FastSwap

Mathias Hall-Andersen[1] [0000−0002−0195−6659]

mathias@hall-andersen.dk

Abstract. FastSwap is a simple and concretely efficient contingent payment scheme for complex predicates. FastSwap only relies on symmetric primitives (in particular symmetric encryption and cryptographic hash functions) and avoids 'heavy-weight' primitives such as general ZKP systems. FastSwap is particularly well-suited for applications where the witness or predicate is large (on the order of MBs / GBs) or expensive to calculate (e.g. $\geq 2^{30}$ computation steps or memory). Additionally FastSwap allows predicates to be implemented using virtually any computational model (including branching execution), which e.g. enables practitioners to express the predicate in smart contract languages already familiar to them, without an expensive transformation to satisfiability of arithmetic circuits. The cost of this efficiency during honest execution is a logarithmic number of rounds during a dispute resolution in the presence of a corrupted party. Let the witness be of size $|w|$ and the predicate of size $|P|$, where computing $P(w)$ takes n steps. In the honest case the off-chain communication complexity is $|w| + |P| + c$ for a small constant c, the on-chain communication complexity is c' for a small constant c'. In the malicious case the on-chain communication complexity is $O(\log n)$ with small constants. Concretely with suitable optimizations the number of rounds (on-chain transactions) for a computation of 2^{30} steps can be brought to 2 in the honest case with an estimated cost of ≈ 2 USD on the Ethereum blockchain[1] and to 14 rounds with an estimated cost of ≈ 4 USD in case of a dispute. It is noted that the corrupted party can be made to pay the transaction cost in case of dispute.

Keywords: Contingent payments, Concrete efficiency, Fair exchange, Smart contracts, Provable security, Universal composability, Authenticated data structures.

1 Introduction

1.1 Setting

The setting of FastSwap (and prior work) is one in which there are three parties:

[1] At the time of writing, using a gas price of 10 Gwei (1 ETH = 10^9 Gwei) and with price of Ethereum at 160 USD/ETH. Assuming a one-time library contract has already been published.

The Judge. The judge is an honest party with public state: we assume that the other parties of the protocol can retrieve the full view[2] of the judge at any time. Associated with the judge is the action which is taken whenever a correct witness is provided by the prover – for contingency payments this is transfer of funds from one account to another. The judge is deterministic and our goal is to limit the storage requirements and execution time of the judge.

The Auditor. The auditor is a (possibly malicious) party, which can contest the validity of a witness provided by the prover. In contingent payments the auditor takes the role of the buyer wishing to buy a witness $x \in \mathcal{I}$ for a predicate $P : \mathcal{I} \to \{0, 1\}$, such that $P(x) = 1$.

The Prover. The prover is a (possibly malicious) party, which wishes to convince the judge that she possesses a witness for the predicate. In contingent payments the prover takes the role of the seller wishing to sell a witness $x \in \mathcal{I}$, while guaranteeing payment in exchange for x.

A naive protocol would be to have the prover send x directly to the judge, which then simply verifies $P(x) = 1$. However, if the description or execution time of the predicate is long, this collides with our goal of limiting the computation required by the judge, furthermore, even when both parties are honest this protocols leaks the witness x to the environment since the state of the judge is public.

1.2 Prior Work

Zero-Knowledge Contingent Payments (ZKCP) The zero-knowledge contingent payment construction [4] (by Gregory Maxwell) requires a zero-knowledge proof system able to express the predicate, a semantically secure encryption scheme (Enc) and a collision resistant hash function (CRH). The original formulation is in terms of a seller (acting as the prover), selling a witness to a predicate P to the buyer (acting as the auditor) in exchange for financial compensation. The scheme operates as follows: for a public o, C (chosen by the seller), the seller proves to the buyer in zero-knowledge that he knows w, k st.

$$o = \mathsf{CRH}(k), P(w) = 1, C = \mathsf{Enc}(k, w) \tag{1}$$

The seller then sends o, C and the proof π to the buyer, who aborts the protocol in case π is invalid. Otherwise the buyer posts a transaction (acting as the judge) to the blockchain, which can only be spend by revealing a preimage of o. The seller claims the funds of the transaction using k, whereby the buyer learns k and is able to decrypt C to obtain the witness. Variations of this scheme has been considered[2][10] in applications where supplying π itself leaks information about the witness, e.g. whenever π itself constitutes a 'witness'[3].

[2] State and inputs/outputs

[3] An example being Proofs-of-Storage, where a Proof-of-Knowledge for a Proof-of-Storage on a given challenge is itself a Proof-of-Storage.

FairSwap The FairSwap[1] protocol (by Stefan Dziembowski, Lisa Eckey, Sebastian Faust) avoids the need for a Zero-Knowledge proof system at the cost of transmitting the entire encrypted computation trace. Additionally FairSwap requires that the predicate be computed using a straight-line program. The execution model is a computational circuit: an acyclic graph wherein every vertex/gate applies an operation to its children/inputs. The scheme operates by having the prover evaluate and encrypt the full execution trace (initial inputs and outputs of every gate), then the prover computes a Merkle commitment to the encrypted execution trace and sends this to the judge. The encrypted execution trace is transfered to the auditor, who recomputes the Merkle tree and verifies that it is consistent with the one held by the judge. Then the decryption key is sent by the prover to the judge and the auditor decrypts the execution trace. If any gate is applied incorrectly (or the output of the computation is not accepting), the auditor can prove Merkle paths to the inputs of the erroneously applied gate and convince the judge that the prover is malicious. The FairSwap protocol (as formulated) assumes that the full predicate description is available to the judge, which makes it best suited for applications where the predicate is has a small description but potentially a long running time: the example in the paper being the computation of a Merkle hash which allows the purchasing of files, where the linear communication complexity of FairSwap in the length of the trace is optimal. FastSwap is inspired by the FairSwap protocol.

1.3 Features of FastSwap

Simple & efficient primitives. The FastSwap protocol does not reply on 'heavy weight' primitives like zero-knowledge proof systems, a central goal of FastSwap is to provide concrete efficiency for a wide class of very large predicates.

Constant communication in the honest case. The communication complexity during honest execution is the size of the program, the size of the witness and a small constant. The communication complexity is independent of the length of the execution for the predicate.

Logarithmic communication for dispute resolution. In case of a malicious prover or auditor, dispute resolution for an execution trace of n steps is completed within $O(\log n)$ rounds and $O(\log n)$ communication with small constants.

Flexible execution model. Previous work require that the predicate is implemented via straight-line program, FastSwap additionally supports efficient branching execution and RAM machines. One possible application is to enable efficient compilation of existing smart contract languages to predicates for contingent payments.

Efficient for large program descriptions. The program description of the predicate need only be available to the prover and auditor, this allows executing program with large descriptions. This also allows deployment of a generic ‘interpreter & dispute resolution‘ judge contract, which can be reused for selling different witnesses to different predicates by different parties.

2 Notation

Symbols enclosed in angle brackets $\langle \cdot \rangle$ represents unique symbols (‘atoms’), e.g. $\langle Identifier \rangle$ is simply a symbol recognized by all participants in the protocol. The length of a bit string s is denoted by $|s|$. Throughout the article κ will denote a security parameter.

3 Primitives

3.1 Symmetric Encryption

Definition 1 (Symmetric Key Encryption). *A symmetric encryption schemes is a family two algorithms running on 1^κ (omitted for brevity):*

- *A PPT algorithm, which samples uniformly from the key space* $k \xleftarrow{\$} \mathcal{K}_\kappa$
- *A PPT algorithm ‘encryption’* $\mathsf{Enc} : \mathcal{K}_\kappa \times \mathcal{M} \to \mathcal{C}_\kappa$
- *A PPT algorithm ‘decryption’* $\mathsf{Dec} : \mathcal{K}_\kappa \times \mathcal{C}_\kappa \to \mathcal{M}$

Satisfying perfect completeness:

$$\forall m \in \mathcal{M} : 1 = \mathbb{P}[\mathsf{Dec}(k, \mathsf{Enc}(k, m)) = m : k \xleftarrow{\$} \mathcal{K}_\kappa]$$

Definition 2 (Semantic Security). *A family of symmetric encryption schemes (Definition 1) is said to be semantically secure if for all pairs of PPT algorithms (A_1, A_2), there exists a negligible function* negl *st.*

$$\begin{aligned} 1/2 + \mathsf{negl}(\kappa) \geq \mathbb{P}[b' = b \wedge |m_1| = |m_2| : (m_1, m_2) \leftarrow A_1(1^\kappa), \\ b \xleftarrow{\$} \{0,1\}, k \xleftarrow{\$} \mathcal{K}_\kappa, b' \leftarrow A_2(1^k, \mathsf{Enc}(k, m_b))] \end{aligned}$$

3.2 Collision Resistant Hashes

Definition 3 (Cryptographic Hash). *A family of cryptographic hash functions consists of an efficient deterministic algorithm running on 1^κ:*

- *A polynomial time algorithm ‘hash‘* $\mathsf{CRH} : \{0,1\}^* \to \mathcal{H}_\kappa$

Where $\forall h \in \mathcal{H}_\kappa : |h| = \kappa$

Definition 4 (Collision Resistantance). *A hash function family (Definition 3) is said to be collision resistant if for every PPT algorithm A, there exists a negligible function* negl *st.*

$$\mathsf{negl}(\kappa) \geq \mathbb{P}[m \neq m' \wedge \mathsf{CRH}(m) = \mathsf{CRH}(m') : (m, m') \leftarrow A(1^\kappa)]$$

3.3 Binding & Hiding Commitments

Definition 5 (Commitment). *A commitment scheme is a family of two efficient algorithms running on 1^κ (omitted for brevity):*

- *A PPT algorithm 'commit'* $\mathsf{Comm} : \mathcal{R}_\kappa \times \mathcal{M} \to \mathcal{C}_\kappa$
- *A PPT algorithm 'open'* $\mathsf{Open} : \mathcal{R}_\kappa \times \mathcal{M} \times \mathcal{C}_\kappa \to \{0, 1\}$

Satisfying perfect completeness:

$$\forall m \in \mathcal{M} : 1 = \mathbb{P}[\mathsf{Open}(r, m, c) = 1 : r \xleftarrow{\$} \mathcal{R}_\kappa, c \leftarrow \mathsf{Comm}(r, m)]$$

Definition 6 (Computationally Binding Commitment). *A commitment scheme (Definition 5) is said to be computationally binding if for all PPT algorithm A there exists a negligible function* negl *st.*

$$\mathsf{negl}(\kappa) \geq \mathbb{P}[m_1 \neq m_2 \wedge \mathsf{Open}(r_1, m_1, c) = 1 \wedge \mathsf{Open}(r_2, m_2, c) = 1 : (c, r_1, r_2, m_1, m_2) \leftarrow A(1^\kappa)]$$

Definition 7 (Computationally Hiding Commitment). *A commitment scheme (Definition 5) is said to be computationally hiding if for all pairs of PPT algorithms (A_1, A_2) there exists a negligible function* negl *st.*

$$1/2 + \mathsf{negl}(\kappa) \geq \mathbb{P}[b' = b : (m_1, m_2) \leftarrow A_1(1^\kappa), b \xleftarrow{\$} \{0, 1\}, r \xleftarrow{\$} \mathcal{R}_\kappa, b' \leftarrow A_2(1^k, \mathsf{Comm}(r, m_b))]$$

4 Authenticated Computation Structures

Definition 8 (Authenticated Data Structure). *An authenticated data structure scheme consists of a set of possible states $\mathcal{S}_\kappa$, a set of tags $\mathcal{T}_\kappa$, a set of possible operations $\mathcal{O}$, a set of results $\mathcal{R}$, a set of descriptions of initial states $\mathcal{I}$ and four deterministic polynomial time algorithms:*

- $\mathsf{Initial} : \mathcal{I} \to \mathcal{S}$. *Construct an initial state from a description.*
- $\mathsf{Tag} : \mathcal{S} \to \mathcal{T}_\kappa$. *Compute a succinct 'tag' of the state.*
- $\mathsf{Apply} : \mathcal{S} \times \mathcal{O} \to \mathcal{S} \times \mathcal{R} \times \mathcal{P}_\kappa$. *Apply an operation to the state, yielding a new state, a result*[4]*, and a proof of correctness, which can be verified using the operation and the tags of the previous and resulting state.*
- $\mathsf{Verify} : \mathcal{T}_\kappa \times \mathcal{T}_\kappa \times \mathcal{O} \times \mathcal{R} \times \mathcal{P}_\kappa \to \{1, 0\}$. *Verifies the execution of an operation on the state corresponding to the tag of the previous state and tag of the resulting state after application of the operation.*

[4] e.g. the associated value of a key lookup in a tree structure

Satisfying perfect completeness:

$$S \in \mathcal{S}, O \in \mathcal{O} : \mathsf{Verify}(T, T', R, O, \pi) = 1 \textit{ where} \\ (S', R, \pi) \leftarrow \mathsf{Apply}(S, O), T \leftarrow \mathsf{Tag}(S), T' \leftarrow \mathsf{Tag}(S')$$

Computation is formulated in terms of 'Authenticated Computation Structures', which can be seen as an authenticated data structure scheme, wherein the operation is uniquely defined by the current state of the data structure and an immutable 'environment'.

Definition 9 (Authenticated Computation Structure). *An authenticated computation structure scheme consists of an input space $\mathcal{I}$ containing descriptions of of initial computations states, a space of possible computation structures $\mathcal{S}$, a space of possible 'environments' $\mathcal{E}$, a set of 'tag' values $\mathcal{T}_\kappa$, a set of proofs $\mathcal{P}_\kappa$ and five deterministic polynomial time algorithms:*

- $\mathsf{Initial} : \mathcal{I} \to \mathcal{S}$. *Construct an initial state from a description.*
- $\mathsf{Tag} : \mathcal{S} \to \mathcal{T}_\kappa$. *Produce a succinct tag corresponding to the structure.*
- $\mathsf{Step} : \mathcal{E} \times \mathcal{S} \to \mathcal{S}$. *Progresses the computation by 'a single step'.*
- $\mathsf{Prove} : \mathcal{E} \times \mathcal{S} \to \mathcal{P}_\kappa$. *Produce a succinct proof of correct execution.*
- $\mathsf{Verify} : \mathcal{E} \times \mathcal{T}_\kappa \times \mathcal{T}_\kappa \times \mathcal{P}_\kappa \to \{1, 0\}$. *Verify the execution of a step.*

Satisfying perfect completeness:

$$e \in \mathcal{E}, S \in \mathcal{S} : \mathsf{Verify}(e, T, T', \pi) = 1 \textit{ where} \\ S' \leftarrow \mathsf{Step}(e, S), \pi \leftarrow \mathsf{Prove}(e, S), T \leftarrow \mathsf{Tag}(S), T' \leftarrow \mathsf{Tag}(S')$$

i.e. verification succeeds for every pair of successive computation structures.

The primitive is directly related to authenticated data structures (Defintion 8) and can be generically constructed from such schemes by defining a function $\mathsf{Operation} : \mathcal{S} \to \mathcal{O} \times \mathcal{P}_\kappa$ which takes the state of the data structure and returns the next operation to apply and a proof, then deriving an implementation of the algorithms above in the obvious way. A concrete example of this pattern is provided in Section 7. The motivation for adding the environment argument is to permit I to contain input encrypted under a key contained in the environment, such that I leaks at most the length of the input.

Definition 10 (Computational Integrity). *An authenticated computation structure scheme is said to provide integrity, if for every PPT algorithm A, there exists a negligible function negl such that:*

$$\mathsf{negl}(\kappa) \geq \mathbb{P}[\mathsf{Step}(e, S) \neq S' \wedge \mathsf{Verify}(e, T, T', \pi) = 1 : \\ (e, S, S', \pi) \leftarrow A(1^\kappa), T \leftarrow \mathsf{Tag}(S), T' \leftarrow \mathsf{Tag}(S')]$$

Note that computational integrity (Definition 10) implies in particular that $\mathsf{Tag} : \mathcal{S} \rightarrow \mathcal{T}_\kappa$ is a collision resistant hash function (Definition 4). For later convience we define some simple functions which are derieved from an authenticated computational structure scheme:

Definition 11 ($\mathsf{Terminate} : \mathcal{E} \times \mathcal{S} \rightarrow \mathbb{N}_+$). *$\mathsf{Terminate}$ repeatedly applies Step and returns the number of steps before an accepting or rejecting state is reached. Formally, with the patterns being matched by preference from top to bottom:*

$$\begin{aligned} \mathsf{Terminate}(e, S) &:= 1 \text{ where } S \in \{\langle Accept\rangle, \langle Reject\rangle\} \\ \mathsf{Terminate}(e, S) &:= 1 + \mathsf{Terminate}(e, S') \text{ where } S' \leftarrow \mathsf{Step}(e, S) \end{aligned}$$

Where $\langle Accept\rangle$ and $\langle Reject\rangle$ is uniquely recognized accepting and rejecting terminal states respectively.

Definition 12 ($\mathsf{StepN} : \mathcal{E} \times \mathcal{S} \times \mathbb{N}_+ \rightarrow \mathcal{S}$). *$\mathsf{StepN}$ applies Step a specified number of times and returns the resulting state. Formally, with the patterns being matched by preference from top to bottom:*

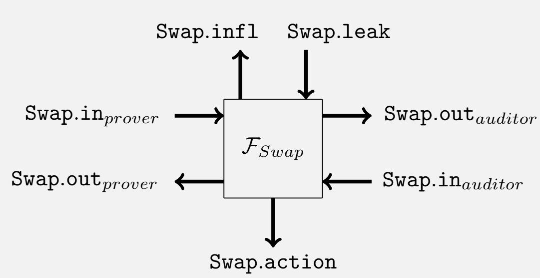
$$\begin{aligned} \mathsf{StepN}(e, S, 1) &:= S \\ \mathsf{StepN}(e, S, *) &:= S \text{ where } S \in \{\langle Accept\rangle, \langle Reject\rangle\} \\ \mathsf{StepN}(e, S, n) &:= \mathsf{StepN}(e, S', n-1) \text{ where } S' \leftarrow \mathsf{Step}(e, S) \end{aligned}$$

Where $\langle Accept\rangle$ and $\langle Reject\rangle$ is uniquely recognized accepting and rejecting terminal states respectively. One can think of StepN as returning the n'th step of the computation right-padded by the final accepting/rejecting state.

5 Ideal Functionalities

We formulate the behavior of FastSwap using the universal composability (UC) framework with the style and notation of Cramer, et al. [8]. The $\mathcal{F}_{Swap}$ functionality captures the desired behavior of a contingent exchange protocol:

Agent $\mathcal{F}_{Swap}$



Initialization: set `can_abort` $\leftarrow 1$

- Wait for one of three messages on `Swap.infl`.
  - ◇ $\langle Auditor\rangle$: Mark the prover as corrupted, by ignoring any message on $\texttt{Swap.in}_{auditor}$. Whenever a message of the form $(\langle Send\rangle, m)$ is received on `Swap.infl` act as if m was received on $\texttt{Swap.in}_{auditor}$. Whenever a message m is output on $\texttt{Swap.out}_{auditor}$, also output m on `Swap.leak`
  - ◇ $\langle Prover\rangle$: Mark the prover as corrupted, by ignoring any message on $\texttt{Swap.in}_{prover}$. Whenever a message of the form $(\langle Send\rangle, m)$ is received on `Swap.infl` act as if m was received on $\texttt{Swap.in}_{prover}$. Whenever a message m is output on $\texttt{Swap.out}_{prover}$, also output m on `Swap.leak`
  - ◇ $\langle Honest\rangle$. Indicating no corruption.

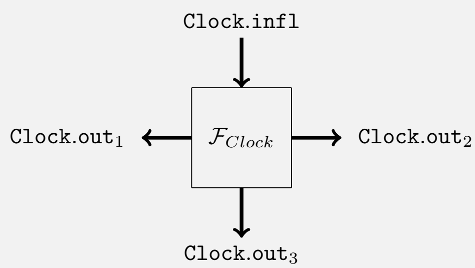
  Ignore any subsequent corruption messages.
- Any time, on input $\langle Abort\rangle$ on `Swap.infl`, $\texttt{Swap.in}_{auditor}$ or $\texttt{Swap.in}_{prover}$ and if `can_abort` $= 1$, then abort the protocol:
  - Output $\perp$ on $\texttt{Swap.out}_{prover}$.
  - Output $\perp$ on $\texttt{Swap.out}_{auditor}$.
  - Output $\perp$ on `Swap.leak`.
  - Ignore any further messages on any in port.
- On input P on $\texttt{Swap.in}_{auditor}$:
  - Store P.
  - Output P on $\texttt{Swap.out}_{prover}$.
  - Output $|P|$ on `Swap.leak`.
- On input w on $\texttt{Swap.in}_{prover}$:
  - Store w.
  - Output $|w|$ on `Swap.leak`.
- On input $\langle Swap\rangle$ on $\texttt{Swap.in}_{prover}$, when both P, w has been set:
  - Set `can_abort` $\leftarrow 0$.
  - Output w on $\texttt{Swap.out}_{auditor}$.
  - If either party is corrupted leak the entire state of the functionality on `Swap.leak`: every message sent and received by the functionality.
- On input $\langle Action\rangle$ on `Swap.infl`, when `can_abort` $= 0$:
  - Interpret P as a description of a computable function.
    Output $P(w)$ on `Swap.action` and `Swap.leak`.

The $\mathcal{F}_{Swap}$ functionality leaks its entire state after `can_abort` $= 0$ whenever a corrupted party is present. Intuitively we can accept to leak the witness to the world in case of corruption after the protocol cannot be aborted, since after `can_abort` $= 0$ the corrupted party will posses the witness and could publish this (outside the scope of the protocol) regardless. Hiding of the witness must only be ensured as long as `can_abort` $= 1$ or whenever both parties are honest.

The separation of the $\langle Swap \rangle$ and $\langle Action \rangle$ messages, enables the implementation to run some 'dispute' protocol in case one of the parties is corrupted, before delivering the output on `Swap.action`. The leaked state after $\langle Swap \rangle$ can be used to simulate the leakage of this 'dispute' protocol.

The $\mathcal{F}_{Clock}$ functionality models n monotonically increasing clocks, where the drift between any pair of clocks is bounded by a constant Δ:

Agent $\mathcal{F}_{Clock}(n, \Delta)$



Initialization. Set $t \leftarrow 0$, For $i \in [1, n]$, set $t_i \leftarrow t$, output t_i on $\texttt{Clock.out}_i$.

- On input $\langle Clock \rangle$ on `Clock.infl`. Set $t \leftarrow t + 1$. For $i \in [1, n]$, set $t_i \leftarrow \max\{t_i, t - \Delta\}$. For $i \in [1, n]$ output t_i on $\texttt{Clock.out}_i$.
- On input $(\langle Update \rangle, t'_i)$ on `Clock.infl`. If $t_i < t'_i \leq t$, set $t_i \leftarrow t'_i$ and output t_i on $\texttt{Clock.out}_i$. Otherwise ignore the message.

This formulation allows instantiation of the functionality using a blockchain which offers 'finality' guarantees; ensuring that the view of the honest parties cannot be rolled back past finalized blocks. Furthermore, one needs to assume that the view of any node is at most Δ blocks behind the most recently finalized block.

The $\mathcal{F}_{Channel}$ functionality models an authenticated and encrypted channel between the prover and auditor, which guarantees in-order delivery of messages:

Agent $\mathcal{F}_{Channel}$

Initialization: create two empty lists: set $M_{prover} \leftarrow \epsilon, M_{auditor} \leftarrow \epsilon$.

- Wait for one of three messages on `Channel.infl`
  - ◇ $(\langle Corrupt \rangle, p) : p \in \{\langle Prover \rangle, \langle Auditor \rangle\}$: Mark the party p as corrupted and allow control of the ports of p as follows:
    - ∗ By ignoring any message on $\texttt{Channel.in}_p$.
    - ∗ Whenever a message of the form $(\langle Send \rangle, m)$ is received on `Channel.infl` act as if m was received on $\texttt{Channel.in}_p$.

- ∗ Whenever a message of the form $(\langle Recv \rangle, m)$ is received on `Channel.infl` output m on $\texttt{Channel.out}_p$.
- ∗ Whenever a message m is output on $\texttt{Channel.out}_p$ output m on `Channel.leak` instead of $\texttt{Channel.out}_p$.

◇ $\langle Honest \rangle$. Indicating no corruption.

Ignore any subsequent corruption messages.

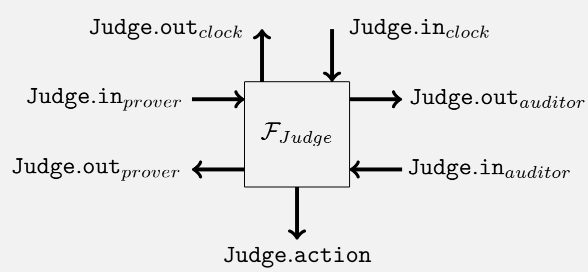
– On input m on $\texttt{Channel.in}_{prover}$:
  - Push m to the back of $M_{auditor}$
  - Output $(\langle Auditor \rangle, |m|)$ on `Channel.leak`.

– On input m on $\texttt{Channel.in}_{auditor}$:
  - Push m to the back of M_{prover}
  - Output $(\langle Prover \rangle, |m|)$ on `Channel.leak`.

– On input $(\langle Deliver \rangle, p)$ on `Channel.infl`:
  - If M_p is not empty, pop the front-most element m and output m on $\texttt{Channel.in}_p$.

The judge is instantiated with a description D of its transition function, which both parties must agree upon. Whenever the judge receives input, this is provided to all parties and leaked, reflecting that the state of the judge is completely public. The judge furthermore has access to a clock functionality and an 'action' port, which will later correspond to the action port of the $\mathcal{F}_{Swap}$ functionality:

Agent $\mathcal{F}_{Judge}$



– Wait for one of three messages:
  - ◇ $\langle Auditor \rangle$: Mark the prover as corrupted, by ignoring any message on $\texttt{Judge.in}_{auditor}$. Whenever a message of the form $(\langle Send \rangle, m)$ is received on `Judge.infl` act as if m was received on $\texttt{Judge.in}_{auditor}$.
  - ◇ $\langle Prover \rangle$: Mark the prover as corrupted, by ignoring any message on $\texttt{Judge.in}_{prover}$. Whenever a message of the form $(\langle Send \rangle, m)$ is received on `Judge.infl` act as if m was received on $\texttt{Judge.in}_{prover}$.

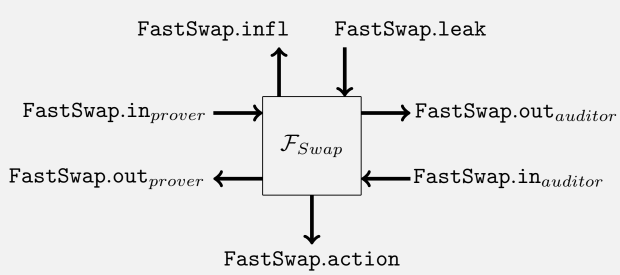
◇ $\langle Honest \rangle$. Indicating no corruption.

Ignore any subsequent corruption messages.

- Whenever t_{new} is received on $\texttt{Judge.in}_{clock}$, store $t \leftarrow t_{new}$.
- On input D on $\texttt{Judge.in}_{auditor}$: output D on `Judge.leak`, output D on $\texttt{Judge.out}_{auditor}$, store D.
- On input D' on $\texttt{Judge.in}_{prover}$: if $D \neq D'$, output $\perp$ on $\texttt{Judge.out}_{prover}$, output $\perp$ on $\texttt{Judge.out}_{auditor}$ and abort the protocol, by ignoring any subsequent messages on all in ports. Otherwise set $S \leftarrow \epsilon$ and begin processing input messages.
- On input m on $\texttt{Judge.in}_{auditor}$: output (m, t) on `Judge.leak`, output (m, t) on $\texttt{Judge.out}_{auditor}$, output (m, t) on $\texttt{Judge.out}_{prover}$, update the state $(S, r) \leftarrow D(S, \langle Auditor \rangle, m, t)$, if $r \neq \epsilon$ output r on `Judge.action`.
- On input m on $\texttt{Judge.in}_{prover}$: output (m, t) on `Judge.leak`, output (m, t) on $\texttt{Judge.out}_{auditor}$, output (m, t) on $\texttt{Judge.out}_{prover}$, update the state $(S, r) \leftarrow D(S, \langle Prover \rangle, m, t)$, if $r \neq \epsilon$ output r on `Judge.action`.

The *FastSwap* functionality enables the two parties to agree on the initial state of a authenticated computation scheme, then allows the prover to input an environment. If repeated application of `Step` on the initial state with the given environment terminates in an accepting state the functionality outputs 1 on `FastSwap.action`, otherwise the functionality outputs 0. When both parties are honest the functionality leaks only the environment and the accepting/rejecting outcome of the computation, in particular it does not leak the initial state:

Agent $\mathcal{F}_{FastSwap}$



Initialization: set `can_abort` $\leftarrow 1$

- Wait for one of three messages on `FastSwap.infl`.
  ◇ $\langle Auditor \rangle$: Mark the prover as corrupted, by ignoring any message on $\texttt{FastSwap.in}_{auditor}$. Whenever a message of the form $(\langle Send \rangle, m)$ is received on `FastSwap.infl` act as if m was re-

ceived on $\texttt{FastSwap.in}_{auditor}$. Whenever a message m is output on $\texttt{FastSwap.out}_{auditor}$, also output m on FastSwap.leak
- ⋄ $\langle Prover \rangle$: Mark the prover as corrupted, by ignoring any message on $\texttt{FastSwap.in}_{prover}$. Whenever a message of the form $(\langle Send \rangle, m)$ is received on FastSwap.infl act as if m was received on $\texttt{FastSwap.in}_{prover}$. Whenever a message m is output on FastSwap.out, also output m on FastSwap.leak
- ⋄ $\langle Honest \rangle$. Indicating no corruption.

Ignore any subsequent corruption messages.

- Any time, on input $\langle Abort \rangle$ on FastSwap.infl, $\texttt{FastSwap.in}_{auditor}$ or $\texttt{FastSwap.in}_{prover}$ and if can_abort $= 1$, then abort the protocol:
  - Output $\perp$ on $\texttt{FastSwap.out}_{prover}$.
  - Output $\perp$ on $\texttt{FastSwap.out}_{auditor}$.
  - Output $\perp$ on FastSwap.leak.
  - Ignore any further messages on any in port.
- On input I' on $\texttt{FastSwap.in}_{auditor}$:
  - Store I'.
  - If the prover is corrupted, output I' on FastSwap.leak.
  - Output $\langle Input \rangle$ on FastSwap.leak
- On input I on $\texttt{FastSwap.in}_{prover}$, when I' has been set:
  - Compute $S \leftarrow \mathsf{Initial}(I)$.
  - Compute $S' \leftarrow \mathsf{Initial}(I')$.
  - If $S \neq S'$ then abort the protocol (as if $\langle Abort \rangle$ was received).
- On input e on $\texttt{FastSwap.prover}_{in}$:
  - Set can_abort $\leftarrow 0$
  - Output e on $\texttt{FastSwap.out}_{auditor}$.
  - Output e on FastSwap.leak.
  - If either party is corrupted leak the entire state of the functionality on FastSwap.leak: every message sent and received by the functionality.
- On input $\langle Action \rangle$ on FastSwap.infl, when can_abort $= 0$:
  - Compute $n \leftarrow \mathsf{Terminate}(e, S)$.
  - Output $\mathsf{StepN}(e, S, n) \stackrel{?}{=} \langle Accept \rangle$ on FastSwap.action and FastSwap.leak.

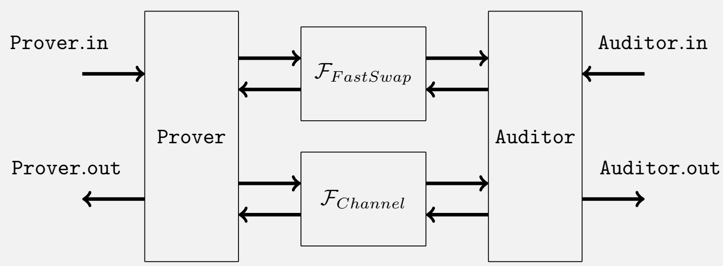
We implement the $\mathcal{F}_{Swap}$ functionality using: $\mathcal{F}_{FastSwap}$, $\mathcal{F}_{Channel}$, a semantically secure encryption scheme (Definition 2)[5] and a sufficiently expressive authenticated computational structure scheme (Definition 9):

The authenticated computational structure scheme must allow expression of the $\mathsf{Dec} : \mathcal{K}_\kappa \times \mathcal{C}_\kappa \to \mathcal{M}$ function as well as the set of predicates. The set of environments for the computational structure scheme must contain $\mathcal{K}_\kappa$. Furthermore we assume that input descriptions $\mathcal{I}$ can be provided in the form (P, W)

[5] For which we require a computable description, hence the application of the IND-CPA encryption scheme is non-blackbox.

where P is the description of a predicate and W is the input to the predicate, st.: repeated application of the $\mathsf{Step} : \mathcal{E} \times \mathcal{S} \to \mathcal{S}$ functions computes $P(e, W)$, where $e \in \mathcal{E}$ is the environment. We can therefore transform the problem in $\mathcal{F}_{Swap}$ of evaluating the predicate P on w, into the problem of repeatedly applying the Step function to the initial state described by $I = (P \circ \mathsf{Dec}(e, \cdot), W)$ where $W \leftarrow \mathsf{Enc}(e, w)$, with $e \in \mathcal{K}_\kappa$ provided as the environment of the Step function. Intuitively this enables us to swap a constant size key in place of the actual witness, which additionally provides semantic hiding of the witness from the auditor while the protocol can still be aborted and from the environment in case of honest execution.

Protocol Π_{Swap} : implement $\mathcal{F}_{Swap}$ from $\mathcal{F}_{FastSwap}$ and $\mathcal{F}_{Channel}$



◇ Respect the abort of $\mathcal{F}_{FastSwap}$:
- Any time on input $\perp$ on `FastSwap.out`$_{auditor}$:
  Output $\perp$ on `Auditor.out`.
- Any time on input $\langle Abort \rangle$ on `Auditor.in`:
  Output $\langle Abort \rangle$ on `FastSwap.in`$_{auditor}$.

Act analogously for the prover.

◇ Any time on input $\langle Action \rangle$ on `Swap.infl`:
- Output $\langle Action \rangle$ on `FastSwap.infl`.

– Auditor. On input P on `Auditor.in`:
- Output P on `Channel.in`$_{auditor}$

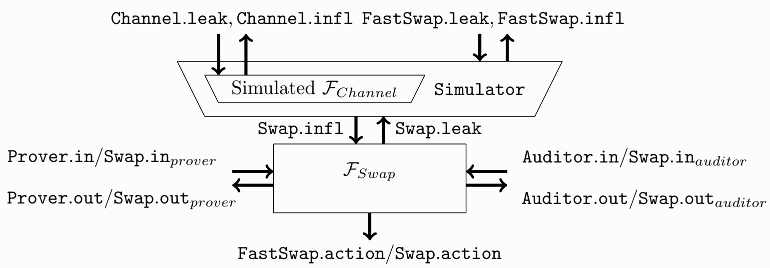
– Prover. On input P' on `Channel.out`$_{prover}$:
- Store P'.
- Output P' on `Prover.out`.

– Prover. On input w on `Prover.in`:
- Sample $k \xleftarrow{\$} \mathcal{K}_\kappa$.
- Compute $E \leftarrow \mathsf{Enc}(k, w)$.
- Define $I = (P' \circ \mathsf{Dec}, E)$.
- Output E on `Channel.in`$_{prover}$.
- Output I on `FastSwap.in`$_{prover}$.

– Auditor. On input E' on `Channel.out`$_{auditor}$:
- Define $I' = (P \circ \mathsf{Dec}, E')$
- Output I' on `FastSwap.in`$_{auditor}$.

- Prover. On input $\langle Swap \rangle$ on `Prover.in`:
  - Output k on $\texttt{FastSwap.in}_{prover}$
- Auditor. On input k on $\texttt{FastSwap.out}_{auditor}$.
  - Output $\mathsf{Dec}(k, E')$ on `Auditor.out`

Simulator $\mathcal{S}_{Swap}$: simulate Π_{Swap} using $\mathcal{F}_{Swap}$



Respect the abort: any time, on input $\langle Abort \rangle$ on `FastSwap.infl`, $\texttt{FastSwap.in}_{auditor}$ or $\texttt{FastSwap.in}_{prover}$: Output $\langle Abort \rangle$ on `Swap.infl`, $\texttt{Swap.in}_{auditor}$ or $\texttt{Swap.in}_{prover}$ respectively. On $\perp$ on `Swap.leak`, output $\perp$ on `FastSwap.leak`

Wait for the corruption pattern for both $\mathcal{F}_{FastSwap}$ and $\mathcal{F}_{Channel}$ (the class of environments is assumed corruption respecting: corrupting the same parties for every functionality):

`Case 1.` Neither party is corrupted:

- On input $|P|$ on `Swap.leak`:
  - Simulate sending $0^{|P|}$ on $\texttt{Channel.in}_{auditor}$.
- On input $|w|$ on `Swap.leak`:
  - Sample $k \xleftarrow{\$} \mathcal{K}_\kappa$.
  - Compute $E \leftarrow \mathsf{Enc}(k, 0^{|w|})$.
  - Simulate sending E on $\texttt{Channel.in}_{prover}$.
- On simulated input E on $\texttt{Channel.in}_{auditor}$:
  - Output $\langle Input \rangle$ on `FastSwap.leak`.
- On input $\langle Action \rangle$ on `FastSwap.infl`:
  - Output $\langle Action \rangle$ on `Swap.infl`.
- On $P(w)$ on `Swap.leak`:
  - Output k on `FastSwap.leak` (as 'e').
  - Output $P(w)$ on `FastSwap.leak`.

`Case 2.` Auditor is corrupted, Prover is honest:
(**Note:** not simulatable, requires random oracle, see proof section.)

- On input P on `Swap.leak`:
  - Simulate sending P on $\texttt{Channel.in}_{auditor}$.
- On simulated input P' on $\texttt{Channel.out}_{prover}$.
  - Store P'
- On input $|w|$ on `Swap.leak`:
  - Sample $k \xleftarrow{\$} \mathcal{K}_\kappa$.
  - Compute $E \leftarrow \mathsf{Enc}(k, 0^{|w|})$.
  - Simulate sending E on $\texttt{Channel.in}_{prover}$.
- On simulated input E' on $\texttt{Channel.in}_{auditor}$:
  - Output $\langle \mathit{Input} \rangle$ on `FastSwap.leak`.
- On $w, P(w)$ on `Swap.leak`:
  - Reprogram Dec using the RO, such that $\mathsf{Dec}(k, E) = w$.
  - Output k on `FastSwap.leak` (as 'e').
  - Output $w, P(w)$ on `FastSwap.leak`.
- On input $\langle \mathit{Action} \rangle$ on `FastSwap.infl`:
  - Output $\langle \mathit{Action} \rangle$ on `Swap.infl`.

`Case 3.` Auditor is honest, Prover is corrupted:

- On input P on `Swap.leak`:
  - Simulate sending P on $\texttt{Channel.in}_{auditor}$.
- On simulated input P' on $\texttt{Channel.out}_{prover}$.
  - Store P'
- On input w on `Swap.leak`:
  - Sample $k \xleftarrow{\$} \mathcal{K}_\kappa$.
  - Compute $E \leftarrow \mathsf{Enc}(k, w)$.
  - Simulate sending E on $\texttt{Channel.in}_{prover}$.
- On simulated input E' on $\texttt{Channel.in}_{auditor}$:
  - Output $\langle \mathit{Input} \rangle$ on `FastSwap.leak`.
- On $w, P(w)$ on `Swap.leak`:
  - Output k on `FastSwap.leak` (as 'e').
  - Output $w, P(w)$ on `FastSwap.leak`.
- On input $\langle \mathit{Action} \rangle$ on `FastSwap.infl`:
  - Output $\langle \mathit{Action} \rangle$ on `Swap.infl`.

Lemma 1 **($\mathit{\Pi}_{Swap} \Diamond \mathcal{F}_{FastSwap} \Diamond \mathcal{F}_{Channel} \geq_{comp} \mathcal{F}_{Swap}$).** *$\mathit{\Pi}_{Swap}$ implements $\mathcal{F}_{Swap}$ using $\mathcal{F}_{FastSwap}$ and $\mathcal{F}_{Channel}$ with respect to all computationally bounded (PPT) environments.*

Proof. *By case analysis on the corruption pattern of the environment:*

`Case 1.` Consider the hybrid $\mathcal{H}_{Swap}$ which is equal to $\mathcal{S}_{Swap}$, except where w is extracted from $\mathcal{F}_{Swap}$ and E is derived as $E \leftarrow \mathsf{Enc}(k, w)$. The difference in the distributions is the leakage on `FastSwap.leak`: $\mathcal{S}_{Swap}$ leaks $E' \leftarrow \mathsf{Enc}(k, 0^{|w|})$, since $|0^{|w|}| = |w|$ the distributions must be computationally indistinguishable by

the assumption that $\mathsf{Enc} : \mathcal{K}_\kappa \times \mathcal{M} \to \mathcal{C}_\kappa$ *is a CPA secure encryption scheme 2.*

`Case 2.` *We assume that* $\mathsf{Enc}, \mathsf{Dec}$ *is non-commiting and implemented using a random oracle (e.g. using a construction from [7]). Consider again a hybrid* $\mathcal{H}_w$ *which is equal to* $\mathcal{S}_{Swap}$*, except where* w *is extracted from* $\mathcal{F}_{Swap}$ *and* E *is replaced with* $E_w \leftarrow \mathsf{Enc}(k, w)$ *since* $|0^{|w|}| = |w|$*,* E_w *and* E *must be computationally indistinguishable by the assumption that* $\mathsf{Enc} : \mathcal{K}_\kappa \times \mathcal{M} \to \mathcal{C}_\kappa$ *is a CPA secure encryption scheme 2. Furthermore since* $k \xleftarrow{\$} \mathcal{K}_\kappa$ *the probability that the environment has queried the oracle on any of the queries made during* $\mathsf{Dec}(k, E)$ *prior to receiving* k *is negligible, hence reprogramming is successful with overwhelming probability. Hence* $\mathcal{H}_w$ *and* $\mathcal{S}_{Swap}$ *are computationally indistinguishable.*

A simulatable alternative in the standard model is to deploy non-committing encryption without random oracles, however this significantly impedes efficiency since k *must have the same size as the witness and hence the communication with the judge would be linear in the size of the witness.*

`Case 3.` *Since a corrupted prover leaks the secret witness of the protocol (before* `can_abort` $\leftarrow 0$*), this simulation is trivial and the distributions are equal.*

The inability to simulate this protocol in the standard model whenever $|k| < |w|$ is inherent to the structure of the scheme: When the auditor is corrupt we need to output to the environment a message E which is indistinguishable from an encryption of the witness, however since the prover is honest only $|w|$ is leaked, hence E must be uncorrelated with w. Later we must output k to the environment st. $\mathsf{Dec}(k, E) = w$ (except with negligible probability), however this implies communication at rates greater than channel capacity: since E is uncorrelated with the message w it could be sampled the receiver directly, then w is transmitted by sending k.

In practical terms this means that the auditor can obtain an encryption of the witness and then abort the protocol without paying. We note that the prior works mentioned earlier (would) also require such non-commiting encryption to achieve simulation security. This is due to the similarity between all these scheme of exchanging a decryption key which enables decryption of the witness, which has been encrypted and exchanged 'off-chain' priorly.

6 The FastSwap Protocol

6.1 Protocol

The protocol is parameterized by a timeout Δ_{action}. The judge maintains a timer D_{action}, when D_{action} expires the judge outputs the current value of the *result* variable on the action port as the output of the protocol[6]. To simplify the description we assume that the transition function of the judge is sent to the judge functionality by both players at the start of the protocol and that upon receiving $\perp$ the honest party aborts the protocol. This allows us to treat the judge as a third party in the protocol.

The overall idea of FastSwap is to have both parties agree on a commitment of the initial state, with both parties knowing the opening of the commitment. In case of contingent payments the auditor/buyer would then deposit funds at the judge. Subsequently the prover reveals the environment by sending it directly to the judge, at this point a unique[7] execution trace is now defined by the environment and the initial state inside the commitment. In the honest case, where the trace is accepting, the auditor simply lets the timer D_{action} expire, after which the action is assumed complete:

FastSwap : Honest Execution

- Auditor:
  - Sample $R' \xleftarrow{\$} \mathcal{R}_\kappa$.
  - Compute $S_1' \leftarrow \mathsf{Initial}(I')$.
  - Compute $T_1' \leftarrow \mathsf{Tag}(S_1')$.
  - Compute $C' \leftarrow \mathsf{Comm}(R', T_1')$.
  - Send R' to the prover.
  - Send C' to the judge.
- Judge
  - Receive C' from the auditor.
  - Set $result \leftarrow 0$.
  - Start D_{action} with timeout Δ_{action}.
- Prover:
  - Receive R from the prover.
  - Compute $S_1 \leftarrow \mathsf{Initial}(I)$.
  - Compute $T_1 \leftarrow \mathsf{Tag}(S_1)$.
  - Compute $C \leftarrow \mathsf{Comm}(R, T_1)$.
  - If $C \neq C'$ (from the judge) abort the protocol.
  - Send e to the judge.
- Judge
  - Receive e from the prover.
  - Set $result \leftarrow 1$.
  - Reset D_{action} with timeout Δ_{action}.

[6] In blockchain applications for contingency payments, the judge contract can be converted into a wallet contract after the expiry of D_{action} where *result* denotes which party is allowed to withdraw the funds.

[7] By 'unique', we mean that neither party can break the binding property of the commitment scheme and Tag function, hence can only posses one such trace. Since the state is significantly larger than the commitments it is clearly not unique in the strict sense.

- Auditor:
  - Compute $m \leftarrow \mathsf{Terminate}(e, S'_1)$.
  - Compute $S'_m \leftarrow \mathsf{StepN}(e, S'_1, m)$.
  - If $S'_m = \langle Accept \rangle$ terminate the protocol.
    Otherwise proceed to dispute resolution (see below).

The intuition for the dispute resolution protocol is to maintain two pointers l and r into the computation trace of the prover. The pointer l will always point to a computation step that both parties agree on (initially S_1, the state inside the commitment). The pointer r (when defined), will point to a computation step where $S_r \neq S'_r$. We then search for the greatest value of l and the smallest value of r, by using an interactive binary search mediated by the judge to ensure message delivery. Eventually $r - l = 1$ and the prover uses the authenticated computation structure scheme to show correct transition from S_l to S_r, with a succinct proof:

FastSwap : Dispute Resolution

- Auditor:
  - Send $\langle Dispute \rangle$ to the judge.
- Judge:
  - Set $result \leftarrow 0$
  - Set $l \leftarrow 1, r \leftarrow \perp$. Define $m = (r - l)/2$ (initially $m = \perp$).
  - Reset D_{action} with timeout Δ_{action}
- While $r = \perp$ or $r - l > 1$:
  - Prover:
    - If $r = \perp$ (first iteration):
      - Compute $n \leftarrow \mathsf{Terminate}(e, S_1)$.
      - Locally set $r \leftarrow n$.
      - Send n to the judge.
    - Compute $S_m \leftarrow \mathsf{StepN}(e, S_1, m)$[a].
    - Compute $T_m \leftarrow \mathsf{Tag}(S_m)$.
    - Send T_m to the judge.
  - Judge:
    - If $r = \perp$ (first iteration), set $r \leftarrow n$.
    - Store T_m.
    - Set $result \leftarrow 1$
    - Reset D_{action} with timeout Δ_{action}
  - Auditor:
    - Compute $S'_m \leftarrow \mathsf{StepN}(e, S'_1, m)$
    - If $T_m = \mathsf{Tag}(S'_m)$, send $\langle Left \rangle$ to the judge.
    - Otherwise, send $\langle Right \rangle$ to the judge.
  - Judge:
    - If received $\langle Left \rangle$, set $l \leftarrow m$, set $T_l \leftarrow T_m$.

- $*$ If received $\langle Right \rangle$, set $r \leftarrow m$, set $T_r \leftarrow T_m$.
- $*$ Set $result \leftarrow 0$
- $*$ Reset D_{action} with timeout Δ_{action}

– Prover:
  - Compute $S_l \leftarrow \mathsf{StepN}(e, S_1, l)$.
  - Compute $\pi_{l \mapsto r} \leftarrow \mathsf{Prove}(e, S_l)$.
  - If $l = 1$, send $(\pi_{l \mapsto r}, R, T_1)$ to the judge.
    Otherwise send $\pi_{l \mapsto r}$ to the judge.

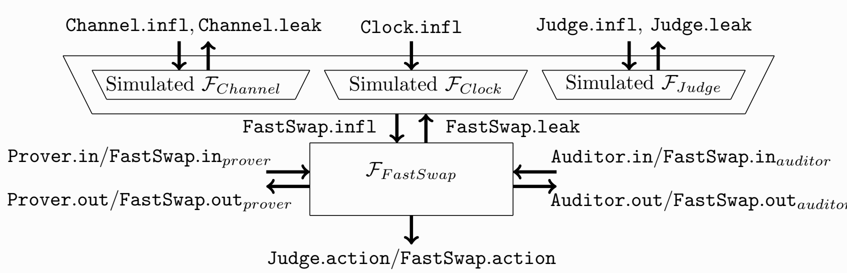
– Judge:
  - Set $result \leftarrow 1$.
  - If $l = 1$ and $\mathsf{Open}(R, T_1, C') = 0$, set $result \leftarrow 0$.
  - If $r = n$ and $T_r \neq \mathsf{Tag}(\langle Accept \rangle)$, set $result \leftarrow 0$
  - If $\mathsf{Verify}(e, T_l, T_r, \pi_{l \mapsto r}) = 0$, set $result \leftarrow 0$.

[a] Note that m is defined at this point.

The dispute resolution protocol additionally guarantees that if the output is 1, the auditor is corrupted, if the output is 0, the prover must be corrupted. This allows the judge to optionally trigger penal action towards the dishonest party (e.g. in a smart contract environment, this might be seizing collateral added to the contract during the start of the protocol) in the cases where the dispute resolution protocol is triggered.

6.2 Simulator

Simulator $\mathcal{S}_{FastSwap}$: simulate $\Pi_{FastSwap}$ using $\mathcal{F}_{FastSwap}$



`Case 1.` Neither party is corrupted:

– On input $\langle Input \rangle$ on `FastSwap.leak`:
  - Sample $R' \xleftarrow{\$} \mathcal{R}_\kappa$.
  - Compute $C' \leftarrow \mathsf{Comm}(R', \epsilon)$.
  - Simulate output R' on $\texttt{Channel.in}_{auditor}$.
  - Simulate output C' on $\texttt{Judge.in}_{auditor}$.

– On input e on `FastSwap.leak`:

  - Wait for simulated input R on $\texttt{Channel.out}_{prover}$.
  - Simulate output e on $\texttt{Judge.in}_{prover}$.
- On expiry of D_{action} (inside judge simulation):
  - Output $\langle \mathit{Action} \rangle$ on `FastSwap.infl`.
  - Output 1 on `Judge.leak`.

`Case 2.` Auditor is corrupted, Prover is honest:

- On input I' (auditors initial state) on `FastSwap.leak`:
  - Sample $R' \xleftarrow{\$} \mathcal{R}_\kappa$.
  - Compute $S'_1 \leftarrow \mathsf{Initial}(I')$.
  - Compute $T'_1 \leftarrow \mathsf{Tag}(S'_1)$.
  - Compute $C' \leftarrow \mathsf{Comm}(R', T'_1)$.
  - Simulate output R' on $\texttt{Channel.in}_{auditor}$.
  - Simulate output C' on $\texttt{Judge.in}_{auditor}$.
- On $e, S_{FastSwap}$ on `FastSwap.leak` ($S_{FastSwap}$ is the leaked state):
  - Store $S_{FastSwap}$.
  - Simulate output e on $\texttt{Judge.in}_{prover}$.
- On $\langle \mathit{Dispute} \rangle$ on `Judge.leak`.
  - Simulate the dispute resolution protocol using $S_{FastSwap}$, by observing the messages from the corrupted auditor using `Judge.leak` and simulating the messages on $\texttt{Judge.in}_{prover}$ of the honest prover according to the dispute resolution protocol.
- On expiry of D_{action}:
  - Output $\langle \mathit{Action} \rangle$ on `FastSwap.infl`.
  - Obtain res on `FastSwap.leak`: output res on `Judge.leak`.

`Case 3.` Auditor is honest, Prover is corrupted:

Due to $\mathcal{F}_{FastSwap}$ leaking the auditors initial state when the prover is corrupted the simulation is very similar to the case of a corrupted auditor:

- On input I' (auditors initial state) on `FastSwap.leak`:
  - Sample $R \xleftarrow{\$} \mathcal{R}_\kappa$.
  - Compute $S'_1 \leftarrow \mathsf{Initial}(I')$.
  - Compute $T'_1 \leftarrow \mathsf{Tag}(S'_1)$.
  - Compute $C' \leftarrow \mathsf{Comm}(R, T'_1)$.
  - Simulate output R on $\texttt{Channel.in}_{prover}$.
  - Simulate output C on $\texttt{Judge.in}_{auditor}$.
- On $e, S_{FastSwap}$ on `FastSwap.leak` ($S_{FastSwap}$ is the leaked state):
  - Store $S_{FastSwap}$.
  - Output e on `Judge.leak`.
- On $\langle \mathit{Dispute} \rangle$ on `Judge.leak`.
  - Simulate the dispute resolution protocol using $S_{FastSwap}$, by observing the messages from the corrupted auditor using `Judge.leak` and simulating the messages on $\texttt{Judge.in}_{prover}$ of the honest auditor according to the dispute resolution protocol.
- On expiry of D_{action}:
  - Output $\langle \mathit{Action} \rangle$ on `FastSwap.infl`.
  - Obtain res on `FastSwap.leak`: output res on `Judge.leak`.

Lemma 2 **($\Pi_{FastSwap} \Diamond \mathcal{F}_{Judge} \Diamond \mathcal{F}_{Channel} \Diamond \mathcal{F}_{Clock} \geq_{comp} \mathcal{F}_{FastSwap}$).** *$\Pi_{FastSwap}$ implements $\mathcal{F}_{FastSwap}$ using $\mathcal{F}_{Judge}$, $\mathcal{F}_{Channel}$ and $\mathcal{F}_{Clock}$ with respect to all computationally bounded (PPT) environments.*

Proof. *By case analysis on the corruption pattern:*

`Case 1`. ***Neither party is corrupted:***

The prover posses a valid witness and $\langle Dispute \rangle$ is not sent to the judge by the auditor. Hence the leakage in the real execution is comprised solely of the leakage in the honest execution part of the protocol. The output on `FastSwap.action` is always 1, if neither party aborts and the output 1 on `Judge.leak` is consistent with the final value of result outputted on `Judge.action` in the real execution.

`Case 2`. ***Auditor is corrupted, Prover is honest:***

The leakage from the simulation of the honest part of the protocol has exactly the same distribution as the real protocol. We therefore focuses on the simulation of the dispute resolution (recall that we obtain the entire state of $\mathcal{F}_{FastSwap}$):

Since the prover is honest it follows that $\langle Accept \rangle = \mathsf{StepN}(e, S_1, n)$ where $n \leftarrow \mathsf{Terminate}(e, S_1)$. Except with negligible probability $S_1 = S_1'$ by computational integrity of the authenticated computation scheme (Definition 10) and binding of the commitment scheme (Definition 6). We claim an invariant of the loop in the protocol:

$$l < r \leq n \text{ and } T_l = \mathsf{Tag}(S_l) \text{ and } T_r = \mathsf{Tag}(S_r)$$

This is immediately obvious from inspection of the dispute protocol. Upon termination of the loop $r - l = 1$ and $\mathsf{Verify}(e, T_l, T_r, \pi_{l \mapsto r}) = 1$ with probability 1 (by completeness of the authenticated computation scheme), furthermore whenever $r = n$, we have $S_r = \langle Accept \rangle$ hence $T_r = \mathsf{Tag}(\langle Accept \rangle)$ also with probability 1. Therefore the simulated judge always outputs 1, which is consistent with `FastSwap.action`.

`Case 3`. ***Auditor is honest, Prover is corrupted:***

The leakage from the simulation of the honest part of the protocol has exactly the same distribution as the real protocol. We therefore focuses on the simulation of the dispute resolution (recall that we obtain the entire state of $\mathcal{F}_{FastSwap}$):

Since the auditor is honest it follows that $\langle Accept \rangle \neq \mathsf{StepN}(e, S_1', m)$ where $m \leftarrow \mathsf{Terminate}(e, S_1')$, hence the judge should output 0. We first establishes an invariant of the loop in the protocol: $T_l = \mathsf{Tag}(S_l')$ and at least one of the following holds:

- $\Diamond$ $l < r \leq n$ *and* $T_r \neq \mathsf{Tag}(S_r')$
- $\Diamond$ $l < r = n$ *and* $T_r \neq \mathsf{Tag}(\langle Accept \rangle)$

The invariant holds initially where $r = n$ and $l = 1$, since $T_1 = \mathsf{Tag}(S'_1)$ is established during the honest part of the protocol and $\forall i \in [1, n] : S'_i \neq \langle Accept \rangle$ (otherwise $\langle Accept \rangle = \mathsf{StepN}(e, S'_1, m)$ as well). During the protocol the corrupted auditor provides T_w with $l < w < r$ and the invariant is maintained:

- *If $T_w = \mathsf{Tag}(S'_w)$, then $l \leftarrow w$.*
  Hence $T_l = \mathsf{Tag}(S'_l)$ is maintained and r, T_r is unchanged.

- *If $T_w \neq \mathsf{Tag}(S'_w)$, then $r \leftarrow w$.*
  Hence $T_r \neq \mathsf{Tag}(S'_r)$ is established and l, T_l is unchanged.

Upon termination of the loop: $r - l = 1$, $T_l = \mathsf{Tag}(S'_l)$ and:

- *If $r = n$ and $T_r \neq \mathsf{Tag}(\langle Accept \rangle)$, the output is always 0.*
- *If $T_r \neq \mathsf{Tag}(S'_r)$, then $\mathsf{Verify}(e, T_l, T_r, \pi_{l \mapsto r}) = 0$ except with only negligible probability, by computational integrity (Definition 10) of the authenticated computation scheme. Hence the output is 0.*

7 Instantiation of FastSwap

In this section we propose a simple 'Ethereum-like' instantiation of the FastSwap protocol, based on an authenticated Patricia trie over a sparse memory space. The state is a tuple $(pc, I, \mathcal{R}_{reg}, S)$ consisting of:

- An instruction pointer $pc \in \mathbb{N}_+$ pointing to a cell.
- An optional word-sized instruction I (which might be ϵ).
- A register bank $\mathcal{R}_{reg}$ containing word-sized registers $r_1, \ldots, r_n$.
- An authenticated data structure S over a memory space of M words.

The memory space is provided by simply ameliorating a Patricia trie[8] with a superimposed Merkle tree (see e.g. [6] appendix D for details), which allows proving memory lookups by providing at most $2 \cdot \log(M)$ hashes of size κ, where M is the size of the memory space. We let $\mathsf{Prove}_{Patricia}, \mathsf{Verify}_{Patricia}, \mathsf{Apply}_{Patricia}$ be the associated algorithms of the authenticated Patricia trie. We let $\mathcal{R}_{reg}[r_i]$ denote the looking up the value of the register r_i and $\mathcal{R}_{reg}[r_i \leftarrow v]$ denote a new register bank, where the value v is assigned to the register r_i.

Tag function. We define $\mathsf{Tag}((pc, I, \mathcal{R}_{reg}, S)) = \mathsf{CRH}((\mathsf{Tag}_{Patricia}(S), pc, I, \mathcal{R}_{reg}))$. Meaning the full register bank, Merkle root, current instruction and program counter is provided during the verification.

[8] Radix tree with a radix of 2.

Step function. For efficiency and simplicity reasons the instantiation limits the number of operations on the memory space during every step to at most one, this is done by using a '2-cycle' register machine, where every instruction in the instruction set takes two applications of Step to execute. The Step function operates as follows, with the state being matched occurring on the left:

$$\begin{aligned}
\mathsf{Step}(e, (pc, \epsilon, \mathcal{R}_{reg}, S)) &:= (pc, I, \mathcal{R}_{reg}, S) \\
&\quad \text{where } (*, I, *) \leftarrow \mathsf{Apply}_{Patricia}(S, load(pc)) \\
\mathsf{Step}(e, (pc, load(i,j), \mathcal{R}_{reg}, S)) &:= (pc+1, \epsilon, \mathcal{R}_{reg}[r_i \leftarrow M], S) \\
&\quad \text{where } (*, M, *) \leftarrow \mathsf{Apply}_{Patricia}(S, load(\mathcal{R}_{reg}[r_j])) \\
\mathsf{Step}(e, (pc, store(i,j), \mathcal{R}_{reg}, S)) &:= (pc+1, \epsilon, \mathcal{R}_{reg}, S') \\
&\quad \text{where } (S', *, *) \leftarrow \mathsf{Apply}_{Patricia}(S, store(\mathcal{R}_{reg}[r_i], \mathcal{R}_{reg}[r_j])) \\
\mathsf{Step}(e, (pc, jump(i,j), \mathcal{R}_{reg}, S)) &:= (\Delta, \epsilon, \mathcal{R}_{reg}, S) \\
&\quad \text{where if } \mathcal{R}_{reg}[r_j] > 0 \text{ then } \Delta = \mathcal{R}_{reg}[r_i] \text{ else } \Delta = (pc+1) \\
\mathsf{Step}(e, (pc, mult(i,j), \mathcal{R}_{reg}, S)) &:= (pc+1, \epsilon, \mathcal{R}_{reg}[r_i \leftarrow a \cdot b], S) \\
&\quad \text{where } a = \mathcal{R}_{reg}[r_i], b = \mathcal{R}_{reg}[r_j] \\
\mathsf{Step}(e, (pc, add(i,j), \mathcal{R}_{reg}, S)) &:= (pc+1, \epsilon, \mathcal{R}_{reg}[r_i \leftarrow a + b], S) \\
&\quad \text{where } a = \mathcal{R}_{reg}[r_i], b = \mathcal{R}_{reg}[r_j] \\
\mathsf{Step}(e, (pc, env(i), \mathcal{R}_{reg}, S)) &:= (pc+1, \epsilon, \mathcal{R}_{reg}[r_i \leftarrow e], S)
\end{aligned}$$

Additionally there are two predefined values of pc corresponding to an accepting and a rejecting state. If either of these addresses are reached, Step replaces the state with some predefined canonical $\langle Accept \rangle$ or $\langle Reject \rangle$ state not otherwise reachable, regardless of the contents of the register bank or memory space:

$$\begin{aligned}
\mathsf{Step}(e, (pc_{accept}, \epsilon, \mathcal{R}_{reg}, S)) &:= \langle Accept \rangle \\
\mathsf{Step}(e, (pc_{reject}, \epsilon, \mathcal{R}_{reg}, S)) &:= \langle Reject \rangle
\end{aligned}$$

The Step function can always be made complete by mapping any non-conforming state to $\langle Reject \rangle$.

Prove function. The prove function outputs the register bank and a proof for the authenticated Patricia trie in case of a memory operation:

$$\begin{aligned}
\mathsf{Prove}(e, (pc, \epsilon, \mathcal{R}_{reg}, S)) &:= (\epsilon, pc, \mathcal{R}_{reg}, \mathsf{Tag}_{Patricia}(S), I, \pi_{lookup}) \\
&\quad \text{where } (*, I, \pi_{lookup}) \leftarrow \mathsf{Apply}_{Patricia}(S, lookup(pc)) \\
\\
\mathsf{Prove}(e, (pc, load(i,j), \mathcal{R}_{reg}, S)) &:= (load(i,j), pc, \mathcal{R}_{reg}, \mathsf{Tag}_{Patricia}(S), R, \pi_{lookup}) \\
&\quad \text{where } (*, R, \pi_{lookup}) \leftarrow \mathsf{Apply}_{Patricia}(S, lookup(\mathcal{R}_{reg}[r_j])) \\
\\
\mathsf{Prove}(e, (pc, store(i,j), \mathcal{R}_{reg}, S)) &:= (store(i,j), pc, \mathcal{R}_{reg}, \mathsf{Tag}_{Patricia}(S), \mathsf{Tag}_{Patricia}(S'), \pi_{store}) \\
&\quad \text{where } (S', *, \pi_{store}) \leftarrow \mathsf{Apply}_{Patricia}(S, store(\mathcal{R}_{reg}[r_i], \mathcal{R}_{reg}[r_j]))
\end{aligned}$$

$$\begin{aligned}
\mathsf{Prove}(e, (pc, jump(i,j), \mathcal{R}_{reg}, S)) &:= (jump(i,j), pc, \mathcal{R}_{reg}, \mathsf{Tag}_{Patricia}(S)) \\
\mathsf{Prove}(e, (pc, mult(i,j), \mathcal{R}_{reg}, S)) &:= (mult(i,j), pc, \mathcal{R}_{reg}, \mathsf{Tag}_{Patricia}(S)) \\
\mathsf{Prove}(e, (pc, add(i,j), \mathcal{R}_{reg}, S)) &:= (add(i,j), pc, \mathcal{R}_{reg}, \mathsf{Tag}_{Patricia}(S)) \\
\mathsf{Prove}(e, (pc, env(i), \mathcal{R}_{reg}, S)) &:= (env(i), pc, \mathcal{R}_{reg}, \mathsf{Tag}_{Patricia}(S))
\end{aligned}$$

Verify function. The verify function follows the approach of computing the resulting tag from the proof directly. Then verifies that the proof corresponds to the current tag and that the new tag is equal to the one provided:

$$\begin{aligned}
\mathsf{Verify}(e, T, T', \pi) := T = T_{before} \wedge T' = T_{after} \wedge \mathsf{Validate}(e, \pi) = 1 \\
\text{where } T_{after} \leftarrow \mathsf{TagAfter}(e, \pi), T_{before} \leftarrow \mathsf{TagBefore}(e, \pi)
\end{aligned}$$

With TagBefore extracting the 'previous' tag from the proof:

$$\begin{aligned}
&\mathsf{TagBefore}(e, (\epsilon, pc, \mathcal{R}_{reg}, T, I, \pi_{lookup})) := \mathsf{CRH}((T, pc, \epsilon, \mathcal{R}_{reg})) \\
&\mathsf{TagBefore}(e, (load(i,j), pc, \mathcal{R}_{reg}, T, R, \pi_{lookup})) := \mathsf{CRH}((T, pc, load(i,j), \mathcal{R}_{reg})) \\
&\mathsf{TagBefore}(e, (store(i,j), pc, \mathcal{R}_{reg}, T, T', \pi_{store})) := \mathsf{CRH}((T, pc, store(i,j), \mathcal{R}_{reg})) \\
&\mathsf{TagBefore}(e, (jump(i,j), pc, \mathcal{R}_{reg}, T)) := \mathsf{CRH}((T, pc, jump(i,j), \mathcal{R}_{reg})) \\
&\mathsf{TagBefore}(e, (mult(i,j), pc, \mathcal{R}_{reg}, T)) := \mathsf{CRH}((T, pc, mult(i,j), \mathcal{R}_{reg})) \\
&\mathsf{TagBefore}(e, (add(i,j), pc, \mathcal{R}_{reg}, T)) := \mathsf{CRH}((T, pc, add(i,j), \mathcal{R}_{reg})) \\
&\mathsf{TagBefore}(e, (env(i), pc, \mathcal{R}_{reg}, T)) := \mathsf{CRH}((T, pc, env(i), \mathcal{R}_{reg}))
\end{aligned}$$

With TagAfter extracting the 'resulting' tag from the proof, by simulating the step function using the data provided in the proof string:

$$\begin{aligned}
&\mathsf{TagAfter}(e, (\epsilon, pc, \mathcal{R}_{reg}, T, I, \pi_{lookup})) := \mathsf{CRH}((T, pc, I, \mathcal{R}_{reg})) \\
&\mathsf{TagAfter}(e, (load(i,j), pc, \mathcal{R}_{reg}, T, R, \pi_{lookup})) := \mathsf{CRH}((T, pc+1, \epsilon, \mathcal{R}_{reg}[r_i \leftarrow R])) \\
&\mathsf{TagAfter}(e, (store(i,j), pc, \mathcal{R}_{reg}, T, T', \pi_{store})) := \mathsf{CRH}((T', pc+1, \epsilon, \mathcal{R}_{reg})) \\
&\mathsf{TagAfter}(e, (jump(i,j), pc, \mathcal{R}_{reg}, T, \pi_{store})) := \\
&\qquad \mathsf{CRH}((T, \text{ if } \mathcal{R}_{reg}[r_j] > 0 \text{ then } \mathcal{R}_{reg}[r_i] \text{ else } pc+1, \epsilon, \mathcal{R}_{reg})) \\
&\mathsf{TagAfter}(e, (mult(i,j), pc, \mathcal{R}_{reg}, T, \pi_{store})) := \mathsf{CRH}((T, pc+1, \epsilon, \mathcal{R}_{reg}[r_i \leftarrow r_i \cdot r_j])) \\
&\mathsf{TagAfter}(e, (add(i,j), pc, \mathcal{R}_{reg}, T, \pi_{store})) := \mathsf{CRH}((T, pc+1, \epsilon, \mathcal{R}_{reg}[r_i \leftarrow r_i + r_j])) \\
&\mathsf{TagAfter}(e, (env(i), pc, \mathcal{R}_{reg}, T, \pi_{store})) := \mathsf{CRH}((T, pc+1, \epsilon, \mathcal{R}_{reg}[r_i \leftarrow e]))
\end{aligned}$$

With Validate : $\mathcal{E} \times \mathcal{P}_\kappa \to \{1, 0\}$ validating the memory operations by applying the verification of the authenticated data structure used to emulate a large memory space:

$$\mathsf{Validate}(e, (\epsilon, pc, \mathcal{R}_{reg}, T, I, \pi_{lookup})) := \mathsf{Verify}_{Patricia}(T, T, lookup(pc), I, \pi_{lookup})$$

$\mathsf{Validate}(e, (load(i,j), pc, \mathcal{R}_{reg}, T, R, \pi_{lookup})) := \mathsf{Verify}_{Patricia}(T, T, lookup(\mathcal{R}_{reg}[r_j]), R, \pi_{lookup})$
$\mathsf{Validate}(e, (store(i,j), pc, \mathcal{R}_{reg}, T, T', \pi_{store})) := \mathsf{Verify}_{Patricia}(T, T', store(\mathcal{R}_{reg}[r_i], \mathcal{R}_{reg}[r_j]), \epsilon, \pi_{store})$
$\mathsf{Validate}(e, *) := 1;$ For all other operations.

8 Concrete Efficiency Considerations

In this section we cover a few simple optimizations which are of less theoretical interest, but can improve the concrete efficiency of FastSwap greatly. This section is aimed at potential implementors.

Reusing the judge. Often deploying the code of a smart contract has significant cost of its own. However, note that the functionality of the judge does not depend on the predicate, but only on the authenticated computation structure scheme. Hence the code can reused between swaps or separated into a library which can be shared my multiple independent contracts.

High-level execution language. Rather than applying the Step function of the authenticated computation structure directly the prover and auditor can execute a more efficient higher level language where each instruction decomposes into a sequence of simpler low-level instructions from authenticated computation structure scheme. In case of a dispute the offending high-level instruction must be unpacked into its lower-level instructions and dispute resolution carried out at the lower layer. For instance this enables the use of hardware acceleration for cryptographic primitives in the high-level language while using a function call to a naive implementation in the low-level language.

'Just-In-Time' authenticated data structures. Rather than apply operations directly to the authenticated data structure used in the authenticated computation structure, concrete efficiency can often be gained by representing the data more efficiently during applications of the Step function and only ameliorate the data structure with the authentication data at states revealed during dispute resolution. An example of this is executing a higher level language, where each instruction corresponds to a long sequence of instructions on the For instance, when the memory space is represented as a Patricia trie, then when calling during Prove and Tag a Merkle tree is temporarily imposed over the data structure. This enables application of authenticated data structures which would otherwise inhibit concrete efficiency, e.g. RSA or CDH based vector commitments [5], which would only have to be computed over logarithmically many snapshots of the vector representing the memory space in case of dispute, rather than updated at every step of the computation during the honest execution.

Reduce computational complexity during dispute resolution. Rather than naively recomputing S_w from S_1 during dispute resolution, resulting in $n \log n$ computation steps, this can easily be reduced to n steps, by simply storing the state S_l corresponding to the left pointer and computing S_w from S_l whenever $l < w$ and from S_1 otherwise.

Efficient language. Language designers are likely to want a language close to the that of the underlaying smart contract language in which the verifier is implemented. This is due to the verifier essentially being an interpreter for the source language, the size of which is directly proportional to the cost of deploying the judge contract. Additionally high-level instructions of the underlaying smart contract language (like signature verification and cryptographic hash function evaluation) can be provided in the source language. Application of such high-level functions might greatly simply the implementation of the decryption of the witness inside the predicate.

Send multiple tags during dispute resolution. The number of rounds during dispute resolution can be reduced by a constant $\log_2 c$, by having the prover send 2 tags $T_{w_1}, \ldots, T_{w_c}$, then having the auditor send the index of the last match l and first mismatch r. For a computation of 2^{30} steps, letting $c = 2^5$, this reduces the number of interactions with the judge during dispute from 62 to 14.

Limit storage in the judge contract. The previous optimization introduces a significantly increased storage requirement on the judge (e.g. 32 hashes stored every iteration during dispute resolution). Some smart contract execution environments, in particular the Ethereum virtual machine, sets the price of storage very high (20000 'gas' per 256 bits[6][9]), compared to the price of memory (e.g. call arguments) or computation. In particular the cost of:

- Sending 32 words of 256 bits to the contract is $\approx$ 100 gas[6].
- Computing a Merkle tree over 32 words of 256 bits is $\approx$ 3000 gas[6][10].
- Storing 32 words is 640000 gas[6].

Hence it is significantly cheaper[11] to have the judge compute a Merkle tree over the arguments (tags) and store the root. Then having the auditor prove a path to the (at most) two leafs which corresponds to updated l and r values. This is possible because every input to the judge, not only its current state, is public and therefore available to the auditor.

9 Further Research

9.1 Constructions of authenticated computation structures.

Unlike authenticated data structures where a proof must prove the correct execution of a full operation, the proofs for authenticated computation structures need only prove a single step of computation which can be arbitrarily small. In some cases this might enable significantly more efficienct proofs than those for authenticated data structures under the same cryptographic assumptions:

[9] Of which 15000 can be recouped by later clearing the memory.
[10] Using 64 invocations of the `SHA3` instruction.
[11] Our estimates for $c = 2^5$ is a 80 - 90 % 'gas' saving

In Section 7, we have described a concrete instantiation wherein the map lookup is a single instruction in the language. For our concrete instantiation this results in proofs of size $\log M$, with M being the size of the memory space. Alternatively low level operations for walking the authenticated data structure can be provided by the language and smaller atomic steps in the lookup can be proved instead. As a simple example consider lookups (*load* instructions) in the authenticated Patricia trie of Section 7, but where the state additionally contains a cryptographic hash digest for an 'authenticated' node inside the Patricia tree. Hence the proof becomes an instance of:

– An instruction pointer *pc*.
– An instruction I (which might be ϵ)
– A finite number of fixed-sized registers $r_1, \ldots, r_n$.
– A tag for an authenticated Patricia tree T.
– A node pointer H_{node}.

Whenever $I \neq load(i, j)$, the verifier operates as in Section 7. Whenever $I = load(i, j)$ and $H_{\text{node}} = \epsilon$, the verifier checks that $H_{\text{node}} \leftarrow T$ in the subsequent tag. Whenever $I = load(i, j)$ and $H_{\text{node}} \neq \epsilon$, the proof additionally consists of a node in the Patricia tree, $Node(prefix, len, H_{\text{left}}, H_{\text{right}})$, and the verifier checks that $H_{\text{node}} = \mathsf{CRH}(Node(prefix, len, H_{\text{left}}, H_{\text{right}}))$ and that $H_{\text{node}} \leftarrow H_{\text{left}}$ or $H_{\text{node}} \leftarrow H_{\text{right}}$ in the subsequent tag, depending on whether the lookup in the tree progresses left/right based on $r_j[len]$. When the leaf is reached, verify it similarly, set $I \leftarrow \epsilon$, set $H_{\text{node}} \leftarrow \epsilon$. For updates, where the new hash is propagated up though the tree, a similar process must be repeated in the opposite direction, then $T \leftarrow H_{\text{node}}$ at the leaf. Using this approach, the proof size can be made constant in M while the number of rounds during dispute grows by at most $\log \log M$ times.

References

1. Author, Stefan Dziembowski, Author, Lisa Eckey, Author, Sebastian Faust. FairSwap: How To Fairly Exchange Digital Goods. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Series: CCS '18. ACM, New York (2018), https://doi.org/10.1145/3243734.3243857.
2. Author, Matteo Campanelli, Author, Rosario Gennaro, Author, Steven Goldfeder, Author, Luca Nizzardo. Zero-Knowledge Contingent Payments Revisited: Attacks and Payments for Services. https://doi.org/10.1145/3133956.3134060
3. Author, Henning Pagnia, Author, Felix C. Gartner. On the impossibility of fair exchange without a trusted third party. 1999.
4. BitcoinWiki, Zero Knowledge Contingent Payment, 2016, `https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment`
5. Author, Dario Catalano, Author, Dario Fiore. Vector Commitments and Their Applications. Public-Key Cryptography – PKC 2013 https://doi.org/10.1007/978-3-642-36362-7_5
6. Author, Gavin Wood, Title, Ethereum: A secure decentralized generalized transaction ledger, 2018-08-16 (version e7515a3).
7. Author, Jesper Buus Nielsen. Separating Random Oracle Proofs from Complexity Theoretic Proofs: The Non-committing Encryption Case. Advances in Cryptology — CRYPTO 2002. https://doi.org/10.1007/3-540-45708-9_8
8. Author, Ronald Cramer, Author, Ivan Bjerre Damgård, Author, Jesper Buus Nielsen. Secure Multiparty Computation and Secret Sharing (1st edition). **ISBN-13**: 978-1107043053.
9. Author, Wacław Banasik, Author, Stefan Dziembowski, Author, Daniel Malinowski. Title, Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts. Computer Security – ESORICS 2016. ESORICS 2016. Lecture Notes in Computer Science, vol 9879. Springer. https://doi.org/10.1007/978-3-319-45741-3_14
10. Author, Georg Fuchsbauer. Title, WI Is Not Enough: Zero-Knowledge Contingent (Service) Payments Revisited. CCS '17 Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Pages 229-243. https://doi.org/10.1145/3133956.3134060