Short Paper: Deploying PayWord on Ethereum

Abstract. We revisit the 1997 PayWord credit-based micropayment scheme from Rivest and Shamir. We observe that smart contracts can be used to augment this system, apply to 'claim or refund' paradigm of cryptocurrencies to remove the counter-party risk inherent in PayWorld, and use a smart contract to 'staple' real value (in Ether) to payments in the system. Our implementation is more concise than any Ethereum payment channel we are aware of and the offline payments are very compact values (264 bits). It only uses hash functions and not digital signatures. EthWord becomes cheaper than standard Ethereum transfers when more than 16 payments between the same participants are made and appears to maintain its advantage for up to 1700+ transactions, at which point signature-based payments become cheapest. The main drawback of EthWord is the moderate gas price of using the system—despite dropping signatures, it is still priced out of the micropayments use-case. Like any payment channel, requires only two on-blockchain function calls to open and close the channel, while allowing the rest of the payments to be made off-blockchain.

1 Introduction

PayWord is a credit-based payment system, envisioned for small payments proposed by Rivest and Shamir [22]. The mechanics we will turn to later, but for now, the reader can think of tokens being issued that have some value. The key advantage of PayWord is its efficiency and succinctness drawn from using only hash functions. A limitation of PayWord is that tokens do not have inherent value; their value is based on the trust assumption that a counter-party will honour the value ascribed to them. With Ethereum and other blockchain technologies, we can fix this issue by stapling cryptocurrency to the token through the use of a smart contract. Finally, while Ethereum already has internal functionality for payments, EthWord enables payments to be made off-blockchain and settled once on-blockchain.

This transformation turns PayWord from a trust-based credit system to an escrow-based payment system; not unlike offline payment channels and networks being proposed for Bitcoin — the Lightning Network being the most prominent [20]. It is known that an Ethereum-based payment channel will be less complex than a Bitcoin one, since most of the complexity of Bitcoin-based payments channels comes from Bitcoin's limited scripting language [13]. EthWord is a uni-directional payment channel that can be chained into a payment network and has very compact (*e.g.,* 256-bit) payments. It thus might be an interesting primitive to enhance in the same ways other payment channels [4,20] have been: adding features [10], increasing efficiency [6,15], and adding transactional privacy [7,12,8,23].

2 Background

Beginning in the 1980s, a significant amount of the cryptographic literature has been devoted to the design of e-cash systems. In the 1990s, many startups worked toward deployment of this technology but most ultimately failed [17]. By late 2008, when Bitcoin was first proposed [16], innovation on both the academic and commercial side of digital cash had dried up. Now Bitcoin's success has breathed new life into the field: cryptocurrencies have billion dollar market capitalizations and academic conferences like *Financial Cryptography* are again publishing papers on financial cryptography.

At first glance, Bitcoin seems like a major departure from the e-cash systems from the 80s and 90s. In reality, its 'academic pedigree' is a novel combination of pre-existing ideas [18]. Similarly, researchers are re-discovering long lost ideas from the e-cash literature and finding new ways to apply them in a blockchain world. For example, blinded coins were a staple of e-cash [3] that re-emerge, along with accumulators [24], in post-Bitcoin systems like zcash [14,25]. Enabling micropayments through lottery-based probablistic payments of macropayments was explored in the 90s [21,27,9] and re-emerged for Bitcoin [19]. In this paper, we 're-discover' the 1997 payment system PayWord from Rivest and Shamir [22].

3 Preliminaries

Hash chains. A hash chain [11] is constructed by iteratively applying a public one-way hash function $\mathsf{H}()$ on a random value s. Let the notation $\mathsf{H}^{i+1}(s) = \mathsf{H}(\mathsf{H}^i(s))$. A hash chain of length $n+1$ is:

$$\left\langle s, \mathsf{H}(s), \mathsf{H}^2(s), \mathsf{H}^3(s), \ldots, \mathsf{H}^{n-1}(s), \mathsf{H}^n(s) \right\rangle$$

where s (technically equivalent to $\mathsf{H}^0(s)$) is called the *seed* and $\mathsf{H}^n(s)$ is called the *tip*. Given the hash is preimage resistant against a computationally bounded adversary, knowing some value in the chain $\mathsf{H}^x(s)$ does not reveal any values 'up' the chain from it, including the seed: $\left\langle s, \ldots, \mathsf{H}^{x-1}(s) \right\rangle$. Conversely the value $\mathsf{H}^x(s)$ can be iteratively hashed to produce the rest of the values 'down' the chain ending up producing the tip value.

Recognition. If Alice meets Bob at a party, Bob can give the tip of a chain to Alice as a token [1]. Later when Bob meets Alice again, he can provide $\mathsf{H}^{n-1}(s)$ as proof he is the the same person that gave her the token. On the subsequent visit, he provides $\mathsf{H}^{n-2}(s)$ and so on for n visits. Of course, Bob could more directly provide Alice with his public key and sign messages each visit, however hash chains avoid the relatively expensive public key operations of a signature scheme.

Payments. In PayWord, recognition is used for credit-based payments. A PayMaker generates a length $n+1$ hash chain and provides a signed[1] tip to a

[1] The signature is for non-repudiation of the financial arrangement, not for future authentication.

PayTaker. They agree that each preceding value in the hash chain has a specified unit of value owed to the PayTaker by the PayMaker. For example, say n is 100 and the value of each hash in the chain is a \$1 debt owed to the PayTaker. To expense \$27, the PayMaker provides $\mathsf{H}^{n-27}(s)$ to the PayTaker who will verify that hashing it 27 times produces the signed tip. The PayMaker can increase the amount by sending further hashes, up to \$100 (the *capacity*), after which, the payment channel is *exhausted* and must be reinitiated.

Payment channels. Payment channels were reconceived for Bitcoin [4,20] to offer offline payments between Alice, Bob, and possibly with some intermediaries relaying transactions. In Bitcoin, payment channels work the same as EthWord (in other words, EthWord *is* a payment channel) but involve setting up a number signed transactions (some pushed to the blockchain and others held in reserve) and the payments themselves are one or more full and signed Bitcoin transaction. While EthWord is a payment channel, it is a simple one. It can only send payments from the PayMaker to the PayTaker (thus it is *unidirectional*) and it can only send payments in increasing amounts (thus it is *monotonic*). Making a bidirectional payment channel, where payment values can be increased and decreased arbitrarily, is interesting future work.

Pay50. A recent blog post by Di Ferrante argues for the simplicity of Ethereum-based payment channels (relative to Bitcoin) and he offers a '50 lines of code' Solidity implementation of a uni-directional, monotonic channel we will name Pay50 for the purposes of this paper [5]. As a deliberate barebones implementation, it is simple and it relies on offline payments to be signed by the sender. We describe it further in the next section.

4 EthWord Implementation

EthWord is a line-by-line replication of Pay50, replacing the use of digital signatures with hash functions as described in the original PayWord proposal. We slightly modernize Pay50 to make it compliant with changes introduced in the Solidity language.[2] We replicate Pay50 to enable an isolated comparison between a signature-based approach (Pay50) and hash-based (EthWord) approach.

The primary issue with PayWord is that payments have no actual value and only represent an agreement to pay. In EthWord, we staple Ethereum's internal currency ether (ETH) to the payments through a smart contract to give them real value. Thus EthWord eliminates the counter-party risk in accepting payments that is inherent in PayWord, and this is only possible because payments are backed by both a digital currency and a decentralized execution environment.

Both Pay50 and EthWord follow the standard paradigm used in the literature to eliminate counterparty risk (sometimes called *claim-or-refund* [2]). If Bob (the PayMaker) wants to send up to X `ETH` to Alice (the PayTaker), he prepays by loading X `ETH` into a smart contract that the PayTaker can withdraw from when specified conditions are met. The PayMaker also sets a deadline for the

[2] The source code of Pay50 and EthWord are included in the appendix.

1. The PayMaker runs the constructor of EthWord.
2. The PayMaker opens the contract by specifying the identity of PayTaker, the validity period of the channel, how much each hash is worth, and funds the contract. The PayMaker will send the contract address to PayTaker.
3. The PayTaker will check the parameters of the contract to ensure it is funded, how long she has to settle the account before the PayMaker can withdraw his deposited funds, and the total amount of the deposited funds. When satisfied, she stores the hashchain tip offline.
4. Offline, the PayMaker will make payments by sending hash values. The PayTaker will check that the value iteratively hashes to the tip for a correct number of iterations corresponding to amount of payment she expects. If PayMaker wants to make successive payments, they send a new hash that represents the new total amount to be paid to the PayTaker.
5. At any time while the contract is open, PayTaker can submit a hash value and receive the appropriate payment. If the PayTaker has not run this function and the validity period expires, the PayMaker can withdraw all the money in the contract and close it.

Protocol 1: The on-blockchain and off-blockchain steps in EthWord payments

PayTaker to withdraw, after which he can release the escrowed funds back to himself. The PayTaker checks that the contract is properly formed and funded; only then will she accept payments from the PayMaker.

4.1 EthWord Code Design

As EthWord is a modification of Pay50, we will discuss the design of both in parallel. Both use the constructor to initially setup the contract. In addition, they have two functions that both close the channel: one is used by the PayTaker to claim a payment and other is used by the PayMaker to dissolve the contract after it has timed out. EthWord is summerized in Protocol 1. The constructor for both Pay50 and EthWord establishes the core components of the contract:

- PayTaker: `msg.sender` for the contract creation.
- PayMaker: an address passed into the constructor.
- Total available funds: the constructor allows an amount of Ether to be transferred to the payment channel contract (the constructor is marked `payable`).
- Timeout: a validity period passed into the constructor. The contract also stores the block timestamp of when the constructor was run. These values are added together and when they exceed any future block timestamp, the self-destruct function is permitted to run allowing the PayMaker a refund.

Note that as implemented, the timeout functionality in both is timestamp dependent which could enable the PayMaker to refund earlier than allowed, or

EthWord Function	Gas	ETH	USD
Channel	312 031	0.00539	$0.689
closeChannel (50)	18 905	0.00033	$0.042
closeChannel (100)	22 205	0.00038	$0.049

Table 1. Function cost. Since claim is dependent on how long the hash chain is for the claimed payment, we shows costs for length 50 and length 100.

alternatively be locked out from refunding, as a result of miner manipulation of the timestamps. This is feasible for adjustments of approximately 900 seconds. Therefore, the contract timeout should be considered 'fuzzy' and should be sufficient when run over the course of hours, days, or years; which is standard for proposed use cases of payment channels.

For EthWord specifically (not Pay50), the constructor also establishes the payword tip and the amount of Ether each payword is worth. Consider a contract that holds 1.00 `ETH` in escrow and the PayTaker supplies proof that they are entitled to, say, 0.45 of the 1.00 ETH by invoking this function. For Pay50, the proof is the claimed amount as signed by PayMaker and the contract validates the signature. For EthWord, it is a payword (*i.e.,* the output of a hash function). In this case, the PayMaker might make a hash-chain of length 100, construct the contract with the tip, specify the value of each hash as 0.01 `ETH`, and funds the contract with 1.00 `ETH`.

Note that the contract does not care about the length of the hash chain because there is no simple way (nor reason) for the PayTaker to actually verify the length of the hash-chain. If it is too short, then PayMaker cannot make payments after a certain point. If it is longer, than PayTaker needs to stop accepting payments in excess of what she has verified to the contract to hold. Similarly, since the length of the hashchain is unknown to PayTaker, the contract does not require a specific amount to be funded. The PayTaker will just treat this amount as the maximum.

The PayMaker can make an offline payment to the PayTaker by sending a hash (again see Protocol 1). Note that the hash is in no way bound to the identity of the PayTaker—the smart contract binds the use of the hash to the PayTaker. Next, note that technically the PayTaker can compute the chain and submit any hash from this chain to `claim()`, however they are incentivized to send the most valuable hash. For this same reason, the PayMaker can then later 'up the payment' by sending a more valuable hash to the PayTaker. This can be repeated until the PayMaker runs out of hashes or PayTaker wants to run `claim()`. This is called *replace-by-incentive* [13] in the payment channel literature.

4.2 Forming payment networks

Consider a third party, in addition to the PayMaker and the PayTaker, called an intermediary. If PayMaker establishes an EthWord channel with the intermediary and the intermediary establishes an EthWord channel with PayTaker, and both channels use the same tip, then payments can be routed through the intermediary without trusting it. This requires one small modification: PayTaker can run

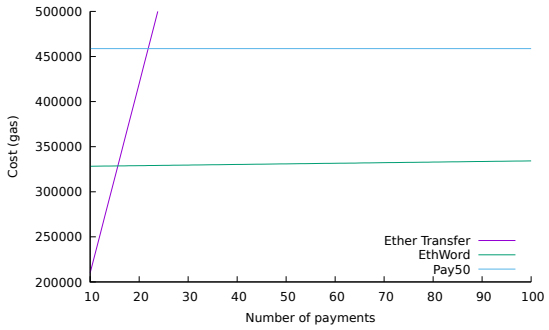

Fig. 1. The total gas cost of running the payment channels EthWord, Pay50, and the internal Ether Transfer as a function of the number of payments. Internal transfers are most economical up to 16 transactions, then EthWord is most economical, and we extrapolate that Pay50 (at a gas cost of around 460 000) will only become more economical when the transactions exceed 1700.

`claim()` in both contracts. It can also admit further modifications: for example, the intermediary might modify `claim()` so that it keeps some fraction of the total payout as a fee.

5 Evaluation

Footprint. Relative to Pay50, EthWord does not add to the lines of code; in fact, it even shaves a few off. The more important property is the size of the payment sent to the receiver; this is reduced from a digital signature to a hash or from 776 bits to 264 bits. Note that it is even possible to reduce EthWord to 256-bits; an extra 8 bit value representing the length of the hash from the tip is included for a more convenient loop.

Gas Costs. As of January 15th, 2019, the weighted average price of 1 gas is 17.26×10^{-9} ETH[3] and the exchange rate of 1 ETH to USD is \$127.85.[4] Table 1 shows the gas costs of each function in EthWord if run successfully. The cost of the claim function includes checking if the provided payment (hash) is part of the hash chain (if when iteratively hashed, it results in the tip value). Thus the cost of claiming will vary on how many times the hash must be iterated. For example, consider a channel with 100 payment values worth 0.01 `ETH` each. Running claim on the payment value representing 0.05 `ETH` will require hashing the value 5 times. The payment value of 0.95 `ETH` will require 95 hashes.

Figure 1 shows the total gas cost of running the payment channels EthWord, Pay50, and the internal ether transfer as a function of the number of payments (from 1 to 100). At 100, the cost by EthWord is 334 236 which is still about 30% less than the cost of running Pay50 that must verify a digital signature (*i.e.,* Pay50 uses Solidity's `ecrecover` with some additional processing logic).

[3] https://ethgasstation.info/
[4] https://coinmarketcap.com/

Contract Security. As mentioned above, the contract depends on manipulatable timestamps for allowing a refund after an elapsed time. This is not a problem but requires the PayTaker to be aware. Further, once the contract is refundable, the PayTaker can still close the contract and receive payment assuming they have a payment proof. If PayTaker and PayMaker try to close the contract at the same time, transaction ordering will be arbitrary, subject to a gas auction, and subject to miner manipulation. Once again, we do not consider this a problem as long as the PayTaker is aware. Before the timeout, they have exclusive control over closing the contract.

Last, consider a case where the PayTaker is given a payment of 0.45 `ETH` from a contract holding 1.00 `ETH`. After receiving the 0.45 `ETH` at the address of the PayTaker (call it T), note that T may be a contract address and if so, it's fallback function will be allowed to run. Logically, this function could recall close and result in an addition 0.45 `ETH`— a reentrancy attack. The mitigation is the standard one: using `send` which does provide enough gas to T's fallback function to make an additional function call.

6 Discussion

Porting to Bitcoin. Bitcoin's scripting language is purposely limited, compared to Ethereum, to ensure scripts execute efficiently and support Bitcoin's core functionality of digital money. Many PayWord components are supported in Bitcoin script, including but not limited to locking transactions with a hash image that requires a pre-image to spend; and the ability to iteratively hash elements. However it does support looping nor dynamically changing how an output can be split. A moderate extension to Bitcoin's scripting language could enable PayWord on Bitcoin; one proposal is MicroBTC [26].

Micropayments. With a total gas cost (to construct, open, and claim within a contract) of $0.75 or more, EthWord (or other Ethereum-based payment networks) are not suitable for true micropayments. Even to send $100 of value, it represents a 0.75% fee. The simplest internal Ethereum transaction costs 21 000 gas so EthWord will have to replace 16 transactions to pay for itself.

Prepaying. A second limitation that underlies almost all payment channels is the fact that payments have to be *prepaid*. Without some broader economic infrastructure, a payment channel or network is similar to using prepaid cards, something most customers choose to do with fiat currencies unless if they are compensated (generally, customers pay for credit; preloading a card or account is giving the merchant credit which they should also pay for[5]). If Alice were to pay all her bills for a single year using EthWord (or other payment channel), she would have to have enough Ether for an entire year on the first day of the year. More many people, this would be a cash flow issue.

Trickling. One issue in payments is fairness or fair exchange. When the payment is made on-blockchain for a token that is already on-blockchain, the

[5] For example, a $50 Apple Store prepaid card might sell for $40 or using a preloaded Starbucks app might result in rewards that can be redeemed for future purchases.

swap of payment for token can be made atomic. However when the purchase is off-blockchain, either the purchased good or the payment has to be released first, leading to counter-party risk. Some purchases are divisible (*e.g.,* electricity purchased to charge an electric car) and in these cases, payment channels like EthWord are useful for *trickling* small payments in exchange for small divisions of the purchased good. If one party unfairly aborts, the value that is forfeited is small and bounded. Trickling can also be used for sending funds via an untrusted intermediary when the payment network approach cannot be used — *e.g.,* if the intermediary is a mixing service that is anonymizing the payment stream amongst other indistinguishable output payment streams.

References

1. R. Anderson, F. Bergadano, B. Crispo, J.-H. Lee, C. Manifavas, and R. Needham. A new family of authentication protocols. *SIGOPS Oper. Syst. Rev.*, 32(4), 1998.
2. I. Bentov and R. Kumaresan. How to use bitcoin to design fair protocols. In *CRYPTO*, 2014.
3. D. Chaum. Blind signatures for untraceable payments. In *CRYPTO*, 1982.
4. C. Decker and R. Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *SSS*, 2015.
5. M. Di Ferrante. Ethereum payment channel in 50 lines of code. Medium, June 2017.
6. S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski. Perun: Virtual payment channels over cryptographic currencies. IACR ePrint, 2017.
7. M. Green and I. Miers. Bolt: Anonymous payment channels for decentralized currencies. In *CCS*, 2017.
8. E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg. Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. In *NDSS*, 2017.
9. S. Jarecki and A. Odlyzko. An efficient micropayment system based on probabilistic polling. In *Financial Cryptography*, 1997.
10. R. Khalil and A. Gervais. Revive: Rebalancing off-blockchain payment networks. In *CCS*, 2017.
11. L. Lamport. Password authentication with insecure communication. *CACM*, 24(11), 1981.
12. G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi. Concurrency and privacy with payment-channel networks. In *CCS*, 2017.
13. P. McCorry, M. Möser, S. F. Shahandasti, and F. Hao. Towards bitcoin payment networks. In *Information Security and Privacy*, 2016.
14. I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *IEEE Symposium on Security and Privacy*, 2013.
15. A. Miller, I. Bentov, R. Kumaresan, and P. McCorry. Sprites: Payment channels that go faster than lightning. *CoRR*, abs/1702.05812, 2017.
16. S. Nakamoto. Bitcoin: A peer-to-peer electionic cash system. Unpublished, 2008.
17. A. Narayanan, J. Bonneau, E. W. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technologies*. Princeton, 2016.
18. A. Narayanan and J. Clark. Bitcoin's academic pedigree. *CACM*, 60(12), 2017.
19. R. Pass and a. shelat. Micropayments for decentralized currencies. In *CCS*, 2015.
20. J. Poon and T. Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2015.

21. R. L. Rivest. Electronic lottery tickets as micropayments. In *Financial Cryptography*, 1997.
22. R. L. Rivest and A. Shamir. PayWord and MicroMint: two simple micropayment schemes. In *Security Protocols*, 1996.
23. S. Roos, P. Moreno-Sanchez, A. Kate, and I. Goldberg. Settling payments fast and private: Efficient decentralized routing for path-based transactions. In *NDSS*, 2018.
24. T. Sander and A. Ta-Shma. Auditable, anonymous electronic cash. In *CRYPTO*, 1999.
25. E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE Symposium on Security and Privacy*, 2014.
26. Z. Wan. Personal communciations.
27. D. Wheeler. Transactions using bets. In *Security Protocols*, 1997.

Note to reviewers: If accepted, these will be moved to a GitHub repository and the paper will fit in the page limit. But to balance anonymity with providing full details to the reviewers, we include them here.

A Pay50 Source Code [5] in Solidity

```
pragma solidity ^0.4.18;

//Code by @mattdf (modernized slightly by _anon_)

contract Channel {

    address public channelSender;
    address public channelRecipient;
    uint public startDate;
    uint public channelTimeout;

    mapping (bytes32 => address) public signatures;

    function Channel(address to, uint timeout) public payable {
        channelRecipient = to;
        channelSender = msg.sender;
        startDate = now;
        channelTimeout = timeout;
    }

    function closeChannel(bytes32 h, uint8 v, bytes32 r, bytes32 s, uint
         value) public {
        address signer;
        bytes32 proof;
        bytes memory prefix = "\x19Ethereum Signed Message:\n32";
        bytes32 prefixedHash = keccak256(prefix, h);
        signer = ecrecover(prefixedHash, v, r, s);
        if (signer != channelSender && signer != channelRecipient) revert();

        proof = keccak256(this, value);
        require(proof == h);

        if (signatures[proof] == 0)
            signatures[proof] = signer;
        else if (signatures[proof] != signer) {
            if (!channelRecipient.send(value)) revert();
            selfdestruct(channelSender);
        }

    }

    function channelTimeout() public {
        require(startDate + channelTimeout <= now);
        selfdestruct(channelSender);
    }

}
```

B **EthWord** Source Code in Solidity

```
pragma solidity ^0.4.18;

contract Channel {

    address public channelSender;
    address public channelRecipient;
    uint public startDate;
    uint public channelTimeout;
    uint public channelMargin;
    bytes32 public channelTip;

    function Channel(address to, uint timeout, uint margin, bytes32 tip)
        public payable {
        channelRecipient = to;
        channelSender = msg.sender;
        startDate = now;
        channelTimeout = timeout;
        channelMargin = margin;
        channelTip = tip;
    }

    function closeChannel(bytes32 _word, uint8 _wordCount) public {

        require(msg.sender == channelRecipient);
        bytes32 wordScratch = _word;
        for (uint i = 1; i <= _wordCount; i++) {
            wordScratch = keccak256(wordScratch);
        }

        require(wordScratch == channelTip);
        require(channelRecipient.send(_wordCount * channelMargin));
        selfdestruct(channelSender);
    }

    function channelTimeout() public {
        require(now >= startDate + channelTimeout);
        selfdestruct(channelSender);
    }

}
```