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). The main drawback of EthWord is the moderate gas price of using the system, $7, which prices it out of the micropayments use-case. Despite this, we see value in exploring alternatives to Ethereum's internal payment system: EthWord works offline and is cheaper when more than 20 payments between the same participants are made.

1 Introduction

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 [18]. By late 2008, when Bitcoin was first proposed [17], 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 [19]. 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 [25], in post-Bitcoin systems like zcash [15,26]. Enabling micropayments through lottery-based probablistic payments of macropayments was explored in the 90s [22,28,9] and re-emerged for Bitcoin [20]. In this paper, we 're-discover' the 1997 payment system PayWord from Rivest and Shamir [23].

PayWord is a credit-based payment system, envisioned for small payments. The mechanics we will turn to later, but for now, the reader can think of tokens being issued that have some value. The key 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 a escrow-based payment system; not unlike offline payment channels and networks being proposed for Bitcoin — the Lightning Network being the most prominent [21]. After presenting our system, EthWord, we discuss its relation to payment channels (see Section 3.2). 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 [14]. 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 [4,21] have been: adding features [10], increasing efficiency [6,16], and adding transactional privacy [7,13,8,24].

2 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 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.

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

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. While open, PayMaker can always add additional funds to the contract or extend the validity period. 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

3 EthWord

EthWord is a succinct Ethereum-based smart contract written in Solidity (see Appendix A). 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.

We follow the standard paradigm used in the literature to eliminate counter-party 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 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.

3.1 Code Design

EthWord use a constructor to establish the owner of the contract. The `open()` function initializes the payment channel with the relevant information (see Protocol 1). It is the only function that can be run after the constructor. After `open()` and during the validity period, the PayTaker can run `claim()` — if successful, the contract destructs. The PayMaker can run `renew()` to extend the validity period into the future.

For example, the PayMaker might make a hash-chain of length 100, opens the channel with the tip, specifies the value of each hash as 0.01 `ETH`, and funds the contract with 1 `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[2] 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 amount of `ETH` held by the contract can be increased after opening, through the default function, but this requires one on-blockchain transaction and also requires the PayTaker to reference the blockchain to see the increase. Thus, if increases are used too often, it becomes more economical to simply send `ETH` directly with Ethereum's built-in payment system than using EthWord.

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* [14] in the payment channel literature (see next section).

3.2 Relation to Payment Channels

Payment channels were originally proposed [4,21] in Bitcoin 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.

---

[2] ride

System	SLOC	Payment Size
Pay50	37	776 bits
EthWordLite	31	264 bits

Table 1. Comparison of code complexity and size of offline payments. Notes: (1) we modernized Pay50 slightly; (2) both implementations comply with linter SolHint; (3) simple lines of code (SLOC) calculated by GitHub.

3.3 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 `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.

3.4 Porting to Bitcoin

Bitcoin's scripting language is purposely limited compared to Ethereum to ensure Bitcoin scripts execute efficiently and are generally related to Bitcoin's core functionality of digital money. Many of the components of PayWord are present 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 either looping nor any enforcement on how an output can be split—*i.e.,* once an unspent output evaluates to true, it can be spent any way specified. Therefore PayWord appears to require a moderate extension to Bitcoin's scripting language for implementation; one proposal such is MicroBTC [27].

4 Evaluation

Footprint. 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 barebones implementation, it is simple but it relies on offline payments to be signed by the sender. We deploy a lightweight version of EthWord that strips out some of the functionality (*e.g.,* more flexible funding) and security protections (*e.g.,* the state machine) of the code in Appendix A. This version, called EthWordLite, is a line-by-line replication of Pay50, substituting in EthWord functionality as relevant. As seen in Figure 1, EthWordLite 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

Function	Gas	ETH	USD
Constructor	317 299	0.0067	\$5.88
Open	12 129	0.0002	\$0.22
Claim (50)	21 511	0.0005	\$0.40
Claim (100)	25 772	0.0005	\$0.48

Table 2. Cost of running the basic contract functions. Since claim is dependent on how long the hash chain is for the claimed payment, the function shows costs for length 50 and length 100. See Figure 1 for more on this.

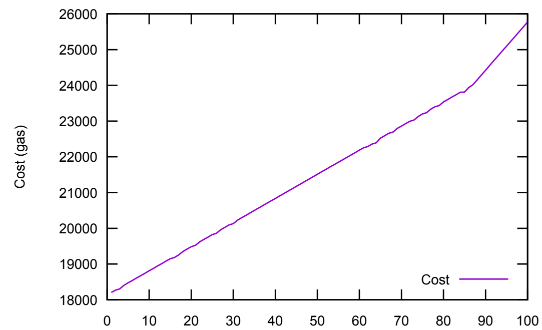

Fig. 1. The gas cost of the claim function as a function of the length of the hash chain. The gas cost of a claim in Pay50 is around 42 000.

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 EthWordLite 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 2018, the price of 1 gas is 21×10^{-9} ETH[3] and the exchange rate of 1 ETH to USD is \$882.92.[4] We show 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 \$1 each. Running claim on the payment value representing \$5 will require hashing the value 5 times. The payment value of \$95 will require 95 hashes and thus cost more.

In Figure 1, we show the gas cost of the claim function as the number as the length of the chain varies from 1 to 100. At 100, the cost is 25 772 which is still about half the cost of running a claim function that must verify a digital signature (*e.g.,* Pay50 uses the `ecrecover` operation in Solidity with some additional processing logic).

[3] https://ethstats.net/
[4] https://coinmarketcap.com/

Contract Security. In EthWord, the contract is capable of paying out the ETH it holds, requiring a transfer function like `send()`. Transfer functions introduce the possibility of a reentrancy attack, particularly when the payment is intended to be less than the capacity of the channel—*e.g.,* an attack in this case might extract double, triple, etc. We implement a state machine to enforce that functions are executed in the correct order and this includes transitioning from an open state (required to enter the function) to a locked state that will encapsulate the transfer and is not a valid state for any function. This is in addition to using `send()`, which offers little gas to the contract receiving the funds. We analyze the contract with the static analysis tool Oyente [12] to help confirm its security against reentrancy attacks.

Oyente identifies a timestamp dependency since our timeout uses time (rather than block number). This would allow a malicious miner to increase or decrease the refund time by 15 minutes, which we accept. Oyente also identifies a transaction ordering dependency. Investigate why and either fix or explain why it doesn't matter.

We also do other small things to ensure proper execution: (1) we whitelist addresses who can run each function, and (2) we use assertions to check invariances. For example, we provide a function to extend the validity period. A simple mistake such as using a signed `int` to represent the extension would permit a negative extension, enabling PayMaker to withdraw sooner than specified — thus we assert that extensions are positive numbers.

5 Discussion

Micropayments. With a total gas cost (to construct, open, and claim within a contract) of $7 or more, EthWord (or other Ethereum-based payment networks) are not suitable for true micropayments. Even to send $100 of value, it represents a 7% fee. EthWordLite is slightly leaner than EthWord but still costs . . . XXXX. The simplest internal Ethereum transaction costs 21 000 gas so EthWord will have to replace 20 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 swap of payment for token can be made atomic. However when the purchase

[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.

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. L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *CCS*, 2016.
13. G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi. Concurrency and privacy with payment-channel networks. In *CCS*, 2017.
14. P. McCorry, M. Möser, S. F. Shahandasti, and F. Hao. Towards bitcoin payment networks. In *Information Security and Privacy*, 2016.
15. 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.
16. A. Miller, I. Bentov, R. Kumaresan, and P. McCorry. Sprites: Payment channels that go faster than lightning. *CoRR*, abs/1702.05812, 2017.
17. S. Nakamoto. Bitcoin: A peer-to-peer electionic cash system. Unpublished, 2008.
18. A. Narayanan, J. Bonneau, E. W. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technologies*. Princeton, 2016.
19. A. Narayanan and J. Clark. Bitcoin's academic pedigree. *CACM*, 60(12), 2017.
20. R. Pass and a. shelat. Micropayments for decentralized currencies. In *CCS*, 2015.

21. J. Poon and T. Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2015.
22. R. L. Rivest. Electronic lottery tickets as micropayments. In *Financial Cryptography*, 1997.
23. R. L. Rivest and A. Shamir. PayWord and MicroMint: two simple micropayment schemes. In *Security Protocols*, 1996.
24. 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.
25. T. Sander and A. Ta-Shma. Auditable, anonymous electronic cash. In *CRYPTO*, 1999.
26. 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.
27. Z. Wan. Personal communciations.
28. D. Wheeler. Transactions using bets. In *Security Protocols*, 1997.

A EthWord Source Code in Solidity

```
pragma solidity ^0.4.0;

contract EthWords {

  //Initialization Variables

  address public owner;
  address public receiver;
  uint public expirationTime;
  uint public wordValue;
  bytes32 public root;
  //uint public balance; //Not sure we need this: will discuss

  //State Machine
  enum States {Init,Open,Locked}
  States state; // if state is public, reentry could reset it

  //Check sender is owner
  modifier checkOwner(){
    require(msg.sender == owner);
    _;
  }

  //Check sender is receiver
  modifier checkReceiver() {
    require(msg.sender == receiver);
    _;
  }

  //Check that contract has been expired
  modifier checkTime() {
    require(now > expirationTime);
    _;
  }

  //Check that the state of the state machine
  modifier checkState(States _state) {
    require (state == _state);
    _;
  }

  //Constructor
  function EthWords() public
  {
    owner = msg.sender;
    state = States.Init;
  }

  /**
  * @dev Specifies and opens the payment channel
  * @param _receiver Receiver's address
  * @param _validityTime Amount of time (in minutes) before Owner can claim
  * @param _wordValue Amount of Eth (in wei) each payword is worth
  * @param _wordRoot Root word for the paywords
  */

```

```
  function open(address _receiver, uint _validityTime, uint _wordValue,
      bytes32 _wordRoot) public
    payable
    checkOwner
    checkState(States.Init)
  {
    receiver = _receiver;
    expirationTime = now + _validityTime * 1 minutes;
    wordValue = _wordValue;
    root = _wordRoot;
    state = States.Open;
  }

  /**
  * @dev A function allowing receiver to claim payment
  * @param _word The receiver's payword
  * @param _wordCount The receiver's assertion of what the payword is worth
  */

  function claim(bytes32 _word, uint _wordCount) public
    checkReceiver
    checkState(States.Open)
  {
    // Lock the state to prevent reentrance
    state = States.Locked;

    // Compute the hashchain to get the root
    bytes32 wordScratch = _word;
    for (uint i = 1; i <= _wordCount; i++){
      wordScratch = keccak256(wordScratch);
    }

    // Check if root is correct
    if(wordScratch != root) {
      state = States.Open;
      revert();
    }

    // If reached, root is correct so pay the two parties
    if (msg.sender.send(_wordCount * wordValue)) {
      selfdestruct(owner);
     }

    // If send fails, unlock the function for future calls
    state = States.Open;

  }

  /**
  * @dev A function to extend the validity of the contract
  * @param _validityTime Time (minutes) to extend the validity period by
  */

  function renew(uint _validityTime) public
    checkOwner
    checkState(States.Open)
    {
      assert(_validityTime>0);
      expirationTime += _validityTime * 1 minutes;
```

```
    }

  /**
  * @dev Refund the contract to the owner after validty period is expired
  */

  function refund() public
    checkOwner
    checkTime
    checkState(States.Open)
    {
      selfdestruct(owner);
    }

  /**
  * @dev Fallback function allows payment at any time
  */

  function() public payable { }

}
```

B Pay50 Source Code [5] in Solidity

```
pragma solidity ^0.4.18;

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

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;

        signer = ecrecover(h, 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);
    }

}
```

C EthWordLite 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 {

        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);
    }

}
```