Smart Contract Derivatives

Kostis Karantias[1], Aggelos Kiayias[1,3], and Dionysis Zindros[1,2]

[1] IOHK
[2] University of Athens
[3] University of Edinburgh

Abstract. The abilities of smart contracts today are confined to reading from their own state. It is useful for a smart contract to be able to react to events and read the state of other smart contracts. In this paper, we devise a mechanism by which a *derivative* smart contract can read data, observe the state evolution, and react to events that take place in one or more *underlying* smart contracts of its choice. Our mechanism works even if the underlying smart contract is not designed to operate with the derivative smart contract. Like in traditional finance, derivatives derive their value (and more generally state) through potentially complex dependencies. We show how derivative smart contracts can be deployed in practice on the Ethereum blockchain without any forks or additional assumptions. We leverage any NIPoPoWs mechanism (such as *FlyClient* or *superblocks*) to obtain succinct proofs for arbitrary events, making proving them inexpensive for users. The latter construction is of particular interest, as it forms the first *introspective* SPV client: an SPV client for Ethereum in Ethereum. Last, we describe applications of smart contract derivatives which were not possible prior to our work, in particular the ability to create decentralized *insurance* smart contracts which insure an underlying on-chain security such as an ICO, as well as futures and options.

1 Introduction

Smart contracts [4,16] on blockchain [11] platforms have limited capabilities even when developed in Turing Complete languages such as Solidity. They are executed in their own isolated environment, with full access to their own state, but limited access to what is happening in the rest of the blockchain system. This inherently limits them to performing isolated tasks, unless they interoperate with smart contracts designed explicitly to work together with them.

In this work, we put forth a mechanism which allows so-called *derivative* smart contracts to read the (potentially private) state of other, so-called *underlying*, smart contracts, inspect any events they have fired and when, and more generally react arbitrarily to any changes in the execution

of other contracts. Notably, unlike any previous mechanism, the underlying contract may not be designed (or willing) to work with the derivative contract and hence our mechanism allows it to remain *agnostic* to the interaction. Like financial derivatives, smart contract *derivatives* can derive their value from the performance of, potentially multiple, underlying contracts. The dependence between the contracts can be arbitrary.

We develop our solution in the form of a Solidity contract which can be used as an *oracle* to create a derivative contract. We give three options for the instantiation of the oracle contract. The first is based on a special Solidity feature and is the cheapest to implement and use. The second is based on the BTCRelay [5] design and requires helpful users to submit every block to this oracle contract. Finally the third draws from the design in [8] and harnesses the power of Non-Interactive Proofs of Proof-of-Work (NIPoPoWs) [7] for efficiency. The oracle smart contract may be of independent interest, as it functions as an Ethereum SPV client running within Ethereum itself and is the first such *introspective* SPV client of its kind.

Previous work. Granting smart contracts access to data external to their blockchain environment has been studied in the context of oracles in which additional trust assumptions are made by the introduction of a trusted third party or committee [17] or in the form of an oracle [18]. The generic transfer of information *between* blockchains without the introduction of additional assumptions has been studied in the context of sidechains [8,5,12,9].

Contributions. Our contributions are summarized as follows:

1. We posit the problem of smart contract derivatives and put forth a construction solving it without additional assumptions.
2. We propose three instantiations of our oracle; the first relying on features of Solidity, the second inspired from the BTCRelay design and the third utilizing NIPoPoWs.
3. We introduce the first *introspective* SPV client, a client for a blockchain system running within that same blockchain system.
4. We discuss how our scheme can be used to instantiate some standard financial derivatives: insurance, futures and options.

2 Introspective SPV

Notation. We use $x \xrightarrow{?} a$ to denote the Merkle Tree [10] inclusion proof for tree root a and element x. We use $x \xrightarrow{?} a[k]$ to denote the Merkle-

Patricia Trie proof for the assignment of key k to x in the MPT rooted at a. The verification result of a concrete proof π is denoted $(\cdot \xrightarrow{\pi} \cdot) \in \{\mathsf{true}, \mathsf{false}\}$.

For our construction, we define an *oracle* smart contract that can answer queries regarding other smart contracts. Notably, the oracle contract is decentralized, i.e., it does not make additional trust assumptions and so does not require a trusted third party or committee. The oracle contract can answer the following queries about arbitrary underlying contracts:

1. Retrieve the value of a private state variable of the underlying contract at any point in the past.
2. Recall whether a particular event was fired by the underlying contract at any point and retrieve the values of the event parameters.
3. Report on any past calls made to the underlying contract's methods, whether these were done by other contracts or by normal accounts, including the values given to the parameters and money paid during the call.

Solidity already has some provisions for smart contract interoperability. For example, a token contract usually follows the ERC-20 interface [15], which allows any other contract to inspect its state and perform actions on it.

Even the most helpful smart contracts currently deployed however would come with limitations. Specifically, reading incoming and outgoing transactions and events emitted is currently impossible. While we could manage to partially work around those limitations with a smart contract framework for helpful contracts that records all relevant transactions and events and makes them easily accessible, it is important to remember that smart contracts are immutable and cannot be changed once deployed. Thus, this solution would not work for existing smart contracts that we may be interested in.

Additionally, this solution comes with extra storage requirements. Storage on Ethereum costs disproportionately more that any other operation, and this cost would have to be paid by the unlucky downstream users of the smart contract. Naturally this presents the need for a solution that does not incur such costs on the downstream users, which we are going to present shortly.

Private variable lookup. Assume a legacy smart contract has a variable of interest that happens to be private. This means that with regular Solidity methods this variable cannot be accessed. We show how this can be worked around with the help of an untrusted third party who

provides some proof. Provided one knows an actual block hash b on the best chain, one only has to check two MPT proofs to ensure what the value of the private variable px is, namely $px \xrightarrow{?} \mathsf{storageRoot}[loc(px)]$ and $(\_, \_, \mathsf{storageRoot}, \_) \xrightarrow{?} b.\mathsf{stateRoot}[addr]$ where $loc(px)$ refers to the persistent storage location of the variable px and $addr$ refers to the smart contract address.

Detecting transactions. Recall that Ethereum stores an MPT root of all transactions in its header. Thus the MPT proof $tx \xrightarrow{?} b.\mathsf{transactionsRoot}[H(tx)]$ suffices as proof that $tx \in b$. These proofs are already used in practice to prevent front-running [1].

The above operations can be performed as long as our smart contract can verify that a block header b is part of the current chain. We propose several mechanisms of doing so.

`BLOCKHASH` opcode. Ethereum offers the `BLOCKHASH` opcode that allows a smart contract to obtain previous block hashes. This functionality makes ensuring that a provided block b is in the best chain trivial: the contract extracts $b.\mathsf{height}$, invokes `BLOCKHASH` for that height number and compares $H(b)$ with the result of the `BLOCKHASH` invocation. If those match, the block is valid. Unfortunately this functionality is limited to the past 256 blocks [16]. There is a proposal to remove this artifical limit which is expected to be implemented in a future hard fork of Ethereum [3]. For Ethereum, this is the ideal solution to the block verification problem, resulting in the least possible costs for proving events.

BTCRelay-style SPV. BTCRelay [5] rose to prominence in 2016 as a way to provide Bitcoin SPV client capabilities to any Ethereum smart contract. Every block is submitted to the contract by helpful but untrusted participants and a header-chain is formed and confirmed. A convenient mapping is kept so that it can be decided if any block is in the current best header-chain. BTCRelay also offers incentives for submitters of blocks, where the submitters get rewarded whenever the blocks they have posted are used to prove some event. This scheme can be used for block verification of the Ethereum chain on Ethereum — an "ETCRelay."

NIPoPoWs. NIPoPoWs [7,2] are succinct strings that prove statements about some chain. Their succinctness makes them perfect candidates to use as proofs for block inclusion on an Ethereum smart contract. Details on their use for this scenario are presented in [8]. Note that this use comes with a host of incentives via collateralization that should be implemented for use in our Introspective SPV client.

Implementation. We summarize all these functionalities in the complete *Introspective SPV* contract shown in Algorithm 1. This is, to our knowledge, the first contract that is an SPV client for its host chain.

Algorithm 1 Introspective SPV contract for Ethereum, on Ethereum.

```
1: contract introspective-spv
2:     function submit-block(b, π)
3:         if ¬verify(b, π) then
4:             return ⊥
5:         end if
6:         valid-blocks ∪= {b}
7:     end function
8:     ▷ verify-* functions return false if b ∉ valid-blocks
9:     function verify-tx(tx, b, π)
10:        return tx --π--> b.transactionsRoot[H(tx)]
11:    end function
12:    function verify-storage(val, loc, addr, b, π)
13:        return ∃σ : val --π[0]--> σ[loc] ∧ (_, _, σ, _) --π[1]--> b.stateRoot[addr]
14:    end function
15:    function verify-event(evt, addr, b, π)
16:        return evt.src = addr ∧ ∃i, r_l : evt ∈ r_l ∧ (_, _, r_l, _) --π--> b.receiptsRoot[i]
17:    end function
18: end contract
```

We remark that using any storage is not necessary and it is only used for illustrative purposes. All functions can be made to operate based on only arguments they receive, without compromising their security.

3 Concrete Instances

We now move to some notable applications that can be accomplished by contracts which build on the Introspective SPV functionality.

Insurance. A quite useful application of a smart contract derivative is the ability to provide *insurance* for an underlying smart contract. This is a contract between an *insurer* and a *policyholder* account. The contract works as follows. Initially, the *insurer* creates the insurance contract, depositing a large amount of money to it to be used as liabilities in case of claims. Subsequently, after checking that the deposited amount secured against liabilities is sufficient, the future *policyholder* account deposits the *premium* as a payment to the insurance contract, which signs them up for the insurance. The premium can also be paid in installments if desired. Once the premium has been paid, the policy is activated for the particular policyholder.

The derivative smart contract insures against a covered loss event which pertains to an underlying smart contract. Unlike traditional insurance contracts, assessing whether a claim is valid or not is not left up to the insurer or courts of law, but is determined by the smart contract in a predetermined and decentralized manner. As such, there can be no disputes on whether a coverage claim is valid.

One such example constitutes insuring an underlying *ICO* smart contract [13] against a specified loss condition. The condition describes what the policyholder considers to be a failed outcome of the ICO. For instance, the policyholder can specify that the ICO's success requires that there are at least 5 whale investors, defined as investors each of which has deposited more than $1,000,000 in a single transaction over the course of the ICO's fundraising period.

Insurance claims in this example are made as follows. If the insured determines that there has been a loss event (i.e., there have been fewer than 5 whale investors), then at the end of the ICO's fundraising period, they submit a claim. This claim does not include any proof. The opening of a claim initiates a *contestation* period during which the insurer can submit a counter-claim illustrating that the claim was fraudulent. This counter-claim *does* include a proof, which consists of 5 transactions made to the ICO smart contract each of which pertains to a different investor and is valued more than $1,000,000. This counter-claim proof can be checked for validity by using the means described previously. If there are no counter-claims within the contestation period, then the claimant is compensated. In case the policyholder acts adversarially, making a fraudulent claim, the insurer can always present this counter-claim and avoid paying compensation. In case the insurer acts adversarially, electing not to pay compensation when required to do so, the policyholder will make a claim against which the adversarial insurer will not be able to provide a counter-claim.

It is noteworthy that the contract should be resistant to attacks where a malicious policyholder continuously makes false claims that the honest insurer has to defend against causing them monetary loss. To prevent such attacks, the smart contract may request some collateral from the policyholder when claiming, that is taken from them if they are making a false claim and returned to them when they are making a truthful claim. Such incentive mechanisms have been the subject of extensive study in previous work [14].

Options. Traditionally an option is a contract between a *holder* and a *writer*. It allows the holder to either buy (*call option*) or sell (*put option*)

an underlying asset to the writer at a specified *strike price* and until a specified *expiry date* [6]. If the holder elects to exercise the option, the writer is obligated to complete the trade. An option can be traded on the open market like any other asset. Buying an option ensues that the existing holder forfeits their contractual rights and transfers them to the buyer, making them effectively the new holder.

Centralized exchanges have a plethora of ways of enforcing the legal obligations of writers. Specifically, if a writer does not fulfill the valid request of a holder, their account may be frozen, their funds may be seized and further action may be taken against them through the traditional legal system.

To implement options in a decentralized manner we assume the existence of a clearing house, which can be implemented as its own smart contract, that will insure the holder against the event that the writer does not fulfill her contractual obligations. A responsible options buyer only buys an option that comes with a guaranteed insurance similar to the one outlined in the previous section. If the writer fails to fulfill her contractual obligations, a claim with the clearing house is started. A successful claim would be of the form: "On some block that occurred after the start of the option contract and before its expiry, I (the holder) requested to exercise my option. By the block of expiry, no event was fired indicating that my writer acted to fulfill my request for the requested price and amount." After the contestation period, the holder is refunded by the clearing house smart contract.

Futures. Similar to an option, a future is a contract between two parties. The defining difference from an option is that the holder is *obligated* to perform a specified action (either buy or sell) on the underlying asset at the strike price and on the specified expiry date [6]. For simplification the expiry date can be described as a block height inside the smart contract. It is easy to see how this system can also be implemented with the help of a clearing house, similarly to an option. Plainly, in case of fraud, the policyholder could claim that "Between the start of the agreement and the expiry, no Solidity event was fired indicating that the writer bought or sold from me the agreed amount at the agreed price." In case of a fraudulent policyholder, all the clearing house has to do is provide proof that this event was fired in some block in the period of interest.

On the availability of insurers. Exchanges implicitly offer insurance for their users by keeping track of how much money they store with them and making sure they are not over-exposed to risk. Banks implicitly offer insurance for their customers based on their credit-worthiness. The

same out-of-band criteria can apply for any institution wishing to insure on-chain. An insurer can create an off-chain agreement with the party who can cause a loss and claim, or rely on some on-chain collateral, potentially denominated in multiple tokens/currencies, to be automatically compensated in case of misbehavior.

References

1. Lorenz Breidenbach, Phil Daian, Florian Tramèr, and Ari Juels. Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1335–1352, 2018.
2. Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. Flyclient: Superlight clients for cryptocurrencies. *IACR Cryptology ePrint Archive*, 2019:226, 2019.
3. Vitalik Buterin. EIP 210: Blockhash refactoring. Available at: https://eips.ethereum.org/EIPS/eip-210.
4. Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.
5. Joseph Chow. BTC Relay. Available at: https://github.com/ethereum/btcrelay.
6. John Hull. *Options, Futures and Other Derivatives*. Pearson, 2017.
7. Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. Non-Interactive Proofs of Proof-of-Work. In *International Conference on Financial Cryptography and Data Security*. Springer, 2020.
8. Aggelos Kiayias and Dionysis Zindros. Proof-of-Work Sidechains. In *International Conference on Financial Cryptography and Data Security Workshop on Trusted Smart Contracts*. Springer, 2019.
9. Sergio Damian Lerner. Drivechains, sidechains and hybrid 2-way peg designs, 2016.
10. Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.
11. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Available at: https://bitcoin.org/bitcoin.pdf, 2008.
12. Paul Sztorc. Drivechain - the simple two way peg, November 2015. http://www.truthcoin.info/blog/drivechain/.
13. Jason Teutsch, Vitalik Buterin, and Christopher Brown. Interactive coin offerings. *Available at: https://people.cs.uchicago.edu/~teutsch/papers/ico.pdf*, 2017.
14. Jason Teutsch and Christian Reitwießner. Truebit: a scalable verification solution for blockchains, 2018.
15. Fabian Vogelsteller and Vitalik Buterin. ERC-20 token standard. Available at: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md, 2015.
16. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.
17. Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security*, pages 270–282, 2016.
18. Fan Zhang, Sai Krishna Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating Web Data Using Decentralized Oracles for TLS. *arXiv preprint arXiv:1909.00938*, 2019.