Unifying Lightweight Blockchain Client Implementations

Damian Gruber
NEC Laboratories Europe
damian.gruber@neclab.eu

Wenting Li
NEC Laboratories Europe
wenting.li@neclab.eu

Ghassan Karame
NEC Laboratories Europe
ghassan.karame@neclab.eu

***Abstract*—Lightweight clients are gaining increasing adoption in existing blockchain deployments, owing to their reduced resource consumption. There are currently a number of libraries that implement lightweight clients (e.g., BIP37, Electrum, LES, filter commitments). Notice that these libraries are intrinsically different and require significant effort to be integrated across blockchain platforms. Additionally, lightweight clients require the cooperation of full nodes, which are expected to invest in their computational (to run filters) and bandwidth resources in order to serve lightweight clients. Existing blockchains however offer no rewards for full nodes in exchange—which offers little incentives for full nodes to correctly serve lightweight clients.**

In this paper, we shed light on this problem and we show that smart contracts provide a natural and fair environment to deploy and provision filters for lightweight clients. Namely, we propose a scheme, SmartLight, that enables the integration of filters/libraries within smart contracts to support a wide range of lightweight client instantiations. We show that SmartLight can integrate payment routines to reward full nodes to serve lightweight clients. SmartLight can be integrated without modifications in existing blockchains that support smart contracts.

I. Introduction

The massive success of Bitcoin has unveiled a truly genuine breakthrough: the *blockchain*. The blockchain allows transactions, and any other data, to be securely stored and verified without the need for any centralized authority and while scaling to a large number of nodes. As such, the blockchain has fueled innovation in the last couple of years, and a number of innovative applications have already been devised by exploiting the secure and distributed provisions of the blockchain. Examples include Ethereum [4], Hyperledger [6], Ripple [9], and R3 [8], among others.

Most existing blockchains require considerable storage and computing resources. For instance, a typical Ethereum installation requires more than 30 GB of disk space, and requires considerable time to download and locally index blocks and transactions that are contained in the blockchain. In addition to space usage, users need to verify the correctness of broadcasted blocks and transactions in the network.

To enable the use of resource-constrained devices within the blockchain, most platforms support a *lightweight* mode of operation, where lightweight clients only need to download and process a small part of the blockchain.

Workshop on Decentralized IoT Security and Standards (DISS) 2018
18 February 2018, San Diego, CA, USA
ISBN 1-891562-51-7
https://dx.doi.org/10.14722/diss.2018.23010
www.ndss-symposium.org

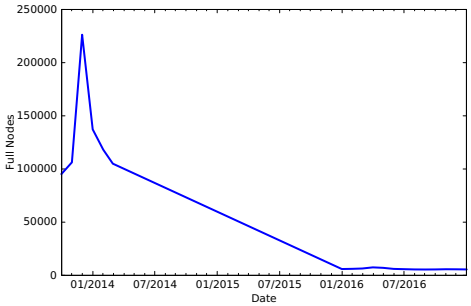

Fig. 1: Number of reachable full Bitcoin nodes between 01/2014 and 01/2017 [17], [18].

Currently, a number of libraries support various filters in order to enhance clients' privacy (e.g., Ethereum's LES). Those filters can be defined with a target false-positive rate in an attempt to hide the addresses of lightweight clients. Clients have to manually download these libraries and do not have any flexibility in choosing their filter types. For instance, in Ethereum, the LES library does not employ any filter, and does therefore not offer any address privacy for lightweight clients. In Bitcoin, only Bloom filters are supported. Notice that the literature includes a number of alternative filters such as speed-optimized Bloom filters [24], and Cuckoo filters [22].

In existing deployments, full nodes are required to filter, process, and forward transactions corresponding to each client without any reward/renumeration. This offers little (if any) incentives for full nodes to support lightweight clients. We argue that this is one of the reasons to explain the sharp decrease in the numbers of full nodes in the network (cf. Figure 1). Given that the interaction between lightweight and full nodes is not recorded in the blockchain, it is also very challenging to estimate the number of lightweight clients in the system and to evaluate the quality of service that they witness.

In this paper, we shed light on this problem and we propose a solution, SmartLight, that enables the integration of filters/libraries within smart contracts to support a wide range of lightweight client instantiations. SmartLight contracts are constructed by lightweight clients and act as a broker service between these clients and the full nodes. SmartLight enables lightweight clients to rely on arbitrary filters and enables a fair rewarding scheme to correct full nodes in exchange of their service. We analyze the provisions of SmartLight, and evaluate its feasibility and performance within the Ethereum framework.

II. Lightweight Clients

Blockchains maintain an ever-growing ledger comprising of blocks and transactions. We distinguish three different roles in existing systems: *miners* or validators are nodes that participate

	Protocol/Library	Support for filters
Bitcoin	BIP 37	Supports Bloom filters for privacy
	Download everything	Results in perfect privacy
	Electrum	Reveal addresses, get UTXOs
	Filter commitments	Lightweight clients are presented with filters
Litecoin [7]	BIP37	Supports Bloom filters for privacy
Dogecoin [2]	BIP37	Supports Bloom filters for privacy
Ethereum	LES	No privacy

TABLE I: Overview of libraries for lightweight clients.

in the consensus routine to confirm blocks/transactions. Miners are typically rewarded for participating in the validation process (e.g., by receiving coinbase transactions [26]). *Full nodes* are blockchain clients which maintain a complete copy of the blockchain. These nodes do not have to participate in the consensus (i.e., they do not mine) but verify the correctness of each received block/transaction in the network and help disseminating correct information within the blockchain. In contrast to full nodes, *lightweight clients* do not download and process the complete blockchain. Instead, they connect to full nodes which only forward the transactions they request. Notice that full nodes are not incentivized for this service. As such, *full nodes only have little incentives to (correctly) serve lightweight clients.* This could explain the reason why there are currently millions of lightweight client installation, and only 6,000 full nodes that are reachable in the Bitcoin system [18].

Current blockchains offer a number of libraries to support lightweight clients. Arguably, Bitcoin's *BIP37* [11] emerges as the most widely used protocol. BIP37 allows lightweight clients to insert their addresses into Bloom filters [19], which are then outsourced to full nodes. Full nodes filter transactions against the provided Bloom filters, and only forward matching transactions to lightweight clients. Full nodes additionally compute a compact proof, which allows lightweight clients to verify the inclusion of relayed transactions within blocks. This process is referred to as Simplified Payment Verification (SPV) [26]. Since those filters correspond to clients, full nodes need to process each transaction for each connected lightweight client. Because Bloom filters only allow for approximate membership queries, full nodes cannot directly learn the addresses that a lightweight client inserted into a filter.

Electrum [3] adopts a significantly different approach. Electrum lightweight clients rely on dedicated Electrum servers, which index the blockchain. Electrum lightweight clients can query those servers for *unspent transaction outputs* (UTXOs) given their addresses. Here, Electrum servers learn the clients' addresses and their corresponding transactions and balances—resulting in a lack of privacy towards those servers. Filter commitments [16] emerge as another workable option to support lightweight clients in Bitcoin. In this approach, lightweight clients receive a filter of all transactions/addresses included in a given block; lightweight clients then request the block if the filter matches their addresses/transactions.

Ethereum [4] is expected to rely on the Light Ethereum Subroutine (LES) [29] which allows lightweight clients to query for their balances. Table I summarizes the provisions of libraries for lightweight clients in a number of blockchain technologies.

Notice that each of these schemes (and possible instantiations) requires a different protocol/library, which has to be implemented and maintained by developers, and ultimately downloaded and installed by lightweight clients. Owing to this cumbersome process, blockchain systems typically do not provide multiple lightweight client schemes. For instance, apart from Bloom filters, one can rely on speed-optimized Bloom filters [24], which reduce the amount of hash function evaluations required, or using Cuckoo filters [22] which have been proposed as a replacement for Bloom filters. Alternatively, one can make use of Prefix filters [27] which only store strict prefixes of elements. However, modifying the BIP37 protocol to support any other filter in addition to Bloom filters entails considerable implementation work.

Bloom and Cuckoo Filters: Bloom filters are equipped with a set of k hash functions, which map into their bit array. For inserting an element, the k hash functions are evaluated, and the corresponding bits are set. To verify membership of an element, the corresponding k positions are inspected. Notice that optimized Bloom filters only differ in how they compute the hash functions. On the other hand, Cuckoo filters maintain an array of *buckets*, where each bucket can store up to 4 *fingerprints*. An element is inserted into a Cuckoo filter by computing its fingerprint and inserting this fingerprint into any of two possible buckets. If both buckets are full, an element is evicted and has to be relocated. If this recursive procedure does not terminate within a pre-defined number of steps, the filter is considered full. The corresponding membership verification procedure computes the fingerprint of the given element, and searches both possible buckets for this fingerprint.

III. SmartLight

In this section, we present our solution, SmartLight. Before describing our solution in detail, we start by outlining the main intuition behind SmartLight.

A. System and Threat Model

We consider an open (i.e, public) blockchain platform that supports Turing-complete scripting language to describe a smart contract (e.g., Ethereum).

We assume that a lightweight client $\mathcal{C}$ wishes to join the blockchain network without contributing in the consensus process. More specifically, the client is interested in receiving a subset of transactions without investing any resources to receive or validate other transactions. For that purpose, $\mathcal{C}$ is interested in connecting to a number T of full nodes and ask them to forward his transactions of interest. This is practically achieved in existing lightweight client implementations by outsourcing a filter $\mathcal{F}$ which defines the condition upon which a transaction should be forwarded to $\mathcal{C}$. An efficient filter requires compact storage space and quick lookup time. As discussed earlier, a number of filters enforce false-positives in order to conceal the actual transactions that $\mathcal{C}$ is interested in receiving. To incentivize full nodes to perform this service, we assume that $\mathcal{C}$ is willing to offer a small fee for up to T full nodes that forward correct transactions matching $\mathcal{C}$'s filter.

We assume that both the lightweight clients and the full nodes are rational entities, e.g., see [13] for a similar assumption. By rational, we mean that these entities will only deviate from the protocol if such a strategy increases their profit in the system. For instance, lightweight clients are interested in receiving all the transactions that match their filters while paying the minimum amount of fees in the network. Likewise, full nodes are interested in acquiring fees from lightweight clients e.g., while minimizing their invested computational resource (i.e., without executing their filters), or by optimizing their bandwidth (i.e., by selectively withholding the transmission of transactions). Moreover, nodes might be

interested to profile lightweight clients/full nodes [12] by associating IP addresses with blockchain account addresses.

We assume that malicious nodes are computationally bounded and cannot control consensus in the network. For example, in a Proof of Work-based consensus, we assume that these nodes cannot control more than 33% of the computing power in the network [23]. However, a resource-constrained adversary can attempt to "eclipse" lightweight clients by compromising all the full nodes that they connect to. By doing so, the adversary is able to control the view of any given lightweight client, selectively withhold the delivery of transactions and/or blocks [23].

B. Intuition & Overview

SmartLight builds upon smart contracts in the blockchain in order to offer flexible, fair, and transparent support for lightweight clients.

Recall that smart contracts refer to self-contained code that is executed by all blockchain nodes. For example, Ethereum [28] is a decentralized platform that enables the execution of arbitrary applications (or contracts) on its blockchain. Owing to its support for a Turing-complete language, Ethereum offers an easy means for developers to deploy their distributed applications in the form of smart contracts. Interestingly, lightweight clients rely on specific library implementations, and do not make use of smart contracts. Namely, lightweight client support is currently implemented outside the smart contracts paradigm.

In contrast to existing implementations of lightweight clients, SmartLight allows users to invoke specially-crafted smart contracts that orchestrate the interaction between lightweight clients and the full nodes. By leveraging smart contracts, SmartLight provides clients with full flexibility in terms of the number of full nodes to contact, the lightweight client protocol/library, and even the employed filter type. By doing so, SmartLight can be integrated without modifications in existing blockchains that support smart contracts.

If a lightweight client wishes support from full nodes, it first implements a SmartLight contract. Such a contract can be deployed in the blockchain by sending a transaction comprising SmartLight's payload. The payload consists of three routines: an assignment routine, a filtering routine, and a payment routine.

Clearly, a solution that requires the outsourcing of transaction filtering through smart contracts does not scale well with the number of blockchain nodes if each node is expected to execute all such contracts. This is why SmartLight relies on an assignment routine, which restricts the set of full nodes that are expected to execute the contract. For example, clients can specify a set of whitelisted server account addresses or a random prefix of account addresses. Full nodes that match the assignment condition will establish a connection with the client later on to provide filtering service. We show that malicious full nodes cannot influence/abuse the assignment routine in SmartLight to increase their advantage in the network.

In the filtering routine, SmartLight defines which transactions the corresponding lightweight client is interested in. This routine could comprise a filter, which accumulates the client's blockchain addresses while offering some privacy guarantees (through false positives). Full nodes compare all incoming transactions against the filtering routine and check if any of the included addresses match the filter. This can be instantiated with any space-efficient filter structure. Notice that SmartLight can also be instantiated without any filter, or using filters that feature 0% false-positives.

SmartLight further incorporates a routine to pay/reward those full nodes (i.e., that satisfy the assignment routine) which correctly filter the lightweight clients' transactions. This offers incentives for full nodes to execute only those contracts to which they are assigned. Notice that this does not prevent other full nodes from running the contract and forwarding transactions to lightweight clients; this behavior will not be however rewarded by the SmartLight contract.

Notice that full nodes can attempt to claim rewards independently of whether they correctly executed the contract. SmartLight prevents that misbehavior by allowing lightweight clients to pay in exchange for every transaction that the full nodes forward. To minimize communication overhead, rewards for transactions are aggregated in batches. Alternatively, SmartLight can incorporate payment routines in order to ensure fair rewards to full nodes. By doing so, SmartLight does not only incentivizes full nodes to *correctly* filter all lightweight clients' transactions, but also disincentivizes lightweight clients from "free-riding" in the blockchain by embedding the payment routines in the contract.

We show that SmartLight achieves these properties without compromising the privacy of lightweight clients and/or full nodes. Namely, SmartLight ensures that network layer information (such as IP addresses) are only visible to those nodes which share a given contract for forwarding transactions.

C. Protocol Specification

Recall that, in SmartLight, a lightweight client deploys a contract in the blockchain which handles its interaction with full nodes. This contract consists of three routines: an assignment routine, a filtering routine, and a payment routine.

Assignment Routine: The assignment routine restricts the set of full nodes that are expected to execute the rest of the contract (and to filter transactions for the lightweight client). Clearly, it is a desirable goal to prevent full nodes from influencing the assignment process (e.g., to prevent Eclipse attacks on clients).

One plausible assignment approach is based on the combined use of account and IP addresses (cf. Algorithm 1). Here, a full node is assigned to a lightweight client if the last n bits of its active account address match the corresponding bits set by the client in the contract (see `isAssigned()`). Since we assume an open Blockchain system, nodes can create different accounts on the fly (i.e., Sybil attack [21]) to enforce their assignment to a particular contract. This is exactly why the assignment routine relies on IP addresses as a unique identifier of nodes; such an approach ensures that any given IP address will get a single assignment—irrespective of the number of accounts held by the node. To maintain backwards compatibility with Bitcoinj [1], SmartLight also incorporates a routine for whitelisting; a full node can be allowed to serve a client if it has been whitelisted by the client (see `isAllowed()`).

Once a full node whose account satisfies the assignment condition agrees to provide the filtering service, it will reply to the SmartLight contract with a transaction `agree()`. This transaction includes its IP address encrypted by the client's public key, its account address, as well as a signature under

the claimed account.[1] As we show later, this approach prevents nodes to associate a given IP address to an account address; the IP address can only be decrypted by the lightweight client. Given the IP address, the lightweight client initiates the connection to assigned nodes using an off-chain channel to receive filtered transactions. We stress at this point that the IP address of the lightweight client will only be visible to full reachable nodes that match the assignment routine. Notice that full nodes that are located behind a NAT (Network Address Translation) can also provide reachable IP/port combination addresses to the client in a similar way.

Filtering Routine: This routine defines which transactions a full node should forward to the client. Specifically, it implements a `contains()` predicate, which takes addresses as arguments. Full nodes extract all addresses from a received transaction and verify their membership using this procedure by invoking the contract in local execution environment such as EVM (Ethereum Virtual Machine). If any address in a transaction matches, the transaction will be forwarded to the client. Notice that there are different possibilities to implement the membership verification procedure. For instance, this can be realized through Bloom filters, optimized Bloom filters, Cuckoo filters, or Prefix filters (or no filters at all). In Algorithm 2, we show how to integrate Bloom filters within SmartLight.

Payment Routine: This routine specifies the rewarding mechanism in exchange for the service of the full nodes. Such a mechanism should achieve two goals. First, full nodes should be able to claim rewards for their services (i.e., for filtering the blockchain). Second, lightweight clients should only pay in exchange for correct services.

One possible way to achieve this would be that lightweight nodes reward each transaction that matches the membership test, which is forwarded by the full nodes. Algorithm 1 implements such an approach. Here, the contract maintains the client's `balance` and allows the client to increase this balance (through function `addtoBalance()`). The contract further specifies a per-transaction `reward`, which determines the amount of wei (10^{-18} ether) that a full node receives per correctly relayed transaction. Clearly, the reward amount depends on the filter type and the requested work. The higher is the amount of execution effort to filter transactions, the higher is the reward amount. Full nodes that do not agree with the reward amount simply do not run the contract. This offers considerable incentives for lightweight clients to set a reasonable reward amount.

In this approach, whenever they receive a transaction from full nodes, lightweight clients send back a signed commitment which consists of a tuple $ack = \langle sequence', Account \rangle$ if the transaction matches the given filter and the SPV proof of the transaction is correct. Here, $sequence'$ denotes the current total number of received transactions, and $Account$ refers to the full node's account address. At any point in time, full nodes can present ack to the contract to trigger the `claim()` procedure. The latter checks if ack is well-formed; if so, it issues a reward for $sequence' - sequence$ transactions where $sequence$ is the latest (highest) sequence number that the full node $Account$ has claimed from SmartLight. Consider the case where a full node has forwarded N transactions to the client and received N acknowledgments (i.e, $ack_1 ... ack_N$). To claim the reward for the service of these N transactions, the full node submits a `claim()` transaction that includes the last acknowledgment ack_N. Similarly, to reward the next N forwarded transactions, full nodes only need to submit `claim(`ack_{2N}`)`.

Notice that this approach ensures that full nodes are rewarded for every correct transaction that they forward. Whenever the client fails to send ack for the received transaction, the full nodes stop providing the filtering service. Moreover, this approach allows full nodes claim the reward in batch—thus reducing transaction fees as well as transaction storage overhead. One can also envision the reliance of fair exchange protocols [14], [25] within SmartLight to achieve fairness guarantees.

[1] Recall that in Ethereum, the account address and the signature are part of the standard protocol specification to validate the authenticity of the transaction originator. Therefore, the smart contract just needs to verify sender's account address given the assignment condition and save the encrypted IP address to the ledger.

Algorithm 1: SmartLight

Input : $accountPrefix$; `// Prefix of the accounts as assignment requirement`
$accountWL$; `// White list of accounts`
$balance$; `// Initially zero`
$servers$; `// Initially empty list`
n; `// # matching bits for assignment`
$lastSeqs$; `// Mapping: Accounts to sequence numbers. Initially empty`
$pubKey$; `// Client's public key`
r; `// Per-message reward in wei`

Function *addtoBalance()*
  $balance$ += $tx.value$;

Function *isAssigned(Account)*
  if $Account \in accountWL$ **then**
    return *true*;
  $n' \leftarrow$ Number of trailing bits shared in $Account$ and $accountPrefix$;
  return $n' \geq n$;

Function *agree(*$Enc_{pubKey}(IP)$*)*
  $Account \leftarrow tx.origin$;
  if *isAssigned(Account);*
  then
    $servers \leftarrow servers \cup \langle Account, Enc_{pubKey}(IP) \rangle$;
    $lastSeqs[Account] \leftarrow 0$;

Function *claim(ack, sig)*
  Check sig for ack, using $pubKey$;
  $\langle sequence', Account \rangle \leftarrow ack$;
  if $Account \in servers$;
  then
    $sequence \leftarrow lastSeqs[Account]$;
    $d \leftarrow sequence' - sequence$;
    if $d > 0$ && $balance \geq d*r$;
    then
      Send $d*r$ wei to $tx.origin$;
      $balance$ -= $d*r$;
      $seq[Account] \leftarrow sequence'$;

Function *contains(e)*

Algorithm 2: Sketch of a SmartLight implementation of a Bloom Filter.

Input : k; `// Number of hash functions`
$filter$; `// Pre-populated filter (bit array)`

Function *contains(e)*
  for *i = 0; i < k; i++* **do**
    if $filter[h_i(e)] \neq 1$ **then**
      return *false*;
  return *true*;

D. Security & Privacy Analysis

Security: We first consider a rational full node $\mathcal{A}_f$ that wants to increase its profits while minimizing his efforts of filtering transactions. Since $\mathcal{C}$ will verify each received transaction before providing an acknowledgement that can be used to claim the reward, $\mathcal{A}_f$ are incentivized to correctly execute the filter computation in order to forward correct transactions. In this respect, Sybil attacks constitute the only viable strategy for $\mathcal{A}_f$ to maximize its profit in the system. Namely, $\mathcal{A}_f$ can attempt to create several account addresses that all match the assignment routine in the hope of multiplying his revenues while filtering transactions only once. SmartLight deters this strategy by relying on IP address identifiers for nodes; this means that a rational adversary would have to serve on a number of IP addresses in order to create multiple identities in the network. More specifically, SmartLight prevents Sybil attacks based on IP address by letting the lightweight client to initiate the connection to the regular node. In this case, the adversary has to invest considerable resources (e.g., subscribe multiple Internet service contracts, contaminate WAN routers or run a Botnet) in order to obtain multiple identities. We therefore argue that SmartLight achieves similar security guarantees with respect to rational full nodes when compared to existing lightweight client implementations—who are also not immune against node compromise or Botnet attacks.

Notice that resource bounded full nodes cannot completely monopolize all the connections to lightweight clients since they cannot prevent other honest full nodes from committing to the contract and serving $\mathcal{C}$. Namely, as long as $\mathcal{C}$ is connected to at least one honest node, the adversary is not able to eclipse $\mathcal{C}$. Recall that $\mathcal{C}$ can also have a predefined white list of full nodes in the contract too.

With respect to security against rational lightweight clients, the payment routine in SmartLight guarantees that the clients pay for all the forwarded transactions. The payment routine operates in a "pre-paid" mode. The clients first have to charge the contract with sufficient credits, and the contract will execute the reward distribution to the serving nodes. Since the adversary cannot control the consensus of the network and assuming that the majority of the blockchain network is honest, the contract will follow the payment routine and the network will finally reward any full nodes who present a valid ack token. Since full nodes expect an authenticated acknowledgment for each forwarded transaction, these nodes stop serving the client if the client refuses to provide a valid acknowledgment. Therefore, given N forwarded transactions, SmartLight ensures that the regular nodes are credited for forwarding at least $N-1$ transactions—effectively capturing the worst case scenario where the last forwarded transaction is not acknowledged by the lightweight client (and the full node thus stops forwarding).

Privacy: We now show that SmartLight prevents curious nodes from associating IP addresses of full nodes/lightweight clients with their respective blockchain account addresses.

Recall that SmartLight relies on account address information to perform node assignment and makes use of IP address information to identify each full node in case a full node possesses multiple qualified account addresses. A full node whose account address matches the assignment submits an encrypted IP address that can only be decrypted by the lightweight client (that will use this info to establish the off-chain channel). If a full node is not assigned according to the contract, it cannot learn the IP address of the client; similarly, if a full node does not commit to the filtering service, the lightweight client is not able to learn any extra information about the full node. Moreover, to further prevent the connected full nodes from learning the IP addresses associated with their accounts, lightweight clients can establish an off-chain channel via anonymizing networks, such as TOR [20], to prevent the disclosure of their real IP addresses.

	Ether	USD
Deploy client contract (Bloom)	0.033	0.36
Deploy client contract (Cuckoo)	0.050	0.54
Claim reward	$\approx$ 0.001	0.01
Increase client balance	0.001	0.02

TABLE II: Evaluation in terms of entailed blockchain fees.

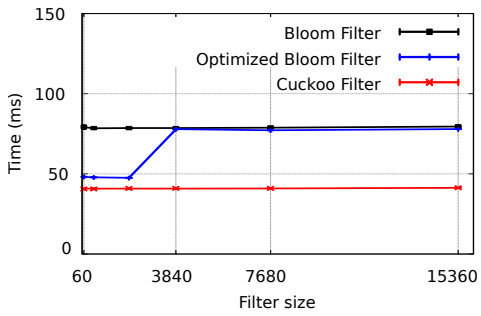

Fig. 2: Evaluation: Lookup time w.r.t. filter size. We present measurements for Bloom filters, optimized Bloom filters, and Cuckoo filters.

IV. Implementation & Evaluation

A. Methodology

In order to evaluate the feasibility of our proposal, we implemented SmartLight, instantiated with standard and optimized Bloom filters, as well as Cuckoo filters, respectively.

SmartLight' contracts were written in Solidity for the Ethereum blockchain. In our implementation, we relied on the Solidity realtime compiler [10], version 0.3.6. The experiments were conducted with `geth` [5], version 1.4.18. In order to translate gas into ether, we assume a price of 20 Gwei per unit of gas. We further assume a price of 10.89 USD per ether. In our evaluation for blockchain fees, the filters were initialized to represent 120 elements. Our experimental setup for measuring the runtime performance of the studied filters relies on an 8-core machine (Intel Xeon CPU E3-1230 V2, 3.30 GHz), has 16 GiB memory and an SSD. We implemented the filters in Java, and configured them with a target false-positive rate of 0.01%. For computing SPV proofs, we adapted the Bitcoinj [1] library, version 0.15-SNAPSHOT. The dataset underlying our runtime measurements consists of Bitcoin blocks 418.000 to 418.999, which have been mined between June 26th, and July 2nd, 2016. We measure the costs incurred by SmartLight in terms of entailed fees in the Ethereum blockchain. To this end, we analyze the amount of gas (Ethereum's unit for fees) consumed for deploying SmartLight on the Ethereum blockchain, and for interacting with the contract (i.e., costs resulting from sending transactions to the contract). We also measure the (monetary) costs a full node has to bear for executing the filtering routine, depending on the filter type. Specifically, we measure the time that a full node requires to perform membership verification of the addresses appearing in all investigated transactions and to compute the corresponding SPV proofs. Each data point in our plots is averaged over 10 independent runs; where appropriate, we also present the corresponding 95% confidence intervals.

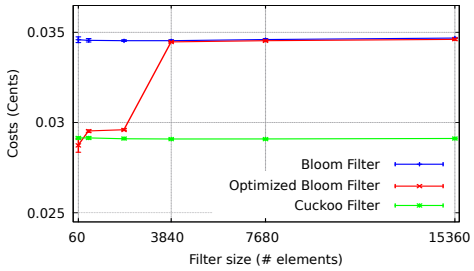

Fig. 3: Evaluation of costs in cents. We present measurements for Bloom filters, optimized Bloom filters, and Cuckoo filters, and we vary the number of elements inserted into the filters.

B. Evaluation

Contract deployment: Table II shows the costs in terms of fees that lightweight clients have to pay for deploying SmartLight contracts in Ethereum. We observe that SmartLight with Bloom filters (0.36 USD) result in approximately 50% lower blockchain fees, when compared to Cuckoo filters (0.54 USD) . This results from the fact that the code for Bloom filters is significantly smaller than that of Cuckoo filters. Notice that the fees incurred by deploying speed-optimized Bloom filters is almost equivalent to those for standard Bloom filters (the code only differs in a few lines).

Contract Execution Time: Figure 2 depicts the address membership verification time depending on the filter type with respect to the number of addresses inserted within the filter. Bloom filters require approximately 80 ms to verify membership—irrespective of the filter size. Speed-optimized Bloom filters are significantly faster for small filters since they only evaluate a single hash function, instead of two hash functions. Cuckoo filters allow for faster address membership verifications, when compared to both standard and optimized Bloom filters. Notice that, for Cuckoo filters, the membership verification time does not depend on the filter size.

Executing assignment routine: Assignment does not result in any blockchain fees since full nodes can execute the routine locally – they do not need to send a transaction.

Executing filtering routine: Figure 3 depicts the costs for performing membership verification of the addresses in the 1.000 blocks and to compute SPV proofs for matched transactions. We estimated the costs of executing the filters by adapting the costs of Amazon EC2 c3.2xlarge instances (0.248 USD per hour [15]). Notice that transaction filtering does not result in any blockchain fees. Again, full nodes can perform the computation locally. We observe that for small filters (i.e., filters that contain few elements), this process is cheaper with optimized Bloom filters (0,0295 cents), when compared to standard Bloom filters (0,035 cents). This is due to an additional performance optimization which allows optimized Bloom filters to evaluate a single hash function only. This is however not possible for larger filters, where two hash function evaluations are necessary. We further observe that Cuckoo filters (0,029 cents) outperform the other studied filters. We finally point out that these costs are orders of magnitude lower than the contract deployment costs.

Executing payment routine: Table II shows the costs associated with payment, namely for a client to increase its balance, and for a full node to claim a reward. In both cases, the required transactions cost approximately 1 cent. Our results suggest that Cuckoo filter contract deployment cost more gas because of their complexity, but incur less costs for transaction filtering.

V. Outlook

In this paper, we proposed SmartLight, a solution that enables the integration of filters/libraries within smart contracts to support a wide range of lightweight client instantiations. SmartLight can integrate rewarding mechanisms for full nodes to serve lightweight clients. Our findings suggest that SmartLight can be easily deployed without modifications within smart contracts and can be instantiated with a multitude of existing filters, such as Bloom or Cuckoo filters. SmartLight can additionally capture various other lightweight client implementations, such as committed Bloom filters, among others.

We argue that SmartLight motivates the unification of various existing lightweight client implementations under one umbrella and transparently orchestrates/regulates the interaction between lightweight clients and full nodes. SmartLight offers considerable incentives for full nodes to correctly serve lightweight clients by incorporating rewarding mechanisms. We therefore hope that our findings motivate further research in this area.

References

[1] "Bitcoinj – A library for working with Bitcoin," https://github.com/bitcoinj/bitcoinj/.

[2] "Dogecoin," http://dogecoin.com/.

[3] "Electrum Bitcoin Wallet," https://electrum.org/.

[4] "Ethereum – Homestead Release," https://www.ethereum.org/, accessed: 2016-11-25.

[5] "Ethereum Go client," https://github.com/ethereum/go-ethereum/wiki/geth.

[6] "Hyperledger – Blockchain Technologies for Business," https://www.hyperledger.org/.

[7] "Litecoin: Open source p2p digital currency," https://litecoin.org/.

[8] "R3," http://www.r3cev.com/.

[9] "Ripple," https://ripple.com/.

[10] "Solidity realtime compiler," https://ethereum.github.io/browser-solidity/.

[11] "Connection Bloom filtering," https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki, 2012.

[12] E. Androulaki, G. O. Karame, M. Roeschlin, T. Scherer, and S. Capkun, "Evaluating user privacy in bitcoin," in *International Conference on Financial Cryptography and Data Security*. Springer, 2013, pp. 34–51.

[13] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef, "Transparent data deduplication in the cloud," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 886–900.

[14] N. Asokan, M. Schunter, and M. Waidner, "Optimistic protocols for fair exchange," in *Proceedings of the 4th ACM conference on Computer and communications security*. ACM, 1997, pp. 7–17.

[15] A. W. S. (AWS), "EC2 Instance Pricing," https://aws.amazon.com/ec2/pricing/reserved-instances/pricing/, accessed: 2016-11-24.

[16] A. Back, "Bloom filtering, privacy," https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2015-February/007500.html, 2015.

[17] BitcoinPulse, "Dynamic monitoring of bitcoin," accessed: 2017-01-30.

[18] Bitnodes, "BGlobal Bitcoin Node Distribution," https://bitnodes.21.co/, accessed: 2017-02-15.

[19] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[20] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 21–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251375.1251396

[21] J. R. Douceur, "The sybil attack," in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 251–260.

[22] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 2014, pp. 75–88.

[23] A. Gervais, H. Ritzdorf, G. O. Karame, and S. Capkun, "Tampering with the delivery of blocks and transactions in bitcoin," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 692–705.

[24] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better bloom filter," in *European Symposium on Algorithms*. Springer, 2006, pp. 456–467.

[25] J. Liu, W. Li, G. O. Karame, and N. Asokan, "Towards fairness of cryptocurrency payments," *CoRR*, vol. abs/1609.07256, 2016. [Online]. Available: http://arxiv.org/abs/1609.07256

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

[27] P. Todd, "Privacy and blockchain data," https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2014-January/004019.html.

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

[29] F. Zsolt, "Light Ethereum Subprotocol (LES)," https://github.com/zsfelfoldi/go-ethereum/wiki/Light-Ethereum-Subprotocol-\%28LES\%29, accessed on 2016-12-22.