learn. Since we consider a setting of up to t corruptions, and by the definition of $\mathcal{F}_{\mathsf{Offline}}$ in Figure A.1, we know that they learn at most t points for an input. In particular, the t points the adversary and environment learn by combining values from hidden_inputs and offline_shares is insufficient to constrain the actual input. We now argue that the adversary only learns up to t points of the output of each linear combination and multiplication. For linear combinations, since parties combine secret shares they already have, this directly follows. For multiplication, as Beaver triples are only used once in our protocol, the adversary only learns t shares of the product because of Beaver multiplication. In particular, since t shares is not enough to constrain the actual product, and offline_shares and factor_polys are generated randomly in the real world, the adversary cannot detect if points from offline_shares′ and polynomials from factor_polys′ are used, instead of points from offline_shares and polynomials from factor_polys.

It now immediately follows that output_shares′ can be constructed. output_shares represents the shares of outputs from honest parties from line 33 of Figure

A.4. Output shares of honest parties can be faked by doing the following: as the adversary has up to t shares for the output, and we know the actual value of the output from $\mathsf{outputs}$, a t-degree polynomial can be constructed that is consistent with these $< t + 1$ points. Then, simulated output shares for honest parties are points directly from this polynomial. It is also immediate that

$$(\mathsf{inputs}, \mathsf{hidden\_inputs}, \mathsf{offline\_shares}, \mathsf{factor\_shares}, \mathsf{output\_shares}, \mathsf{outputs})$$

is indistinguishable from

$$(\mathsf{inputs}, \mathsf{hidden\_inputs}', \mathsf{offline\_shares}', \mathsf{factor\_shares}', \mathsf{output\_shares}', \mathsf{outputs}).$$

This is because, as seen above, the only constraint is that $\mathsf{output\_shares}'$ must be consistent with $\mathsf{outputs}$, and that holds from the definition of $\mathsf{output\_shares}'$.

□

Note that the definition of the second set of shares above is how $\mathcal{S}_{\mathsf{ABB}}$ generates them in lines 4-14 of Figure A.7. This means that the shares and public reconstructions generated by $\mathcal{S}_{\mathsf{ABB}}$ are indistinguishable from those in an actual execution of Π_{Online}.

We now consider the following lemma, demonstrating that, in Π_{Online}, all state transitions in a contract will eventually be performed:

Lemma 6. *Consider an execution of the real-world protocol Π_{Online}. Consider a contract in the protocol. We have that all honest parties will receive the same state transitions in the same order from each contract. Additionally, all honest parties will eventually resolve each of these transitions.*

Proof. First, note that all parties eventually receive contract inputs from $\mathcal{F}_{\mathsf{Atomic}}$ in the same order. By lines 7-29 of Figure A.2, all honest parties will input values to contracts in the same order. Since contracts are required to output state transitions in the same order if variables are defined in the same order, we have that all honest parties will eventually receive the same state

transitions in the same order from each contract.

There are three types of transitions to consider: linear combinations, multiplications, and outputs. We consider each of these in turn, showing that, in all cases, all honest parties eventually resolve the transition:

- T is type LIN_COMB: By lines 10-12 of Figure A.4, all honest parties immediately resolve the transition.

- T is type MULT: In lines 19-21 of Figure A.4, each honest party sends hidings of factors to every other party. Because of eventual code execution, each of these messages is eventually delivered. In the worst case, parties must wait for shares from all other honest parties for robust interpolation in lines 23 and 24. Because of eventual delivery, this will eventually happen. Parties then resolve this transition in lines 26-28. This means that each honest party will eventually resolve a MULT transition.

- T is type OUTPUT: In line 33 of Figure A.4, each honest party sends its share to every other party. Because of eventual delivery, these messages are all eventually delivered. In the worst case, interpolation of the output on line 35 requires all honest shares - however, because all honest shares are eventually delivered, the output will eventually be interpolated.

□

We now consider the following lemma, describing $\mathcal{W}$ in the ideal world:

Lemma 7. *Consider the ideal world consisting of $\mathcal{Z}$, $\mathcal{S}_{\mathsf{ABB}}$, the dummy parties $D_1 \dots D_n$, $\mathcal{W}$, and $\mathcal{F}_{\mathsf{ABB}}$. For every $n > 0$, at the nth activation of $\mathcal{Z}$, the* runqueue *in $\mathcal{W}$ is empty or the* delay *in $\mathcal{W}$ is non-zero.*

Proof. The proof of this is similar to the one for Lemma 3. The differences are the following.

(1) References to specific line numbers of $\mathcal{F}_{\mathsf{ABB}}$ and $\mathcal{S}_{\mathsf{ABB}}$ change.

(2) Indefinite delay does not occur, because all outputs from $\mathcal{F}_{\mathsf{ABB}}$ are accounted for by the following:

- All inputs to Π_{Online} are eventually delivered to all honest parties by the definition of $\mathcal{F}_{\mathsf{Atomic}}$ and line 29 of Figure A.2. This handles outputs from line 29 of $\mathcal{F}_{\mathsf{ABB}}$.

- All outputs are eventually delivered to all honest parties by Lemma 6. This handles outputs from line 27 of $\mathcal{F}_{\mathsf{ABB}}$.

□

We now consider the following lemma, which is the analogue of Lemma 4, the only difference being computational indistinguishability rather than perfect simulation.

Lemma 8. *Consider the real world consisting of $\mathcal{Z}$, $\mathcal{D}$, the parties $P_1, \ldots, P_n$ running Π_{Online}, $n(n-1)$ $\mathcal{F}_{Async}$ functionalities, $\mathcal{F}_{\mathsf{Offline}}$, $\mathcal{F}_{\mathsf{Atomic}}$, and $\mathcal{W}$. Consider the ideal world consisting of $\mathcal{Z}$, $\mathcal{S}_{\mathsf{ABB}}$, $D_1, \ldots, D_n$, $\mathcal{F}_{\mathsf{ABB}}$, and $\mathcal{W}$. For all $k > 0$, on the kth activation of $\mathcal{Z}$, the following two statements hold:*

(i) Let z_0 denote the input on which $\mathcal{Z}$ is activated. Let $z_1, \ldots, z_{k-1}$ denote the outputs from $\mathcal{Z}$ to the real world, and let z_d denote the last non-P_i activation in $z_1, \ldots, z_{k-1}$. The distribution on the steps performed in the real world, on inputs $z_0, z_1, \ldots, z_l$, and the distribution of those performed within $\mathcal{S}_{\mathsf{ABB}}$'s simulation in the ideal world are computationally indistinguishable.

(ii) The distribution on the messages activating $\mathcal{Z}$ in the real world is computationally indistinguishable from the distribution on the messages activating $\mathcal{Z}$ in the ideal world.

Proof. We prove this through mutual induction.

Let $k = 1$. Clearly, (i) and (ii) hold in this case.

Now, choose $k > 1$, and assume that (i) and (ii) hold for all $k' < k$. By the inductive hypothesis on (ii), we know that the transcript of $\mathcal{Z}$ is computationally indistinguishable in both worlds, up to activation $k-1$. This means that the distributions on the message written by $\mathcal{Z}$ and the ITM written to by $\mathcal{Z}$ in both worlds are computationally indistinguishable for activation $k-1$. We now show that (i) holds for k by considering each ITM to which $\mathcal{Z}$ could write a message at the end of activation $k-1$:

- $\mathcal{Z}$ wrote to $\mathcal{W}$ in activation $k-1$: Let us consider the ideal world. By a combination of Lemma 7 and line 13 of $\mathcal{W}$ in Figure 3.2, $\mathcal{S}$ activates $\mathcal{S}_{\mathsf{ABB}}$. We know by the inductive hypothesis on (i) that the distribution on the steps executed in the actual and simulated real worlds are indistinguishable, up to the point of an activation of P_i. However, by lines 5-6 of Figure A.5, $\mathcal{S}_{\mathsf{ABB}}$ retrieves leaks from $\mathcal{F}_{\mathsf{ABB}}$ and simulates inputs in the simulated real world with lines 31 and 33 of Figure A.5. This makes the distribution on the steps indistinguishable to the point of the ADVANCE message. By line 17 of Figure A.5, $\mathcal{S}_{\mathsf{ABB}}$ simulates the ADVANCE message internally, as would happen in the real world. When $\mathcal{S}$ receives an output from the simulated real world and generates an output for $\mathcal{Z}$ the simulated real world will have an indistinguishable distribution on the steps executed within it to those steps executed in the actual real world. Thus, (i) will hold for k.

- $\mathcal{Z}$ wrote to $\mathcal{S}/\mathcal{A}$ in activation $k-1$: We know by the inductive hypothesis on (i) that the distribution on the steps executed in the actual and simulated real worlds are indistinguishable, up to the point of an activation of P_i. However, by lines 5-6 of Figure A.5, $\mathcal{S}_{\mathsf{ABB}}$ retrieves

leaks from $\mathcal{F}_{\mathsf{ABB}}$ and simulates inputs in the simulated real world with lines 31 and 33 of Figure A.5. This makes the distribution on the steps indistinguishable to the point of the adversary receiving a message. By lines 9 and 11 of Figure A.5, $\mathcal{S}_{\mathsf{ABB}}$ forwards these messages to the simulated real world, as would happen in the real world. When $\mathcal{S}$ receives an output from the simulated real world and generates an output for $\mathcal{Z}$ the simulated real world will have an indistinguishable distribution on the steps executed within it to those steps executed in the actual real world. Thus, (i) will hold for k.

- $\mathcal{Z}$ wrote to D_i/P_i in activation $k-1$: The two distributions being compared do not change. (i) thus holds for k.

Since (i) holds for k in all cases shown above, (i) holds for k. We now show that (ii) holds for k as well.

As argued earlier, the distributions on the message written by $\mathcal{Z}$ and the ITM written to by $\mathcal{Z}$ in both worlds are computationally indistinguishable for activation $k-1$. This means that the distribution of z_{k-1}^{REAL}, the message written by $\mathcal{Z}$ in the real world is indistinguishable from the distribution of z_{k-1}^{IDEAL}, the message written by $\mathcal{Z}$ in the ideal world. We, therefore, consider each ITM to which $\mathcal{Z}$ could write a message in activation $k-1$:

- $\mathcal{Z}$ wrote to D_i/P_i in activation $k-1$: In the real world, P_i passes the input to $\mathcal{F}_{\mathsf{Atomic}}$ in line 4 of Figure A.2. After a codeblock is appended to $\mathcal{W}$, the activation ends with no outgoing message. In the ideal world, D_i passes the input to $\mathcal{F}_{\mathsf{ABB}}$. In line 6 of 5.1, $\mathcal{F}_{\mathsf{ABB}}$ appends a codeblock to $\mathcal{W}$ and the activation ends with no outgoing message. In both worlds, as there is no outgoing message, $\mathcal{Z}$ is automatically invoked with no incoming message, and (ii) holds for k.

- $\mathcal{Z}$ wrote to another tape in activation $k-1$: Since (i) holds for k and z_{k-1} is not an activation of D_i/P_i, we know that the distribution of the

steps performed in the real and simulated real worlds are computationally indistinguishable. $\mathcal{Z}$ may receive an output from two tapes in the real world: the tapes of honest parties and from $\mathcal{D}$. We show that in both cases, $\mathcal{S}_{\mathsf{ABB}}$ causes indistinguishable outputs in the ideal world:

- $\mathcal{Z}$ receives an output from $\mathcal{D}$: $\mathcal{S}_{\mathsf{ABB}}$ receives an output from its internal simulation by (i). $\mathcal{S}_{\mathsf{ABB}}$ causes a similar output to happen in the ideal world through line 23 in Figure A.5. The function adv_handler in Figure A.7 passes all messages to corrupt contract handlers in line 3 of A.7. These are used to calculate the view of corrupt parties and the values they would have for output shares. When simulated honest parties generate outputs, $\mathcal{S}_{\mathsf{ABB}}$ uses these calculated output shares in lines 5-14 of Figure A.7 to simulate output shares for honest parties. These are then forwarded to $\mathcal{Z}$ in line 15. By Lemma 5, any shares contained within the output message, including secret shares included in messages from the simulated $\mathcal{F}_{\mathsf{Atomic}}$ functionality, the simulated $\mathcal{F}_{\mathsf{Offline}}$ functionality, or simulated honest parties, are indistinguishable from those in the real world. This means that the message from $\mathcal{S}_{\mathsf{ABB}}$ that activates $\mathcal{Z}$ is indistinguishable from the one that $\mathcal{D}$ sends $\mathcal{Z}$ in the real world. Thus, (ii) holds for k.

- $\mathcal{Z}$ receives an output from an honest P_i: $\mathcal{S}_{\mathsf{ABB}}$ receives an output from a simulated P_i by (i). $\mathcal{S}_{\mathsf{ABB}}$ causes a similar output to happen in the ideal world through line 21 in Figure A.5, which calls the party_handler function defined in Figure A.6. We now explain how it works by discussing various goals of this function.

  Note that Π_{Online} relies on $\mathcal{F}_{\mathsf{Atomic}}$ for underlying consensus. In $\mathcal{F}_{\mathsf{Atomic}}$, as seen in line 1 of Figure 3.7, a blockchain is stored internally. $\mathcal{F}_{\mathsf{ABB}}$ also stores a blockchain internally on line 1 of Figure 5.1. The first goal of $\mathcal{S}_{\mathsf{ABB}}$ is to set these blockchains equal - more specifically, as simulated honest parties output definitions of inputs and contracts that are not already included in $\mathcal{F}_{\mathsf{ABB}}$'s BC, $\mathcal{S}_{\mathsf{ABB}}$ must instruct $\mathcal{F}_{\mathsf{ABB}}$ to include them. $\mathcal{S}_{\mathsf{ABB}}$ does this

through lines 5-16 and lines 17-26 of Figure A.6. These inputs and contracts might have originally been input to $\mathcal{F}_{\mathsf{ABB}}$ by honest parties. If so, $\mathcal{F}_{\mathsf{ABB}}$ would already have received those inputs and $\mathcal{S}_{\mathsf{ABB}}$ would have already retrieved leaks related to them. $\mathcal{S}_{\mathsf{ABB}}$ merely needs to instruct $\mathcal{F}_{\mathsf{ABB}}$ to include those inputs in its internal blockchain, and $\mathcal{S}_{\mathsf{ABB}}$ does with lines 7, 16, 19, and 31 of Figure A.6. If, instead, these are inputs and contracts from originally input to the simulated protocol by adversarial parties, $\mathcal{S}_{\mathsf{ABB}}$ must instruct dummy adversarial parties to send these to $\mathcal{F}_{\mathsf{ABB}}$ first. This happens in lines 8-11 and 22-26 of Figure A.6.

The second goal of $\mathcal{S}_{\mathsf{ABB}}$ is to schedule input and contract definition messages for delivery to the dummy D_i in the ideal world. These are the definition messages in $\mathcal{F}_{\mathsf{ABB}}$ on line 29 of Figure 5.1. $\mathcal{S}_{\mathsf{ABB}}$ does this on lines 32-33 of Figure A.6.

The third goal of $\mathcal{S}_{\mathsf{ABB}}$ is to schedule outputs from contracts for delivery to the dummy D_i in the ideal world. These are outputs from contracts in $\mathcal{F}_{\mathsf{ABB}}$ in line 27 of Figure 5.1. $\mathcal{S}_{\mathsf{ABB}}$ does this on lines 35-36 of Figure A.6. We quickly note that the outputs in the real world and ideal functionality will be indistinguishable, because line 11 of $\mathcal{F}_{\mathsf{ABB}}$ in Figure 5.1 performs transitions one-by-one in the same order as does $\mathsf{ContractHandler}$ in Π_{Online}. Since the same transitions are performed, the output from $\mathcal{F}_{\mathsf{ABB}}$ will be the same as what is publicly reconstructed in Π_{Online}.

Because of the delivery mechanism used in $\mathcal{F}_{\mathsf{ABB}}$ in lines 31-33 of Figure 5.1, $\mathcal{S}_{\mathsf{ABB}}$ instructs delivery of all messages appended above through lines 37-38 of Figure A.6. Once $\mathcal{S}_{\mathsf{ABB}}$ instructs delivery, the message that activates $\mathcal{Z}$ in the ideal world will be indistinguishable from that of the real world. Therefore, (ii) holds for k.

Thus, in all cases, (ii) holds for k. Since we have shown that (i) and (ii) hold for k, we have that the lemma holds for all k. □

We now give a proof of UC-security, which follows directly from Lemma 8, by means of the following theorem:

Theorem 2. *Protocol* Π_{Online} *securely realizes* $\mathcal{F}_{\mathsf{ABB}}$ *in the* ($\mathcal{F}_{\mathsf{Atomic}}$, $\mathcal{F}_{\mathsf{Offline}}$, $\mathcal{F}_{\mathsf{Async}}^{n(n-1)}$)*-hybrid model, in the presence of a static adversary corrupting up to* t *parties for* $t < \frac{n}{3}$

Proof. By (ii) of Lemma 8, we have that the transcripts of $\mathcal{Z}$ in both worlds are computationally indistinguishable. But this implies that

$$\{\mathsf{EXEC}_{\mathsf{REAL}}^{\pi,\mathcal{D},\mathcal{W},\mathcal{Z}}(\mathsf{z})\}_{z\in\{0,1\}^*} \stackrel{c}{\equiv} \{\mathsf{EXEC}_{\mathsf{IDEAL}}^{\mathcal{F},\mathcal{S},\mathcal{W},\mathcal{Z}}(\mathsf{z})\}_{z\in\{0,1\}^*}$$

must hold. This is because the transcript represents the $\mathcal{Z}$'s entire view of the execution - if these probability ensembles were not indistinguishable, the transcripts of both worlds could be distinguished based on the distribution of the output bit, contradicting Lemma 8. Thus, UC security holds for $\mathcal{F}_{\mathsf{ABB}}$. □

CHAPTER 6

CONCLUSION AND FUTURE WORK

This thesis provides an approach to designing more modular asynchronous protocols in the universal composability framework by providing a wrapper for eventual code execution. This simplifies protocol design by replacing the standard fetch mechanism for eventual delay from the literature with a new keyword eventually. We argue that asynchronous protocols defined this keyword are clearer to define and easier to analyze, by giving examples of functionalities in both approaches. We also use this keyword to prove the security of Bracha's reliable broadcast protocol. We finally define an asynchronous arithmetic black box for MPC in what we call the Smart Contract Model of MPC, and provide a UC-realization of it.

There are several promising directions for future work:

- We defined our eventual code execution wrapper by leveraging the import approach to polynomial-bounded execution. However, we do not actually use import in defining our protocols or functionalities. Understanding import and how it can be used in ITM definitions more deeply is essential to increasing usability of the UC framework.

- We constructed an asynchronous arithmetic black box as a UC-idealization of asynchronous, robust MPC. Future work would build more complicated MPC protocols using our ABB as a building block.

APPENDIX A

UC MODELS FOR ABB

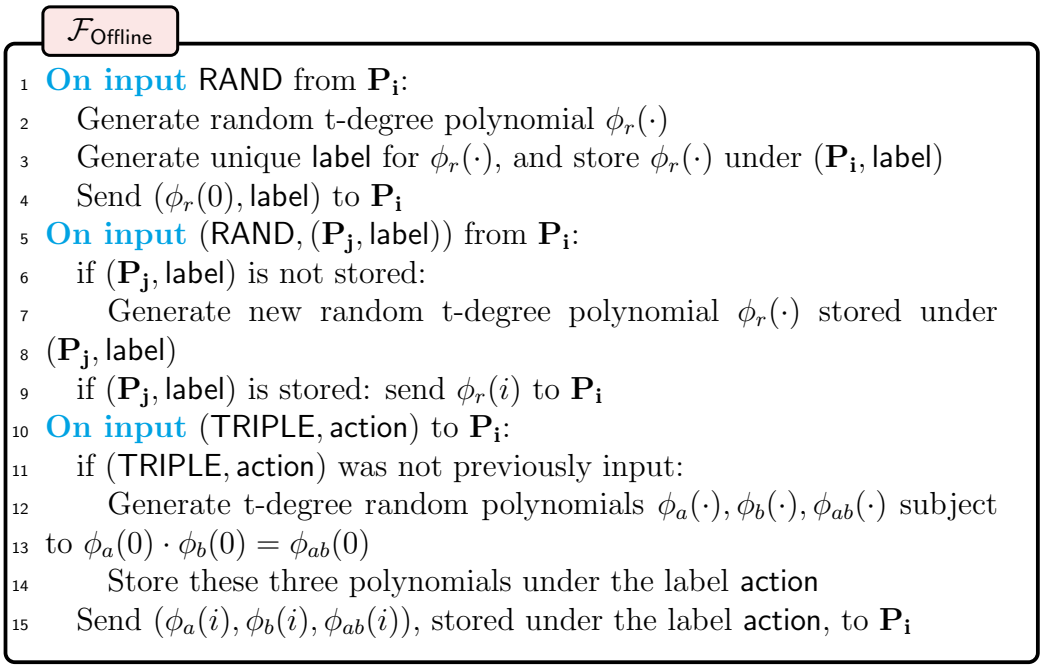

Figure A.1: Offline phase functionality

Π_{Online}

Initialize $\mathsf{BC} \leftarrow \emptyset$, $\mathsf{handlers} \leftarrow \emptyset$, $\mathsf{buffer} \leftarrow \emptyset$
On input $(\mathsf{INPUT}, x, \mathsf{contract\_id})$ from $\mathcal{Z}$:
  Send rand to $\mathcal{F}_{\mathsf{Offline}}$ and receive (r, label) in response
  Input $(\mathsf{INPUT}, x - r, \mathsf{label}, \mathsf{contract\_id})$ to $\mathcal{F}_{\mathsf{Atomic}}$
On input $(\mathsf{CONTRACT}, \mathcal{C})$ from $\mathcal{Z}$:
  Input $(\mathsf{CONTRACT}, \mathcal{C})$ to $\mathcal{F}_{\mathsf{Atomic}}$
On input BC' from $\mathcal{F}_{\mathsf{Atomic}}$:
  if $|\mathsf{BC}| \geq |\mathsf{BC}'|$: return
  $\mathsf{msgs} \leftarrow \emptyset$
  for pos in $[|\mathsf{BC}| + 1, |\mathsf{BC}'|]$:
    $\mathsf{input} \leftarrow \mathsf{BC}'[\mathsf{pos}]$
    if input is of form $((\mathsf{CONTRACT}, \mathcal{C}), \mathbf{P_j})$:
      $k \leftarrow$ number of contracts in $\mathsf{BC}'[1 : \mathsf{pos} - 1]$
      Label $\mathcal{C}$ with $\mathsf{contract\_id} = k + 1$
      $\mathsf{handlers}[\mathsf{contract\_id}] \leftarrow$ new contract handler for $\mathcal{C}$
      Pop all elements in buffer for $\mathcal{C}$ and input to $\mathsf{handlers}[\mathsf{contract\_id}]$
      Append $(\mathsf{CONTRACT}, \mathcal{C}, \mathsf{contract\_id}, \mathsf{pos})$ to msgs
    if input is of form $((\mathsf{INPUT}, y, \mathsf{label}, \mathsf{contract\_id}), \mathbf{P_j})$:
      If $(\mathbf{P_j}, \mathsf{label})$ was previously in BC': continue
      Send $(\mathsf{LABEL}, (\mathbf{P_j}, \mathsf{label}))$ to $\mathcal{F}_{\mathsf{Offline}}$ and receive $[\![r]\!]_t^i$
      $k \leftarrow$ number of inputs by $\mathbf{P_j}$ in $\mathsf{BC}'[1 : \mathsf{pos} - 1]$
      $[\![\mathbf{P_j}^{k+1}]\!]_t^i \leftarrow y + [\![r]\!]_t^i$
      $\mathcal{C} \leftarrow$ contract labeled with $\mathsf{contract\_id}$ in $\mathsf{BC}'[1 : \mathsf{pos} - 1]$
      If $\mathcal{C} \neq \perp$:
        Input $(\mathbf{P_j}, [\![\mathbf{P_j}^{k+1}]\!]_t^i)$ to $\mathsf{handlers}[\mathsf{contract\_id}]$
        Append $(\mathsf{INPUT}, \mathsf{contract\_id}, \mathsf{pos})$ to msgs
  $\mathsf{BC} \leftarrow \mathsf{BC}'$
  msgs.extend(PerformTransitions())
  Output msgs if $|\mathsf{msgs}| > 0$
On input msg from $\mathcal{F}_{\mathrm{async}}^{(\mathbf{P_j}, \mathbf{P_i})}$:
  $\mathsf{msgs} \leftarrow \mathsf{PerformTransitions}(\mathbf{P_j}, \mathsf{msg})$
  Output msgs if $|\mathsf{msgs}| > 0$
PerformTransitions():
  $\mathsf{msgs} \leftarrow \emptyset$
  For handler in handlers: msgs.extend(handler.do_actions())
  Return msgs
$\mathsf{PerformTransitions}(\mathbf{P_i}, \mathsf{msg})$:
  $\mathsf{contract\_id} \leftarrow$ contract id from msg
  If $\mathsf{contract\_id}$ in handlers: input $(\mathbf{P_i}, \mathsf{msg})$ to $\mathsf{handlers}[\mathsf{contract\_id}]$
  Else: Append $(\mathbf{P_i}, \mathsf{msg})$ to buffer
  Return PerformTransitions()

Figure A.2: Protocol for the online phase

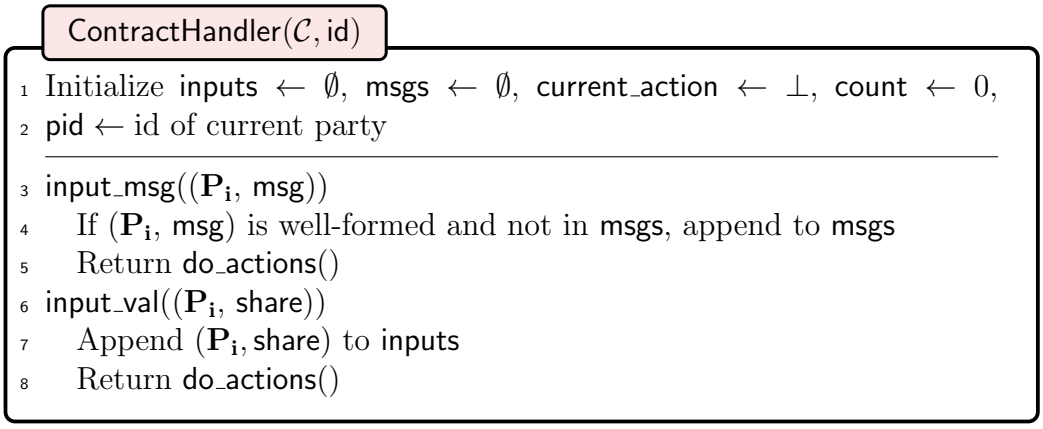

Figure A.3: Initialization and inputs for ContractHandler

ContractHandler($\mathcal{C}$, id) **(actions)**

```
do_actions():
  If current_action = ⊥:
    While current_action = ⊥:
      Pop input from queue and input → C
      current_action ← C.get
    Return ∅ if current_action = ⊥
    count ← count + 1
  (type, vars) ← current_action
  If type is of LIN_COMB:
    (coeffs=[a_0, ..., a_n], in_vals=[1, [[x_1]]_t^i, ..., [[x_n]]_t^i], sum_label) ← vars
    (sum_label, coeffs · in_vals) → C
    Set current_action = ⊥ and return do_actions()
  If type is of MULT:
    ([[x]]_t^i, [[y]]_t^i, prod_label) ← vars
    If shares have not yet been broadcast:
      Send (TRIPLE, (id, count, prod_label)) to F_Offline
      Receive ([[a]]_t^i, [[b]]_t^i, [[ab]]_t^i) from F_Offline
      Compute [[x − a]]_t^i ← [[x]]_t^i − [[a]]_t^i and [[y − b]]_t^i ← [[y]]_t^i − [[b]]_t^i
      factor1_msg ← (MULT, count, 1, [[x − a]]_t^i)
      factor2_msg ← (MULT, count, 2, [[y − b]]_t^i)
      For each P_i, send factor1_msg and factor2_msg to P_i
      Append (pid, factor1_msg) and (pid, factor2_msg) to msgs
    x − a ← Robust-Interpolate(shares of x − a from msgs)
    y − b ← Robust-Interpolate(shares of y − b from msgs)
    If x − a ≠ ⊥ and y − b ≠ ⊥:
      [[xy]]_t^i = (x − a)(y − b) + [[b]]_t^i(x − a) + [[a]]_t^i(y − b) + [[ab]]_t^i
      (outvar, [[xy]]_t^i) → C
      Set current_action = ⊥ and return do_actions()
    Else: Return ∅
  If type is of OUTPUT:
    (out_label, [[x]]_t^i) ← C
    If shares have not yet been broadcast:
      For each P_i, send (OUTPUT, (out_label, count), [[x]]_t^i) to P_i
      Append (pid, (OUTPUT, (out_label, count), [[x]]_t^i)) to msgs
    x ← Robust-Interpolate(shares of x from msgs)
    If x ≠ ⊥:
      (OUTPUT, out_label) → C
      Set current_action = ⊥
      Return [(OUTPUT, (id, out_label), x)] + do_actions()
    Else: Return ∅
```

Figure A.4: do_actions() in ContractHandler

$\mathcal{S}_{\mathsf{ABB}}$

Simulate parties $\mathbf{P'_1}, \ldots, \mathbf{P'_n}$ running protocol Π'_{online}, $n(n-1)$ $\mathcal{F}'_{\mathsf{async}}$ point-to-point channels, $\mathcal{F}'_{\mathsf{atomic}}$, $\mathcal{F}'_{\mathsf{offline}}$, dummy adversary $\mathcal{D}'$, and wrapper $\mathcal{W}'$
idealqueue $\leftarrow \emptyset$, idealdelay $\leftarrow 0$, outputs $\leftarrow \emptyset$, corrupthandlers $\leftarrow \emptyset$

On every activation, except for activations in the middle of a subroutine: Execute get_leaks()
On input (WRITE, $\mathbf{P_i}$, msg) from $\mathcal{Z}$:
  If $\mathbf{P_i}$ is corrupt:
    Send message to $\mathcal{D}'$, instructing $\mathcal{D}'$ to forward msg to $\mathbf{P'_i}$
On input (WRITE, $\mathcal{W}$, msg) from $\mathcal{Z}$:
  Send msg to $\mathcal{W}'$
On input (ADVANCE,) from $\mathcal{W}$:
  delay $\leftarrow$ max(delay $- 1, 0$)
  If delay $= 0$ and runqueue is not empty:
    Send (DELAY, 1), 1 ⓣ to $\mathcal{W}$ and receive (OK,) back from $\mathcal{W}$
    delay $\leftarrow 1$
  Send (ADVANCE,), 1 ⓣ to $\mathcal{W}'$
On input (LEAKS, leaks) from $\mathcal{W}$:
  Execute parse_leaks(leaks)
On input (OUTPUTS, partyoutputs) from $\mathbf{P_i}$':
  Execute party_handler(partyoutputs, $\mathbf{P_i}'$)
On input msg from $\mathcal{D}'$:
  Execute adv_handler(msg)
get_leaks():
  Send (READ,) to $\mathcal{W}$ and pass output to parse_leaks
parse_leaks(leaks):
  For leak in leaks:
    If leak is an eventual code execution leak:
      Append leak to idealqueue
      If leak is about an value input to $\mathcal{F}_{\mathsf{ABB}}$ from an honest $\mathbf{P_i}$:
        Direct the simulated party $\mathbf{P_i}$' to input 0 to Π'_{online}
      If leak is about an honest $\mathbf{P_i}$ creating contract $\mathcal{C}$:
        Direct the simulated party $\mathbf{P_i}$' to add $\mathcal{C}$ to Π'_{online}
    Else if leak is outputs from $\mathcal{F}_{ABB}$:
      outputs$'$ $\leftarrow$ outputs included in leak
      If |outputs$'$ $>$ outputs|: outputs $\leftarrow$ outputs$'$

Figure A.5: Simulator for ABB

$\mathcal{S}_{\mathsf{ABB}}$ (continued)

party_handler(partyoutputs, $\mathbf{P_i}'$):
  For output in partyoutputs:
    If output is related to defining a contract or a variable:
      pos ← position in BC according to the output message
      If pos = |BC| and output is related to defining a contract:
        $\mathcal{C}$ ← contract being created according to output
        queue_pos ← position in idealqueue for creating $\mathcal{C}$
        If queue_pos = ⊥:
          Get sender $\mathbf{P_j}$ of $\mathcal{C}$ according to output
          Direct $\mathbf{P_j}$ to input $\mathcal{C}$ to $\mathcal{F}_{\mathsf{ABB}}$ - $\mathbf{P_j}$ will be corrupt
          get_leaks()
          queue_pos ← position in idealqueue
        For each corrupt $\mathbf{P_i}$:
          Create ContractHandler for $\mathcal{C}$
          Append handler to corrupthandlers
        execute(queue_pos)
      If pos = |BC| and output is related to defining a variable:
        id, $\mathbf{P_j}$ ← contract id and input sender according to output
        queue_pos ← position in idealqueue for (id, $\mathbf{P_j}$)
        tx ← $\mathcal{F}'_{\mathsf{Atomic}}$.BC[pos] (note $\mathcal{F}'_{\mathsf{Atomic}}$ is in $\mathcal{S}_{\mathsf{ABB}}$'s memory)
        (x − r), label ← from inside tx
        If queue_pos = ⊥:
          r ← from inside $\mathcal{F}'_{\mathsf{Offline}}$, using key label
          Direct $\mathbf{P_j}$ to input x, id to $\mathcal{F}_{\mathsf{ABB}}$
          get_leaks()
          queue_pos ← position in idealqueue
        For each corrupt $\mathbf{P_j}$:
          Get $[\![\mathsf{r}]\!]_t^j$ from inside $\mathcal{F}_{\mathsf{Offline}}$ using key label
          Compute $[\![\mathsf{x}]\!]_t^j \leftarrow (x - r) + [\![\mathsf{r}]\!]_t^j$
          Input $[\![\mathsf{x}]\!]_t^j$ to appropriate corrupt contract handler
        execute(queue_pos)
      queue_pos ← position in idealqueue for GenOutputs(pos) for $\mathbf{P_i}$
      execute(queue_pos)
    Else if output related to producing an output:
      queue_pos ← position in idealqueue for delivering output to $\mathbf{P_i}$
      execute(queue_pos)
  queue_pos ← position in idealqueue for delivering all outputs to $\mathbf{P_i}$
  Pop idealqueue[queue_pos] and send EXECUTE, queue_pos to $\mathcal{W}$
execute(queue_pos):
  Pop idealqueue[queue_pos] and send EXECUTE, queue_pos to $\mathcal{W}$
  On receipt (OK,), get_leaks()

Figure A.6: Handler for outputs from simulated parties

$\mathcal{S}_{\mathsf{ABB}}$ (continued)

adv_handler(msg):
  If msg is from $\mathcal{F}'_{\mathsf{Async}}$ and is not an share of an output:
    Pass msg to appropriate corrupt contract handler
  If msg is from $\mathcal{F}'_{\mathsf{Async}}$ and is a share of an output:
    If a polynomial has been defined for the output referenced in msg:
      Replace the share in msg with the corresponding share from the generated polynomial
    If a polynomial has not been generated:
      Input messages from the simulated runqueue until all simulated corrupt handlers for this contract produce an output share.
      Generate and store a t-degree polynomial consistent with corrupt output shares and the actual output value
      Replace the share in msg with the corresponding share from the generated polynomial
  Forward msg to $\mathcal{Z}$
Corrupt ContractHandlers:
  These are the same ContractHandlers from Π_{Online}, except that:
    Any messaged generated for $\mathcal{F}_{\mathsf{Offline}}$ are forwarded to $\mathcal{D}'$, directing it to direct the simulated corrupt party to send the message to the simulated $\mathcal{F}'_{\mathsf{Offline}}$. Outputs are routed into ContractHandler
    Any messages generated for $\mathcal{F}_{\mathsf{Async}}$ are suppressed. Any output shares generated are stored to later generate consistent polynomials and simulated honest output shares.

Figure A.7: Handler for simulated adversary messages and view of corrupt parties

REFERENCES

[1] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols." Cryptology ePrint Archive, Report 2000/067, Aug. 2019, https://eprint.iacr.org/2000/067.

[2] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols." in *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2001, pp. 136—145.

[3] J. Katz, U. Maurer, B. Tackmann, and V. Zikas, "Universally composable synchronous computation," in *Theory of Cryptography, 10th Theory of Cryptography Conference, TCC 2013*, Mar. 2013, pp. 477–498.

[4] I. Damgård and J. B. Nielsen, "Universally composable efficient multiparty computation from threshold homomorphic encryption," in *Advances in Cryptology - CRYPTO 2003*, D. Boneh, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 247–264.

[5] R. Cohen, "Asynchronous secure multiparty computation in constant time," in *Public-Key Cryptography – PKC 2016*, C.-M. Cheng, K.-M. Chung, G. Persiano, and B.-Y. Yang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 183–207.

[6] S. Coretti, J. Garay, M. Hirt, and V. Zikas, "Constant-round asynchronous multi-party computation based on one-way functions," in *Advances in Cryptology – ASIACRYPT 2016*, J. H. Cheon and T. Takagi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 998–1021.

[7] G. Bracha, "An asynchronous [(n-1)/3]-resilient consensus protocol," in *Proceedings of the third annual ACM symposium on Principles of distributed computing*. ACM, 1984, pp. 154–162.

[8] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. Miller, "Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3319535.3354238 pp. 887–903.

[9] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979. [Online]. Available: http://doi.acm.org/10.1145/359168.359176

[10] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols." in *ACM Conference on Computer and Communications Security*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 31–42.

[11] "Ethereum." [Online]. Available: https://ethereum.org/