Contingent Payments

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

Aarhus University, mathias@hall-andersen.dk

This document serves as an introduction and extended motivation, the goal is to make intelligible and motivate the remaining article for people with an MSc in computer science. Therefore this document serves as a brief introduction to the problem statement, the notion of cryptographic assumptions, a crash course on the universal composabilitity framework and the setting of FastSwap.

1 Introduction

1.1 Prisoners Dilemma

Consider the experience of buying/selling license keys over the internet:

Seller: Imagine you are running a shop which sells license keys to a wide selection of commercial software. A customer buys a license key for a piece of office software, he pays using Visa, you send him the license key. However a few days later he disputes the payment, saying that the key did not work. You know this to be false.

Buyers: Now suppose you wish to buy a license to a piece of commercial software. You go online, find a seeming legitimate looking website and buy a license key using Bitcoin. However when you receive the key, it does not work. You try to contact the seller, but no response.

What is described above is a modern variant of the classical prisoners dilemma: whichever party moves first is at risk of being cheated: the buyer can finalize the payment (using Bitcoin) and get scammed by the seller, or the seller can finalize the sale (sending the key) and get scammed by the buyer. What largely saves people in modern society from the constant experience of being defrauded by strangers is the existence of strong judicial systems which can enforce harsh punishments that far outweighs the limited gain from cheating in the interaction: essentially by arbitrarily imposing a cost to defecting in the game. However, this approach is ill suited if:

1. The participants cannot agree on a judicial system to trust, this issue compounds if the stakes are very high, which makes corrupting the judicial systems a more viable strategy.
2. The participants are anonymous, such that no penalty can be enforced.

The latter is especially devastating since it even curtails the possibility of locally keeping a score of ‘good’ and ‘bad’ merchants/customers.

1.2 Contingent Payments

In this thesis we, in particular, suggest a practical scheme for solving these kinds of 'Contingent Payments' where payment occurs if and only if the input (e.g. serial key) satisfies a predicate (e.g. serial key check algorithm). Further examples of such problems might be:

- Buying proofs. The predicate consists of a formal proof checker which checks every application of derivation rules in a proof provided by the seller.

- Out sourcing computation. Where the buyer encodes the properties of a solution to some large problem that he wishes to have solved, e.g. solving some resource allocation problem.

- Cross-chain swaps / interaction. Where the seller exchanges a signed transaction on one blockchain for monetary compensation on another. This can potentially be generalized to interactions with complex smart contracts across chains by including the smart contract code in the predicate.

- Trustless bug-bounties. Where the buyer compiles a vulnerable program e.g. a file-parser, into a contingent payment predicate and wishes to buy a file such that the seller clearly illustrates control of the program counter (e.g. by setting it to a magical value).

- File transfer. Where the buyer wishes to obtain a file with a particular hash. This is the particular use case considered in the FairSwap paper.

In general any kind of digital good that can be defined by a computable program. The goal of thesis has been to devise a scheme that would be practical for all the aforementioned applications and many more.

1.3 Cryptographic Assumptions

We will make wide application of cryptographic primitives and ultimately reduce security of the FastSwap protocol to cryptographic assumptions. As an example of both cryptographic primitives and cryptographic assumptions, consider a simple and widely applied primitive: authenticated encryption (Definition 1).

Definition 1 (Authenticated Encryption). *An authenticated encryption scheme consists of a family indexed by a security parameter κ of pairs of two PPT (polynomial in κ) algorithms:*

- $\mathsf{Seal} : \mathcal{K}_\kappa \times \mathcal{M} \to \mathcal{C}$*, encrypts a plaintext yielding a ciphertext.*
- $\mathsf{Open} : \mathcal{K}_\kappa \times \mathcal{C} \to \mathcal{M} \cup \{\bot\}$*, decrypts and authenticates the ciphertext, yielding either $\bot$ or a decrypted message.*

Satisfying perfect completeness for every message.

$$\forall m \in \mathcal{M} : 1 = \mathbb{P}_{k \xleftarrow{\$} \{0,1\}}[\mathsf{Open}(k, \mathsf{Seal}(k, m)) = m] \tag{1}$$

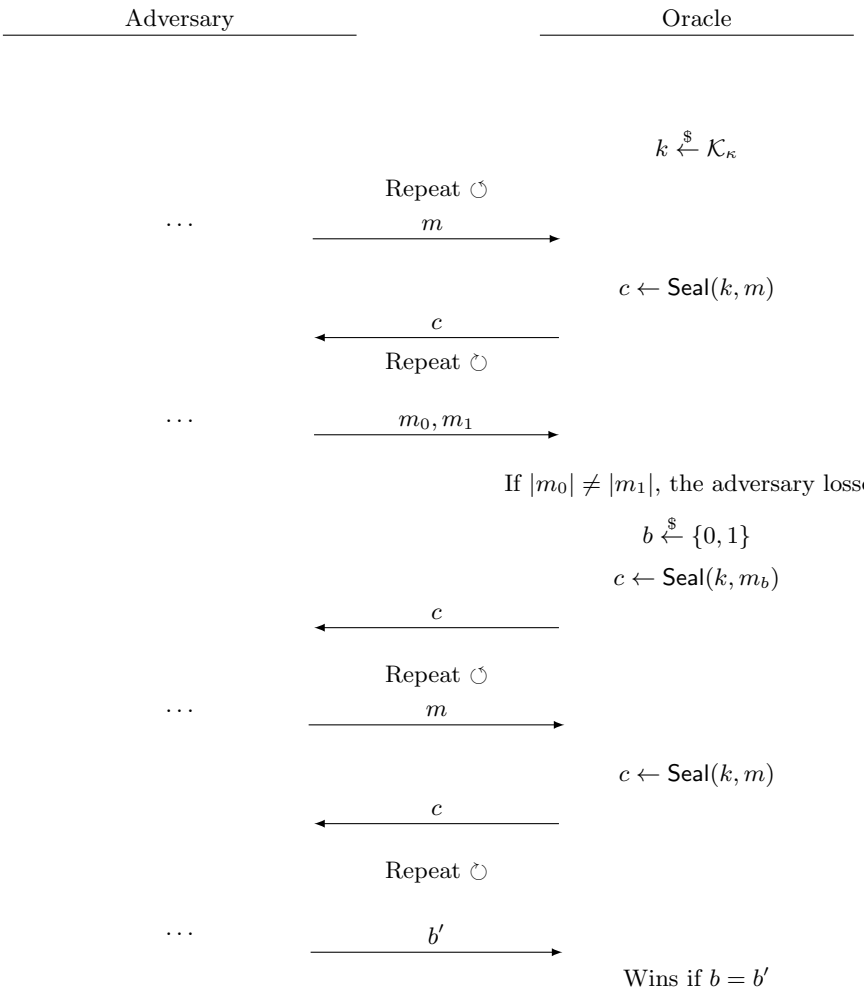


Fig. 1. IND-CPA Game

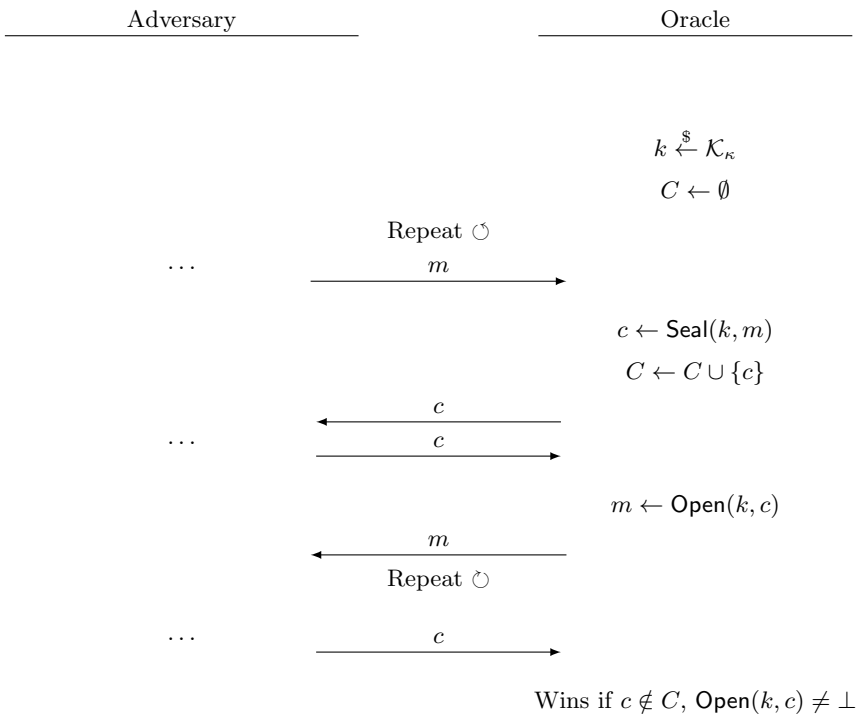


Fig. 2. SUF-CMA Game

The previous definition (Definition 1) is not concerned with security, in particular it does not rule out 'clearly insecure', but perfectly complete instantiations like $\mathsf{Seal}(k, m) = m$. In order to define what 'secure' means for primitives, we formulate security games (see Figure 1, Figure 2) played by an 'adversary' against an 'oracle', 'security' then means that the winning probability for any polynomial adversary is bounded by some expression: usually reflecting that the adversary only does negligibly better than the naive strategy. For IND-CPA[1] security, he require that ciphertexts are indistinguishable to the adversary, even when he is allowed to encrypt arbitrary message:

Definition 2 (IND-CPA Security). *An authenticated encryption scheme is said to be IND-CPA if for every PPT adversary $\mathcal{A}$ playing the IND-CPA game (Figure 1), there exists a negligible function $\mathsf{negl}(\kappa)$ such that the probability of $\mathcal{A}$ winning the IND-CPA game is bounded by $1/2 + \mathsf{negl}(\kappa)$.*

Note in particular that it implies that encryption cannot be deterministic, observe also that in Figure 1 the game can only be won if $|m_0| = |m_1|$, reflecting that IND-CPA encryption does not hide the length of messages. For SUF-CMA[2] security we intuitively require that the adversary can only create valid ciphertexts by querying the oracle:

Definition 3 (SUF-CMA Security). *An authenticated encryption scheme is said to be SUF-CMA if for every PPT adversary $\mathcal{A}$ playing the SUF-CMA game (Figure 2), there exists a negligible function $\mathsf{negl}(\kappa)$ such that the probability of $\mathcal{A}$ winning the SUF-CMA game is bounded by $\mathsf{negl}(\kappa)$.*

A cryptographic assumption is a conjecture that there exists a primitive satisfying the security definition:

Conjecture 1. There exists an IND-CPA and SUF-CMA secure authenticated encryption scheme.

This is clearly an open conjecture[3], since in particular it would imply $\mathbf{P} \neq \mathbf{NP}$: For both games the oracle $\mathcal{O}$ is $\mathbf{PPT}$ if $\mathbf{P} = \mathbf{NP}$ then $\mathcal{A}$ can non-deterministically guess the random tape R of $\mathcal{O}$, then simulate $\mathcal{O}_R$ on his queries and compare the responses of the simulated oracle and the real transcript to distinguish between correct guesses. Concretely for the IND-CPA game $\mathcal{A}$ can simply guess the key in non-deterministic polynomial time and accept iff. $\mathsf{Open}(k, c) = m_0$, thus winning the IND-CPA game except with negligible probability.

1.4 Universal Composability & The Real/Ideal Paradigm

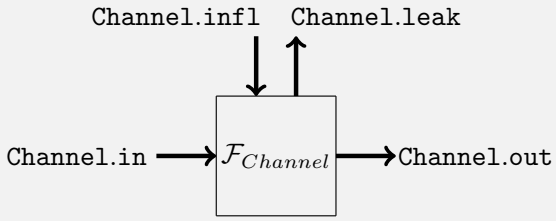
What does it mean for protocols to be 'secure'? Merely claiming that a protocol is 'secure' is a meaningless statement, for two prime reasons: what is the exact

[1] Indistinguishability under Chosen Plaintexts
[2] Strong Unforgability under Chosen Message Attack
[3] Like all cryptographic assumptions.

claimed function of the protocol? Under what conditions is the functionality ensured[4]?

The first step in formalizing security consists in formulating exact answers to these two questions. In order to formalize exactly what the function of our protocol is we define so-called 'ideal functionalities', an ideal functionality in some ways corresponds to a trusted third-party which all parties can interact directly with. The desired behavior of the protocol is defined by the ideal functionality: the ideal functionality is the definition of 'secure' and in particular does not depend on cryptographic assumptions. This is best illustrated with an example: consider a formulation of a simple ideal functionality for an in-order one-directional encrypted authenticated channel:

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



Initialize: Set $i \leftarrow 0$, $\mathbf{m} \leftarrow \emptyset$.

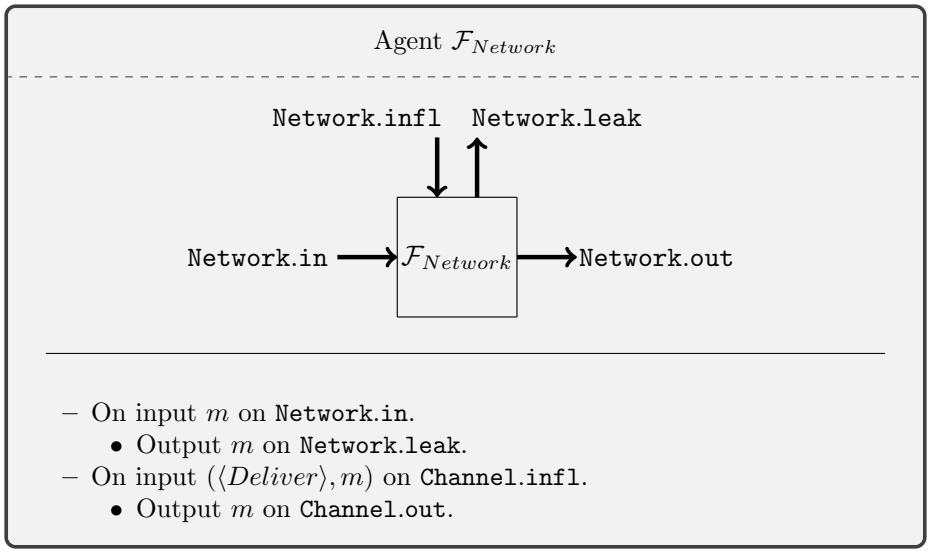
- On input m on `Channel.in`.
  - Set $i \leftarrow i + 1$.
  - Set $\mathbf{m} \leftarrow \mathbf{m} \cup \{(i, m)\}$.
  - Output $(i, |m|)$ on `Channel.leak`.
- On input $(\langle Drop \rangle, i)$ on `Channel.infl`.
  - If there exists $(i, m) \in \mathbf{m}$, update $\mathbf{m} \leftarrow \mathbf{m} \setminus \{(i, m)\}$.
- On input $(\langle Deliver \rangle, i)$ on `Channel.infl`.
  - Pick m, st. $(i, m) \in \mathbf{m}$, if no such m, ignore the message.
  - If $\exists (m', i') \in \mathbf{m}.i' < i$, ignore the message.
  - Update $\mathbf{m} \leftarrow \mathbf{m} \setminus \{(i, m)\}$.
  - Output m on `Channel.out`.

Note: The functionality does not consider corruptions, i.e. what guarantees are provided if either endpoint is compromised, at the start of the protocol ('static corruption') or later ('adaptive corruption'). Such extensions would be required to for example capture 'forward secrecy'.

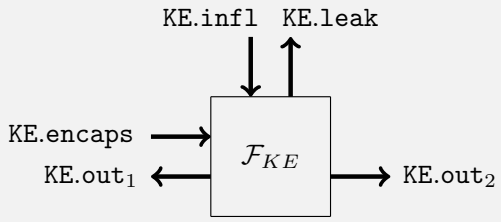
[4] e.g what kind of adversaries is it secure against?

As the names indicate `Channel.infl` and `Channel.leak` are dedicated to the 'adversary'. They reflect what the adversary can 'influence' and 'learn' respectively. Additionally the adversary can drop and deliver messages at his discretion using the influence port, this behavior corresponds roughly to the scenario where the authenticated channel allows for packets to be dropped (e.g. IPsec) and the adversary controls the network. Note that the adversary cannot inject messages, replay messages or deliver them out-of-order.

We also require ideal functionalities to model the underlaying network, as well as some means of key-establishment. The $\mathcal{F}_{Network}$ functionality reflects the adversaries total control of the network:



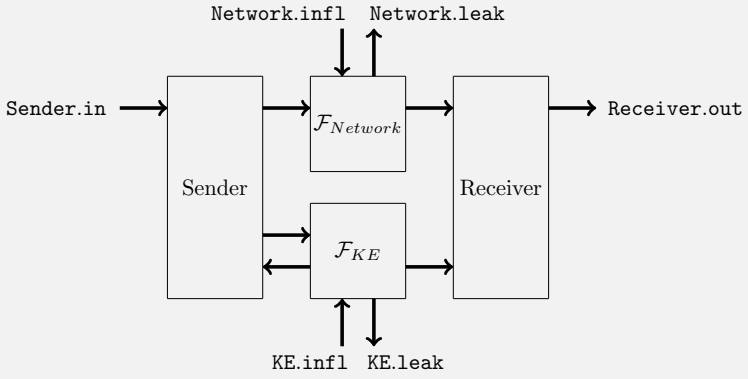
The $\mathcal{F}_{KE}$ allows the establishment of a shared secret with a length chosen by one of the parties, the 'adversary' learns only the length of the shared secret. In reality $\mathcal{F}_{KE}$ is quite hard to instantiate in practice using e.g. $\mathcal{F}_{Network}$, due to Man-in-the-middle type attacks, but for illustration purposes we will simply assume that this is given.



Initialize: Set $k \leftarrow \epsilon$.

- On input κ on `KE.encaps`.
  - Pick $k \xleftarrow{\$} \{0,1\}^\kappa$.
  - Output κ on `KE.leak`.
- On input $\langle Deliver \rangle$ on `KE.infl`.
  - If $k = \epsilon$ ignore the message.
  - Output k on `KE.out`$_1$.
  - Output k on `KE.out`$_2$.

Protocol $\Pi_{Channel}$



Initialize: Sender set $k \leftarrow \epsilon$, $i \leftarrow 0$. Receiver set $k \leftarrow \epsilon, i \leftarrow 0$.

- Sender:
  On input k' on `KE.out`$_1$.
  - Set $k \leftarrow k'$.
- Sender:
  On input m on `Sender.in`.
  - Set $i \leftarrow i + 1$.
  - If $k = \epsilon$, ignore the message.

    * Send κ on `KE.encaps`.
  - If $k \neq \epsilon$:
    * Compute $c \leftarrow \mathsf{Seal}(k, (i, m))$.
    * Output c on `Network.in`.

– Receiver:
  On input k' on `KE.out2`:
  - Set $k \leftarrow k'$.
– Receiver:
  On input x on `Network.out`:
  - $p \leftarrow \mathsf{Open}(k, x)$
  - If $p = \perp$ ignore the message.
  - Parse p as (i', p), otherwise ignore the message.
  - If $i' \leq i$ ignore the message.
  - Set $i \leftarrow i'$.
  - Output m on `Receiver.out`.

Note: for simplicity reasons $\Pi_{Channel}$ drops the messages if no key has been established. However the functionality also allows dropping of messages at will and we shall see that $\Pi_{Channel}$ still instantiates $\mathcal{F}_{Channel}$.

Security in the universal composability framework is defined by indistinguishably between the ideal world (the ideal functionality) and the real world (the protocol) to an 'environment' which is given access to all the external ports i.e. the ports not used by the protocol. However, the difference between the ideal and the real world is allowed to be 'trivial', in particular the $\Pi_{Channel}$ protocol and the $\mathcal{F}_{Channel}$ ideal functionality does not have the same set of ports, which would allow trivial distinguishing. To make precise what 'trivial' means, we introduce a simulator: an algorithm no more powerful that the environment which is given access to the ports of the functionality and can then emulate the exposed ports of the protocol. The justification is intuitively that any piece of information that the adversary ('environment') learns during interaction in the protocol, he could also obtain by interacting with the simulator and hence directly from the ideal functionality with no additional 'powers'.

There are two 'parameters' in the definition above which can be tweaked: the 'power' of the environment and the notion of indistinguishability. In this article we shall concern ourselves only with PPT environments and negligible statistical distance (computational indistinguishability): the environment is a probabilistic polynomial time Turing machine and allowed to succeed with negligible advantage, which essentially allows the use of cryptographic assumptions in the protocol. One could consider stronger classes of adversaries e.g. computationally unbounded ones, or stricter notions of indistinguishability e.g. equality between the two distributions.

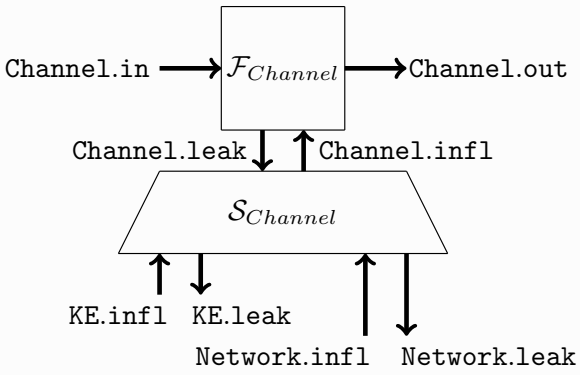
Intuitively, we can think of the $\mathcal{F}_{Network}$ and $\mathcal{F}_{KE}$ functionalities as the premises for providing $\mathcal{F}_{Channel}$ by construction using our protocol and a corresponding simulator. Note for instance: the 'weaker'[5] $\mathcal{F}_{Network}$ and $\mathcal{F}_{KE}$ are, and the 'strong'[6] the $\mathcal{F}_{Channel}$ functionality is, the harder it is to provide an instantiation of $\mathcal{F}_{Channel}$ from $\mathcal{F}_{Network}, \mathcal{F}_{KE}$ against a given set of adversaries.

We denote composition of agents by $\Diamond$, e.g. $\Pi_{Channel} \Diamond \mathcal{F}_{Network} \Diamond \mathcal{F}_{KE}$ is the $\Pi_{Channel}$ protocol where the ports of $\Pi_{Channel}$ has been matched to the corresponding ports of the functionalities. We write $\Pi_{Channel} \Diamond \mathcal{F}_{Network} \Diamond \mathcal{F}_{KE} \geq_{comp} \mathcal{F}_{Channel}$ to indicate that $\Pi_{Channel} \Diamond \mathcal{F}_{Network} \Diamond \mathcal{F}_{KE}$ is 'at least as secure' as $\mathcal{F}_{Channel}$ with respect to computationally bounded environments. More formally, to show $\Pi_{Channel} \Diamond \mathcal{F}_{Network} \Diamond \mathcal{F}_{KE} \geq_{comp} \mathcal{F}_{Channel}$ our task is:

1. Define a simulator $\mathcal{S}_{Channel}$.
2. Show for any PPT environment $\mathcal{Z}$, the distributions $\mathcal{S}_{Channel} \Diamond \mathcal{F}_{Channel} \Diamond \mathcal{Z}$ and $\Pi_{Channel} \Diamond \mathcal{F}_{Network} \Diamond \mathcal{F}_{KE} \Diamond \mathcal{Z}$ are statistically indistinguishable.

The latter is done by creating a reduction from the environment $\mathcal{Z}$ which successfully distinguishes to an adversary $\mathcal{A}$ breaking a cryptographic assumption; in a style superficially similar to Karp reductions known from complexity theory.

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



Initialize: Let $C \leftarrow [\,]$ (a map).

- On $\langle Deliver \rangle$ on `KE.infl`:
  - Set $k \xleftarrow{\$} \mathcal{K}_\kappa$.
- On $(i, |m|)$ on `Channel.leak`:
  - If $k = \epsilon$:
    - Send $(\langle Drop \rangle, i)$ on `Channel.infl`.
  - Otherwise:
    - Compute $c \leftarrow \mathsf{Seal}(k, (i, 0^{|m|}))$.

[5] Allowing more influence, leaking more state.
[6] Allowing less influence, leaking less state.

    * Set $C[c] \leftarrow i$.
    * Output c on `Network.leak`.
- On $(\langle Deliver \rangle, x)$ on `Network.infl`:
  - If $x \notin C$, ignore the message.
  - Let $i \leftarrow C[x]$.
  - For $j \in [1, i-1]$, output $(\langle Drop \rangle, j)$ on `Channel.infl`.
  - Output $(\langle Deliver \rangle, i)$ on `Channel.infl`.

Theorem 1 ($\Pi_{Channel} \lozenge \mathcal{F}_{Network} \lozenge \mathcal{F}_{KE} \geq_{comp} \mathcal{F}_{Channel}$)**.**

Proof. Using the simulator $\mathcal{S}_{Channel}$, show:

$$\mathcal{S}_{Channel} \lozenge \mathcal{F}_{Channel} \lozenge \mathcal{Z} \equiv_{stat} \Pi_{Channel} \lozenge \mathcal{F}_{Network} \lozenge \mathcal{F}_{KE} \lozenge \mathcal{Z} \qquad (2)$$

We let $\mathcal{H}_0 = \mathcal{S}_{Channel} \lozenge \mathcal{F}_{Channel} \lozenge \mathcal{Z}$ be the distribution of the original simulation and consider indistinguishable hybrids $\mathcal{H}_\ell$ for all $\ell > 0$ defined as follows:

> *Consider the sequence of hybrids $\ell > 0$, where $\mathcal{H}_\ell$ is derived from $\mathcal{H}_0$ st. the simulator 'extracts' $m_1, \ldots, m_\ell$ from $\mathcal{F}_{Channel}$ and leaks the encryptions of m_i on `Network.leak`, rather than letting $m_i = 0^{|m|}$ for $i \in [1, \ell]$ as done in $\mathcal{S}_{Channel}$. Note that $\mathcal{H}_{q(\kappa)}$ for some polynomial q is the distribution where every query made by $\mathcal{Z}$ is answered with $\mathsf{Seal}(k, m_i)$, i.e. $\forall \delta > 0 : \mathcal{H}_{q(\kappa)} = \mathcal{H}_{q(\kappa)+\delta}$. Suppose there exists an adversary that succeeds in distinguishing between $\mathcal{H}_\ell$ and $\mathcal{H}_{\ell+1}$, then there exists a PPT adversary $\mathcal{A}$ winning the IND-CPA with non-negligible probability. Construct $\mathcal{A}$ as follows: run $\mathcal{S}_{Channel} \lozenge \mathcal{F}_{Channel} \lozenge \mathcal{Z}$, whenever m_i is sent:*
> - *If $i \leq \ell$ send m_i to the encryption oracle.*
> - *If $i = \ell + 1$ send $(0^{|m_i|}, m_i)$ as the challenge to the oracle.*
> - *If $i > \ell + 1$ send $0^{|m_i|}$ to the encryption oracle.*
>
> *Clearly if $b = 0$, the distribution is $\mathcal{H}_\ell$ otherwise $\mathcal{H}_{\ell+1}$. Hence $\mathcal{A}$ wins the IND-CPA game with non-negligible advantage.*

It follows that $\mathcal{H}_\infty$ derieved from $\mathcal{S}_{Channel}$ where $c_i \leftarrow \mathsf{Seal}(k, m_i)$ always, is indistinguishable from $\mathcal{H}_0$. Lastly it remains to argue that $\mathcal{H}_\infty$ is indistinguishable from $\Pi_{Channel} \lozenge \mathcal{F}_{Network} \lozenge \mathcal{F}_{KE} \lozenge \mathcal{Z}$. Clearly the distributions of the ciphertexts c is equal (k is sampled with the same distribution), however the simulator replaces Open with the map C (crucial for the reduction to IND-CPA in the previous sequence of hybrids):

> *Suppose $\mathcal{H}_\infty \not\equiv_{stat} \Pi_{Channel} \lozenge \mathcal{F}_{Network} \lozenge \mathcal{F}_{KE} \lozenge \mathcal{Z}$, then there exists $\mathcal{A}$ winning the SUF-CMA game with non-negligible probability $p(\kappa)$. Claim: if $\mathcal{H}_\infty \not\equiv_{stat} \Pi_{Channel} \lozenge \mathcal{F}_{Network} \lozenge \mathcal{F}_{KE} \lozenge \mathcal{Z}$, then $\mathcal{Z}$ sends $(\langle Deliver \rangle, x)$ st. $\mathsf{Open}(k, x) \neq \perp$ and $x \notin C$ with probability $p(\kappa)$. Produce $\mathcal{A}$ as follows: let $q(\kappa)$ be a polynomial bound on the number of $(\langle Deliver \rangle, x)$ queries, pick $j \xleftarrow{\$} [1, q(\kappa)]$, let $i \leftarrow 0$. Run $\Pi_{Channel} \lozenge \mathcal{F}_{Network} \lozenge \mathcal{F}_{KE} \lozenge \mathcal{Z}$, whenever Seal must be evaluated, call the tagging oracle in the SUF-CMA game, when $\mathcal{Z}$ sends $(\langle Deliver \rangle, x)$ on `Network.infl`:*

- *If $i = j$, send x as the forgery to the oracle in the SUF-CMA game.*
- *If $i \neq j$ send x to the check oracle in the SUF-CMA game and obtain p.*
- *Set $i \leftarrow i + 1$.*

With probability $1/q(\kappa)$ the guess for the message index is correct, hence with non-negligible probability $p(\kappa)/q(\kappa)$ $\mathcal{A}$ wins the SUF-CMA game.

It is also possible to prove security solely in the game-based paradigm, however the universal composabilitity framework provides one central advantage, the UC theorem:

Theorem 2 (Universal Composabilitity). *Let Π_F, Π_G be protocols implementing $\mathcal{F}_F, \mathcal{F}_G$ respectively. If $\Pi_F \Diamond \mathcal{F}_G \geq_{comp} \mathcal{F}_F$ and $\Pi_G \Diamond \mathcal{F}_H \geq_{comp} \mathcal{F}_G$ then $(\Pi_F \Diamond \Pi_G) \Diamond \mathcal{F}_H \geq_{comp} \mathcal{F}_F$.*

Intuitively the UC theorem states that if a protocol (e.g. Π_F) uses an ideal functionality (e.g. $\mathcal{F}_G$), this functionality can be safely swapped for any subprotocol implementing the functionality (e.g. Π_G). Besides providing modularity, this enables us to eliminate potential issues where the interplay between protocols leads to a system that is 'insecure' overall.

1.5 Blockchains

For this thesis we will concern ourselves less with the exact details of the underlaying blockchain and merely use it as an instantiation of an ideal functionality. Essentially blockchains offer the promise of an 'honest-but-curious' party for hire: a smart contract can be created by any other participant in the protocol, its code inspected and a guarantee is provided that this simulated party will act according to the protocol specification.

Most current blockchains validate smart contract execution in time that is linear in the length of the computation triggered by the transactions. This is due to the naive method of ensuring the integrity of the state transition: every transaction is simply re-executed by every node in the network. This places a natural constraint on the amount of computation that can be ensured by the blockchain: no more than the slowest node in the network can execute. Succinct proofs of computational integrity with sub-linear verification time has only seen limited application.

Besides introducing a natural upper limit this also leads to a high price of executing code 'on-chain', since the resource is finite and users essentially engage in an auction for running time. This motivates our desire to reduce the amount of computation on-chain, both to enable scaling and to reduce costs. Additionally, smart contracts are 'honest-but-curious' which is undesirable when dealing with private data.

2 Contributions

I claim to have designed the first truly practical contingent payment protocol, FastSwap, described in the accompanying article. In particular the distinguishing features of FastSwap are:

- Concrete efficiency. FastSwap does not depend on zero-knowledge proofs which often entails significant computational overhead for the prover.

- Honest communication is optimal (the size of the program description and the witness). In particular it is independent of the length of the computation.

- Dispute communication is logarithmic in the length of the computation. The dispute resolution contract is independent of the predicate and need only be deployed once globally.

- The computational model is flexible, in particular branching RAM programs can be efficiently executed, which enables easy complication of existing languages to FastSwap predicates.

- Concrete optimizations can enable the executions of complex high-level languages, even when the language during dispute is very simple (leading to a small dispute resolution contract). This is likely to yield significant performance increases in practice for long-running predicates.

FastSwap

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

Aarhus University, mathias@hall-andersen.dk

Abstract. FastSwap enables a simple and concretely efficient contingent payments for complex predicates. FastSwap only relies on symmetric primitives (semantically secure 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. 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 efficiently express the predicate in imperative languages already familiar to them, without an expensive transformation to e.g. 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 a penalty in case of dispute.

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

1 Introduction

FastSwap chieftly enables 'Contingent Payments', wherin a buyer holds a predicate P and funds, whereas a seller holds a purported witness w for which she claims $P(w) = 1$. The goal of a contingent payment protocol is to enable the buyer to lear w and ensure payment iff. w satisfies $P(w) = 1$. Examples of potential applications of contingent payments include:

[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.

- **Buying signatures on documents.** For example paying for a certificate iff. the signature on the certificate is correct and particular fields are set.

- **Buying transactions.** Where the buyer wishes to purchase a signed transaction for some blockchain, which causes a particular smart contract (included in the predicate) to reach a given state.

- **Purchasing solutions to hard problems.** e.g. paying for the factorization of composite numbers or solutions to resource allocations problems.

- **Purchasing a file with a particular hash.** In which the buyer wishes to pay for a large file transfer iff. he receives the full file and its cryptographic hash matches a particular value.

- **Trustless bug-bounties.** Where the input (e.g. a file) must violate some safety constrains (e.g. demonstrate arbitrary control of the control flow) of a potentially vulnerable program (e.g. an image parser) to be considered a valid witness.

- **Buying formal proofs.** The predicate can contain a formal proof checker and the witness is required to be a formal proof which proves a desired statement.

We facility this by introduction a very limited 'honest-but-curious' judge, which possesses only $poly(\log)$ computation & communication in $|w|, |P|$ and n (the length of the computation $P(w)$). Furthermore we want to ensure that when both parties are honest w is not leaked to the environment. These constraints each preclude the naive protocol wherein the buyer sends the funds to the judge, the seller sends w to the judge and the judge pays the seller iff. $P(w) = 1$. The motivation for this setting is enabling the instantiation of the judge efficiently via smart contract.

Like prior works [1] [4], FastSwap enables the judge to learn the whether the exchange was successful, therefore the action of the judge upon learning the output of the predicate can be made essentially arbitrary, additionally FastSwap can easily be extended to multiple 'buyers'. Due to these features we believe it might be useful to consider the more general application of enabling 'off-chain' computation of predicate by a (potentially malicious) 'prover' under the scrutiny of one/more (potentially malicious) 'auditors', in the presence of a judge incapable of computing the predicate alone. We note that constructions like FastSwap has the ability to be significantly more efficient in practice that the application of current Zero-Knowledge proof systems to elevate the problem. Hence we frame FastSwap in this more general setting.

1.1 Prior Work

Zero-Knowledge Contingent Payments (ZKCP). The Zero-Knowledge Contingent Payment (ZKCP) construction [4] (by Gregory Maxwell) requires a zero-knowledge proof system capable over the relation induced by 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 for a predicate P to a 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'[2].

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.

[2] 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.

Comparison. Let $|w|$ be the size of the witness, let $|P|$ be the size of the predicate description, let n be the length of the computation of $P(w)$. We compare the complexity and computational models of the two prior works to FastSwap in Figure 1 and Figure 2. Note that ZKCP does not have a 'dispute resolution' phase. FastSwap reduces the communication complexity compared to FairSwap, additionally FastSwap provides more freedom in the choice of computational model, in particular allowing efficient execution of branching RAM machines, which enables relatively easy and efficient compilation of existing imperative smart contract languages.

Name	Computational Model	Comm. (off-chain)	Comm. (on-chain)
ZKCP (zk-SNARK)	Arithmetic Circuit	$\Theta(\|P\| + \|w\|)$	$\Theta(1)$, 2 rounds
FairSwap	Computational Circuit	$\Theta(\|P\| + \|w\| + n)$	$\Theta(\|P\|)$, 2 rounds
FastSwap	RAM Machines	$\Theta(\|P\| + \|w\|)$	$O(1)$, 2 rounds

Fig. 1. Complexity of honest execution.

Name	Communication (on-chain)	Rounds (on-chain)
ZKCP (zk-SNARK)	–	–
FairSwap	$\Theta(\log n)$	$\Theta(1)$
FastSwap	$\Theta(\log n)$	$\Theta(\log n)$

Fig. 2. Complexity of dispute resolution.

Recall that generic transformation of a program in the RAM model running in n steps, requires a circuit of size $n^3 \log n$. Hence efficient compilation of e.g. existing smart contract language to FairSwap (or e.g. ZKCP with zk-SNARKs) predicates is unlikely, while this is one of the envisioned applications of FastSwap. One heuristic argument (without rigors game theoretic backing), as to why we believe the dispute resolution complexity is less crucial than the honest execution for many real-world applications is that both systems, FairSwap and FastSwap, allows the judge to discern which party is malicious. Hence a penalty can be enforced by having both parties deposit collateral with the judge prior to the swap, which can be seized / send to the honest party in case of malicious behavior.

1.2 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 (One-Time Semantic Security). *A family of symmetric encryption schemes (Definition 1) is said to be one-time semantically secure if for all pairs of PPT algorithms $(\mathcal{A}_1, \mathcal{A}_2)$, there exists a negligible function negl st.*

$$1/2 + \mathsf{negl}(\kappa) \geq \mathbb{P}[b' = b \wedge |m_1| = |m_2| : (m_1, m_2) \leftarrow \mathcal{A}_1(1^\kappa), \\ b \xleftarrow{\$} \{0,1\}, k \xleftarrow{\$} \mathcal{K}_\kappa, b' \leftarrow \mathcal{A}_2(1^k, \mathsf{Enc}(k, m_b))]$$

Note that unlike the ordinary IND-CPA definition, we do not require the encryption scheme to be indistinguishable across multiple encryption queries. In particular $\mathsf{Enc} : \mathcal{K}_\kappa \times \mathcal{M} \to \mathcal{C}$ can be deterministic.

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 $\mathcal{A}$, there exists a negligible function negl st.*

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

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 $\mathcal{A}$ there exists a negligible function negl st.*

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

Definition 7 (Computationally Hiding Commitment). *A commitment scheme (Definition 5) is said to be computationally hiding if for all pairs of PPT algorithms* $(\mathcal{A}_1, \mathcal{A}_2)$ *there exists a negligible function* negl *st.*

$$1/2 + \mathsf{negl}(\kappa) \geq \mathbb{P}[b' = b : (m_1, m_2) \leftarrow \mathcal{A}_1(1^\kappa), \\ b \xleftarrow{\$} \{0,1\}, r \xleftarrow{\$} \mathcal{R}_\kappa, b' \leftarrow \mathcal{A}_2(1^k, \mathsf{Comm}(r, m_b))]$$

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, optionally yielding a result. Produce a proof of correct application of the operation, which can be verified using only the tags of the original 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.*

Satisfying perfect completeness:

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

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:

$$\begin{aligned}\forall e \in \mathcal{E}, S \in \mathcal{S} : &\mathsf{Verify}(e, T, T', \pi) = 1 \text{ where} \\ &S' \leftarrow \mathsf{Step}(e, S), \pi \leftarrow \mathsf{Prove}(e, S), \\ &T \leftarrow \mathsf{Tag}(S), T' \leftarrow \mathsf{Tag}(S')\end{aligned}$$

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

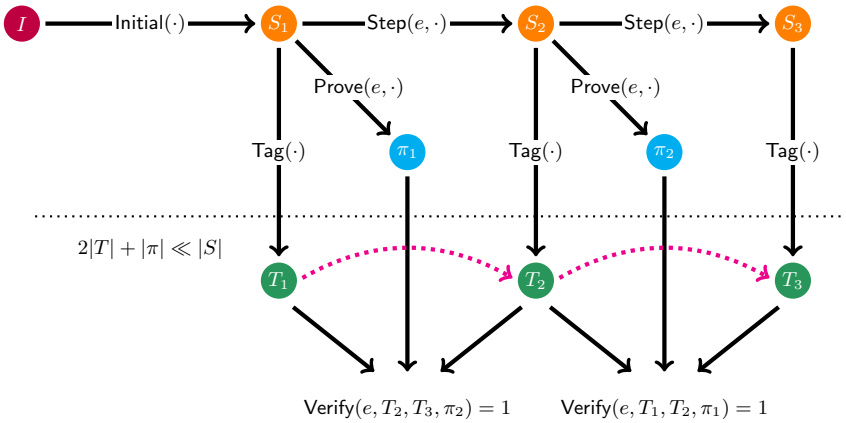


Fig. 3. Relation between the algorithms for auth. computation structures

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.

Definition 10 (Computational Integrity). *An authenticated computation structure scheme is said to provide computational integrity, if* $\mathsf{Tag} : \mathcal{S} \to \mathcal{T}_\kappa$ *is a collision resistant hash (Defintion 3) and for every PPT algorithm* $\mathcal{A}$*, there exists a negligible function* negl *such that:*

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

i.e. it is intractable to compute a tag and a proof that passes verification, except when the tag corresponds to the successor state.

For later convience we define some simple functions which are derieved from any authenticated computational structure scheme:

Definition 11 (Terminate : $\mathcal{E} \times \mathcal{S} \to \mathbb{N}_+$). *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:*

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

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

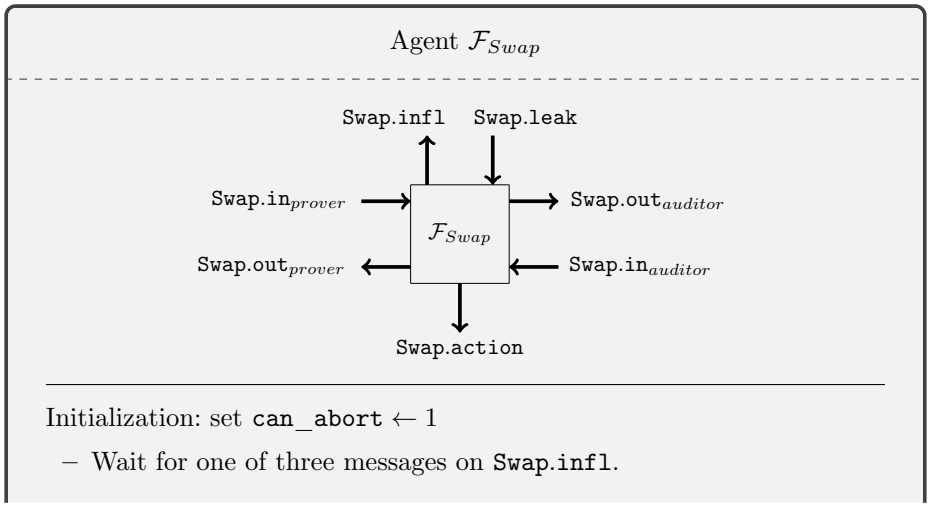
Definition 12 (StepN : $\mathcal{E} \times \mathcal{S} \times \mathbb{N}_+ \to \mathcal{S}$). *StepN applies Step a specified number of times and returns the resulting state. Formally, with the patterns being matched by preference from top to bottom:*

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

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:



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.

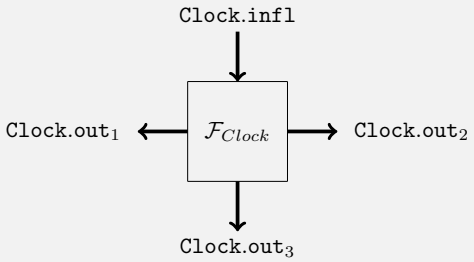
  Ignore any subsequent corruption messages.
- Any time, on input $\langle Abort \rangle$ on `Swap.infl`, $\texttt{Swap.in}_{auditor}$ or $\texttt{Swap.in}_{prover}$ and if $\texttt{can\_abort} = 1$, then abort the protocol:
  - Output $\perp$ on $\texttt{Swap.out}_{prover}$.
  - Output $\perp$ on $\texttt{Swap.out}_{auditor}$.
  - 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 $\texttt{can\_abort} \leftarrow 0$.
  - 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 $\texttt{can\_abort} = 0$:
  - 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 $\texttt{can\_abort} = 0$ 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 $\texttt{can\_abort} = 0$ 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 $\texttt{can\_abort} = 1$ 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 $M_{prover} \leftarrow \epsilon, M_{auditor} \leftarrow \epsilon$.

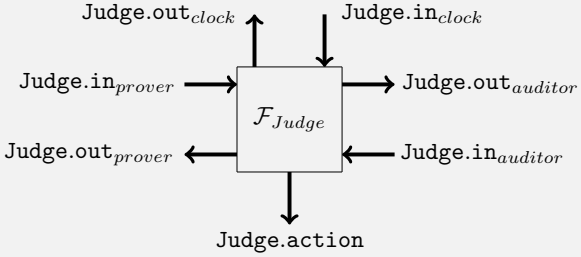
- Wait for one of three messages on `Channel.infl`
  - $(\langle Corrupt \rangle, p) : p \in \{\langle Prover \rangle, \langle Auditor \rangle\}$: Mark the party p as corrupted and allow control of the ports of p as follows:
    - By ignoring any message on $\texttt{Channel.in}_p$.

    - ∗ Whenever a message of the form $(\langle Send \rangle, m)$ is received on $\texttt{Channel.infl}$ act as if m was received on $\texttt{Channel.in}_p$.
    - ∗ Whenever a message of the form $(\langle Recv \rangle, m)$ is received on $\texttt{Channel.infl}$ output m on $\texttt{Channel.out}_p$.
    - ∗ Whenever a message m is output on $\texttt{Channel.out}_p$ output m on $\texttt{Channel.leak}$ 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 $\texttt{Channel.leak}$.
- On input m on $\texttt{Channel.in}_{auditor}$:
  - Push m to the back of M_{prover}
  - Output $(\langle Prover \rangle, |m|)$ on $\texttt{Channel.leak}$.
- On input $(\langle Deliver \rangle, p)$ on $\texttt{Channel.infl}$:
  - 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 $\mathcal{F}_{Judge}$

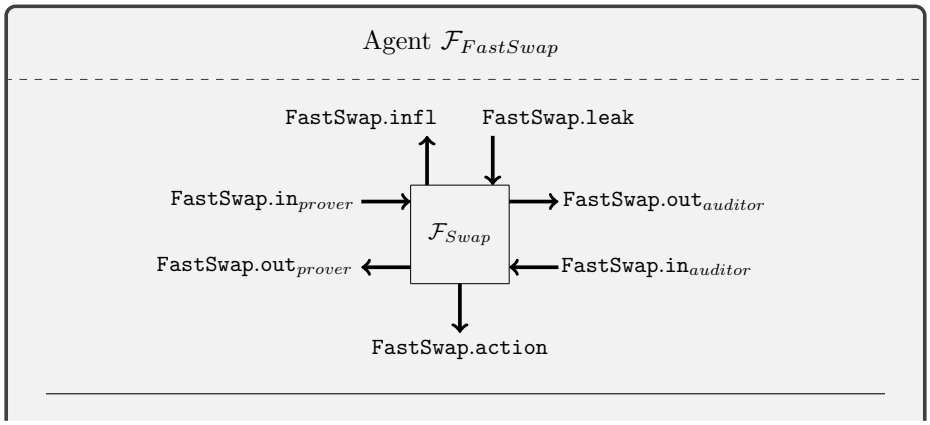


- Wait for one of three messages:
  - ◇ $\langle Auditor \rangle$: Mark the prover as corrupted, by ignoring any message on $\texttt{Judge.in}_{auditor}$. Whenever a message of the form $(\langle Send \rangle, m)$ is received on $\texttt{Judge.infl}$ act as if m was received on $\texttt{Judge.in}_{auditor}$.

  - ◇ $\langle Prover \rangle$: 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:



Initialization: set $\texttt{can\_abort} \leftarrow 1$

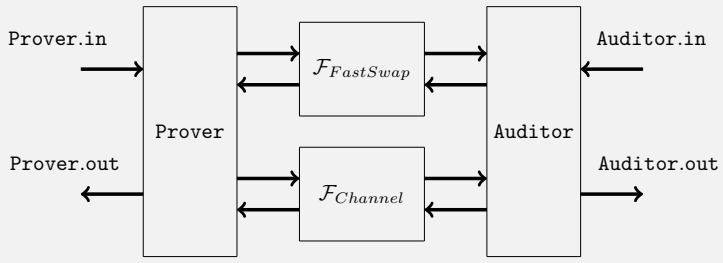
- Wait for one of three messages on $\texttt{FastSwap.infl}$.
  - $\diamond$ $\langle Auditor \rangle$: Mark the prover as corrupted, by ignoring any message on $\texttt{FastSwap.in}_{auditor}$. Whenever a message of the form $(\langle Send \rangle, m)$ is received on $\texttt{FastSwap.infl}$ act as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

  Ignore any subsequent corruption messages.
- Any time, on input $\langle Abort \rangle$ on $\texttt{FastSwap.infl}$, $\texttt{FastSwap.in}_{auditor}$ or $\texttt{FastSwap.in}_{prover}$ and if $\texttt{can\_abort} = 1$, then abort the protocol:
  - Output $\perp$ on $\texttt{FastSwap.out}_{prover}$.
  - Output $\perp$ on $\texttt{FastSwap.out}_{auditor}$.
  - Output $\perp$ on $\texttt{FastSwap.leak}$.
  - 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 $\texttt{FastSwap.leak}$.
  - Output $\langle Input \rangle$ on $\texttt{FastSwap.leak}$
- 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 $\texttt{can\_abort} \leftarrow 0$
  - Output e on $\texttt{FastSwap.out}_{auditor}$.
  - Output e on $\texttt{FastSwap.leak}$.
  - If either party is corrupted leak the entire state of the functionality on $\texttt{FastSwap.leak}$: every message sent and received by the functionality.
- On input $\langle Action \rangle$ on $\texttt{FastSwap.infl}$, when $\texttt{can\_abort} = 0$:
  - Compute $n \leftarrow \mathsf{Terminate}(e, S)$.
  - Output $\mathsf{StepN}(e, S, n) \stackrel{?}{=} \langle Accept \rangle$ on $\texttt{FastSwap.action}$ and $\texttt{FastSwap.leak}$.

We implement the $\mathcal{F}_{Swap}$ functionality using: $\mathcal{F}_{FastSwap}$, $\mathcal{F}_{Channel}$, a semantically secure encryption scheme (Definition 2)[3] 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) 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 $\texttt{FastSwap.out}_{auditor}$: Output $\perp$ on `Auditor.out`.
- Any time on input $\langle Abort \rangle$ on `Auditor.in`: Output $\langle Abort \rangle$ on $\texttt{FastSwap.in}_{auditor}$.

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 $\texttt{Channel.in}_{auditor}$

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

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

- Output P' on `Prover.out`.

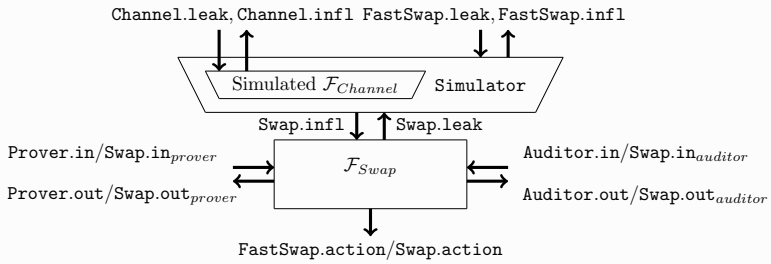
– Prover. On input w on `Prover.in`:
- Sample $k \xleftarrow{\$} \mathcal{K}_\kappa$.
- Compute $E \leftarrow \mathsf{Enc}(k, w)$.
- Define $I = (P' \circ \mathsf{Dec}, E)$.
- Output E on $\texttt{Channel.in}_{prover}$.
- Output I on $\texttt{FastSwap.in}_{prover}$.

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

– 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 without random oracles, 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 $\langle Input \rangle$ 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 $\langle Action \rangle$ on `FastSwap.infl`:
  - Output $\langle Action \rangle$ 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 $\langle Input \rangle$ 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 $\langle Action \rangle$ on `FastSwap.infl`:
  - Output $\langle Action \rangle$ on `Swap.infl`.

Lemma 1 ($\Pi_{Swap} \Diamond \mathcal{F}_{FastSwap} \Diamond \mathcal{F}_{Channel} \geq_{comp} \mathcal{F}_{Swap}$)**.** *$\Pi_{Swap}$ 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`. Neither party is corrupted:

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 $\mathsf{Enc} : \mathcal{K}_\kappa \times \mathcal{M} \rightarrow \mathcal{C}_\kappa$ is a CPA secure encryption scheme (Definition 2).

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

We assume that $\mathsf{Enc}, \mathsf{Dec}$ is non-committing 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 $\mathsf{Enc} : \mathcal{K}_\kappa \times \mathcal{M} \rightarrow \mathcal{C}_\kappa$ is a CPA secure encryption scheme (Definition 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 symmetric encryption, e.g. encrypting with a one-time pad, 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`. Auditor is honest, Prover is corrupted:

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-committing 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[4]. 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[5] 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.

[4] 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.

[5] 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.

- Set $result \leftarrow 0$.
- Start D_{action} with timeout Δ_{action}.

– Prover:
- Receive R from the auditor.
- 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}.

– 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 $result \leftarrow 0$.

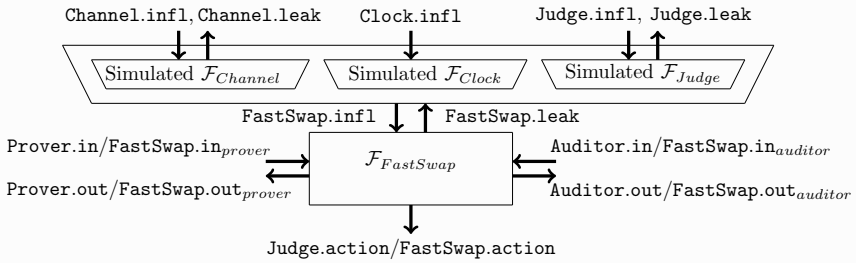
[a] Note that m is defined at this point.

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 Security Proof

We simulate $\Pi_{FastSwap}$ using $\mathcal{F}_{FastSwap}$ as follows:

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 $\langle Action \rangle$ 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 $\langle Dispute \rangle$ 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 $\langle Action \rangle$ 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 $\langle Dispute \rangle$ 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 $\langle Action \rangle$ on `FastSwap.infl`.
  - Obtain res on `FastSwap.leak`: output res on `Judge.leak`.

Lemma 2 ($\Pi_{FastSwap} \lozenge \mathcal{F}_{Judge} \lozenge \mathcal{F}_{Channel} \lozenge \mathcal{F}_{Clock} \geq_{comp} \mathcal{F}_{FastSwap}$)**.** *$\Pi_{FastSwap}$ 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 $\mathcal{F}_{FastSwap}$), in particular that the leakage is consistent with the output on `FastSwap.action`.

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 $\mathcal{F}_{FastSwap}$*), in particular that the leakage is consistent with the output on* `FastSwap.action`.

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:*

- $\lozenge$ $l < r \leq n$ *and* $T_r \neq \mathsf{Tag}(S'_r)$
- $\lozenge$ $l < r = n$ *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[6] 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.

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:

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

[6] Radix tree with a radix of 2.

$$\mathsf{Step}(e, (pc, env(i), \mathcal{R}_{reg}, S)) := (pc + 1, \epsilon, \mathcal{R}_{reg}[r_i \leftarrow e], S)$$

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:

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

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:

$$\begin{aligned}\mathsf{Prove}(e, (pc, \epsilon, \mathcal{R}_{reg}, S)) := (\epsilon, pc, \mathcal{R}_{reg}, \mathsf{Tag}_{Patricia}(S), I, \pi_{lookup}) \\ \text{where } (*, I, \pi_{lookup}) \leftarrow \mathsf{Apply}_{Patricia}(S, lookup(pc))\end{aligned}$$

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

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

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

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 $\mathsf{TagBefore}$ extracting the 'previous' tag from the proof:

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

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

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

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

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

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

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][7]), 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][8].
- Storing 32 words is 640000 gas[6].

Hence it is significantly cheaper[9] 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:

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}.

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

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