Hidden $\boldsymbol{\Delta}$-fairness: A Novel Notion for Fair Secure Two-Party Computation

Saskia Bayreuther[1], Robin Berger[1], Felix Dörre[1], Jeremias Mechler[1], and Jörn Müller-Quade[1]

KASTEL, Karlsruhe Institute of Technology, Karlsruhe, Germany
`firstname.surname@kit.edu`

Abstract. Secure two-party computation allows two mutually distrusting parties to compute a joint function over their inputs, guaranteeing properties such as *input privacy* or *correctness*.
For many tasks, such as joint computation of statistics, it is important that when one party receives the result of the computation, the other party also receives the result. Unfortunately, this property, which is called *fairness*, is unattainable in the two-party setting for arbitrary functions. So weaker variants have been proposed.
One such notion, proposed by Pass *et al.* (EUROCRYPT 2017) is called *Δ-fairness*. Informally, it guarantees that if a corrupt party receives the output in round r and stops participating in the protocol, then the honest party receives the output by round $\Delta(r)$. This notion is achieved by using so-called *secure enclaves*.
In many settings, Δ-fairness is not sufficient, because a corrupt party is *guaranteed* to receive its output before the honest party, giving the corrupt party an advantage in further interaction. Worse, as Δ is known to the corrupt party, it can abort the protocol when it is most advantageous.
We extend the concept of Δ-fairness by introducing a new fairness notion, which we call *hidden Δ-fairness*, which addresses these problems. First of all, under our new notion, a corrupt party may not benefit from aborting, because it may not, with probability $1/2$, learn the result first. Moreover, Δ and other parameters are sampled according to a given distribution and remain unknown to the participants in the computation.
We propose a 2PC protocol that achieves hidden Δ-fairness, also using secure enclaves, and prove its security in the Generalized Universal Composability (GUC) framework.

Keywords: Two-party computation · trusted computing · Δ-fairness.

1 Introduction

Secure two-party computation (2PC) allows two mutually distrusting parties to jointly compute a function over private inputs by exchanging messages with each other. 2PC has applications in settings such as auctions or comparing of statistics. Even if one of the parties is corrupt, properties such as correctness or privacy of the inputs can be achieved. Another desirable but not usually guaranteed property in 2PC is fairness. Fairness guarantees, colloquially speaking, that either *all* parties receive the output, or *no* party does.

However, fairness as stated above, also known as *complete fairness*, is hard to achieve, and in the case of 2PC and for arbitrary (efficiently computable) functions, impossible to realize [10]. Gordon *et al.* showed that certain non-trivial functions can be computed in the two-party setting with complete fairness [14] under suitable cryptographic assumptions. It has since been an open question whether this result also extends to other functions. Thus, various weaker versions of fairness have been introduced over the years which aim to improve on efficiency and the range of computable functions. The research in fair multi-party computation (MPC) reaches back to the 1980s [19,10], and has addressed a wide range of fairness notions, from *complete fairness* [14,9,11], *partial fairness*[13,1], and *gradual release fairness* [4] to *fairness with penalties* [3].

Δ-fairness. As opposed to the above notions, which are detached from any measure of time, we address *Δ-fairness*, originally proposed by Pass *et al.* together with a Δ-fair 2PC protocol [17], π_Δ. Here, fairness is linked to time which is measured in rounds. In a protocol that operates in rounds, Δ-fairness is achieved if one party in that protocol gets the output in r rounds and the other party gets the output after at most $\Delta(r)$ rounds for a polynomial function Δ.

Pass *et al.*'s protocol π_Δ relies on the ideal trusted computing interface $\mathcal{G}_{att}$ [17]. $\mathcal{G}_{att}$ is a globally shared trusted computing functionality in the universal composability with global setup (GUC) framework. With a globally shared functionality, all parties access the same instance of this functionality. A secure processor, modelled with $\mathcal{G}_{att}$, guarantees confidentiality and integrity of the programs executed in isolation (called enclaves) and the stored secrets, even if the host is malicious. Secure processors also provide a feature called *remote attestation*, where any party can confirm the integrity of the executed program. Similar to modern secure processors such as Intel SGX [12], $\mathcal{G}_{att}$ features *anonymous* remote attestation. The implementation in Intel SGX uses Direct Anonymous Attestation [5] to allow the revocation of individual processors in case of a compromise while still preserving anonymity. $\mathcal{G}_{att}$ models anonymous remote attestation by sharing the same private key between all secure processors to create an attestation, hiding the signing party.

In many protocols achieving complete fairness, in case of a premature abort by the adversary, the output is set to a trivial value and only if the adversary behaves honestly until the output is available, all parties receive an output. With Δ-fairness, despite a premature abort by the adversary, the honest party is guaranteed to receive an output eventually. However, Δ-fairness as proposed by Pass *et al.* has the major disadvantage that, by definition, the current delays are known to the parties and the corrupt party receives the output before the honest party with certainty, which may result in practical advantages for the adversary when the remaining runtime of the protocol can be manipulated.

Consider the example where two parties each decide to buy or sell stocks based on a (secret) analysis of public data. To make a better decision, they want to compare their analyses and calculate how much profit they would make. They do so via the Δ-fair two-party computation protocol. If their predictions match, they both want to take action (*i.e.*, buy or sell the stocks). Otherwise, no action will be taken. (In this example, we assume that a corrupt input can stop the other party, but not change their behaviour.) The time at which the result of the comparison is learned is crucial, as the earlier one buys/sells, the better the price. If the corrupt party receives the output before the honest party with certainty, the corrupt party may even break the agreement to buy or sell the stocks all alone. Further, if a party aborts the computation prematurely such that it receives the result before a deadline, *e.g.* before the stock market closes and the other party would receive the result after the deadline, the aborting party has a major advantage in its buying or selling decision.

Hidden Δ-fairness. In this paper, we address the problem of adversarial advantage in Δ-fair protocols. We introduce a new variant of Δ-fairness, called *hidden Δ-fairness*. Informally, hidden Δ-fairness differs from standard Δ-fairness in that the delay is not known to either party, by hiding the parameters inside the enclave instances of $\mathcal{G}_{att}$. If a 2PC protocol achieves hidden Δ-fairness, the corrupt party can receive the output in r rounds and either delay the honest party's output until for $\Delta(r)$ rounds or the honest party also receives the output by in r rounds. However since the concrete values of r and $\Delta(r)$ are hidden, it does not learn how much time it has, and its advantage is reduced compared to standard Δ-fairness. Using $\mathcal{G}_{att}$, we can realize hidden Δ-fairness efficiently.

In the above example, if the adversary has no knowledge about the time at which it and the other party receive the output, and if it does not receive the result before the honest party with certainty, the advantage of the adversary declines. The adversary could receive the result at a time at which the stock prices may no longer match their analysis. Since the honest party may receive the output at the same time as the adversary, buying or selling the stocks all alone might no longer be feasible.

For a two-party computation protocol with an adversary corrupting at most one party, we define *hidden Δ-fairness* as follows.

Definition 1 (Hidden Δ-fairness, informal). *If a 2PC protocol operates in rounds and Δ is a polynomial unknown to both parties,* hidden Δ-fairness *guarantees that if the corrupt protocol party receives the output in r rounds, the honest party receives the output either in r rounds or at least in $\Delta(r)$ rounds.*

We further introduce a protocol $\pi_{h\Delta}$, which allows two parties to compute any efficiently computable function while achieving hidden Δ-fairness. We base our protocol on Pass *et al.*'s protocol π_{Δ}.

When using trusted computing, *i.e.* enclaves, as opposed to cryptographic tools like garbeled circuits for MPC protocols, most of the complexity vanishes. However, a trivial solution, that may come to mind first, is not possible. Consider a 2PC protocol where two parties with access to an enclave each wish to compute a joint function on private inputs. Both parties would send their input to its enclave. One enclave sends its value to the other enclave over a previously established secure channel. The receiving enclave computes the function and sends the output back to the sending enclave; afterwards both enclaves return the output to the parties respectively. As long as both parties are honest and the receiving party is trustworthy, this approach works. In two-party computation with a dishonest majority, the above protocol can be exploited if the receiving enclave is corrupt. As communication between the enclaves goes through the parties, the malicious party can drop the message containing the output (*i.e.* aborting the protocol prematurely) meant for the honest party while receiving the output itself, breaking fairness. Additionally, Pass *et al.* showed [17], both parties are required to have access to a enclave in order to UC-realize 2PC in the above explained $\mathcal{G}_{att}$ model.

Therefore, some mechanism enforcing that both parties eventually receive the output despite premature abort needs to be established.

We consider a synchronous protocol execution that operates in rounds. In each round, a party receives a message, processes the message to generate a new message, which is sent to other parties participating in this protocol instance. Both $\pi_{h\Delta}$ and π_{Δ} utilize the parameters δ and Δ. δ is the delay, such that, after being initialized and decreased each round, upon expiration, a party can receive the output of the computation. Δ, as stated above, is a polynomial function by which the delay δ is reduced during the protocol.

In π_{Δ}, where both parties start with the same delay, if the first party aborts the protocol prior to sending a message, it prevents second party to enter a new round while the first party can execute the protocol for an additional round. Since delays are updated once each round, the aborting party is in the lead. In $\pi_{h\Delta}$ the adversary only receives the output before the honest party with a probability of 50% which is achieved by carefully choosing the initial delays. In addition, we limit the adversary's ability to predict the remaining delay δ based on the initial delay and Δ by randomly choosing the initial delay and Δ instead of using predefined values and keeping the parameters hidden in the parties secure processors. From a game theory perspective, hidden-Δ fairness improves the fairness guarantees in comparison to Δ-fairness. We show the security of our protocol using the generalized universal composability framework [7].

In the following, we give an informal description of our protocol $\pi_{h\Delta}$.

1.1 Protocol Description

We assume that two parties have access to a secure processor which is used to start an enclave each. Upon given the private inputs, the enclaves are used to compute the function on the inputs and release the output to the hosts after a certain amount of time. To achieve that both enclaves release the output at the same time, the enclaves jointly negotiate the program parameters.

- Upon initialization, both enclaves individually draw a random initial delay $\delta_i[0]$, a polynomial function $\Delta_i(\cdot)$ and a bit $b_i, i \in \{0, 1\}$.
- After establishing a secret channel and calculating a common coin $c \in \{0, 1\}$ from both bits b_0 and b_1, the enclaves commonly determine the parameters $\delta'[0] := \delta_c[0]$ and $\Delta'(\cdot) := \Delta_c(\cdot)$ used for this run.

- Enclave e_c sets its initial delay $\delta[0]$ to ${\delta'[0] \cdot r}/{\Delta'(1)}$, while enclave e_{1-c} sets its initial delay to $\delta'[0]$.
- Now, the enclaves start exchanging acknowledgement (`ack`) messages. In each round r, each enclave receives a message, updates the remaining delay $\delta[r] := {\delta[r-1] \cdot r}/{\Delta'(r)}$ and creates a message for the opposite enclave.
- If $\delta[r]$ of one party drops below a threshold, *both* parties now are able to retrieve their output.

To illustrate the above example of stock trading, consider the simple instance of the above introduced protocol in Fig. 1. Imagine both parties are able to buy or sell stocks until a deadline denoted with $r = 6$. On the left hand side of Fig. 1, both parties are honest and run the protocol as described. When the first party's delay drops below 1, *both* parties are able to retrieve the output in round 4 and thus have time to trade stocks until round 6. On the right hand side, p_0 is corrupt and does not send its `ack` message in round 2. Honest party p_1 sends its round 2 message to p_0, meaning that p_0 can run the protocol for another round. Since the delay of p_0 is 1 at this time, it can receive the output until round 6 and be able to trade stocks. The honest party, with delay $\delta_1[2] = 16$, will *not* be able to do so until the deadline is over. If the corrupt party had aborted one round earlier, neither party could trade stocks with delays $\delta_0[2] = 4$ and $\delta_1[1] = 64$ respectively. If the corrupt party had aborted one round later, both party would be able to trade stocks, as the delay of the corrupt party is < 1. As the parameters Δ and $\delta[0]$ are hidden, the time at which an abort is most beneficial to p_0 is unknown.

1.2 Contribution

In this paper we make the following contributions. We address various disadvantages of the original Δ-fairness. First, we define a new notion of fairness, namely hidden Δ-fairness based on Δ-fairness. We do so via the ideal functionality $\mathcal{F}^{f,h\Delta}$ which hides the current delay from both parties, allows for a variable output order and delays the output for both parties. We also define the accompanying protocol $\pi_{h\Delta}$, that implements a hidden Δ-fair two-party computation protocol and that securely realizes $\mathcal{F}^{f,h\Delta}$ in the GUC framework.

1.3 Related Work

Many secure MPC protocols require cryptographic tools like garbled circuits or secret sharing. Another promising option for providing confidentiality and privacy in secure MPC is trusted computing. With formalising trusted computing devices, or trusted execution environments (TEEs) through a globally shared ideal functionality, Pass *et al.* [17] laid the groundwork for provable MPC using trusted computing. They show that two-party computation in this setting requires that both involved parties need to be equipped with a TEE.

Similarly, Barbosa *et al.* [2] addresses formalizing trusted computing with the focus on composition and attestation in systems relying on hardware-based

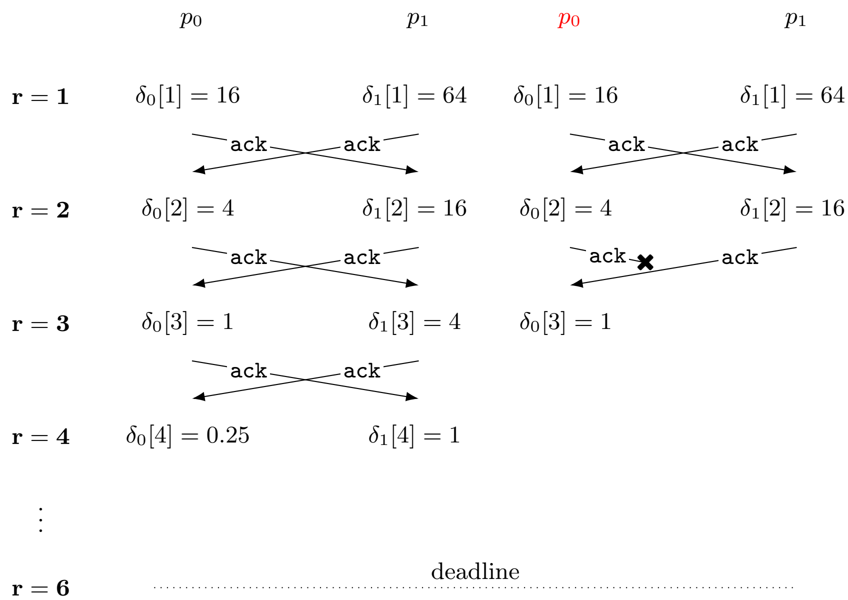

Fig. 1. $c = 0, \Delta(r) = 4, \delta[1] = 256$. Left: both parties honest, right: p_0 corrupt

trusted computing. Their work only provides formalization posing application to MPC as future work.

Aside from this formalization of trusted computing in MPC, other work shows that TEEs can be used to provide the basis for MPC in practice. Similar to Pass *et al.*, Choi *et al.* [8] explores the adaption of TEEs in MPC. They focus on practical solutions for utilizing the security guarantees of trusted computing as well as the practical differences between commercially available TEEs. They also survey existing work utilizing TEEs in MPC, presenting a comparison of different techniques used in MPC, addressing properties like integrity of code and data, and overhead. While our work pushes the boundaries of achievable fairness with idealized trusted computing, their work does not specifically address fairness but rather other limiting factors like mobile friendly MPC.

Paul *et al.* [18] propose a MPC protocol based on TEEs and public ledgers. They provide an attack on existing work [9], demonstrating that the fairness guarantee vanishes if the underlying MPC protocol does not guarantee correctness. Based on this attack, they propose a fair MPC protocol that is resistant against the aforementioned attack. In comparison to our work, their approach requires a public ledger in addition to TEEs on every party.

1.4 Outline

In Section 2 we briefly introduce the G(UC) framework, describe the concept of Trusted Computing and define the global functionality $\mathcal{G}_{att}$. We also give definitions of several building blocks. In Section 3.1 we define the hidden Δ-fair 2PC ideal functionality $\mathcal{F}^{f,h\Delta}$. The protocol $\pi_{h\Delta}$ is described and defined in Section 4. The security proof is given in Section 5 using the GUC framework. Section 6 concludes this work.

2 Preliminaries

This section introduces the (G)UC framework, trusted computing and the ideal functionality $\mathcal{G}_{att}$ and defines building blocks.

2.1 Notation

Let $\lambda \in \mathbb{N}$ denote the security parameter. For some hybrid H_i, let out_i denote its output. Let $\mathsf{negl}(\lambda)$ denote an unspecified negligible function in λ. For some set X, let $x \overset{\$}{\leftarrow} X$ denote that x is sampled uniformly at random from X. For $a \leq b \in \mathbb{N}$, let $[a, b] = \{a, \ldots, b\}$.

2.2 (Generalized) Universal Composability

In this paper, we use a variant of the well-known Universal Composability (UC) framework [6] called *Generalized Universal Composability* (GUC) [7]. In the following, we give a very short and intentionally incomplete introduction to (G)UC security. For a complete treatment, we refer the interested reader to the respective papers.

Following the real-ideal paradigm [16], Universal Composability allows to analyze the security of a probabilistic polynomial-time (PPT) protocol π by comparing it to an *ideal functionality* $\mathcal{F}$, which captures the task to be performed, as well as the desired security guarantees.

Real Execution. In more detail, the protocol π (whose execution is called the *real execution*) is executed in the presence of an adversary $\mathcal{A}$, modelled as a PPT Turing Machine, and an environment $\mathcal{Z}$, which is modelled as a non-uniform PPT Turing Machine. When $\mathcal{Z}$ is started, it gets an input consisting of the security parameter in unary notion, as well as some (not necessarily computable) non-uniform advice. $\mathcal{Z}$, serving as an interactive distinguisher, chooses and provides the inputs to the protocol parties and also receives their output. Throughout the execution, it may freely communicate with the adversary, which may corrupt an arbitrary subset of protocol parties. Corrupted parties are jointly controlled by the adversary and may arbitrarily deviate from the protocol, capturing the setting of *active* or *malicious* or *byzantine* corruption. Moreover, the adversary controls the communication network, which we assume to provide ideally authenticated communication. In particular, the adversary may report all messages exchanged between protocol parties to the environment. Thus, depending on the adversary, the environment may have full control and information of the execution: It knows all inputs and outputs and may observe all communication. Furthermore, the adversary may simply receive commands from the environment (*e.g.* which parties to corrupt or which messages to send on behalf of corrupted parties). The adversary may also report all information it receives to the environment. This adversary is called the *dummy adversary* $\mathcal{D}$. At the end of the execution, the environment outputs a single bit.

Ideal Execution. In the execution with the ideal functionality $\mathcal{F}$, the protocol parties do not exchange messages with each other. Instead, they directly send their inputs to the ideal functionality, which performs the desired task by definition. Then, it returns the results to the parties, which output them to the environment. As the ideal functionality is incorruptible and its communication with the parties ideally secure, this *ideal execution* is secure by definition. Also part of the execution with an ideal functionality is the adversary. Like in the real execution, it can corrupt parties to learn their inputs and outputs. It may also interact with the ideal functionality through interfaces on behalf of the corrupted parties as well as through interfaces provided for the adversary. These interfaces capture adversarial influence that cannot be ruled out in principle, *e.g.* the delay of outputs, and depend on the functionality and the model of execution.

Proving Security. To prove the security of π, *i.e.* that it realizes the ideal functionality $\mathcal{F}$, it is necessary to prove that no PPT environment can distinguish between an interaction with π and an adversary and interaction with $\mathcal{F}$ and an adversary. If this holds, then all properties guaranteed by $\mathcal{F}$, which is secure by definition, carry over to the execution of π—otherwise, the executions would not be indistinguishable. At first glance, this may seem impossible to prove, as in the execution of π, the protocol parties execute messages, whereas in the execution of $\mathcal{F}$, no messages are exchanged.

To bridge this gap, the proof of security entails to prove the existence of a so-called *simulator*, which acts as the adversary in the ideal execution. The task of the simulator is to *simulate* the execution of the real protocol—while actually interacting with the ideal functionality $\mathcal{F}$. In particular, the simulator must simulate the messages the protocol parties would send in the real execution. However, this must be done with what little information the ideal functionality provides. In particular, the simulator usually does not know the inputs (and often the outputs) of the honest parties. Nevertheless, it must be able to simulate their messages in an indistinguishable way.

Realizing an Ideal Functionality. The above definition can be captured in the following, still informal, definition of realizing an ideal functionality.

Definition 2 (Realizing an Ideal Functionality, informal). *Let π be a PPT protocol and let $\mathcal{F}$ be an ideal functionality. We say that π* (UC-) realizes *$\mathcal{F}$ if* *for every PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that for every PPT environment $\mathcal{Z}$, the output of $\mathcal{Z}$ in the execution with π and $\mathcal{A}$ and in the execution with $\mathcal{F}$ and $\mathcal{S}$ is computationally indistinguishable.*

Synchronous Execution. Similar to [17], we assume a synchronous and round-based model of execution. In particular, we assume that the environment calls each party once each round. This allows the parties to keep track of the current round. We also assume that all messages sent by honest parties in round r are delivered at the beginning of round $r + 1$. If a party receives no message from the opponent party in one round, it assumes the opponent party has halted the execution. The party then discards all future incoming messages.

Time and Clocks in (G)UC. In our protocol and functionalities, we make use of the "time" passed since a protocol has started. Both functionalities and protocols measure time passed in rounds. One round is a "tick" on a global network clock. We say that a functionality is *clock-aware* if it can query the current time.

2.3 Trusted Computing

We use the concept of *trusted computing*, *i.e.*, we assume the existence of a secure processor that provides integrity and confidentiality of loaded programs, executions of programs, and stored information, and that provides anonymous attestation over its programs and data. An instance of a program on a secure processor we call an *enclave*.

We utilize the secure processor abstraction $\mathcal{G}_{att}$ (see Definition 3). $\mathcal{G}_{att}$ is a global ideal functionality, *i.e.* accessible from multiple protocols. It captures the anonymous attestation abstractions which is the core of secure computing and that is implemented by actual trusted execution processors, *e.g.* Intel SGX [12], Arm Trustzone and others. The immediate and uninterruptable inputs/outputs between a party and $\mathcal{G}_{att}$ model the fact that an adversary controlling the network still cannot prevent a party from interacting with its own enclave. Therefore, $\mathcal{G}_{att}$ models parties that have a secure processor embedded in their system. The secure processor cannot communicate directly with the outside world, but only interacts with the parties hosting the secure processor, which can then relay messages to other parties. $\mathcal{G}_{att}$ has two stateful enclave operations, `install` and `resume` which we explain below.

On initialization, $\mathcal{G}_{att}$ generates a pair of keys $(\mathsf{mpk}, \mathsf{sk})$ of signature scheme Σ. Every party can query the public key mpk. Deviating from [17], we allow any party *and* the adversary to install (and run) enclaves. Any party that has an attested secure processor, *i.e.* that can run enclaves can call enclave operations and produce attestations under secret key sk. This is a modification to the original definition, where only registered parties can run enclaves. Note that the same key pair $(\mathsf{mpk}, \mathsf{sk})$ is used to sign (and verify) *every* call to an enclave. As the signature contains the enclave ID, but not the identity of its owner, this models *anonymous attestation.* Using the `install` operation, a party p can load a program prog with identifier idx on its local processor. Upon installing, $\mathcal{G}_{att}$ chooses a random identifier *eid* for each installed enclave program. $\mathcal{G}_{att}$ holds the current state of each installed program in a field T. After installation, p can resume the program prog with input inp statefully using `resume`. $\mathcal{G}_{att}$ executes prog using the current status mem and the input inp and sends outp and a signature σ over the program, output, idx and eid back to p.

Formally, $\mathcal{G}_{att}$ is defined as follows:

Definition 3 (Global functionality $\mathcal{G}_{att}$ (based on [17])). *$\mathcal{G}_{att}$ interacts with a set of parties and is parameterized with an EUF-CMA secure signature scheme Σ. On initializing, draw $\mathsf{mpk}, \mathsf{sk} \xleftarrow{\$} \Sigma.\mathsf{KeyGen}$ and set $T = \emptyset$. When a party p or the adversary requests the master public key, send mpk to p. $\mathcal{G}_{att}$ has the following stateful enclave operations:*

- *When receiving* `install`(idx, prog) *from a party p or the adversary, if p is honest, assert $idx = sid$, and generate nonce $eid \in \{0,1\}^\lambda$, store $T[eid, p] := (idx, \mathsf{prog}, \vec{0})$*
- *When receiving* `resume`(eid, inp) *from p, if there exists no entry $(idx, \mathsf{prog}, \mathsf{mem}) := T[eid, p]$, abort. Otherwise, calculate $(\mathsf{outp}, \mathsf{mem}') = \mathsf{prog}(\mathsf{inp}, \mathsf{mem})$, update $T[eid, p] := (idx, \mathsf{prog}, \mathsf{mem}')$, calculate signature $\sigma := \Sigma.\mathsf{Sig}_{\mathsf{msk}}(idx, eid, \mathsf{prog}, \mathsf{outp})$ and send (outp, σ) to p.*

The environment $\mathcal{Z}$ can access $\mathcal{G}_{att}$ acting as a corrupt party or acting as an honest party for non-challenge protocol instances with different session identifiers than challenge sid.

2.4 INT-CTXT Security

In the following, we will state the definition of INT-CTXT security, which informally captures the property that an adversary, given a set of ciphertexts with plaintexts of its choice, cannot create a new ciphertext that can be successfully decrypted.

Definition 4 (INT-CTXT Security). *Let $\mathsf{AE} = (\mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme. Let $\mathrm{Exp}^{\mathrm{IND-CTXT}}_{\mathcal{A},\mathsf{AE}}(\lambda, z)$ be the output of the following experiment:*

1. *Generate a key $\mathsf{sk} \xleftarrow{\$} \{0,1\}^\lambda$. Furthermore, initialize $C = \emptyset$ and $\mathrm{win} = \mathrm{false}$.*
2. *Let $\mathcal{O}_{\mathsf{Enc}}(\mathsf{sk}, m)$ denote the following oracle:*
   (a) *$c \leftarrow \mathsf{Enc}(\mathsf{sk}, m)$ and add c to C.*
   (b) *Return c.*
3. *Let $\mathcal{O}_{\mathrm{Vfy}}(\mathsf{sk}, c)$ denote the following oracle:*
   (a) *If $c \notin C$ and $\perp \neq m \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$, set $\mathrm{win} = \mathrm{true}$.*
   (b) *Return $m \neq \perp$.*
4. *Execute $\mathcal{A}^{\mathcal{O}_{\mathsf{Enc}}(\mathsf{sk},\cdot), \mathcal{O}_{\mathrm{Vfy}(\mathsf{sk},\cdot)}}(1^\lambda, z)$.*
5. *If $\mathrm{win} = \mathrm{true}$, return 1. Otherwise, return 0.*

We say that AE is INT-CTXT-secure *if for every PPT adversary $\mathcal{A}$, there exists a negligible function negl such that for every $\lambda \in \mathbb{N}$ and every $z \in \{0,1\}^*$, it holds that $\Pr[\mathrm{Exp}^{\mathrm{INT-CTXT}}_{\mathcal{A},\mathsf{AE}}(\lambda, z) = 1] \leq \mathsf{negl}$.*

3 Hidden $\boldsymbol{\Delta}$-fair Functionality $\mathcal{F}^{f,h\Delta}$

In this section we define our requirements for the variant of the Δ-fairness property, which we call *hidden Δ-fairness*, via the ideal functionality $\mathcal{F}^{f,h\Delta}$.

3.1 Ideal Functionality $\mathcal{F}^{f,h\Delta}$ for hidden $\boldsymbol{\Delta}$-fair 2PC

In short, Δ-fairness in multi-party computation means that, when Δ is a function and time can be measured in rounds, when the malicious party receives the output of the computation in r rounds, the honest party receives the output in at most round $\Delta(r)$. In our functionality, this is achieved via a delay, that is initially set to an exponentially high value.[1] Each round the delay is reduced according to Δ, *i.e.* divided by Δ. When the delay has expired, *i.e.* dropped below a threshold, the parties can obtain the output of the computation.

As we motivated in the introduction, the original Δ-fairness (as given in [17, Fig. 10]) enables the adversary to exploit the knowledge of the delay and the guaranteed first output delivery in some applications. These advantages, such as the knowledge of exact delays, enables and incentivizes the adversary to abort the protocol prematurely at a specific point in time. We address these issues in this section by stating three requirements for our notion *hidden Δ-fairness* (hΔ). Afterwards, we show how these requirements are implemented in the proposed hidden-Δ functionality $\mathcal{F}^{f,h\Delta}$. We have the following requirements.

1. *Hidden delay*: Current delay is hidden from the adversary.
2. *Variable output order*: The adversary cannot always receive the output before the honest party does.
3. *Delayed output*: Prevent the adversary from receiving the output immediately when requested.

Our functionality achieves these requirements as follows.

Hidden delay We achieve this requirement by choosing the initial delay $\delta[0]$ and the delay reduction function $\Delta(\cdot)$ at random from appropriately defined sets. These parameters are drawn by the functionality and are not passed to the adversary. These parameters define the delay in each round.

Variable output order The functionality chooses a bit b at random. The adversary receives the output in r rounds when the delay is below a certain threshold. The honest party, depending on the bit b, either also receives the output in r rounds or at least in $\Delta(r)$ rounds.

[1] The functionality offers the option to return no output to the honest party in case of a very early abort. Since the initial delay is exponentially high, the polynomially bounded adversary never actually receives the output.

Delayed output The adversary can issue to (`abort`, sid), upon which the functionality stores the current delay $d = \delta[r]$ as well as the remaining delay of the honest party, denoted as D. Upon issuing (`output`, sid), depending on the bit b and if more time has passed than the stored values d resp. D, the output is returned to the calling party. If no abort took place, the output is available to both parties when the current delay is below a threshold.

In the following we provide a detailed description of the functionality $\mathcal{F}^{f,h\Delta}$ as well as the formal definition (Definition 5). The functionality uses several variables that are specific for one session, each denoted with a session identifier sid. For clarity, we abbreviate any variable n_{sid} with n. To distinguish the parameters drawn by the functionality and in the protocol, in the following, we use the notation $\Delta_{\mathcal{F}}$ and $\delta_{\mathcal{F}}$.

$\mathcal{F}^{f,h\Delta}$ works as follows. The functionality operates in rounds. It internally holds state of the current round, denoted as r, which is periodically increased, starting at $r = 0$. In each round the functionality updates the delay as explained below. Each party is called once a round to call one of the functionality's functions, also explained below. After receiving the inputs from both parties, the functionality randomly draws a function $\Delta_{\mathcal{F}}(\cdot) \xleftarrow{\$} f_{\hat{a}}$ where $f_{\hat{a}} := \{\Delta_{\mathcal{F}}(r) : r \mapsto \hat{a} \cdot r\}_{\hat{a} \in [2,\lambda^2]}$. The functionality also randomly draws a bit b and initializes a flag $z = \perp$. The functionality uses $\Delta_{\mathcal{F}}(\cdot)$ to calculate $\delta_{\mathcal{F}}[r]$ every round.

The parties can call two functions. First, a party can choose to call (`abort`, sid). If this abort takes place before round $r = 5$, the opponent party shall never receive an output. In the case of an abort prior to round 5, the functionality thus sets $z = j$ where p_j is the opponent party. We include this feature due to practical constraints in the real protocol where the adversary can abort the protocol on behalf of a corrupted party before the honest party has had a chance to start the protocol properly, which happens in round 5, preventing the honest party to receive the output. We note that this limitation is also present in the original notion of Δ-fairness and seems to be inherent to the considered setting. However, [17] does not capture this explicitly in the ideal functionality. For the parameters used in [17], this is not necessary, as the honest party would receive the output in the ideal execution after a super-polynomial number of rounds if the adversary aborts early on. As (G)UC executions only take a polynomial number of steps, this output does not actually occur.

If the abort takes place in round $r = 5$ or later, the functionality stores the current round $r^* := r$, the delay $d := \delta_{\mathcal{F}}[r^*]$ and the delay of previous round, *i.e.* $D := \Delta_{\mathcal{F}}(\delta_{\mathcal{F}}[r^*])$. In theory both parties can call (`abort`, sid), however no honest party would benefit from aborting.

The second function of the functionality a party can call is (`output`, sid) to request the output from the functionality. There are two cases. In the first case, (`abort`, sid) has not been called by a party. Then, if $\delta_{\mathcal{F}}[r] < 1$ for current round r and after asserting that $(\mathtt{outp}_0, \mathtt{outp}_1)$ has been stored[2], the functionality returns the respective output to the calling party.

[2] Anytime a party calls the functionality, this assertion is verified. If this assertion fails in any case the functionality stops its execution.

In the second case, in a previous round r^*, (`abort`, sid) has been called. If the corrupt party is calling, it is returned the output if more rounds has passed since round r^* than stored value d. If the honest party, say p_j, is calling (`output`, sid), it is returned the output when more rounds has passed since round r^* than d in case $b = j$, or, when more rounds has passed since round r^* than D in case $b \neq j$. The honest party is only returned the output when (`abort`, sid) has been called after round 4, *i.e.* if $z \neq j$.

Definition 5 (Hidden Δ-fair 2PC functionality $\mathcal{F}^{f,h\Delta}$). *$\mathcal{F}^{f,h\Delta}$ interacts with parties p_0 and p_1 in a session denoted with sid.*

1. *When receiving (*`compute`*, sid,* inp_i*) from p_i where $i \in \{0,1\}$, if p_{1-i} has sent (*`compute`*,* inp_{1-i}*), let $(\mathsf{outp}_0, \mathsf{outp}_1) = f(\mathsf{inp}_0, \mathsf{inp}_1)$. Store $(sid, \mathsf{outp}_0, \mathsf{outp}_1)$. Initialize $z = \perp$.*
2. *Draw random bit $b \xleftarrow{\$} \{0,1\}$, function $\Delta_{\mathcal{F}}(\cdot) \xleftarrow{\$} f_{\hat{a}}$ and an initial delay $\delta_{\mathcal{F}}[0] = 2^x$ where $x \xleftarrow{\$} [\lambda/2, \lambda]$. Set $r = 0$ and periodically increase r. Set $\delta_{\mathcal{F}}[4] := \delta_{\mathcal{F}}[0]$.*
3. *When receiving (*`abort`*, sid) from p_i in round $r = 5$, set $z = 1 - i$.*
4. *In each round from round $r = 5$, calculate $\delta_{\mathcal{F}}[r]$ according to $\Delta_{\mathcal{F}}(\cdot)$*
5. *When receiving (*`abort`*, sid) from p_i in round $r \geq 5$,*
   (a) Assert $(sid, \mathsf{outp}_0, \mathsf{outp}_1)$ has been stored.
   (b) Store $r^* := r$, $D := \Delta_{\mathcal{F}}(\delta_{\mathcal{F}}[r^*])$ and $d := \delta_{\mathcal{F}}[r^*]$.
6. *When receiving (*`output`*, sid) from p_j in round r, proceed as follows*
   - *Assert $(sid, \mathsf{outp}_0, \mathsf{outp}_1)$ has been stored.*
   - *If* `abort` *has not been called and if $\delta_{\mathcal{F}}[r] < 1$, return* outp_j *to p_j.*
   - *Else, if* `abort` *has been called,*
     (a) Let $r' := r - r^*$ denote the rounds passed since `abort` *has been called.*
     (b) If $b = j$, if $r' > d$ and $z \neq j$, return outp_j *to p_j.*
     (c) If $b \neq j$,
       i. If j corresponds to party p_j corrupted by $\mathcal{A}$, if $r' > d$ and $z \neq j$, return outp_j *to p_j.*
       ii. Otherwise, if $r' > D$ and $z \neq j$, return outp_j *to p_j.*
   - *Otherwise, return $\perp$.*
7. *When receiving (*`status`*, sid) from $\mathcal{A}$,*
   - *Return $\perp$ if output returned to less than two parties.*
   - *Return* `finished` *if output returned to both parties.*

4 Hidden Δ-fair Two-Party Computation Protocol

This section describes our proposed protocol $\pi_{h\Delta}$ which follows the structure of the original protocol π_{Δ} (as specified in [17, Figure 12]). The full protocol can be found in Algorithm 2 which internally calls the enclave program in Algorithm 1.

4.1 System Model

In the following we define the system model used in our proposed protocol. Let (p_0, p_1) be a set of two parties participating in a two-party computation protocol that operates in rounds. Let r denote the current round. p_0 and p_1 have a private input inp_0 resp. inp_1. After executing the 2PC protocol, the parties receive the output outp_0 resp. outp_1. Let $\mathsf{AE} = (\mathsf{Enc}, \mathsf{Dec})$ be an IND-CPA- and IND-CTXT-secure symmetric encryption scheme. $(g, p) \xleftarrow{\$} \mathsf{GenGrp}(1^\lambda)$ are generator and group modulus of group $\mathbb{Z}_p^*$ where $|p| = \lambda$ is published by a trusted third party. Let $\Sigma = (\mathsf{Sign}, \mathsf{Ver})$ be an EUF-CMA-secure digital signature scheme.

To define the parameters for the 2PC protocol, let $\delta[r]$ be the *delay field* and $\Delta(r)$ the *delay reduction function*. Let $f_a := \{\Delta(r) : r \mapsto a \cdot r\}_{a \in [2,\lambda]}$ be a family of functions. $\delta[r]$, being recursively defined dependent of Δ, is the remaining run time of the 2PC protocol. $\delta[0]$ is initialized with some positive integer $x \in [\lambda/2, \lambda]$, $\delta[0] := 2^x$. For round r, $\delta[r]$ is defined recursively, $\delta[r] := \delta[r-1] \cdot r / \Delta(r)$.

4.2 Design Requirements

Recall the three goals, hidden delay, variable output order and delayed output, previously defined for $\mathcal{F}^{f,h\Delta}$ in Section 3.1. We describe how the protocol realizes these goals. We also describe the security implications of these realizations. Afterwards, we give a detailed description of our proposed protocol.

Hidden delay Similar to the functionality, the protocol hides the delay from the parties by letting the enclaves choose the initial delay $\delta[0]$ and delay reduction function $\Delta(\cdot)$ randomly. Imagine there is a deadline in the future such that it is beneficial for the adversary when it receives the output before the deadline and the honest party after the deadline. Then there is one round at which it must abort such that this scenario takes place[3]. The hidden delay prevents the adversary from selectively aborting the protocol prematurely at this round and thus brings the probability that the adversary aborts the protocol at its desired point in time to $1/r^*$ where r^* is the total number of rounds if no party aborts. The number of rounds r^* is defined by the initial delay $\delta[0]$ and $\Delta(\cdot)$. If these parameters were known, the adversary could always abort the protocol prematurely such that it has an advantage in the underlying protocol utilizing the 2PC (*e.g.* calculating statistics) by knowing the result early with a significant time margin or such that the honest party receives the result at a point in time where the underlying protocol has already finished.

Variable output order The enclaves draw a random bit each which get combined into a common coin. The coin, unknown to the parties, decides which party obtains the lower delay right from the start. If both parties are honest, the

[3] If the adversary aborts before said round both parties would receive the output after the deadline since the delays are too high, if the adversary aborts after this round both parties would receive the output before the deadline, see Figure 1

protocol makes the output available to both parties at the same time. If one party is corrupt and aborts prematurely, the coin decides if both parties receive the output simultaneously or if the honest party receives the output after the corrupt party in case of a premature abort. This mechanism reduces the probability to receive the output first 50% as opposed to 100% if the output order were fixed in favor of the adversary.

Delayed output Each party's enclave holds state of the current delay. Only if $\delta[r^*] < 1$ in round r^*, a party can obtain the output from the enclave. If no party aborts the protocol prematurely, both parties can obtain the output in round r^*, independent of the common coin c. If a party, say p_i, aborts the protocol prematurely, either both parties can obtain the output in r' rounds if $c \neq i$ or the honest party receives the output at least in $\Delta^2(r')$ rounds. We achieve this by setting the initial delay of p_i to $\frac{\delta[0]\cdot r}{\Delta(r)}$ where $\delta[0]$ is p_{1-i}'s initial delay. Then, in case of an abort, p_{1-i} is "two rounds behind".

4.3 Protocol Description

The protocol works as follows. If any assertion fails the party aborts execution and hands control back to $\mathcal{Z}$.

- On initialization, the parties run a setup phase, where the parties' enclaves commonly decide on a initial delay $\delta[0]$, the delay reduction function $\Delta(\cdot)$ and a common coin c. They also exchange their private inputs for later evaluation.
- If $c = i$, p_i's initial delay is $\delta[1]$. Otherwise, p_i's initial delay is $\delta[0]$ to accommodate a higher delay for p_i.
- After initialization, the parties exchange `ack` messages. In each round an `ack` message is received, the enclave can update its current delay according to $\Delta(\cdot)$. If the delay of p_c has expired, both parties can obtain the output from their enclave.
- If a party aborts and stops sending `ack` messages before $\delta[r] < 1$, the timer decreases only linearly for both parties. When one party received an `ack` message in the current round while the other party did not, the first party's delay is either equal to or lower than the second party's delay by a factor of $\Delta(\Delta(r))/r^2$, depending on the coin c.

In order to illustrate the last point, consider the following example, depicted in Figure 2. After initialization, the parties decided on the parameters $\Delta(r) = 4\cdot r$, $c = 0$, and $\delta[0] = 64$. Since $c = 0$, p_0's initial delay in round 1 is $\delta_0[1] = 16$, whereas p_1's initial delay is $\delta_1[1] = 64$. First, imagine that p_0 is corrupt and does not send its `ack` message, denoted as m_0, to p_1 in round $r = 2$. p_1, acting honestly, sends its `ack` message m_1 to p_0 in round $r = 2$. Then, in round $r = 3$, p_0 can call its enclave using message m_1. After this round, p_0's delay is $\delta_0[3] = 1$. Since p_1 could never enter round $r = 3$, its delay is $\delta[2] = 16$. We have that $\frac{(4r\cdot\delta_0[3])\cdot 4r}{r^2} = 4 \cdot 4 \cdot \delta_0[3] = \delta_1[2]$.

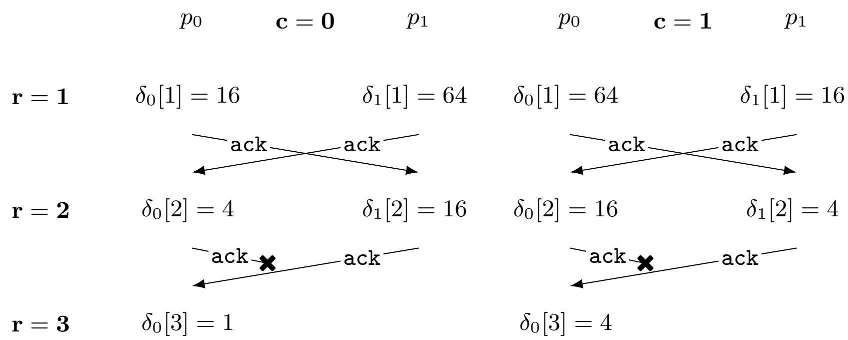

Fig. 2. Simplified protocol procedure of corrupt p_0 and honest p_1 where $\Delta(r) = 4r$ and $\delta[0] = 64$. Left: $c = 0$, right: $c = 1$

Consider the same example except that $c = 1$. Then, $\delta_0[1] = 64$ and $\delta_1[1] = 256$. Again, p_0 can enter round $r = 3$, thus $\delta_0[3] = 4$. p_1's round counter remains at $r = 2$ and $\delta_1[2] = 4$. In this case we have that $\delta_0[3] = \delta_1[2]$.

Note that if a party stops receiving messages and thus stops invoking its enclave functions, the round counter stops progressing and round counters might deviate from the actual time. This is why we included an explicit round counter that is increased in the enclave program instead of the implicit round counter in π_Δ. Upon aborting, the delays only decrease linearly, which is denoted by decreasing δ^* periodically in the protocol. This means that every round a party does not invoke its enclave the delay δ^* is decreased by 1[4].

Now let us look at the different sets f_a resp. $f_{\hat{a}}$ used by the protocol resp. the functionality to draw the function $\Delta(\cdot)$ resp. $\Delta_{\mathcal{F}}$. As demonstrated in the above example, in the protocol $\pi_{h\Delta}$, after p_0 has aborted, the delay for the honest party p_1 is $\Delta(\cdot) \cdot \Delta(\cdot)$ times the delay of p_0 when $c = 0$, *i.e.* the function $\Delta(\cdot)$ applied twice. In order to match this to the functionality, the functionality draws $\Delta_{\mathcal{F}}$ from $f_{\hat{a}}$ where $\hat{a}$ is in $[2, \lambda^2]$ as opposed to a which is in $[2, \lambda]$.

4.4 Parameter Selection and Performance

An appropriate selection of λ depends on the concrete security margin one is able to accept for the event of an adversary managing to obtain the output before a deadline and the honest party obtaining it after the deadline. An appropriate probability might be 2^{-10}, setting $\lambda = 2^{11}$. Inherently, when the protocol chooses the adversary to go first, there is a message where aborting allows the adversary to obtain the output before the deadline and the honest party getting it after. Therefore the security is limited by the number of exchanged protocol messages.

---

[4] In practice this can be implemented via trusted clocks [15] or by relying on digital signatures of an external time stamping server.

Algorithm 1 Enclave part $\mathsf{prog}_{\pi_{h\Delta}}[f, p_0, p_1, i]$ of $\pi_{h\Delta}$, based on original protocol π_Δ by [17]. $f_a := \{\Delta(r) : r \mapsto a \cdot r\}_{a \in [2,\lambda]}$ is a family of functions.

On initialize
- $r := 1$
- $x \xleftarrow{\$} [\lambda/2, \lambda]$
- $\delta[0] := 2^x$
- $\Delta(\cdot) \xleftarrow{\$} f_a$
- $b_i \xleftarrow{\$} \{0, 1\}$
- $Y = (\delta[0], \Delta(\cdot), b_i)$
- $\delta[1] = \frac{\delta[0] \cdot r}{\Delta(r)}$

On input (keyex)
- $\alpha \xleftarrow{\$} \mathbb{Z}_p^*$
- **return** g^α

On input (keygen,g^β)
- assert keyex has been called
- $sk := (g^\beta)^\alpha$
- $ct := \mathsf{Enc}_{\mathsf{sk}}(Y)$
- **return** ct

On input (send,inp_i, ct')
- assert keygen has been called
- $Y' := \mathsf{Dec}_{\mathsf{sk}}(ct')$
- parse $(\delta'[0], \Delta'(\cdot), b_{1-i}) := Y'$
- $c := b_i \oplus b_{1-i}$
- **if** $c = 1 - i$ **then**
  - $\Delta(\cdot) := \Delta'(\cdot)$
  - $\delta[1] := \delta'[0]$
- **end if**
- $ct := \mathsf{Enc}_{\mathsf{sk}}(\mathsf{inp}_i, c)$
- **return** ct

On input (receive,ct')
- assert keyex, keygen and send have been called, ct' not seen
- $(\mathsf{inp}_{1-i}, c') := \mathsf{Dec}_{\mathsf{sk}}(ct')$
- store $(\mathsf{outp}_0, \mathsf{outp}_1) = f(\mathsf{inp}_0, \mathsf{inp}_1)$
- assert $c' = c$
- $ct := \mathsf{Enc}_{\mathsf{sk}}(\delta[1])$
- **return** ct

On input (ack,ct')
- assert receive has been called, ct' not seen and $\mathsf{Dec}_{\mathsf{sk}}(ct') \neq \perp$
- $r := r + 1$
- **if** $c = i$ **then**
  - calculate $\delta[r] := \frac{\delta[r-1] \cdot r}{\Delta(r)}$
- **else**
  - **if** $\mathsf{Dec}_{\mathsf{sk}}(ct') =: \delta' < 1$ **then**
    - $\delta[r] = \delta'$
  - **else**
    - calculate $\delta[r] := \frac{\delta[r-1] \cdot r}{\Delta(r)}$
  - **end if**
- **end if**
- $ct = \mathsf{Enc}_{\mathsf{sk}}(\delta[r])$
- start decreasing $\delta^* := \delta[r]$ periodically
- **return** $(ct, \perp)$

On input (output,v)
- **if** $v \neq \perp$ **then return** v
- **end if**
- assert ack has been called
- assert $\delta^* < 1$
- **return** outp_i

Algorithm 2 Protocol $\mathsf{prot}_{\pi_{h\Delta}}[sid, f, p_0, p_1, i]$ $\pi_{h\Delta}$ running on party i, based on original protocol π_Δ by [17].

On input inp_i from $\mathcal{Z}$
- $eid := \mathcal{G}_{att}.\mathtt{install}(sid, \mathsf{prog}_{\pi_{h\Delta}}[f, p_0, p_1, i])$
- $(g^a, \sigma) := \mathcal{G}_{att}.\mathtt{resume}(eid, \mathtt{keyex})$
- send (eid, g^a, σ) to p_{1-i}, await (eid', g^b, σ') from p_{1-i}
- assert $\Sigma.\mathsf{Ver}_{\mathsf{mpk}}((sid, eid', \mathsf{prog}_{\pi_{h\Delta}}[f, p_0, p_1, 1-i], g^b), \sigma')$
- $(ct, \cdot) := \mathcal{G}_{att}.\mathtt{resume}(eid, (\mathtt{keygen}, g^b))$, send ct to p_{1-i}, await ct'
- $(ct, \cdot) := \mathcal{G}_{att}.\mathtt{resume}(eid, (\mathtt{send}, \mathsf{inp}_i))$, send ct to p_{1-i}, await ct'
- $(ct, \cdot) := \mathcal{G}_{att}.\mathtt{resume}(eid, (\mathtt{receive}, ct'))$, send ct to p_{1-i}, await ct'
- repeat: $(ct, \cdot) := \mathcal{G}_{att}.\mathtt{resume}(eid, (\mathtt{ack}, ct'))$, send ct to p_{1-i}, await ct'

On input output from $\mathcal{Z}$
- **return** $\mathcal{G}_{att}.\mathtt{resume}(eid, (\mathtt{output}, \perp))$

Being only a single symmetric decryption and encryption, the calculation of acknowledgements is very fast (on our test system, AES-128-GCM in SGX on an Intel Xeon D-1718T, this operation takes 0.3ms), so the limiting factor for releasing the output is network delay. The remote attestation is done with the EPID mechanism provided by the SGX SDK to establish an ecdh-256-bit key between the enclaves. For the chosen security parameter, this means that in this setting releasing the output over a good internet connection with 10ms latency takes around 10 seconds. The computation of the desired functionality beforehand happens at near-native speed inside the enclave.

5 Security

In this section we prove the security of our protocol, *i.e.* we prove in Theorem 1 that $\pi_{h\Delta}$ securely GUC-realizes $\mathcal{F}^{f,h\Delta}$ in the $\mathcal{G}_{att}$-hybrid model.

For the purpose of clarity, we abbreviate
$\mathcal{G}_{att}.\mathtt{install}(sid, (\mathsf{prog}_\pi[sid, f, p_0, p_1]))$ with $\mathcal{G}_{att}.\mathtt{install}(\mathsf{prog}_\pi)$.

Theorem 1. *Assume the DDH assumption holds,* AE *is an IND-CPA and INT-CTXT secure and perfectly correct encryption scheme, and* Σ *is an EUF-CMA secure and perfectly correct signature scheme. Then* $\pi_{h\Delta}$ *GUC-realizes* $\mathcal{F}^{f,h\Delta}$ *in the* $\mathcal{G}_{att}$*-hybrid model.*

By design, both the ideal functionality and the protocol draw the parameters for the delay function independently and at random. Both sets of parameters are unknown to the adversary. To ensure that the execution of the ideal protocol is indistinguishable from the simulation of the real protocol, the simulator has to ensure the parties receive the same output at the same time in both worlds. The same output is guaranteed by extracting the corrupt party's input from its call to $\mathcal{G}_{att}$. The functionality, given the input from the honest party and the extracted output from the corrupt party, can calculate the correct output using the function f. The same output delay is achieved by "ignoring" the protocol's delay parameters but instead wait for the functionality to release the outputs. Once the adversary has been instructed to abort the protocol, *i.e.* when it no longer receives instructions from the environment and thus stops invoking the protocol and sending messages, the simulator sends ($\mathtt{abort}$, sid) to $\mathcal{F}^{f,h\Delta}$. In this case, $\mathcal{F}^{f,h\Delta}$ stores the current round r^* and delays $D := \Delta(\delta[r^*])$ and $d := \delta[r^*]$. D is the delay of party p_i for $i \neq c$. Whenever the adversary requests the output, the simulator calls ($\mathtt{output}$, sid) on $\mathcal{F}^{f,h\Delta}$. When the output is available in $\mathcal{F}^{f,h\Delta}$, it returns the output of the corrupt party to $\mathcal{S}$.

Proof. We show that for the dummy adversary $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that for all PPT environments $\mathcal{Z}$, it holds that $\pi_{h\Delta} \geq \mathcal{F}^{f,h\Delta}$. On the ideal side, $\mathcal{F}^{f,h\Delta}$ interacts with parties p_0 and p_1 in a session denoted with sid. In the simulation, $\pi_{h\Delta}$ interacts with parties $\tilde{p_0}$ and $\tilde{p_1}$. Whenever $\mathcal{Z}$ instructs $\mathcal{A}$ to corrupt $\tilde{p_i}$, $\mathcal{S}$ corrupts p_i. We first consider the case where $\tilde{p_0}$ resp. p_0 are corrupt. The case where $\tilde{p_1}$ resp. p_1 are corrupt is symmetrical. Afterwards we consider the case where both parties are honest. $\mathcal{S}$ passes through information between $\mathcal{Z}$ and $\mathcal{A}$, and $\mathcal{A}$ and $\mathcal{G}_{att}$.

The simulator works as follows:

Definition 6 (Simulator $\mathcal{S}$, p_0 corrupt).

- *When $\mathcal{Z}$ instructs $\mathcal{S}$ to install $\pi_{h\Delta}$ in the enclave with* `eid` *e_0 of $\tilde{p_0}$, pass through the instruction $\mathcal{G}_{att}.\mathtt{install}(\mathsf{prog}_{\pi_{h\Delta}})$ to install $\pi_{h\Delta}$ on $\tilde{p_0}$. Create an enclave with* `eid` *e_1 for $\tilde{p_1}$ and install $\pi_{h\Delta}$ by calling $\mathcal{G}_{att}.\mathtt{install}(\mathsf{prog}_{\pi_{h\Delta}})$ and set $\mathsf{inp}_1 = \vec{0}$.*
- *When $\mathcal{Z}$ instructs $\mathcal{A}$ to call $\mathcal{G}_{att}.\mathtt{resume}(e_0, (\mathtt{send}, \mathsf{inp}_0))$, extract inp_0 from this call and send inp_0 to $\mathcal{F}^{f,h\Delta}$.*
- *Whenever $\mathcal{S}$ receives call to $\mathcal{G}_{att}$ from $\mathcal{Z}$ for $\tilde{p_0}$, pass through the call.*
- *Whenever $\mathcal{S}$ receives message from $\tilde{p_0}$, call $\mathcal{G}_{att}$ to create response.*
- *Whenever $\mathcal{S}$ receives a pair (m, σ) from $\tilde{p_0}$ with a valid signature σ for a message m, where m has not been previously output from $\mathcal{G}_{att}$, output $\bot_\Sigma$ and halt simulation.*
- *Whenever $\mathcal{S}$ receives a valid ciphertext from $\tilde{p_0}$, which has not been previously output from $\mathcal{G}_{att}$, output $\bot_{\mathsf{Enc}}$ and halt simulation.*
- *When $\mathcal{S}$ receives no message $(ct, \cdot)$ from $\tilde{p_0}$ that would result in $\tilde{p_1}$ calling $\mathcal{G}_{att}.\mathtt{resume}(e_1, (\mathtt{ack}, ct))$ for the first time, send (*`abort`*,sid) to $\mathcal{F}^{f,h\Delta}$.*
- *Whenever $\mathcal{S}$ receives no message from $\tilde{p_0}$ in a later round, send (*`abort`*,sid) to $\mathcal{F}^{f,h\Delta}$.*
- *When $\mathcal{Z}$ instructs $\mathcal{S}$ to call $\mathcal{G}_{att}.\mathtt{resume}(e_0, (\mathtt{output}, v))$, send (*`output`*,sid) to $\mathcal{F}^{f,h\Delta}$.*
- *When $\mathcal{F}^{f,h\Delta}$ returns $\bot$, return $\bot$ to $\mathcal{A}$.*
- *When $\mathcal{F}^{f,h\Delta}$ returns outp_0, pass through the call $\mathcal{G}_{att}.\mathtt{resume}(e_0, (\mathtt{output}, \mathsf{outp}_0))$.*

To see that the environment's output in the simulated execution is indistinguishable from its output in the real execution, we consider the following hybrid games:

Hybrid H0 Identical to the execution of the real protocol $\pi_{h\Delta}$ and the adversary $\mathcal{A}$.

Hybrid H1 Execution of an ideal functionality $\mathcal{F}_1$ that reports all inputs to the adversary and lets the adversary perform arbitrary outputs. The simulator $\mathcal{S}_1$ executes the protocol on behalf of the honest parties, making outputs through $\mathcal{F}_1$ and behaves like the dummy adversary for corrupted parties and handles calls to $\mathcal{G}_{att}$ of corrupted parties like the dummy adversary.

Hybrid H2 $\mathcal{F}_2$ is identical to $\mathcal{F}_1$. $\mathcal{S}_2$ is identical to $\mathcal{S}_1$ except that it will abort if it receives a pair (σ, m) where σ is a correct signature for m, but m has not previously been output by $\mathcal{G}_{att}$.

Hybrid H3 In H3 the UC experiment allows $\mathcal{S}_3$ to use $\mathcal{G}'_{att}$ instead of $\mathcal{G}_{att}$ which allows $\mathcal{S}_3$ to use a random key sk' instead of the real key sk used in $\mathcal{G}_{att}$. Otherwise $\mathcal{S}_3$ is identical to $\mathcal{S}_2$. $\mathcal{F}_3$ is identical to $\mathcal{F}_2$.

Hybrid H4 $\mathcal{F}_4$ is identical to $\mathcal{F}_3$. $\mathcal{S}_4$ is identical to $\mathcal{S}_3$ except that will abort, if it receives ciphertext that has not previously been output by $\mathcal{G}'_{att}$.

Hybrid H5 $\mathcal{F}_5$ is identical to $\mathcal{F}_4$ except that $\mathcal{F}_5$ can receive inputs to calculate the function and forward the corrupt party's output to $\mathcal{S}_5$ when called. $\mathcal{S}_5$ is identical to $\mathcal{S}_4$ except that it uses input $\vec{0}$ for the honest party. Additionally, $\mathcal{S}_5$ extracts the corrupt party's input inp_0 from its call to $\mathcal{G}_{att}$. When receiving calls $\mathcal{G}'_{att}.\mathtt{resume}(e_0, (\mathtt{output}, v))$ for $\tilde{p_0}$, it calls $\mathcal{F}_5$ to receive the output outp_0 and lets $\tilde{p_0}$ call $\mathcal{G}'_{att}.\mathtt{resume}(e_0, (\mathtt{output}, \mathsf{outp}_0))$.

Hybrid H6 $\mathcal{F}_6$ is identical to $\mathcal{F}_5$ except that now the ideal functionality draws the bit b, an initial delay $\delta[0]$ and a delta reduction function $\Delta(\cdot)$ as defined in $\mathcal{F}^{f,h\Delta}$ to return the output to the parties. $\mathcal{S}_6$ is identical to $\mathcal{S}_5$ except that it lets parties call $\mathcal{G}_{att}.\mathtt{resume}(e_i, (\mathtt{output}, \mathsf{outp}_i))$ when the output takes place in $\mathcal{F}_6$.

Hybrid H7 In H7 the UC experiment requires the use of $\mathcal{G}_{att}$ again. $\mathcal{F}_7$ is identical to $\mathcal{F}_6$. $\mathcal{S}_6$ uses the real $\mathcal{G}_{att}$ again, otherwise it is identical to $\mathcal{S}_6$.

Hybrid H8 Execution of the ideal protocol of $\mathcal{F}^{f,h\Delta}$ and the simulator $\mathcal{S}$.

Claim 1. out_0 and out_1 are identically distributed.

Proof. Since H1 only makes syntactical changes, the claim follows straight forward from the definition of H1.

Claim 2. Assuming Σ is an EUF-CMA secure signature scheme, H2 aborts with negligible probability and out_1 is computationally indistinguishable from out_2.

Proof. In H2, $\mathcal{S}_2$ aborts when it receives a pair (σ, m) where σ is a correct signature m but m has not been previously output by $\mathcal{G}_{att}$. We use the execution of H2 to build a reduction to the EUF-CMA security of Σ. Let E_Σ be the event where $\mathcal{S}_2$ outputs $\perp_\Sigma$. Assume for the sake of contradiction that E_Σ takes place with non-negligible probability, that is, $Pr[E_\Sigma]$ is non-negligible. Then there exists an adversary $\mathcal{A}'$ against the EUF-CMA security of Σ that wins with non-negligible probability. Let $\mathcal{C}$ be the challenger playing the EUF-CMA game with $\mathcal{A}'$. $\mathcal{C}$ generates a pair of keys $(sk, pk) \leftarrow \Sigma.\mathsf{Gen}$ and sends pk to $\mathcal{A}'$. $\mathcal{A}'$ works as follows:

- $\mathcal{A}'$ internally emulates H2.
- $\mathcal{A}'$ uses H2 to generate a signature σ via the signature oracle for message m^* that triggers E_Σ.
- $\mathcal{A}'$ forwards (m^*, σ) to $\mathcal{C}$.

The challenger outputs $\Sigma.\mathsf{Vfy}(m^*, \sigma)$. Here, the advantage of $\mathcal{A}'$ in the EUF-CMA game is identical to $Pr[E_\Sigma]$ which is a contradiction to the EUF-CMA security of Σ. It follows that the probability that $Pr[E_\Sigma]$ is bounded by the advantage of $\mathcal{A}'$ in the EUF-CMA game which is a negligible function $\eta_{\mathsf{EUF\text{-}CMA}}$, $\mathcal{A}$ that $Pr[E_\Sigma] \leq \eta_{\mathsf{EUF\text{-}CMA}}$. With this, $|Pr[out_2 = 0] - Pr[out_1 = 0]|$ is negligible, which concludes the claim.

Claim 3. Assuming the DDH assumption holds, out_2 and out_3 are computationally indistinguishable.

Proof. In $\mathsf{H3}$, the simulator uses $\mathcal{G}'_{att}$ with random key sk'. Suppose that $|Pr[out_2 = 0] - Pr[out_3 = 0]|$ is non-negligible. Then there exists an adversary $\mathcal{A}'$ that wins in the DDH game with non-negligible probability. Let $\mathcal{C}$ be the challenger in the DDH game. Let $\mathcal{G}$ be a group with generator g and order q. $\mathcal{C}$ randomly draws a bit $b \xleftarrow{\$} \{0,1\}$. $\mathcal{C}$ draws two elements $a, b \xleftarrow{\$} \mathbb{Z}_q$ and an element $z \xleftarrow{\$} \mathcal{G}$. It calculates g^a, g^b, g^{ab} and sends $g^a, g^b, c = b \cdot g^{ab} + (1-b) \cdot z$ to $\mathcal{A}'$. Then, $\mathcal{A}'$ proceeds as follows.

- Internally emulate $\mathsf{H2}$ but use the keys received by $\mathcal{C}$ instead of the real key sk.
- Receive bit b' from $\mathcal{Z}$ and send b' to $\mathcal{C}$.

If $\mathcal{C}$'s bit is 0 then $\mathcal{Z}$'s view is identically distributed to $\mathsf{H3}$. If $\mathcal{C}$'s bit is 1 then $\mathcal{Z}$'s view is identically distributed to $\mathsf{H2}$. Therefore the advantage of $\mathcal{A}$ in the DDH game is identical to $\mathcal{Z}$'s advantage in distinguishing $\mathsf{H2}$ and $\mathsf{H3}$, namely $|Pr[out_2 = 0] - Pr[out_3 = 0]|$. By assumption, $|Pr[out_2 = 0] - Pr[out_3 = 0]|$ is non-negligible, which leads to a contradiction. Hence, the claim follows.

Claim 4. Assuming AE is INT-CTXT secure, out_3 and out_4 are computationally indistinguishable.

Proof. In $\mathsf{H4}$ $\mathcal{S}$ aborts when receiving a ciphertext that has not been previously output by $\mathcal{G}_{att}$. Similar to E_Σ, assume that $Pr[E_{\mathsf{Enc}}]$ is non-negligible. Then there exist an adversary $\mathcal{A}'$ against the INT-CTXT security of AE that wins with non-negligible probability. Let $\mathcal{C}$ be the challenger playing the INT-CTXT game with $\mathcal{A}'$. $\mathcal{C}$ generates a key $sk \leftarrow \mathsf{AE.Gen}$. $\mathcal{A}'$ works as follows:

- $\mathcal{A}'$ internally emulates $\mathsf{H3}$.
- $\mathcal{A}'$ uses $\mathsf{H3}$ to generate ciphertext c^* via the encryption oracle which it sends to $\mathcal{C}$.

$\mathcal{C}$ decrypts c^* to obtain m^* and returns 1 if $m^* \neq \perp$ and c^* not seen before. Here, the advantage of $\mathcal{A}'$ in the INT-CTXT game is identical to $Pr[E_{\mathsf{Enc}}]$ which is a contradiction to INT-CTXT security of AE. It follows that the probability that $Pr[E_{\mathsf{Enc}}]$ is bounded by the advantage of $\mathcal{A}'$ in the INT-CTXT game which is a negligible function $\eta_{\mathsf{INT\text{-}CTXT}}$, *i.e.* that $Pr[E_{\mathsf{Enc}}] \leq \eta_{\mathsf{INT\text{-}CTXT}}$, which concludes the claim.

Claim 5. Assuming AE is IND-CPA secure, out_4 and out_5 are computationally indistinguishable.

Proof. In $\mathsf{H5}$, $\mathcal{S}_5$ extracts the input of the corrupt party and sets the input of the honest party to $\vec{0}$. We use $\mathsf{H5}$ to build a reduction to the IND-CPA security of AE. Assume for the sake of contradiction that out_4 and out_5 are not indistinguishable, that is, $|Pr[out_4 = 0] - Pr[out_5 = 0]|$ is non-negligible. Then there exists an adversary $\mathcal{A}'$ against the IND-CPA security of AE that wins with non-negligible probability.

Let $\mathcal{C}$ be an adversary playing the IND-CPA game with $\mathcal{A}'$. $\mathcal{C}$ draws a random bit $b \xleftarrow{\$} \{0,1\}$. $\mathcal{A}'$ works as follows.

- $\mathcal{A}'$ internally emulates $\mathsf{H4}$.
- $\mathcal{A}'$ receives the input denoted as inp of the honest party from $\mathcal{F}_4$.
- $\mathcal{A}'$ sends $m_0 = \mathsf{inp}, m_1 = \vec{0}$ to $\mathcal{C}$ which uses the encryption oracle to encrypts m_i according to b and sends back the resulting ciphertext c.
- In the message from $\mathcal{G}'_{att}$ that would include the encryption of inp $\mathcal{A}'$ uses c instead.
- At the end of the execution of $\mathsf{H4}$ receive bit b' from $\mathcal{Z}$ and send b' to $\mathcal{C}$.

If $\mathcal{C}$'s bit is 0 then $\mathcal{Z}$'s view is identically distributed to $\mathsf{H4}$. If $\mathcal{C}$'s bit is 1 then $\mathcal{Z}$'s view is identically distributed to $\mathsf{H5}$. As the probability $|Pr[out_4 = 0] - Pr[out_5 = 0]|$ that $\mathcal{Z}$ can distinguish between $\mathsf{H4}$ and $\mathsf{H5}$ is non-negligible, we have a contradiction to the IND-CPA security of AE. This concludes the claim.

Claim 6. Assuming the IND-CPA security of AE holds, out_5 and out_6 are computationally indistinguishable.

Proof. In $\mathsf{H6}$ $\mathcal{F}_6$ draws the parameters at random. The time the output is available now is defined by the parameters drawn by $\mathcal{F}_6$. The proof that the exchanged messages containing the parameters are IND-CPA secure is symmetrical to Claim 5 via the standard hybrid argument with the difference that the parameters drawn in $\mathsf{H5}$ and the dummy parameters in $\mathsf{H6}$ are encrypted. Given the IND-CPA security of exchanged messages, we are left to show that the parties receive the output at the same time in $\mathsf{H5}$ and $\mathsf{H6}$. By definition the parameters (subscript f denotes the functionality's parameters, subscript p the protocol's parameters) b_f and $\delta_f[0]$ drawn by $\mathcal{F}_6$ and parameters c_p and $\delta_p[0]$ in $\mathsf{H5}$ are identically distributed. b_f is identically distributed as c_p, as b_f is drawn uniformly random and c_p is the XOR of two uniformly random drawn bits. $\delta_f[0]$ $\delta_p[0]$ are sampled from the same set, *i.e.* $\delta_f = \delta_p = 2^x$ where $x \xleftarrow{\$} [\lambda/2, \lambda]$. In $\mathsf{H5}$, $\Delta_p[\cdot]$ is drawn from f_a where $a \in [2, \lambda]$. In $\mathsf{H6}$, $\Delta_f[\cdot]$ is drawn from f_a where $a \in [2, \lambda^2]$. When $b_f = 1$, and p_0 aborted previously in round r, $\mathcal{F}_6$ uses the delay for the honest party $D_f = \Delta_f(\delta_f[r])/r$, whereas in $\mathsf{H5}$ p_1's delay is $D_p = \Delta_p(\Delta_p(\delta[r]))$ when p_0's delay is $\delta_f[r]$ resp. $\delta_p[r]$. As Δ_p is drawn from the family of functions f_a where $a \in [2, \lambda]$ and Δ_f is drawn from a different family of functions $f_{\hat{a}}$ where $\hat{a} \in [2, \lambda^2]$, the delays are distributed identically. When $b = 0$, the delay for both parties is identical, *i.e.* $\delta_f[r]$ which is identically distributed as $\delta_p[r]$. In $\mathsf{H6}$ this is achieved by decrypting the other party's delay δ' and checking if $\delta' < 1$.

Let q be the maximum amount of ciphertexts sent in the `keygen`, `send`, `receive` and `ack` steps of the protocol. Then, we define additional hybrids Gi for $0 \leq i \leq q$. In Gi, the first i ciphertexts are identical to the ciphertexts sent, when using the parameters matching the protocol. All later ciphertexts are identical to the ciphertexts sent when using the parameters from the ideal functionality. It holds that $\mathsf{G0} = \mathsf{H6}$ and $\mathsf{Gq} = \mathsf{H5}$. Assume, for the sake of contradiction, that an adversary can distinguish between $\mathsf{G0}$ and Gq with non-negligible probability, then we can construct an adversary that can break the IND-CPA security of AE by randomly choosing a $1 \leq i \leq q$, constructing the first $i - 1$ ciphertexts using the parameters from the protocol, the last $q - i$ ciphertexts using the parameters from the ideal functionality and embedding the IND-CPA challenge in the ith position. The challenge messages are the messages encrypted in step i if the parameters of the protocol or the ideal functionality are used. Depending on the challenge-bit from the IND-CPA challenger, this adversary simulates $\mathsf{Gi}-1$ or Gi. Therefore, this adversary can then break the IND-CPA security of AE with a tightness-loss of $1/q$ and therefore with non-negligible probability.

Claim 7. Assuming the DDH assumption holds, out_6 and out_7 are computationally indistinguishable.

Proof. As in $\mathsf{H7}$ the simulator uses the real $\mathcal{G}_{att}$ with real key sk again, the proof is symmetrical to Claim 3 and thus omitted.

Claim 8. out_7 and out_8 are identically distributed.

Proof. This follows directly from the fact that $\mathsf{H7}$ and $\mathsf{H8}$ use the same functionality and simulator.

Definition 7 (Simulator $\mathcal{S}$, both parties honest). *When both parties are honest, the simulator works as follows:*

- *For $i \in \{0,1\}$, create enclaves e_i on $\tilde{p}_i$ and install $\pi_{h\Delta}$ by calling $\mathcal{G}_{att}.\mathtt{install}(\mathsf{prog}_{\pi_{h\Delta}})$ and set $\mathsf{inp}_i = \vec{0}$.*
- *Execute protocol for both parties and exchange messages. When receiving a message from $\tilde{p}_i$ that results in $\tilde{p}_{1-i}$ calling $\mathcal{G}_{att}.\mathtt{resume}(e_{1-i}, (\mathtt{ack}, ct'))$, pass through the call.*
- *Every time $\mathcal{S}$ is activated, call ($\mathtt{status}$,sid) on $\mathcal{F}^{f,h\Delta}$. If returned i, let $\tilde{p_i}, i \in \{0,1\}$ make call $\mathcal{G}_{att}.\mathtt{resume}(e_i, (\mathtt{output}, v))$ for arbitrary v and stop protocol for p_i. If returned $\mathtt{finished}$ and party j has not received output yet, let $\tilde{p_j}, j \in \{0,1\}$ make call $\mathcal{G}_{att}.\mathtt{resume}(e_j, (\mathtt{output}, v))$ for arbitrary v. If returned $\perp$, continue exchanging $\mathtt{ack}$ messages.*

Hybrid $\mathsf{H0}$ Identical to the real protocol $\pi_{h\Delta}$ and the adversary $\mathcal{A}$.

Hybrid $\mathsf{H1}$ Execution of an ideal functionality $\mathcal{F}_1$ that reports all inputs to the adversary and lets the adversary perform arbitrary outputs. The simulator $\mathcal{S}_1$ executes the protocol on behalf of the honest parties, making outputs through $\mathcal{F}_1$ and behaves like the dummy adversary for corrupted parties and handles calls to $\mathcal{G}_{att}$ of corrupted parties like the dummy adversary.

Hybrid $\mathsf{H2}$ In $\mathsf{H2}$ the UC experiment allows the $\mathcal{S}_3$ to use $\mathcal{G}'_{att}$ instead of $\mathcal{G}_{att}$ which uses a random key sk' instead of the real key sk used in $\mathcal{G}_{att}$. Otherwise $\mathcal{S}_2$ is identical to $\mathcal{S}_1$. $\mathcal{F}_2$ is identical to $\mathcal{F}_1$.

Hybrid $\mathsf{H3}$ $\mathcal{F}_3$ is identical to $\mathcal{F}_2$ except that $\mathcal{F}_3$ can receive inputs to calculate the function and forward the corrupt party's output to $\S_3$ when called. $\mathcal{S}_3$ is identical to $\mathcal{S}_2$ except that it uses input $\vec{0}$ for both parties. When receiving calls $\mathcal{G}'_{att}.\mathtt{resume}(e_i, (\mathtt{output}, v))$ for $\tilde{p_1}$, it instructs $\mathcal{F}_3$ to return the output to p_i and lets $\tilde{p}_i$ call $\mathcal{G}'_{att}.\mathtt{resume}(e_i, (\mathtt{output}, v))$. It uses the output to the adversary from $\mathcal{F}_3$ to retrieve the output for the parties in the protocol.

Hybrid H4 $\mathcal{F}_4$ is identical to $\mathcal{F}_3$ except that it randomly draws a bit b, an initial delay $\delta[0]$ and a delta reduction function $\Delta(\cdot)$ to determine the time the output is returned to parties. $\mathcal{S}_4$ is identical to $\mathcal{S}_3$ except that it calls (`status`) on $\mathcal{F}^{f,h\Delta}$ on every activation and lets party p_i call $\mathcal{G}_{att}.\texttt{resume}(e_i, (\texttt{output}, v))$ when returned i (or `finished`) by $\mathcal{F}_4$.

Hybrid H5 In H5 the UC experiment only allows using $\mathcal{G}_{att}$ again. $\mathcal{F}_5$ is identical to $\mathcal{F}_4$. $\mathcal{S}_5$ is identical to $\mathcal{S}_4$ that is uses real $\mathcal{G}_{att}$ again.

Hybrid H6 Execution of the ideal protocol of $\mathcal{F}^{f,h\Delta}$ and the simulator $\mathcal{S}$.

Claim 9. out_0 and out_1 are identically distributed.

Proof. As changes in H1 are only syntactical, the claim follows straight forward from the definition of H1.

Claim 10. Assuming the DDH assumption holds, out_1 and out_2 are computationally indistinguishable.

Proof. In H2, $\mathcal{S}_2$ uses $\mathcal{G}'_{att}$ with random key sk. Due to symmetry to Claim 3, the claim follows.

Claim 11. Assuming AE is IND-CPA secure, out_2 and out_3 are computationally indistinguishable.

Proof. In H3, $\mathcal{S}_3$ sets the input values of both parties to $\vec{0}$. Due to symmetry to Claim 5, the claim follows.

Claim 12. Assuming AE is IND-CPA secure and $\delta[0]$, $\Delta(\cdot)$ and b are drawn at random, out_3 and out_4 are computationally indistinguishable.

Proof. In H4 $\mathcal{F}_4$ uses randomly drawn parameters to determine the output time. Due to symmetry to Claim 6, the claim follows.

Claim 13. Assuming the DDH assumption holds, out_4 and out_5 are computationally indistinguishable.

Proof. In H5 $\mathcal{S}_5$ uses real $\mathcal{G}_{att}$ again. The claim follows directly from Claim 3.

Claim 14. out_5 and out_6 are identically distributed.

Proof. As $\mathcal{F}_5$ is identical to $\mathcal{F}^{f,h\Delta}$, the claim follows straight forward from the definition of H5.

6 Conclusion

In this work we proposed the new fairness notion *hidden Δ-fairness* for two-party computation based on Pass et al.'s work on the formal abstraction for trusted computing via the globally shared functionality $\mathcal{G}_{att}$ and the accompanied Δ-fair ideal functionality for 2PC. We presented the new fairness notion via an ideal functionality $\mathcal{F}^{f,h\Delta}$ and defined an efficient 2PC protocol $\pi_{h\Delta}$ based on Pass et al.'s 2PC protocol π_{Δ} and showed that $\pi_{h\Delta} \leq \mathcal{F}^{f,h\Delta}$ in the GUC framework. Our functionality and protocol hides the current delay, allows for a variable output order and prevents the adversary from receiving the output immediately, which opens up new use cases for two-party computation. Future work includes investigating fairness in 2PC with transparent enclaves or other enclave models.

References

1. Bailey, B., et al.: General partially fair multi-party computation with vdfs. IACR ePrint (2022)
2. Barbosa, M., et al.: Foundations of hardware-based attested computation and application to sgx. In: EuroS&P (2016)
3. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: CRYPTO (2014)
4. Blum, M.: How to exchange (secret) keys. TOCS (1983)
5. Brickell, E., et al.: Direct anonymous attestation. In: CCS (2004)
6. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS (2001)
7. Canetti, R., et al.: Universally composable security with global setup. In: TCC (2007)
8. Choi, J.I., et al.: Secure multiparty computation and trusted hardware: Examining adoption challenges and opportunities. Security and Communication Networks (2019)
9. Choudhuri, A.R., et al.: Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In: CCS (2017)
10. Cleve, R.: Limits on the security of coin flips when half the processors are faulty. In: STOC (1986)
11. Cohen, R., et al.: From fairness to full security in multiparty computation. Journal of Cryptology (2022)
12. Costan, V., Devadas, S.: Intel sgx explained. IACR ePrint (2016)
13. Gordon, S.D., Katz, J.: Partial fairness in secure two-party computation. In: EUROCRYPT (2010)
14. Gordon, S.D., et al.: Complete fairness in secure two-party computation. IACR ePrint (2008)
15. Liang, H., Li, M.: Bring the missing jigsaw back: Trustedclock for sgx enclaves. In: Proceedings of the 11th European Workshop on Systems Security (2018)
16. Lindell, Y.: How to simulate it–a tutorial on the simulation proof technique. Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich (2017)
17. Pass, R., et al.: Formal abstractions for attested execution secure processors. In: EUROCRYPT (2017)
18. Paul, S., Shrivastava, A.: Efficient fair multiparty protocols using blockchain and trusted hardware. In: LATINCRYPT (2019)
19. Yao, A.C.C.: How to generate and exchange secrets. In: FOCS (1986)