# EventWarden: A Decentralized Event-driven Proxy Service for Outsourcing Arbitrary Transactions in Ethereum-like Blockchains

Chao Li
Beijing Key Laboratory of Security and Privacy in Intelligent Transportation
Beijing Jiaotong University
Beijing, China
Email: li.chao@bjtu.edu.cn

Balaji Palanisamy
School of Computing and Information
University of Pittsburgh
Pittsburgh, USA
Email: bpalan@pitt.edu



***Abstract*—Transactions represent a fundamental component in blockchains as they are the primary means for users to change the blockchain state. Current blockchain systems such as Bitcoin and Ethereum require users to constantly observe the state changes of interest or the events taking place in a blockchain and requires the user to explicitly release the required transactions to respond to the observed events in the blockchain. This paper proposes `EventWarden`, a decentralized event-driven proxy service for users to outsource transactions in Ethereum-like blockchains. `EventWarden` employs a novel combination of smart contracts and blockchain logs. `EventWarden` allows a user to create a proxy smart contract that specifies an interested event and also reserves an arbitrary transaction to release. Upon observing the occurrence of the prescribed event, anyone in the Blockchain network can call the proxy contract to earn the service fee reserved in the contract by proving to the contract that the event has been recorded into blockchain logs, which then automatically triggers the proxy contract to release the reserved transaction. We show that the reserved transaction can only get released from the proxy contract when the prescribed event has taken place. We also demonstrate that as long as a single member in the Blockchain network is incentivized by the service fee to call the proxy contract after the prescribed event has taken place, the reserved transaction is guaranteed to get released. We implement `EventWarden` over the Ethereum official test network. The results demonstrate that `EventWarden` is effective and is ready-to-use in practice.**

## I. Introduction

Blockchains such as Bitcoin [1] and Ethereum [2] are ledgers of transactions performed by nodes in blockchain networks on a global state. Transactions form the fundamental component of blockchains as they are the primary means for users to change the blockchain state. For example, transactions allow users to transfer funds among each other in Bitcoin-like blockchains and in Ethereum-like blockchains that support smart contracts [3], transactions enable users to create new smart contracts and invoke functions within existing smart contracts. Even though many scenarios require users to release a transaction, it is very common to see that users need to release transactions for responding to prior state changes in the blockchain in a timely manner. Thus, current approaches require users to constantly keep monitoring their interested events. Event-driven Transaction (*ET*) refers to a class of service that enables a user to outsource a transaction to get executed to change the blockchain state immediately after a certain state change (or event) has taken place in the blockchain. Many scenarios require event-driven transactions in practice. For example, in a smart-contract-powered game such as the popular CryptoKitties [4] in Ethereum, Alice may want to purchase a digital kitty with a rare color (i.e., via a transaction) once the kitty appears in the market (i.e., an event) while Alice may not be capable of watching the market for days. She may use the event-driven transaction by outsourcing the transaction to someone else and require the transaction to get released to purchase the kitty immediately after the kitty becomes available. In another example, if Bob attends a smart-contract-powered auction [5] and would like to increase his bid (i.e., via a transaction) once a bid higher than his old bid appears (i.e., an event), Bob may outsource the transaction and require the transaction to get executed only after the event occurs. Besides the above-mentioned use cases of event-driven transactions, recent works on smart-contract-powered applications and protocols ranging from on-chain designs to side-chain and off-chain designs are heavily employing an emerging design pattern called *challenge-response* [6]–[9], which can also benefit from adopting event-driven transactions. Specifically, in the design of a multi-party smart-contract-powered protocol, a naive design pattern is to first verify the correctness of an action performed by a participant and then change the blockchain state based on the verified action. However, the verification of actions may be quite expensive in Ethereum-like blockchains and may become even impossible in some cases. In contrast, in the *challenge-response* design pattern, each action is assigned a time period called the challenge period, during which other participants may choose to challenge the correctness of the action with counterexamples. Participants performing incorrect actions will then be found out and their security deposits will be confiscated. Therefore, the *challenge-response* design

pattern incentivizes all participants to stay honest, which significantly reduces the cost of implementing a protocol. It is easy to see that the timely responses for challenging incorrect actions are vital to such designs, otherwise the designs become insecure. Event-driven transactions can thus facilitate the emerging *challenge-response* design pattern by outsourcing transactions for challenging events such as disputes or suspicious actions.

Most of the current implementations (e.g., BlueOrion [10] and Oraclize [11]) of event-driven transactions (*ET*) are heavily centralized. These services require the users to entirely trust the centralized servers and their security properties are solely limited to a single point of trust. More importantly, even in scenarios when the service providers are considered trustworthy, the services are still prone to unpredictable security breaches or insider attacks that are beyond the control of the service providers [12], [13]. On the other hand, the emergence of Blockchain technologies such as Ethereum [2] and smart contracts [3] provides significant potential for new security designs that support a decentralized implementation of *ET* to overcome the single point of trust issues associated with centralized approaches. Recently, a few works [7], [14]–[18] have focused on such decentralized designs in Ethereum. Nevertheless, their designs only support a certain type of event (e.g., reach of deadlines, disputes in state channels) or a single type of transaction in Ethereum (i.e., function invocation transaction).

In this paper, we propose `EventWarden`, a decentralized event-driven proxy service for users to outsource any type of transaction in Ethereum-like blockchains. `EventWarden` employs a novel combination of smart contracts and blockchain logs. `EventWarden` allows a user to create a proxy smart contract that specifies an interested event and also reserves an arbitrary transaction to release. Upon observing the occurrence of the prescribed event, anyone in the Blockchain network can call the proxy contract to earn the service fee reserved in the contract by proving to the contract that the event has been recorded into blockchain logs, which then automatically triggers the proxy contract to release the reserved transaction. We show that the reserved transaction can only get released from the proxy contract when the prescribed event has taken place. We also demonstrate that as long as a single member in the Blockchain network is incentivized by the service fee to call the proxy contract after the prescribed event has taken place, the reserved transaction is guaranteed to get released.

In summary, this paper makes the following key contributions:

- To the best of our knowledge, `EventWarden` is the *first* decentralized proxy service designed for the general use of event-driven transactions in Ethereum-like blockchains.
- After the service has been set up, `EventWarden` completely isolates the service execution from the state of users, without requiring any assistance from the user side.
- We emphasize that `EventWarden` is a general approach that supports *all* types of transaction in Ethereum, including fund transfer transaction, contract creation transaction and function invocation transaction.
- `EventWarden` also supports *all* types of events in Ethereum as long as the corresponding smart contracts write events into the blockchain logs.
- We implement `EventWarden` over the Ethereum official test network. The results demonstrate that `EventWarden` is effective and is straight-forward to be used in practice.

The rest of this paper is organized as follows: We discuss related work in Section II and introduce preliminaries in Section III. In Section IV, we propose the architecture designed for `EventWarden`. Then, in Section V, we propose the protocol designed for `EventWarden`. We present the security analysis of `EventWarden` in Section VI and the implementation and evaluation of `EventWarden` in Section VII. Finally, we conclude in Section VIII.

## II. Related work

In this section, we briefly review related studies and discuss relevant techniques, which can be roughly divided into four categories.

### A. Native mechanisms

Bitcoin was designed with a native mechanism named *Timelocks* [19]. Each transaction in Bitcoin can be set with a *nLocktime* and the transaction can only be accepted by the network after the time point indicated by *nLocktime*. Besides, each Bitcoin transaction may involve one or multiple Unspent Transaction Outputs (UTXOs) and each UTXO may include an UTXO-level timelock named *Check Lock Time Verify* (CLTV) that makes the UTXO available only after the specified time point. We consider this *Timelocks* mechanism a good way to schedule fund transfer transactions and events relative to time, but it is difficult to be generalized to support other types of transaction in Ethereum-like blockchains beyond Bitcoin or other types of events.

### B. Client-side tools

There are many tools at the client side that can achieve event-driven transactions. For example, *parity* [20], a popular Ethereum client, allows its users to prescribe a time point that they would like a transaction to be sent into the Ethereum network by the client. However, since the scheduled transaction is locally stored at users' machines before the prescribed time point, the usage of such tools demands users' machines to keep connecting with the blockchain network, which fails to isolate the service execution with the state of users.

### C. Centralized services

Oraclize [11] is a blockchain oracle service that takes the role of a trusted third party (TTP) to execute a pre-scheduled transaction on behalf of a user at a future time point. Similarly, BlueOrion [10] enables date related payments on the Stellar Ecosystem [21], an Ethereum-like blockchain. The limitations of these centralized services include both a single point of trust and a single point of control.

### D. Decentralized services

A recent project called *Ethereum Alarm Clock* [14] allows a user to deploy a request contract to the Ethereum network with a future time-frame as well as a reward and if any account is interested in the reward, the account can invoke the request contract during the prescribed time-frame to earn the reward by making the function invocation transaction maintained by the request contract get executed. However, this scheme supports only a single type of transaction and a single type of event. A more recent work [16] further employs threshold secret sharing [22] to offer privacy protection for scheduling function invocation transactions that involve sensitive arguments (e.g., bid, vote). Nevertheless, this work only supports function invocation transaction and events relative to time. Another recent work named PISA [7] enables parties in state channels to delegate to a third party, called the custodian, to cancel execution forks on their behalf. However, this work only supports events relative to state channels.

In summary, our work in this paper tackles the key limitations of the state-of-the-art decentralized approaches using smart contracts [7], [14], [16]. To the best of our knowledge, `EventWarden` is the *first* decentralized proxy service designed for the general use of event-driven transactions in Ethereum-like blockchains.

## III. Preliminaries

In this section, we discuss the preliminaries about smart contracts and Ethereum. While we discuss smart contracts in the context of Ethereum [3], we note that our solutions are also applicable to a wide range of other Ethereum-like blockchains.

### A. Account types

There are two types of accounts in Ethereum, namely External Owned Accounts (EOAs) and Contract Accounts (CAs). To interact with the Ethereum blockchain, a user needs to own an EOA by locally creating a pair of keys. Specifically, the public key $pk_{EOA}$ can generate a 20-byte address $addr(EOA)$ to uniquely identify the EOA and the private key $sk_{EOA}$ can be used by the user to sign transactions or other types of data. Then, any user can create a smart contract by sending out a contract creation transaction from a controlled EOA. The 20-byte address $addr(CA)$ of the created smart contract is generated in a deterministic and predictable way and becomes the unique identity of the contract account.

### B. Transactions and messages

The state of Ethereum blockchain can only be changed by the external world (i.e., EOAs) using transactions. A transaction is a serialized binary message sent from an EOA that contains the following key elements:

- *recipient*: the recipient account address;
- *value*: the amount of ether[1] to send to the recipient;
- *data*: the binary data payload;

[1] The native cryptocurrency in Ethereum, denoted by Ξ.

Depending on the type of *recipient*, transactions can be divided into three categories.

**Fund transfer transaction**: A transaction with an EOA as *recipient* and a non-empty *value* is a fund transfer transaction, which is used to transfer an amount of ether from the EOA creating the transaction to the *recipient* EOA.

**Function invocation transaction**: When a transaction involves an CA as *recipient* as well as a non-empty *data*, it is usually a function invocation transaction for calling a function within an existing smart contract.

**Contract creation transaction**: In Ethereum, there is a special type of transaction for creating new smart contracts. Such a transaction, usually called a contract creation transaction, carries a special *recipient* address 0x0, an empty *value* and a non-empty *data* payload. A smart contract (or contract) in Ethereum is a piece of program created using a high-level contract-oriented programming language such as *Solidity* [23]. After compiling into a low-level bytecode language called Ethereum Virtual Machine (EVM) code, the created contract is filled into a contract creation transaction as the *data* payload.

To make the transaction get executed to change the state of Ethereum blockchain, the transaction should be broadcasted to the entire Ethereum network formed by tens of thousands of miner nodes. Following the Proof-of-Work (PoW) consensus protocol [1], all the miners in Ethereum competitively solve a blockchain puzzle and the winner packages the received transactions into a block and appends the new block to the end of Ethereum blockchain. From then on, it is hard to tamper with the blockchain state updated by the transaction (i.e., transferred fund, executed function or created contract) as each miner maintains a copy of the new block and is aware of changes made by transactions within the new block and an adversary has to falsify majority of these copies in order to change the network consensus about the glocal state.

There is another important concept in Ethereum that is highly relevant to transactions (or *tx*), namely the messages (or *msg*). In Ethereum, smart contracts can be programmed in a way that enables EVM-level opcodes for system operations, such as `CREATE` and `CALL`, through which contracts are able to perform advanced operations such as creating new contracts or calling other contracts by sending out messages to change the blockchain state. It is worth noting that CAs are not able to change the blockchain state by themselves. Instead, any message creation at a CA must be triggered by a transaction sent from EOAs or another message created at other CAs, which is also triggered by EOAs from the root. This property offers two approaches to update the state of a smart contract (say, CA1) in Ethereum: (1) send a function invocation transaction ($tx$) to CA1, which directly updates the state of CA1; (2) send a $tx$ to another contract CA2, where the $tx$ triggers CA2 to send a message $msg$ to CA1 and update the state of CA1. In fact, our study found that the second approach supports not only function invocation transactions, but also the other two types of transactions in Ethereum.

### C. Gas system

In order to either deploy a new contract or call a deployed contract in Ethereum, one needs to spend Gas. Based on the complexity of the contract or that of the called function, an amount of ether needs to be spent in order to purchase an amount of Gas, which is then paid to the miner that creates the new block.

### D. Events and logs

Events are inheritable members of smart contracts and are used for emphasizing the effect of each transaction [3]. When an event is called along with a function invocation transaction, the LOG opcodes in Ethereum virtual machine store the event in the transactions log, as a part of the transaction receipt. These logs are associated with the address of the contract and will be incorporated into the blockchain and stay there as long as a block is accessible. Meanwhile, each block maintains a Merkle Patricia tree for receipts of all transactions within the block and the root of this tree, denoted as the *receiptsRoot*, is stored in the block header. For example, a contract similar to CryptoKitties [4] may be designed in a way such that each function invocation transaction for releasing a kitty to the market notifies the appearance of the kitty with an event describing the properties of the kitty (e.g., $event(kitty\_color)$). Similarly, an auction contract may employ events to notify the update of the highest bid (e.g., $event(updated)$). With events, users are capable of tracking the state changes of interested smart contracts in a very efficient way. Moreover, the facts that events are recorded in transaction receipts as logs and the Merkle root of transaction receipts are written into block header imply that the existence of events is provable using *receiptsRoot* and the reliability of the Merkle proof is endorsed by the reliability of blockchain. Motivated by these findings, we employ events and logs in the design of `EventWarden`.

In the next two sections, we present the architecture and protocol designed for `EventWarden`, respectively.

## IV. EventWarden: Architecture

In this section, we first present the key components of the architecture for event-driven transactions. We then present a naive user-driven architecture and finally introduce the proxy-contract-driven architecture designed for `EventWarden`.

### A. The four key components

An architecture for event-driven transaction (*ET*) includes four key components. First, it requires a storage place for maintaining the elements of a scheduled transaction, including *recipient*, *value* and *data* presented in Section III-B. Second, it requires an amount of ether not less than the amount indicated by the *value* element, which is guaranteed to be available when the event occurs. Third, it requires an EOA to execute the broadcasting of the scheduled transaction after the event to change the blockchain state as expected. Finally, it requires the logic to indicate the occurrence of the event. We summarize the four components as *et-data*, *et-ether*, *et-executor* and *et-logic*.

Next, we present two different architectures for *ET* and discuss how they handle the four key components, respectively. We start by introducing the user-driven architecture shown in Fig. 1.(a), where *ET* is simply performed by a user who needs to handle all these components by herself and also needs to keep connecting with the blockchain network. We then introduce the proxy-contract-driven architecture in Fig. 1.(b), where a proxy contract is employed to manage *et-data*, *et-ether* and *et-logic* and an EOA is recruited to take the role of *et-executor* so that the user is totally relieved from all the burdens of handling components and keeping on-line.

### B. The user-driven architecture

As introduced in Section III-B, a straight-forward approach of implementing event-driven transactions would be to ask the user to store the elements of the scheduled transaction at her local machine and make the transaction get broadcasted via a client-side tool such as *parity* [20] once an event occurs. In this scenario, user stores *et-data* at a local machine (i.e., a PC) with the local programs handling *et-logic* and employs a controlled EOA to both provide *et-ether* and serve as *et-executor*. Then, upon finding an occurrence of the prescribed event, the client-side tool pushes the scheduled transaction into the blockchain network from the local machine and the amount of ether indicated by the *value* element is transferred from the controlled EOA to the address indicated by the *recipient* element. In summary, this user-driven architecture is easy to be implemented. However, even though this approach allows a user to be physically absent during the occurrence of events, it still demands the local machine that maintains *et-data* to be connected with the blockchain network, which makes user-driven architecture hard to be adopted as a general approach that can isolates the service execution completely from the user side after the service has been set up.

**Algorithm 1:** Proxy contract logic

**Input :** $blockNum, blockData, MerkleProof, et\text{-}event, et\text{-}data, et\text{-}ether.$
**Output:** $msg.$

```
receiptsRoot ← verifyBlock(blockNum, blockData);
receipt ← verifyProof(receiptsRoot, MerkleProof);
result ← verifylog(receipt, et-event);
if result == TRUE then
    msgRelease(et-data, et-ether);
end
```

### C. The proxy-contract-driven architecture

To isolate the service execution from the user side, all the four components need to be migrated from the user. A naive way of realizing this goal would be to ask the user to outsource all the components to an EOA recruited from the Ethereum network so that the recruited EOA would be able to complete *ET* on behalf of the user. However, there are three severe consequences that are likely to occur: (1) the recruited EOA may falsify *et-data*; (2) the recruited EOA may

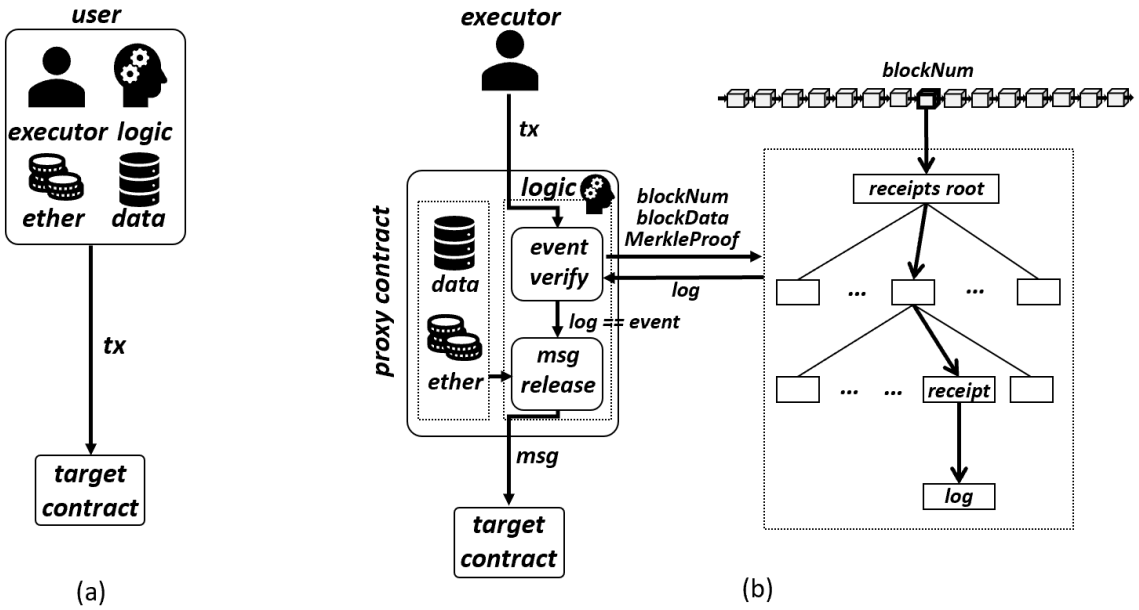


Fig. 1: The architectures for Event-driven Transaction. (a) the user-driven architecture (b) the proxy-contract-driven architecture.

embezzle received *et-ether*; (3) the recruited EOA may violate the indication of *et-logic*. All these undesirable execution results can hardly be prevented in this naive approach. Hence, a natural question is that, can we design an architecture for *ET* that offers all-or-nothing execution results? That is, we would like either the scheduled transaction to be correctly executed along with the occurrence of the prescribed event, or the scheduled transaction to be never executed and the *et-ether* to be transferred back to the user. To achieve this goal, we recognize that *et-data*, *et-ether* and *et-logic* need to be handled by a trusted party and executed in a deterministic manner, so we design a proxy smart contract ($C_{proxy}$) that can act on behalf of a user to manage the three components in a decentralized, trustworthy and deterministic way.

As illustrated in Algorithm 1, with the proxy contract, *ET* can get completed in two steps:

- *Event verify*: Immediately after the event occurs, any EOA in Ethereum is capable of calling the proxy contract $C_{proxy}$ with a function invocation transaction taking three arguments, namely the number of block carrying the event as log inside a transaction receipt ($blockNum$), the data of that block ($blockData$) and the Merkle proof for proving the existence of the receipt ($MerkleProof$). With these arguments, the proxy contract is capable of first verifying the correctness of $blockData$ according to $blockNum$ and fetching the $receiptsRoot$ from $blockData$ (line 1), then verifying the existence of the declared $receipt$ using $MerkleProof$ and $receiptsRoot$ (line 2) and finally verifying the existence of the declared log and checking whether the log is equivalent to the event prescribed by user, namely *et-event* (line 3).
- *Message release*: In case that the log correctly indicates the occurrence of the prescribed event, the proxy contract releases the reserved transaction by sending out a message using *et-data* and *et-ether*.

| notation | description |
|---|---|
| $U$ | a user of Event-driven Transaction (ET) |
| $E$ | an executor in ET |
| $C$ | a smart contract |
| $C.fun()$ | function $fun()$ within contract $C$ |
| $\Rightarrow$ | invoke a function within a contract |
| $addr(*)$ | an address of an EOA or a CA |

TABLE I: Summary of notations.

The proxy-contract-driven architecture can effectively prevent the three undesirable consequences: (1) the data recorded in $C_{proxy}$, namely *et-data*, can only get compromised by attacking the Ethereum blockchain; (2) the amount of ether in $C_{proxy}$, namely *et-ether*, is only allowed to be either transferred to the address of *recipient* or to the user using a native *selfdestruct* function, so no one would be able to embezzle *et-ether*; (3) the correctness of implementation of *et-logic* is provable, which we leave to Section VI to present in detail. Next, we present the protocol designed for `EventWarden`.

## V. EventWarden: Protocol

In this section, we start by describing the event-driven transaction (ET) as a two-phase process in the context of the proposed proxy-contract-driven architecture. We then present the protocol designed for `EventWarden`. Throughout Section V, we assume that the user is scheduling a function invocation transaction to call a function within a target smart contract denoted as $C_{target}$. While we present the protocols in the context of a single type of transaction, we note that our protocols are also applicable to the other two types. We

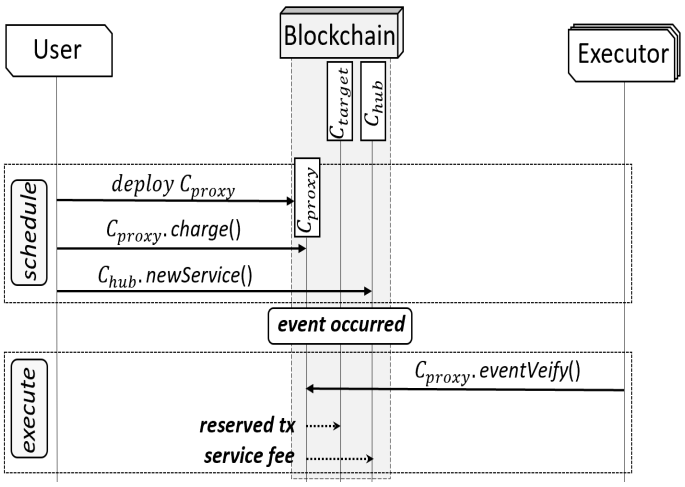


Fig. 2: Protocol sketch

summarize the notations that will be used in this section and in the rest of this paper in TABLE I.

### A. ET as a two-phase process

We describe the *ET* problem as a two-phase process in the context of the proposed proxy-contract-driven architecture:

- *ET.schedule*: The user $U$ creates a proxy contract $C_{proxy}$ with a balance (if needed) and sends her service request to a service hub contract $C_{hub}$, which notifies the $C_{proxy}$ to potential executors $Es$.
- *ET.execute*: Upon detecting the occurrence of the event specified in $C_{proxy}$, executor $E$ can invoke $C_{proxy}$ with a function invocation transaction to trigger the release of the reserved transaction in the form of a message.

### B. The protocol

We now present the protocol in detail. We sketch the protocol in Fig. 2 and present the formal description in Fig. 3. Concretely, the protocol employs a service hub contract ($C_{hub}$) to manage all $ET$ service requests, so executors only need to track events released from $C_{hub}$ to get notified about new $ET$ service requests. We next describe the two-phase process in detail.

***ET.schedule***: User $U$ first deploys a proxy contract $C_{proxy}$. User $U$ then needs to send an amount of ether via function $charge()$ to $C_{proxy}$. The amount of ether received by $C_{proxy}$ should be the sum of service fee for paying the executor and *et-ether* (if not empty). After that, user $U$ sets up a new *ET* service with $C_{hub}$ via function $newService()$ and specifies the service details, namely the address (i.e., $addr(*)$) of the proxy contract $C_{proxy}$. Upon getting notified by $C_{hub}$ about this new service request, executors $Es$ can start monitoring the occurrence of the event specified in the corresponding $C_{proxy}$.

***ET.execute***: Upon detecting the occurrence of the event specified in $C_{proxy}$, any executor $E$ is capable of constructing $blockNum$, $blockData$ and $MerkleProof$ locally and call function $eventVerify()$ inside $C_{proxy}$ with a function invocation transaction. Upon getting invoked, function $eventVerify()$ follows Algorithm 1 to verify the occurrence of the specified event. Function $eventVerify()$ then call a private $msgRelease()$ function to release the prescribed message and then transfer the service fee to the executor.

**ET.schedule:**

1. User $U$ deploys a proxy contract $C_{proxy}$.
2. User $U \Rightarrow C_{proxy}$: $charge(ether)$.
3. User $U \Rightarrow C_{hub}$: $newService(addr(C_{proxy}))$.

**ET.execute:**

4. Any executor $E \Rightarrow C_{proxy}.eventVerify(blockNum, blockData, MerkleProof)$.

Fig. 3: Formal protocol

## VI. Security analysis

In this section, we present the security analysis for the proposed `EventWarden`.

**Lemma 1.** *The reserved transaction can only get released from the proxy contract when the prescribed event has taken place.*

*Proof.* As illustrated in Algorithm 1, upon getting invoked by a function invocation transaction, the logic inside the proxy contract, namely the $eventVerify()$ function would verify the occurrence of the prescribed event before releasing the reserved transaction. Concretely, verifying the occurrence of the prescribed event is equivalent to verifying the existence of the log for storing the event in the blockchain. As presented in Section III-D, the reliability of the verification is endorsed by the reliability of blockchain. Together, as long as we could trust the blockchain, we could also trust the correctness of the verification done in the proxy contract. □

**Lemma 2.** *As long as a single member in Ethereum is incentivized by the service fee to call the proxy contract after the prescribed event has taken place, the reserved transaction is guaranteed to get released.*

*Proof.* As we have discussed in Section IV-C, the proposed proxy-contract-driven architecture isolates *et-executor* from *et-data*, *et-ether* and *et-logic*. Concretely, `EventWarden` outlines no eligibility requirements for becoming executors. Any EOA in Ethereum are eligible for providing services for `EventWarden`. This is because that the vital components, namely *et-data*, *et-ether* and *et-logic* are maintained by the trustworthy proxy contract, so the need for verifying the qualification of executors is minimized. Thanks to this strategy, every member from the huge Ethereum community becomes a potential service provider for `EventWarden` and the solid underlying community could make `EventWarden` easier to get launched in practice. Moreover, anyone in Ethereum is capable of constructing $blockNum$, $blockData$ and $MerkleProof$ locally and calling the $eventVerify()$

```
1  pragma solidity ^0.5.0;
2  import "RLPReader.sol";
3
4  contract C_proxy {
5    ...
6    using RLPReader for RLPReader.RLPItem;
7    using RLPReader for bytes;
8    function eventVerify(
9          bytes memory _rlpBlockData, uint _blockNum,
10         bytes memory _rlpMerkleRoot, uint _idMerkleRoot,
11         bytes memory _rlpMerkleBranch, uint _idMerkleBranch,
12         bytes memory _rlpMerkleLeaf
13         ) public {
14     // event verify
15     require(keccak256(_rlpBlockData) ==
16         blockhash(_blockNum));
17     RLPReader.RLPItem[] memory lsBlockHeader =
18         _rlpBlockData.toRlpItem().toList();
19     bytes32 receiptsRoot =
20         bytes32(lsBlockHeader[5].toUint());
21     require(keccak256(_rlpMerkleRoot) == receiptsRoot);
22     RLPReader.RLPItem[] memory lsMerkleRoot =
23         _rlpMerkleRoot.toRlpItem().toList();
24     bytes32 merkleBranch =
25         bytes32(lsMerkleRoot[_idMerkleRoot].toUint());
26     require(keccak256(_rlpMerkleBranch) == merkleBranch);
27     RLPReader.RLPItem[] memory lsMerkleBranch =
28         _rlpMerkleBranch.toRlpItem().toList();
29     bytes32 merkleLeaf =
30         bytes32(lsMerkleBranch[_idMerkleBranch].toUint());
31     require(keccak256(_rlpMerkleLeaf) == merkleLeaf);
32     RLPReader.RLPItem[] memory lsMerkleLeaf =
33         _rlpMerkleLeaf.toRlpItem().toList();
34     bytes memory rlpReceipt = lsMerkleLeaf[1].toBytes();
35     RLPReader.RLPItem[] memory lsReceipt =
36         rlpReceipt.toRlpItem().toList();
37     bytes memory rlpLog = lsReceipt[3].toBytes();
38     require(keccak256(rlpLog) == specifiedEvent);
39     // message release
40     msgRelease();
41     msg.sender.transfer(service_fee);
42   }
43   ...
44 }
```

Function 1: The *eventVerify()* function

```
1  contract C_proxy {
2    ...
3    address payable _recipient = ...;
4    uint _value = ...;
5    function fundTransfer() private {
6      _recipient.transfer(_value);
7    }
8
9    address _recipient = ...;
10   string _function_selector = ...;
11   uint _arg1;
12   uint _arg2;
13   function functionInvocation() private {
14     _recipient.call(abi.encodeWithSignature(
15         _function_selector, _arg1, _arg2));
16   }
17
18   bytes _bytecode = ...;
19   function contractCreation() private {
20     address _deployedAddr;
21     bytes memory _bc = _bytecode;
22     assembly {
23       _deployedAddr := create(0, add(_bc, 0x20), mload(_bc))
24     }
25   }
26   ...
27 }
```

Function 2: The *msgRelease()* function

function to complete the service, so the service is completely decentralized. This fact also facilitates the reliability of the service because it is impracticable for anyone to intercept all function invocation transactions sent by different EOAs from different places in the world for calling $eventVerify()$. □

## VII. Implementation and Evaluation

In this section, we present the implementation and evaluation for `EventWarden` in detail.

### A. Implementation

We programmed both the proxy contract and hub contract using the contract-oriented programming language *Solidity* [23] and we tested the contracts over the Ethereum official test network *Rinkeby* [24]. We employed the *solidity-rlp* library [25] for decoding data encoded with the Ethereum RLP (Recursive Length Prefix) rules [3] in smart contracts. Next, we present our detailed implementation for the two most challenging functions, namely *eventVerify()* and *msgRelease()*, respectively.

The implementation for function *eventVerify()* in the proxy contract $C_{proxy}$ is shown as Function 1. The contract '*RLPReader.sol*' is imported (line 2,6-7) from the *solidity-rlp* library for decoding encoded data input to the *eventVerify()* function. The functions *verifyBlock()*, *verifyProof()* and *verifyLog()* specified in Algorithm 1 are implemented at line 15-20, line 21-34 and line 35-37 in Function 1, respectively. Specifically, *verifyBlock()* is implemented by first fetching the hash of block specified by the input $\_blockNum$ from the blockchain and verifying that the input $\_rlpBlockData$ has the same hash value (line 15-16), then decoding the input $\_rlpBlockData$ as a list of components (line 17-18) and finally picking out the value for $receiptsRoot$ from the list (line 19-20). After that, *verifyProof()* is implemented by first verifying the input $\_rlpMerkleRoot$ and decoding $\_rlpMerkleRoot$ to get the hashed value for the branch node on the path of Merkle proof, namely $merkleBranch$ (line 21-25), then verifying the input $\_rlpMerkleBranch$ and decoding it to get the hashed value for the leaf node $merkleLeaf$ (line 26-30) and finally verifying the input $\_rlpMerkleLeaf$ and decoding it to get the data for the declared transaction receipt $rlpReceipt$ (line 31-34). In the third step, *verifyLog()* is implemented by decoding $rlpReceipt$ to get the data for the declared log $rlpLog$ (line 35-37). Finally, after verifying the equivalence between the log and the specified event (line 38), $msgRelease()$ is invoked to release the reserved transaction (line 40) and the service fee is sent to the executor (line 41).

The function *msgRelease()* could be implemented as one of three functions shown in Function 2, which depends on the type of reserved transaction. Specifically, *msgRelease()* for releasing a reserved fund transfer transaction, a reserved function invocation transaction or a reserved contract creation transaction could be implemented as line 3-7, line 9-16 or line 18-25 in Function 2, respectively.

| Phase | Step | Function | Gas | UDS |
|---|---|---|---|---|
| **ET.schedule** | 1 | deploy $C_{proxy}$ | 889764 | \$2.60 |
| | 2 | $charge()$ | 21497 | \$0.06 |
| | 3 | $newService()$ | 45612 | \$0.13 |
| **ET.execute** | 4 | $eventVerify()$ | 175674 | \$0.51 |
| **Other** | | $close()$ | 13662 | \$0.04 |

TABLE II: Key functions and their cost in Gas and USD.

### B. Evaluation

Similar to recent work on blockchain-based platforms and protocols [26], [27], the key focus of our evaluation is on measuring gas consumption, namely the amount of transaction fees spent in the protocol. This is due to the fact that the execution complexity in Ethereum is measured via gas consumption. It is worth noting that, in Ethereum, the amount of gas that a single transaction may spend is bounded by a system parameter and hence, the time overhead of executing functions inside smart contracts is small, usually in the scale of hundreds of milliseconds.

In TABLE II, we list the key functions in the programmed smart contracts that interact with protocol participants during different phases of the protocol and the cost of these functions in both Gas and USD. The cost in USD was computed through $cost(USD) = cost(Gas) * GasToEther * EtherToUSD$, where $GasToEther$ and $EtherToUSD$ were taken as their mean value during the first half of the year 2019 recorded in *Etherscan* [28], which are $1.67 * 10^{-8}$ Ether/Gas and 175 USD/Ether, respectively. As illustrated by the results, in `EventWarden`, the completion of a service only requires a user to deploy the proxy contract $C_{proxy}$ (\$2.60), transfer the required amount of ether to $C_{proxy}$ via function $charge()$ (\$0.06) and set up a new service at the hub contract $C_{hub}$ via function $newService()$ (\$0.20) during *ET.schedule* and an executor to invoke *eventVerify()* (\$0.51) during *ET.execute*, which costs only \$3.30 in total. The user may also choose to call function $close()$ to shut down the service and get back her ether from the proxy contract, which costs \$0.04.

In summary, our implementation and evaluation demonstrate that `EventWarden` is effective and is ready-to-use.

## VIII. Conclusion

This paper proposes `EventWarden`, a decentralized event-driven proxy service for users to outsource any type of transaction in Ethereum-like blockchains. `EventWarden` employs a novel combination of smart contracts and blockchain logs. `EventWarden` allows a user to create a proxy smart contract that specifies an interested event and also reserves an arbitrary transaction to release. Upon observing the occurrence of the prescribed event, anyone in the Blockchain network can call the proxy contract to earn the service fee reserved in the contract by proving to the contract that the event has been recorded into blockchain logs, which then automatically triggers the proxy contract to release the reserved transaction. We show that the reserved transaction can only get released from the proxy contract when the prescribed event has taken place. We also demonstrate that as long as a single member in the Blockchain network is incentivized by the service fee to call the proxy contract after the prescribed event has taken place, the reserved transaction is guaranteed to get released. We implement `EventWarden` over the Ethereum official test network. The results demonstrate that `EventWarden` is effective and is ready to be used in practice.


## Acknowledgement

This paper was supported by the Fundamental Research Funds for the Central Universities (Grant No. 2019RC038).


## Reference


[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, p. 37, 2014.

[3] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014.

[4] "Cryptokitties," https://www.cryptokitties.co/.

[5] "Writing a sealed-bid auction contract," https://programtheblockchain.com/posts/2018/03/27/writing-a-sealed-bid-auction-contract/.

[6] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi, "Fastkitten: practical smart contracts on bitcoin," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 801–818.

[7] P. McCorry, S. Bakshi, I. Bentov, S. Meiklejohn, and A. Miller, "Pisa: Arbitration outsourcing for state channels," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 16–30.

[8] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," *White paper*, pp. 1–47, 2017.

[9] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1353–1370.

[10] "Blueorion," https://blueorion.cc/.

[11] "Oraclize," http://www.oraclize.it/.

[12] J. Chen, S. Yao, Q. Yuan, K. He, S. Ji, and R. Du, "Certchain: Public and efficient certificate audit based on blockchain for tls connections," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 2060–2068.

[13] S. Hu, C. Cai, Q. Wang, C. Wang, X. Luo, and K. Ren, "Searching an encrypted cloud meets blockchain: A decentralized, reliable and fair realization," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 792–800.

[14] "Ethereum alarm clock," https://www.ethereum-alarm-clock.com/.

[15] J. Ning, H. Dang, R. Hou, and E.-C. Chang, "Keeping time-release secrets through smart contracts." *IACR Cryptology ePrint Archive*, vol. 2018, p. 1166, 2018.

[16] C. Li and B. Palanisamy, "Decentralized privacy-preserving timed execution in blockchain-based smart contract platforms," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 2018, pp. 265–274.

[17] G. K. Feridun Mert Celebi, Paul Fletcher-Hill and D. Que, "Kimono:trustless secret sharing using time-locks on ethereum," https://github.com/hillstreetlabs/kimono, 2018.

[18] C. Li and B. Palanisamy, "Silentdelivery: Practical timed-delivery of private information using smart contracts," *arXiv preprint cs/0101027*, 2019.

[19] A. M. Antonopoulos, *Mastering Bitcoin: Programming the open blockchain*. " O'Reilly Media, Inc.", 2017.

[20] "Parity," https://www.parity.io/ethereum/.

[21] "Stellar," https://www.stellar.org/.

[22] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[23] "The solidity contract-oriented programming language," https://github.com/ethereum/solidity.

[24] "Rinkeby: Ethereum official testnet," https://www.rinkeby.io/#stats.

[25] "Solidity-rlp," https://github.com/hamdiallam/solidity-rlp.

[26] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 327–344.

[27] S. Das, V. J. Ribeiro, and A. Anand, "Yoda: Enabling computationally intensive contracts on blockchains with byzantine and selfish nodes," *NDSS*, 2019.

[28] "Etherscan: gas price," https://etherscan.io/chart/gasprice.