# Formal Verification of Contract Signing Protocol

CS3211 Parallel & Concurrent Programming AY18/19 Semester 2

Yan Hong Yao Alvin (A0159960R), Hu Yiqing (A0167053H), Au Liang Jun (A0173593W)

# 1 Introduction

This report provides a review on our project and contains the following sections:



# 2 Problem Description

Contract signing is an important part of any transaction. With communications occurring more frequently across the internet, there is a need for digital signing methods and protocols that mirror pen-and-paper contracts. Design of protocols to ensure that certain guarantees are met for digital signing need to meet some requirements, such as fairness, that we will discuss later.

Formal verification is an effective and precise method to prove the correctness of protocols. We use a powerful model checking tool, Process Analysis Toolkit (PAT), which can automatically verify if a modeled protocol satisfies some defined properties [1].

In this report, we will formally verify the properties of the Abuse-free Optimistic Contract Signing Protocol, introduced by G. Juan, J. Markus and M. Philip [2].

## 2.1 Abusefree Optimistic Contract Signing Protocol

### Stakeholders

The protocol involves 2 main parties which are signing the contract. We will designate them as P1 and P2. P1 will be the initiator of the protocol by sending the first message with his promise to P2. The third party involved is called the Trusted Party. The Trusted Party is a neutral arbitrator and is called upon to resolve any disputes during the process of the protocol.

## Protocol

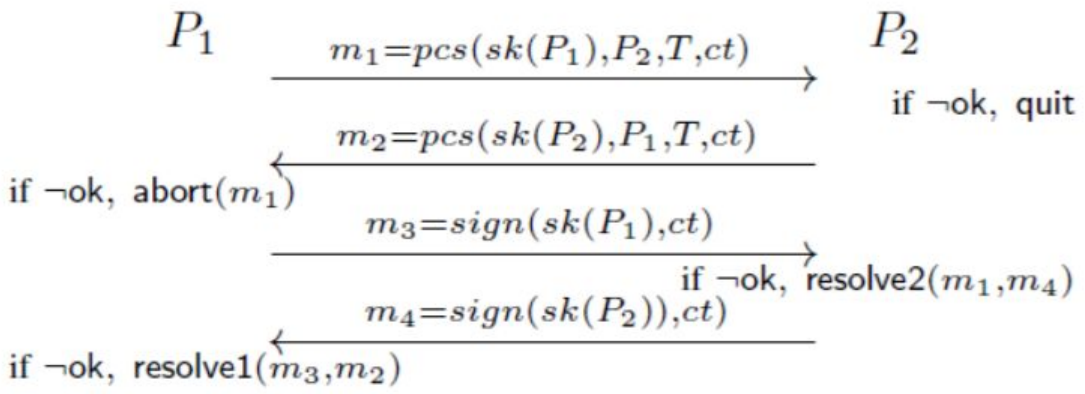


Figure 2.1 Protocol Flow

The steps of the protocol are as follows:

**1.** P1 initiates by sending a promise to sign the contract to P2 ($m_1$ in Figure 2.1)

**2.** P2 checks the validity of P1's promise, and if valid, P2 sends a promise to sign the contract to P1 ($m_2$ in Figure 2.1). If P1's promise is not valid, P2 quits the protocol and does not send anything to P1.

**3.** P1 checks the validity of P2's promise, and if valid, P1 sends his signature on the contract to P2 ($m_3$ in Figure 2.1). If P2's promise is not valid, P1 sends an abort request to the Trusted Party, containing his promise, and stores the return message from the Trusted Party.

**4.** P2 checks the validity of P1's signature, and if valid, sends his own signature on the contract back to P1 ($m_4$ in Figure 2.1). If P1's signature is not valid, P2 sends a resolve2 request to the Trusted Party, containing P1's promise and P2's signature, and stores the return message from the Trusted Party.

**5.** P1 checks the validity of P2's signature, and if valid, the protocol will end. If P2's signature is not valid, P1 sends a resolve1 request to the Trusted Party, containing P2's promise and P1's signature, and stores the return message from the Trusted Party. Then the session ends.

The Trusted Party can be involved in 3 cases - when receiving an abort request, resolve1 request or a resolve2 request. The Trusted Party will also keep track of the status of the contract - whether it has not been contacted to arbitrate for this contract before in which case `status == INIT`, or whether it has responded to an abort request, resolve1 request, or resolve2 request. The Trusted Party also saves important information it needed to arbitrate, in `sig`. The steps taken by the Trusted Party are detailed as follows.

If `status == INIT`:

**Abort Request**: The Trusted Party sets `status = ABORT`, and signs P1's promise, sets `sig = Signed(P1Promise)` and sends it back to P1. This sign indicates that the Trusted Party has received an abort request, and acknowledges that this contract is aborted and not signed.

**Resolve1 Request**: The Trusted Party sets `status = RESOLVE1`, and stores P1's signature. It then converts P2's promise into P2's signature, and stores this as well. It concatenates and signs, P1's

and P2's signature on the contract, sets `sig = Sign(P1Signature + P2Signature)` and sends this back to P1. This ensures P1 has P2's signature and it knows the Trusted Party has sent it.

**Resolve2 Request**: The Trusted Party sets `status = RESOLVE2`, and stores P2's signature. It then converts P1's promise into P1's signature, and stores this as well. It concatenates and signs P1's and P2's signature on the contract, sets `sig = Sign(P1Signature + P2Signature)` and sends this back to P2. This ensures P2 has P1's signature and it knows the Trusted Party has sent it.

If `status == ABORT` or `status == RESOLVE1` or `status  == RESOLVE2`:

The Trusted Party simply sends the value stored in `sig` back to the party that sent the request. Notably, once an abort has been granted on a contract, the Trusted Party will only send the message `Signed(P1Promise)` to confirm the contract has been aborted, and never send out any signatures. Once a resolve has been performed for the contract, the Trusted Party will only send the signatures of both parties, and never send an abort confirmation.

One interesting thing to note about the process is that there can exist a race condition when the following situation happens: P2 sends a valid promise, but the promise is delayed or dropped by the network. P1 would timeout waiting for the promise and request an abort on the contract from the Trusted Party. However, P2 would also wait for P1's signature, and since P1 does not send his own signature back to P2, P2 can also timeout and request for a resolve2 from the Trusted Party. Then depending on the network and timing, if P1's request reaches the Trusted Party first, an abort on the contract would occur. If P2's request reaches the Trusted Party first, both P2 and P1 would then get each other's signatures on the contract.

### Guarantees

There are 3 main guarantees that the protocol achieves, which are defined here.

**Fairness**: No participant should be left in the position of having sent another participant his signature on the contract but not having received the others' signature.

**Optimistic:** The Trusted Party will only be involved if something unexpected happens in the signing process, such as a participant trying to cheat, or network communication is lost.

**Abuse-free:** If a contract protocol is not abuse-free, then it is possible for one party, say Alice, at some point to convince a third party, Val, that Bob is committed to the contract, whereas Alice is not yet committed [2].

## 2.2 Attacker

Since the exchanges between the parties can occur over a public network, there can be attackers on the network that can perform actions which could break the guarantees of the protocol. We consider in the task a powerful attacker that can control the network, which is able to read packets, alter packets, block or delay packets and create packets and inject them on the network.

# 3 System Modelling

## 3.1 Message & Router Implementation

To simulate a typical network infrastructure, we have implemented a router which parties send messages to, and the router forwards the message to its intended recipient. For this, we have implemented one channel to the router, as well as one channel from the router to each party.

```
channel toRouter 10;

channel routerToP1 10;
channel routerToP2 10;
channel routerToTrusted 10;
```

Figure 3.1.1 Message Channels

Note that we have decided to create a large buffer within each channel. This is similar to real networks where hosts may send messages on the network and proceed to perform other tasks. A channel with 0 buffer size would be a synchronous barrier, which would make sending parties wait on the channel until the other party receives it, which is contrary to a realistic model.

```
Router() =
    toRouter?src.dest.msgtype.data1.data2.ct -> Forward(src, dest, msgtype, data1, data2, ct);


Forward(src, dest, msgtype, data1, data2, ct) =
    if (dest == P2IP)
    {
        routerToP2!src.dest.msgtype.data1.data2.ct
        -> Router()
    }
    else if (dest == P1IP)
    {
        routerToP1!src.dest.msgtype.data1.data2.ct
        -> Router()
    }
    else if (dest == TRUSTEDIP)
    {
        routerToTrusted!src.dest.msgtype.data1.data2.ct
        -> Router()
    } else {
        Router()
    };
```

Figure 3.1.2 Router Implementation

Our router's implementation simply listens on the `toRouter` channel, which receives input from P1, P2 and the Trusted Party (as illustrated in Figure 3.1.2). Once it receives a message, it will proceed to the `Forward()` process, which takes in the message, checks for the recipient's IP address (defined with

global constants), and forwards it to the correct recipient. After it forwards the message, the router will return to listening for further messages on the channels indefinitely.

| **Msgtype** | **Promise** | **Sign** | **Abort** | **Resolve1** | **Resolve2** | **AbortConfirm** | **ResolveConfirm** |
|---|---|---|---|---|---|---|---|
| **Field 1** | Source IP | | | | | | |
| **Field 2** | Destination IP | | | | | | |
| **Field 3** | Message Type (e.g. Promise, Signature, Abort request) | | | | | | |
| **Field 4** | Promise | Signature | P1Promise | P1Signature | P2Signature | Confirmation | P1Signature (signed) |
| **Field 5** | EMPTY | EMPTY | EMPTY | P2Promise | P1Promise | EMPTY | P2Signature |
| **Field 6** | Contract | | | | | | |

Figure 3.1.3. Message Encoding Scheme

We now provide the scheme for how the messages sent between each party are structured in our implementation (Figure 3.1.3). As in the code, each message passed between parties consists of 6 fields. We have based our implementation on protocols such as HTTPS to structure the information. The first 3 fields are general header fields, which contain information such as the source and destination IP for routing, and the type of message sent to the other party knows what actions to take. The next 3 fields contain important data needed in the message, such as the promise and signatures to send, as well as the information of the contract.

## 3.2 Cryptography Implementation

In this section, we discuss our implementation of the cryptographic protocol. Due to the difficulty in implementing a cryptographic suite that meets the requirements of the protocol, we have used "classroom" RSA to simulate the promise and signature aspects of the problem. In RSA, each party has a private key, $D$, a public key $E$ and a modulus value $N$.

To encrypt a message, $M$, in RSA, the step is to perform $M^E \bmod N = C$. Given an encrypted value $C$, only the party with the private key $D$ can decrypt it with $C^D \bmod N = M$.

Further, by performing $M^D \bmod N = S$, any party can take this value $S$ and compute $S^E \bmod N = M$. This means that only the party who knows $D$ could have possibly produced this value of $S$, which allows for the property of non-repudiation, and works as a form of signing.

When P1 and P2 sends a promise in their model, they will send their own private key, encrypted with the Trusted Party's public key. Only the Trusted Party will be able to decrypt this promise to obtain P1 or P2's private key from their promise.

When P1 and P2 signs a contract, they will perform the signing step as mentioned, where they take the contract, $M$, and perform $M^D \bmod N$ and sends it to the other party.

When the Trusted Party has to sign a message for abort or resolve, it will also perform the RSA signing by encrypting the message, $M$, with its public key - $M^D \bmod N = T$. In that way, any party receiving $T$ can compute $T^E \bmod N$ using the Trusted Party's values, and obtain the confirmation message. That way,

the party knows that only the Trusted Party could have sent the abort or resolve confirmation. We will elaborate and illustrate this in our code when discussing the implementation of P1 and P2.

## 3.3 Party Implementation

```
P1() =
    [p1State == GENERATE]P1GenerateValues()
    []
    [p1State == SENDPROMISE]P1SendPromise()
    []
    [p1State == RECPROMISE]P1ReceivePromise()
    []
    [p1State == SENDSIGN]P1SendSign()
    []
    [p1State == RECSIGN]P1ReceiveSign()
    []
    [p1State == END || p1State == QUIT]Skip;
```

```
P2() =
    [p2State == GENERATE]P2GenerateValues()
    []
    [p2State == RECPROMISE]P2ReceivePromise()
    []
    [p2State == SENDPROMISE]P2SendPromise()
    []
    [p2State == RECSIGN]P2ReceiveSign()
    []
    [p2State == SENDSIGN]P2SendSign()
    []
    [p2State == END || p2State == QUIT]Skip;
```

Figure 3.3.1 P1 P2 Implementation

Figure 3.3.1 shows the abstracted implementation of P1 and P2. We utilise a state variable for P1 and P2 to allow a general choice for which action they will execute, to follow the steps of the protocol. Depending on the state, the party will be able to pass the guard condition and therefore only have the choice to proceed to the process that takes the correct action as described in the protocol.

Each has a state `GENERATE`, for the start of the protocol, before sending messages and a state `END` for the conclusion of the protocol. However if P1 or P2 initiated a certain action that caused the default perfect path to be deviated, the initiator will exit the protocol with a `QUIT` state. Note that the `QUIT` status is purely reserved for premature termination of the session.

```
P1GenerateValues() =
    p1GeneratePromise{p1Promise = call(Pow, P1PRIVATE, TRUSTPUBLIC) % TRUSTMOD}
    -> p1GenerateSignature{p1Sign = call(Pow, CONTRACT, P1PRIVATE) % P1MOD}
    -> p1SetStateSP{p1State = SENDPROMISE}
    -> P1();

P2GenerateValues() =
    p2GeneratePromise{p2Promise = call(Pow,P2PRIVATE,TRUSTPUBLIC) % TRUSTMOD}
    -> p2GenerateSignature{p2Sign = call(Pow,CONTRACT,P2PRIVATE) % P2MOD}
    -> p2SetStateRP{p2State = RECPROMISE}
    -> P2();
```

Figure 3.3.2 Promise/Signature Generation

We would like to illustrate the use of RSA here in generating a promise and signature. In Figure 3.3.2, P1 generates his own promise and signature on the contract first and stores them to send to P2. Note that to generate his promise, he performs the step $P1Private^{TrustPublic}$ *mod TrustMod*. This is the encryption step of RSA and no party other than Trusted Party with the private key should be able to decrypt this to obtain the plain P1Private within any feasible computational time. P1 also generates his signature on the contract, by performing the step $Contract^{P1Private}$ *mod P1Mod*. This step is known as digital signing in RSA.

We allocated an "initiator" and a "responder" role to P1 and P2 to begin the protocol; after the `GENERATE` state, P1 will be set to a `SENDPROMISE` state, while P2 will be set to a `RECPROMISE` state. P1 will then attempt to send its promise to P2.

```
P1SendPromise() =
    toRouter!P1IP.P2IP.PROMISETYPE.p1Promise.EMPTY.CONTRACT
    -> p1SetStateRP{p1State = RECPROMISE}
    -> P1()
    []
    p1Terminate{p1State = QUIT}
    -> toRouter!P1IP.P2IP.PROMISETYPE.EMPTY.EMPTY.EMPTY
    -> P1();

P1ReceivePromise() = routerToP1?src.dest.msgtype.data1.data2.ct
    -> P1ReceivePromiseAction(src, dest, msgtype, data1, data2, ct);
```

Figure 3.3.3 P1 Sending Promise

We would like to highlight a peculiarity in our implementation of sending promise and signature (as seen in Figure 3.3.3). The first choice here is P1 sending his promise to P2 through the router, before moving on to the `RECPROMISE` state. This represents the case where P1 will send his promise as per normal, following the protocol. The second choice is when P1 intends to quit. It still sends the promise message, but with `EMPTY` in the data fields instead. This is not part of the usual operation of the protocol, but is a method we implement to signal that a party has unexpectedly terminated during the protocol. This could be due to scenarios like losing connection to the network or system failure.

```
P2ReceivePromise() = routerToP2?src.dest.msgtype.data1.data2.ct
    -> P2ReceivePromiseAction(src, dest, msgtype, data1, data2, ct);

P2ReceivePromiseAction(src, dest, msgtype, data1, data2, ct) =
    if(msgtype != PROMISETYPE || ct != CONTRACT || data1 == EMPTY)
    {
        p2SetStateE{p2State = END}
        -> P2()
    }
    else
    {
        p2StoreReceivedPromise{p2RecPromise = data1}
        -> p2SetStateSP{p2State = SENDPROMISE}
        -> P2()
    };
```

Figure 3.3.4 P2 Receiving Promise

P2 will constantly listen from the router for a message when P2 is in the `RECPROMISE` state (Figure 3.3.4). When he receives a message, he will unpack it and if the data is invalid, or the contract is not the contract P2 wishes to sign, P2 will simply end. However, if the message passes those checks, P2 will move on to the `SENDPROMISE` state where P2 will send its promise to P1.

```
P2SendPromise() =
    toRouter!P2IP.P1IP.PROMISETYPE.p2Promise.EMPTY.CONTRACT
    -> p2SetStateRS{p2State = RECSIGN}
    -> P2()
    []
    p2Terminate{p2State = QUIT}
    -> toRouter!P2IP.P1IP.PROMISETYPE.EMPTY.EMPTY.EMPTY
    -> P2();

P2ReceiveSign() = routerToP2?src.dest.msgtype.data1.data2.ct
    -> P2ReceiveSignAction(src, dest, msgtype, data1, data2, ct);
```

Figure 3.3.5 P2 Sending Promise

P2 in its `SENDPROMISE` state (Figure 3.3.5) is similar to P1 , in that its given the same choices as P1: P2 may wish to send a valid promise or send an invalid one. If P2 chooses to send a valid promise, it will attach the correct source and destination IP, the promise message type, its own promise, and the contract to be signed. Otherwise, P2 will send an empty message. P2 will move on to the `RECSIGN` state, where it will wait for P1 to send its signature.

However if P2 chooses to not send its promise, it will move to a `QUIT` state, sending a similar `EMPTY` invalid message and exit accordingly .

```
P1ReceivePromise() = routerToP1?src.dest.msgtype.data1.data2.ct
    -> P1ReceivePromiseAction(src, dest, msgtype, data1, data2, ct);

P1ReceivePromiseAction(src, dest, msgtype, data1, data2, ct) =
    if (msgtype != PROMISETYPE || ct != CONTRACT || data1 == EMPTY)
    {
        // send abort if invalid promise received
        toRouter!P1IP.TRUSTEDIP.ABORTTYPE.p1Promise.EMPTY.CONTRACT
        // receive abort confirmation from trusted party
        -> routerToP1?f1.f2.f3.f4.f5.f6
        // validate abort confirmation
        -> P1ReceiveTrustedAbortConfirmation(f1, f2, f3, f4, f5, f6)
    }
    else
    {
        p1StoreReceivedPromise{p1RecPromise = data1}
        -> p1SetStateSS{p1State = SENDSIGN}
        -> P1()
    };
```

Figure 3.3.6 P1 Receiving Promise

P1 is similar to P2 in its `RECPROMISE` state (Figure 3.3.6) such that it will continuously listen to the router channel waiting for a message. When it receives the message, P1 will unpack it and analyze if the promise is valid by checking if the message type is correct, and if the contract is the contract it wishes to sign and if the promise is not empty. If the promise is valid, P1 will store it and proceed to send its signature to P2 by moving on to the `SENDPROMISE` state.

However, as seen from the code snippet above, if P2 sends an invalid signature, P1 will proceed to request the Trusted Party to abort. P1 will attach the correct source and destination IP, as well as the

abort message type, followed by its own promise and the contract to the Trusted Party to request a abort confirmation. We elaborate on how the Trusted Party handles the abort in section 3.4.

```
P1ReceiveTrustedAbortConfirmation(src, dest, msgtype, data1, data2, ct) =
    if (msgtype == ABORTCONFIRMATION && p1Promise == call(Pow,data1,TRUSTPRIVATE)%TRUSTMOD)
    {
        StoreAbortConfirmation{p1TrustedConfirmation = data1}
        -> p1SetStateE{p1State = END}
        -> P1()

    }
    else if (msgtype == RESOLVECONFIRMATION)
    {
        p1GetP2Sign{p1RecSign = call(Pow, data1, TRUSTPUBLIC) % TRUSTMOD}
        -> p1SetStateE{p1State = END}
        -> P1()
    }
    else
    {
        p1AbortFail{p1State = END}
        -> P1()
    };
```

Figure 3.3.7 P1 Receiving Abort Confirmation

Figure 3.3.7 illustrates what happens when P1 requests an abort. It awaits a confirmation that when received, it will attempt to decrypt with the Trusted Party's public key and check if is of the correct message type. If the result of decryption is P1's promise, P1 will store this confirmation and exit. However, if the message type is `RESOLVECONFIRMATION`, it means that the given contract has already been signed before, and P1 is able to simply obtain P2 signature by decrypting the message. P1 will then store this signature and then quit. Any other response received will cause P1 to assume that the confirmation is invalid and quit accordingly.

If this alternative abort path is not taken, P1 will be in its `SENDSIGN` state (Figure 3.3.8).

```
P1SendSign() =
    toRouter!P1IP.P2IP.SIGNTYPE.p1Sign.EMPTY.CONTRACT
    -> p1SetStateRS{p1State = RECSIGN}
    -> P1()
    []
    p1Terminate{p1State = QUIT}
    -> toRouter!P1IP.P2IP.SIGNTYPE.EMPTY.EMPTY.EMPTY
    -> P1();

P1ReceiveSign() = routerToP1?src.dest.msgtype.data1.data2.ct
-> P1ReceiveSignAction(src, dest, msgtype, data1, data2, ct);
```

Figure 3.3.8 P1 Sending Signature

If P1 chooses to send a valid signature, P1 will move on to the `RECSIGN` state, where it will listen to the router for P2's response, else it will set its state to `QUIT` and send an `EMPTY` invalid message.

```
P2ReceiveSignAction(src, dest, msgtype, data1, data2, ct) =
    if(msgtype != SIGNTYPE || ct != CONTRACT || data1 == EMPTY || call(Pow, data1, P1PUBLIC) % P1MOD != CONTRACT)
    {
        // send resolve2 if invalid sign received
        toRouter!P2IP.TRUSTEDIP.RESOLVE2TYPE.p2Sign.p2RecPromise.CONTRACT
        // receive resolve2 confirmation from trusted party
        -> routerToP2?f1.f2.f3.f4.f5.f6
        // validate resolve2 confirmation
        -> P2ReceiveTrustedResolve2Confirmation(f1, f2, f3, f4, f5, f6)
    }
    else
    {
        p2StoreReceivedSign{p2RecSign = data1}
        -> p2SetStateSS{p2State = SENDSIGN}
        -> P2()
    };
```

Figure 3.3.9 P2 Receiving Signature

When P2 receives the the signature from P1 (Figure 3.3.9), it will unpack it and attempt to check for its validity. In the event the message is invalid (message type is incorrect, contract is invalid, signature is empty, or signature is not actually signing the contract that was agreed upon), P2 will ask the Trusted Party to resolve. P2 does this by tagging a message with the `RESOLVE2TYPE` along with its signature and P1's promise to the Trusted Party.

If P2 receives a valid signature, it will move on to the `SENDSIGN` state, where it will attempt to send its signature to P1.

```
    P2ReceiveTrustedResolve2Confirmation(src, dest, msgtype, data1, data2, ct) =
        if(msgtype == ABORTCONFIRMATION||call(Pow,data2,TRUSTPUBLIC)% TRUSTMOD != p2Sign)
        {
            resolve2Fail{p2State = END}
            -> P2()
        }
        else if(msgtype == RESOLVECONFIRMATION)
        {
            p2StoreResolve2Confirmation{p2TrustedConfirmation = data1}
            -> p2GetP1Sign{p2RecSign = call(Pow, p2TrustedConfirmation, TRUSTPUBLIC) % TRUSTMOD}
            -> p2SetStateE{p2State = END}
            -> P2()
        }
        else
        {
            resolve2FailDefault{p2State = END}
            -> P2()
        };
```

Figure 3.3.10 P2 Receiving Resolve2 Confirmation

If P2 has send a resolve2 request to the Trusted Party, it will then receive a response from the Trusted Party. Figure 3.3.10 illustrates how P2 processes this response. If the message type is of `ABORTCONFIRMATION`, it means that P1 has already requested an abort and P2 is unable to resolve; alternatively if the confirmation is invalid (decrypting the 2nd part of the confirmation with Trusted Party's public key does not yield P2's own signature), P2 will then simply set its state and exit.

However, if the message type is of `RESOLVECONFIRMATION` and it passed the previous checks, P2 will store the first part of the confirmation and extract P1's signature from the confirmation by decrypting it with Trusted Party's public key. P2 will then set its state and end.

Any other derivatives will be deemed invalid and P2 will simply end.

```
P2SendSign() =
    toRouter!P2IP.P1IP.SIGNTYPE.p2Sign.EMPTY.CONTRACT
    -> p2SetStateE{p2State = END}
    -> P2()
    []
    p2Terminate{p2State = QUIT}
    -> toRouter!P2IP.P1IP.SIGNTYPE.EMPTY.EMPTY.EMPTY
    -> P2();
```

Figure 3.3.11 P2 Sending Signature

However if P2 receives a good signature, he can send a valid signature (Figure 3.3.11) and its job is done. It will set its state to `END` and exit. Once again, if P2 intends to sends an invalid signature, it will set its state to `QUIT` and send an `EMPTY` message.

```
P1ReceiveSignAction(src, dest, msgtype, data1, data2, ct)=
    if (msgtype != SIGNTYPE || ct != CONTRACT || data1 == EMPTY || call(Pow, data1, P2PUBLIC) % P2MOD != CONTRACT)
    {
        // send resolve1 if invalid sign received
        toRouter!P1IP.TRUSTEDIP.RESOLVE1TYPE.p1Sign.p1RecPromise.CONTRACT
        // receive resolve1 confirmation from trusted party
        -> routerToP1?f1.f2.f3.f4.f5.f6
        // validate resolve1 confirmation
        -> P1ReceiveTrustedResolve1Confirmation(f1, f2, f3, f4, f5, f6)
    }
    else
    {
        p1StoreReceivedSign{p1RecSign = data1}
        -> p1SetStateE{p1State = END}
        -> P1()
    };
```

Figure 3.3.12 P1 Receiving Signature

When P1 receives a message from P2 while in its `RECSIGN` state, P1 will unpack the message and analyze it for its validity (Figure 3.3.12). If the message type is not of the signature type/the contract is not of the contract that was previously agreed upon/the signature field is empty/the signature is not actually signing the actual contract, P1 will request the Trusted Party to resolve by sending its own signature, P2's promise and package it with the `RESOLVE1TYPE` message type.

If P1 otherwise receives a valid signature, P1 will store the signature, set its state to `END` and exit, officially ending an optimistic session in the protocol.

```
P1ReceiveTrustedResolve1Confirmation(src, dest, msgtype, data1, data2, ct) =
    if (msgtype == ABORTCONFIRMATION||call(Pow,data2,TRUSTPUBLIC)% TRUSTMOD != p1Sign)
    {
        resolve1Fail{p1State = END}
        -> P1()
    }
    else if (msgtype == RESOLVECONFIRMATION)
    {
        p1StoreResolve1Confirmation{p1TrustedConfirmation = data1}
        -> p1GetP2Sign{p1RecSign = call(Pow, p1TrustedConfirmation, TRUSTPUBLIC) % TRUSTMOD}
        -> p1SetStateE{p1State = END}
        -> P1()
    }
    else
    {
        resolve1FailDefault{p1State = END}
        ->P1()
    };
```

Figure 3.3.13 P1 Receiving Resolve1 Confirmation

If P1 has sent a `RESOLVE1TYPE` message to Trusted Party, it will wait until it receives a response (Figure 3.3.13). If the response if a message of `ABORTCONFIRMATION` type, it means the contract signing has already been aborted, P1 will then simply set its state and exit. Similarly if the confirmation received is not valid (decrypting the 2nd part of the confirmation with the Trusted Party's public key does not yield P1's own signature), P1 will set its state and exit.

Alternatively, if all the checks previously passes and the message type is of `RESOLVECONFIRMATION`, P1 will save the first part of the confirmation, and obtain P2's signature by decrypting that confirmation with Trusted Party's public key. P1 will then set its state and exit.

Any other variation is deemed invalid and P1 will simply set its state and exit.

# 3.4 Trusted Party Implementation

Here, we discuss the actions that Trusted Party can undertake.

```
TrustedParty() = routerToTrusted?src.dest.msgtype.data1.data2.ct
    -> TrustedPartyAction(src, dest, msgtype, data1, data2, ct);

TrustedPartyAction(src, dest, msgtype, data1, data2, ct) =
    if ((trustedStatus == RESOLVED1 || trustedStatus == RESOLVED2) && trustedContract == ct)
    {
        trustedToRouter!TRUSTEDIP.src.RESOLVECONFIRMATION.confirmation.auth.CONTRACT
        -> TrustedParty()
    }
    else if (trustedStatus == ABORTED && trustedContract == ct)
    {
        trustedToRouter!TRUSTEDIP.src.ABORTCONFIRMATION.confirmation.EMPTY.CONTRACT
        -> TrustedParty()
    }
```

Figure 3.4.1 Status Checking

The Trusted Party initially listens indefinitely on the channel from the router for any message from P1 or P2 requiring its intervention. Upon receiving a message, it will proceed to the process `TrustedPartyAction()`, which makes an action depending on the type of intervention and the status of

Trusted Party's actions on the contract before, as described in the protocol. We first look at the situation described in the protocol where Trusted Party has already been called to intervene on the contract before.

In the first `if` case in Figure 3.4.1, the Trusted Party has already received and granted a resolve request on this contract. In this case, the Trusted Party checks that the contract it is mediating is correct and then proceeds to send the saved values of *Signed(P1Signature)* and *Signed(P2Signature)* back to the party who sent the request.

In the second case, the Trusted Party has already received and granted an abort request on this contract. In this case, the Trusted Party checks that the contract it is mediating is correct and then proceeds to send the saved value of *Signed(P1Promise)* back to P1, which is the party that is able to send abort requests.

Next, we consider the cases when the Trusted Party did not previously receive requests regarding the contract.

## Abort

```
TrustedPartyAction(src, dest, msgtype, data1, data2, ct) =
    if ((trustedStatus == RESOLVED1 || trustedStatus == RESOLVED2) && trustedContract == ct)
    ...
    else if (trustedStatus == ABORTED && trustedContract == ct)
    ...
    else if (msgtype == ABORTTYPE)
    {
        setStatusAbort{trustedStatus = ABORTED}
        -> storeContract{trustedContract = ct}
        -> generateAbortConfirmation{confirmation = call(Pow, data1, TRUSTPRIVATE) % TRUSTMOD}
        -> trustedToRouter!TRUSTEDIP.src.ABORTCONFIRMATION.confirmation.EMPTY.CONTRACT
        -> TrustedParty()
    }
```

Figure 3.4.2 Abort Handling

In the case of receiving an abort request, the Trusted Party's actions are straightforward and correspond directly to the protocol description. The Trusted Party will set the status to `ABORTED`, signs P1's promise with its private key, stores this value and sends the confirmation back to P1 (expandd section in Figure 3.4.2).

## Resolve1/Resolve2

```
TrustedPartyAction(src, dest, msgtype, data1, data2, ct) =
    if ((trustedStatus == RESOLVED1 || trustedStatus == RESOLVED2) && trustedContract == ct)
    ...
    else if (trustedStatus == ABORTED && trustedContract == ct)
    ...
    else if (msgtype == ABORTTYPE)
    ...
    else if(msgtype == RESOLVE2TYPE && call(Pow, data1, P2PUBLIC) % P2MOD == CONTRACT)
    ...
    else if(msgtype == RESOLVE1TYPE && call(Pow, data1, P1PUBLIC) % P1MOD == CONTRACT)
    {
        storeP1Signature{trustedResolveSign = data1}
        -> setStatusResolved1{trustedStatus = RESOLVED1}
        -> storeContract{trustedContract = ct}
        -> generateP2Private{generatedPrivate = call(Pow, data2, TRUSTPRIVATE) % TRUSTMOD}
        -> generateP2Signature{generatedSignature = call(Pow, CONTRACT, generatedPrivate) % P2MOD}
        -> generateResolved1Confirmation{confirmation = call(Pow, generatedSignature, TRUSTPRIVATE) % TRUSTMOD}
        -> generateAuth{auth=call(Pow, data1, TRUSTPRIVATE) % TRUSTMOD}
        -> toRouter!TRUSTEDIP.src.RESOLVECONFIRMATION.confirmation.auth.CONTRACT
        -> TrustedParty()
    }
    else
    ...;
```

Figure 3.4.3 Resolve Handling

In the case of receiving a resolve request, both Resolve1 and Resolve2 are very similar, and follow the protocol closely. Resolve1 is illustrated in the expanded section of Figure 3.4.3. Trusted Party will then check if P1 or P2 has indeed signed the contract, they did, Trusted Party will continue after storing the signature .The Trusted Party will set the status to `RESOLVED1` (respectively, `RESOLVED2`), generate P1's (respectively, P2's) private key by decrypting P1's (respectively, P2's) promise with its own private key, and create P1's (respectively, P2's) signature on the contract using the private key. Then, the Trusted Party will sign both P1's and P2's signatures on the contract with its private key. Both P1 and P2's signatures will be sent as part of the message back to the party that requested the resolve request.

```
TrustedPartyAction(src, dest, msgtype, data1, data2, ct) =
    if ((trustedStatus == RESOLVED1 || trustedStatus == RESOLVED2) && trustedContract == ct)
    ...
    else if (trustedStatus == ABORTED && trustedContract == ct)
    ...
    else if (msgtype == ABORTTYPE)
    ...
    else if(msgtype == RESOLVE2TYPE && call(Pow,data1,P2PUBLIC) % P2MOD == CONTRACT)
    ...
    else if(msgtype == RESOLVE1TYPE && call(Pow,data1,P1PUBLIC) % P1MOD == CONTRACT)
    ...
    else
    {
        toRouter!TRUSTEDIP.src.INVALIDCONFIRMATION.EMPTY.EMPTY.EMPTY
        -> TrustedParty()
    };
```

Figure 3.4.4 Invalid Request Handling

Finally, if none of the criterion is met, Trusted Party will deem the request invalid an send an `INVALIDCONFIRMATION` (Figure 3.4.4)..

## 3.5 Attacker Implementation

Lastly we are going to explore the actions of the Attacker model we implemented. Because the communication channel between P1,P2 and Trusted Party are assumed to be insecure, there are many ways to "attack" the contract signing process. We have decided to implement a Denial-of-Service (DOS) attacker.

```
RouterSelector() =
    secureRouter{routerState = SECURE} -> Router()
    []
    attacker{routerState = ATTACKER} -> Attacker();
```

Figure 3.5.1 Attacker Instantiation

We model the attack method on the communication between the 3 parties by having the Attacker taking up the role of the router. As in Figure 3.5.1, the `RouterSelector()`, which is instantiated together with the System, allows for the choice of a secure router, or one that is compromised by the Attacker. Unlike a benign router, the Attacker can choose to intercept and inject unintelligible messages into the original messages.

```
Attack(src, dest, msgtype, data1, data2, ct) =
    if(dest == P2IP)
    {
        routerToP2!src.dest.msgtype.data1.data2.ct
        -> Attacker()
        []
        routerToP2!GARBAGE.GARBAGE.GARBAGE.GARBAGE.GARBAGE.GARBAGE
        -> Attacker()
    }
    else if (dest == P1IP)
    {
        routerToP1!src.dest.msgtype.data1.data2.ct
        -> Attacker()
        []
        routerToP1!GARBAGE.GARBAGE.GARBAGE.GARBAGE.GARBAGE.GARBAGE
        -> Attacker()
    }
    else if(dest == TRUSTEDIP)
    {
        routerToTrusted!src.dest.msgtype.data1.data2.ct
        -> Attacker()
        []
        routerToTrusted!GARBAGE.GARBAGE.GARBAGE.GARBAGE.GARBAGE.GARBAGE
        -> Attacker()
    } else {
        Attacker()
    };
```

Figure 3.5.2 Attacker Behaviour Implementation

The Attacker initially behaves the same as a Router process as shown in Figure 3.1.2; however, instead of forwarding the messages with `Forward()`, it utilises the `Attack()` process. As observed, the `Attack()`

is similar to the normal `Forward()` in the way that it reads the destination and source IP address and forwards the messages. However, at anytime when relaying the messages, the Attacker may wish to send `GARBAGE` in all the message fields in place of the actual message. `GARBAGE` is a negative integer value that has no possibility of being used in any valid message passed in our model, thus the recipient that detects a `GARBAGE` in its fields will deem the message invalid.

## 3.6 Overall System Implementation

```
System() = P1() ||| P2() ||| TrustedParty() ||| RouterSelector();
```

Figure 3.6.1 System Implementation

We initialise the protocol by having all parties run interleaved, since each party should be able to perform events independently without needing synchronisation in this model.

# 4 Investigated Properties

In this section, we cover the properties that we investigated with the help of our model. Before we explore them, we introduce the definition of the following conditions in Figure 4.1. In non-compromised environments, the `noattacker` condition will be true. This will be a default condition for most of our assertions, as the main objective is to verify the protocol in normal circumstances. We will explore the immunity of the protocol against attack in the final part of the section.

```
#define noattacker (routerState == SECURE);
#define withattacker (routerState == ATTACKER);
```

Figure 4.1 Router State

### Correctness

The first property of our model explored is not related to the protocol, but specific to our implementation. We first ensured that the implementation of our protocol is correct, that P1 and P2 were indeed only responding to the other party when it could verify that message sent by the other is authentic.

```
// 1. P2 only sends promise when it receives valid P1 promise
#define p2sendpromise (p2State == RECSIGN);
#define p2receivesgoodpromise (p2RecPromise == p1Promise && p2State != QUIT);
#assert System |= [](p2sendpromise -> p2receivesgoodpromise);
```

Figure 4.2 Correctness Assertion 1

The above assertion performs the aforementioned verification. If P2 sends its promise (indicated with a state change to `RECSIGN`), we assert that it had received a valid promise from P1 (indicated by its received promise being equal to P1's generated promise).

```
// 2. P1 only sends signature when it receives valid P2 Promise
#define p1sendsign (p1State == RECSIGN);
#define p1receivesgoodpromise (p1RecPromise == p2Promise && p1State != QUIT);
#assert System |= [](p1sendsign -> p1receivesgoodpromise);
```

Figure 4.3 Correctness Assertion 2

Similarly, if P1 has sent its signature, we assert that it must have received a valid P2 promise.

## Completeness

The next property explored is completeness. Under all conditions, the protocol should enable both parties to reach a terminated state, regardless of whether they have sent or received signatures.

```
// 3. Both processes eventually end or quit
#define processesend (p1State == END || p1State == QUIT) && (p2State == QUIT || p2State == END);
#assert System |= [](noattacker -> <>processesend);
```

Figure 4.4 Completeness Assertion 1

From Figure 4.4, `processesend` checks that both parties eventually end up in `QUIT` or `END` states, which represent termination.

Note that we do not explore the deadlock freeness of the entire System. As the `TrustedParty()` and `Router()` are processes that never terminate, we are unable to verify that they do not deadlock. Instead, we use the above assertion to check that both parties do not pause indefinitely and will eventually reach an end.

## Fairness

As defined in section 2, Fairness is ensured when no party is left in the position of having sent another participant his signature on the contract but not having received the other's signatures. Arguably the most important property, we verify that this property is enforced in the protocol.

At the simplest level, we ensure that if a party receives a promise from the other, he will eventually get the other's signature. This encompasses both the scenarios in which the other party cooperates and sends his signature, and the scenario which the other party terminates and his signature needs to be obtained from the Trusted Party.

```
// 5. If P1 receives P2's promise and doesn't quit, it will get P2's signature
//    Covers both success scenario and resolve1
#define p1receivep2promise (p1RecPromise == p2Promise && p1State != QUIT);
#define p1receivep2sign (p1RecSign == p2Sign);
#assert System |= [](p1receivep2promise && noattacker -> <>p1receivep2sign);
```

Figure 4.5 Fairness Assertion 1

In the Figure 4.5, we first define the event of P1 receiving P2's promise, indicated by verifying that its received promise is equal to P2's generated promise. In addition, we add the condition that P1 has not quit, for if it has, it will not be able to perform any further actions. We then define the event of P1 receiving P2's signature as P1's received signature equating to P2's generated signature.

We then assert that as long as P1 does not receive P2's promise (the "always" clause is necessary as we need to ensure P1 does not quit at a later point of time), it will eventually receive P2's signature.

```
// 6. If P2 receives P1's promise and doesn't quit, it will get P1's signature
//    Covers both success scenario and resolve2
#define p2receivep1promise (p2State != QUIT && p2RecPromise == p1Promise);
#define p2receivep1sign (p2RecSign == p1Sign);
#assert System |= [](p2receivep1promise && noattacker -> <>p2receivep1sign);
```

Figure 4.6 Fairness Assertion 2

Similarly, if P2 receives P1's promise and does not quit, it must successfully obtain P1's signature one way or another.

```
// 4. If P1 sends signature, both parties will eventually get signature
#define bothreceivesign (p1RecSign == p2Sign && p2RecSign == p1Sign);
#assert System |= [](p1sendsign && noattacker -> <>bothreceivesign);
```

Figure 4.7 Fairness Assertion 3

Figure 4.7 summarises Fairness Assertion 1 and 2, and broadly claims that if P1 sends its signature, both parties will eventually receive the other signature. The clause ensuring that both parties do not quit is not necessary here, as P1's option of quitting happens before he sends his signature, and even if P2 chooses to quit, he will first save P1's signature before doing so. P1 can then obtain P2's signature from the Trusted Party.

```
// 7. If trusted party has once aborted, neither party will receive signature
#define trustedstatusaborted (trustedStatus == ABORTED);
#assert System |= [](trustedstatusaborted -> []!bothreceivesign);
```

Figure 4.8 Fairness Assertion 4

Furthermore, we ensure that the Trusted Party adheres to fairness. If the Trusted Party has once served an abort request (indicated with its status being `ABORTED`), neither parties should receive signatures anymore. This is as the abort request could not have been served if either have already received signatures, and once the abort request is served the Trusted Party should not entertain any future requests for signatures.

```
// 8. If trusted party has once resolved, he will not entertain aborts
#define trustedstatusresolve (trustedStatus == RESOLVED1 || trustedStatus == RESOLVED2);
#define abortconfirmation (p1TrustedConfirmation == call(Pow, p1Promise, TRUSTPRIVATE) % TRUSTMOD);
#assert System |= [](trustedstatusresolve -> []!abortconfirmation);
```

Figure 4.9 Fairness Assertion 5

In the same manner, once the Trusted Party has resolved, he should not entertain requests from P1 to abort. If he does, he may deprive a party from obtaining the signature via a resolve request in future. In Figure 4.9, we define `trustedstatusresolve` to be the Trusted Party having resolved a request. `abortconfirmation` checks if P1 has received an abort confirmation before. The assertion thus checks that once Trusted Party has resolved, P1 should never receive an abort confirmation.

## Optimism

The second key property the assertion aims to guarantee, optimism guarantees that the Trusted Party will only be involved in the event that both parties are unable to execute the protocol on their own.

```
// 9. If there are no interruptions, trusted party will remain in INIT state
#define trustedinit (trustedStatus == INIT);
#define bothnoquit (p1State != QUIT && p2State != QUIT);
#assert System |= [](bothnoquit && noattacker -> trustedinit);
```

Figure 4.10 Optimism Assertion 1

We first define `bothnoquit` as both parties not terminating prematurely. `trustedinit` checks for the Trusted Party still in it's `INIT` state, which indicates that it has not received any communication. The assertion thus checks that if both parties did not quit (and therefore face no interruptions), the Trusted Party will remain in its uncontacted `INIT` state. As such, if the assertion holds, the Trusted Party is not involved in regular scenarios.

## Abusefree

The third key trait of being Abusefree is verified by ensuring the Trusted Party only resolves when he receives valid signatures from the request sender. For example, P1 sends resolve1 requests to the Trusted Party seeking to obtain P2's signature. The protocol verifies P1 is committed to the contract by requiring his signature, as such preventing abuse.

```
// 10. Trusted party will only resolve if he received a valid requester signature
#define trustedresolve1 (trustedStatus == RESOLVED1);
#define trustedreceivevalidp1sign(trustedResolveSign == p1Sign);
#define trustedresolve2 (trustedStatus == RESOLVED2);
#define trustedreceivevalidp2sign(trustedResolveSign == p2Sign);
#assert System |= [](((trustedresolve1 -> trustedreceivevalidp1sign) &&
                    (trustedresolve2 -> trustedreceivevalidp2sign));
```

Figure 4.11 Abusefree Assertion 1

We simply assert that if Trusted Party has a resolve status, he must have received a valid signature (stored in `trustedResolveSign`) that corresponds to the sender of the request.

## DOS Immunity

As covered earlier, our model includes a Denial-of-Service attacker. We test for Fairness and Completeness with the attacker present.

```
// 11. Assertion 4 with attacker (Fairness)
#assert System |= [](p1sendsign && withattacker -> <>bothreceivesign);

// 12. 13. Assertion 5 and 6 with attacker (Fairness)
#assert System |= [](<>p1receivep2promise && withattacker -> <>p1receivep2sign);
#assert System |= [](<>p2receivep1promise && withattacker -> <>p2receivep1sign);
```

Figure 4.12 DOS Immunity Assertions 1, 2 and 3

The assertions shown in Figure 4.12 are simply replicated from the Fairness section, with the noattacker condition modified to `withattacker` (as defined in Figure 4.1). If the model is resistant to DOS attacks, the assertions for Fairness will return true.

```
// 14. Assertion 3 with attacker (Completeness)
#assert System |= [](withattacker -> <>processesend);
```

Figure 4.13 DOS Immunity Assertion 4

The fourth assertion (Figure 4.13) for this property simply tests for Completeness, modified in the same way as the above. If both processes are able to terminate even when the Attacker is present, the assertion should pass verification.

# 5 Experiments

## 5.1 Simulation

As a simulation run through involves over 40 states, with many "internal" events (e.g. P1 changing state after performing an action), we will feature and illustrate certain noteworthy events in place of showing the entire state graph.

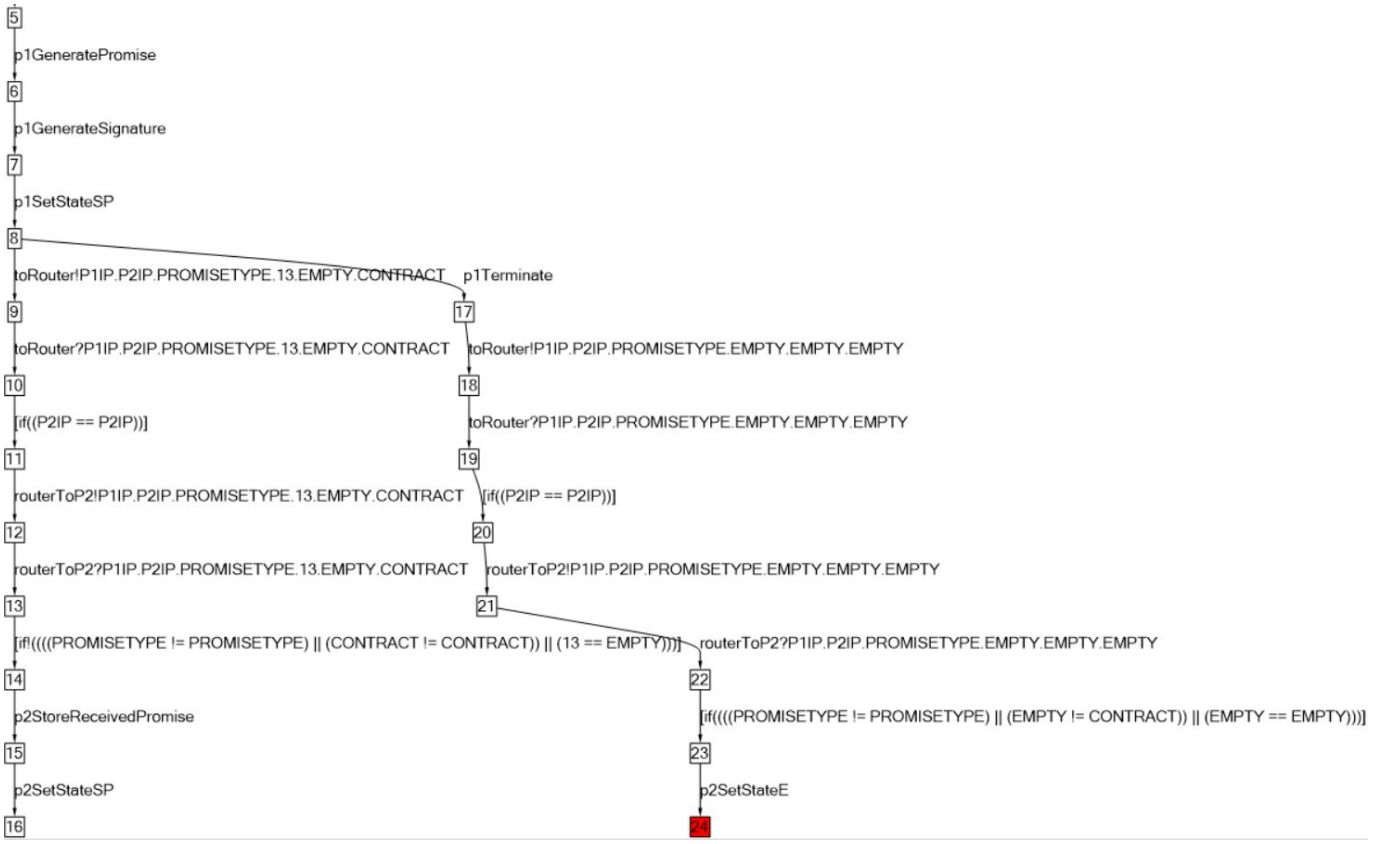


Figure 5.1 P1 Sending Promise

### Initialisation

Before the actual protocol communication proceeds, the parties will first generate their promise and signature. In Figure 5.1, we see P1's generation of promise and signatures in states 5 and 6. In simulations, these internal events belonging to both parties' can be interleaved. However, each party needs to complete this stage before they are ready to send or receive promises.

### Message Transmission

The above figure also illustrates how message passing works in our model. P1 transmits a message by placing its source, destination, message type, promise and contract into the channel (state 8) with the router listening, where the router will receive the message (state 9) and put it through a series of checks to determine where the message should be sent to (state 10). In the same manner, it puts the message through the channel where the destination party picks it up (state 11 and 12).

## Choice to Send Bad Response

In Figure 5.1, P1 at initialisation time is shown. Upon generating it's signature and promise (state 8), its role in the protocol would be to send a promise to P2. At this juncture, it is given the option to either send the promise (left path), or choose to quit by sending a bad response (right path). At all points where the parties are expected to send a response, the option to send a bad response is present. We will explore what happens in the event of a bad response later below under Trusted Party.


Figure 5.2 P1 and P2 Termination

## Termination

Figure 5.2 illustrates the scenario where both parties are able to receive the other's signature without incident. P2 receives and stores P1's signature (state 35), sending his own (state 37) before terminating (state 38). P1 receives and stores P2's signature (state 45) and ends as well. Under normal conditions, state 47 is the final state that takes place.

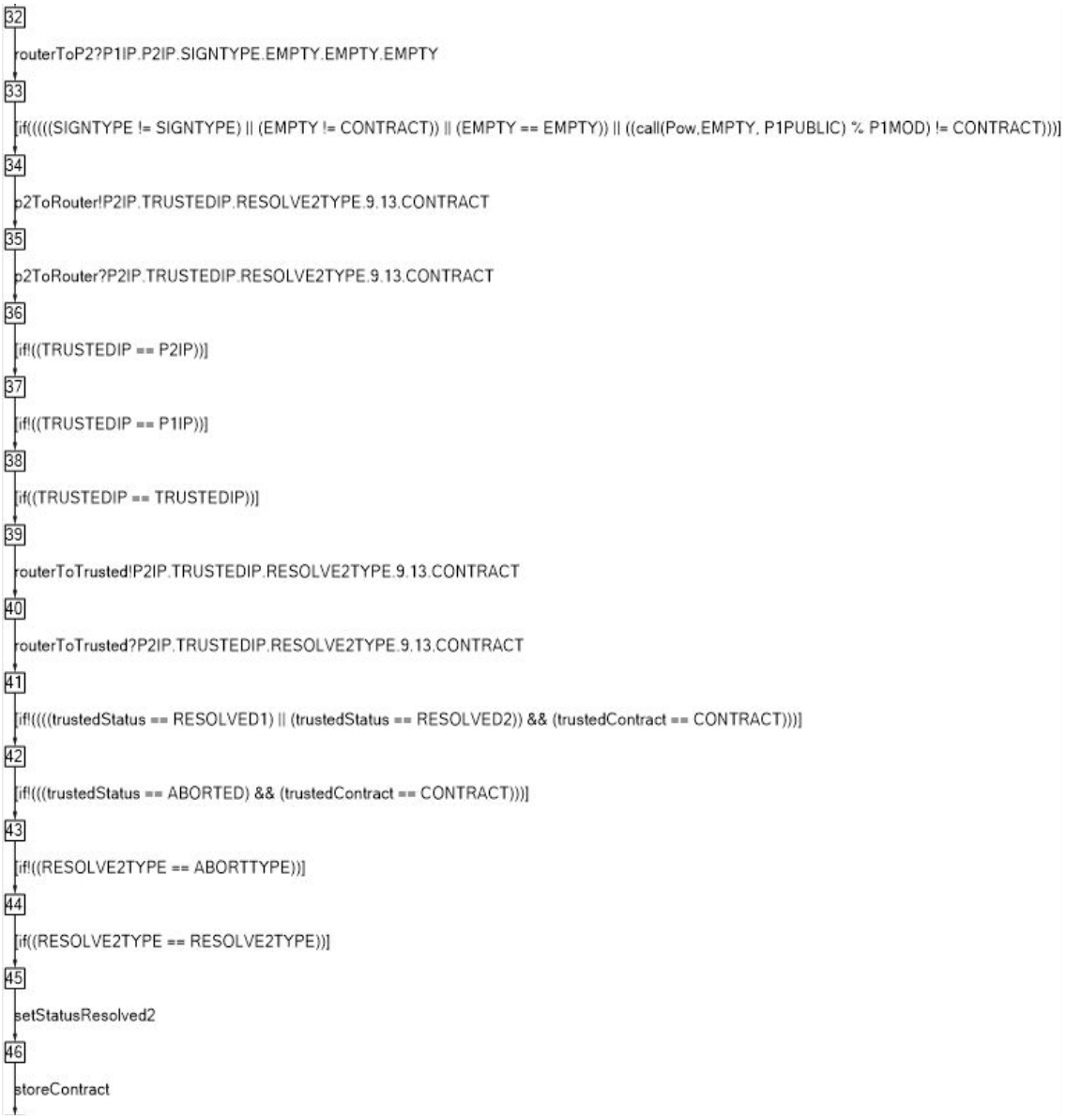


Figure 5.3 P2 Sending Resolve2 Request

## Request to Trusted Party

Figure 5.3 illustrates the sending of a resolve2 request from P2 to the Trusted Party. Initially, P2 receives a bad response in place of a signature (state 32). Upon verifying that it is indeed bad (state 33), P2 immediately fires off a resolve2 request to the Trusted Party (state 34 to 40). The Trusted Party first checks if he has previously answered any requests (state 41 and 42), to ensure that a previous decision is not overruled by the new request. In the event that there is no previous request, he then inspects the current request (state 43 and 44) and handles it (state 45 and 46).

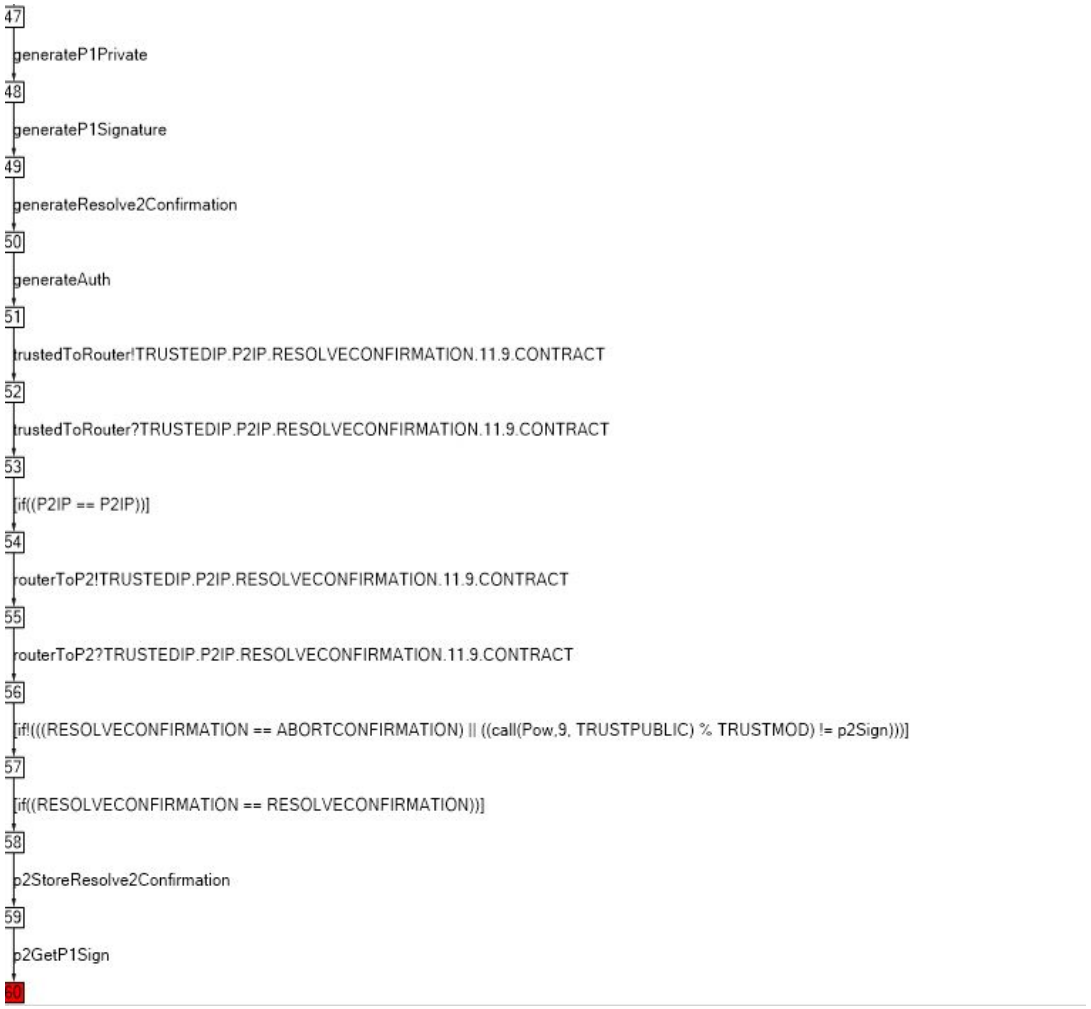


Figure 5.4 Trusted Party Handling Resolve2 Request

## Trusted Party Handling Requests

Figure 5.4 resumes from the end of Figure 5.2. The Trusted Party generates P1's signature, as well as the resolve2 confirmation (state 47 to 50), then transmits the message to P2 (state 51 to 57). Upon receiving the resolve2 confirmation, P2 stores it and extracts P1's signature from within, thereby obtaining P1's signature from the Trusted Party.

## 5.2 Verification

| No. | Assertion | Pass/Fail |
|---|---|---|
| 1 | Correctness Assertion 1 | Pass |
| 2 | Correctness Assertion 2 | Pass |
| 3 | Completeness Assertion 1 | Pass |
| 4 | Fairness Assertion 1 | Pass |
| 5 | Fairness Assertion 2 | Pass |
| 6 | Fairness Assertion 3 | Pass |
| 7 | Fairness Assertion 4 | Pass |
| 8 | Fairness Assertion 5 | Pass |
| 9 | Optimism Assertion 1 | Pass |
| 10 | Abusefree Assertion 1 | Pass |
| 11 | DOS Immunity Assertion 1 | Fail |
| 12 | DOS Immunity Assertion 2 | Fail |
| 13 | DOS Immunity Assertion 3 | Fail |
| 14 | DOS Immunity Assertion 4 | Fail |

Figure 5.2.1 Verification Results in PAT

### DOS Immunity

As we had expected, the protocol in its basic form is unable to guarantee Fairness when an attacker is present. A DOS attacker with complete control over the network is able to block communications whenever he wishes, hence allowing for situations where promises or one signature is exchanged successfully before the attacker interferes. Further signatures cannot be exchanged, and thus there is no way to guarantee either or both parties will receive the signatures when some promises/signatures have already been received (i.e. no Fairness). This explains why DOS Immunity Assertion 1,2 and 3 (defined in Figure 4.12) fails.

DOS Immunity Assertion 4 tests for Completeness, which we had not anticipated to fail in the presence of an attacker. As such, we have produced the witness trace of the failing example for analysis (Figure 5.2.2)


Figure 5.2.2 Witness Trace of DOS Immunity Assertion 4

From Figure 5.2.2, we can identify that P1 is sending his promise to P2 (state 8). The attacker, which reads from the `toRouter` channel intercepts the communication and replaces the content with `GARBAGE` (states 9 to 11). P2, upon receiving the invalid message, ends as intended (states 12 to 15). However, P1 having sent a valid message with no intention to quit, still awaits some response (either valid or invalid) from P2.

We believe that this is actually a flaw of the protocol. In the protocol, there is no specification for resending requests and messages. This means that P1 or P2 will perceive one message as the intention of the other party without verifying it, even if the intention is unintelligible.

For example, if P1 sends P2 a valid promise, P1 will believe that it's intentions have been properly sent through the channel and expects some sort of response from P2. However if the message has been intercepted and replaced with some sort of invalid message, P2 will determine it as P1's intention to quit and thus exiting himself. This miscommunication in intentions will result in P1 deadlocking by waiting forever for a response that will never come.

However, as explained in section 6.1, having a timeout and resend system will allow P1 and P2 to reiterate their intentions and guard against these kind of miscommunications. While timeout is a concept that exists in real-life networking, we did not manage to incorporate it in our model as the original task did not introduce it. This prevents the guarantee of completion that DOS Immunity Assertion 4 checks for.

## Other Properties

The remaining properties of Correctness, Completeness, Fairness, Optimism and Abusefree are successfully verified under non-compromised networks.

Of note, the latter 3 properties are the key guarantees of the protocol. Successful verification implies that the protocol is as described, as far as we have investigated. Completeness also ensures that both parties will eventually terminate, regardless of whether the protocol proceeded to completion.

Correctness is also crucial in the validity of our implementation, and in turn findings; if it were not verified, then our results would not hold enough truth to suffice as a formal verification.

# 6 Discussion

## 6.1 Limitations

The following section explores the aspects of the protocol which our model is unable to fully capture.

### Timeout

Normally, if either party has not received an intended response in specific amount of time, they would re-send their message. This specific amount of time is hardcoded as a "timeout". However since the PAT timeout is actually recording the number of events that has occurred since time 0 till the time of incidence. Implementing a timeout is extremely infeasible. As a P1 or P2 follows the same timer, ie: the number of events occured. P1 could basically carry out N number of events while P2 has not even begun yet. This synchronized event timer makes it impossible to model P1 and P2 as separate entities.

The clear benefits of having a timeout is that the current implemented attacker model will not be effective at disrupting the protocol. P1 , P2 and the Trusted Party will repeatedly re-send messages if there is no acknowledgement. Given a bounded attacker that is not able to permanently block the communication channel, P1 and P2 should eventually be able to carry out their contract signing.

### Classroom RSA implementation

The problem with our classroom RSA implementation is that the `var` data type we use to store our values is capped at 32 bits. Thus we are unable to use primes that are 2 digits or greater. This has the unfortunate side effect of making our public and private keys exactly the same. However, because we intended it to be just a proof of concept, small primes do not inhibit our ability to model the RSA signing, encryption and decryption.

Another problem with our implementation is that P2 is unable to verify P1's promise and vice versa. This opens the protocol to an attack where P1 is able to send an invalid promise and convinces P2 of P1's honest intentions of signing the contract. This should not be possible under the cryptographic system described in the protocol. Further, we unable to encrypt large parts of messages to guarantee authenticity for all segments of the protocol . This makes our implementation intrinsically vulnerable.

### A "Smarter"Attacker

Initially, it was planned to implement a smarter form of attacker. This smarter Attacker is supposed to be able to impersonate P1 and P2 to trick P1 and P2 that they are signing the contract with each other, however in reality they are actually signing 2 separate contracts with the attacker respectively.

Usually an entity's public key is signed by a Certificate authority (CA) so that people wishing to communicate with this party is able to obtain their legitimate public keys. However because the existence of a CA was not guaranteed, we decided that all public keys are public that means everyone

has access to them.

Thus when P1 or P2 sends its signature, they signed a contract with their private keys. That means when the other party receives the signature, they are able to decrypt it with the respective public keys that everyone has access to, to validate the signature. We have essentially guaranteed authenticity for the signatures. This means that the "smarter" Attacker is unable to forge fake valid signatures or promises to trick P1 and P2 to sign a new contract of the Attacker's choice. . However if the contracts do not change. A "smarter" Attacker is able to orchestrate a replay attack to make either P1 or P2 resign an old contract. However, since our implementation focused on a single session, this attack was not considered.

Alternatively, if a CA was implemented, the attacker could somehow force P1 and P2 to accept a fake CA that informs P1 and P2 that the attacker's public key is actually the public key of the person they are trying to sign a contract with. However, the team decided not to model this scenario with the CA as it requires specific vulnerabilities that may not always be present, though it has occured before in security incidents such as Lenovo Superfish[3].

## 6.2 Extensions

In this section we will explore possible extensions in the implementations, to verify the protocol in a wider range of scenarios.

### Multi-Session Implementation

One possible extension we considered is a multi-session Implementation. Where P1 and P2 will continue signing contracts from a set of contracts. It is noted that this set of contracts would be limited due to the 32 bit nature of our system.

P1 and P2 will take polls to decide who initiates the signing of which contract first by using a simple `math.rand()` function. Additionally, a separate queue of signed contracts, aborted contracts as well as resolved contracts would also be kept to better simulate the actions of the Trusted Party.

We could model this potentially be having P1 and P2 enter either a "receive function state" or a "sender function state" to initiate this multi-Session protocol. However it should also be noted that a not okay response sent would also have to be altered to suit this multi-session approach.

### Multi-Party Contract Signing

Another possible extension we considered is the possibility of a Multi-Party contract signing. We aimed to model this by either doing some sort of token passing method, or a polling method where the Trusted Party would be given the additional task of coordinating this multi-party contract signing

The problem with the token passing method is that given the limited number of primes we can use with an integer ceiling of $2^{32}$ bits, we have exhausted all the possible primes that are to be used. Additionally, verifying signatures will become extremely tasking for the $N-1$ Party in a $N$ party system, making it computationally unfair for those parties. Additionally, providing authenticity and integrity would amount to roughly $O(N(N-1)/2)$ complexity or $O(N^2)$, as every $N$th message would have to account for the previous $N-1$th message, which is not efficient.

The other method is to appoint the Trusted Party the task of gathering all the signatures, but gives the Trusted Party too much power and oversight. Thus, a practical implementation of this method might not be adopted.

# 7 Conclusion

In conclusion, our model has verified the key traits of Abuse-Free Optimistic Contract Signing Protocol as specified in the 1999 paper by Garay, Juan & Jakobsson, Markus & Mackenzie, Philip. We were able to create a model that represented the protocol fairly accurately, and run verifications that investigated various properties. Our experiments have successfully verified the 3 essential requirements - fairness, abuse-free and optimistic as well as an additional property of being complete.

However, these traits do not hold up in the event of an attacker, specifically a DOS attacker. The same verified assertions that test Fairness and Completion break down, due to the sheer control of a DOS attacker as well as limitations in the protocol. In that aspect, the protocol was found to be flawed. We have discussed a way that would make the protocol more resistant to 3rd party influences.

Looking ahead, our implementation could potentially be improved to model more elements that would be present in a real-life implementation of the protocol. We have discussed some such areas, including multi-session and multi-party contract signing in this report.

# 8 References

[1] Sun, J., Liu, Y., Dong, J. S. & Pang, J. PAT: Towards Flexible Verification under Fairness. Computer Aided Verification Lecture Notes in Computer Science 709–714 (2009).

[2] Garay, J. A., Jakobsson, M. & Mackenzie, P. Abuse-Free Optimistic Contract Signing. Advances in Cryptology — CRYPTO' 99 Lecture Notes in Computer Science 449–466 (1999).

[3] Brewster, T (2015, Feb 19) How Lenovo's Superfish 'Malware' Works And What You Can Do To Kill It.Forbes. Retrieved from: https://www.forbes.com/sites/thomasbrewster/2015/02/19/superfish-need-to-know/

Bîrjovanu, C. V. & Bîrjovanu, M. An Optimistic Fair Exchange E-commerce Protocol for Complex Transactions. Proceedings of the 15th International Joint Conference on e-Business and Telecommunications (2018).

Kähler, D., Küsters, R. & Wilke, T. A Dolev-Yao-Based Definition of Abuse-Free Protocols. Automata, Languages and Programming Lecture Notes in Computer Science 95–106 (2006).

Process Analysis Toolkit (PAT) 3.5 User Manual