Formal Verification of a Contract Signing Protocol

CS3211 Project 1

Alvin, Yiqing, Liang Jun

Task

Contract Signing Protocol

Allows a set of participants to exchange messages with each other in order to arrive at a state in which **each of them has a pre-agreed contract signed by others**

Contract Signing Protocol

An important property of contract signing protocols is 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' signatures**.

Contract Signing Protocol

Party 1

P_1 = public key
$sk(P_1)$ = private key

Contract

ct = contract id

Party 2

P_2 = public key
$sk(P_2)$ = private key

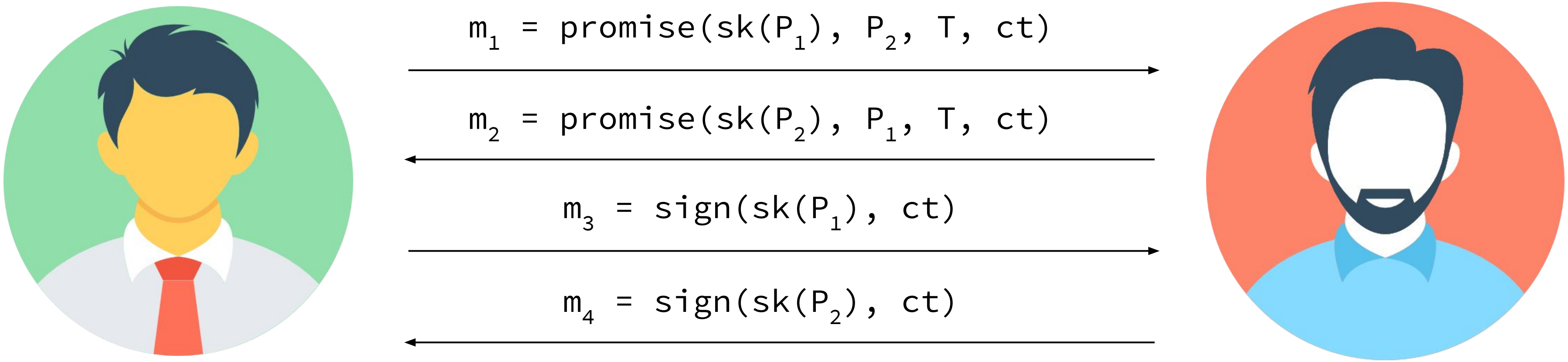
Contract Signing Protocol

Trusted Party

```
T = public key
sk(T) = private key
```

P_i = public key
$sk(P_i)$ = private key

Regular Scenario



$m_1 = promise(sk(P_1), P_2, T, ct)$

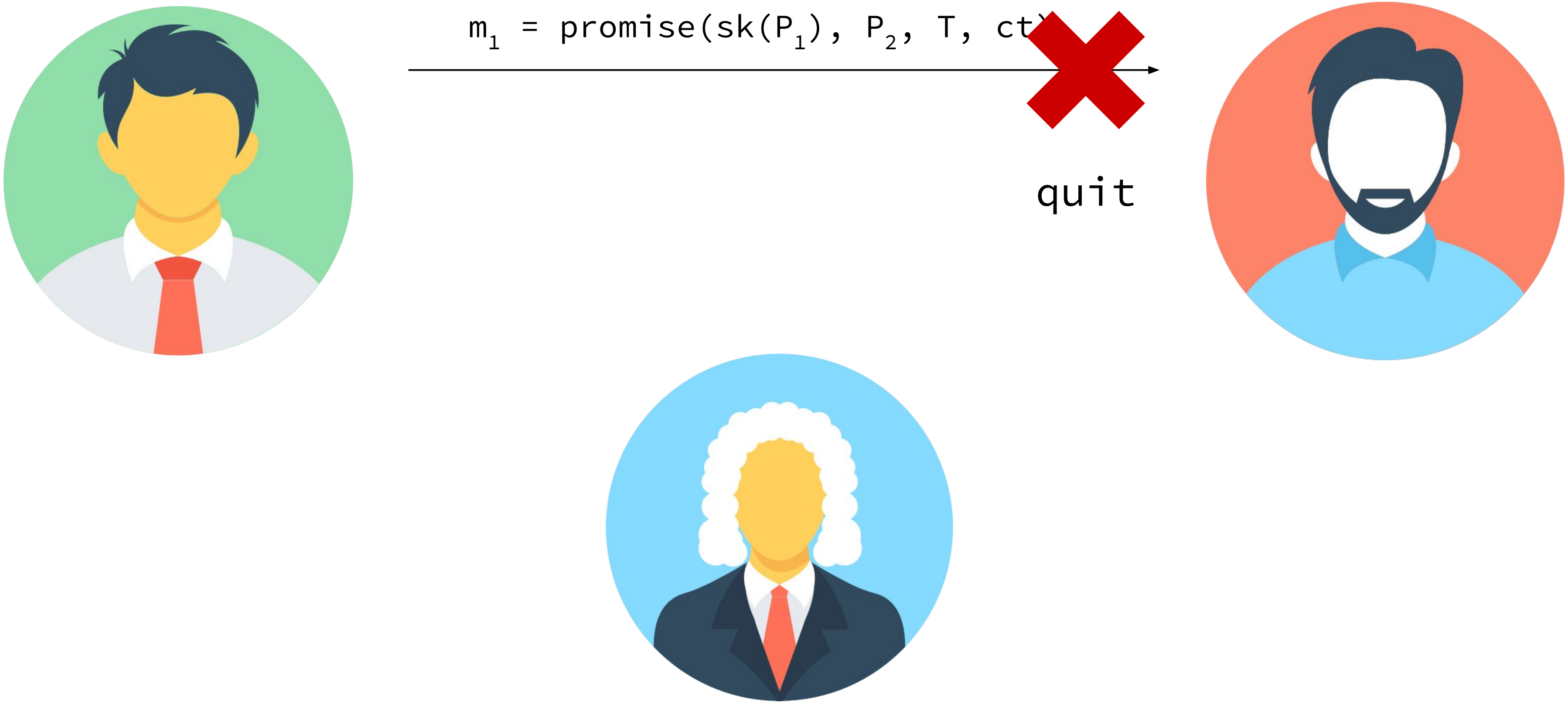
$m_2 = promise(sk(P_2), P_1, T, ct)$

$m_3 = sign(sk(P_1), ct)$

$m_4 = sign(sk(P_2), ct)$

P_i = public key
sk(P_i) = private key

Abnormal Scenario

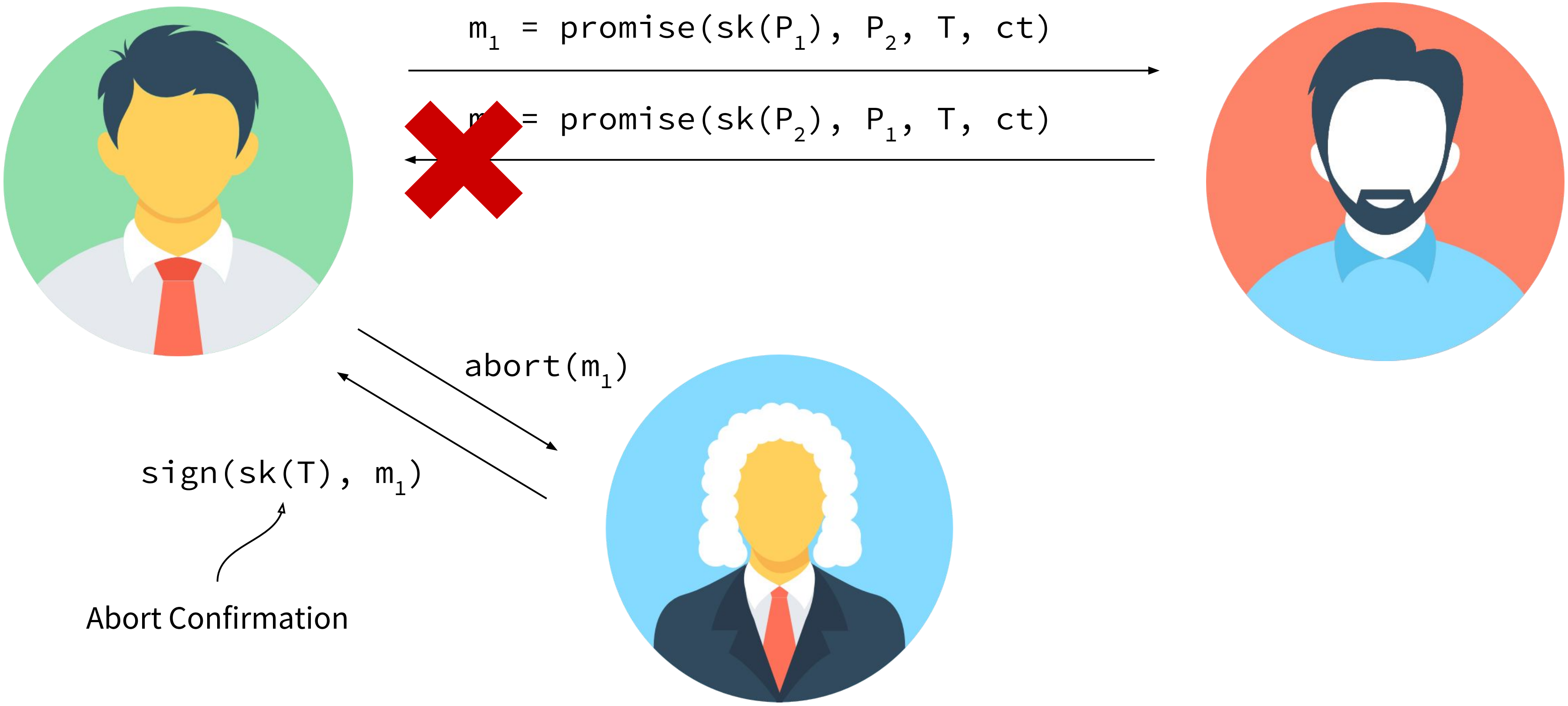


m_1 = promise(sk(P_1), P_2, T, ct)

quit

P_i = public key
$sk(P_i)$ = private key

Abnormal Scenario



m_1 = promise(sk(P_1), P_2, T, ct)

m_2 = promise(sk(P_2), P_1, T, ct)

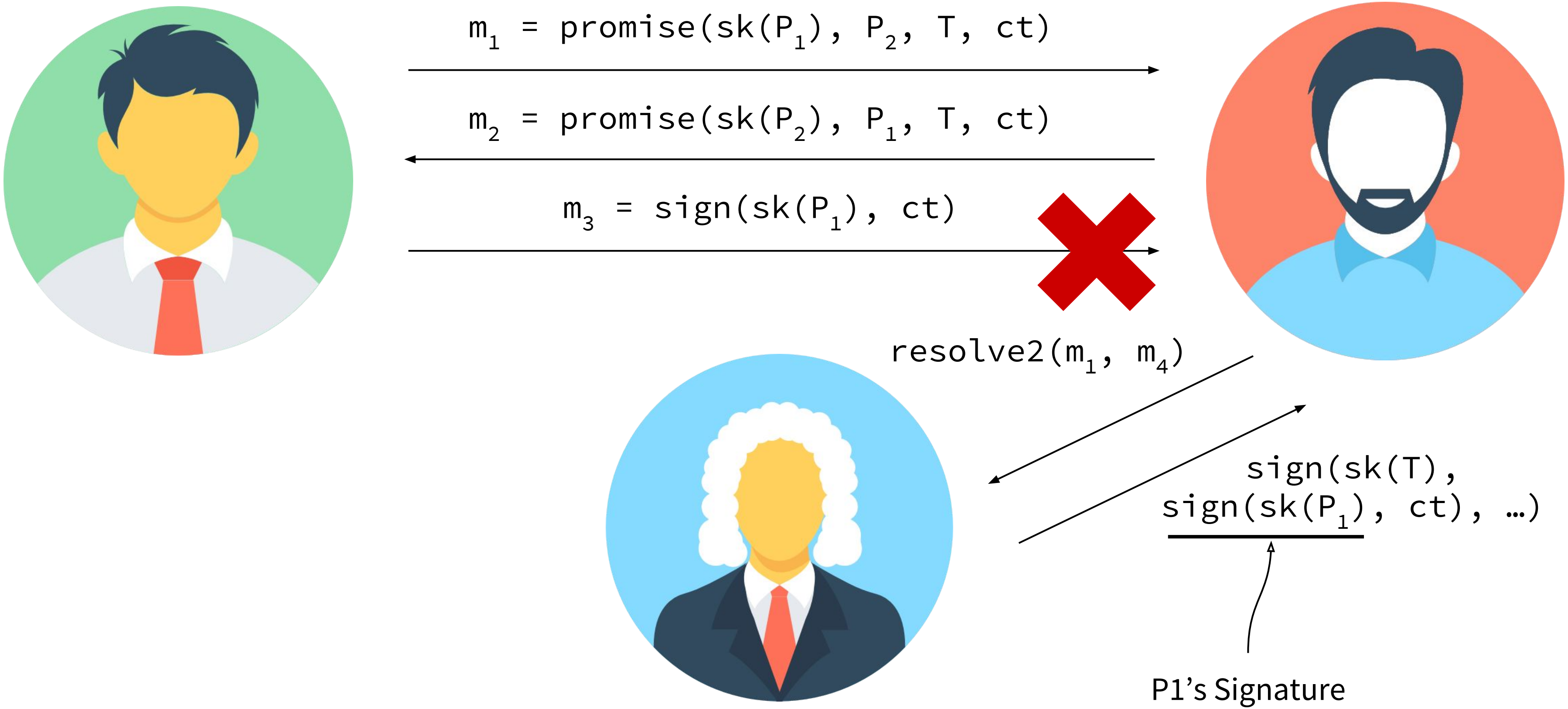
abort(m_1)

sign(sk(T), m_1)

Abort Confirmation

P_i = public key
$sk(P_i)$ = private key

Abnormal Scenario



$m_1 = \text{promise}(sk(P_1), P_2, T, ct)$

$m_2 = \text{promise}(sk(P_2), P_1, T, ct)$

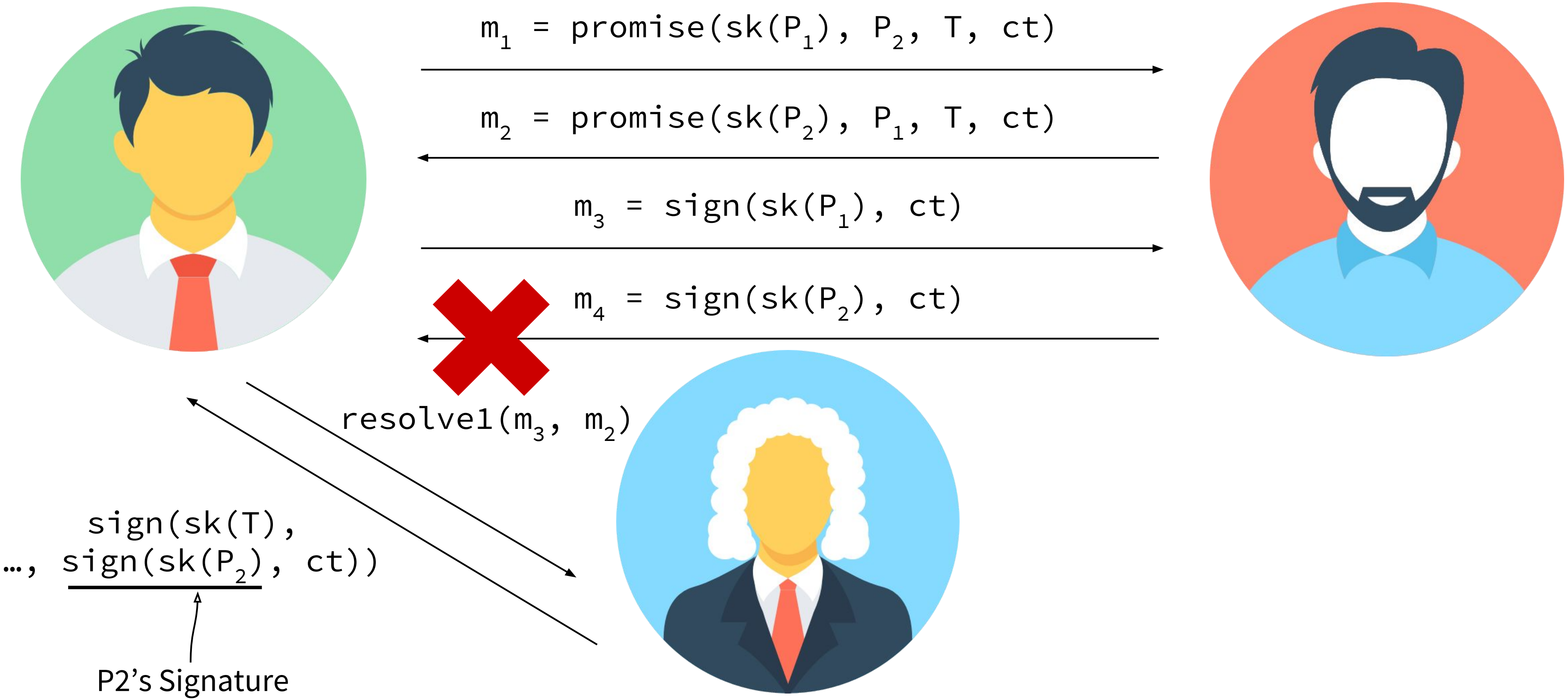
$m_3 = \text{sign}(sk(P_1), ct)$

$\text{resolve2}(m_1, m_4)$

$$\text{sign}(sk(T), \underbrace{\text{sign}(sk(P_1)}_{\text{P1's Signature}}, ct), \ldots)$$

Abnormal Scenario

P_i = public key
sk(P_i) = private key



m_1 = promise(sk(P_1), P_2, T, ct)

m_2 = promise(sk(P_2), P_1, T, ct)

m_3 = sign(sk(P_1), ct)

m_4 = sign(sk(P_2), ct)

resolve1(m_3, m_2)

sign(sk(T),
…, sign(sk(P_2), ct))

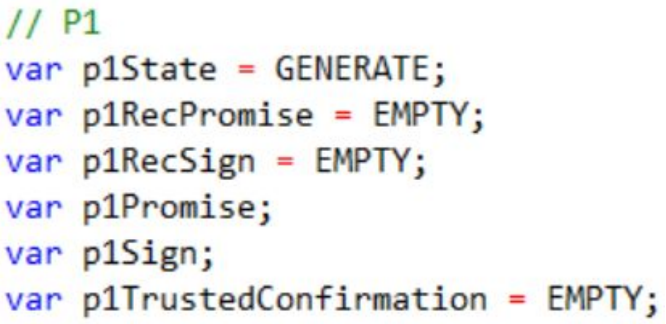
P2’s Signature

Abusefree Optimistic Contract Signing Protocol

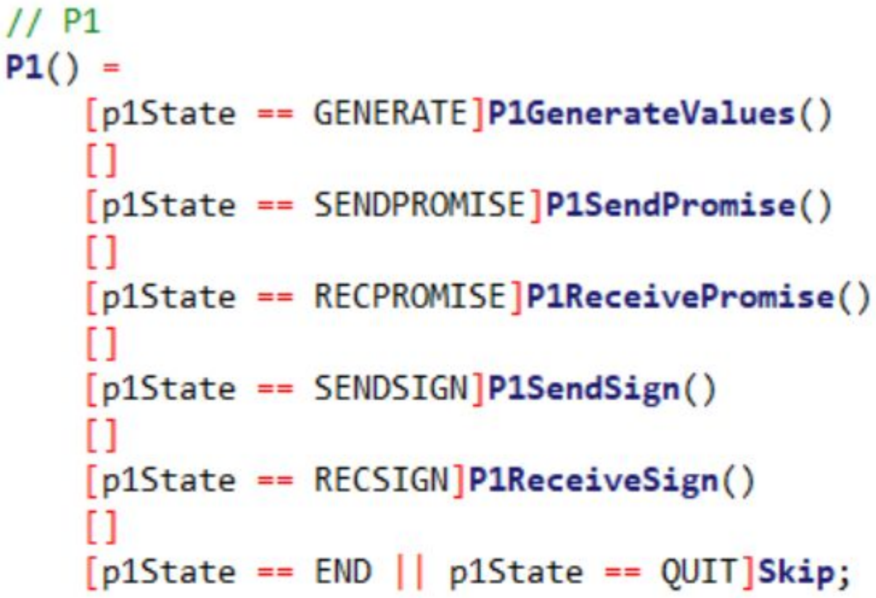
J. A. Garay, M. Jakobsson, and P. D. MacKenzie (1999)

Implementation

P1() P2()



```
// P1
var p1State = GENERATE;
var p1RecPromise = EMPTY;
var p1RecSign = EMPTY;
var p1Promise;
var p1Sign;
var p1TrustedConfirmation = EMPTY;
```



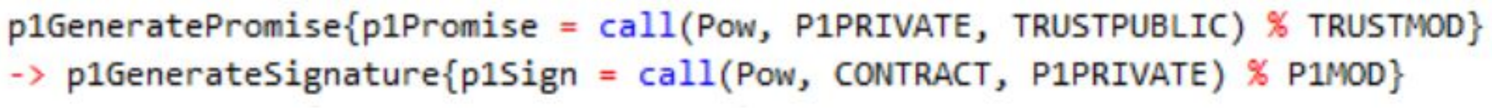
```
// P1
P1() =
    [p1State == GENERATE]P1GenerateValues()
    []
    [p1State == SENDPROMISE]P1SendPromise()
    []
    [p1State == RECPROMISE]P1ReceivePromise()
    []
    [p1State == SENDSIGN]P1SendSign()
    []
    [p1State == RECSIGN]P1ReceiveSign()
    []
    [p1State == END || p1State == QUIT]Skip;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

RSA Signing

Classroom RSA used to simulate the generation of promise and signature.

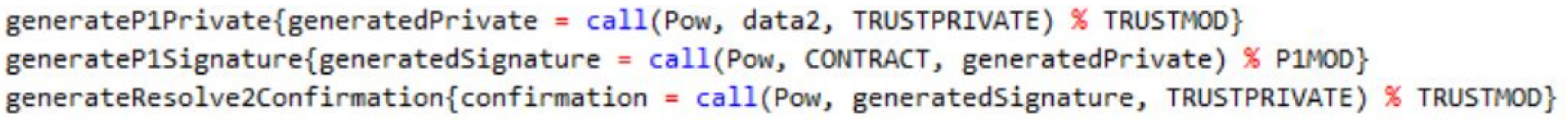
P1 or P2 promise is their private key encrypted by Trusted Party's public key.

P1 or P2 signature is the contract signed (encrypted) by their private key.



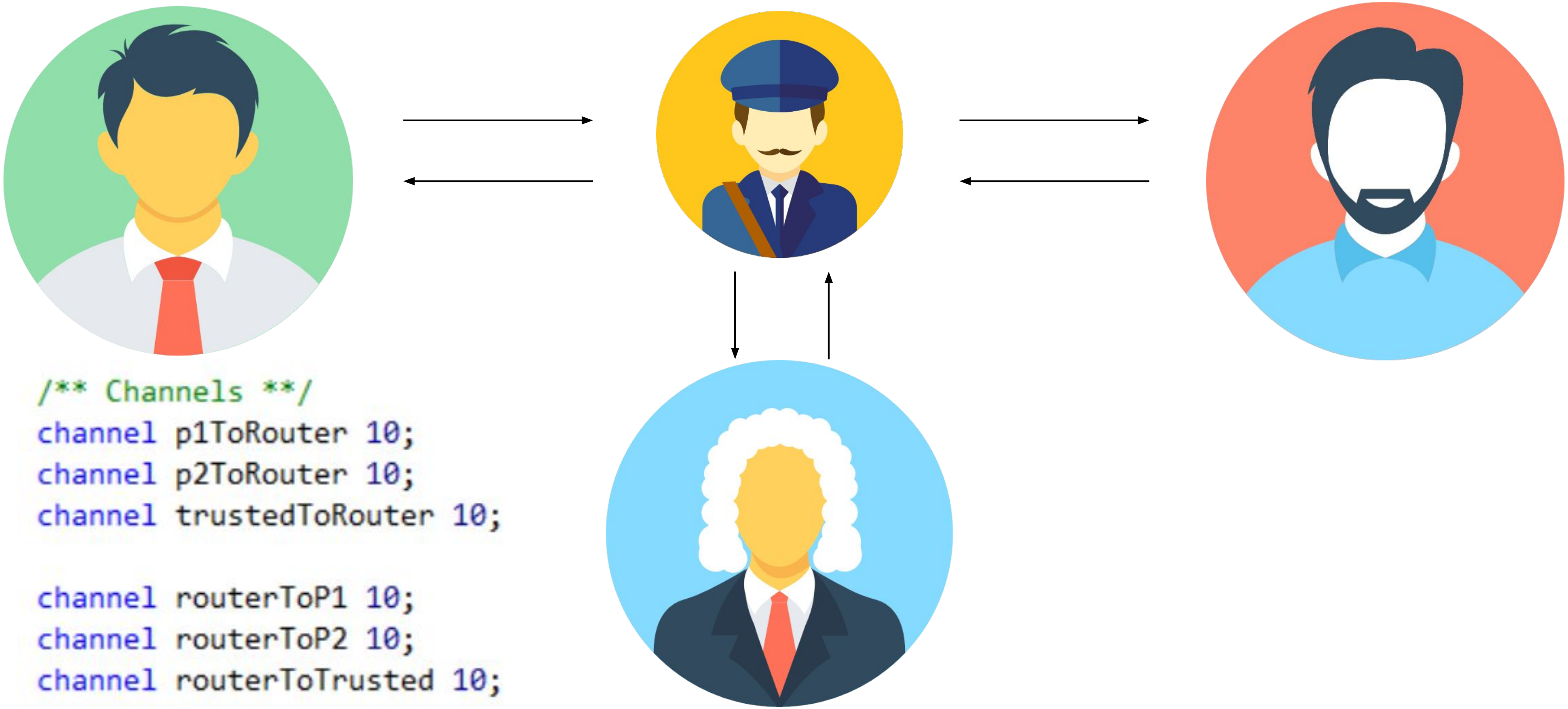
```
p1GeneratePromise{p1Promise = call(Pow, P1PRIVATE, TRUSTPUBLIC) % TRUSTMOD}
-> p1GenerateSignature{p1Sign = call(Pow, CONTRACT, P1PRIVATE) % P1MOD}
```

Trusted party can obtain P1 or P2's private key by decrypting the promise with their private key, and use it to sign the contract to form P1 or P2's signature.

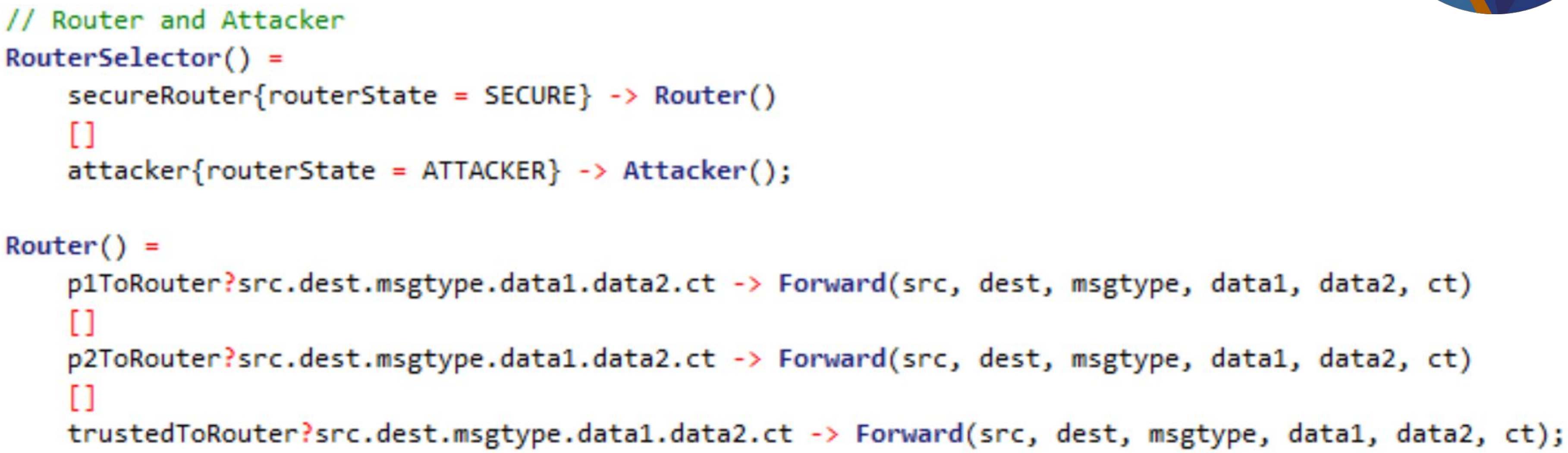


```
generateP1Private{generatedPrivate = call(Pow, data2, TRUSTPRIVATE) % TRUSTMOD}
generateP1Signature{generatedSignature = call(Pow, CONTRACT, generatedPrivate) % P1MOD}
generateResolve2Confirmation{confirmation = call(Pow, generatedSignature, TRUSTPRIVATE) % TRUSTMOD}
```

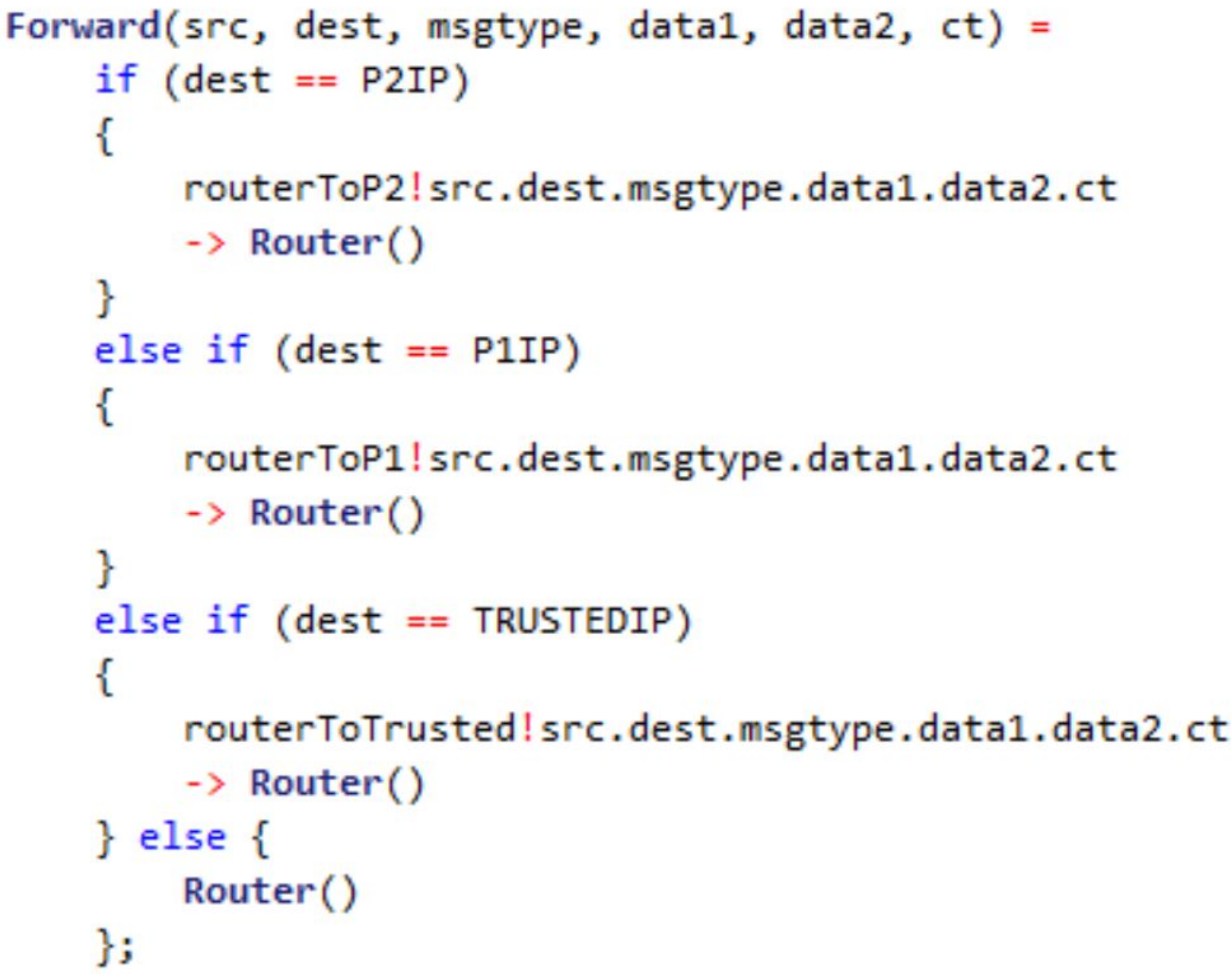
Router(): A Middleman



```
/** Channels **/
channel p1ToRouter 10;
channel p2ToRouter 10;
channel trustedToRouter 10;

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



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

Router() =
    p1ToRouter?src.dest.msgtype.data1.data2.ct -> Forward(src, dest, msgtype, data1, data2, ct)
    []
    p2ToRouter?src.dest.msgtype.data1.data2.ct -> Forward(src, dest, msgtype, data1, data2, ct)
    []
    trustedToRouter?src.dest.msgtype.data1.data2.ct -> Forward(src, dest, msgtype, data1, data2, ct);
```

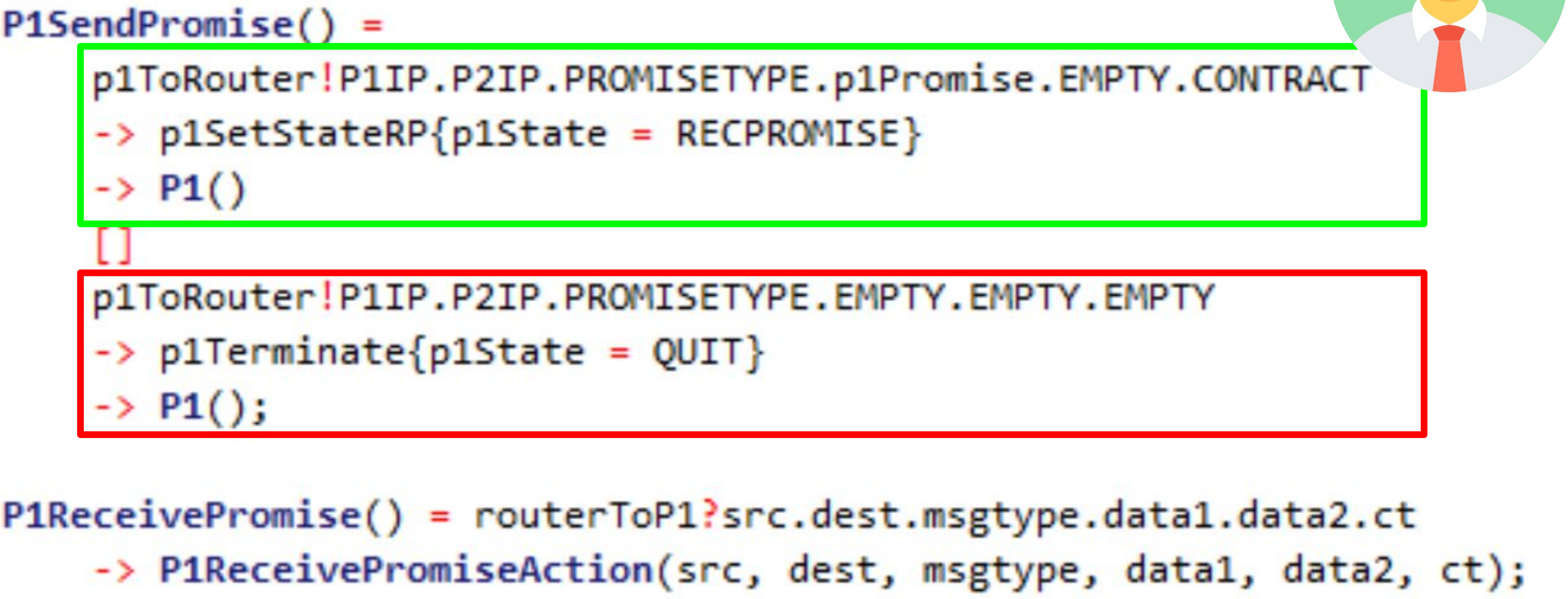


```
Forward(src, dest, msgtype, data1, data2, ct) =
    if (dest == P2IP)
    {
        routerToP2!src.dest.msgtype.data1.data2.ct
        -> Router()
    }
    else if (dest == P1IP)
    {
        routerToP1!src.dest.msgtype.data1.data2.ct
        -> Router()
    }
    else if (dest == TRUSTEDIP)
 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

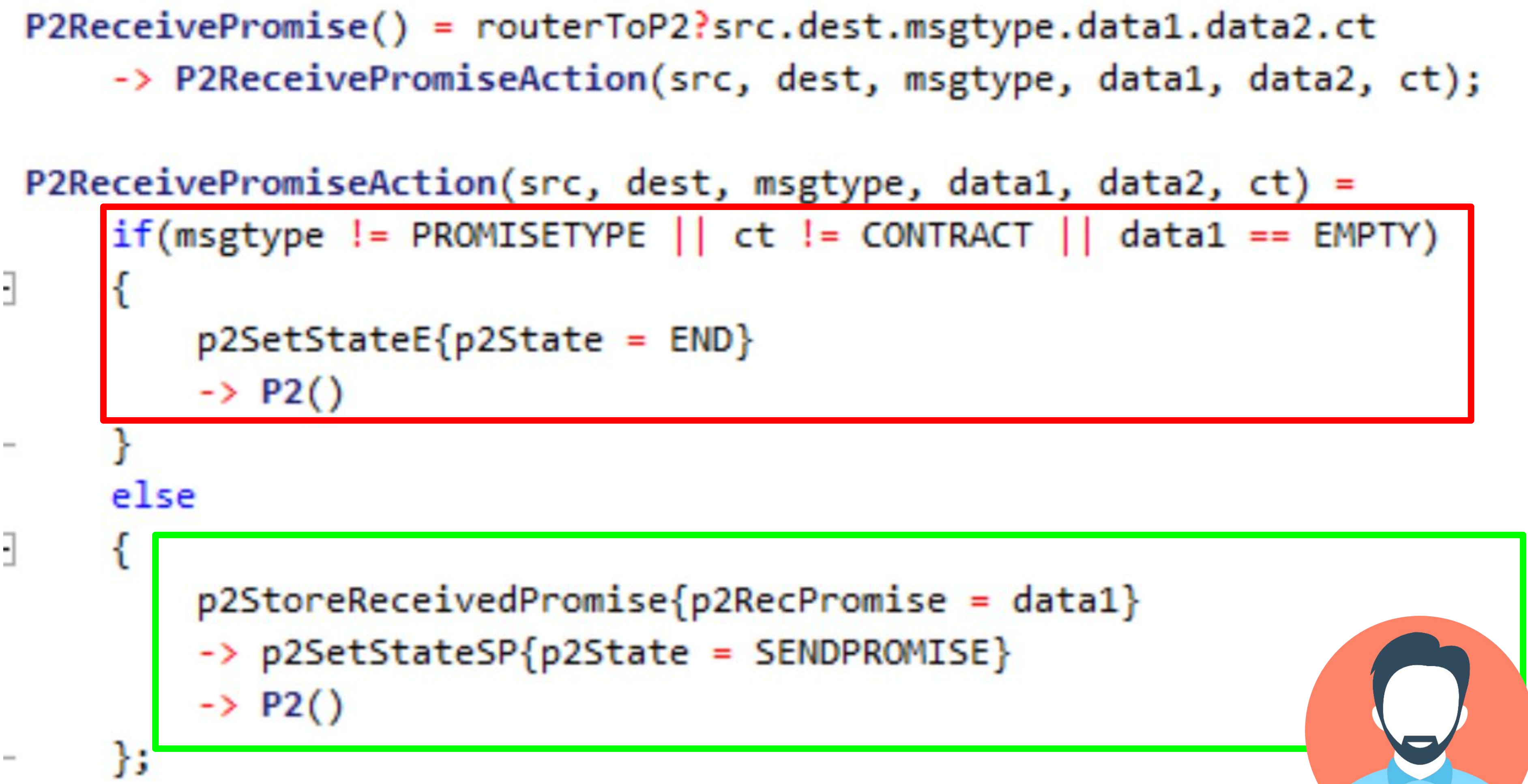
Message Encoding

Msgtype:	**Promise**	**Sign**	**Abort**	**Resolve1**	**Resolve2**	**Abort Confirmation**	**Resolve Confirmation**
field1:	Source IP						
field2:	Dest IP						
field3:	Msg Type						
field4:	Promise	Signature	P1Promise	P1Sign	p2Sign	Confirmation	Confirmation
field5:	EMPTY	EMPTY	EMPTY	P2Promise	P1Promise	EMPTY	Auth
field6:	CONTRACT						

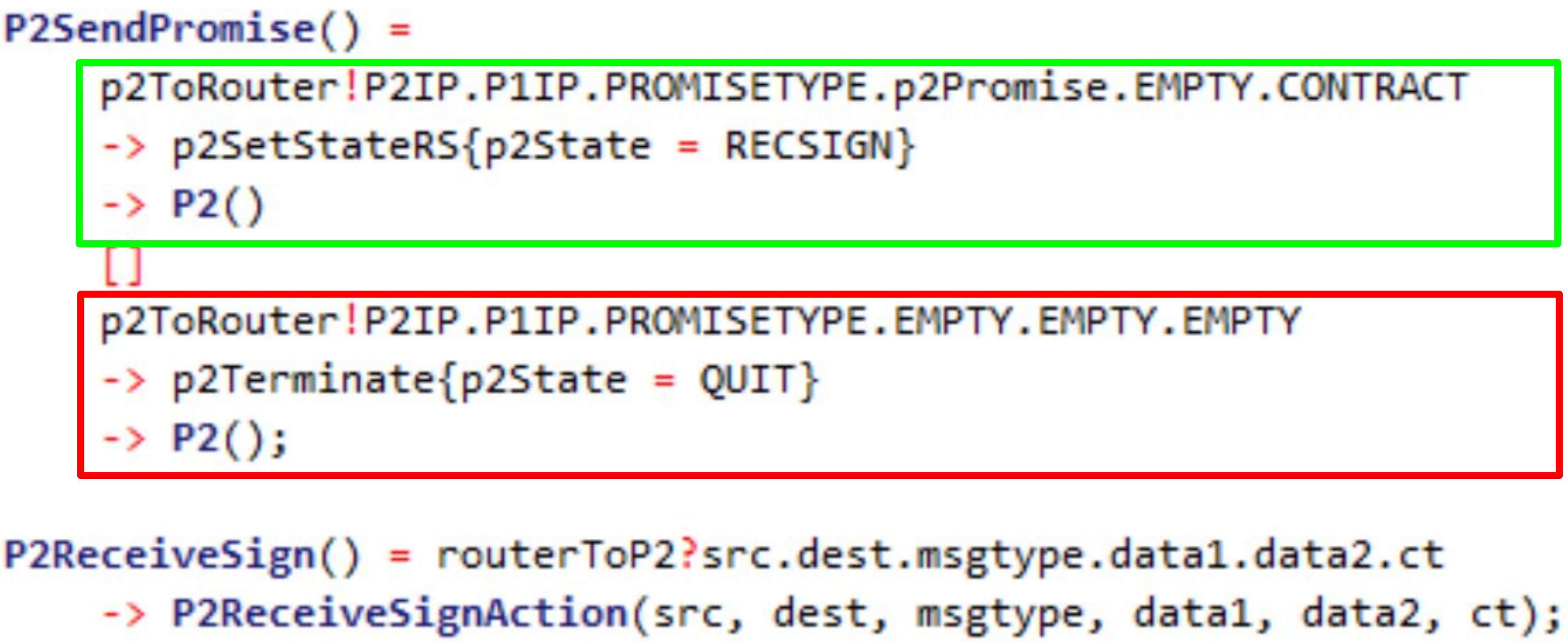
```
P1GenerateValues() =
    p1GeneratePromise{p1Promise = call(Pow, P1PRIVATE, TRUSTPUBLIC) % TRUSTMOD}
    -> p1GenerateSignature{p1Sign = call(Pow, CONTRACT, P1PRIVATE) % P1MOD}
    -> p1SetStateSP{p1State = SENDPROMISE}
    -> P1();

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



```
P1SendPromise() =
    p1ToRouter!P1IP.P2IP.PROMISETYPE.p1Promise.EMPTY.CONTRACT
    -> p1SetStateRP{p1State = RECPROMISE}
    -> P1()
    []
    p1ToRouter!P1IP.P2IP.PROMISETYPE.EMPTY.EMPTY.EMPTY
    -> p1Terminate{p1State = QUIT}
    -> P1();

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



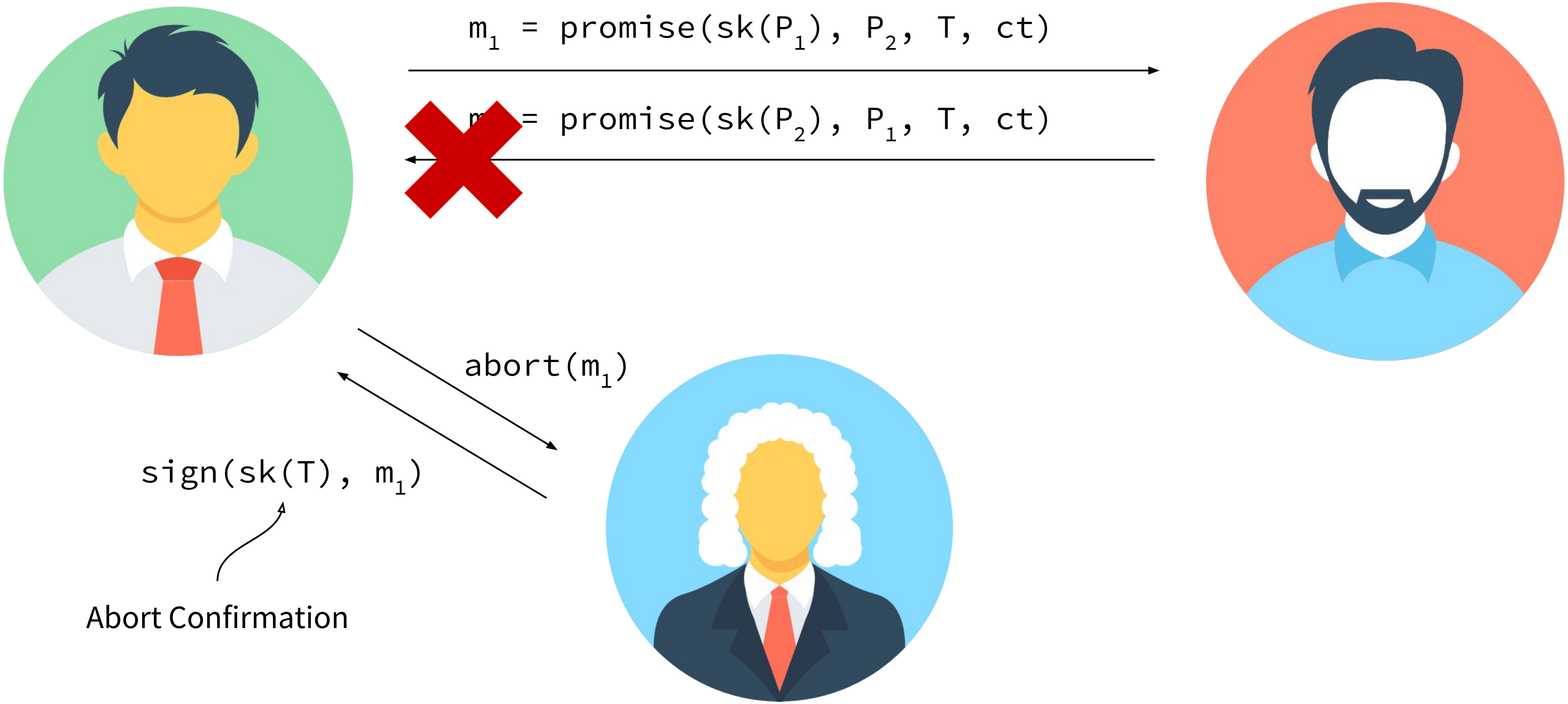
```
P2ReceivePromise() = routerToP2?src.dest.msgtype.data1.data2.ct
    -> P2ReceivePromiseAction(src, dest, msgtype, data1, data2, ct);

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()
    };
```



```
P2SendPromise() =
    p2ToRouter!P2IP.P1IP.PROMISETYPE.p2Promise.EMPTY.CONTRACT
    -> p2SetStateRS{p2State = RECSIGN}
    -> P2()
    []
    p2ToRouter!P2IP.P1IP.PROMISETYPE.EMPTY.EMPTY.EMPTY
    -> p2Terminate{p2State = QUIT}
    -> P2();

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

P_i = public key
$sk(P_i)$ = private key

Abnormal Scenario



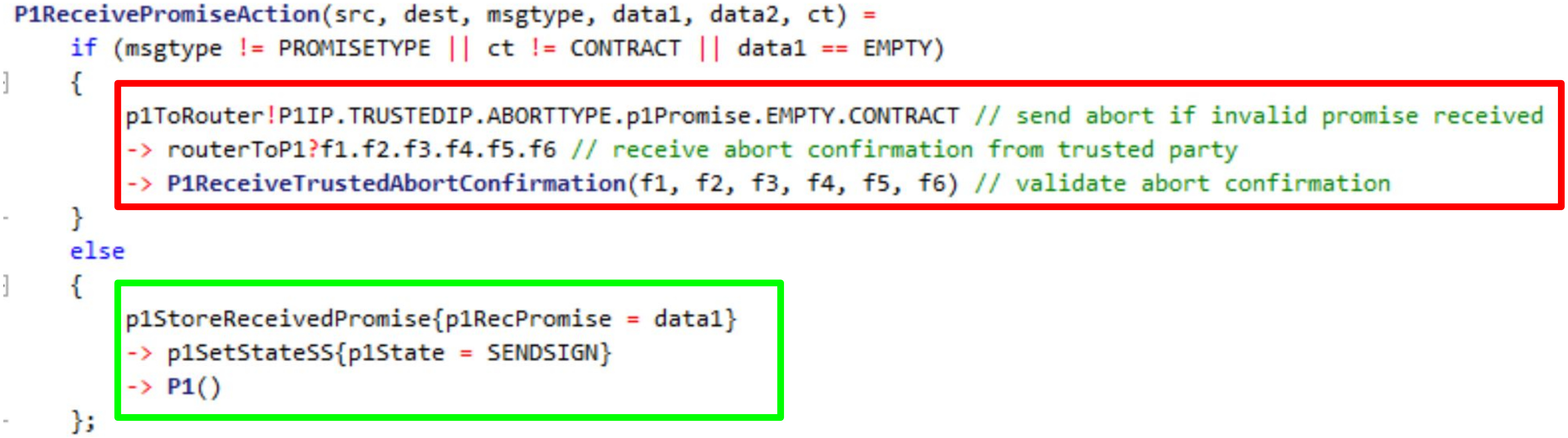
m_1 = promise(sk(P_1), P_2, T, ct)

m_2 = promise(sk(P_2), P_1, T, ct)

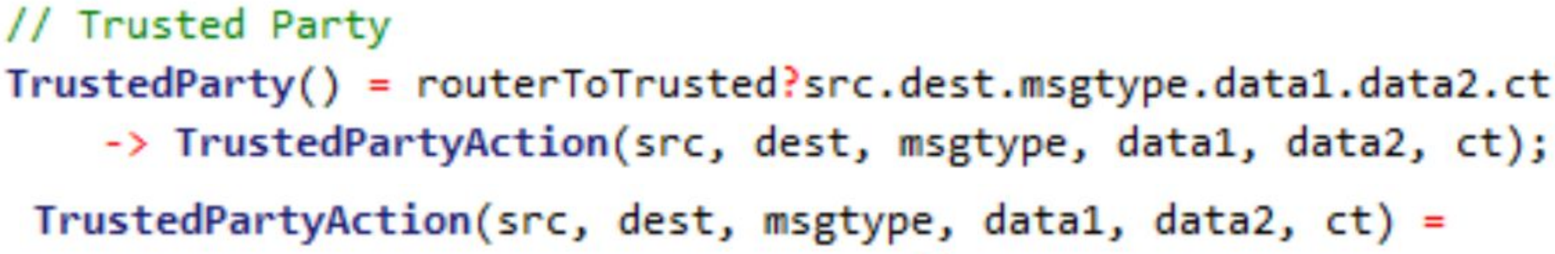
abort(m_1)

sign(sk(T), m_1)

Abort Confirmation

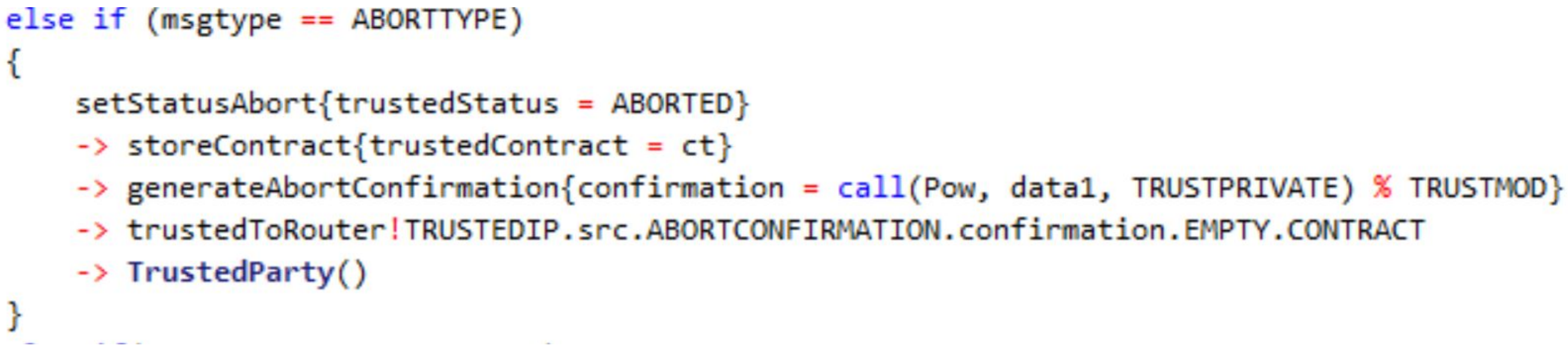


```
P1ReceivePromiseAction(src, dest, msgtype, data1, data2, ct) =
    if (msgtype != PROMISETYPE || ct != CONTRACT || data1 == EMPTY)
    {
        p1ToRouter!P1IP.TRUSTEDIP.ABORTTYPE.p1Promise.EMPTY.CONTRACT // send abort if invalid promise received
        -> routerToP1?f1.f2.f3.f4.f5.f6 // receive abort confirmation from trusted party
        -> P1ReceiveTrustedAbortConfirmation(f1, f2, f3, f4, f5, f6) // validate abort confirmation
    }
    else
    {
        p1StoreReceivedPromise{p1RecPromise = data1}
        -> p1SetStateSS{p1State = SENDSIGN}
        -> P1()
    };
```

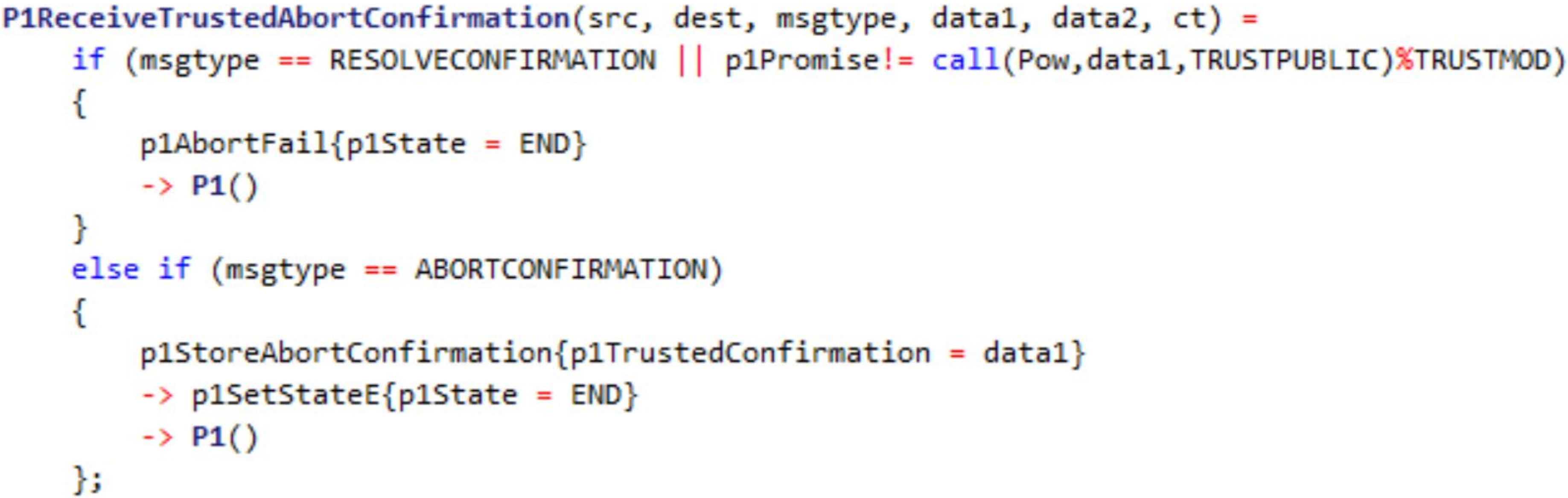


```
// Trusted Party
TrustedParty() = routerToTrusted?src.dest.msgtype.data1.data2.ct
    -> TrustedPartyAction(src, dest, msgtype, data1, data2, ct);

 TrustedPartyAction(src, dest, msgtype, data1, data2, ct) =
```

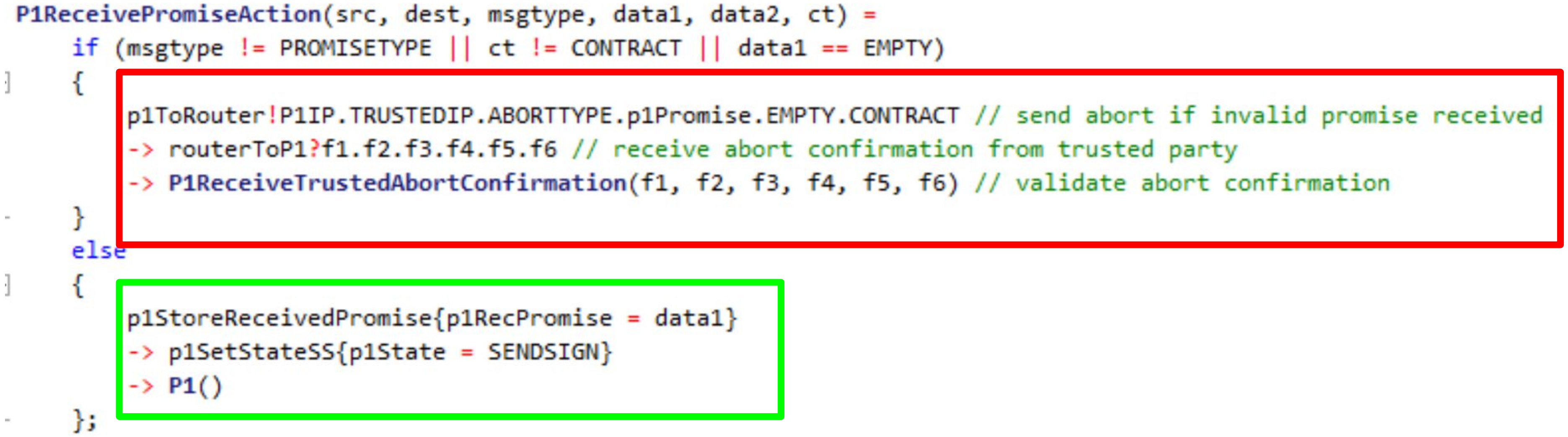
**other if conditions omitted for brevity



```
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()
}
```



```
P1ReceiveTrustedAbortConfirmation(src, dest, msgtype, data1, data2, ct) =
    if (msgtype == RESOLVECONFIRMATION || p1Promise!= call(Pow,data1,TRUSTPUBLIC)%TRUSTMOD)
    {
        p1AbortFail{p1State = END}
        -> P1()
    }
    else if (msgtype == ABORTCONFIRMATION)
    {
        p1StoreAbortConfirmation{p1TrustedConfirmation = data1}
        -> p1SetStateE{p1State = END}
        -> P1()
    };
```

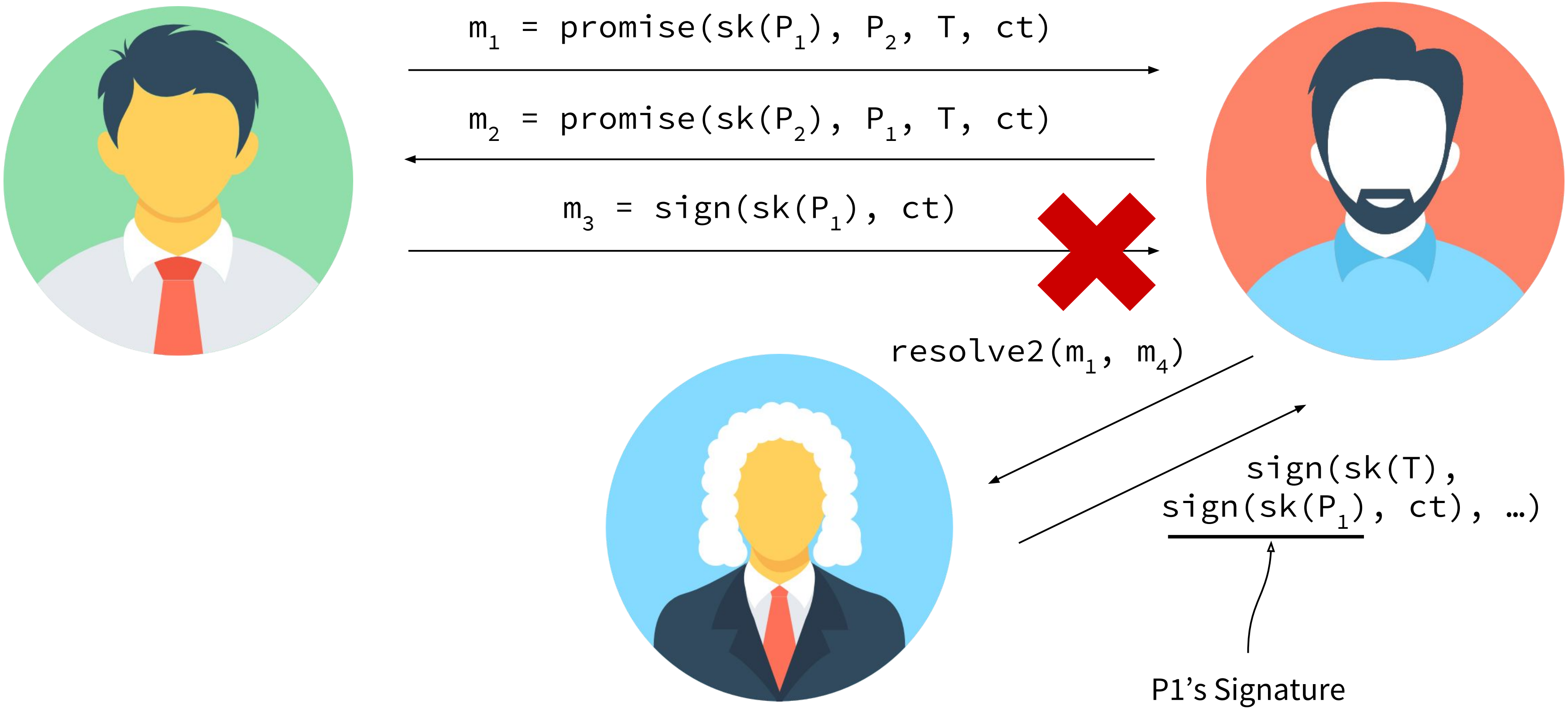


```
P1ReceivePromiseAction(src, dest, msgtype, data1, data2, ct) =
    if (msgtype != PROMISETYPE || ct != CONTRACT || data1 == EMPTY)
    {
        p1ToRouter!P1IP.TRUSTEDIP.ABORTTYPE.p1Promise.EMPTY.CONTRACT // send abort if invalid promise received
        -> routerToP1?f1.f2.f3.f4.f5.f6 // receive abort confirmation from trusted party
        -> P1ReceiveTrustedAbortConfirmation(f1, f2, f3, f4, f5, f6) // validate abort confirmation
    }
    else
    {
        p1StoreReceivedPromise{p1RecPromise = data1}
        -> p1SetStateSS{p1State = SENDSIGN}
        -> P1()
    };
```

```
P1SendSign() =
    p1ToRouter!P1IP.P2IP.SIGNTYPE.p1Sign.EMPTY.CONTRACT
    -> p1SetStateRS{p1State = RECSIGN}
    -> P1()
    []
    p1ToRouter!P1IP.P2IP.SIGNTYPE.EMPTY.EMPTY.EMPTY
    -> p1Terminate{p1State = QUIT}
    -> P1();

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

P_i = public key
$sk(P_i)$ = private key

Abnormal Scenario



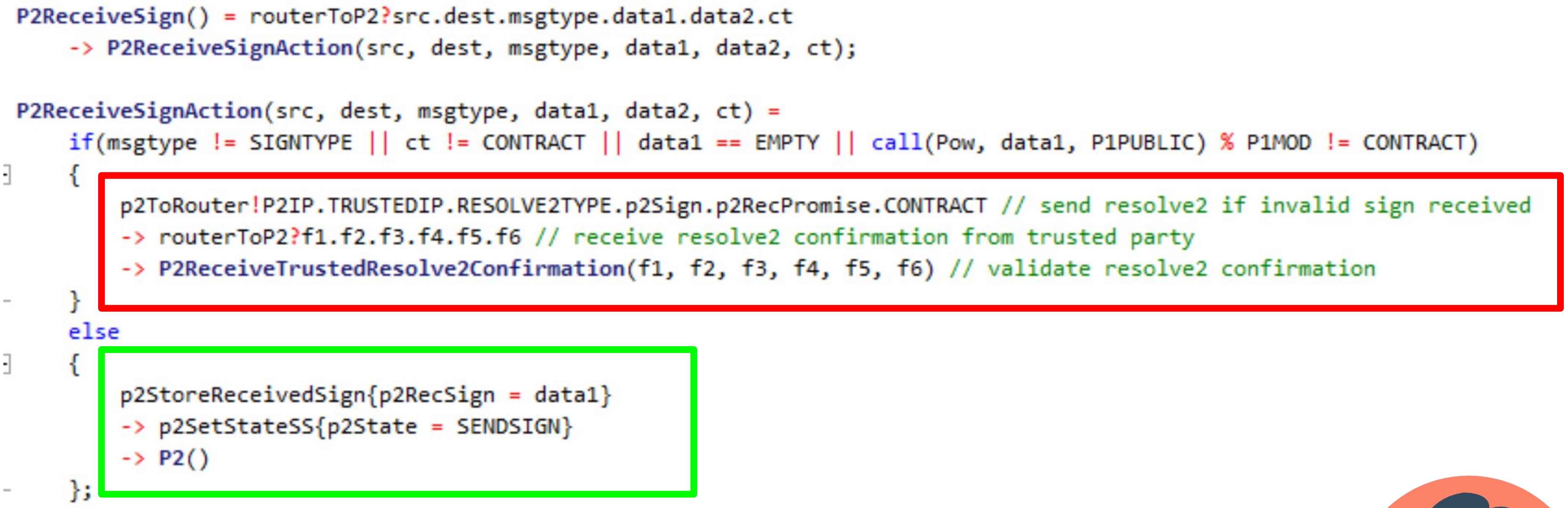
$m_1 = \mathrm{promise}(sk(P_1), P_2, T, ct)$

$m_2 = \mathrm{promise}(sk(P_2), P_1, T, ct)$

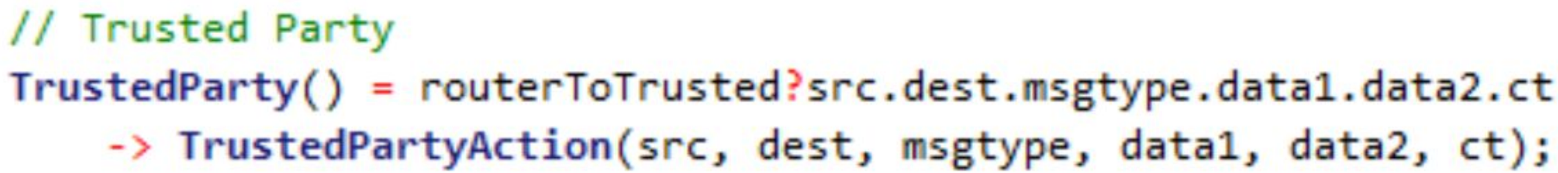
$m_3 = \mathrm{sign}(sk(P_1), ct)$

$\mathrm{resolve2}(m_1, m_4)$

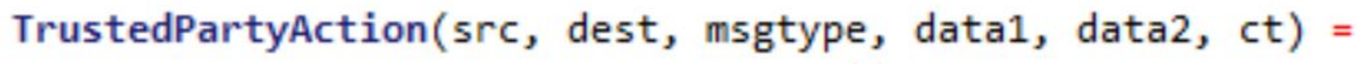
$\mathrm{sign}(sk(T), \underbrace{\mathrm{sign}(sk(P_1))}_{\text{P1's Signature}}, ct), \ldots)$



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

P2ReceiveSignAction(src, dest, msgtype, data1, data2, ct) =
    if(msgtype != SIGNTYPE || ct != CONTRACT || data1 == EMPTY || call(Pow, data1, P1PUBLIC) % P1MOD != CONTRACT)
    {
        p2ToRouter!P2IP.TRUSTEDIP.RESOLVE2TYPE.p2Sign.p2RecPromise.CONTRACT // send resolve2 if invalid sign received
        -> routerToP2?f1.f2.f3.f4.f5.f6 // receive resolve2 confirmation from trusted party
        -> P2ReceiveTrustedResolve2Confirmation(f1, f2, f3, f4, f5, f6) // validate resolve2 confirmation
    }
    else
    {
        p2StoreReceivedSign{p2RecSign = data1}
        -> p2SetStateSS{p2State = SENDSIGN}
        -> P2()
    };
```

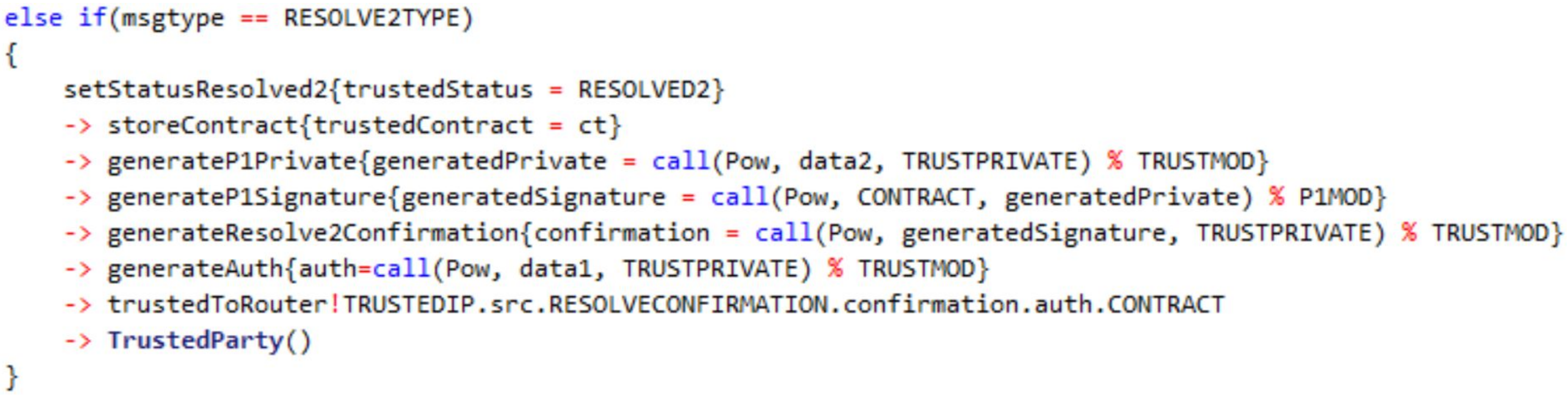


```
// Trusted Party
TrustedParty() = routerToTrusted?src.dest.msgtype.data1.data2.ct
    -> TrustedPartyAction(src, dest, msgtype, data1, data2, ct);
```

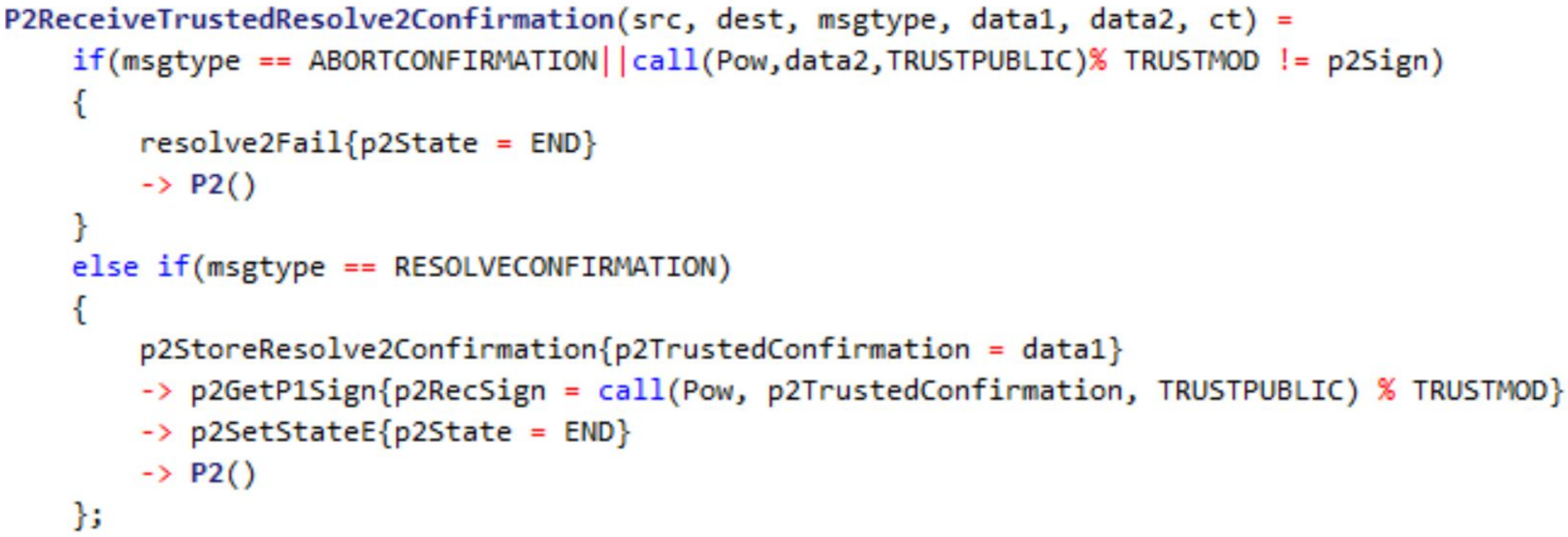


```
TrustedPartyAction(src, dest, msgtype, data1, data2, ct) =
```

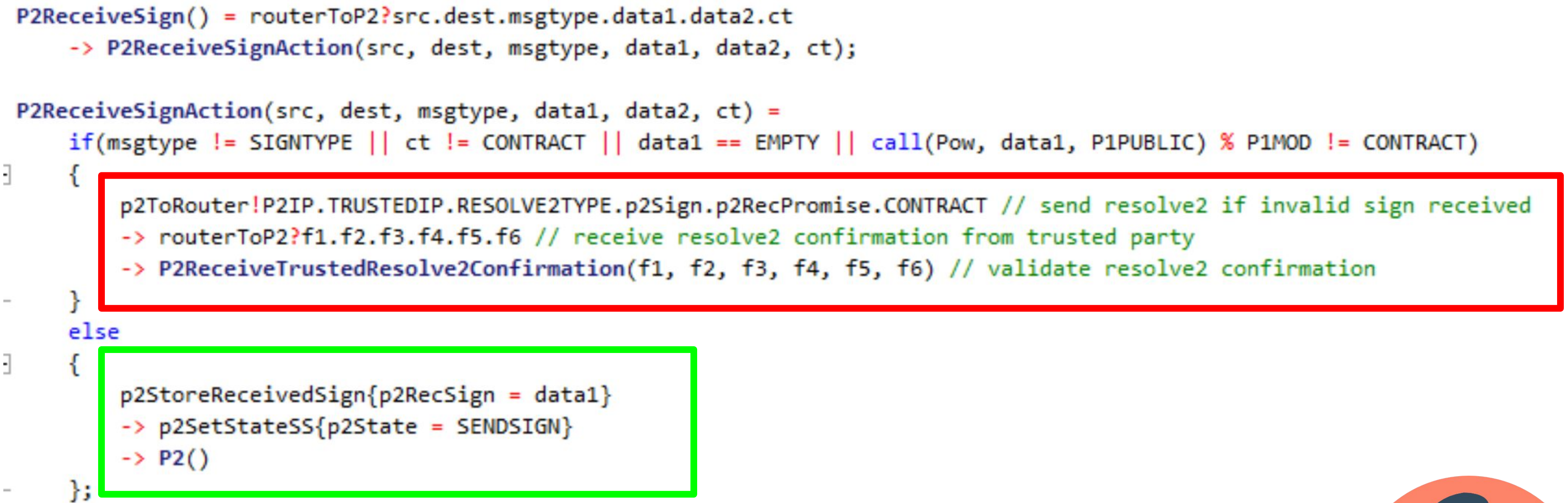
**other if conditions omitted for brevity



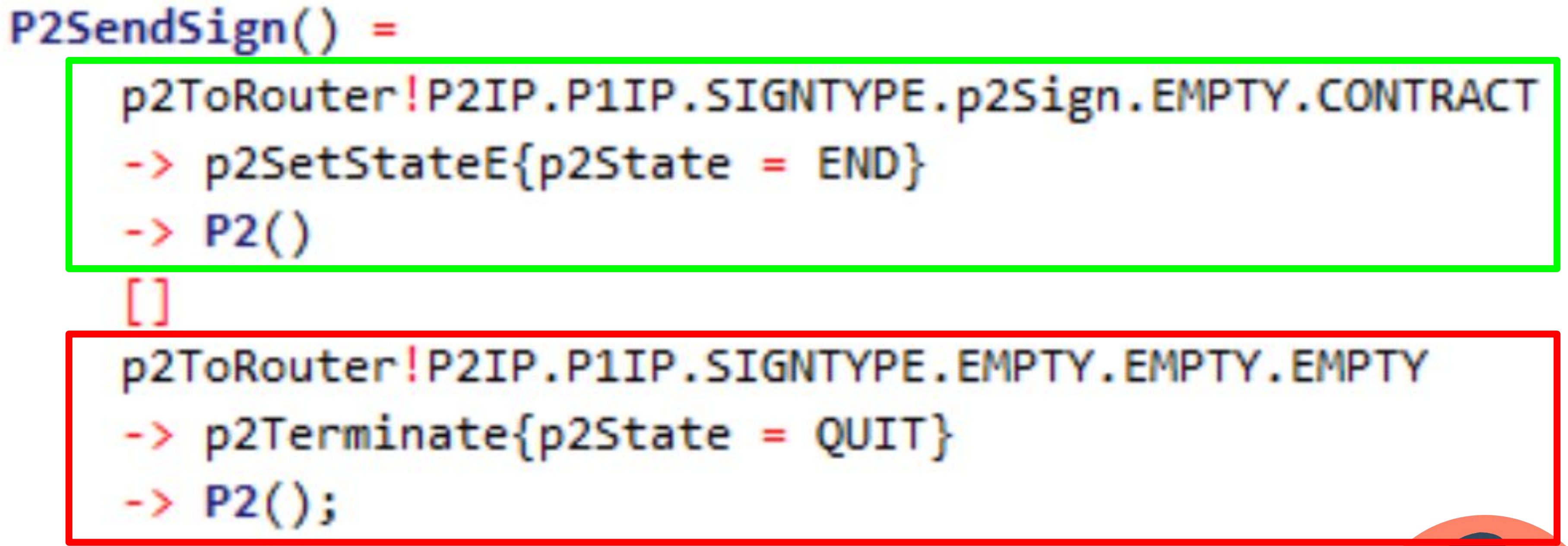
```
else if(msgtype == RESOLVE2TYPE)
{
    setStatusResolved2{trustedStatus = RESOLVED2}
    -> storeContract{trustedContract = ct}
    -> generateP1Private{generatedPrivate = call(Pow, data2, TRUSTPRIVATE) % TRUSTMOD}
    -> generateP1Signature{generatedSignature = call(Pow, CONTRACT, generatedPrivate) % P1MOD}
    -> generateResolve2Confirmation{confirmation = call(Pow, generatedSignature, TRUSTPRIVATE) % TRUSTMOD}
    -> generateAuth{auth=call(Pow, data1, TRUSTPRIVATE) % TRUSTMOD}
    -> trustedToRouter!TRUSTEDIP.src.RESOLVECONFIRMATION.confirmation.auth.CONTRACT
    -> TrustedParty()
}
```



```
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()
    };
```



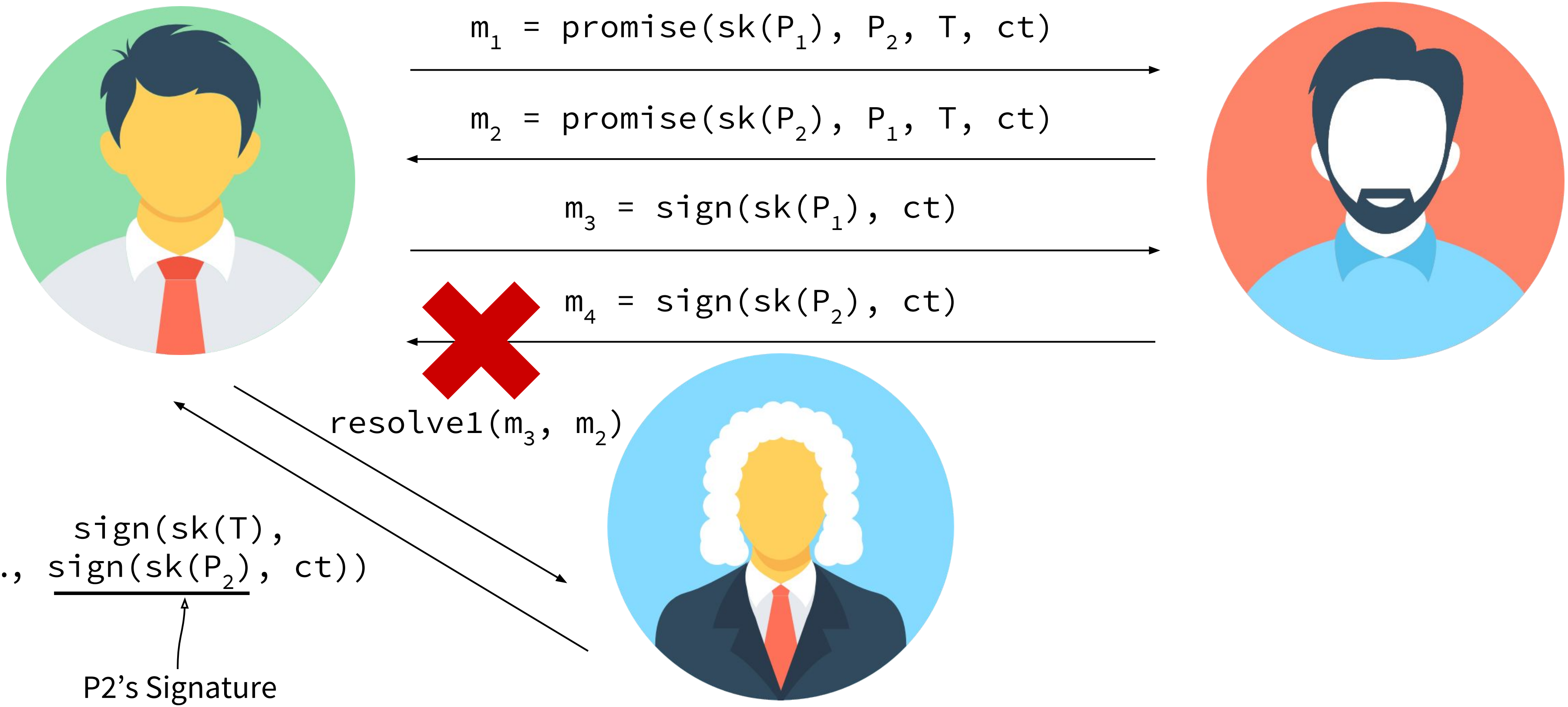
```
P2ReceiveSign() = routerToP2?src.dest.msgtype.data1.data2.ct
    -> P2ReceiveSignAction(src, dest, msgtype, data1, data2, ct);

P2ReceiveSignAction(src, dest, msgtype, data1, data2, ct) =
    if(msgtype != SIGNTYPE || ct != CONTRACT || data1 == EMPTY || call(Pow, data1, P1PUBLIC) % P1MOD != CONTRACT)
    {
        p2ToRouter!P2IP.TRUSTEDIP.RESOLVE2TYPE.p2Sign.p2RecPromise.CONTRACT // send resolve2 if invalid sign received
        -> routerToP2?f1.f2.f3.f4.f5.f6 // receive resolve2 confirmation from trusted party
        -> P2ReceiveTrustedResolve2Confirmation(f1, f2, f3, f4, f5, f6) // validate resolve2 confirmation
    }
    else
    {
        p2StoreReceivedSign{p2RecSign = data1}
        -> p2SetStateSS{p2State = SENDSIGN}
        -> P2()
    };
```



```
P2SendSign() =
    p2ToRouter!P2IP.P1IP.SIGNTYPE.p2Sign.EMPTY.CONTRACT
    -> p2SetStateE{p2State = END}
    -> P2()
    []
    p2ToRouter!P2IP.P1IP.SIGNTYPE.EMPTY.EMPTY.EMPTY
    -> p2Terminate{p2State = QUIT}
    -> P2();
```

P_i = public key
$sk(P_i)$ = private key

Abnormal Scenario



$m_1 = \text{promise}(sk(P_1), P_2, T, ct)$

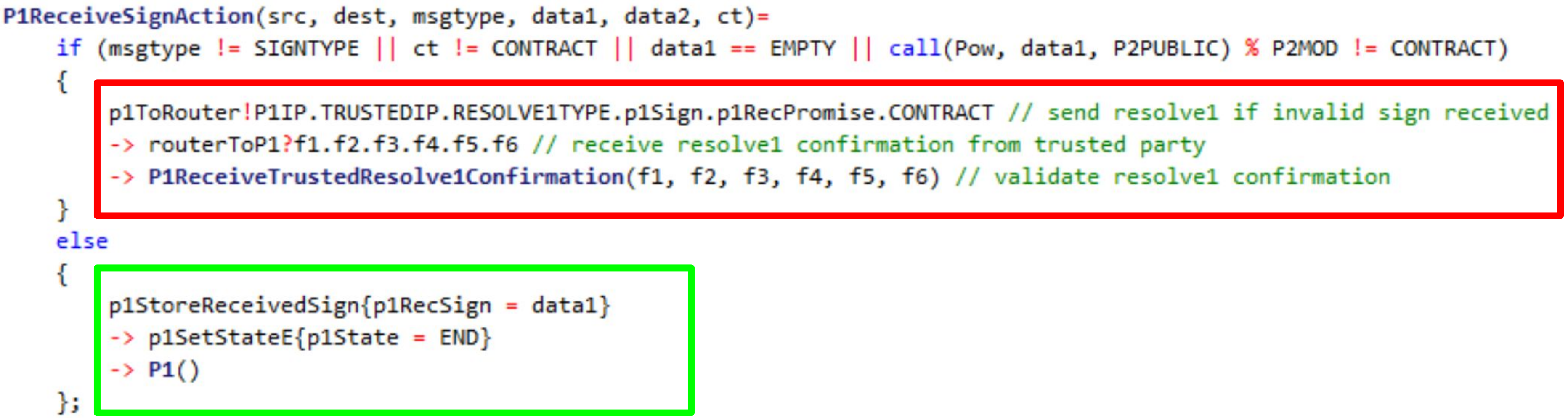
$m_2 = \text{promise}(sk(P_2), P_1, T, ct)$

$m_3 = \text{sign}(sk(P_1), ct)$

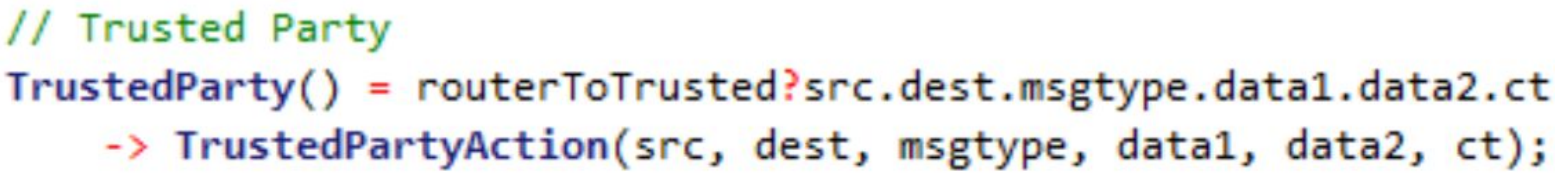
$m_4 = \text{sign}(sk(P_2), ct)$

$\text{resolve1}(m_3, m_2)$

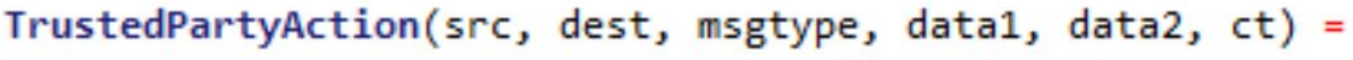
$\text{sign}(sk(T), \ldots, \underbrace{\text{sign}(sk(P_2), ct)}_{\text{P2's Signature}})$



```
P1ReceiveSignAction(src, dest, msgtype, data1, data2, ct)=
    if (msgtype != SIGNTYPE || ct != CONTRACT || data1 == EMPTY || call(Pow, data1, P2PUBLIC) % P2MOD != CONTRACT)
    {
        p1ToRouter!P1IP.TRUSTEDIP.RESOLVE1TYPE.p1Sign.p1RecPromise.CONTRACT // send resolve1 if invalid sign received
        -> routerToP1?f1.f2.f3.f4.f5.f6 // receive resolve1 confirmation from trusted party
        -> P1ReceiveTrustedResolve1Confirmation(f1, f2, f3, f4, f5, f6) // validate resolve1 confirmation
    }
    else
    {
        p1StoreReceivedSign{p1RecSign = data1}
        -> p1SetStateE{p1State = END}
        -> P1()
    };
```

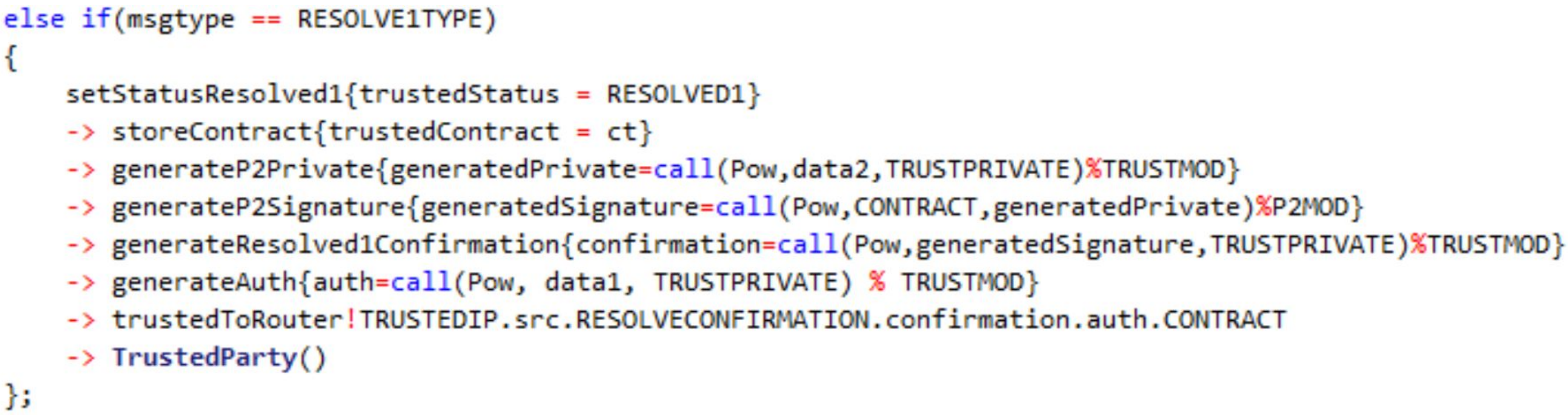


```
// Trusted Party
TrustedParty() = routerToTrusted?src.dest.msgtype.data1.data2.ct
    -> TrustedPartyAction(src, dest, msgtype, data1, data2, ct);
```

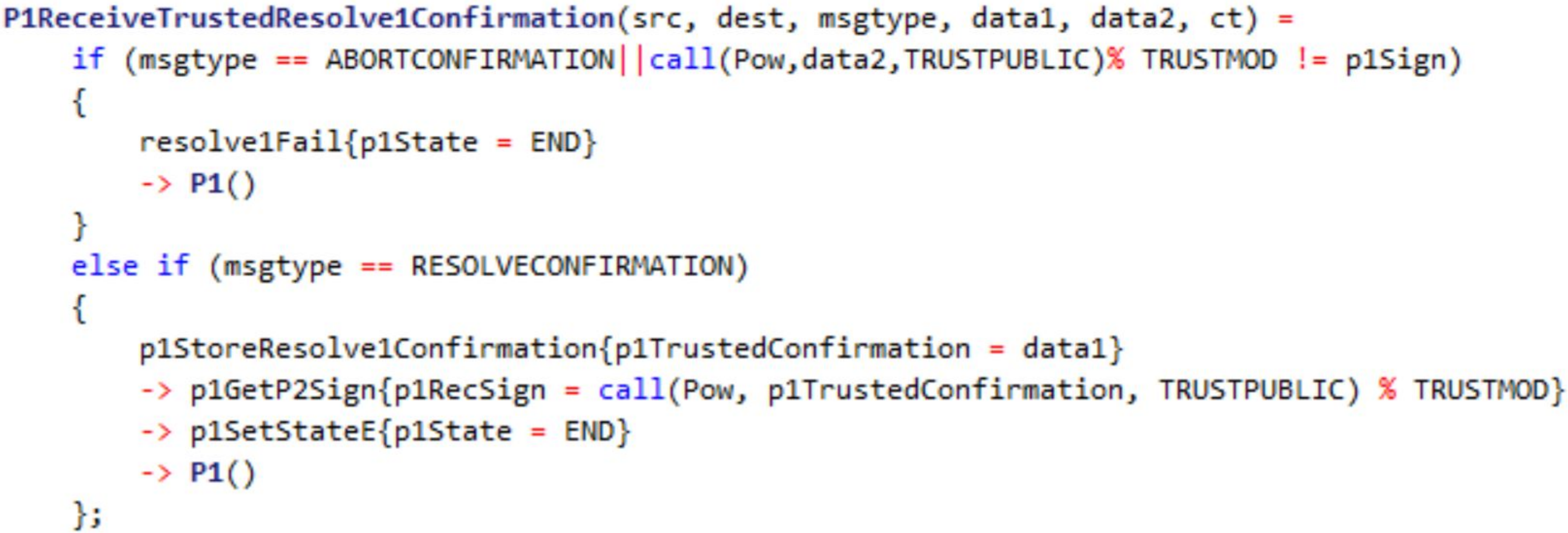


```
TrustedPartyAction(src, dest, msgtype, data1, data2, ct) =
```

**other if conditions omitted for brevity



```
else if(msgtype == RESOLVE1TYPE)
{
    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}
    -> trustedToRouter!TRUSTEDIP.src.RESOLVECONFIRMATION.confirmation.auth.CONTRACT
    -> TrustedParty()
};
```

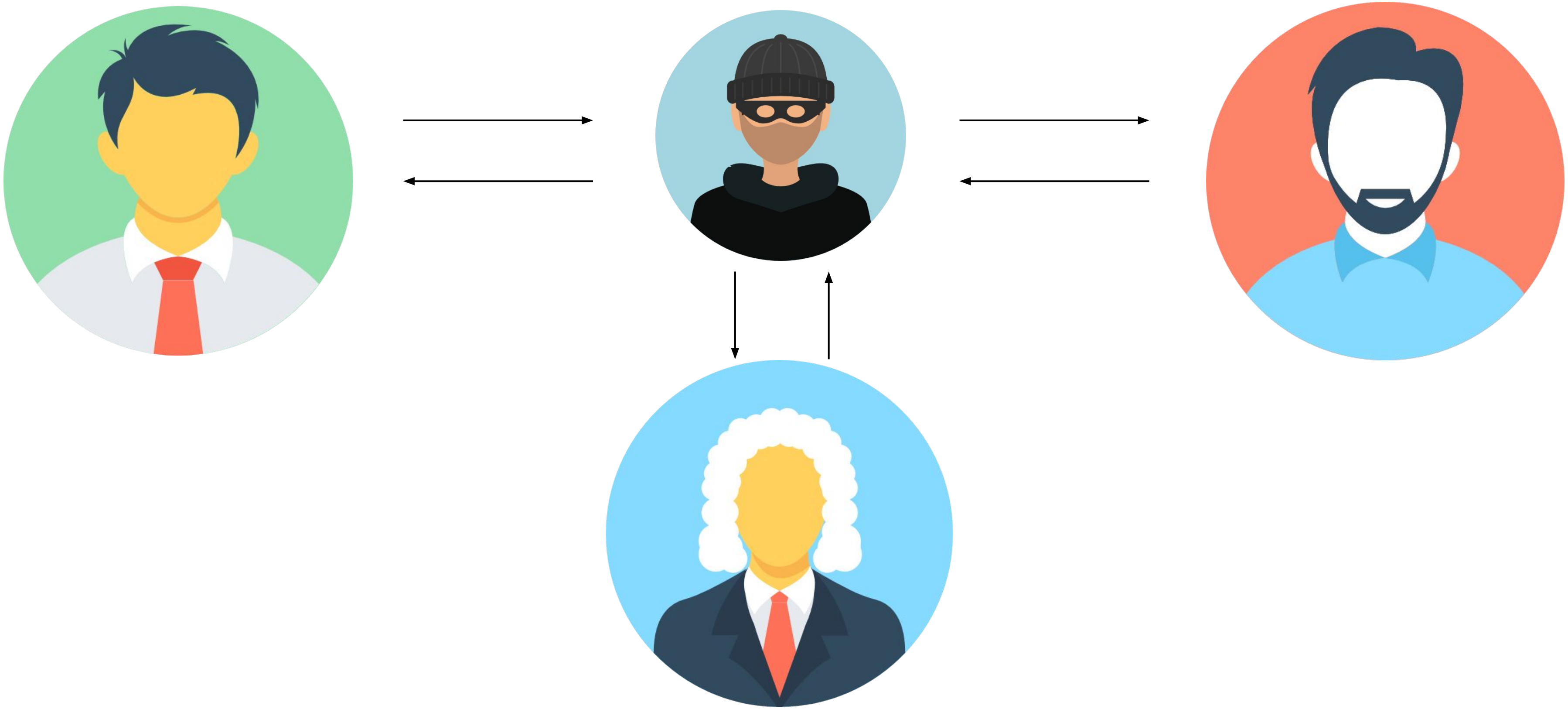


```
P1ReceiveTrustedResolve1Confirmation(src, dest, msgtype, data1, data2, ct) =
    if (msgtype == ABORTCONFIRMATION||call(Pow,data2,TRUSTPUBLIC)% TRUSTMOD != p1Sign)
    {
        resolve1Fail{p1State = END}
        -> P1()
    }
    else if (msgtype == RESOLVECONFIRMATION)
    {
        p1StoreResolve1Confirmation{p1TrustedConfirmation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

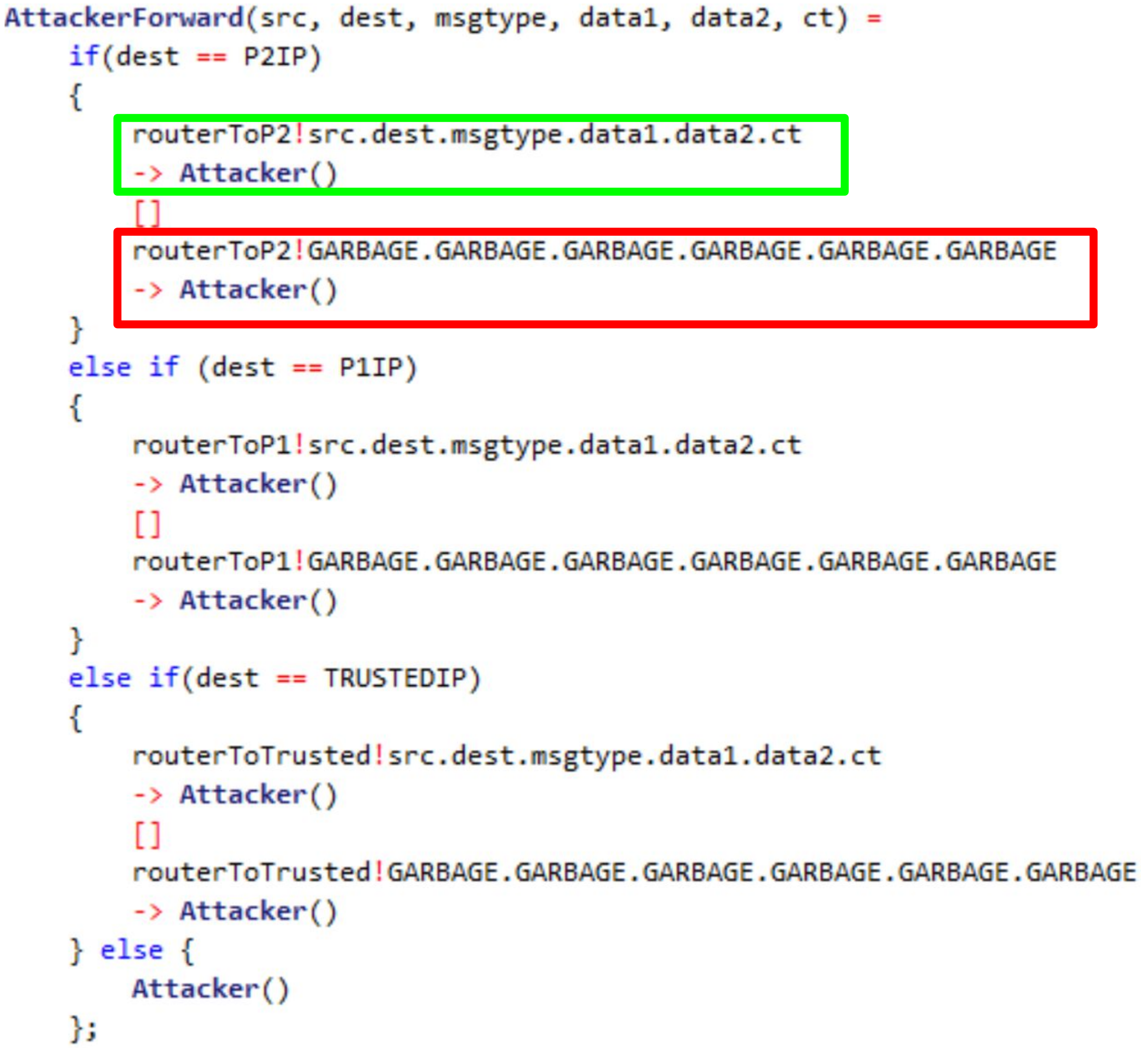
Attacker

All the messages are sent over public network. That is, there is an attacker controlling the network, and can block, inject, alter and read messages over the network.

Attacker(): A Compromised Router

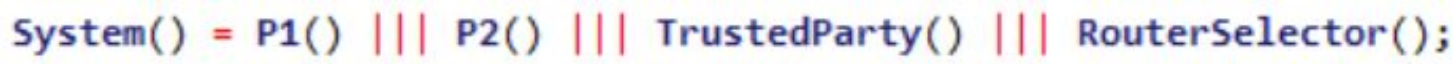


```
// Router and Attacker
RouterSelector() =
    secureRouter{routerState = SECURE} -> Router()
    []
    attacker{routerState = ATTACKER} -> Attacker();

Router() =
    p1ToRouter?src.dest.msgtype.data1.data2.ct -> Forward(src, dest, msgtype, data1, data2, ct)
    []
    p2ToRouter?src.dest.msgtype.data1.data2.ct -> Forward(src, dest, msgtype, data1, data2, ct)
    []
    trustedToRouter?src.dest.msgtype.data1.data2.ct -> Forward(src, dest, msgtype, data1, data2, ct);

Attacker() =
    p1ToRouter?src.dest.msgtype.data1.data2.ct -> AttackerForward(src, dest, msgtype, data1, data2, ct)
    []
    p2ToRouter?src.dest.msgtype.data1.data2.ct -> AttackerForward(src, dest, msgtype, data1, data2, ct)
    []
    trustedToRouter?src.dest.msgtype.data1.data2.ct -> AttackerForward(src, dest, msgtype, data1, data2, ct);
```



```
AttackerForward(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()
    };
```

Full System

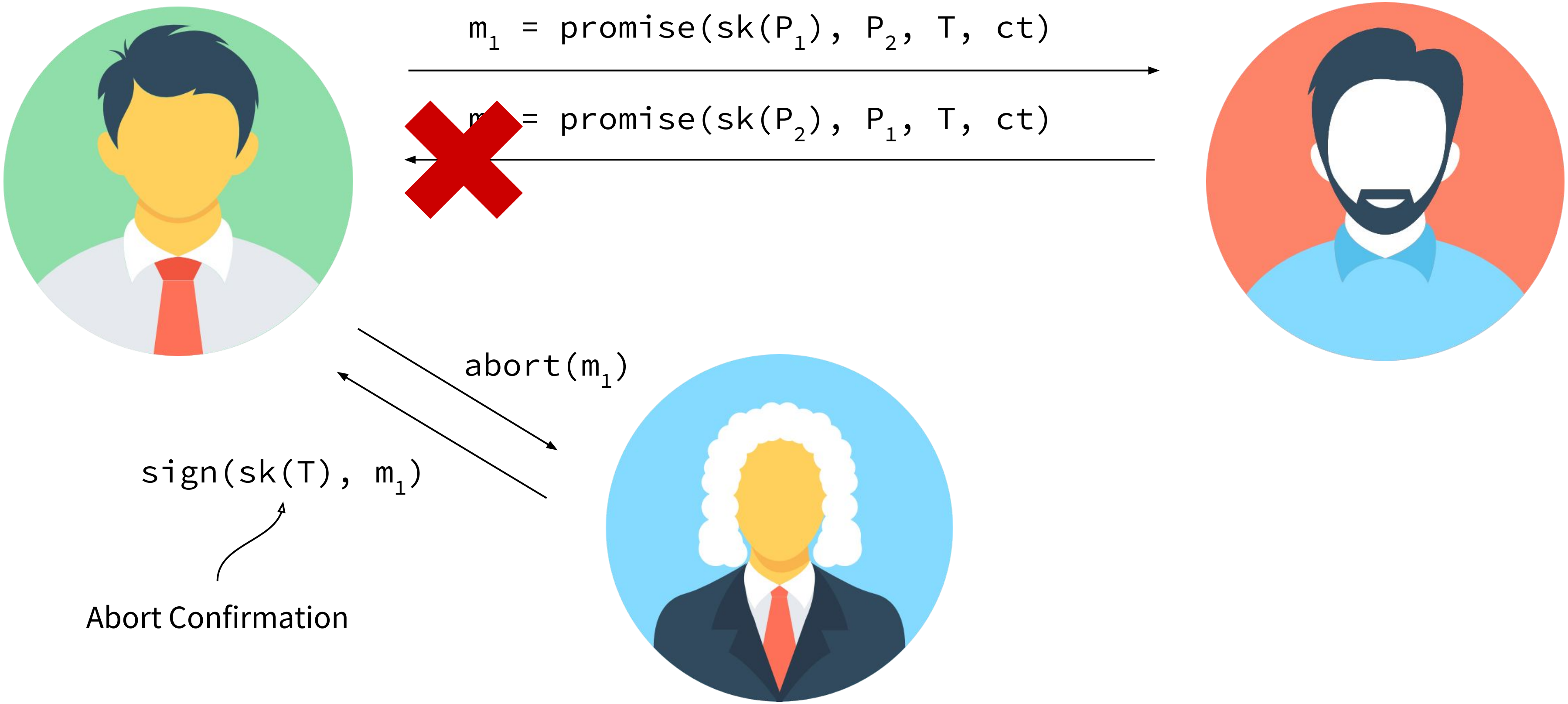


```
System() = P1() ||| P2() ||| TrustedParty() ||| RouterSelector();
```

Demo

P_i = public key
$sk(P_i)$ = private key

Abnormal Scenario



m_1 = promise(sk(P_1), P_2, T, ct)

m_2 = promise(sk(P_2), P_1, T, ct)

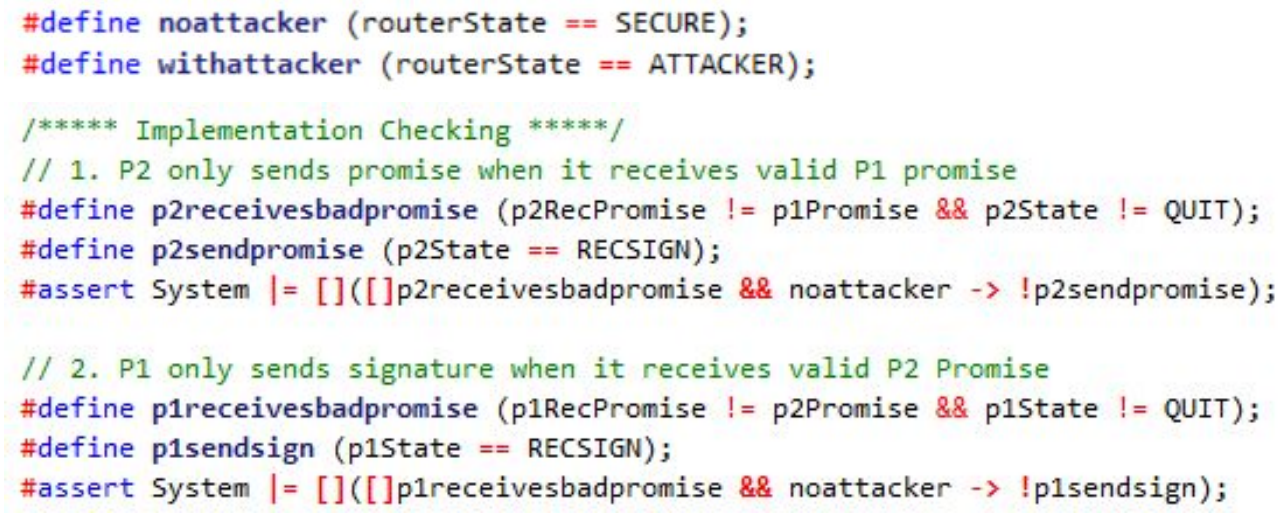
abort(m_1)

sign(sk(T), m_1)

Abort Confirmation

Assertions

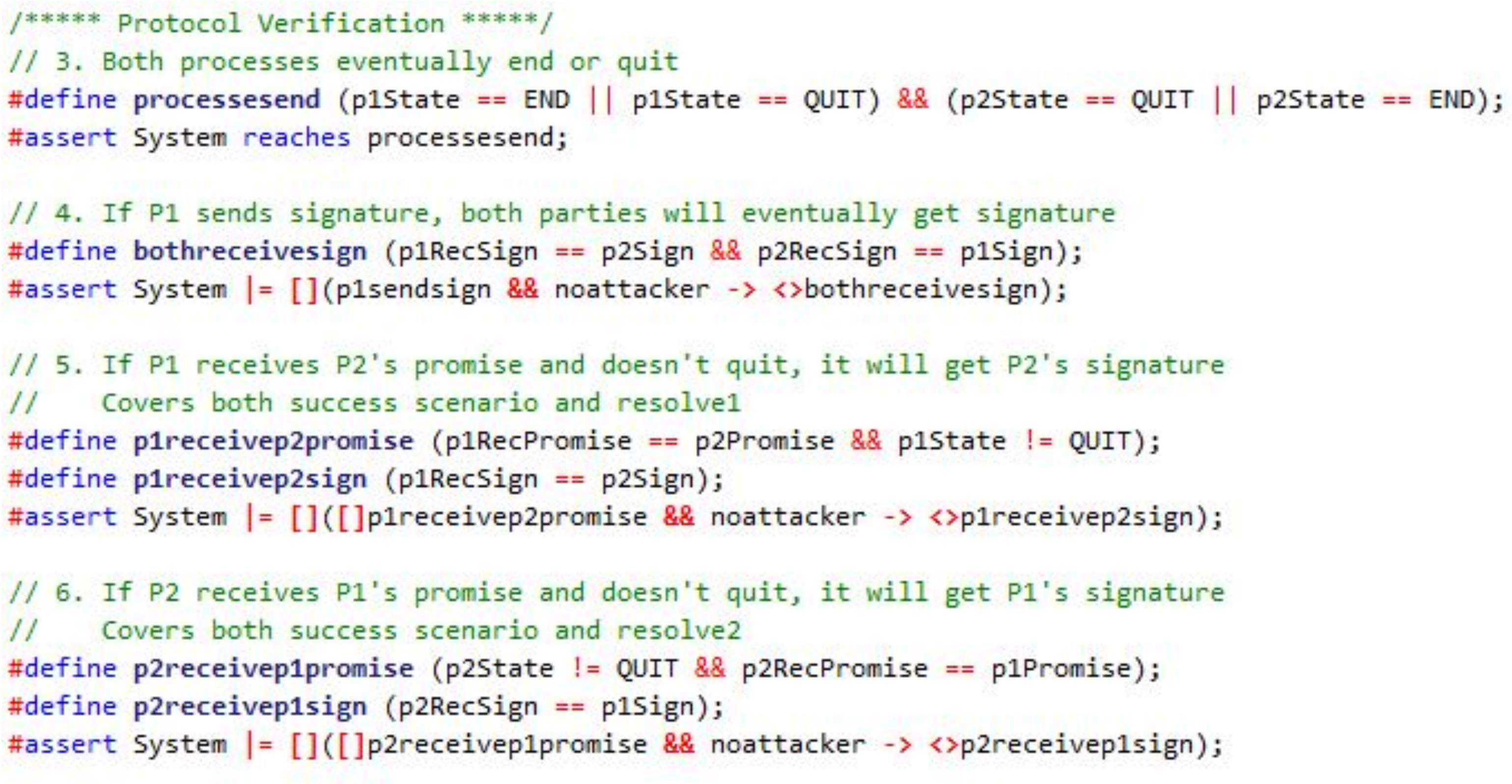
Implementation Checking



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

/***** Implementation Checking *****/
// 1. P2 only sends promise when it receives valid P1 promise
#define p2receivesbadpromise (p2RecPromise != p1Promise && p2State != QUIT);
#define p2sendpromise (p2State == RECSIGN);
#assert System |= []([]p2receivesbadpromise && noattacker -> !p2sendpromise);

// 2. P1 only sends signature when it receives valid P2 Promise
#define p1receivesbadpromise (p1RecPromise != p2Promise && p1State != QUIT);
#define p1sendsign (p1State == RECSIGN);
#assert System |= []([]p1receivesbadpromise && noattacker -> !p1sendsign);
```

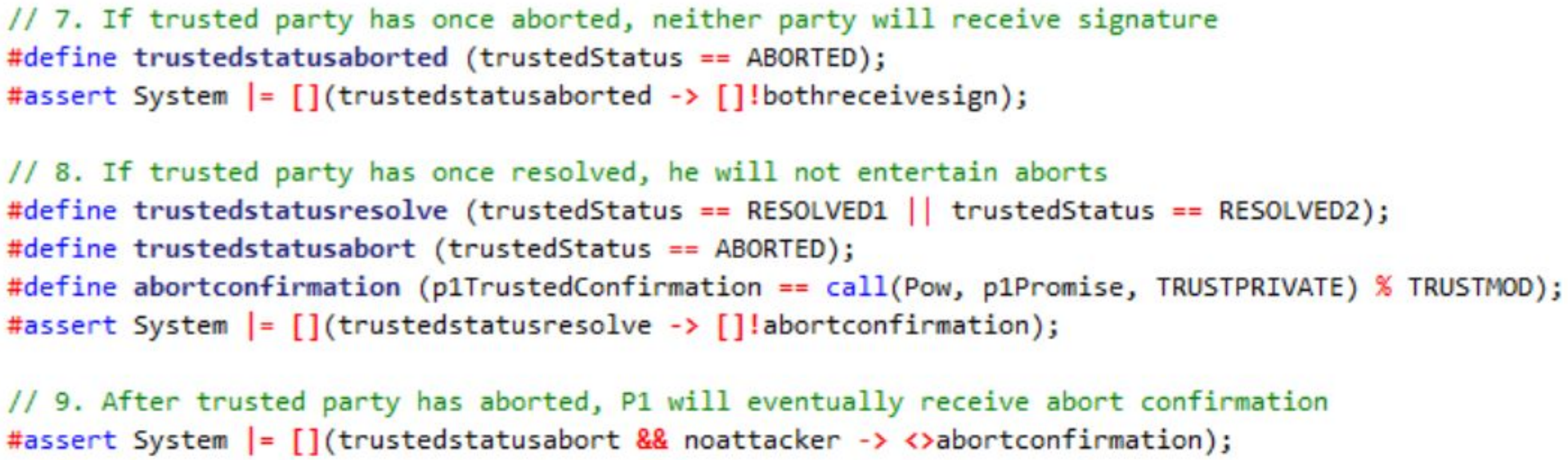
Should pass if our implementation of the protocol is correct

Verification of Protocol



```
/***** Protocol Verification *****/
// 3. Both processes eventually end or quit
#define processesend (p1State == END || p1State == QUIT) && (p2State == QUIT || p2State == END);
#assert System reaches processesend;

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

// 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);

// 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);
```

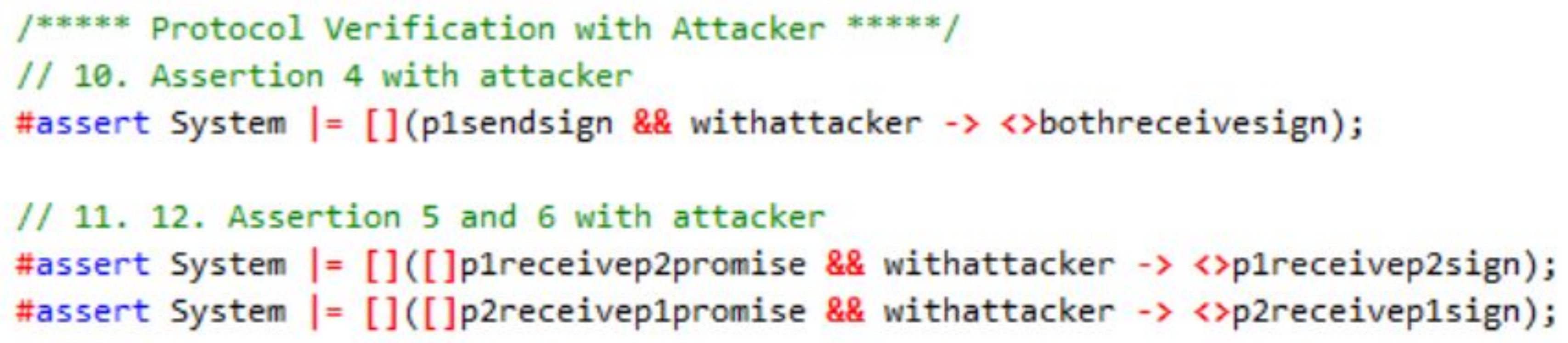
Verification of Protocol



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

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

// 9. After trusted party has aborted, P1 will eventually receive abort confirmation
#assert System |= [](trustedstatusabort && noattacker -> <>abortconfirmation);
```

Should pass if protocol works as described

Verification of Protocol (with Attacker)



```
/***** Protocol Verification with Attacker *****/
// 10. Assertion 4 with attacker
#assert System |= [](p1sendsign && withattacker -> <>bothreceivesign);

// 11. 12. Assertion 5 and 6 with attacker
#assert System |= []([]p1receivep2promise && withattacker -> <>p1receivep2sign);
#assert System |= []([]p2receivep1promise && withattacker -> <>p2receivep1sign);
```

Should pass if protocol works as described even with an attacker

Limitations

1. RSA Signing
2. Attacker Behaviour