Using Universal Composability to Implement Off-Chain Payment Channels: Appendix

IC3

Follow

The Initiative for CryptoCurrencies and Contracts (IC3)

--

Listen

Share

by Jun-You Liu (Cornell, IC3), Surya Bakshi (UIUC, IC3), Shreyas Gandlur (Princeton), Ankush Das (CMU), and Andrew Miller (UIUC, IC3)

This research was done in collaboration with

[the UIUC Decentralized Systems Lab (DSL)

](/@dsl_uiuc).

This post serves as the appendix to our post, "Using Universal Composability to Implement Off-Chain Payment Channels," so please start there.

UC in somewhat more detail

In more detail,

Z

is the environment, which is the distinguisher. Its goal is to distinguish the two worlds it's talking to, which one is the ideal world and which one is the real world. F_spec

is the ideal functionality in the ideal world. It is like the "specification", a term more familiar to developers, meaning that's the goal of what they need to achieve at the end of development. π_ideal

is a group of parties running the ideal protocol (dummy as forwarding what Z

tells them to F_spec

) in the ideal world. The correspondent of π_ideal

is π_impl

, which is in the real world. It is the same group of parties running the "real" protocol, which is the "implementation" of the "specification" in the ideal world. F_assum

is the ideal functionality in the real world, which is used when the context is a hybrid world. For now, an intuition is that it is an ideal functionality in the real world, and we assume it works as expected matching its specification. In addition, the adversary (A

) controls the corrupted parties in π_impl

. When Z

is trying some edge cases that it thinks it'll let it distinguish the two worlds, it tells A

to control the corrupted parties to do whatever the Z

wants. Lastly, simulator, S

, which is in the ideal world, is the role responsible for simulating what an adversary is doing in the real world. (See another section in the appendix for more information on the simulator. We can skip the details for now.)

We could have an interesting and more realistic analogy to understand this setting. Z

, the distinguisher, is the client who asks you to implement a specification(F_spec + π_ideal

), and we, as developers, need to implement the spec (with π_impl + F_assum

). If Z

cannot distinguish between the specification and our implementation, we are done. Nice job. However, as we know, most clients are very picky, they probably interact with our program not in a “normal” way. At this moment, it’s like they are telling the adversary(A

) to break our implementation. If they still cannot distinguish between the specification and our implementation even if they try to break it, we are truly successful, making the client indistinguishable between specification and implementation.

One downside of the UC framework is that, despite modularity, models and proof for complex protocols can be unwieldy and difficult to understand. This proves to be a challenge in verifying or working with existing UC definitions.

Real World Functionality (Blockchain Smart Contract and Synchronous P2P Channel)

The ideal functionality in the real world would be smart contract logic as well as synchronous p2p channel logic. Smart contract logic is responsible for handling the on-chain operation when a payment channel is closed. Also, when a challenge happens between the sender and the receiver due to part of them being malicious, they have to seek help from the smart contract logic as a judicial judge. As for the synchronous channel logic, we view it as a reliable method to send messages from the sender to the receiver. We assume this channel is so reliable that messages sent are always guaranteed to be received.

Let’s take a look at the code part to get a better understanding.

The following code snippet is basically three tasks that the smart contract is in charge of. It has to make sure either the sender or the receiver can close the payment channel. It is worth mentioning that the payment channel has not been closed at this moment. Instead, it enters into a period of time called the “challenging period”. Within the “challenging period”, if the party who closes the channel is dishonest, the opposite party can send a “challenge” to the smart contract. It is similar to how a person appeals a judicial case to the judge realistically. The only condition is that the latter state should be fresher than the candidate closing state. Therefore, this leads to the “challenge” function. Lastly, “settle” function can be called after the “challenging period” is due, and then the payment channel is truly closed. Sender and receiver can withdraw their collateralized asset on-chain.

On the other hand, all off-chain operations are covered in the following figure. As we introduced before, off-chain operations are basically just a reliable channel used to send messages. That is what “send_to” function is doing.

Local Protocol for the sender and receiver

Protocol exists in the real world. It is the rules that each party in the real world should follow. Therefore, both sender and receiver are running the same protocol code. They share the same following code snippets.

Pay and Close

Payment in the real world is via the off-chain synchronous channel, so we could see from the code snippet that after the sender calls “pay”, it starts with checking its own balance, updating the balance sheet for both of them, sign the latest state with its signature, and then send it to the real-world ideal functionality. Because the message is annotated as a “send” message, so the ideal functionality could recognize it is an off-chain message, it would be delivered to the receiver within one unit of round (representing the transmission delay in the synchronous channel).

When a party wants to close a message, it simply calls “close”. It checks whether the payment channel is open or not, and then sends the state of balance to the ideal functionality in the real world, waiting for this close channel message to be mined.

Receive pay (offchain)

With the code snippet, it is easy to understand. It is triggered in the receiver ITM when an off-chain payment is received. First, it checks whether the payment channel is open, and then many subsequent checks come after. It checks if the balances are valid, nonce is correct, and signature is correct. After all checks, it updates the latest balances of the sender and receivers, keeping its own copy at its side.

Receive uncooperative close

This is when “challenge” is triggered, what the receiver would be called. The receiver would be notified that the sender has already sent a close channel message and that message is mined. So the receiver has to check whether the caching final state of balance in the payment channel is correct or not. Yet, to be consistent, the receiver always responds to the payment channel with the latest state, no matter what the caching final state in the payment channel is. In this way, the receiver does

not need to take care of if the sender is honest or malicious, making protocol simpler.

Therefore, we see that the receiver check if the payment channel is open first, and then always recognize a close channel message from the sender is not with the correct balance, wrap the correct latest state, and then send it to the ideal functionality, waiting for this challenged close channel message is mined before the deadline of challenge period..

Adversary

Adversaries in the real world are dummy. What they actually do is to forward every message to the environment. It is proved that the dummy adversary is the most powerful adversary in Universally Composable security framework. So it does not have any.

Simulator

Simulator is a special role in the ideal world. It acts as the adversary in the ideal world. What it's doing is creating a real world scenario internally, using it to "simulate" what is happening in the real world and report back to the environment. It aims to make the ideal world indistinguishable to the real world.

Since Simulator is the most complex role in the ideal world, let's break it into 5 smaller pieces, which is easier to explain one by one. They are "get_ideal_wrapper_leaks", "wrapper_poll", "sim_get_leaks", and "sim_party_output".

First, on each activation, the Simulator has to get the leaks in the ideal wrapper, using "get_ideal_wrapper_leaks". The point of this operation is to make sure the Simulator can know what has happened to date in the ideal world, and then it has to decide what to tell the internal simulated world.

Second, "sim_get_leaks", is a function to obtain the leaks from the internal simulated world. If we switch our perspective to the internal world, the Simulator is like the environment of the internal simulated world. The simulator tells the simulated world what to do. Therefore, what message a real world returns back to the environment is what the simulated world returns back to the Simulator.

By calling "sim_get_leaks", the Simulator can control the workflow of what's going on in the simulated world, like you get the godview while playing games. And the Simulator can decide what message or command it should tell to other ITMs in the ideal world. So that all ITMs in the ideal world can coordinate to give the same output as what the real world gives to the environment.

Third, every time when the environment "poll" wrapper to proceed the next steps within an environment, it has to activate the simulator at the same time, which is handled in "wrapper_poll". In this way, the Simulator can sense each proceeding the environment executes and react to it.

Last but not least, "sim_party_output", which tackles the final output when the internal simulated world ends and outputs the final message to the Simulator. It is different from the procedure in "sim_get_leaks" because it gets the final message from the simulated world and then the simulated world closes, while in "sim_get_leaks", what Simulator gets is the leaks when the simulated world is proceeding.

Wrapper

In the existing UC framework, synchronous and asynchronous communication models complicate the design of protocols and ideal functionalities. Therefore, we introduced a new construction, "wrapper", in SaUCy. It facilitates the verification and understanding of security proof.

Here in our payment channel example, we use wrapper as the synchronous communication model. The goal of it is to offload all communication specific codes from the protocols and functionalities to it, mainly schedule or delay code blocks.

The following code snippet is how we schedule a code block. For example, this is a payment code block, it is scheduled a delay before actually executing the payment, which is the "pay_callback" function. In this way, we can offload it to wrapper and make this ITM clean.

Wrapper provides a standard interface to schedule code blocks and it can also be triggered by the environment or the adversary to forcibly execute a specific code block. This ensures that protocol progress will not be stalled indefinitely.

SaUCy snippet

Our version of payment channels does not make any claims about privacy. This is captured by the use of self.leak()

, which is used to leak information to the adversary. self.leak(("pay",v))

self.leak("close")

leak everything: the entire amount of every payment.

self.sid

is a record for session id. In the payment channel example, we might not need to use it. But we make the SaUCy framework flexible enough to tackle more complex implementations of other protocols. It is possible that there are more than one session in a protocol, so this self.sid

is used to identify which session id we are in.

f2p

is the “function to party” channel. As we know from the high level schematic diagram, UC consists of many roles (realistically is interactive turing machines, ITMs) communicating with each other, they have channels between them used for communication. self.P_s

and self.P_r

are the sender and receiver id respectively. self.b_{s|r}

records the balance of the sender and the receiver. The payment channel ideal functionality includes a delay.

self.schedule

is another function we implement in SaUCy. It is used to schedule codeblocks to be executed with a delay set by the adversary. In other words, delay a function with at most a certain round. Round is the basic unit of time in SaUCy. When several actions happen simultaneously, it means they all happen in the same round. It is really useful in our payment channel example. If there is an off-chain operation, let’s say a simple payment, it takes 1 round to complete the transfer from the sender to the receiver. Thus, we need to delay the payment accounting with at most 1. On the other hand, if it is an on-chain operation, let’s say a close message, we could schedule the actual close message with a delta

time unit of delay, simulating the close message is mined after delta

round.

self.yield()

is a special command to report back to the environment when there is nothing else to do in this function. We could simply think of it as a return

in a normal function.

If you have any questions or comments about this post, do not hesitate to contact us. The corresponding author is Jun-You Liu (

[jl3956@cornell.edu

](mailto:jl3956@cornell.edu)).

We thank Sarah Allen, IC3 Community Manager, for her help with this blog post.