# Summary of protocol

The FluidSwaps protocol is a multi-chain system, allowing swaps between Cardano, Bitcoin and Ethereum tokens. There are a few elements of blockchain systems that FluidSwaps utilizes to allow this. Namely, the ability for cryptographic signatures to be verified across chains, and atomic swap implementations, which uses time-lock systems together with signature schemes to allow secure swaps.

FluidSwaps also employs an idea they call "smart wallets" to allow Ethereum and Bitcoin private keys to sign Cardano transactions, effectively allowing Ethereum and Bitcoin wallets to own Cardano assets. Together with allowing swaps between the chains, the protocol effectively brings the ecosystems together.

This audit is intended to review whether or not the smart contracts implemented in FluidSwaps effectively achieves its goals in a secure manner. While care has been taken to cover security issues, there is no guarantees that all security problems are uncovered.

## Review of Atomic Swap implementation

The Bitcoin script and Solidity smart contracts that are used are both fairly straight forward, and appear to be correctly implemented. While this audit ended up covering these two contracts, the reader should be made aware that the auditor's expertise is primarily in Cardano and its script system.

## Finding - 1 - SEVERE

The function `validate_tx_end` appears to be implemented incorrectly, the upper bound of the `transaction_validity_range` is asserted to be less than the deadline, this allows any transaction created BEFORE the deadline to be cancelled, which is opposite of the expected behaviour. Swaps should only be allowed to be cancelled AFTER the deadline.

**Relevant code:**

```
 Cancel -> {
   expect validate_tx_end(self, deadline) == True
   list.has(self.extra_signatories, owner_ada_pubkey_hash)
}

pub fn validate_tx_end(tx: Transaction, deadline: Int) {
  trace cbor.diagnostic(deadline)
  when tx.validity_range.upper_bound.bound_type is {
    Finite(end) -> end < deadline
    _ -> False
  }
}
```

**Recommended action**

We should be checking that the transaction is created AFTER the deadline. In which case, checking the transaction's upper bound is actually incorrect. The script should instead be checking for the transaction's lower bound, ensuring that this lower bound is larger than the deadline. This ensure's that the transaction cannot be valid until AFTER the deadline (the transation only begins to be valid AFTER the deadline).

```
 Cancel -> {
   expect validate_tx_end(self, deadline) == True
   list.has(self.extra_signatories, owner_ada_pubkey_hash)
}

pub fn validate_tx_end(tx: Transaction, deadline: Int) {
  trace cbor.diagnostic(deadline)
  when tx.validity_range.lower.bound_type is {
    Finite(begin) -> begin > deadline
    _ -> False
  }
}
```

## Review of `global.ak`, "Smart Accounts"

The "Smart Account" scripts are used such that Bitcoin and Ethereum wallets can sign specific messages that are validated on the Cardano chain. These messages are instructions for the Cardano chain, as an example, "send x number of y tokens to z address" would be signed by the Bitcoin/Ethereum wallet, then the resulting signature would be used to approve a particular transaction on the Cardano blockchain. This effectively allows Bitcoin/Ethereum users to own any Cardano assets within these scripts.

## Finding - 2 - INFORMATIONAL

These "Smart Accounts" have a large amount of logic, but seemingly no test cases can be found within the repository, with such a large amount of logic, it is difficult to reason about the security of the scripts.

**Recommended action**

Add suitable testing for these scripts. Some basic testing with generated signatures at least would be recommended.

## Finding - 3 - INFORMATIONAL

The code within `global.ak` are quite inefficient, a large portion of it is fairly redundant. It appears that there are a large number of separate fields that are signed by the Bitcoin/Ethereum wallets, and the fields are validated by reconstructing what is assumed to be the user's intention.

However, the fields `policy, assetname, amount, spendingscriptflag, pubkey, stakingscriptflag, stakekey, datumflag, datum, payment_index` are all used to simply reconstruct a singular output at `payment_index` that fulfills all of these constraints. It is completely possible to make the user sign a single cbor encoded `Output` to encompass this entire logic. The only thing that would make this a little more difficult is the `min_utxo_value` required in this `Output`, but this can be accounted for by not checking the `lovelace` value of the `Output`.

**Relevant code:**

```
 Send {
   signature,
   utxos,
   policy,
   assetname,
   amount,
   policyInput,
   assetnameInput,
   amountInput,
   spendingscriptflag,
   pubkey,
   stakingscriptflag,
   stakekey,
   datumflag,
   datum,
   payment_index,
   fee,
   signer,
 } -> {
   let list_utxos =
     list.foldr(
       utxos,
       [],
       fn(utxo, xs) {
         [
           utxo.transaction_id,
           bytearray.from_int_big_endian(utxo.output_index, 1),
           ..xs
         ]
       },
     )
   expect datum_converted: ByteArray = datum
   let message_array: List<ByteArray> =
     list.push([], bytearray.from_int_big_endian(fee, 20))
       |> list.push(bytearray.from_int_big_endian(payment_index, 1))
       |> list.push(datum_converted)
       |> list.push(bytearray.from_int_big_endian(datumflag, 1))
       |> list.push(stakekey)
       |> list.push(bytearray.from_int_big_endian(stakingscriptflag, 1))
       |> list.push(pubkey)
       |> list.push(bytearray.from_int_big_endian(spendingscriptflag, 1))
       |> list.push(bytearray.from_int_big_endian(amountInput, 20))
       |> list.push(assetnameInput)
       |> list.push(policyInput)
       |> list.push(bytearray.from_int_big_endian(amount, 20))
       |> list.push(assetname)
       |> list.push(policy)

   let concat_array = concat_array(list.concat(list_utxos, message_array))
   let message_hash_btc = get_hash_from_bytearray_btc(concat_array)
   let message_hash_eth = get_hash_from_bytearray_eth(concat_array)
   //TODO da checkare
   //verify I am sending the right amount to the right user from the data i have in the first output
   //verify the second output goes to the starting one - the amount it had - 1 ADA
   //address validation receiver is missing
   let sending_value = from_asset(policy, assetname, amount)
```

```
  let sending_value = from_asset(policy, assetname, amount)
  let incoming_value =
    if amountInput > 0 {
      from_asset(policy, assetname, amount)
    } else {
      zero
    }

  expect Some(contract_input) = find_input(self.inputs, own_ref)
  expect Some(payment_output) = list.at(self.outputs, payment_index)

  let starting_value =
    find_inputs_value(self.inputs, contract_input.output.address)
  let final_value =
    find_outputs_value(self.outputs, contract_input.output.address)

  let flag_utxo_present = list.has(utxos, own_ref)

  let basic_validation = and {
      flag_utxo_present,
      or {
        validate_signature_key(pubKey, message_hash_btc, signature),
        validate_signature_key(pubKey, message_hash_eth, signature),
      },
    }
  let expected_datum = calculate_datum(datumflag, datum)
  let expected_address =
    calculate_address(
      spendingscriptflag,
      stakingscriptflag,
      pubkey,
      stakekey,
    )
  and {
    quantity_of(payment_output.value, policy, assetname) == quantity_of(
      sending_value,
      policy,
      assetname,
    ),
    assets.match(
      final_value,
      merge(
        merge(
          merge(starting_value, negate(sending_value)),
          incoming_value,
        ),
        negate(from_lovelace(fee)),
      ),
      >=,
    ),
    basic_validation,
    payment_output.address == expected_address,
    payment_output.datum == expected_datum,
    all_inputs_present(utxos, self.inputs),
    only_external_signer_inputs_extra(utxos, self.inputs, signer),
  }
}
```

#### Recommended action

It is possible to reduce the amount of code significantly, and also increase the user's fine control over the outputs, it would even be possible to allow users to send multiple outputs with something like this:

```
Send { signature, utxos, outputs, signer } -> {
  let message_array: List<ByteArray> =
    list.push([], cbor.serialise(utxos))
      |> list.push(cbor.serialise(outputs))

  let message_hash_btc =
    get_hash_from_bytearray_btc(cbor.serialise(message_array))
  let message_hash_eth =
    get_hash_from_bytearray_eth(cbor.serialise(message_array))

  let basic_validation = or {
      validate_signature_key(pubKey, message_hash_btc, signature),
      validate_signature_key(pubKey, message_hash_eth, signature),
    }

  and {
    all_outputs_present(outputs, self.outputs),
    all_inputs_present(utxos, self.inputs),
    only_external_signer_inputs_extra(utxos, self.inputs, signer),
  }
}
```