# Counterfactual: Generalized State Channels

Jeff Coleman, Liam Horne, and Li Xuanji

*L4*[*]

(Dated: June 12, 2018)

**Abstract.** State channels are an important technique for reducing fees for blockchain users. Within their scope of applicability, they allow users to transact with each other without paying blockchain transaction fees and with instant finality, and are the only technique that securely realises the latter property. We describe *generalized* state channels, a construction that allows users to install new functionality in an existing channel without touching the blockchain, using *counterfactual instantiation* of contracts within a channel. We present an object-oriented approach built on top of ethereum that encapsulates functionality and state in *counterfactually instantiated contracts*, providing numerous privacy, efficiency and security benefits over a monolithic approach, and describe a new object-oriented *metachannel* approach to building state channel networks. We analyze the unique security assumptions of channels and describe third-party services that channel users can benefit from using.

## 1 Introduction

State channels enhance blockchain performance by taking state-modifying operations off of a blockchain and executing them directly between defined sets of participants. Payment channels [23] were the first type of state channel to be described, using off-chain interactions to modify ownership of locked Bitcoin, thereby allowing users to make "off-chain payments" to each other. The term "state channels" generalizes this approach beyond payments, encompassing all types of blockchain state modification which operate within a security paradigm comparable to that of the payment channel.

### 1.1 State Channels: Basic Overview

State channels let parties securely modify locked portions of blockchain state called **state deposits**. These deposits are typically held in **multisignature wallets**, where the participants to the state channel are the signers to the multisig.

Participants update the state channel by exchanging off-chain messages. These messages describe an update to the state deposit, for instance, a payment that changes the balance between two parties in a payment channel, or a player's next move in a chess state channel. Participants can continue exchanging these off-chain updates without incurring any on-chain fees until they choose to close the channel, at which point the most recent state update is sent to the on-chain multisig as a single transaction, and the state deposit is withdrawn to the parties in accordance with the final state.

Participants store copies of these off-chain state updates. Because every message is cryptographically signed and the multisig contains code to verify these signatures and interpret these messages, participants preserve the ability to realize the most recent state update on-chain at any time. Parties are prevented from submitting old messages by their counterparties: if Alice submits an old state, Bob is given an opportunity to "rebut" it by broadcasting a more recent state. This design allows participants to treat updates within a state channel as "final", despite taking place entirely off-chain.

[*]Contact L4 Research at research@l4v.io

### 1.2 Comparison to other Techniques

State channels are one of several techniques for moving state modification off of a given blockchain, relocating those operations to a secondary environment where they can be performed at lower cost. Among these techniques, one important differentiator is whether the relocated operations introduce additional consensus assumptions which, should they fail, will permanently compromise long-term guarantees (such as persistence of asset ownership). **Sidechaining** [10] is an example of a technique that falls in this category; if state is sidechained to a different blockchain and that blockchain's consensus mechanism fails, owners or beneficiaries of that state may lose everything delegated to that blockchain, even while the original blockchain remains secure.

By contrast, techniques like state channels (as well as **Plasma** [22]) allow users to restore their state to the original blockchain (assuming it has remained secure), even when the off-chain state modification-mechanism has failed, permitting a method for users to restore their state to the original blockchain if it itself has remained secure.

To enact these "safety net" provisions, both Plasma and state channels depend on the assumption that new transactions can be published to (and finalized on) the underlying blockchain. Such transactions are needed not only when depositing to or withdrawing from a state deposit on that chain, but also so that the blockchain can serve as a trusted third party able to resolve certain types of dispute amongst the mechanism's participants. This assumption specifically includes censorship resistance as well, because selective censorship of a participant's submissions to the chain can prevent that participant from safely exiting a channel or Plasma chain. Therefore, state channels' safety properties depend directly on the safety and liveness properties of the blockchains where their state deposits are locked, as well as on specific capabilities of channel participants and their delegates.

Within the category of "safety net retaining" approaches, state channels are further differentiated from Plasma in that they proceed by **total consent**. Any change of state within a state channel requires explicit cryptographic consent from all parties designated as "interested" in that part of the state. Prior to depositing state in the state deposit, a party's "interest" is assumed to be defined by the rules of the underlying blockchain itself. When they enter into state channels, parties choose other parties with whom they wish to interact, and add those parties to the set whose consent must be obtained to perform operations on deposited state. Because of this continual need for obtaining and proving direct consent, state channels depend heavily on the records and responsiveness of state channel participants (or their delegates).

Together, all these factors make state channels one of the mostly tightly constrained techniques for improving the performance of blockchain-based applications. Yet despite all these restrictions, state channels enjoy many advantages when compared to regular on-chain operations.

### 1.3 Benefits of State Channels

First, by reducing use of a blockchain itself, which must be processed by the entire network, the total overhead associated with channel based applications is radically reduced. This is especially true for applications where a large number of operations are conducted which affect only a small number of parties directly. Individual channels typically have smaller participant sets than Plasma, and this means less total duplication of the computational operations within the channel. As a result, state modifying operations within a state channel can be conducted extremely cheaply, more cheaply than under almost any other blockchain performance enhancement technique. Techniques such as micropayments and streaming payments, which would be prohibitively expensive if enacted in the form of regular on-chain transactions, become viable when channelized.

Second, because the state controlled by a state channel can be finalized very rapidly, state channels can remove the need to await "confirmations" from a blockchain before an operation is

considered immune to unexpected reversal, resulting in significant user experience improvements. The apparently extreme requirement of total consent is the source of this rapid finality, and it also permits cooperating and available participants to make immediate withdrawals of state from the state deposits their channels control. By contrast, Plasma chains mandate a delay when withdrawing state to the parent chain.

Third, when state channels are connected together into **state channel networks** such as the Lightning [23] or Raiden [3] networks, it becomes possible for parties who do not directly share a state deposit to enter, via chaining of consecutive channelized agreements, into channelized agreements. This allows for extremely wide and rapid interaction across large networks of participants without any on-chain transactions, although with certain restrictions on the type of interaction, and at some additional degree of overhead in terms of the amount of locked state required to support large networks.

If architectures can be developed which maximise state channels' advantages, while minimizing their costs and inconveniences, significant improvements in the capabilities of blockchain based applications are possible. To date, however, the number and quality of state channel implementations remain extremely limited. There are several reasons for this.

One is that the efficient channelization of individual applications requires significant engineering and game theoretic design effort. State channel client software must account for and properly handle a wide range of possibilities and edge cases. There are no standard libraries, reference implementations, or tutorials on how to implement channelized applications. At present each team must start essentially from scratch, and make an enormous investment of time and expertise to produce workable results.

But the most critical barrier, from our perspective, is that most blockchain users do not run a single application exclusively over long periods of time. Because entering into and exiting per-application state channels still require the submission of normal on-blockchain transactions, with all the fees and waiting which that experience entails, the benefits of channelization are limited to repeated operations within the application itself. For application-specific channels to provide a net economic benefit to the end user, a large number of consecutive operations within the same application must be the norm. Few applications fit this specific profile. To make matters worse, forcing users to wait every time they begin using an application undercuts the usability benefits state channels purport to offer.

### 1.4 A Generalized Framework for State Channels

In this paper, we present an alternative approach for channelizing blockchain-based applications. Rather than require each application developer to build an entire state channel architecture from scratch, we propose to build a generalized framework where state is deposited once, and can then be utilized by any application compliant to the framework. Neither the installation nor the opening and closing of new applications will require any on-chain transactions. The user experiences channelization by default, and is also able to make normal transactions with comparable cost and convenience to any non-channel method of storing and protecting their blockchain state.

From the developer's perspective, the generalized framework manages the "heavy lifting" of the channel architecture, while presenting a simple and understandable API of basic functionality to the application. This frees application designers from having to also be state channel experts when they merely re-use standard functionality such as payment, conditional payment, atomic swap, etc. For the advanced developer new functionality can be designed and added to the framework, benefiting from the same lack of requirement for new on-chain transactions for the user in the general case, but also permitting many applications to re-use new functionality once it has been developed for the general framework. Our expectation is that foundational shared functionality

will be carefully vetted, formally verified, etc. so that the high costs of providing reliable security guarantees are shared amongst application developers rather than being repeated each time a new application is developed and deployed.

Our approach is open source, modular and contract-based. We begin by detailing the requirements and objectives for the design, and then continue by describing the specifics of how basic components are implemented within our framework. In addition to the basic functionality of the framework itself, there exist important surrounding services which will keep channelized applications usable, cheap, and performant. Some of those services, as well as the more advanced techniques for heavy optimisation within the framework, will be referred to only briefly with details left to be explored in further research.

## 2 Requirements & Objectives

In order to provide a generalization which can support as many use cases as possible, we first need to understand the criteria that state channels will be expected to meet. Only then can we describe the set of use cases which meet those criteria within our proposed approach. We begin by identifying four base criteria. The first is the one we consider most foundational:

> **The Safety Criterion**: Parties who entrust state to properly initialized state channels should not significantly increase the risk of that state being manipulated in unauthorized ways, even when counterparties/intermediaries to their channel are both malicious and unreliable (within the bounds of an explicitly specified threat model). The risk of being unable to withdraw the state from the state channel is here explicitly included.

This inextricable requirement, sometimes also referred to as trustlessness, "being trust free", or simply "being secure", is the core of any state channel construction. The entire purpose of a state channel is to move operations off of a blockchain *without* exposing participants to unacceptable risks. The definition of significance is left up to the specific threat model to which a particular construction is addressed. We discuss some of the threat models appropriate to realistic state channel environments in Section 3.

Our second requirement is that new operations can be carried out within the channel, and that these operations have the same degree of finality and irreversibility as the analogous operation performed directly on the chain itself. This requirement ensures that the channel supports actual features, i.e., that users can modify the state in interesting ways beyond simply depositing and withdrawing it.

We consider a state modification to have been carried out in a channel if it is **finalized**, i.e., when one party is incentivized and able to bring the state modification on-chain:

> **Finality in a Channel**: Blockchain property $X$ is finalized within a state channel when there exists some channel participant $A$ who cannot be prevented from realizing $X$ on the chain if they choose to, and who prefers the realization of $X$ to any conflicting property they are similarly assured of being able to realize.

Note that under this definition there need be no actual place (such as a disk/memory location or shared "mini chain") where channelized state is "stored". Rather, a channelized state exists when one or more channel participants have been convinced that the state has been finalized. In practice a participant's software does this for them by comparing received messages (as well as relevant blockchain information) against a threat model in order to decide the channel state that it reports to the user; and by responding to new information from users/counterparties/blockchains

in ways that maintain the finality of the state (sometimes with the aid of appropriately incentivized delegates).

In addition to the two security-oriented base criteria, there are also two performance-oriented base criteria we would like to identify.

> **The Responsiveness Desideratum**: If all parties to a channel are online and in agreement about some currently finalized state, and some participant(s) fulfilling the Finality Criterion for that state request to withdraw it, the remaining parties should prefer immediate approval of the withdrawal request over delaying approval or refusing to respond at all.

and

> **The Off-Chain Desideratum**: Unless a state withdrawal is being made, or an on-chain operation becomes necessary to maintain the Finality Criterion for previously finalized state, all parties should prefer finalizing new state transitions within an existing channel to finalizing the same state transition on the blockchain itself.

Although it is easy to devise a state channel where instantaneous withdrawals are *possible*, it is the Responsiveness Criterion that ensures instant state withdrawals are also the *normal* case, and this is critical to ensuring that use of a state channel as the default user experience remains practical. The Off-Chain Criterion, meanwhile, keeps new state changes within the channel wherever possible so that we maximize the availability of that state to new channelized applications going forward. The combined fulfilment of these desiderata results in a clear separation between the default, preferred case of near-instant off-chain interaction and the easy, accessible capability to remove state from that off-chain environment at any time. These are the guidelines we want to adhere to in order to keep channelized applications strictly superior to their on-chain alternatives for real-world users.

Beyond these four base criteria, there are many other objectives which both inform our design, and can be used to compare and contrast the differing approaches to state channels which have been so far proposed. In our view, an effective state channel design should (in no particular order):

- Minimize the use of on-chain operations and storage
- Permit participants to install and use new channel-based applications and types of channel operation *without* requiring on-chain operations
- Support high rates of throughput
- Support privacy-preserving operations and techniques, including with respect to whether a channel is in use at all
- Permit deployment on diverse types of hardware (from embedded systems up to high performance clusters)
- Prevent tracked state and/or number of required transactions from needlessly ballooning over time (both for performance and DoS reasons)
- Minimise capital requirements both within a single channel and across a network of channels
- Permit on-chain components to remain unchanged for year-like timescales

- Allow composability of channels, so that applications need not differentiate between the net state available to a particular set of parties and the individual channel relationships that contribute to making the different parts of that state available
- Support multiple parallel operations happening at the same time against a single state deposit (i.e. closing out one application's channel operations while continuing another without interruption)
- Support on-chain modification (including deposit or partial withdrawal) of underlying state deposits *without* closing and restarting channel operations
- Use a modular approach to encapsulate different features and types of channel operation (both for security and to minimize duplicated effort when deploying new channel-based applications)

There are doubtless further objectives for state channel design that can be identified, and we hope to explore them in subsequent research. This list, however, can serve as a starting metric against which to evaluate our proposed design and the broader state channel ecosystem as it presently exists.

## 3 Threat Models

End users will be willing to trust and use state channels to the extent that doing so does not unneccesarily increase the risk of their state being manipulated in unauthorized ways. This decision depends on their chosen threat model, i.e.,

1. What he considers to be possible actions of his counterparties, the environment, and in general anything outside of his own control,
2. What actions he considers himself capable of performing, and
3. What he considers unauthorized manipulation of state.

The threat model determines the extent to which a state channel interaction is trustless, since actions outside of the threat model are trusted not to occur. Hence it is important to document realistic threat models and analyze the security and performance of our construction under them.

In this section, in addition to threat models, we specify a variety of **protocols** and **behaviours**. A behaviour[1] specifies rules that a compliant client abides by, while a protocol is the set of rules enforced by a state channel smart contract.

### 3.1 Common Assumptions

The safety of state channels relies on two assumptions that are not assumptions required to safely transact on a blockchain.

***3.1.1 Reliance on Blockchain Liveness.*** State channel threat models assume that various properties of the underlying chain can be succesfully preserved during any attack. Some of these assumptions, like the assumption that on-chain contributions to state deposits are successfully locked after a given number of confirmations (or some similar finality metric), mirror the security expectations that are typically placed on blockchains by users in general. Others, like the property

[1] We use this word interchangeably with "strategy" in the game theoretic sense.

of censorship resistance for a given channel participant with respect to a certain size of submission window, constitute tuneable parameters that may improve the performance and capabilities of state channels when state deposits are stored on blockchains which more effectively satisfy the desired property.

***3.1.2 Reliance on Participant Availability.*** State channel threat models assume a more active and capable set of participants than the standard user of a public blockchain. For example, while all users of cryptography-based blockchains must have the ability to securely store and use some type of private key(s), most blockchain applications presently assume that possession of these keys is a sufficient condition to retrieve and modify the current state of a users' blockchain-based rights and assets. By contrast, in order to be confident as to the current state of an open channel a state channel user may be required to store both their keys and a variety of records on what has occurred within the channel, including both their own actions and certain signed messages which have been received from channel counterparties[2]. State channel users must also be confident of being able to connect to the blockchain (and hence the internet) at some regular frequency so that they can detect and respond to outdated updates if a counterparty publishes them to the underlying blockchain where the state deposit resides. Although many of these capabilities can either be tuned to acceptable levels or safely outsourced, their effect on the overall threat model must still be accounted for, and we aim to be explicit about these assumptions whenever possible.

***3.1.3 Absence of External Incentives.*** Like the threat models used in most cryptoeconomic constructions, we include some models that emphasize a highly local set of economic incentives. They are not designed to model or remain secure in the face of external incentives that exceed the economic incentives contained within the protocol itself (e.g., bribery, threats of death); under that eventuality applications may or may not maintain their guarantees, depending on the details of the application and the type of external incentive. Without direct control of a counterparty's entire payoff table, no local economic mechanism can do better than imposing some cost on deviations from incentivized behavior. We do note, however, that some guarantees depend more directly on cryptographic assumptions than on economic ones, and that these guarantees may persist even if a counterparty is paid arbitrarily large sums to deviate from local incentives.

***3.1.4 Absence of Software Errors.*** Software errors are not explicitly handled within these models. Errors in software, especially when they involve key handling or smart contract code, can result in total loss or destruction of state for some or all parties. Nonetheless we include the following desideratum for all state channel designs:

> **The Recoverability Desideratum**: Whenever possible state channel software should be designed so that failures within the channel are recoverable, provided that all parties to the channel cooperate in enacting any necessary steps.

This desideratum explicitly leans on an "everyone is honest" assumption as the condition for recovering from software errors. In the worst case, a smart contract bug could allow one party to steal all the funds in the channel. Clearly, in this case the strategy of "steal all the funds" dominates all others. We believe it is worthwhile to support this type of recovery because there will be many cases where external incentives (e.g., reputation) outweigh local financial incentives.

[2]Precisely how many records of both kind need to be stored depends on the protocol design itself; in general, there is a trade-off between minimizing communication overhead and minimizing message storage requirements.

### 3.2 Griefing

In analyzing cryptoeconomic systems, griefing is the ability of a participant to deviate from protocol-specified behaviour, possibly at some cost to themself, in order to harm another participant, without directly benefiting the griefer[3].

In the following two sections, we define two main griefing strategies that our threat models have to take into consideration, namely unavailability griefing and posting stale state.

### 3.3 Unavailability Griefing

We define "unavailability griefing" as the specific strategy of "not being available", that is, deviating from the following behaviour:

> **Available Behaviour**: A user realizes available behaviour when he signs all valid state updates and proposes updates off-chain instead of on-chain where possible.

Available behaviour models parties cooperating to realise the responsiveness and off-chain desiderata as much as possible. If all parties behave in this way, transaction fees are reduced to the absolute minimum. An attacker who deviates from it is said to grief other members of the channel, and does so by incurring transaction fees. For example, consider a state channel between Alice and Bob. Alice proposes a valid state update, but Bob simply chooses not to sign it. Alice is forced to go to chain with the most recently accepted state, which requires Alice to pay an on-chain transaction fee.

This same griefing strategy also exist in other cryptoeconomic systems; we give two representative examples of concrete attacks. In Casper, any validator can stop casting votes, causing all validators to lose money. In a Plasma "Proof of Authority" chain, the authority can stop committing any headers, forcing all participants to withdraw state to the main ethereum chain, paying fees in the process.

#### *3.3.1 Why Unavailability Griefing Strategies Exist.*

In all the examples presented above, the griefing strategy is not punishable because of **speaker/listener fault equivalence** [26]: the blockchain cannot distinguish between the case where Bob is unavailable, and the case where Alice is merely claiming that Bob is unavailable. Another way to say this is that unavailability is not a uniquely attributable fault [2]. If such a dispute happens in a channel, Alice's act of going on-chain is equivalent to claiming that Bob is unavailable, but the blockchain cannot tell which of them are at fault.

#### *3.3.2 Mitigating Unavailability Griefing.*

Techniques to mitigate unavailability griefing are an active area of research. There are several potential solutions that could be used in state channels:

1. Use a trusted third party as a "witness", relying on them as the source of truth for data availability. This is a trusted party in the sense that a malicious witness, cooperating with a malicious channel participant, would be able to unfairly impose penalties on other channel participants.

2. Use a "semitrusted" third party who can gain private knowledge about data availability in the channel. The third party is trusted to use this information to e.g. set insurance premiums for griefing insurance. See section 7.2.

[3] If the deviation benefits the attacker, this means the protocol is not incentive-compatible, which is a more serious weakness than the possibility of griefing.

3. Commit to respond a certain way, analogous to "throwing away the steering wheel" in a game of chicken, or comitting to launch a second-strike nuclear retaliation in Mutually Assured Destruction scenarios.

4. Pooling data availability in such a way that an attacker must grief a large number of channels or none of them.

### 3.4 Posting Stale State

In contrast to unavailability, posting stale state is in most cases an attributable fault. If an attacker successfully submits old state and that state does not get challenged, he will have reverted state that is finalized in a channel. A concrete example of such an attack in a payment channel is for an attacker to complete a payment via a channel, receive the good in return, and then some time later submit the old state on-chain. We can model posting stale state as deviation from the following behaviour:

> **Up-to-date Behaviour**: In cases where it is necessary to post state to chain, such as when responding to another user or when dealing with an unavailable counterparty, a user realizes up-to-date behaviour when he posts the most recent state that he is able to post.

The chance of an attack succeeding and hence the incentive to deviate is excacerbated by the presence of environmental risks that cannot be completely eliminated:

- Chain congestion: large amounts of high fee transactions are submitted to the network by other users, temporarily raising both the size of fee required for timely execution of transactions and the uncertainty that any given transaction will enter the mempool of a randomly selected miner's node.

- Network disconnection: a user's computer, cell phone, or other channel utilizing device becomes disconnected from the internet due to power failure, telecommunications issue, or other cause.

Exogenous sources of chain congestion, i.e. sources outside the control of any party in a channel, will always occur. We observe that transaction load of public blockchains exhibits very "spikey" behaviour (e.g. Status ICO, Cryptokitties launch), which combines with the inelastic supply of transaction capacity to lead to periods of extremely high fees. This causes problems on-chain as well, for e.g., bitcoin "dust UTXOs" [7] with denominations too small to spend (relative to high prevailing transaction costs). Chain congestion need not even be completely outside the control of an attacker: groups of people who hold significant assets in channels and would collectively benefit from closing out stale state could collude to flood the chain with transactions or otherwise DOS the network. If it is necessary to contemplate coordination of this kind, more complex attacks are possible, for instance miners colluding to censor closeout transactions.

***3.4.1 Detecting Attacks.*** In general, we must assume that there is some base non-zero probablility that any attack will succeed, suggesting that in the absence of penalties for submitting stale state, it "never hurts to try". This suggests that the following rule be adopted:

> **Punish Stale State Protocol**: If provably stale state is submitted, the submitter is penalized.

Note that not all cancelled exits are "provably stale state". In the case of a payment channel, an attempted closeout with nonce $n$ can be cancelled by another with nonce $n+1$, but that does not mean that the first submitter (call her Alice) acted maliciously. This cancellation could happen if Alice has proposed an update to nonce $n+1$ and sent her signature confirming that update, but the counterparty has not responded to it. In that case, Alice must go to chain with nonce $n$, knowing that the counterparty has nonce $n+1$. Hence, the behaviour of the counterparty constitutes unavailability griefing. In general, the problem of making sure that this does not happen, i.e., that either all parties receive the signatures for nonce $n+1$ or none of them do, is equivalent to fair exchange of signatures, which is impossible without a third party [20]. The case where Alice goes to chain with nonce $n$ is equivalent to her using the chain as a third party, and paying fees for it.

A way to maximize the amount of provably stale state is for clients to adopt the following rule:

> **No-Skip Client Rule**: For sequential ordered updates, do not sign two updates in a row if a countersignature has not been received for the first update.

where a sequential ordered updates refers to cases like nonces in a payment channel. The protocol can then identify submissions of provably stale state and punish them.

> **No-Skip Protocol For Payment Channels**: If a participant $p$ begins a dispute and posts an update with nonce $n$ which is later overwritten by an update with nonce strictly greater than $n+1$, then $p$ loses a security deposit.

This can also be relaxed so that $p$ simply pays for the fees for the second update instead of losing a security deposit. Of course, adopting no-skip behaviour increases communication costs, and if the value at risk is not very high we might not want to incur this cost. In later sections, we discuss a construction that uses a root nonce to allow us to control this trade-off more finely by choosing when to "collapse" a channel, such that no application state is changed, but all signed messages before the collapsed update is provably stale.

***3.4.2 Penalties.*** The appropriate penalty for submitting provably stale state will depend on the amount of money at risk and the likelihood that an attack will succeed. The probability of success for any attack will depend on the length of the closeout window.

### 3.5 Initiator-Pays-Fees

We analyze a protocol where the initiating party pays for fees in a dispute. This is useful to analyze because it is easy to implement and because many existing channel implementations [6, 24] use it.

> **Initiator-Pays-Fees Protocol**: When a dispute goes on-chain, the initiating party pays the transaction fee.

This protocol is incentive-compatable in that available behaviour is a Nash equilibrium, provided the total possible reduction in utility caused by griefing is less than any single party's ownership of the state deposit. Most of the time, it is also the only Nash equilibrium, the exception being when one party can grief the other, e.g., at the last move of a chess game, in which case both available behaviour and a non-available strategy of not signing the last update are Nash equilibria. A participant willing to enter all state channels under this protocol implicitly assumes a threat model of "the attacker is not able to deviate from protocol-specified behaviour unless it strictly benefits them". We consider this an unrealistic threat model.

### 3.6 Economic Risk

Having discussed how protocol parameters can be tuned to handle expectations of a counterparty's ability to attacks by committing attributable faults, we turn to the question of designing protocols and choosing threat models to handle expectations of a counterparty's ability or likelihood to perform unavailability griefing, which is not an attributable fault. To do this, we introduce the concept of economic risk, the value (utility) that a participant can lose through griefing. We introduce three properties a threat model can have with regard to the amount of economic risk.

***3.6.1 No Risk.*** In these threat models, a user bears no economic risk, i.e., he requires that counterparties cannot reduce his payoff (compared to the case where the counterparty were not griefing him). These are the most constraining threat models; very few interactions can be conducted under them. If a protocol admits griefing, then clearly this risk model cannot be one adopted by both parties. Indeed, even in the case where there is arguably "no application logic" (i.e., payment channels) griefing is possible because someone must pay the fees for setting up the initial state-deposit-holder. After that point, the counterparty can walk away; this is yet another consequence of the impossibility of fair exchange without a third party.

However, it is possible for one party in a state channel to operate under this threat model, provided another party doesn't. An example of this would be a one-way payment channel where the payer is required to pay all the fees for setting up a channel. In this case the payer would be relying on external incentives like reputation. Another example where the roles are reversed would be a well-funded startup offering a dapp to pay for people to set up the requisite channels and relying on some external anti-sybil metric (e.g., send us a photo with your passport) as an external mechanism to limit their financial risk.

***3.6.2 Bounded Economic Risk.*** In these threat models, a user is willing to lose a bounded amount of utility to griefing. This is a useful threat model because many users will happily expose themselves to some small amount of risk in exchange for large improvements in performance or capability in the average-case. Of course, any state channel interaction has an implicit bound because users always have the option to walk away, letting the counterparty claim all the state deposits. What we mean by bounded economic risk is a case where the bound is below the value of the state deposit - typically much lower. This implies that users would only participate in interactions of bounded length, so that the cost of carrying out the entire interaction off-chain is below his threshold. For instance, a tic-tac-toe game lasts at most 9 moves, and a user would be willing to participate in it if the expected cost of making 9 on-chain moves is below his threshold.

For games without an explicit bound on the length, we might modify the rules of the game to enforce one, for instance:

> **Bounded Economic Risk Protocol For Chess**: When more than $k$ moves are played on-chain and the game has not ended, the pot is returned to the players (i.e, split evenly).

Modifying the protocol in this way may change the nature of the game. For instance, in the above protocol with $k = 2$ any player not one move away from checkmate can play the move "proceed to a draw". Clearly, this is a different game from chess, and not a very interesting one to play, and this continues to hold for small $k$.

"Economic risk" must capture more than "on-chain state with economic value"; it must include anything that someone would not want to lose, for example time value of money locked up, or the mental effort expended in playing a chess game.

***3.6.3 Bounded Griefing Factor.*** The griefing factor is the ratio of the cost incurred by an attacker to the ratio of damage he does. For instance, 1:2 means the attacker has to spend $1 to destroy $2 of value. When we say bounded griefing factor, we mean that the griefing factor for every participant is bounded away from zero. This is a useful threat model since people will accept larger amounts of risk if another participant must still incur significant costs in order to impose the realisation of those risks. The speaker-listener fault equivalence shows that the griefing factor bounds cannot all be better than 1:1; a simple protocol that realises 1:1 asymptotically in a 2-party channel is:

> **1:1 Griefing Factor Protocol**: For on-chain moves, transaction costs must be split equally between participants

Per-channel bounds are not enough because they may still permit a form of mass griefing. However, whether even a globally bounded griefing factor is sufficient depends on a user's model of the degree to which their own harm ranks highly in utility for agents they expect to interact with via channels. Further, if we have a bounded griefing factor without bounded economic risk, available behaviour becomes an equilibrium again, but this leans very heavily on assuming parties have extremely large budgets. Realistically speaking, a bounded griefing factor threat model should also implement bounded economic risk.

### 3.7 Other useful threat models

There are other useful threat models that we have not explored, including privacy with respect to counterparties and privacy with respect to intermediate nodes. As an example, for some applications, parties might decide to encode their interaction as a state machine and only counterfactually instantiate a zero-knowledge verifier for the state transition function, meaning they can have disputes on-chain without revealing sensitive information regard what their interaction is about. The existing literature on payment channels also considers privacy with respect to payment amount and routed path (for instance, a routed payment sometimes reveals the amount, the sender and the receiver at least to all intermediary nodes).

Also, the response delegation model is not yet included here (see section 7.2). We shall briefly say that delegation of the availability assumption implies a "1 of $n$" security criterion where all $n$ outsourced services must collude and must either give up their revenue or lose some deposit (depending on the design).

## 4 Previous Work

The state channel technique - locked state deposits and modification of off-chain state via total consent, built on top of a public blockchain to provide security but with zero marginal transactions in the optimistic case - has been used by many projects before ours. Research has explored two important and roughly orthogonal dimensions of the design space: what can be done within a channel, and how channels can be composed into networks. We provide an overview of ideas explored in prior / parallel research along both these dimensions, followed by an analysis of specific projects using our requirements and objectives as well as pointing out how they differ from our work.

### 4.1 Generality Within A Channel

Improving the "generality" of a channel refers to expanding the set of what can be done within it; for instance, state channels were conceived as a generalization of payment channels in the sense that a payment channel can be implemented as a specific type of state channel.

#### 4.1.1 *Payment Channels.*

Payment channels can be seen as an application of the state channel technique for the specific use-case of payments (off-chain token transfer). Proposals for building payment channels on top of Bitcoin were slowly developed through work from Mike Hearn, Alex Aakselrod and bitcointalk forum user hashcoin (for a more comprehensive historical overview, see [5]). Even today, a large amount of research and projects focus on or restrict themselves to the use-case of payment channels. Since payment channels only try to support a small number of applications, some of our criteria (e.g. installing new applications) do not apply to them.

#### 4.1.2 *State Channels.*

Beyond payment channels, the idea of using the state channel technique for arbitrary state transitions (not just payments) has been less intensively explored but still relatively well-known, with multiple projects like Gnosis [16] and Funfair [14] building production applications employing this technique of off-chain trading in prediction markets and for trustless casino games respectively. In the community, these are termed "state channels", often to contrast them to payment channels. An early description of this was in a blog post by the first author [12], and an early proof-of-concept code produced independently by [21], which stores a chess board as counterfactual state and uses interactive verification for checkmate computation.

#### 4.1.3 *Generalized State Channels.*

Our work generalizes existing state channels by introducing new techniques such as counterfactual instantiation as well as a contract-based design framework, both of which we refer to under the term "generalized state channels". When compared to existing "application-specific" state channels, the most apparent difference to the end-user are channelized installation of new functionality, which makes it cheaper (no transactions), instant, and optimistically privacy-preserving. A similar kind of "off-chain instantiation" was independently discovered and implemented in Perun [24] (discussed in section 4.3.4).

### 4.2 Interacting Across Intermediaries

Much existing research on payment channels have focused on exploring the design space of how to structure payments through intermediaries. Suppose that Alice has a payment channel with Bob, and Bob has one with Carol. If Alice pays Bob off-chain and then Bob pays Carol the same amount, this is equivalent to Alice paying Carol off-chain, without requiring a new Alice-Carol payment channel to be set up.

There are various ways to do this trustlessly, i.e., to ensure that Alice pays Bob iff Bob pays Carol. In a Lightning-style Hashed Timelock Contract (HTLC), an equal amount of funds from both payment channels are locked up in a way that they can only be spent if a certain hash is revealed before a certain deadline (thus, locked "by hash" and "by time"). From Bob's point of view, if the locks are set up with the same hash but where the outgoing lock has an earlier deadline than the incoming lock, he is guaranteed that if the outgoing payment is made, he can force the incoming payment to be made as well. For consistency of terminology, we refer to these "direct" payment channels as one-hop payment channels.

The Sprites project [6] observed that this mechanism implies deadlines must be staggered, which implies worst-case capital lockup time grows linearly in the number of hops and hence a total capital lockup cost that grows quadratically in the number of hops. They designed an alternative solution where hash revelation deadline is set to be earlier than the dispute resolution deadline, allowing all locks to have the same deadline. Hence, capital lockup time is independent of the number of hops, and total capital lockup cost is linear in the number of hops.

In contrast to both these schemes, our metachannels construction uses a "rent-a-path" scheme. Instead of setting up a chain of smart contracts per payment, we use a chain of smart contracts to lock up an intermediary's funds for a certain amount of time, during which arbitrarily large numbers of off-chain payments can be made. In the specific case of payment channel networks, the difference

is analogous to the one between circuit-switched and packet-switched networks. Our asymptotic capital costs and lockup times are the same as for sprites-style HTLC. Using the object-oriented framework, we can also build hash timelocked conditional payments, allowing users to implement Lightning-style or Sprites-style payment routing if desired.

### 4.3 Existing Designs

***4.3.1 Lightning Network*** The Lightning Network is arguably the most well-known project using payment channel techniques, and in the short term will likely see the largest tranasction volume by value. It builds off-chain micropayments on top of Bitcoin using some special-purpose opcodes. Lightning Network was proposed in [23] and currently specified in [11].

A Lightning Network channel uses funds locked in a **funding transaction**, a multisignature UTXO created and owned by the parties in the channel. Balance updates are done by signing two assymetrical **commitment transactions**, each of which spends the funding transaction, immediately releases **remote funds** to the counterparty, and start a challenge period after which the broadcaster receives the remaining **local funds**. Commitment transactions have a revocation mechanism: old transactions are explicitly revoked by revelation of a revocation key, and a stale commitment transaction TXO allows anyone with the revocation key to claim all the local funds. In a simple design, all old revocation keys must be stored in memory, making the client-side memory requirement linear in the number of payments. The challenge period is implemented using the OP_CHECKSEQUENCEVERIFY relative timelock opcode, so channels can open for an indefinite amount of time. Cooperative instant withdrawal is supported using **closing transactions** that directly spend the funding transaction; hence, cooperatively closed channels are indistinguishable from a normal multisig. Partial closeout can be supported by cooperatively spending the funding transaction into two outputs, one partial payout and one new funding transaction.

***4.3.2 Raiden*** The Raiden project [3] aims to build payment channel networks using Ethereum smart contracts, supporting ether and ERC20 tokens. Raiden networks can remain open for an indefinite period of time. Cooperative instant withdrawal is not supported, but there is an open issue to add it. Raiden uses the same hashlock mechanism used by the Lightning Network, that is, an intermediary uses different expiration times, which means an $l$-hop path locks up collateral for $O(l)$ time. In addition to payments, Raiden also plans to support atomic swaps. The Raiden project has also built *microraiden*, which has a smaller feature set than "full Raiden", as it supports only unidirectional payment channels with no support for transitive payments.

***4.3.3 Sprites*** The Sprites project [6] constructs payment channel networks that reduce the worst-case collateral cost over Lightning-style HTLC, using what we have been calling Sprites-style HTLC. In addition, they construct state channels to use as a basic "building block" for both one-hop payment channels and for Sprites-style HTLCs. Their state channels are similar to ours in two ways. First, the state transition function can depend on on-chain state (indeed, this functionality is used to refer to a common "source of truth" to determine if the HTLC's hash was revealed in time). Secondly, arbitrary state deposits are supported, since users commit to allowing the state channel to call `handleOutputs` in certain circumstances; indeed, for their payment channels, a separate contract is used to hold the deposited coins. Sprites-style payment channels differ from ours in that they are not meant to be counterfactually instantiated, and fix some design choices such as the choice to use scalar nonces, the signature algorithm, and using a `bytes32` to store state. Sprites-style payment channels also do not support cooperative withdrawal.

***4.3.4 Perun*** Perun channels have three innovations compared to other payment channels which are shared by our approach: first, their technique for routing-through-intermediaries, which they

call **virtual channels**, is very similar to ours in that the virtual channel is "noninteractive" and does not require cooperation from the intermediary for every payment. Secondly, Perun state channels possess the same modularity / parallelism structure that ours does. In their terminology, the state $\sigma$ of a **multistate channel** is split into "substates" $(\sigma_1, \sigma_2 \dots \sigma_n)$ that can be updated and disputed in parallel, which corresponds to our splitting of the counterfactual state within a state channel into counterfactually instantiated contracts[4]. Such parallelism is essential for supporting the rent-an-intermediary paradigm. Thirdly, the set of "substates" can be updated off-chain (e.g., the state of a multistate channel can go from $(\sigma_1, \sigma_2)$ to $(\sigma_1, \sigma_2, \sigma_3)$ without an onchain transaction, which in our paradigm corresponds to counterfactually instantiating a contract with state $\sigma_3$), which allows for installing new applications off-chain.

However, Perun channels lack counterfactual addressing and counterfactual instantiation of code (only state is counterfactually instantiated). This means that while new applications can be installed on-chain, the code for that functionality must already be deployed on-chain. This also means that nanocontracts cannot refer to each other, limiting their compositionality. In addition, there are no generalized state deposits; nanocontracts explicitly "block" a certain amount of ether (and only ether, as an enshrined currency in their construction) and financial outcomes of nanocontracts modify ownership of blocked ether through an explicitly defined API.

The Perun paper itself only discusses multihop payment channels, and construction of multihop state channels and payment channels of length $> 2$ was described in a followup paper [25]. Their construction, which is summarized as recursively building virtual channels, differs from ours, mainly due to the lack of counterfactual addressing. In our paradigm, an intermediary locks up some state to construct a metachannel with a payment object holding the locked state, and the parties of the metachannel then counterfactually instantiate any functionality they want and fund it from the payment object. However, in Perun's paradigm, a dispute in a multihop virtual channel must be (trustlessly) resolved through the intermediary. More concretely, if there is an Alice-Bob metachannel through Ingrid with a lot of functionality instantiated, Alice can grief Ingrid by forcing her to play out all the moves in the metachannel, while in our construction Alice can only do so to Bob, and Alice can only grief Ingrid for the value of the fees needed to instantiate up to the payment object.

## 5 Concepts and Definitions

The core technique used by any state channel is the leveraging of as-yet unrealized on-chain consequences for the enforcement and security of purely off-chain activities. If the channel is constructed properly, many of these consequences will *never* be realized, even in expectation. Yet their presence in the game theoretical model is crucial to the channel's design and safety, incentivizing participants to avoid taking actions that might make these negative consequences actually occur.

A necessary step for any state channel setup, then, must be the transfer of some on-chain state (ether, ownership of tokens, etc) away from the unilateral control of an individual participant so that enforcement actions are able to modify that state. These assets comprise the **state deposit** and are "locked up" for as long as they are in the state channel. After locking up the state deposit, participants can interact "off-chain" within the state channel simply by passing special types of messages to each other[5]. The passing of these messages results in a change of the on-chain leverage

[4] Perun defines state channels in contrast to multistate channels as channels that do not partition the storage in this way, a distinction we do not make. Furthermore, the substate and functionality that uses it is grouped into a "nanocontract".

[5] Here "off-chain" means roughly that each interaction does not require a transaction on the base ethereum chain in the optimistic case and hence incurs zero marginal transaction fees, although of course the actual messaging incurs

(that is, the types of on-chain actions participants are able to perform); in fact, because of our definition of "off-chain interaction", this is the only thing they can do. However, this seemingly limited capability provides enough power to replace some on-chain operations. For instance, in a payment channel, the updates "only" change the leverage, but that is enough that participants can treat the latest signed balance as "the amount of ether I have in this payment channel"[6]. Notice also that while we have some concept of "off-chain state" and "off-chain transactions", no EVM is actually running off-chain; the messages passed between parties to a channel are less about the task of *computing* the state, and more about the task of *convincing* another party that the state can be considered final[7].

### 5.1 Counterfactual Terminology

Since state channel interactions are all about modifying leverage, we spend a lot of time discussing what-could-happen-on-the-blockchain-but-doesn't. On-chain enforcement can be extremely complicated and indeed *needs* to be so, since the full functionality of generalized state channels is implemented simply by modifying the enforcement leverage. Counterfactual terminology is a way to decompose the leverage into smaller, more modular pieces and to discuss it without resorting to long and heavily qualified sentence constructions. Roughly speaking, for any on-chain operation **X** that can be channelized, we use **counterfactual X** to talk about the case where

1. X could happen on-chain, but doesn't
2. The enforcement mechanism allows any participant to unilaterally make X happen
3. Participants can act as though X has happened

For instance, in a payment channel, **X** could be "4 ether is transferred from the smart contract to Alice's account, and 6 ether from the smart contract to Bob's account", and **counterfactual X** would be the state of affairs if both parties have the latest signed copy, which records Alice's balance as 4 and Bob's as 6.

A more complete treatment of counterfactual terminology is more complicated, for example requiring a definition of participants, their capabilities and incentives, "classes" of on-chain operations, and what exactly clause 3 means to participant. See appendix A for a full treatment; however, we recommend reading this section and the following first, so as to have an intuitive understanding of some counterfactual terminology before working through the general definitions.

### 5.2 Counterfactual State

Let us look more closely at how this terminology can be used to describe existing payment and state channel constructions. In ethereum, an account's **state** consists of its nonce, ether balance, contract code, and storage. A properly constructed smart contract will hold **counterfactual state** when every person in some prespecified set of users (chosen to include those affected by the counterfactual state) can unilaterally update the on-chain state to the given counterfactual state.

For example, let us describe a timeout/challenge/finalize mechanism used in existing literature. Counterfactual state is divided into **nonce** and **application-specific state**, where the nonce is

---

some computational costs.

[6] Using our terminology, existing payment channel solutions use the same smart contract to hold the state deposit and to carry out enforcement; off-chain messages contain signed messages that the contract interprets.

[7] In practice, application writers will define some standard message templates which user client software can simply check messages against, instead of requiring users to reason about the messages "from scratch".

a single `uint256`, and the contract defines a fixed participant set. Participants sign updates that include the new application-specific counterfactual state, and updates must increase the nonce. For a payment channel, the application-specific state is a mapping from participant to their balance. For a channelized chess game, the state of a chessboard serves as the application-specific state. During an on-chain dispute the counterfactual state is treated as temporary until the object enters a **finalized** state when a signed update is submitted and a challenge period ellapses without newer state being submitted[8].

### 5.3 Counterfactual Instantiation

Ethereum provides developers with a general-purpose computation platform by supporting smart contracts. Similarly, participants in a state channel should be able to **counterfactually instantiate** smart contracts within a state channel, installing new functionality[9] into the state channel without any on-chain action. After counterfactual instantiation, users will be bound by the terms of the counterfactually instantiated contract.

On-chain smart contracts, generally speaking, are only useful if they control some valuable on-chain state. Since counterfactually instantiated contracts are not actually deployed on-chain, they cannot directly take ownership of on-chain assets. Participants must instead sign commitments to the state-deposit-holder so that the enforcement process uses the counterfactually instantiated contract in some way. Hence, a portion of the state deposit is assigned to the counterfactually instantiated contract. As part of the enforcement leverage, a participant must be able to unilaterally instantiate the counterfactually instantiated contract on-chain and then have the contract control how the state deposit is disbursed.

***5.3.1 Counterfactual Addressing.*** A potential problem arises because in general we do not know the address at which an ethereum contract will be deployed until after it is deployed, since currently the address of a contract depends on the contract creator's address and nonce [19]. The state-deposit-holder does not know in advance which address the counterfactually instantiated contract will have, but needs to make commitments that involve this address. For the same reason, counterfactually instantiated contracts cannot directly refer to each other.

To resolve this problem, we provide a global registry contract that is deployed on-chain to allow referrals to counterfactually instantiated contracts. Here "global" means that one instance of it is shared among all state channels. The registry needs to define the **counterfactual address** of a contract as a deterministic function of the contract's initialization code. The registry provides a **resolve** function that maps counterfactual addresses to ethereum addresses (similar to how DNS maps domain names to IP addresses), and a **deploy** function to deploy new contracts and register them.

A contract is counterfactually instantiated by computing its counterfactual address, and then signing some commitments to the state-deposit-holder that assigns some state deposit to it. Note that *counterfactual instantiation is the act of signing these commitments, not the act of actually calling deploy on-chain.* Once the contract is counterfactually instantiated, users can use its counterfactual address to refer to it.[10]

---

[8] There are other ways to support counterfactual state; in the game of tic-tac-toe, the state of the board can serve as both application-specific state as well as the nonce, because game moves increase the number of filled squares by exactly one. We emphasize that the contracts must be carefully constructed to support counterfactual state.

[9] For instance, games invented after the channel was set up.

[10] Alternatively, any code-based addressing scheme provided by the EVM would also solve this problem and thereby allow for counterfactual instantiation without a registry. For instance, the account abstraction EIP [27] or the skinny CREATE2 EIP both propose changes to make the address depend on the code and a chosen salt.

### 5.4 Minimizing On-Chain Functionality and State

Counterfactual instantiation allows us to instantiate application-specific smart contracts off-chain, but we can also view it more abstractly as a technique to move functionality and state off-chain. Since we want to do this as much as possible, we also counterfactually instantiate the dispute-resolution functionality (e.g. challenge/timeouts[11]). The state deposit holder can then be an $n$-of-$n$ multisignature wallet, and is the only on-chain component that must be deployed for each additional state channel. We think this is the minimum amount of functionality and state that must be placed on-chain; the state deposit must be held by an entity that implements unanimous-consent based operations on it, and a multisig implements exactly this and nothing more. We mark this out as one of the important insights in this paper:

> A sufficiently powerful multisignature wallet is sufficient to act as a state deposit holder.

This is not only *possible*, but in fact a useful design choice offering two main benefits:

1. Privacy. The state channel participants can choose to not share their signed commitments, in which case an external observer will only see a multisig wallet on-chain; participants who wish to hide the existence of a state channel between them can use "the set of all on-chain multisigs" as an anonymity set.

2. Upgradability. If bugs are found in the dispute resolution code, or more generally any state channel code outside the multisig, they can be replaced off-chain if both parties cooperate.

An additional benefit of using a standard multisig wallet as the state-deposit-holder is that any asset which implements support for ownership by a multisig wallet can automatically be controlled by a state channel. For instance, an abstract asset like "the right to set an ENS name" can be placed in a state deposit.

### 5.5 Organizing Off-Chain State and Interactions

***5.5.1 Counterfactual Objects.*** As a modular building block for state channels, we define counterfactual objects as counterfactually instantiated contracts that contain counterfactual state. A state channel consists of a collection of counterfactual objects which refer to each other using counterfactual addressing. This allows us to compartmentalize the functionality, state, and interactions within a state channel as much as possible. Two participants can open a state channel containing micropayment functionality between them by using a counterfactually instantiated payment channel object, and then some time later decide to play an off-chain game of chess by counterfactually instantiating a chess counterfactual object. The logic and state remain separate, and any interaction between them is minimized.

***5.5.2 Assigning Deposited State.*** For a counterfactual object $O$ to control some state deposit held by a multisig $M$, there must be a commitment for $M$ to disburse some portion of that state deposit using the finalized counterfactual state of $O$. For example, if $O$ were a payment counterfactual object with counterfactual address $c$ and counterfactual state $(a, b)$ representing Alice and Bob's balances respectively, there should be commitment from $M$ of the form "if the counterfactual object at counterfactual address $c$ were to exist in a finalized state with counterfactual state $(a, b)$

[11] We refer to these as non-application-specific because an equivalent non-channelized application need not contain this functionality

then send $a$ ether to Alice and $b$ to Bob". As can be seen, these commitments are quite complicated, so our multisig has to be able to perform complex, multi-opcode operations. Also, it is important that we use multisigs for which commitments, once made, cannot be unilaterally revoked.

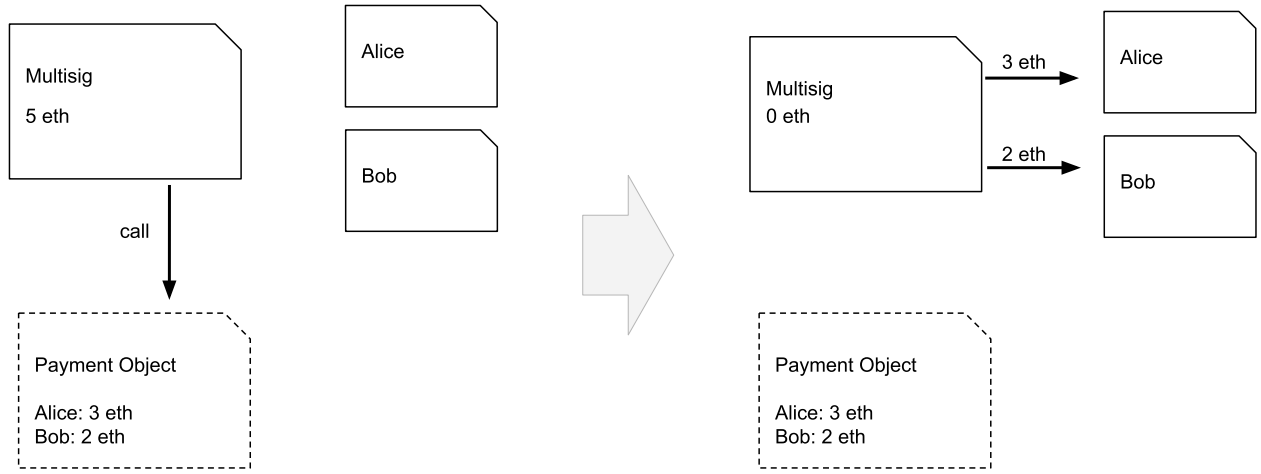



Figure 1: The multisig performs a call to the payment object and uses the result to determine the amounts transferred to Alice and Bob.

***5.5.3 Relations Between Counterfactual Objects.*** Counterfactual objects can depend on each other in many ways. An object can be written so that its constructor reverts unless another counterfactually instantiated contract exists. Alternatively, the objects can refuse to transition into some finalized state until another object is finalized, which we call **conditional finalization**. In both cases, counterfactual addressing is necessary to refer to the other contract. These relations can in theory become arbitrarily complex; an object can refuse to finalize until some other set of objects exists, and one of them has its nonce in a given range, and some other set does not exist, etc.

The counterfactual state of a state channel refers to the local counterfactual state of each object as well as the network of relations between them. When we say that state channels operate by unanimous consent, we mean that no party should be able to unilaterally change this state, unless all parties involved gave that party permission to do so previously.

***5.5.4 Nonce-Dependent Conditional Finalization.*** A specific kind of conditional finalization called nonce-dependent conditional finalization is a key tool we use for organizing counterfactual state. In it, a dependent object $V$ does not finalize until a depended-upon object $U$ is finalized at a given nonce $n$. Then we say that $V$ depends on $U$ being at nonce $n$.

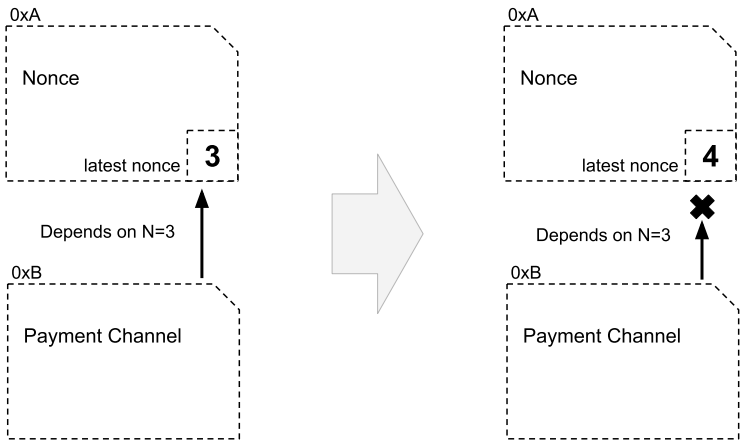



Figure 2: Nonce-Dependent Conditional Finalization. By incrementing the nonce object's nonce to 4, the payment channel object can no longer finalize.

Since nonces are also used for ordering messages, a potential problem arises if we wish to use the same nonce for updates to local states and for dependencies to depend on. For instance, if we had a chess game depend on a payment channel object, sending a micropayment would increment the nonce; hence every time we send the micropayment we have to make sure the chess object is not deleted. One way to avoid this is to not use the same nonce for both purposes, for e.g., by refactoring to use 3 nonces; one for dependency, and two for updates to local state.

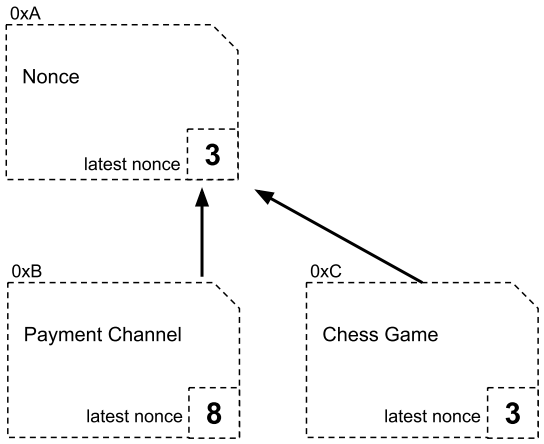



Figure 3: Multiple Dependents. The nonce of `0xA` is used purely for managing dependents, while the nonces of `0xB` and `0xC` are used to update local counterfactual state.

***5.5.5 Atomic State Transitions.*** Using multiple dependents we can atomically make complex state transitions. Let $c$ be a counterfactual object with some counterfactual state, which can take on values $a$ and $b$, and which is currently $a$. Set up one network of counterfactual objects whose existence depends on the state of $c$ being $a$, another network of counterfactual objects whose existence depends on the state of $c$ being $b$, and update the state of $c$ from $a$ to $b$. This also allows us to create more "conceptual" relations between objects. For example, we can counterfactually instantiate a poker counterfactual object and fund it from an existing payment channel by atomically reducing the payment channel balances and increasing the amount of state deposit assigned to the

poker counterfactual object, enforcing a conceptual **conservation of balance** relation between them.

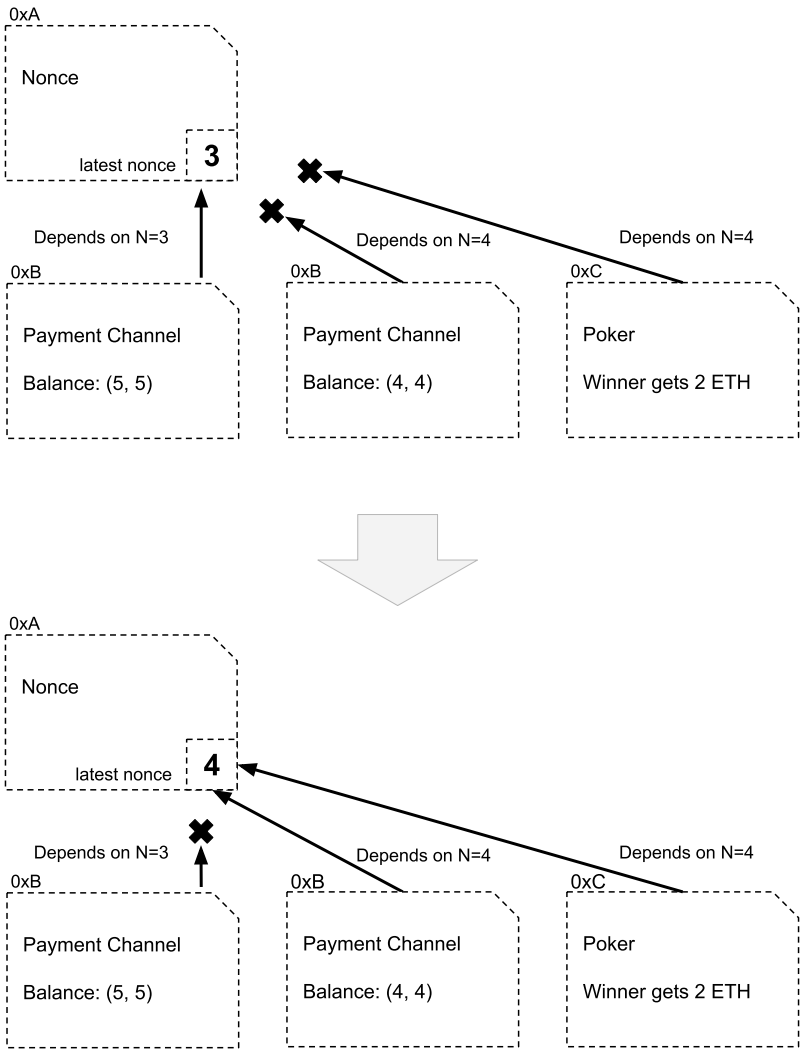



Figure 4: Atomic operations.

An alternative to using a dedicated nonce object is to use vector nonces, in which nonces are defined as a vector of natural numbers, and nonces can be compared lexicographically. Then, the dependent object can depend on the lower-order fields of the nonce.

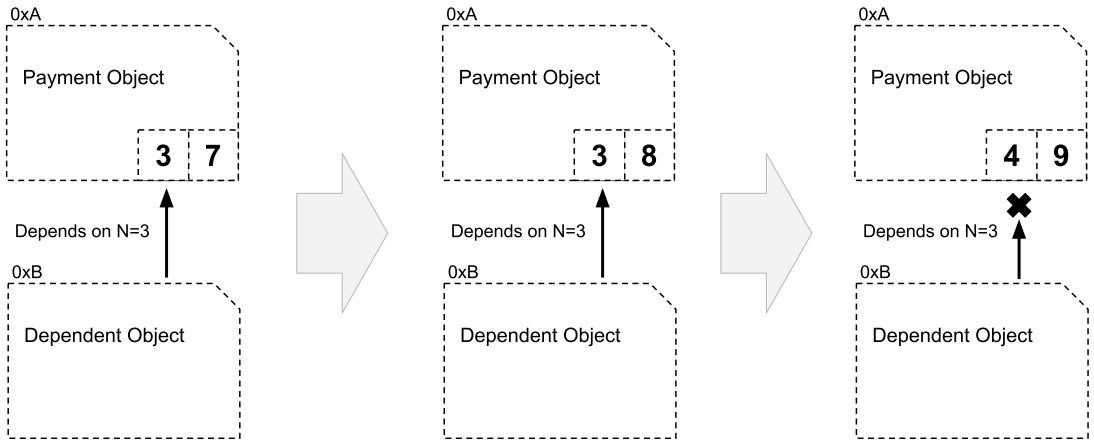



Figure 5: Vector Nonces. The balance is updated by increasing the least significant bit, while the dependent object remains in existence. Then, the balance is updated and the dependent object is deleted atomically.

Nonce-dependent conditional finalization applies transitively; if $A$ depends on $B$ depends on $C$, then $A$ cannot finalize unless $C$ finalizes. This creates a hierarchical relation, where objects can be arranged in a directed acyclic dependency graph. In order to prevent timeouts from "stacking" when trying to finalize a root-to-leaf path in an on-chain dispute, we use a scheme called recursive finalization; all the challenge timeouts can elapse at the same time, and when an object receives an isFinal query it recursively performs an isFinal query on its parent.

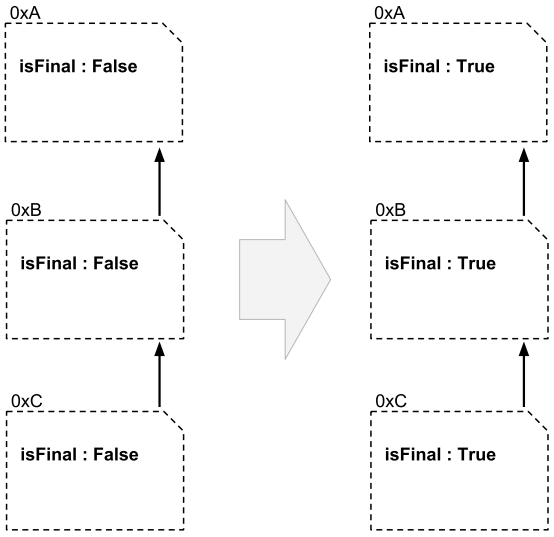



Figure 6: Hierarchical finalization. In a dispute, three counterfactual objects are finalized in a single transaction.

***5.5.6 Root Nonce.*** There should be a "root nonce" counterfactual object which has a scalar nonce, contains no state, and on which all other objects depend. The existence of the root nonce allows us to atomically make arbitrarily complex changes that affect multiple objects, including deleting objects. In addition, in disputes we can respond to stale state that depends on a stale value

of the root nonce by publishing an updated value of the root nonce. If all stale state depends on a stale value of the root nonce, we have an $O(1)$ response to all stale state[12], and we can "collapse" the state of the channel in this way right before going offline.

### 5.6 Interacting with On-Chain State

It is possible to channelize many applications that refer to on-chain state. For instance, two players may wish to enter a bet within a state channel on some well-known future event (e.g. the winner of the 2020 U.S. presidential election) and use an on-chain decentralized oracle like Augur. If the oracle is suitably designed, this is very straightforward; for instance, an Augur market has a fixed address and resolves into an immutable state, so a counterfactually instantiated contract can simply refer to that. This shows that state determined within a counterfactual object can change even when no messages have passed between the parties involved.

### 5.7 The Object-Oriented Approach

We describe our approach of organizing a state channel as counterfactual objects, each of which contain their own functionality and counterfactual state, as an "object-oriented" approach to state channels.

***5.7.1 Conditional Payments.*** One construction we advocate is to separate payment logic and application logic through the use of conditional payment counterfactual objects. A conditional payment counterfactual object contains functionality to observe another counterfactual object (by resolving the counterfacutal address and then performing a message call to it) and then to use the result to disburse the assigned state deposit. The observed object implements the application-specific functionality, and multiple conditional payment objects may observe the same counterfactual object (for instance, in an atomic swap of MKR and REP, the counterfactual payment objects for MKR and for REP should observe the same object). This design provides two advantages:

1. Locality of risk. Software bugs in application logic will only put at risk the value assigned to the conditional payment object, assuming there are no additional bugs in the conditional payment object. This is useful as while payment logic will be heavily scrutinized, a much larger volume and variety of application logic will be written, and bugs in them should be contained as much as possible.

2. Unrestricted state deposits. Anything that can be held by a multisig wallet can be held as a state deposit and assigned to a conditional payment counterfactual object. As a thought experiment, suppose some time in the future an ERC2000 spec is proposed which represents fungible tokens but is incompatible with ERC20; existing users will have no problems using ERC2000 tokens as state deposits.

***5.7.2 Instant Withdrawals and Top-Ups.*** An object-oriented approach easily supports **instant withdrawals** and **instant top-ups**, which means that additional state can be deposited (on-chain) into the channel or withdrawn, with no challenge period. Let us consider an instant withdrawal. First, some ether is assigned to a counterfactual object whose state is conditional on the balance of the multisig. Then, a withdrawal transaction atomically moves funds out of the state channel and reduces the balance assigned to the withdrawal recipient within the state

[12] Without this $O(1)$ response, a counterparty publishing old state in $N$ counterfactual objects might require us to respond with $N$ newer updates, costing $O(N)$ gas.

channel, while a top-up transaction atomically moves funds into the state channel and increases the balance assigned to the funder.

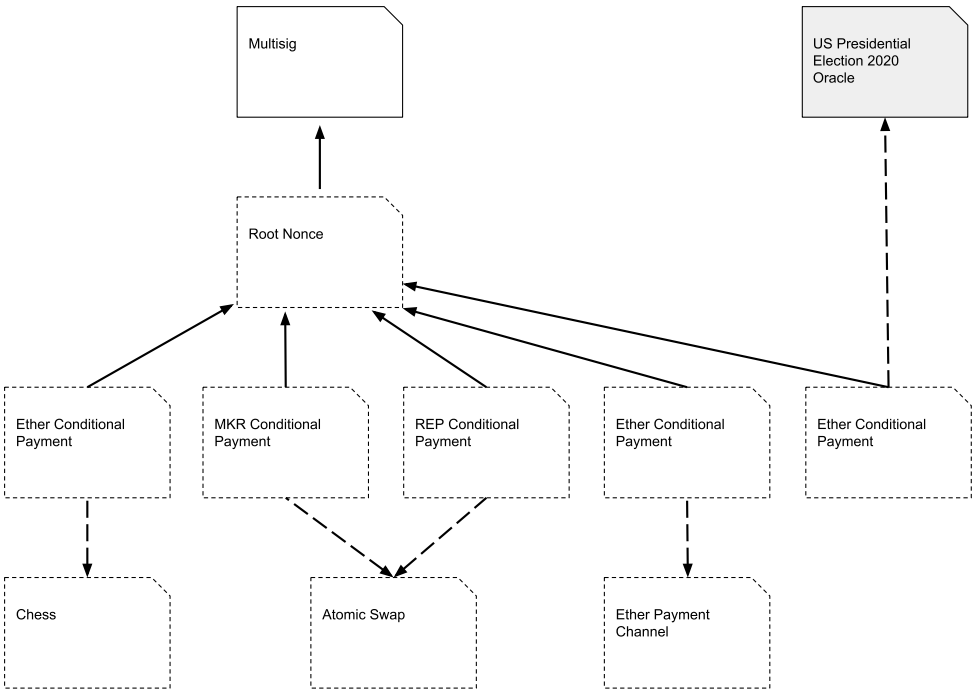



Figure 7: Structure of a typical state channel. Solid arrow denote the "depends-on" relation while dashed ones denote the "observes" relation.

***5.7.3 Benefits.*** We believe that this object-oriented approach maximizes flexibility and modularity by providing the following benefits.

1. Composability of functionality. Individual counterfactual objects can "plug in" with each other in a large variety of ways in a state channel. For instance, once a poker counterfactual object is written, it can be used in a state channel with any kind of pot (simply by having the appropriate kind of conditional payment channel look at it), with any randomness source (by having the poker object look at a "randomness game" counterfactual object). The application writer need only specify the rules of poker and need not be an expert in ERC20 token interfaces or randomness games.

2. Re-use of on-chain design techniques. Since the ethereum state is also organized in an object-oriented manner (smart contracts which combine functionality and state), the EVM is optimized for supporting this organization, and we can benefit from the same design techniques used in designing on-chain dapps. For instance, a common technique is for a widely used contract to share functionality (but not state) via proxy contracts [15], and we can use the exact same technique in channels.

3. Merkelized disputes. If Alice and Bob need to dispute a chess game on-chain, there is no reason to drag an unrelated atomic swap object on-chain. It also gives us a way to contain risks in application logic (bugs, etc) by limiting what is withdrawn to conditional payment channels.

### 5.8 Changing the participant set

Our state channels so far have been $n$-party state channels without any upper bound on $n$. While arbitrarily large payment channels can be decomposed into a collection of 2-party state channels plus payment routing, the same is not true for state channels. For instance, a 4-player poker game played with a nonfungible token (NFT) as part of the stake cannot be decomposed into 2-party state channels plus routing[13], and a channelized 4-player poker game must require consent from all 4 players to update the state.

Counterfactual objects can also specify who is allowed to modify their counterfactual state (e.g. the set of signatures required), in particular to be different from "all state channel participants", which is what we have implicitly been using. Suppose Alice, Bob and Carol set up a 3-party state channel; a payment subchannel which has owner set Alice and Bob has the property that consent from Carol is not required to change the balance[14], and so behaves much like a normal 2-party state channel between Alice and Bob.

### 5.9 Metachannels

Metachannels are an object-oriented solution to the problem of interacting across intermediaries. We describe them for the payments use-case first. Suppose Alice has a state channel with Ingrid, that Ingrid has a state channel with Bob, and that Alice wishes to set up a payment channel with Bob. First, we observe that Alice and Bob can create a counterfactually instantiated payment channel object owned by themselves. Call this object $O$. Now, we must make the counterfactual state of $O$ (i.e., Alice's balance and Bob's balance) meaningful. We do this by creating two **proxy payment** counterfactual objects, one each in the Alice-Ingrid and Ingrid-Bob channels, that do have state deposits assigned to them, and that observe $O$.

---

[13] This is true unless players agree to put a price on the token and use a fungible token as collateral for the NFT within their 2-party channels. However, this introduces an additional "bounded volatility" assumption; for example, in the case where the NFT ends up being worth more than the collateral, the game winner might end up receiving the less valuable collateral instead of the NFT.

[14] This implies that, even if this subchannel mantains Carol's balance, Carol cannot trust that she really owns that balance.

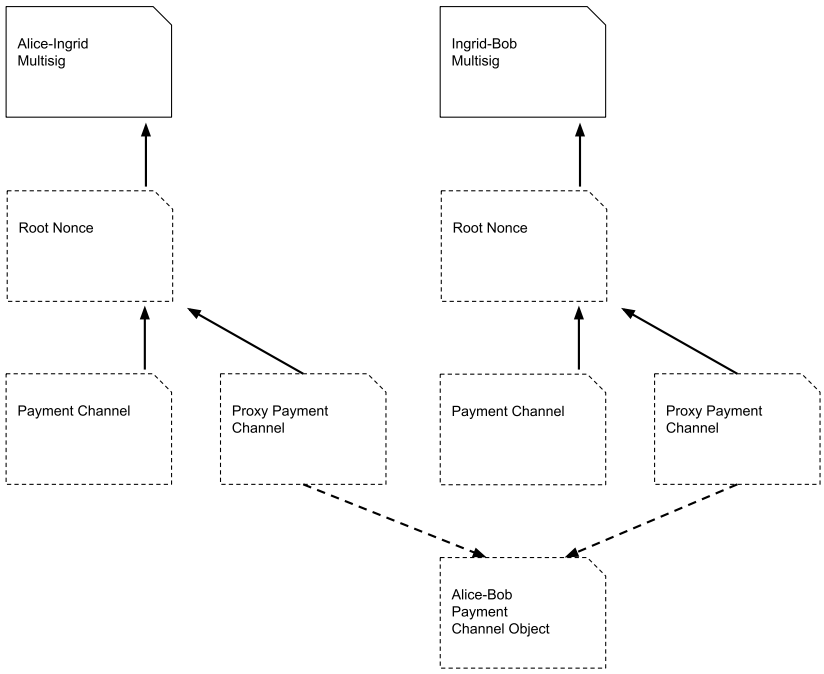



Figure 8: Structure of a metachannel between Alice and Bob through Ingrid, implementing payment channel functionality.

Suppose the Alice-Bob payment counterfactual object has counterfactual state $(a, b)$ representing balances of Alice and Bob respectively. The proxy payment object on the left assigns $a$ to Alice and $b$ to Ingrid, and the proxy payment object on the right assigns $a$ to Ingrid and $b$ to Bob. That way, Ingrid always has $a + b$ assigned to her.

We can use the same "assign Alice's balance to the left and Bob's balance to the right" to generalize this to intermediary chains of arbitrary length.

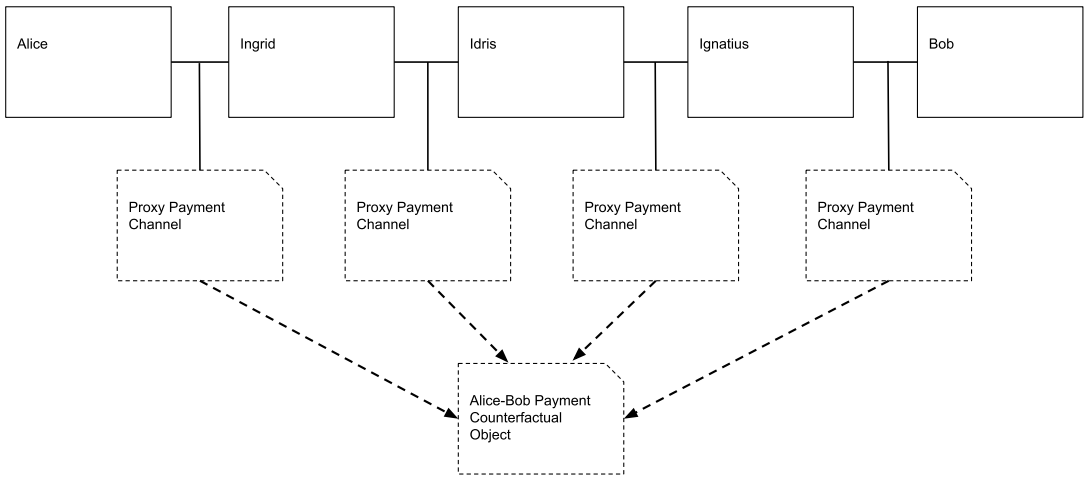



Figure 9: Metachannel between Alice and Bob through an intermediary chain of 3 intermediaries, implementing payment channel functionality.

Lastly, we observe that a sufficiently powerful multisig is sufficient for a state channel, and we

can place multisig functionality into $O$, thereby allowing $O$ to reassign its state deposit to other objects under the control of Alice and Bob. Whereas a normal multisig enforces commitments to it by making calls to other contracts (transferring ether, changing balances in an ERC20 contract), $O$ enforces commitments simply by writing to its own storage.

## 6 Constructing Generalized State Channels

In this section, we provide one construction for generalized state channels, i.e., a set of protocols for creating a state channel, updating it, installing new functionality, and withdrawing state. Note that we have made implementation choices to specify the protocol in detail (e.g., function signatures of contracts and a message format). Different choices are possible, and there are many specific protocols that use the same concepts as in section 5. The specification presented is one point in the design space, and we do not claim that this one is the most gas-optimal or the simplest.

We refer to on-chain contracts using a bold sans-serif font (e.g., ***multisig***). Counterfactually instantiated contract types are referred to with an italic and non-bold vesion of the same font (e.g., *RootNonce*). Protocols are written using a bold font (e.g., **Instantiate**).

### 6.1 Implementation Techniques

When implementing generalized state channels, various implementation choices have to be made that do not change the core concepts or definition, but which are nontrivial enough to warrant describing.

***6.1.1 Delegatecall-based commitment.*** As discussed in section 5, state deposits are assigned to counterfactual objects by signing commitments that commit the multisig to disbursing the state deposit according to the state of some deployed counterfactual object. Because of this, by "commitment" we mean a commitment to perform (atomically) a complex, multi-opcode operation with conditional logic, and the multisig wallet must be able to perform such operations. Since ethereum does not directly support executing opcodes provided via transaction data (e.g., through a hypothetical EVAL opcode), we use the DELEGATECALL opcode[15] to provide multi-operation support in our wallet. DELEGATECALL allows the caller contract $A$ to execute code that lives at a different address $B$ while still running in the context of $A$, retaining $A$'s value of `msg.sender`, `msg.value`, `this`, `this.balance` and storage.

***6.1.2 Counterfactual Commitments.*** Using delegatecall-based commitment, a complex commitment could be made by commiting to perform a DELEGATECALL to an on-chain address containing the "actual" commitment. To avoid having the "actual" commitment be on-chain, we can make the commitment be a commitment to code which is part of a counterfactual object, that is, commit to delegatecalling an on-chain piece of code which performs a lookup and then delegatecalls to that. In our design, this on-chain piece of code is placed in the registry.

***6.1.3 Authentication in the Registry.*** We provide two ways for the registry to authenticate requests to deploy objects. The first way is to check the `msg.sender`. The second is to explicitly provide a list of signatures; this is the technique used for metachannels.

***6.1.4 Multisig-Owner.*** In an $n$-party state channel (without metachannels) each counterfactual object is owned by $n$ parties. One way to achieve this is to have the counterfactual object

[15] The opcode is described in the Ethereum Yellowpaper[28]; in Solidity, a thin wrapper address#delegatecall is provided[4] to delegatecall specific functions in contracts implementing the Solidity ABI.

have an `update` function that must be provided with $n$ signatures. We choose to have the counterfactual object owned by the multisig "on behalf of" the $n$ parties; a commitment to update the counterfactual state of an object $O$ is a commitment to the multisig to issue a CALL with some fixed (at time of commitment signing) parameters to the contract at $O$'s counterfactual address. This has two benefits; first, the group signature verification (which can be quite costly, for example $n$ ECRECOVERs) only needs to be done once instead of twice for a single update. Secondly, updates can be batched in a merkle tree structure (and just the merkle root signed), amortizing the cost of a single group signature verification over multiple updates and deployments.

***6.1.5 Root nonce.*** For our construction in this section, we make the design decision for all objects to depend directly on the root nonce. As discussed previously, this is not necessary for all state channels, and in some cases may be suboptimal.

### 6.2 Onchain Components

***6.2.1 A multisignature wallet.*** We require this multisignature wallet to require consent from all state channel participants (i.e., unanimous consent), such as via an $n$-of-$n$ threshold[16]. This is the object that will be used to control and hold all state deposit, and the sole component in our design which must be instantiated on a secure blockchain layer for any *new* state channel. Since multisig wallets are also useful outside of the context of state channels (e.g., a foundation holding funds in a multisig wallet), we describe some features a multisig wallet must support in order to be useable as a state-deposit-holder for a state channel.

- Requires unanimous consent.
- Is capable of atomically executing complex operations.
- Consent, once given, must not be revocable by just one or a few participants in the channel.
- Commitment must not require or enforce a particular ordering to remain valid.
- Commitments must be secure against replay considerations.
- Commitments must be submittable by entities other than the consenting party themselves.

To accomplish these requirements, our multisig has a public executeTransaction function that can execute arbitrary transactions in the context of the multisig (provided they are signed by all owners) using DELEGATECALL-based commitment, that is, executeTransaction can perform an arbitrary single DELEGATECALL. We also allow executeTransaction to perform a single arbitrary CALL to avoid having to wrap a CALL in a DELEGATECALL. This enables any party to submit a signed transaction by all consenting parties to execute an arbitrary function "on behalf of" the multisig.

***6.2.2 The registry.*** The registry maps counterfactual addresses to the addresses of deployed on-chain contracts. It is a stateful contract and contains in storage a mapping from counterfactual addresses to deployed addresses; we use `bytes32` as the space of counterfactual objects, hence the mapping will have Solidity type `mapping(bytes32 => address)`. The registry provides a total of five functions; we specify four of them here and defer a description of the last one to section 6.7.1. We can divide these functions into two groups; the first two functions provide the bare minimum needed for supporting counterfactual instantiation, and the later two are utility functions.

[16] There are various alternatives that let us use other thresholds while still having $n$-of-$n$ semantics such as including provably invalid keys or assigning multiple keys per party. Inclusion of multiple keys from a single party can be an important technique in implementing hot/cold wallets or other security policies.

- `deploy`: A contract is deployed with the given code and constructor arguments, the counterfactual address is computed, and the counterfactual address `=>` deployed address mapping is written to storage.
- `resolve`: The given counterfactual address is looked up in storage, and the corresponding address is returned.

In our registry, C.code and C.args are combined on the client side by concatenating them; the Solidity ABI specifies that the compiled code of a contract shall execute the constructor, looking for the arguments to the constructor after the compiled code, when broadcast as the argument of a CREATE opcode call. Hence, our deploy function simply accepts a blob of bytecode to pass to CREATE. Hence `deploy` has the signature `(bytes32) => ()`.

The other two functions do not necessarily need to be placed in the registry, but find their logical home here. They are used to support delegatecall-based commitments and counterfactual commitments. They are simple proxy functions; just as one can issue a CALL to an ethereum address, one uses `proxycall` to issue a call to a counterfactual address (that is, to a counterfactually instantiated contract with the given counterfactual address, which has been deployed).

- `proxycall`: The provided counterfactual address is looked up, and a CALL with the provided calldata is made to the resulting address
- `proxydelegatecall`: The provided counterfactual address is looked up, and a DELEGATECALL with the provided calldata is made to the resulting address

For an example implementation of the ***registry*** contract in Solidity, see appendix B.

### 6.3 Counterfactual Objects

To formally define counterfactual objects, we need to clarify an ambiguity in terminology that is inherited from on-chain smart contracts. Using Solidity, the word "smart contract" can refer to

1. The contents of a `contract { ... }` block, which declares the layout of storage as a set of variables, a collection of functions, and a constructor, or
2. The previous concept, but with constructor arguments specified, or
3. An account with code and storage (in contrast to an externally owned account), created as a result of broadcasting a transaction with a CREATE opcode; also sometimes called "an instance of a smart contract"

In our terminology, "counterfactual objects" refer to the third meaning of "smart contract": accounts with code and storage. On-chain, different smart contracts that share code are distinguished by having different addresses; for counterfactual objects, we require them to have an id attribute. This prevents replay attacks across different counterfactual objects and also allows instantiation of multiple counterfactual objects using different ids. Hence a counterfactual object is abstractly defined as an attribute tuple of the form:

$$\mathsf{C} = (\mathsf{C.id}, \mathsf{C.code}, \mathsf{C.state}, \mathsf{C.args})$$

where C.code is the source code of the contract, C.id is the unique identifier, and C.state is the counterfactual state. We use the convention that C.code includes the constructor code; hence broadcasting a transaction containing C.code as well as C.args deploys a counterfactually instantiated contract on-chain.

***6.3.1 The API.*** Since the code of a counterfactual object, C.code, is an arbitrary sequence of bytes, there are many possible ways for them to be implemented. The counterfactual instantiation process takes this bytecode as well as some constructor arguments and uses that to define the initial state of the object, C.state, and the signed copy of both to determine the counterfactual address. In practice, we set a few expectations for every counterfactual object:

1. It stores a unique identifier which is passed as a constructor argument.
2. It is instantiated by a multisignature wallet. We expect the constructor written in C.code to contain logic for assigning the *owner* variable in storage to the *sender* unless otherwise specified.
3. It implements a withdraw function that contains the exact code to be executed by the ***multisig*** using the DELEGATECALL opcode to send the state locked inside the state deposit in the case of an onchain deployment of the counterfactual object.
4. It implements a timeout length to be defined inside C.code.
5. It is parameterized by a mapping of counterfactual addresses to nonce numbers which constitutes the dependencies of the counterfactual object. We require the following condition for finalization: for every key-value pair $k : v$ in the mapping, the counterfactual object with counterfactual address $k$ must be initialized, have nonce value exactly equal to $v$, and (recursively) be finalizable.
6. It is either finalized or not, and once finalized, the state can no longer change. It implements an isFinalized function which can detect if the contract has been put in a finalized state.
7. We expect the withdraw function to assert that the contract is in a finalized state before making any state updates.

### 6.4 Base Protocols

We assume the existence of a well-known address `registry`, a pure function keccak256 which implements the keccak256 collision-resistant hash function, a pure function ecsign(privkey, digest) that takes an ECDSA private key and a digest and returns a signature (the digest signed by the private key), and a pure function ABIEncode that takes a function selector and arguments and encodes them into the right calldata using the Solidity contract ABI.

***6.4.1 On-chain transactions.*** We assume that participants are able to make on-chain transactions such as deploying a contract and transferring ether. For simplicity, we treat these on-chain actions as though they immediately finalize (i.e., we do not consider the risk of them reverting); in practice, participants have to wait for sufficient confirmations after making such transactions before carrying out other actions that depend on the finality of those transactions.

***6.4.2 SignCommitment.*** The SignCommitment protocol commits a multisig to perform a certain action. Actions are classified into three types - send ether, call and delegatecall. In the case of call, the calldata is specified.

```
SignCommitment(multisig, commitment)

for owner in multisig.owners:
    digest := keccak256(commitment, multisig.id)
    commitment := ecsign(owner.privkey, digest)

    for recipient in multisig.owners:
        send commitment to recipient
```

where `commitment` is one of:

1. `(CALL, toAddr, calldata, value)`
2. `(DELEGATECALL, toAddr, calldata)`

We assume that our multisig wallet authenticates submitted commitments (by calling ECRECOVER on the properly constructed digest) and if so, executes them. At dispute time, any party can unilaterally broadcast a properly formatted transaction containing the array of signatures, causing the multisig to execute the transaction committed to.

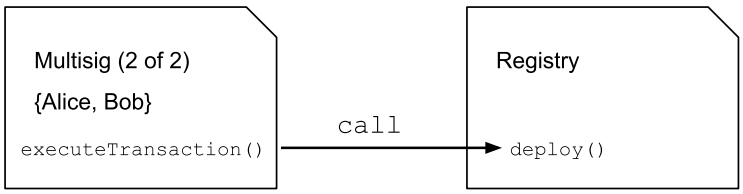
Furthermore, we make the simplifying assumption that only one instance of `signCommitment` is running at any time, and that before the protocol is initiated, all parties agree that they are executing `signCommitment` and on the parameters with which they are executing it. Note that the main loop is executed independently by every owner. Hence, this protocol suffers from the non-fair exchange of signatures problem discussed in section 3.4.1. In a state channel of size $n$, a participant in this protocol will send $n-1$ commitments to other participants and waits to receive $n-1$ commitments before considering the protocol successfully executed; however, a participant may send out his commitment in his execution and not receive the commitments he should receive. The higher-level calling protocol must then handle this failure; for instance see appendix C in the case of initializing a state channel.

***6.4.3 Instantiate.*** This protocol counterfactually instantiates a counterfactual object using sender-based message authentication in the registry.

```
Instantiate(multisig, C)

calldata := ABIEncode("deploy(bytes)", [C.dbytecode])
cfaddress := keccak256(C.dbytecode, [multisig.address])
commitment := (CALL, registry, calldata, 0)
SignCommitment(multisig, commitment)
return cfaddress
```

At dispute time, the multisig performs a call to registry.deploy, deploying the contract and adding the entry to the mapping.

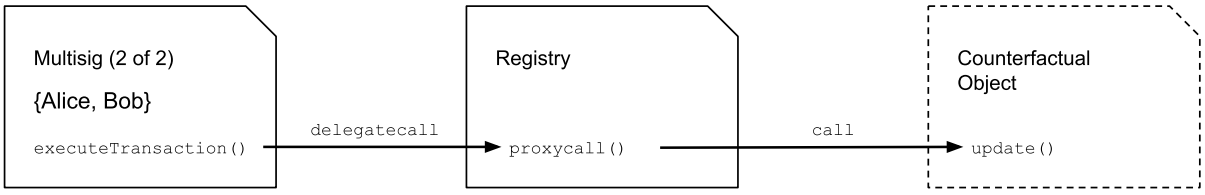



***6.4.4 Update.*** This protocol updates the counterfactual state of a counterfactual object. We assume that partipants store in memory the source code of every counterfactual object whose instantiation they were part of; that is, given a counterfactual address $c$, there exists a counterfactual object $C$ such that $c =$ `keccak256(C.dbytecode, multisig.address)`. Let the mapping from such $c$ to $C$ be called `lookup`, a partial inverse of `keccak256`.

```
Update(multisig, cfAddr, newstate)

C := lookup(cfAddr)
calldata1 := ABIEncode(C.fnSignatures.update, C.id + newstate)
calldata2 := ABIEncode("proxycall(bytes)", calldata1)
commitment := (DELEGATECALL, registry, calldata2)
SignCommitment(multisig, commitment)
```

At dispute time, the multisig performs a delegatecall to registry.proxycall, which calls update with msg.sender set to the multisig.




***6.4.5 CommitWithdrawal.*** This protocol commits the multisig to delegatecall withdraw, assigning parts of a state deposit to a counterfactual object.

```
CommitWithdrawal(multisig, cfAddr)

calldata1 := ABIEncode("withdraw(address, bytes32)", [registry, cfAddr])
calldata2 := ABIEncode("proxydelegatecall(address, bytes32, bytes)", [
    registry, cfAddr, calldata1])
SignCommitment(multisig, (DELEGATECALL, registry, calldata2))
```

At dispute time, the multisig performs a delegatecall to `registry.proxydelegatecall`, which delegatecalls `co.withdraw`, where `co` represents the instantiated counterfactual object. At this point, the transaction is executing code from `co` but with storage set to the multisig, allowing

for e.g., the `co.withdraw` function to transfer ether from the multisig. In order for the code in `co.withdraw` to examine storage from `co`, a CALL has to be made to `co`.


### 6.5 Counterfactual Object Primitives

In this section, we specify some common counterfactual objects that we use in future examples.

***6.5.1 A root nonce.*** Written as $\mathsf{C}_{\mathit{RootNonce}}$, this object has no state outside of its nonce. Its purpose is to be a dependancy for other counterfactual objects that, when its nonce is incremented, can effectively "delete" any counterfactual objects that depend on it.

$$\mathsf{C.state} = (\mathsf{C.nonce})$$

***6.5.2 A payment contract.*** Written as $\mathsf{C}_{\mathit{PaymentChannel}}$, this object simply stores a mapping of parties in the state channel to balances in addition to its nonce. It is simply used as a primitive state management object, from which most other counterfactual objects will draw state from.

$$\mathsf{C.state} = (\mathsf{C.nonce}, \mathsf{C.balances})$$

***6.5.3 A balance refund contract.*** This is written as $\mathsf{C}_{\mathit{BalanceRefund}}$ and parameterized by a threshold $t$ and recipient $p$. The semantics of this contract is: any balance in the multisig above $t$ is assigned to (and withdrawable by) $p$. The name "BalanceRefund" comes from the fact that when a participant puts ether into a state channel, this object is used to assign him ownership of the deposited ether. In the event that a participant deposits ether and then the counterparty goes offline, in the absence of a BalanceRefund object, the deposited ether could be held hostage. The BalanceRefund object allows the depositor to, in that case, retrieve the deposited ether. It turns out that the exact same object can be used for instant withdrawals as well.

$$\mathsf{C.state} = ()$$

All objects also store $\mathsf{C.dependencies}$ which is the mapping of counterfactual addresses to nonce version number requirements.

### 6.6 Protocols

***6.6.1 Creating a new state channel.*** This is the handshake for creating a new state channel *safely* between some number of parties. The goal is for each party to have guarantees that their state cannot be stolen from them when depositing into the multisig as well as to minimize the need for any onchain transactions at all until the entire counterfactual structure has been set up.

1. **Exchange public addresses over a secure communications layer.** Enumerate the set of keys which will be used by each party in $\mathcal{P}$; these keys that will be listed in the ***multisig*** as the owners. All parties must agree on a unique identifier for the to-be-deployed multisig, to prevent replay attacks across different multisigs. In these examples we will use $\lambda_{\mathcal{P}}.\mathsf{id}$ as that unique identifier.

2. **Select a participant to deploy the multisig wallet.**

$$v := randint(0, n)$$

3. **Deploy a unique multisignature wallet.**

$$\textbf{\textit{multisig}} := \textbf{Deploy}(P_v.privkey, \mathit{Multisig}(\mathcal{P}))$$

From here on, we use the convention that the protocols **Instantiate**, **CommitWithdrawal** and **Update** may be written with their first argument ommitted, in which case we set that argument to ***multisig***.

4. **Set up the counterfactual structure to allow the first state deposit to be made.**

(a) Counterfactually instantiate a root nonce contract.

$$r := \textbf{Instantiate}(\mathit{RootNonce})$$

(b) Counterfactually instantiate a balance refund contract.

$$b_0 := \textbf{Instantiate}\left(\mathit{BalanceRefund}\begin{pmatrix}\text{threshold}=0\\ \text{recipient}=P_0\\ \text{dependencies}=\{r{:}0\}\end{pmatrix}\right)$$

(c) Assign state to the counterfactual balance refund contract.

$$\textbf{CommitWithdrawal}(b_0)$$

Note that at this point in the protocol, nothing has been deployed except an empty multisig wallet; `commitWithdrawal` is called on an undeployed counterfactually instantiated object.

5. Make the first state deposit. $P_0$ makes a deposit of $s_0$ into the multisignature wallet.

6. Set up counterfactual structures to allow the next state deposit to be made.

Repeat the process from step (b) for the remaining parties $P_1, \ldots, P_{n-1}$. The process is identical to (b) in structure, but this time since the root nonce has already been counterfactually instantiated, it is only necessary to update it. For the $k$-th step ($k = 1, \ldots, n-1$) follow the following process to allow $P_k$ to safely deposit $s_k$:

(a) Compute the amount of money deposited so far and each party's balance.

$$S_k := \sum_{i=0}^{k-1} s_i$$
$$B_k := \{P_i : s_i \mid 0 \leq i < k\}$$

(b) Counterfactually instantiate a new balance refund contract.

$$b_k := \textbf{Instantiate}\left(\mathit{BalanceRefund}\begin{pmatrix}\text{threshold}=S_k\\ \text{recipient}=P_k\\ \text{dependencies}=\{r{:}k\}\end{pmatrix}\right)$$

(c) Assign the state deposit to this object.

$$\textbf{CommitWithdrawal}(b_k)$$

(d) Counterfactually instantiate a new payment contract to store the already-deposited amount.

$$p_k := \textbf{Instantiate}\left(\mathit{PaymentChannel}\left(\begin{smallmatrix}\mathsf{balances}=B_k\\ \mathsf{dependencies}=\{r{:}k\}\end{smallmatrix}\right)\right)$$

(e) Assign the state deposit to this object.

$$\textbf{CommitWithdrawal}(p_k)$$

(f) Counterfactually increment the root nonce to version $k$.

$$\textbf{Update}(r, k)$$

7. **Make the $k$-th state deposit.** $P_k$ makes a deposit of $s_k$ into the multisignature wallet.
8. **Repeat steps 6 and 7 until all participants have made their state deposit.**
9. **Instantiate a regular payment object and assign state to it.** Now that there is a balance of $\sum_{i=0}^{n-1} s_i$ in the multisignature wallet state deposit and all parties have finished depositing, we can now delete the last balance refund object $b_{n-1}$ and create a basic payments object.

(a) Counterfactually instantiate a payment object.

$$B := \{P_i : s_i \mid 0 \leq i < n\}$$
$$p := \textbf{Instantiate}\left(\mathit{PaymentChannel}\left(\begin{smallmatrix}\mathsf{balances}=B\\ \mathsf{dependencies}=\{r{:}n\}\end{smallmatrix}\right)\right)$$

(b) Assign the state deposit to this object.

$$\textbf{CommitWithdrawal}(p)$$

(c) Counterfactually increment the root nonce to version $n$.

$$\textbf{Update}(r, n)$$

We defer a specification of the rollback protocol and a sketch of the safety proof to appendix C. Intuitively, this protocol is safe because at any point in time someone who has deposited has ownership of the deposited amount assigned to himself either via a PaymentChannel or a BalanceRefund object. We atomically tranisiton between states for which this invariant holds either by updating the root nonce or by depositing money on-chain.

***6.6.2 Making a payment in the payment channel.*** After all steps in the prior section have been completed, parties in the state channel can begin making payments simply by updating the balance of the payments counterfactual object, which only requires one message to be signed. So, for example, if the balances change from $P_i$ having $s_i$ in the payment channel to $s'_i$ then the parties would *counterfactually update the payment contract* to a new value for its balances using the following arguments:

$$\textbf{Update}(p, \{P_i : \ s'_i, \ \mid 0 \leq i < n\})$$

***6.6.3 Installing a new application in the state channel.*** Suppose the parties of the state channel decide to play poker, and each participant decides to put $p_i$ into the pot. We will define $\mathsf{C}_{\mathsf{poker}}$ as the contract that defines the rules of poker. Let the root nonce be at sequence number $k$.

1. Counterfactually instantiate the poker object.

$$o := \textbf{Instantiate}\left(\mathit{Poker}\left(\begin{smallmatrix}\mathsf{pot}=\{P_i:p_i \mid 0\leq i<n\},\\ \mathsf{dependencies}=\{r:k+1\}\end{smallmatrix}\right)\right)$$

2. Assign part of the state deposit to the object.

$$\textbf{CommitWithdrawal}(o)$$

3. Create a new payment object with reduced $s$.

$$p := \textbf{Instantiate}\left(\mathit{PaymentChannel}\left(\begin{smallmatrix}\mathsf{balances}=\{P_i:\ s_i-p_i\}_i\\ \mathsf{dependencies}=\{r:\ k+1\}\end{smallmatrix}\right)\right)$$

4. Assign part of the state deposit to the object.

$$\textbf{CommitWithdrawal}(p)$$

5. Counterfactually increment the root nonce to $k+1$.

$$\textbf{Update}(r, k+1)$$

From here, playing the game is exactly the same as making a payment in the payment object with the only difference being the different arguments the two objects take for updates.

***6.6.4 Adding to the state deposit.*** It is often the case that some participant in a state channel might want to deposit more state into the state channel. For example, $P_0$ might want to deposit $w$ into the multisig. Let the root nonce be at sequence number $q$ and the starting balances in the payment contract be $\{P_i :\ s_i \mid 0 \leq i < n\}$. Let $S = \sum_i s_i$.

1. **Set up the counterfactual structure to allow the state deposit to be made.**
   (a) Counterfactually instantiate the balance refund contract.

$$b := \textbf{Instantiate}\left(\mathit{BalanceRefund}\left(\begin{smallmatrix}\mathsf{threshold}=S\\ \mathsf{recipient}=P_0\\ \mathsf{dependencies}=\{r:q\}\end{smallmatrix}\right)\right)$$

   (b) Assign part of the state deposit to the object.

$$\textbf{CommitWithdrawal}(b)$$

2. **Make the additional state deposit.** $P_0$ makes an additional state deposit of $s_0'$ into the multisignature wallet.
3. **Update the counterfactual state to "delete" the balance refund contract.**
   (a) Counterfactually instantiate a new payment contract.

$$p := \textbf{Instantiate}\left(\mathit{PaymentChannel}\left(\begin{smallmatrix}\mathsf{balances}=\{P_i:s_i \mid 0\leq i<n\}\\ \mathsf{dependencies}=\{r:q+1\})\end{smallmatrix}\right)\right)$$

   (b) Assign part of the state deposit to the object.

$$\textbf{CommitWithdrawal}(p)$$

   (c) Counterfactually increment the root nonce to $q+1$.

$$\textbf{Update}(r, q+1)$$

***6.6.5 Making an instant withdrawal.*** It is also common that some party might want to withdraw part of their state out of the state channel. For example, $P_0$ might want to withdraw $w$ from the state deposit. Let the root nonce be at sequence number $q$ and the starting balances in the payment contract be $\{P_i : s_i \mid 0 \leq i < n\}$. Let $S = \sum_i s_i$.

1. **Set up the counterfactual structure to allow the instant withdrawal to be made.**

   (a) Counterfactually instantiate a balance refund contract.

   $$b := \textbf{Instantiate}\left(\mathit{BalanceRefund}\begin{pmatrix}\text{threshold}=S-w\\ \text{recipient}=P_0\\ \text{dependencies}=\{r{:}q{+}1\}\end{pmatrix}\right)$$

   (b) Assign part of the state deposit to the object.

   $$\textbf{CommitWithdrawal}(b)$$

   (c) Counterfactually instantiate the new payment contract with a reduced balance.

   $$p := \textbf{Instantiate}\left(\mathit{PaymentChannel}\begin{pmatrix}\text{balances}=\{P_0{:}s_0-w\}\cup\{P_i{:}s_i \mid 1\leq i<n\}\\ \text{dependencies}=\{r{:}q{+}1\}\end{pmatrix}\right)$$

   (d) Assign part of the state deposit to the object.

   $$\textbf{CommitWithdrawal}(p)$$

   (e) Counterfactually increment the root nonce to $q + 1$.

   $$\textbf{Update}(r, q + 1)$$

2. **Sign an instant withdrawal transaction.** All parties in $\mathcal{P}$ sign a withdrawal transaction to the multisig. This transaction is executed on-chain.

   $$\textbf{SignCommitment}(\textbf{\textit{multisig}}, (\textsc{transfer}, P_0, w))$$

### 6.7 Metachannels

To give the protocol for creating a metachannel, we have to introduce two new base protocols and one new counterfactual object primitive. The new protocols are modifications of **Instantiate** and **Update** with explicit owner sets, since metachannel participants do not collectively own a multisig. An alternative design choice, which we do not pursue, is to use a counterfactually instantiated multisignature wallet.

***6.7.1 Instantiate With Explicit Owners.*** This protocol counterfactually instantiates a counterfactual object using explicit ecrecover-based message authentication in the registry.

```
InstantiateWithExplicitOwners(owners, C)

cfaddress := keccak256(C.dbytecode, owners)

commitments := {}
for owner in owners:
    digest := keccak256(C.id, C.dbytecode)
    commitments[owner] = ecsign(owner.privkey, digest)
for owner in owners:
    send commitments to party

return cfaddress
```

At dispute time, any member of `owners` can unilaterally broadcast an appropriately formatted transaction containing `commitments`, causing the registry to deploy the contract at `cfaddress`.

***6.7.2 Update With Explicit Owners.*** We assume that counterfactual objects with explicit owners have an `update` function which takes in a new state and signatures, verifies that the signatures sign a digest of the new state, and performs the update.

```
Update(multisig, cfAddr, newstate)

C := lookup(cfAddr)

commitments := {}
for owner in owners:
    digest := keccak256(C.id, newstate)
    commitments[owner] = ecsign(owner.privkey, digest)
for owner in owners:
    send commitments to party
```

***6.7.3 Proxy Payment Contract.*** We introduce a new primitive for metachannels called a proxy payment contract. Written as $\mathsf{C}_{\mathit{ProxyContract}}$, the purpose of this object is to check the balances of an external counterfactual payment object and pay back the parties in the proxy payment contract based on the balances of the external object. It stores four critical variables: the balance of the entire proxy payment contract, the address of the "sender" that is using it as a proxy, the address of the "intermediary" that is acting as the proxy through which the sender uses to send state, and the counterfactual address of the payment contract it depends on.

$$\mathsf{C.state} = (\mathsf{C.nonce}, \mathsf{C.amount}, \mathsf{C.sender}, \mathsf{C.intermediary}, \mathsf{C.pcaddress})$$

This contract is important because it "locks up" state in a state channel as collateral while it is being used in a metachannel. The rules of its withdraw function ensure that the sum of all balances in the external payment contract are exactly equal to $\mathsf{C}_{\mathit{ProxyContract}}.\mathsf{amount}$. Additionally, the function ensures that the intermediary is rewarded with all of the state that is not assigned to the sender in the payment contract.

A metachannel between Alice and Bob with Ingrid as the intermediary is essentially comprised of two proxy payment contracts. One is in the {Alice, Ingrid} state channel with Alice as the sender and the other in the {Ingrid, Bob} state channel where Bob is the sender. In both contracts, Ingrid is the intermediary and the amount is the same. Each respective withdraw function checks the balance of the payment contract it relies on, sends the sender the amount they have in that contract and the intermediary the rest.

An important note for metachannels is that the counterfactual payment contract in this situation has the unique property in our state channel construction that its owner is not any onchain deployed ***multisig***; instead, the owners set and signature verification functions are a part of the counterfactual object itself. We abuse notation slightly and use the convention that a *PaymentChannel* call with an `owners` argument produces such an **explicitly owned payment object** and a call without produces a normal payment object. An explicitly owned payment channel has an additional variable in its state, *PaymentChannel*.owners. We allow the owner set to be modified.

***6.7.4 Protocol for creating a metachannel.*** We assume the Alice-Ingrid state channel has a multisig $m_{av}$, a root nonce with counterfactual address $r_{av}$ with nonce $k$, and a payment channel at address $p_{av}$ with balance $\{\mathsf{Alice} : A, \mathsf{Ingrid} : v_a\}$. We assume that the Ingrid-Bob state channel has a multisig $m_{vb}$, a root nonce with counterfactual address $r_{vb}$ with nonce $l$, and a payment channel at address $p_{vb}$ with balance $\{\mathsf{Ingrid} : v_b, \mathsf{Bob} : B\}$. We set up a metachannel payment channel with balance $\{\mathsf{Alice} : a, \mathsf{Bob} : b\}$ with the constraints $A \geq a, v_b \geq a, B \geq b, v_a \geq b$ (otherwise, trustless transitive payment is not possible).

1. Counterfactually instantiate a payment contract between Alice and Bob.

$$p_{ab} := \textbf{InstantiateWithExplicitOwners}\left(\{a, b, v\}, \textit{PaymentChannel}\left(\begin{smallmatrix}\text{balances=\{Alice:}a\text{,Bob:}b\text{\}}\\ \text{owners=\{Alice,Ingrid,Bob\}}\end{smallmatrix}\right)\right)$$

2. Counterfactually instantiate a proxy contract between Alice and Ingrid.

$$x_{av} := \textbf{Instantiate}(m_{av}, \textit{ProxyContract}\begin{pmatrix}\text{amount=}a+b\\ \text{sender=Alice}\\ \text{intermediary=Ingrid}\\ \text{pcaddr=}p_{ab}\\ \text{dependencies=}\{r_{av}\text{: }k+1\}\end{pmatrix})$$

3. Commit.

$$\textbf{CommitWithdrawal}(m_{av}, x_{av})$$

4. Counterfactually instantiate a new payment contract with reduced balance.

$$p'_{av} := \textbf{Instantiate}\left(m_{av}, \textit{PaymentChannel}\left(\begin{smallmatrix}\text{balances=\{Alice:}A-a\text{,Ingrid:}v_a-b\text{\}}\\ \text{dependencies=}\{r_{av}\text{: }k+1\}\end{smallmatrix}\right)\right)$$

5. Commit.

$$\textbf{CommitWithdrawal}(m_{av}, p'_{av})$$

6. Increment the Alice-Ingrid root nonce.

$$\textbf{Update}(m_{av}, k+1)$$

7. Counterfactually instantiate a proxy contract between Ingrid and Bob.

$$x_{bv} := \textbf{Instantiate}(m_{bv}, \textit{ProxyContract}\begin{pmatrix}\text{amount=}a+b\\ \text{sender=Bob}\\ \text{intermediary=Ingrid}\\ \text{pcaddr=}p_{ab}\\ \text{dependencies=}\{r_{vb}\text{: }l+1\}\end{pmatrix})$$

8. Commit.

$$\textbf{CommitWithdrawal}(m_{bv}, x_{bv})$$

9. Counterfactually instantiate a new payment contract with reduced balance.

$$p'_{vb} := \textbf{Instantiate}\left(m_{vb}, \textit{PaymentChannel}\left(\begin{smallmatrix}\text{balances=\{Ingrid:}v_b-a\text{,Bob:}B-b\text{\}}\\ \text{dependencies=}\{r_{vb}\text{: }l+1\}\end{smallmatrix}\right)\right)$$

10. Commit.

$$\textbf{CommitWithdrawal}(m_{bv}, p'_{vb})$$

11. Increment the Ingrid-Bob root nonce.

$$\textbf{Update}(m_{vb}, l+1)$$

12. Counterfactually update the payment contract to remove Ingrid's involvement.

$$\textbf{Update}(p_{ab}, \mathsf{owners} = \{\mathsf{Alice}, \mathsf{Bob}\})$$

After this, the metachannel payment channel can be used by simply updating the state of $p_{ab}$, which can be done without Ingrid's consent.

## 7 Future Work

### 7.1 Capabilities of generalized state channels

State channels have limitations not present in other scalability solutions like base-layer sharding or Plasma. The most significant of these is that dapps in channels must have a defined participant set. For instance, one cannot send a channelized payment to someone not already in a payment channel network, while it is possible to do this in Plasma. The set of participants in a payment channel network, although large, can be enumerated, and in all existing designs, an on-chain transaction must be made to enlarge that set.

This prevents channels from being usable for some applications. For instance, a SHA3 bounty dapp (a smart contract that trustlessly pays a bounty to someone who can provide a SHA3 collision) depends for its security on "the public" being able to participate. The logic is as follows: if a practical attack on SHA3 existed, "anyone" who knew about it has an incentive (equal to the bounty amount) to reveal this by collecting the bounty; hence, the fact that the bounty is uncollected provides a certain level of guarantee that a practical attack on SHA3 does not in fact exist. If we place this in a channel we run into two problems. First, presumably the person who wishes to collect the bounty (e.g., a disgruntled NSA employee) is not already in a channel, and hence has to join it before collecting it, and we are really providing no benefits over putting the bounty on-chain. Secondly, members of the public not already in the channel cannot actually know that the bounty is "collectible" without joining the channel. Thirdly, even if someone does manage to join the channel and verify that the bounty is collectible, they cannot convince other people of this fact; it could be that the bounty is only offered to the one person trying to verify that it is publicly available.

Another limitation of state channels is that the time granularity cannot be decreased to below that of the root chain. For instance, if ethereum block times are 5 seconds and we very optimistically assume that blocks are somehow finalized right after creation, one cannot enter a contract within a channel such that the contract is only valid for 1 second (e.g., a limit order with 1 second time in force).

### 7.2 Supporting services

We ancitipate an ecosystem of third-party services that will be used by state channel users. For instance, in the context of payment channel networks, a hub is a large pool of capital that can open many small channels with end users, allowing users to route payments through the hub.

The Lightning Network has introduced the term "watchtowers" for third parties who hold a copy of an end user's latest state (and signed messages) to respond on the end user's behalf. This allows end users to go offline for extended periods of times at the expense of weakening the security assumption to that of "at least one watchtower will respond in the event of a dispute". End users

can contract trustlessly with watchtowers to pay them a fee, or pay a fee in the event of a successful dispute, or punish them in the event of not helping to resolve a dispute that they could have.

In the context of state channels, another third party is an insurance provider. These will operate much like traditional insurers in that they collect an upfront premium, then in the event of griefing (defined in this case as on-chain transactions happening which could have happened off-chain), distribute a payout. Since griefing is impossible to eliminate, the insurer is not a trusted third party who has the power to claim that one party is griefing and thereby impose penalties; all it can do is gain private knowledge that one party is griefing. The insurer would collect Alice's signed messages, check that they are valid, and then show them to Bob to attempt to collect a signature. If Bob does not provide a signature, then the insurer raises premiums or refuses to contract with Bob in the future.

### 7.3 Additional techniques and subchannel types

There are other interesting techniques and channel types.

***7.3.1 Off-Chain Time*** See the first author's talk [13].

***7.3.2 Advanced Key Management*** Similarly to how existing institutions like Coinbase benefit from using advanced key management (e.g., hot/cold wallets, spending limits), channel users can benefit from using advanced key management.

***7.3.3 High throughput payments*** When we increase the throughput requirements placed on a payment channel far enough, even the low costs of an in-channel payment might become too high. For instance, the cost of verifying an ECDSA signature (not on-chain, but privately, by a channel participant) can become too high. Techniques revolving around hash revelation can mitigate this.

### Acknowledgments

We thank Vitalik Buterin, Tom Close, Erik Bryn, Josh Stark, Nima Vaziri, Armani Ferrante, Lisa Eckey, Kristina Hostáková, Yoichi Hirai, Sylvain Laurent, and Alex Xiong for their discussion and feedback.

---

[1] Counterfactual conditional — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Counterfactual_conditional, 2018.

[2] A note on data availability and erasure coding. https://github.com/ethereum/research/wiki/A-note-on-data-availability-and-erasure-coding, 2018.

[3] Raiden specification. http://raiden-network.readthedocs.io/en/stable/spec.html, 2018.

[4] Solidity documentation - units and globally available variables. https://solidity.readthedocs.io/en/develop/units-and-global-variables.html, 2018.

[5] Aaron van Wirdum. The history of lightning: from brainstorm to beta. https://bitcoinmagazine.com/articles/history-lightning-brainstorm-beta/, 2018.

[6] Andrew Miller et. al. Sprites and state channels: Payment networks that go faster than lightning. https://arxiv.org/abs/1702.05812, 2017.

[7] Antoine Le Calvez. When the bitcoin dust settles. https://medium.com/@alcio/when-the-bitcoin-dust-settles-878f3431a71a, 2018.

[8] Augusto Hack. Add cooperative channel closing. https://github.com/raiden-network/raiden/issues/217, 2016.

[9] Vitalik Buterin. A next generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2013.

[10] Adam Back et al. Enabling blockchain innovations with pegged sidechains. https://blockstream.com/sidechains.pdf, 2014.
[11] Rusty Russell et. al. Lightning network in-progress specifications. https://github.com/lightningnetwork/lightning-rfc, 2018.
[12] Jeff Coleman. State channels. http://www.jeffcoleman.ca/state-channels/, 2015.
[13] Jeff Coleman. Universal hash time. https://www.youtube.com/watch?v=phXohYF0xGo, 2015.
[14] Jeremy Longley and Oliver Hopton. Funfair technology roadmap and discussion. https://funfair.io/wp-content/uploads/FunFair-Technical-White-Paper.pdf, 2017.
[15] Jorge Izquierdo and Manuel Araoz. EIP 897: ERC delegateproxy. https://eips.ethereum.org/EIPS/eip-897, 2018.
[16] Martin Köppelmann. How offchain trading will work. https://forum.gnosis.pm/t/how-offchain-trading-will-work/63, 2015.
[17] Michael Lewis-Beck, Alan E Bryman, Tim Futing Liao. Counterfactual. the sage encyclopedia of social science research methods.
[18] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.
[19] Oleskii Matiiasevych. How is the address of an ethereum contract computed? https://ethereum.stackexchange.com/questions/760/how-is-the-address-of-an-ethereum-contract-computed, 2016.
[20] Henning Pagnia and Felix C. Gärtner. On the impossibility of fair exchange without a trusted third party. https://www.cs.utexas.edu/~shmat/courses/cs395t_fall04/pagnia.pdf, 1999.
[21] Paul Grau. Lessons learned from making a chess game for ethereum. https://medium.com/@graycoding/lessons-learned-from-making-a-chess-game-for-ethereum-6917c01178b6, 2016.
[22] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. https://plasma.io/plasma.pdf, 2017.
[23] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. https://lightning.network/lightning-network-paper.pdf, 2016.
[24] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Daniel Malinowski. Perun: Virtual payment hubs over cryptographic currencies. https://eprint.iacr.org/2017/635, 2017.
[25] Stefan Dziembowski, Sebastian Faust, Kristina Hostakova. Foundations of state channel networks. https://eprint.iacr.org/2018/320, 2018.
[26] Vitalik Buterin. The triangle of harm. https://vitalik.ca/general/2017/07/16/triangle_of_harm.html, 2017.
[27] Vitalik Buterin. Account abstraction for main chain. https://github.com/ethereum/EIPs/issues/859, 2018.
[28] Gavin Wood. Ethereum: A secure decentralized generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf, 2018.

## A Counterfactual Terminology

In philosophy, the **counterfactual conditional** [1] is a statement of the form "if X then Y", but where X isn't actually the case, making the antecedent "counter to fact". In other words, it is a statement of the form "if X were the case, then Y would be, even though X isn't". For instance, a statement like "if Oswald had not shot Kennedy, then someone else would have" is a counterfactual statement, since the antecedent is false. A counterfactual conditional cannot be evaluated as a truth-functional conditional, since a truth-functional conditional with false antecedent is true[17]; in the Oswald-Kennedy example, it expresses a causal relation between two events, not logical entailment.

In blockchains, we use this term in a different sense, which we have briefly described in 5.1, and which we will define more fully in the rest of this appendix. We first give a general definition of counterfactual terminology, and then restrict this definition to the cases we use in state channels in 6.

### 1 Preliminary Definitions

First, we recapitulate the definitions of **ethereum state** and the **ethereum state transition function** from the white paper [9]. Ethereum state is a mapping of accounts to account state (nonce, ether balance, contract code, and storage). A transaction is a signed message sent from an externally owned account. The state transition function is a function $APPLY(S, TX) \to S'$ that takes a starting state and a transaction and outputs a new state which is the result of applying the transaction to that state.

***a Actors and Capabilities.*** We model externally-owned account owners as actors with certain limited capabilities: they can keep secrets, cannot commit to cooperating with each other, cannot invert hash functions, etc.

***b State Transition.*** Ethereum transactions can be considered state transitions by considering the function $APPLY(\cdot, TX) : T \to T \cup \perp$ where $T$ is the set of valid ethereum states and $\perp$ is the special error state. However, this is too constraining for our purposes; we wish to define for example "transactions" like "account `0x407...` transfers 5 ether to account `0x00b...`" without constraining what nonce the sending account must be at. Hence we define a state transition to be a relation on $T$, to capture this intuitive notion. This definition also includes "block number increases" as a state transition, includes state transitions which might not fit into one transaction, and includes state transitions which have a choice of transactions to fufil them (for instance, an account transfers ether to either one of two accounts).

### 2 Core Definitions

Let $P$ be an actor and $X$ be a state transition, and $S$ be a state (thought of as the current blockchain state). A necessary condition for counterfactual$(P, X)$ to hold is that $P$ is able to apply $X$ to $S$.

> **Reachability**: $P$ is capable of performing a sequence of blockchain operations that results in a state $S'$ such that $X(S, S')$ holds

It is also necessary that the reachability property cannot be revoked by parties other than $P$:

> **Non-Revocability**: No group of actors excluding $P$ is capable of performing a sequence of blockchain operations, without $P$ performing some action, that results in a state $S''$ such that $P$ is not able to apply $X$ to $S''$

If these two conditions are satisfied, we say that counterfactual$(P, X)$ holds for $S$.

***a An Example.*** Consider the smart contract in Figure a[17] and let $S$ be a state where Alice has called fund, where the constant `bobAddr = 0xb7...` is an account controlled by Bob, and where Bob does not know the preimage of y. Let X = 5 ether is transferred from the contract to `bobAddr`.

LockedEscrow.sol

```
pragma solidity ^0.4.19;

contract LockedEscrow {

    bool funded = False;
    address bobAddr = 0xb794f5ea0ba39494ce839613fffba74279579268;

    function fund() public payable {
        require(msg.value == 5 ether);
        self.funded = True;
    }

    function payout(bytes32 password) public {
        if (bytes4(keccak256(password)) == 0x3ac22516) {
            bobAddr.transfer(5 ether);
        }
    }

}
```

Since Bob is not capable of calling releaseFunds with the correct preimage, counterfactual(Bob, $X$) does not hold for $S$. However, suppose Alice communicates the preimage to Bob, resulting in a new state $S'$; then both (1) and (2) hold, hence counterfactual(Bob, $X$) is true of $S'$.

[17] We assume throughout that miners do not front-run and that Alice is willing to have her ether burned irretrievably.

***b A Non-Example.*** Consider a modification of the smart contract where we replace `ens.resolve("bob.eth")` with `msg.sender`, shown in Figure b. Also suppose that in addition to communicating the preimage to Bob, Alice also broadcasts it in a public place.

LockedBounty.sol

```
pragma solidity ^0.4.19;

contract LockedBounty {

    bool funded = False;

    function fund() public payable {
        require(msg.value == 5 ether);
        self.funded = True;
    }

    function payout(bytes32 password) public {
        if (bytes4(keccak256(password)) == 0x3ac22516) {
            msg.sender.transfer(5 ether);
        }
    }

}
```

Now we can no longer say counterfactual(Bob, $X$), because many other actors (members of the public) can call `payout`, violating non-revocability. This corresponds to the fact that applications with measures over public participation (such as this contract) cannot be channelized.

## 3 Knowledge

It is important to note that counterfactual($P, X$) differs from *knowledge* of counterfactual($T, X$). Suppose in `lockedBounty`, Alice reveals the preimage only to Bob and then destroys and forgets her copy the preimage. Then other than Bob, no one else in the world (not even Alice) knows the preimage, hence counterfactual(Bob, $X$) holds. However, since Bob cannot trust that Alice really destroyed her copy of the preimage, "Bob knows counterfactual(Bob, $X$)" does not hold. This shows that, in particular, counterfactual($P, X$) does not imply that $P$ knows counterfactual($P, X$).

## 4 Negative Utility

If we take $X$ to be "Bob burns all ether in his address by sending it to the address 0x00", then counterfactual(Bob, $X$) is always true. Hence counterfactual X for negative-utility $X$ are often not very useful. However, in situations like channelized burnable payments where Alice might want to burn some money belonging to Bob, the statement counterfactual(Alice, $X$) holds. However note that in contrast to on-chain burnable payments this is "reversible" because Alice and Bob could cooperate to unburn the payment. In general, counterfactual terminology is just a statement in first-order logic over a suitably defined universe; whether counterfactual negative-utility statements are useful depends on how you model someone else's utility function and why they would want to cause you harm.

## 5 Threat Models

The threat model each actor is operating in specifies what capabilities each actor has and hence affects the definition of counterfactual terminology through the clause "without $P$ performing some action". While this has not affected any of our examples so far, it is important in examples actually relevant to state channels; in our "default" model of bounded economic risk and bounded griefing factor, in a payment channel (say with $P_i$ having balance $b_i$ for $i = 1, 2$), we can say counterfactual$(P_1, X) \wedge$ counterfactual$(P_2, X)$ where $X$ is "$b_1$ ether is transferred to $P_1$'s account and $b_2$ ether is transferred to $P_2$'s account". In this case, the action in clause 2 includes "$P_1$ fails to respond to stale data" (it is an act of *failing to perform an action*). However, if an actor is acting under the perfectly fair threat model, this is no longer the case, as griefing is possible in the payment channel.

## 6 In State Channels

In a state channel with actors $P_1, \ldots P_N$, for $X$ with positive utility, we use "counterfactual X" as a shorthand to mean "there is common knowledge that counterfactual$(P_i, X)$ for all $i$". This corresponds to our definition of $X$ being finalized in a channel.

## 7 Counterfactual Nouns

Counterfactual N for a noun N is a shorthand meaning roughly counterfactual X where X is N is placed on the blockchain. Hence a rigorous definition depends on $N$.

For instance, if $N$ is "a particular smart contract", a counterfactual contract (or what we have been calling a counterfactually instantiated contract for clarity) "exists" in the sense that interested parties (i.e. parties in the state channel) can unilaterally enforce the same consequences as $N$ (modulo the threat model).

Insofar as "the right to perform a blockchain action" is often described with nouns (e.g., "an ERC-721 token" or "a cryptokitty"), it is also possible to construct contracts which support counterfactual versions of these.

One exception to the above remarks is "counterfactual address", which does not correspond to "ethereum address" (except by analogy, i.e. they serve the same purpose).

## B The registry contract

Registry.sol

```
pragma solidity ^0.4.19;

contract Registry {

  mapping(bytes32 => address) public isDeployed;

  function deploySigned(
      bytes code, uint8[] v, bytes32[] r, bytes32[] s
  ) public returns (address) {

      bytes32 codeHash = getTransactionHash(code);
      address[] memory owners = new address[](v.length);
      for (uint8 i = 0; i < v.length; i++) {
          owners[i] = ecrecover(codeHash, v[i], r[i], s[i]);
      }

      return deploy(code, owners);
  }

  function deploy(bytes code, address[] owners) private returns (address) {

      bytes32 cfAddress = getCounterfactualAddress(code, owners);

      assembly {
          newContract := create(0, add(code, 0x20), mload(code))
      }

      isDeployed[cfAddress] = newContract;

      return newContract;
  }

  function proxyCall(address registry, bytes32 cfAddress, bytes data) public
      {
      address to = Registry(registry).resolve(cfAddress);
      require(to.call(data));
  }

  function proxyDelegatecall(address registry, bytes32 cfAddress, bytes data)
       public {
      address to = Registry(registry).resolve(cfAddress);
      require(to.delegatecall(data));
  }

}
```

Figure 10: The ***registry*** contract implemented in Solidity. Some functionality omitted for brevity.

## C Funding Protocol: Rollback and Safety Proof

Let us sketch out a short safety proof from $P_0$'s point of view. We wish to prove that at every point after his deposit (i.e., step 5) he can recover $s_0$ ether, minus a small amount of fees, from the channel no matter what the others do. Right before step 5 (i.e., after step 4c), a BalanceRefund with threshold 0 and recipient $P_0$ has been comitted to. If, after step 5 but before step 6, the other participants stop responding, $P_0$ can instantiate $r$ and $b_0$, and then call withdraw on $b_0$, sending all the ether in the multisig back to himself. $P_0$'s ability to do this continues to hold up to after the first iteration of 6e, since $b_1$ and $p_1$ cannot be finalized yet. After 6f, $P_0$ can recover $s_0$ ether by instantiating $p_1$. A similar argument holds every time step 6f is executed during another iteration. Furthermore, since $P_0$ never authorizes a withdrawal from the multisig, the requisite calls to `withdraw` on the $p_i$ and $b_0$ will not fail.

We must also consider a fair-exchange failure of `signCommitment`, for instance in step 6f (recalling that 6f executes an `Update` protocol, which contains as a subprotocol an instance of `signCommitment`). This means that $P_0$ has given out his commitment that allows 6f to complete successfully, but has not received the other commitments, so is not guaranteed that he can make 6f happen. In this case, after waiting for some subjective timeout, $P_0$ should instantiate $r$ and try to finalize it to version 0. This either succeeds, or fails because it gets finalized to version 1. In either case, $P_0$ can recover $s_0$ ether by using the withdraw function of either $b_0$ or $p_1$ respectively.

The proof for the other parties $P_k$ for $k > 0$ is similar. However, if $r$ is published and finalized at nonce $j$ with $j < k$ after $P_k$ deposits then $P_k$ will lose money; hence, $P_k$ must challenge and update the nonce to the latest available one, which is $\geq k$.

## 4 Contributing

Notice an issue with the paper? Feel free to submit a pull request to the Github repository hosting this document at https://github.com/counterfactual/paper.