IACR Policy for Cryptology Schools

August 2015*

1 Related Works

Nodes on [4]:

- This model starts with describing how to model execution of *synchronous* protocols that can access a global setup clock.
- In a previous treatment, the clock in UC was local to each party and it would have to receive update messages from the other parties (everyone is doing this operation). Hence, with GUC the environment can control the clock speed and define when clock updates happen (as other protocol sessions might also be accessing it).

There are several works from the past few years that try to model a blockchain within the Universal Composability framework—some attempting to model it in its extendion, (G)UC [**?**, **?**].

Kiayias et al. [4] models a Bitcoin-like blockchain for fair and robust multi-party computation. It is motivated by the impossibility result for fairness in secure MPC [1] and circumeventing it by imposing monetary penalties on participants. The model consists of two global functionalities, $\overline{\mathcal{G}}_{\mathsf{clock}}$ and $\overline{\mathcal{G}}_{\mathsf{blockchain}}$. The blockchain functionality enables the expected functionality like submitting tranasctions, validating them, batching them into blocks, and allowing an adversary to reorder transcations. Because of the GUC framework, the state of the blockchain is available to all parties including the environment and any other protocol sessions (or dummy parties). This work however, fails to prove that their model of the blockchain is GUC-realized in any currently existing blockchain system. Such a security proof is essential as it provide credibility to the possibility of implementing protocols in the $\overline{\mathcal{G}}_{\mathsf{blockchain}}$-hybrid world. Furthermore, the assumptions that are made for the blockchain and what the adversary can do severly limit the scope of adversaries in the rearl-world. The first failure of this model is to consider an adversary which can change the view some parties have of the blockchain state. For example, if the adversary mines a new block and keeps it a secret, or if some nodes have not received new blocks because of communication delays. Another failure is that all transactions in the buffer between blocks are always included in the next block. This, again, prevents a miner-like adversary which can censor transactions and delay their entry into the chain. Finally, the state of the blockchain

*The most recent version of this document can be obtained from `http://www.iacr.org/docs/`.
Editors of this document: M. Abdalla, A. Boldyreva, C. Cachin, A. Kiayias, B. Warinschi (2014).

[1] Fairness in MPC is defined as: either all parties learn the output or none of them do.

is updated at fixed time intervals which does not accurately convey the consensus model of Bitcoin or Ethereum.

Badertscher et al. [1] attempt to solve these problems by allowing a more unrestricted in the GUC framework. The shared functionality in this case is a global clock functionality, $\overline{\mathcal{G}}_{\mathsf{clock}}$, which enables modelling a synchronous system in the UC framework by proceeding in rounds. Because it is a shared functionality, the clock allows any other protocol session in the environment to be synchronized with the challenge protocol. The blockchain functionality is a local functionality (only available to the parties within the protocol session) that allows the adversary to have more power in what it can do. The adversary can inject transactions and modify the state of the chain that all parties that query it can see. This is accomplished by allowing a maximum distance, d, that the adversary can specify and return a prefix of the chain which is at most a distance d from the head of the chain. Furthermore, the adversary can choose exactly which transactions are allowed to be in the next block. The blockchain functionality is modularized by allowing the definition of subroutines that capture extending the blockchain state (specifically for Bitcoin in this paper). The authors of this work admit that the paper's only intent is to model the Bitcoin blockchain hence the choice to use the ledger as only a local functionality. This prevents other protocol sessions from using the same blockchain (definitely a limitation of modelling the reality of a blockchain environment). Furthermore, this paper makes the argument that it is dangerous to have a global ledger functionality as such replacement does not "in general, preserve a realization proof of some ideal functionality $\mathcal{F}$ that is conducted in a ledger-hybrid world, because the simulator in that proof might rely on specific capabilities that are not available any more after the replacement (as the global setup is also replaced in the real world)". It claims that [2] provides a sufficient condition for such a replacement, but that the condition is too strong to be satisfied by any ledger implementation.

Canneti et al. [3] addresses the global PKI and an ideal authentication within the UC with global setup. The specific problem presented in this paper is that the ideal authentication functionality, $\mathcal{F}_{\mathsf{auth}}$,is usually formulated with the desirable property of non-transferrability of authentication. This means that when I send an authenticated message to another person, they are unable to use that proof to convince anyone else of the authentication. The paper realized that the real world PKI model is global *and* that, within it, signatures are globally verifiable. Once a key has signed a message for authentication, that proof is verifiable by and transferrable to anyone else in the system. Therefore, this work models a new relaxed global PKI, relaxes the UC authentication protocol to not require deniability, and formulates new functionalities for authentication and key exchange without deniability. Finally, they propose a new composition theorem allowing substitution of global functionalitites, $\mathcal{F}$ *EUC-realizes* $\mathcal{G}$. The problem being solved relates back to a claim made by Badertscher et al. [1] that replacement of global functionalities with real implementations generally invalidates a realization proof of some functionality that shares state with it. In this paper, this arises as replacement of the UC PKI system with a real one where transferrability is possible invalidates the realization proof of the ideal authentication functionality in the plain-PKI model.

They formulate a new authentication functionality that does not impose non-transferrability and a long lasting global functionality handling certificates. Finally they prove that the certificate functionality guarantees are precisely captured by EU-CMA signatures and a globally-available PKI . This paper however imposes some restrictions on what can be done. For example, there is a limitation that a particular ITI may only register a single key with the Cert and Bulletin Board functionalities. They claim however, that it is possible to realize $\mathcal{F}_{\mathsf{cert\_auth}}$, but a certificate-based approach is not it.

One of the main takeaways in this paper is that you can define a functionality and analyze it for it's properties then prove that it is equivalent to another functionality that realizes this protocol. In this paper that is done by defining

Differentiating $\mathcal{G}_{\mathsf{cert}}^{\mathsf{pid}}$ and $\mathcal{G}_{\mathsf{swk}}^{\mathsf{pid}}$. Questions to answer:

- What is the precise difference between $\mathcal{G}_{\mathsf{cert}}^{\mathsf{pid}}$ and $\mathcal{G}_{\mathsf{cwk}}^{\mathsf{pid}}$ and why is the substitution necessary?

2 Preliminaries

To build up to a ledger functionality, we first need to discuss the building blocks. The first important component is the communication model. Recall from [?] that the communication model is asynchronous where messages between parties can be arbitrarily delayed by the adversary. Though this is the weakest assumptions that one can make about a network, we require a synchronous communication model build on top of UC that can provide some eventual delivery guarantees.

2.1 Synchronous Network

There are two parts that go into modelling a synchronous network in UC: creating a round structure that all ITMs can be synchronized with and requiring maximum delays on message by the adversary. In order to achieve the former, we rely on a pervious work by Katz et al. [?].

$\mathsf{ExecTx}(\mathsf{to}, \mathsf{val}, \mathsf{data}, \mathsf{from})$

$\mathsf{nonces}[\mathsf{from}] \leftarrow \mathsf{nonces}[\mathsf{from}] + 1$
If $\mathsf{balances}[\mathsf{from}] < \mathsf{val}$: **reject**
$\mathsf{balances}[\mathsf{from}] \leftarrow \mathsf{balances}[\mathsf{from}] - \mathsf{val}$
$\mathsf{balances}[\mathsf{to}] \leftarrow \mathsf{balances}[\mathsf{to}] + \mathsf{val}$
$\mathsf{receipts}[\mathsf{from}, \mathsf{nonces}[\mathsf{from}]] \leftarrow \mathsf{CreateTxRef}(\mathsf{val}, \mathsf{from})$
If $\mathsf{to} \in \mathsf{contracts}$:
  $ret \leftarrow \mathsf{Exec}(\mathsf{to}, \mathsf{val}, \mathsf{data}, \mathsf{from})$
  $\mathsf{txs}[\mathsf{from}, \mathsf{nonces}[\mathsf{from}]] \leftarrow ret$

3 Three-Phase Commitment

3.1 Synchronous Bracha Broadcast

Theorem. *Protocol Π_{Bracha} securely realized $\mathcal{F}_{\mathsf{Bracha}}$ in the $\{\mathcal{F}_{\mathsf{BD-SEC}}, \mathcal{F}_{\mathsf{CLOCK}}\}$-hybrid world. Assume a stateic adversary corrupted up to $\frac{n}{3}$ parties.*

Consider the simulator, $\mathcal{S}$, above.

If the dealer $\mathcal{D}$ is honest: In the ideal world, $\mathcal{D}$ gives input v to $\mathcal{F}_{\mathsf{Bracha}}$ which gives leaks it to $\mathcal{S}$. The simulator submits the input to it all of the locl $\mathcal{F}_{\mathsf{BD-SEC}}(\mathcal{D}, p_i)$ for $p_i \in \mathcal{P}$.

ExecContractCreate(addr, val, data, from, private)

$\mathsf{nonces}[\mathsf{from}] \leftarrow \mathsf{nonces}[\mathsf{from}] + 1$
If $\mathsf{balances}[\mathsf{from}] < \mathsf{val}$: **reject**
$\mathsf{balances}[\mathsf{from}] \leftarrow \mathsf{balances}[\mathsf{from}] - \mathsf{val}$
$\mathsf{balances}[\mathsf{to}] \leftarrow \mathsf{balances}[\mathsf{to}] + \mathsf{val}$
$(\mathsf{functions}, \mathsf{args}) := \mathsf{data}$
$r \leftarrow \mathsf{functions.init}(args)$
$\mathsf{contracts}[\mathsf{addr}] = \mathsf{functions}$
$\mathsf{restricted}[\mathsf{addr}] = \mathsf{private}$
If $\neg r$:
  $\mathsf{balances}[\mathsf{from}] \leftarrow \mathsf{balances}[\mathsf{from}] + \mathsf{val}$
  $\mathsf{balanaces}[\mathsf{to}] \leftarrow \mathsf{balances}[\mathsf{to}] - \mathsf{val}$

$\mathcal{S}$ expects to receive $|\mathcal{P}|$ activations from $\mathcal{F}_{\mathsf{Bracha}}$ when ideal world parties attempt to read output from the functionality. In each activation, the simulator sufficiently ensures each party reads messages from all other parties and simualated state changes and increment the local $\overline{\mathcal{F}}_{\mathsf{clock}}$

3.2 Extra

References

[1] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *Annual International Cryptology Conference*, pages 324–356. Springer, 2017.

[2] Ran Canetti, Daniel Shahaf, and Margarita Vald. Universally composable authentication and key-exchange with global pki. In *IACR International Workshop on Public Key Cryptography*, pages 265–296. Springer, 2016.

[3] Ran Canetti, Daniel Shahaf, and Margarita Vald. Universally composable authentication and key-exchange with global pki. In *IACR International Workshop on Public Key Cryptography*, pages 265–296. Springer, 2016.

[4] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 705–734. Springer, 2016.

$\overline{\mathcal{G}}_{\mathsf{ledger}}$

Initialize $\mathsf{txqueue} := \{\}$, $\mathsf{contracts} := \{\}$, $\mathsf{newtxs} := \{\}$, $\mathsf{nonces} := \{\}$ $\mathsf{balances} := \{\}$, $\Delta := 8$, $rnd := 0$

On input (transfer, to, val, data, from) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  If $\mathsf{balances}[\mathsf{fro}] < \mathsf{val}$: **reject**
  $\mathsf{nonces}[\mathsf{from}] \leftarrow \mathsf{nonces}[\mathsf{from}] + 1$
  $\mathsf{newtxs}[\mathsf{from}, \mathsf{nonces}[\mathsf{from}] \leftarrow (transfer, to, val, data, from)$
  leak (transfer, to, val, data, from) to $\mathcal{A}$
On input (contract create, addr, val, data, private, from) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  If $\mathsf{balances}[\mathsf{from}] < \mathsf{val}$: **reject**
  $\mathsf{nonces}[\mathsf{from}] \leftarrow \mathsf{nonces}[\mathsf{from}] + 1$
  $caddr \leftarrow \mathsf{ComputeAddr}(from)$
  If $caddr \neq addr$: **reject**
  If $\mathsf{len}(\mathsf{data}) = 0$: **reject**
  $\mathsf{newtxs}[\mathsf{from}, \mathsf{nonces}[\mathsf{from}] \leftarrow (transfer, to, val, data, from)$
  leak (contract create, addr, val, data, private, from) to $\mathcal{A}$
On input (tick, addr) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  $rnd+ = 1$
  $\mathsf{balances}[\mathsf{addr}] += 1000000$
  For tx **in** $\mathsf{txqueue}[\mathsf{rnd}]$:
    If $tx[0] = \mathsf{transfer}$:
      $(\mathsf{transfer}, \mathsf{to}, \mathsf{val}, \mathsf{data}, \mathsf{from}) \leftarrow tx$
      $\mathsf{ExecTx}(\mathsf{to}, \mathsf{val}, \mathsf{data}, \mathsf{from})$
    If $tx[0] = \mathsf{contractcreate}$:
      $(\mathsf{contractcreate}, \mathsf{addr}, \mathsf{val}, \mathsf{data}, \mathsf{private}, \mathsf{from}) \leftarrow tx$
      $\mathsf{ExecContractCreate}(\mathsf{addr}, \mathsf{val}, \mathsf{data}, \mathsf{private}, \mathsf{from})$

On input (delayTx, from, nonce, rounds) from $\mathcal{A}$:
  $tx \leftarrow \mathsf{newtxs}[\mathsf{from}, \mathsf{nonce}]$
  Add tx to $\mathsf{txqueue}[\mathsf{rnd} + \mathsf{rounds}]$
  Remove tx from newtxs
On input (tick, addr, permutation) from $\mathcal{A}$:
  Apply *permutation* to $\mathsf{txqueue}[\mathsf{rnd}]$
  Run honest party mining with addr

Figure 1: Ideal functionality representing a basic ledger with adversarial methods for delaying/reordering transactions and smart contract support

Protection Wrapper $\mathcal{W}_p$

On input (transfer, to, val, data, from) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  $to \leftarrow$

Figure 2: Protection wrapper for the ledger to maintain indistinguishability.

U_{pay}

$U_{pay}(\mathsf{state}, (\mathsf{input_L}, \mathsf{input_R}), \mathsf{aux}_{in})$:
  If $\mathsf{state} = \perp$: $\mathsf{state} := (0, \emptyset, 0, \emptyset)$
  parse state as $(\mathsf{cred_L}, \mathsf{oldarr_L}, \mathsf{cred_R}, \mathsf{oldarr_R})$
  parse aux_{in} as $\{\mathsf{deposits}_i\}_{i \in \{L,R\}}$
  For $i \in \{L, R\}$:
    If $\mathsf{input}_i = \perp$: $\mathsf{input}_i := (\emptyset, 0)$
    parse input_i as $\mathsf{arr}_i, \mathsf{wd}_i$
    $\mathsf{pay}_i := 0, \mathsf{newarr}_i := \emptyset$
    While $\mathsf{arr}_i \neq \emptyset$:
      $e \leftarrow \mathsf{pop}(\mathsf{arr}_i)$
      If $e + \mathsf{pay}_i \leq \mathsf{deposits}_i + \mathsf{cred}_i$:
        $\mathsf{newarr}_{\neg i} \leftarrow e$
      $\mathsf{pay}_i += e$
    If $\mathsf{wd}_i > \mathsf{deposits}_i + \mathsf{cred}_i - \mathsf{pay}_i : \mathsf{wd}_i := 0$
  $\mathsf{cred_L} += \mathsf{pay_R} - \mathsf{pay_L} - \mathsf{wd_L}$
  $\mathsf{cred_R} += \mathsf{pay_L} - \mathsf{pay_R} - \mathsf{wd_R}$
  If $\mathsf{wd_L} \neq 0$ or $\mathsf{wd_R} \neq 0$:
    $\mathsf{aux}_{out} := (\mathsf{wd_L}, \mathsf{wd_R})$
  Else : $\mathsf{aux}_{out} := \perp$
  $\mathsf{state} := (\mathsf{cred_L}, \mathsf{newarr_L}, \mathsf{cred_R}, \mathsf{newarr_R})$
  Return $(\mathsf{aux}_{out}, \mathsf{state})$

Figure 3: Update function for a payment channel. Given as a parameter to $\mathcal{F}_{\mathsf{state}}$. It defines the format of the state and its updates.

Π_{pay}: $\mathsf{Contract_{pay}}$

Init $(\mathsf{P_L}, \mathsf{P_R})$:
  $\mathsf{deposits}_L, \mathsf{deposits}_R := 0$
On input (deposit) (tx) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  $\mathsf{deposits}_i += \mathsf{tx.value}$
  $\mathsf{out}(\mathsf{deposits}_L, \mathsf{deposits}_R)$
On input (output, $\mathsf{aux_{out}}$, tx) :
  parse $\mathsf{aux_{out}}$ as $(\mathsf{wd_L}, \mathsf{wd_R})$
  For $i \in \{L, R\}$: $\mathsf{send}(P_i, \mathsf{wd_i})$

Figure 4: Contract pay

Π_{pay}

Initialize $\mathsf{arr_i} = \emptyset, \mathsf{pay_i} = 0, \mathsf{wd_i} = 0, \mathsf{paid_i} = 0$
$\mathsf{Contract_{pay}}$ identifier $\mathcal{C}$
Send $(\emptyset, 0) \rightarrow \mathcal{F}_{state}$
On input (ping) from $\mathcal{Z}$:
  Send $(\mathsf{read}) \rightarrow \mathcal{F}_{\mathsf{state}}$
On input $(\mathsf{cred_L}, \mathsf{new_L}, \mathsf{cred_R}, \mathsf{new_R})$ from $\mathcal{F}_{\mathsf{state}}$:
  For $e \in \mathsf{new_i}$:
    Output $(\mathsf{receive}, e)$
    $\mathsf{paid_i} += e$
  Send $(\mathsf{arr_i}, \mathsf{wd} - \mathsf{wdn}) \rightarrow \mathcal{F}_{\mathsf{state}}$
  $\mathsf{arr_i} \leftarrow \emptyset$
  $\mathsf{wdn_i} \leftarrow \mathsf{wd_i}$
On input (pay, \$X) from $\mathcal{Z}$:
  $\mathsf{Contract_{Pay}} \leftarrow \mathcal{G}_{\mathsf{ledger}}.\mathsf{contract}(\mathcal{C})$
  If $\$X \leq \mathsf{Contract_{Pay}.deposits_i} + \mathsf{paid_i} - \mathsf{pay_i} - \mathsf{wd_i}$:
    $\mathsf{arr_i} \leftarrow \$X$
    $\mathsf{pay_i} += \$X$
On input (withdraw, \$X) from $\mathcal{Z}$:
  $\mathsf{Contract_{Pay}} \leftarrow \mathcal{G}_{\mathsf{ledger}}.\mathsf{contract}(\mathcal{C})$
  If $\$X \leq \mathsf{Contract_{Pay}.deposits_i} + \mathsf{paid_i} - \mathsf{pay_i} - \mathsf{wd_i}$:
    $\mathsf{wd_i} += \$\mathrm{X}$

Figure 5: Local protocol for parties to follow for a payment channel between two parties. Parties can pay, deposit into, or withdraw from the channel.

$\mathcal{F}_{\mathsf{state}}(U, \mathcal{C}, \mathcal{P} = \{P_1, ..., P_N\}, \Delta)$

Initialize $\mathsf{aux}_{in} := [\perp]$, $\mathsf{ptr} := 0$, $\mathsf{state} := \emptyset$, $\mathsf{buf} := \emptyset$, $\mathsf{rnd} := 0$

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

$\mathsf{aux}_{in} := \mathcal{G}_{\mathsf{ledger}}.\mathsf{coutput}(\mathcal{C})$

append aux_{in} to buf

$j := |\mathsf{buf}| - 1$

$\mathsf{ptr} := \mathsf{max}(\mathsf{ptr}, j)$

Proceed in rounds starting at $\mathsf{rnd} := 0$:

$v_{\mathsf{rnd},i} := \perp, \forall i \in \mathcal{P}$

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

If $v_{\mathsf{rnd},i} = \perp$:

$v_{\mathsf{rnd},i} := m$

Leak $(i, v_{\mathsf{rnd},i}) \rightarrow \mathcal{A}$

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

If $(\forall v_{\mathsf{rnd},i} : v_{\mathsf{rnd},i} \neq \perp) \vee (\exists v_{\mathsf{rnd},i} : v_{\mathsf{rnd},i} \neq \perp \wedge \mathcal{G}_{\mathsf{ledger}}.\mathsf{rnd} > \mathsf{deadline})$:

$(\mathsf{state}, o) := U(\mathsf{state}, \{v_{\mathsf{rnd},i}\}_{i \in \mathcal{P}}, \mathsf{aux}_{in}[\mathsf{ptr}])$

$\mathsf{rnd} := \mathsf{rnd} + 1$, $\mathsf{deadline} := \mathcal{G}_{\mathsf{ledger}}.\mathsf{rnd} + \Delta$

If $(\forall P_i : P_i.\mathsf{ishonest})$:

$\forall P_i$: **Buffer** $(\mathsf{state}, 1, P_i)$

Else : $\forall P_i$: **Buffer** $(\mathsf{state}, O(\Delta), P_i)$

If $o \neq \perp$:

Send $(\mathsf{transfer}, \mathcal{C}, 0, (output, o), \perp) \rightarrow \mathcal{G}_{\mathsf{ledger}}$

Figure 6: The ideal functionality $\mathcal{F}_{\mathsf{state}}$. The functionality proceeds in rounds and waits for parties to provide input. When all parties have provided input or the round deadline has passed, a state update is executed. Contract output is given to $\mathcal{G}_{\mathsf{ledger}}$in the form of a transaction. Parties must explicitly ping the functionality in order to make progress.

$\mathcal{F}_{\mathsf{pay}}(P_L, P_R, \Delta)$

Initialize $\mathsf{bal_L} := 0, \mathsf{bal_R} := 0$
On input (pay, \$X) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  If $\mathsf{bal}_i < \$X$: ignore
  Leak $(\mathsf{pay}, P_i, \$X) \to \mathcal{A}$
  $\mathsf{bal}_i - = \$X$
  If $P_{\neg i}$ is honest: **Buffer** $((\mathsf{receive}, \$X), 1, P_{\neg i})$
  Else : **Buffer** $((\mathsf{receive}, \$X), O(\Delta), P_{\neg i})$
On input (withdraw, \$X) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  If $\mathsf{bal}_i < \$X$: ignore
  Leak $(\mathsf{withdraw}, P_i, \$X) \to \mathcal{A}$
  $\mathsf{bal}_i - = \$X$
  Send $(\mathsf{transfer}, (\mathbf{sid}, \mathbf{pid}), \$X, \perp, \mathsf{mysidsomehow}) \to \mathcal{G}_{\mathsf{ledger}}$
On input (deliver, msg,P_i) :
  If $(\mathsf{receive}, e) = msg$:
    $\mathsf{bal}_i + = \$X$

Figure 7: The payment channel functionality. Unlike $\mathcal{F}_{\mathsf{state}}$, doesn't need any notion of rounds until it must deal with on-chain transactions for deposits. Buffering for $O(\Delta)$ rounds implies the adversary can choose the number.

Wrapper $\mathcal{W}(\mathcal{F}, \mathcal{C}_1, ..., \mathcal{C}_k)$

Initialize $\mathsf{outputs} := \emptyset$, $\mathsf{buffer} := \emptyset$
On input $(buffer, msg, \delta, P_i)$ from $\mathcal{F}$:
  $\mathsf{buffer}[\mathcal{G}_{\mathsf{ledger}}.\mathsf{rnd} + \delta].\mathsf{append}(msg, P_i)$
On input (read) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  $\mathsf{out} := \mathsf{outputs}[P_i]$
  $\mathsf{outputs}[P_i] := \emptyset$
  Send $\mathsf{out} \to P_i$
All other input m from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  Send $m \to \mathcal{F}$

When activated, do the following subroutine before processing the message:
  For $(msg, P_i) \in \mathsf{buffer}[\mathcal{G}_{\mathsf{ledger}}.\mathsf{rnd}]$:
    Send $(\mathsf{deliver}, msg, P_i) \to \mathcal{F}$
    $\mathsf{outputs}[P_i].\mathsf{append}(msg)$

Figure 8: The wrapper $\mathcal{W}$ that provides common function for all functionalities. In $\mathcal{F}_{\mathsf{state}}$ for example, the wrapper enables functionalities to buffer sending output to the parties in the protocol. When the wrapper sends a message to its functionality $\mathcal{F}$, it does not constitute an ITM to ITM write as they are both running on the same ITM.

$\mathcal{F}_{\mathsf{bcast}}(p_L, p_1...p_n)$

Initialie $\mathsf{buffer} := \emptyset$, $\mathsf{lastRound} \leftarrow -1$, $\mathsf{round} \leftarrow 0$
On input (broadcast, msg) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  If $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid}) \neq p_L$: ignore
  Leak $(\mathsf{msg}, \mathsf{round} + 1) \rightarrow \mathcal{A}$
  $\mathsf{buffer}[\mathsf{round} + 1] \leftarrow \mathsf{msg}$
On input (deliver, msg, to) from $\mathcal{A}$:
  If $\mathsf{to} \notin (p_1, ..., p_n)$: ignore
  $m, r \leftarrow \mathsf{msg}$
  If $\mathsf{m} \in \mathsf{buffer}[r]$:
    Send $(m) \rightarrow \mathsf{to}$

When activated do that following first:
  Send $(\mathsf{clockread},) \rightarrow \overline{\mathcal{G}}_{\mathsf{clock}}$
  $\mathsf{rnd} \leftarrow wait(\overline{\mathcal{G}}_{\mathsf{clock}})$
  If $\mathsf{rnd} > \mathsf{round}$:
    $\mathsf{lastRound} \leftarrow \mathsf{round}$
    $\mathsf{round} \leftarrow \mathsf{rnd}$

$\overline{\mathcal{G}}_{\mathsf{clock}}$

Intialize $\mathsf{registry} := \emptyset$, $\mathsf{dp} := \emptyset$, $\mathsf{sessionT} := \emptyset$
On input (register) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  If $\mathsf{pid} \notin \mathsf{registry}[\mathsf{sid}]$:
    Add pid to $\mathsf{registry}[\mathsf{sid}]$
    If $\mathsf{sid} \notin \mathsf{sessionT}$:
      $\mathsf{sessionT}[\mathsf{sid}] := 0$
On input (clockread) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  If $\mathsf{sid} \notin \mathsf{registry}$: ignore
  Send $\mathsf{sessionT}[\mathsf{sid}] \rightarrow P_i$
On input (clockupdate) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  If $\mathsf{sid} \notin \mathsf{registry}$: ignore
  $\mathsf{dp}[\mathsf{sid}, \mathsf{pid}] := 1$
  If $\forall \mathsf{p}$, $\mathsf{dp}[\mathsf{sid}, \mathsf{p}] = 1$:
    $\mathsf{sessionT}[\mathsf{sid}] + = 1$

$\Pi_{\mathsf{state}}(\mathsf{sid}, \mathsf{pid}, U, \mathcal{C}_{\mathsf{aux}}, \mathcal{C}_{\mathsf{state}}, \mathsf{leader}, \mathsf{peers} = p_1, ...p_n)$

Initialize $\mathsf{round} := 0$, $\mathsf{pinputs} := \emptyset$, $\mathsf{aux_in} = []$, $\mathsf{flag} := \mathsf{OK} \in \{\mathsf{OK}, \mathsf{PENDING}\}$, $\mathsf{aux\_out} := \emptyset$,
$\mathsf{state} := \emptyset, \mathsf{psigs} := \emptyset, \mathsf{lastRound} := -1$
$\mathsf{step} := \mathit{input} \in \{\mathit{input}, \mathit{batch}, \mathit{commit}\}$

If $\mathsf{pid} = \mathsf{leader}$, do the following:

On input (INPUT, v_i, r) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  If $r \neq \mathsf{round}$: ignore
  If first input from P_i in round r: Add v_i to $\mathsf{pinputs}$
  If $\forall p_i, v_i \in \mathsf{pinputs}$:
    Send $(\mathsf{BATCH}, \mathsf{r}, \mathsf{aux_in}, \mathsf{pinputs}) \rightarrow \mathcal{F}_{\mathsf{bcast}}$
On input (SIGN, σ, r) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  If $\mathsf{step} \neq \mathit{commit}$ or $\mathsf{r} \neq \mathsf{round}$ or $\mathsf{Verify}(\sigma, \mathsf{r}, \mathsf{aux\_out}, \mathsf{state}) \neq 1$: ignore
  If first sign from P_i in round r: Add $(P_i, \mathsf{r}, \sigma)$ to psigs
  If $\forall p_i, (p_i, \mathsf{r}, \_) \in \mathsf{psigs}$:
    Send $(\mathsf{COMMIT}, \mathsf{r}, \{\sigma\}_i) \rightarrow \mathcal{F}_{\mathsf{bcast}}$

If $\mathsf{flag} = \mathsf{OK}$:

On input (input, v) from $\mathcal{Z}$:
  If $\mathsf{step} \neq \mathit{input}$ or $r \neq \mathsf{round}$: ignore
  $\mathsf{step} := \mathit{batch}$
  Send $(\mathsf{INPUTS}, \mathsf{v}, \mathsf{round}) \rightarrow \mathsf{leader}$
On input (BATCH, r, aux_in, pinputs) from $\mathcal{F}_{\mathsf{bcast}}$:
  If $\mathsf{step} \neq \mathit{batch}$ or $r \neq \mathsf{round}$: ignore
  $\mathsf{step} := \mathit{commit}$
  todo: how to imply “recent” value of aux_in??
  $\mathsf{state}, \mathsf{aux\_out} := U(\mathsf{state}, \mathsf{pinputs}, \mathsf{aux\_in}, \mathsf{round})$
  $\sigma \leftarrow \mathsf{Sign}(r || \mathsf{aux\_out} || \mathsf{state})$
  Send $(\mathsf{SIGN}, \sigma) \rightarrow \mathsf{leader}$
On input (COMMIT, r, $\{\sigma_r\}_i$) from $\mathcal{F}_{\mathsf{bcast}}$:
  If $\mathsf{r} \neq \mathsf{round}$ or $\mathsf{step} \neq \mathit{commit}$ or $\left(\bigvee_{\sigma_i} \mathsf{Verify}(\sigma_i, \mathsf{r}, \mathsf{aux\_out}, \mathsf{state}) = 0\right)$: ignore
  $\mathsf{lastCommit} := (\mathsf{state}, \mathsf{aux_out}, \{\sigma_r\}_i)$
  $\mathsf{lastRound} := \mathsf{r}$
  $\mathsf{round} := \mathsf{lastRound}{+}1$
  $\mathsf{step} := \mathit{input}$

$\mathcal{F}^{\delta,\ell}_{\mathsf{BD-SEC}}(p_s, p_r)$

Initialize $M := \perp$ and $D := 1$ and $\hat{D} := 1$
On input (m) from $\mathbf{p_s}$:
  Set $D := 1$, $M := m$
  Leak (**send**,M) $\rightarrow \mathcal{A}$
On input (fetch) from p_r:
  Set $D = D - 1$
  If $D = 0$: **Send** (**sent**,M) $\rightarrow p_r$
On input (delay, T) from $\mathcal{A}$:
  If $\hat{D} + T \leq \delta$:
    $D := D + T$, $\hat{D} := \hat{D} + T$
    Send (**delay**,T) $\rightarrow \mathcal{A}$
On input (replace, m', T') from $\mathcal{A}$:
  If p_s corrupted, $D > 0$ and T' is valid:
    $D := T'$ and $M' = m$

$\mathcal{F}_{\mathsf{Bracha}}(\mathcal{D}, \mathcal{P} = p_1, ..., p_n)$

See $\mathcal{F}^{f,Rnd}_{\mathsf{SFE}}$ in Katz.

Simulator S_{Bracha}

Simulate real-world parties $\overline{\mathcal{P}} = p_1, ..., p_n$ and $\mathcal{F}_{\mathsf{BD-SEC}}(p_i, p_j), \forall p_i, p_j \in \overline{\mathcal{P}}$
Simulate instance $\overline{\mathcal{F}}$ of $\mathcal{F}_{\mathsf{clock}}$.
Designate same dealer $\overline{\mathcal{D}}$ as environment.
Simulate dummy adversray $\mathcal{A}_{\mathcal{D}}$

Case #1 (Dishonest $\mathcal{D}$):
On input (input, v) from $\mathcal{Z}$ for $\mathcal{D}$:
  Send (input,v) $\rightarrow \mathcal{A}_{\mathcal{D}}$ *(Passthrough for corrupted parties in real world)*
On input (m) from $\mathcal{Z}$:
  Send (m) $\rightarrow \mathcal{A}_{\mathcal{D}}$
On input (activates, p_j) from $\mathcal{F}_{\mathsf{Bracha}}$:
  If first message in round r:
    Deliver messages from $\mathcal{F}_{\mathsf{BD-SEC}}(p_j, p_i)$ to p_i through (fetch) and simulate state changes.

When protocol terminates, obtain output value v. Deliver $v \rightarrow \mathcal{F}_{\mathsf{Bracha}}$ as the dealer $\mathcal{D}$.

$\Pi_{\mathsf{Bracha}}(\mathcal{D}, \mathcal{P} = p_1, ..., p_n)$ in $\mathcal{F}_{\mathsf{BD-SEC}}$-hybrid

Initialize $\mathsf{BQ} := \frac{\mathsf{ceil}(n+t)}{2}$, $\mathsf{init} := crnd$, $\mathsf{out} := \emptyset$

Dealer $\mathcal{D}$ Protocol
On input (input, m) from $\mathcal{Z}$:
  For $p_i \in \mathcal{P}$:
    Send $\mathsf{VAL}(m) \rightarrow \mathcal{F}_{\mathsf{BD-SEC}}(\mathcal{D}, p_i)$

Party p_i Protocol
On input ($\mathsf{VAL}(m)$) from $\mathcal{F}_{\mathsf{sync},\mathcal{D},p_i}$ (once, round $\mathsf{init}+1$):
  For $p_j \in \mathcal{P}$: **Send** $\mathsf{ECHO}(m) \rightarrow \mathcal{F}_{\mathsf{BD-SEC}}(p_i, p_j)$

On input ($\mathsf{ECHO}(m)$) from $\mathcal{F}_{\mathsf{BD-SEC}}(p_j, p_i)$ (round $\mathsf{init}+2$):
  If received $\mathsf{ECHO}(m)$ from BQ parties:
    For $p_j \in \mathcal{P}$: **Send** $\mathsf{READY}(m) \rightarrow \mathcal{F}_{\mathsf{BD-SEC}}(p_i, p_j)$

On input ($\mathsf{READY}(m)$) from $\mathcal{F}_{\mathsf{BD-SEC}}(p_j, p_i)$ (round $\mathsf{init}+3$):
  If received $\mathsf{READY}(m)$ from $2t+1$ parties:
    $\mathsf{out} := m$
On input (output) from $\mathcal{Z}$:
  If $\mathsf{out} \neq \emptyset$: **Output** out
  Else On j^{th} activation in this round:
    Send (fetch) $\rightarrow \mathcal{F}_{\mathsf{BD-SEC}}(p_j, p_i)$
    $m \leftarrow \mathcal{F}_{\mathsf{BD-SEC}}(p_j, p_i)$

If not received $2t+1$ $\mathsf{READY}(\_)$ messages by $\mathsf{init}+4$:
  Output $\perp$

$\mathcal{F}_{\mathsf{clock}}(\mathcal{P})$

Intialize $\forall p_i \in \mathcal{P} : d_i := 0$

On input (RoundOK) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  $d_i = 1$
  If $\forall p_i \in \mathcal{P} : d_i = 1$:
    $d_i := 0$ for all $p_i \in \mathcal{P}$
  Leak $(\mathsf{switch}, \mathbf{P_i}) \rightarrow \mathcal{A}$
On input (clockread) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  Send $d_i \rightarrow \mathbf{P_i}$

Wrapper $\mathcal{W}_{\mathsf{Eventually}}(\mathcal{F})$

Initialize $\mathsf{crnd} := 0$, $\mathsf{lastcrnd} := -1$, $\mathsf{runqueue} := []$

On input (eventually, codeblock e) from $\mathcal{F}$
  Add e to $\mathsf{runqueue}$
  Leak $e \to \mathcal{A}$
On input (deliver, idx) from $\mathcal{A}$:
  $e \leftarrow \mathsf{runqueue}[idx]$
  Delete $\mathsf{runqueue}[idx]$
  Execute e

On every activation:
  $\mathsf{rnd} \leftarrow \mathcal{F}_{\mathsf{clock}}.\mathsf{clockread}$
  If $\mathsf{rnd} \neq \mathsf{crnd}$:
    $\mathsf{lastcrnd} \leftarrow \mathsf{crnd}$
    $\mathsf{crnd} \leftarrow \mathsf{rnd}$

$\mathcal{F}_{\mathsf{3PC}}(\mathcal{D}, \mathcal{P} = p_1, ..., p_n, V_C)$

Initialize $\mathsf{buffer} := \emptyset$, $\mathsf{pending} := False$
$quorum := 0$, $d_t := -1$

On input (input, T) from $\mathcal{D}$:
  If $\mathsf{pending}$: **reject**
  $\mathsf{pending} = True$
  $d_t = \mathsf{crnd} + 2$
  For $p_i \in \mathcal{P}$:
    Eventually **Send** ready $\to p_i$
On input (status, s) from $\mathbf{P_i} = (\mathbf{sid}, \mathbf{pid})$:
  If not $\mathsf{pending}$: **ignore**
  If first “status“ by $\mathbf{P_i}$:
    If $s = OK$: $ok = ok + 1$
    If $s = Abort$: $abort = abort + 1$
  If $ok \geq V_C$:
    $\mathsf{pending} = False$, $ok, abort = 0$, $d_t = -1$
    For $p_i \in \mathcal{P}$:
      Eventually **Send** commit$T \to p_i$
  If $abort \geq V_A$:
    $\mathsf{pending} = False$, $ok, abort = 0$, $d_t = -1$

On every activation:
  If $\mathsf{pending}$ and $\mathsf{crnd} \geq d_t$:
    Remove last element in buffer
    $d_t = -1$, $ok = 0$, $abort = 0$