Using Universal Composability to Implement Off-Chain Payment Channels

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).

Payment channels are one of the fundamental approaches for scaling cryptocurrency networks. In the academic cryptography literature on payment channels, it has been effective to use universal composability (UC) framework as a way of rigorously modeling and giving security definitions. However, there's been a big gap between the UC model and the actual software implementations of payment channels that have been designed and maintained by cryptocurrency developers, not getting as much benefit from the UC as we could. SaUCy is a project that aims to bridge the world of cryptocurrency developers with the UC framework.

In this post, we present an implementation of a simple payment channel protocol in SaUCy. Unlike existing payment channel implementations, we also provide an executable security specification in the form of a UC ideal functionality. You can execute our payment channel specification to better understand what it does. The protocol also supports test driven analysis directly based on the security specification. This enables end-to-end tests that wouldn't be possible with just the implementation alone. Since our scope is roughly the simplest possible payment channel, it's only a proof of concept, but it gives some view of how in the future, formal security specifications could be packaged along with full-featured payment channels and other blockchain software releases.

Background

Payment channels

Payments channels were one of the first proposed blockchain scaling solutions. They rely on off chain transactions in a channel between two mutually distrusting parties which can be settled on-chain at a later time. Payment channels can be further chained together, and therefore being a graph, where payments can be made between two parties that do not share a channel. The most widely-known example of a payment channel network in the wild is the Lightning Network.

They push transactions between two people off-chain to allow for instant confirmation. They are useful because participants can avoid paying transaction fees and having to wait for confirmation. (There's also the potential for privacy benefits, but that's not the focus of this post). As an example, imagine Alice paying Bob for some continuous service, let's say some fee for wifi access paid out every few seconds. It would be too expensive to make a blockchain transaction every few seconds. Using an off chain payment channel, Alice can make the next payment to Bob with only local communication, no blockchain needed.

At a high level, payment channel participants deposit money in an on-chain contract along with their signatures and then revert to sending signed messages to each other proposing new payments/balances. At some point in the future either of the participants can choose to close the channel with a single on-chain transaction containing the signatures of both participants and claim their coin balance. Only the close transaction goes on-chain.

Security is achieved by requiring signatures to update any state (balance in the off-chain payment channel) and the on-chain contract logic (guaranteed by the blockchain) can ensure channels always close correctly.

In our example we use a simplified payment channel: a unidirectional payment channel. In this channel there is a sender and a receiver, and payments can only be sent in one direction: from the sender to the receiver.

Universal Composability and SaUCy

UC is a way of giving a formal security specification for a protocol. It is a generalization of more common game-based cryptography, but it's especially well suited for blockchain applications because of its compositionality. (The security of a protocol proved under the UC framework is still preserved even when running concurrently with arbitrary other protocols. Just think of all the different applications you might run from a single cryptocurrency wallet.)

Before digging into the details, we emphasize that UC can be tricky to understand. If everything does not immediately click, that is OK! We will go through everything in detail as we define payment channels in SaUCy.

UC defines security in the "ideal world/real world" paradigm. Here is how UC's security definition looks at a high level (more details in the appendix):

There are two worlds, the ideal world and the real world. The ideal world, as its name suggests, contains an idealized version of the protocol in question, called the ideal functionality, corresponding to the F_spec

in the figure. The ideal functionality is a trusted third party that parties can use as a black box to complete tasks we want the protocol to achieve. Parties in the ideal world (π_ideal

, referred to as just "dummy parties") forward all commands coming from the environment to the ideal functionality. On the other hand, in the real world, we define the protocol (π_impl

) that achieves the task we desire under untrusted situations. We then show the UC security of this protocol by comparing it with the idealized version.

Security is proven through the notion of "indistinguishability" between the two worlds. If we can show that an environment, the distinguisher, that gives inputs to the parties and the adversary in both worlds cannot distinguish between the two, given only the outputs of the parties and the adversary, we say that the real world (the actual protocol in question) achieves the same security properties as the ideal world.

For those unfamiliar with UC at all, we found it useful to think of it as analogous to the famous Turing Test. In each round of the test, it is not 100% sure whether it is interacting with a bot or a human being. Instead, it only "distinguishes" when they are different. Same as universally composable security framework. We couldn't tell both of the worlds are the same. Instead, we could only "distinguish" if they are different.

While you can find implementations of payment channels such as Sparky (a simplified one that comes with a nice tutorial), or Lightning and Raiden (production implementations in use today), they don't come with any formal security specification. This is where SaUCy comes in — a library for writing formal security specifications for distributed protocols. In the SaUCy vision of the future, every protocol library module will come with an ideal functionality as its specification. We think of it as a Byzantine-secure upgrade for ordinary API references, since it describes the guarantees you get even when your opponent in the payment channel runs non-compliant software of their own choice.

Building a Payment Channel Protocol with SaUCy

Let's start by explaining the high level picture of payment channels in UC, illustrated below.

As described in the previous section, UC is an ideal/real world paradigm. We use the same color as the UC high level picture is using for better understanding. In the ideal world, F is the ideal functionality, the specification, matching the F_spec

in UC. The real world is an ordinary payment channel protocol. It consists of a smart contract program, and local code for each of the sender S and receiver R, and a p2p channel for the sender and receiver to send messages. A distinction in UC is that the blockchain is modeled as an ideal functionality, so the "smart contract" in the figure is colored with green, meaning it matches the F_assum

in UC. And the p2p channel is also defined as an ideal functionality. (In practice, the p2p channel can be implemented by the whisper network, standard HTTPS protocol, websocket, or something else). The full code for our project and instructions for running it are provided here at Github, though we'll mention some details about it later on.

Since our implementation of the Real World will be familiar to those who have already seen payment channels like Sparky, Lightning, Raiden, etc., the use of an Ideal Functionality will only be familiar to academic cryptographers. So our focus in this blogpost is in clearly explaining the ideal world.

Ideal world (Specification World)

Recall that the main idea of UC is that the Ideal World serves as the specification that our protocol will eventually be compared against. The ideal world defines "what we want". It's necessary to understand exactly what behaviors and guarantees the ideal functionality is offering. The fact that it's written as a trusted service that simply does everything itself, means that this should be tractable — it shouldn't have to use any cryptography or complex protocols to achieve the desired

goal making analyzing its security properties trivial. you don't need to understand cryptography or fault tolerance to work through what the ideal functionality does.

Functionality

The ideal functionality of the payment channel is "F_pay", the specification of payment between the sender and the receiver.

To make things simpler, we assume that a payment channel has already been open between sender and receiver as it's not a very interesting interaction to model. And we split code into the ideal world and the real world.

To turn these into a UC specification, we have to write an ideal functionality that exhibits these properties. In short, we could think of "F_pay" as a trusted black box that could achieve what we expect in the specification and satisfy all security properties.

Let's dive into the code part to get more understanding. There are two main functions in the ideal functionality. One is "pay", and the other is "close".

"pay" is in charge of the off-chain payment from sender to receiver. All "pay" logic is in the above code snippet.

There are several saucy syntax that I will briefly explain, with more details in the appendix (saucy snippet) for those who are interested in. self.leak

is used to leak information to the adversary. self.sid

represents the session id. It is very useful when identifying which payment channel we are referring to. f2p

means "function to party" channel. As we see the UC high level schematic figure, different roles are connected with each other. This f2p

channel is just one of them. self.schedule

is used to schedule codeblocks to be executed with a delay set by the adversary. Using an analogy that devs are more familiar with, it is like the setInterval

in Javascript, delaying a function call with a time interval. Last one is self.yield

. It is like a special version of the return in a function call, triggering the environment when there is nothing else to execute.

I would like to point out the difference between "pay_callback" and "pay" first. When the sender sends amount x to the receiver, it triggers the "pay" function. Due to the transmission delay, even if it's off-chain, the payment is not complete immediately. It takes a unit of time for a payload/packet to be sent from the sender to receiver through some kind of synchronous channel, such as TCP. Therefore, "pay" is the entry point, and "pay_callback" is the true accounting of the latest allocation of the assets in the payment channel to each of them.

In "pay", we need to check that if the current state of the channel is open, otherwise payment is invalid. Another check is whether the sender has enough balance to pay to the receiver. Then, we can see that it calls the function "schedule" to schedule the actual payment, "pay_callback", with at most 1 round of delay, meaning that the actual accounting in "pay_callback" would be sure to complete in at most 1 round later.

Close

"Close" is called when either the sender or receiver wants to close the payment channel between them. This could be either the case that both of them are satisfied with the outcome or the case that the sender is malicious and wants to cheat on the receiver by closing with a not the latest state.

We could discover that here we also have the "schedule" function. It is used to ensure that between the round functionality receiving the close channel message and the round functionality actually close the channel has some delay. The delay is necessary because realistically the transaction of closing a payment channel should be issued on-chain, and then the eventual allocation of assets in the payment channel are settled after the transaction is mined. In short, "close" is the entry point when one party wants to close the channel, and "close_callback" is the actual close of the payment channel.

In "close", we see that there is a condition check to decide whether the delay is "self.delta" or "self.r * self.delta", where "self.delta" is the unit of a block being mined on chain, and "self.r" is a multiplier to ensure the honest party has enough time to react to a malicious counterparty.

In the "if" condition, it checks whether the sender of this close channel message is from the receiver, or it's from an honest sender. If yes, then the delay of the actual close channel should be complete within the "self.delta" round. If not, meaning the close channel message comes from a malicious sender, then the actual close channel message is scheduled longer, which is "self.r * self.delta". What causes this difference? It's because of a very important mechanism called "challenge".

When the sender of a close channel message is either an honest sender or the receiver, it is closing the channel with the

latest state (even if the receiver is malicious, to be rational, it would close the channel with the latest state because it has the most assets in that one). Thus, for the close channel transaction to be mined, it takes “self.delta” round to complete.

On the other hand, if the sender is malicious, it may want to send a previous state which the sender has higher balance to cheat the payment channel. So in this case, after this transaction is mined, the payment channel enters into a state called “challenge period”. It lasts for some amount of unit block time (which is the “self.delta”), letting the receiver have the time to send another transaction including fresher balance of both of them to the payment channel. This operation is called “challenge”. The receiver challenges the previous balance state the sender sent before, and then replaces it. Therefore, it needs “self.r * self.delta”, where “self.r” is an integer usually bigger than 2.

Lastly, in the “close_callback”, it is the actual close of the payment channel. So it first checks whether the payment channel is open, closes it, and then sends the latest state of sender and receiver’s balance as the message back to the sender and receiver.

To recap the security requirements:

Sender:

“By the deadline, the sender must withdraw any remaining balance they haven’t sent yet.”

Receiver:

“If the protocol indicates their balance is X at the time they initiate a withdrawal, they receive at least that much.”

These two security guarantees are provided by the accounting inside the ideal functionality. It maintains who has how much in a trustful way.

Real world

The Real World component of SaUCy is essentially what you would find in an existing payment channels implementation. It includes the smart contract that runs on a blockchain, as well as the local code that the individual parties need to run on their own (their wallet interface).

Additionally, we assume there exists a reliable off-chain p2p channel in the real world, the one that the sender uses to send the payment to the receiver. We also include this into the ideal functionality in the real world. In short, as long as a party in the real world wants to send a message, it has to pass this message to this real-world ideal functionality. And this ideal functionality would reliably decide where this message should be processed as an off-chain message or an on-chain transaction to be mined.

Frequently Asked Questions about the ideal world Specification for payment channels:

However, reading to this point, you might have the following immediate questions.

1. Where’s the cryptography? There are no hash functions or digital signatures in the ideal functionalities, yet clearly the payment channel protocol is based on these. So where did they go?

This is on purpose. “A good ideal functionality hides away as many crypto details as possible of the protocol that implements it.”

The ideal functionality is a specification, so it’s meant to abstract away as many implementation details as possible. For the payment channel, digital signatures are part of how we implement it, but not part of how we state its goals.

1. Is the ideal functionality basically just a smart contract?

The ideal functionality is more powerful than the smart contract. For example, smart contract transactions can have a delay and can be front run, but communication with the ideal functionality is instantaneous. Our ideal functionality for payment channels is especially simple because it leaks everything, i.e., the protocol is full information. But for other cryptographic applications with confidentiality, like private cryptocurrency or anything involving zero knowledge proofs, the corresponding ideal functionality would typically keep some secret data without leaking it.

1. What happens when the parties are bad?

In the real payment channel protocol, there is a smart contract, local code for the sender, as well as local code for the receiver. And of course either the sender or the receiver could be malicious and run some entirely different “attack” code instead. In the ideal specification world, we keep things simple by having the ideal functionality do the work. The sender and receiver are not required to run any particular code at all, they are allowed to invoke the ideal functionality with whatever inputs they want. Ideal functionality is the law.

This is one of the reasons the ideal functionality is simpler to understand than the protocol itself. In the real world protocol, you have to consider what happens if one or the other of the parties is bad. In the ideal world, the functionality is doing all the work and it never fails.

What can you see by running the demo?

Here's the punchline — our module comes with two viewpoints of a Payment Channel. You can run either of them, the ideal world with the functionality that does the work, or the real world with a blockchain protocol. No matter how you run them, the behavior will be the same.

We also provide a template for you to create your own test environment (See env_1.py

for more information). You can set up different scenarios by telling the sender to pay x

amount of money, and then check its balance, and then ask the receiver to close the channel. Or you could also ask the sender to close the channel, and then it triggers a challenge between them due to the protocol mechanism, and then close the channel. Basically, you have full control of the scenario by ordering your own commands.

The environment is the distinguisher. It tries to distinguish which is the ideal world and which is the real world. If you set your test environment correctly, after you run the environment, you should see something like the following,

It means that the environment cannot distinguish between the ideal world and the real world, so the implementation of the protocol matches the specification. And we reach our goal.

Summary

In this post, we explain what Universal Composability and our proposed framework SaUCy are, and we implement payment channels using our framework as an example. We hope developers can use SaUCy to help them develop blockchain protocols to ensure their security without needing to know how cryptography works.

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

[jl3956@cornell.edu

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

In follow up posts we plan to discuss:

- The real-world protocol in detail to explain how to fully use SaUCy
- Some gritty parts of UC
- An overview of UC for programmers

Reference

Code repository: https://github.com/orbxball/uc-contracts/tree/payment

Python saUCy Code repository: https://github.com/sbaks0820/uc-contracts

saUCy: https://github.com/initc3/saucy

Appendix

Please see our post "Using Universal Composability to Implement Off-Chain Payment Channels: Appendix" for further reading.

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