There are two different ways of doing penalties and fee-splitting (where multiple parties share the costs of on-chain transactions) in channels.

In one approach, specific funds or assets are set aside as “deposits”. These “security deposits” can be thought of as belonging to a certain “state channel game” which conserves total asset quantities, and uses as inputs or “moves” the different signed messages passed back and forth amongst “players” in the channel, and/or various states of the blockchain. When one player makes a certain “move” in this game by, for example, publishing an out of date state to the chain, another player may be able to respond on their “move” by providing evidence of this fact (such as a later state which has been signed by the first player), and then the game would allow them to claim the other player’s deposit, or be refunded a tx fee, or whichever penalty/fee the protocol demands

In the second approach, particular funds/assets are not necessarily set aside as deposits. Instead, the assets within a channel may be fully allocated to the set of open “channel games”. Penalty logic also exists, but is not restricted to act on assets of a separate “deposit”. Instead, penalties take “priority” over other channel applications, allowing them to use the value of the assets which may be owned partway through a chess game as part of the penalty structure, for example. This also permits penalties such as “loss of all assets/state” to be enacted, analogous to the Lightning Network’s approach.

On the face of it, the second approach appears to be strictly more flexible, and also more capital efficient for certain penalty structures, than the first. But a natural question exists: can we encode the second approach within the style of the first approach? And yes, it appears that we can.

To do this, we can modify every single application run within the state channel to entail the entire penalty process as part of its own state transition logic. Now, assets can still be allocated fully to individual channels, but all assets are also part of the penalty scheme, since the penalty scheme is contained within every application. In addition, we must have some sort of “default rule” for the base layer channel mechanism which allocates all assets to an instance of the penalty game unless otherwise allocated to another application. This prevents penalties from being temporarily “paused” when an individual application, such as a chess game, is finished.

Obviously, this naive approach has a lot of drawbacks. Depending on the architecture, it may rely on every single game developer correctly implementing the penalty scheme. It may require the closing and re-opening of all applications when the penalty policy needs upgrading. It may require multiple penalty policies and complex logic to be applied beyond individual state channels as part of state channel networks, where the counterparty to the metachannel/virtual channel is not the direct counterparty, and one needs to differentiate between the loss of the assets locked to a metachannel locally, and the net

loss of that asset movement across the entire chain of interactions (a.k.a. misbehavior by the adjacent connection vs misbehaviour by the remote party to the metachannel). If long-running metachannels are “left open in the background” as part of various mechanism design use cases, the cross-application upgrade to a fee policy may be impractical. etc.

For these, as well as many other reasons, I have always steered away from trying to enforce the notion of asset conservation at an architectural

level in state channel designs (as opposed to enforcing it within a subset of the overall channel’s structure, which is obviously quite useful for many purposes).

Last Thursday the excellent Tom Close gave a presentation on his Nitro paper (from February) in the state channel researcher’s call. In the paper, and the call, he claims that the paradigm of “total asset conservation per channel” which the paper proposes is capable of supporting “arbitrary” state channel techniques, which I have expressed skepticism of, both privately before the paper’s release, and again on the call. He asked me to document specific cases where I thought the approach might fail or have serious drawbacks, which is what prompted this post.

Security penalties and fees are not directly treated by the Nitro paper, but the general scheme used would seem to imply that the first of these two approaches I describe in this post is the only one compatible with his approach, and since security penalties are a crucial element of state channel design I thought this was a good place to start. There are a very large number of scenarios that I am skeptical can be handled effectively by the Nitro protocol, which I view as being useful primarily for its clear formalism that targets a specifically identified subset of use cases, rather than as a technique that should be used for arbitrary applications. In particular I am a highly outspoken critic of the tendency for people to view generalized state channels as “just asset transfer with arbitrarily complex conditions”, as I have mentioned many times before in analyzing various other proposals for state channel architecture. But since considering each of those scenarios in relation to Nitro protocol specifically would take quite a bit of time for both of us, I hope this post can at least kick off the conversation in a more concrete way, while also exploring topics that the Ethereum research community more broadly can find useful along the way.

Questions for Tom:

First, (assuming my take on Nitro is correct) is there a better way than my “naive” version to implement security penalties within your approach? (The following questions assume that there is, but I don’t want to propose one on your behalf).

Within such a proposed technique, how would an upgrade of penalty/fee policy for a “ledger channel” counterparty’s misbehaviour look? How about an upgrade of penalty/fee policy for a remote “virtual channel” counterparty’s misbehavior? And in the example, does either have to be aware of the process for the other?

Also, in the proposed technique, does the base channel object have to be directly aware of and implement the penalty/fee policy as well? This appears to be required in my naive approach, but I’m curious to see how this would work in your preferred take.