Privacy-Enhanced Content Discovery for Bitswap

Erik Daniel and Florian Tschorsch
Distributed Security Infrastructures, *Technische Universität Berlin*
{erik.daniel, florian.tschorsch}@tu-berlin.de

***Abstract*—IPFS is a content-addressed peer-to-peer data network, which follows the paradigm of information centric networking. In IPFS, data is exchanged with the Bitswap protocol. For content discovery, Bitswap queries all neighbors for the content, leaking the interest to all neighbors. In our paper, we develop three privacy-enhanced protocols for content discovery, which reduce the interest leak from all neighbors to ideally one content provider. Our protocols use probabilistic data structures like Bloom filter and cryptographic approaches like Private Set Intersection. We implement our protocols as proof of concept and show how they can be integrated into go-bitswap. Furthermore, we provide a performance, security, and privacy evaluation of the three protocols, showing their feasibility trade-offs.**

***Index Terms*—P2P Overlay Network, Information Centric Networking, IPFS, Privacy**

I. Introduction

Information Centric Networking (ICN) [1, 2] changes the way content is retrieved. Instead of the traditional location search for content retrieval, clients directly ask the network for the content. While this approach can speed up content retrieval, privacy concerns exist. The requests for content reveal the client's interest in a specific content to the whole network. This privacy problem is also inherited by content-addressed peer-to-peer (P2P) data networks [3], which basically construct an ICN as P2P overlay. One of these P2P data networks is the InterPlanetary Filesystem (IPFS) [4]. In IPFS, data is identified via a Content Identifier (CID) and exchanged via the Bitswap protocol. Bitswap requests all neighbors for the content via the CID. If content cannot be retrieved from a neighbor, content providers are searched in a Kademlia-based DHT.

In this paper, our primary goal is to reduce the information leakage during content discovery. We explore the design space of different approaches with a focus on protecting client privacy. To this end, we trade off leaking more information of content providers. Specifically, we propose to change the request of specific items (the interest) to a request of an inventory list of all stored items, which can be checked locally by the client to identify suitable content providers. Since a naive implementation of a content inventory would reveal a lot of information and could be highly inefficient, we improve the method by using Probabilistic Data Structures (PDS) and Private Set Intersection (PSI) [5]. PDS can be used for fast membership approximation in large data sets, which can increase the efficiency and introduce plausible deniability for content providers. PSI allows two parties to privately calculate the intersection of two sets, i.e., the overlap of interest and storage, further improving privacy of content providers.

ISBN 978-3-903176-57-7

As a result, we present different Bitswap alternatives: Bloom-Swap, PSI-Swap, and BEPSI-Swap. Each protocol provides different degrees of privacy. In Bloom-Swap, a Bloom filter [6] is used to approximate server items during provider discovery. PSI-Swap obfuscates the item using an elliptic curve Diffie-Hellman PSI protocol (ECDH-PSI) [7, 8, 9]. We chose a PSI protocol based on Diffie-Hellman, since they are easier to implement and faster for low set sizes [10]. The Bloom-filter Extended PSI-Swap (BEPSI-Swap) improves the efficiency of PSI-Swap at the cost of making PSI probabilistic. While our protocols cannot protect the data exchange, they increase content discovery privacy by reducing the leak of interest.

We adapt and implement our protocols in Bitswap as a Proof-of-Concept (PoC). To this end, our implementation maintains most of the Bitswap protocol and strives to be downwards compatible. Based on our PoC and our evaluation, we show the efficiency of the protocols. Due to a low communication overhead and a one-time overhead for content providers, the protocols are provider friendly. Most of the overhead is self-induced on the client side, which we deem acceptable considering the additional privacy gains. While all our protocols increase the privacy of clients, their feasibility trade-off differs. In addition to the privacy gains for clients, Bloom-Swap can reduce communication costs by trading in server privacy. The PSI-based protocols increase the privacy of all parties, including servers. The communication overhead, however, makes them more suitable for limited request rates.

Our contributions can be summarized as follows: (i) we present three different protocols for content discovery with enhanced privacy; (ii) we provide a PoC and show how it can be integrated into Bitswap; (iii) we provide a performance, security, and privacy evaluation revealing feasibility trade-offs.

The remainder of the paper is structured as follows: In Section II, we present related work. We explain the functionality of Bitswap in Section III and present our privacy-enhanced versions in Section IV. The PoC is described in Section V and evaluated in Section VI. Section VII concludes the paper.

II. Related Work

Different security and privacy challenges for ICN exist, which prevent a widespread adoption [11, 12]. Especially the naming of content can result in a number of problems [13]. Most notably for us, a name can reveal the interest in content and therefore violate privacy. Chaabane *et al.* [14] propose to use multiple Bloom filters to protect the name. The client calculates a hierarchical Bloom filter and sends it to the router. The router checks its cache if the stored content is in the

Bloom filter to answer the request. If there is no match, the router checks in the hierarchical order for longest prefix match in its routing table. While our protocols also use Bloom filter, we request the Bloom filter from the server. Another direction to address the problem is using cryptography. For example, Bernardini *et al.* [15] propose to use proxy encryption to prevent the interest leak in ICN. In [16], the authors propose to use homomorphic encryption to increase lookup privacy in a system, where a broker matches consumer and provider. In some cases the content itself also needs to be protected, e.g., for access control, which can be realized by using attribute-based encryption [17, 18].

Solutions for ICN also need to consider forwarding of requests, since in ICN providers do not have to be direct neighbors. In contrast to ICN, IPFS has a slightly different approach for provider discovery. This is in part due to the P2P nature of IPFS, where every peer can be a content provider, requestor and router. However, IPFS has similar privacy challenges to ICN, e.g., access control, provider, and client privacy. Some research proposes to add access control to IPFS mostly using encryption partly in combination with a blockchain [19]. While research trying to improve the privacy of ICN might be applicable to IPFS, research targeting directly the privacy in IPFS is sparse. Balduf *et al.* [20] investigate the privacy problem for Bitswap. The authors propose to use a salted hash instead of the identifier for Bitswap before sending a request. This can also be considered as an insecure version of Private Set Intersection [21].

The most practical PSI protocols can be split into protocols using Diffie-Hellman (DH) and Oblivious Transfer (OT) [10]. While protocols using OT are faster for large balanced set sizes, for lower and unbalanced set sizes DH based protocols are faster. Furthermore, the communication overhead of DH protocols is lower compared to PSI protocols using OT. Practical applications for PSI include contact tracing [7], mobile contact discovery [22], and discovery of leaked passwords [9]. A detailed comparison of PSI protocols can be found in [5].

Our approaches utilize PDS and ECDH-PSI [7, 23], which is a hybrid of two other PSI protocol proposals [8, 9]. To the best of our knowledge, this is the first work analyzing the trade offs and the impact of PDS and PSI for provider discovery in ICN or Bitswap.

III. The Bitswap Protocol

In ICN, every piece of data has a specific identifier [2]. One of the existing naming schemes are self-certifying names. Self-certifying names allow a receiver to check the integrity of the data without additional information. The identifier is used to request the data from the network. Requests are either answered directly or forwarded to other peers.

Bitswap [24] is the default data exchange protocol for IPFS. Blocks are the basic data unit and a file can consist of multiple blocks. Blocks are identified by their Content Identifier (CID). A CID is a self-certifying name and consists of multiple components: a multi-base prefix, a version identifier, a multi-codec identifier, and a multihash. The multihash contains a code for the used hashing function, the length of the digest, and the hash of the file. The multihash is used to identify blocks in storage. Bitswap requests CIDs for specific blocks from the network and also answers requests from other nodes.

The following message exchange for block retrieval refers to Bitswap v1.2.0, which we also denote as Vanilla-Swap. We will denote the requester as client and all other peers in the network as server. First, a client sends a `WANT-HAVE`, containing the client's interest as a list of CIDs, to all neighbors. Neighboring servers receiving the request answer with a `HAVE` or `DontHAVE` depending on the presence or absence of the block. In case the block was not received after a certain delay, the client searches for more servers using the Kademlia-based DHT. When peers storing the content are discovered, one selected peer receives a `WANT-BLOCK`, requesting the block. The server answers with a `BLOCK` containing the block. For simplicity, Fig. 1 shows this exchange for two peers; additional peers would be treated as server, receiving a `WANT-BLOCK` in case the block retrieval fails. After Bitswap received and verified the block all neighbors receive a `CANCEL`.

While there are different message types, all messages are embedded in and sent as Bitswap messages. The `WANT-HAVE`, `WANT-BLOCK`, and `CANCEL` requests are classified as a WantList. `HAVE` and `DontHAVE` are BlockPresences. `BLOCK` and data of the block are also parts of the Bitswap message. As a consequence, the different request can be combined and do not necessarily require additional round trip times.

The `CANCEL` indicates that Bitswap is no longer interested in the CID. This is important for another behavior of Bitswap. Peers store received `WANT-HAVE` requests in a peer specific ledger. In case Bitswap receives a block, it checks the peers' ledger if another peer is also interested in the block. If other peers are interested in the block, Bitswap sends a `HAVE` to the interested peers. The `CANCEL` removes this interest.

There exists an exception to this exchange. Blocks below a certain threshold (default: 1024 Bytes) are sent immediately even if the request is a `WANT-HAVE`.

A block might reference followup blocks, e.g., in case of files. For the followup requests, Bitswap sends an opportunistic `WANT-BLOCK` to one peer and a `WANT-HAVE` to the other peers that previously announced possession of the block.

At the moment, Vanilla-Swap does not forward requests, however, it is under investigation [25].

IV. Privacy-Enhanced Bitswap

The aim of Bitswap is a fast and efficient exchange of data. Preferably, Bitswap should take advantage of the self-scalability of a P2P data exchange, e.g., caching. However, this reveals information to servers. As an example Balduf *et al.* [20] showed the possibility to track clients' behavior by passively monitoring the data request. The problem is to find a server possessing the block, without revealing the interest to other peers. It would be sufficient, if only one server possessing the block is informed about the interest. Nonetheless, the content discovery process of Bitswap leaks the interest to many additional peers.

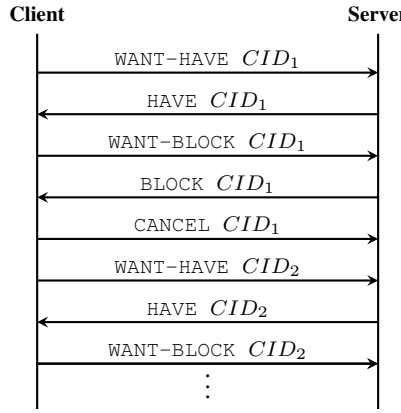


Fig. 1: Vanilla-Swap.

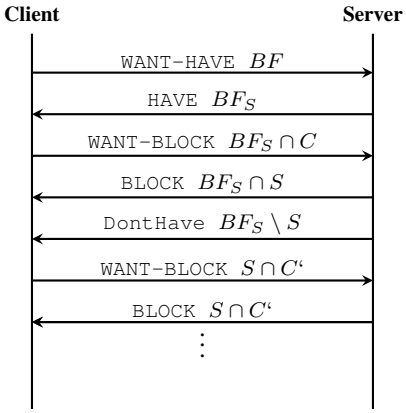


Fig. 2: Bloom-Swap.

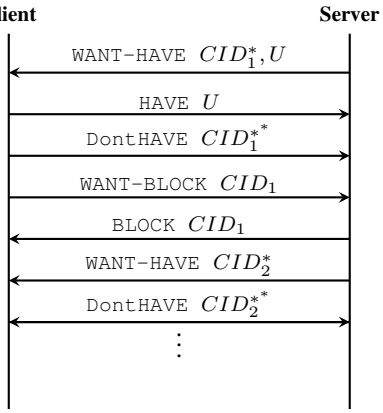


Fig. 3: PSI-Swap/BEPSI-Swap.

Our proposed solutions change the block discovery process to improve client privacy. To this end, we focus on reducing the number of servers which receive the interest and assume semi-honest adversaries. Our protocols are therefore primarily not designed to protect server information. In fact, the protocols can reveal information which is otherwise hidden, i.e., the amount of stored blocks. Furthermore, our protocols cannot protect the actual exchange of the block. It is therefore necessary to disclose interests to a server for the block download.

A. Bloom-Swap

One solution to reduce the information leakage is to replace the request of a specific CID with a general interest indication. The server responds to the request by sending an inventory list of all stored CIDs. This list can be potentially very long and could consume a lot of bandwidth. Therefore, to save space and give the server some plausible deniability the list is encoded as a Bloom filter [6].

Bloom filters (BF) are PDS and can be used to approximate membership. The BF is initially a fixed length set of unset bits. An element, c_i is added to the BF by setting specific bits of the set. The number of bits depends on a predetermined number of hash functions, H. For each hash function, H_j, the value $H_j(c_i)$ is computed. The result determines the bit to set. To check if an element is in the BF, it is checked whether all specific bits are set. Due to collisions of the different hash functions, a false positive is possible, i.e., all bits corresponding to an element not added to the BF are set. False negatives are impossible, if a bit is unset, none of the elements can be associated with the position.

Based on this, we propose Bloom-Swap, illustrated in Fig. 2. In Bloom-Swap, we replace `WANT-HAVE`s of specific CIDs with a request of a BF containing all stored CIDs. After receiving the BF, the client can check, if the CIDs are in the BF. If an element is in the BF, the client can send a `WANT-BLOCK` request, to the server. However, it is possible that a server does not store the block, due to a false positive. Since BF do not have false negatives, however, it is guaranteed that a server does not offer a CID, if the lookup fails. For consecutive requests, we can skip requesting the BF, since it already contains a list of all stored CIDs. If the storage information of the server changes, a new BF needs to be sent. It should be noted that the server can decide, which elements to include in the BF.

B. PSI-Swap

Bloom-Swap protects client interests only. In fact, it reveals even more information about the server than Vanilla-Swap: A client can check all CIDs if they are contained inside the BF to determine, which blocks are stored by the server. A curious entity could now request the BF of all peers, approximating the storage information of the whole network. Furthermore, Bloom-Swap is probabilistic, probably protecting the client's privacy. To increase the cost of such an approximation and therefore further protect the server privacy, we can use PSI.

The goal of PSI is to check whether an element or key is known by both parties, without revealing any additional information. After the execution of PSI, the only information each party learns is the intersection of elements, i.e., if the server would send a `HAVE` or `DontHAVE`. PSI may leak the set size of both parties, but this is often not to be considered sensitive. We propose to adapt the semi-honest ECDH-PSI protocol presented in [7] to Bitswap, creating *PSI-Swap*.

PSI-Swap follows basically the same steps as Vanilla-Swap. A block is requested, the server answers, and the client learns if a server stores the block. The difference is that the server does not know which item the client is interested in until the `WANT-BLOCK` is sent. To accomplish this, the CIDs are transformed into cyclic group points. Specifically, client and server choose a cyclic group and a Hash function H to map a multihash to the cyclic group. Additionally, both choose a random number r_C and r_S. For the discovery, the client transforms the multihash of the CID c_i using the H and r_C:

$$v_i = H(c_i)^{r_C}. \tag{1}$$

Instead of looking up if the server stores v_i, it further transforms the element with r_S to

$$w_i = (H(c_i)^{r_C})^{r_S}, \tag{2}$$

and sends w_i as a `DontHAVE` back. For PSI, the client needs further information from the server. This is similar to Bloom-Swap, a list of all stored blocks. In contrast to Bloom-Swap, this list does not contain the plain multihashes (s_j) but instead transformed multihashes:

$$u_j = H(s_j)^{r_S}. \tag{3}$$

The server sends this list with a `HAVE` to the client. After receiving u_j and w_i the client reverses her r_C transformation:

$$x_i = ((H(c_i)^{r_C})^{r_S})^{\frac{1}{r_C}} = H(c_i)^{r_S}. \tag{4}$$

The x_i can be used to determine the intersection. If there is a u_j equal to x_i, the client knows that the server has c_i. The client can then send a `WANT-BLOCK` with c_i to the server. For consecutive requests, it is sufficient if the server only sends w_i. In Fig. 3, we show the message exchange of PSI-Swap, where U, V, and W represent the sets of u_i, v_i, and w_i.

C. BEPSI-Swap

BEPSI-Swap is a combination of Bloom-Swap and PSI-Swap, namely a Bloom-filter Extended PSI-Swap (BEPSI-Swap). It increases the efficiency using the BF approach of Bloom-Swap at the cost of re-introducing the false positive probability into PSI-Swap. To this end, we change the encoding of the set $U = \{u_j\}$. In PSI-Swap, U is a set/list of all transformed multihashes. In BEPSI-Swap, the elements of U (i.e., u_j) are encoded in a BF containing all transformed multihashes. The message exchange of BEPSI-Swap consists accordingly of the same steps as in PSI-Swap (see Fig. 3).

V. Integration and Implementation

For the integration of our privacy-enhanced Bitswap protocols, our main goal is to maintain downwards compatibility. We therefore maintain Vanilla-Swap functionality, despite any privacy enhancements, and use existing message types and formats. Clients can still decide to request root CIDs with Vanilla-Swap or with our privacy-enhanced protocol. A Vanilla-Swap server can still process and respond to privacy-enhanced requests; the answer, however, would be meaningless for the client. Our implementation is based on `go-bitswap v0.10.1`, the Go implementation of Bitswap [24].

A. General Changes

In general, the Bitswap code consists of four components: (i) server, which is responsible for answering requests; (ii) client, which is responsible for sending requests, selecting peers, and triggering the search for additional provider; (iii) tracer, which records traffic statistics; (iv) network, which converts Bitswap messages to the wire format and sends and receives messages. Accordingly, the client sends and the server processes `WANT-HAVE`, `WANT-BLOCK`, and `CANCEL` messages. Vice versa, the server sends and the client processes `HAVE` and `DontHAVE` messages.

For each of our protocols, we developed and added a utility component to the Bitswap code: one for client and one for server functions. The `ClientUtil` manipulates in- and outgoing requests. The `ServerUtil` handles the sending of the inventory list and takes care of necessary CID adjustments. For Vanilla-Swap, both utility components do nothing. For Bloom-Swap, the `ClientFilterUtil` replaces or answers the `WANT-HAVE` requests of outgoing messages, temporary caching request if necessary. The `ServerFilterUtil` maintains and sends the BF. PSI-Swap and BEPSI-Swap use the same components; the only difference is the sending and handling of U, the transformed server list. The `ServerPsiUtil` deals with the creation of U, and the transformation of v_i into w_i. The `ClientPsiUtil` handles the storage and requesting of U, as well as creation of v_i, x_i. It also interprets x_i, changing it into a `HAVE` or `DontHAVE` depending on the result. The client and server side use different r, reducing the leakage of information in case of newly send U. The `ClientUtil` is located in the network component, manipulating messages just before sending and immediately after receiving the message. Changing the CIDs as close to the wire as possible, allows the least interference with the default Bitswap behavior. The `ServerPsiUtil` is located in the decision engine of the server component. The specific Bitswap protocol can be configured and is decided on startup.

B. Inventory U

In Bitswap, CIDs are requested and sent. A received block needs to match its CID. This makes it difficult to request specific objects with unknown or changing content, e.g., a BF. We introduce DummyCIDs to address specific content of a peer. The DummyCIDs, are $5\,B$ CIDv1 with specific multi-codec identifiers, multihash code, and a digest length of $0\,B$. Similarly, we encoded the BF and for PSI the compressed group element as the digest of a multihash. Afterwards, the multihash, with the BF or group element, is encoded as a CIDv1 with a specific multi-codec identifier.

For the BF, we chose the `bbloom`[1] implementation, which is at the time of writing part of the IPFS blockstore-wrapper. The `bbloom` package uses SipHash-2-4 as its hashing function. Size and number of hashing functions are calculated based on the number of entries and a given false positive rate. The number of entries in the BF increases in steps of 50 entries and depends on the number of elements. The JSONMarshall of the BF is the digest of the CID. For our PoC implementation, we use one fixed false positive rate of $f = 0.0001$.

The main ideas behind Bloom-Swap and BEPSI-Swap is to reduce the communication and memory overhead of U. While we use `bbloom`, other implementations or PDS are also possible. For example, cuckoo filter would allow a simpler communication of updates, since it is not necessary to re-send the whole filter. For a low false positive rate, cuckoo filter can be smaller than Bloom filter.

The communication overhead can be further reduced, by caching U, making it a one time overhead. However, U is unlikely to be static over time. Elements can be added, deleted, or changed all resulting in other CIDs available

[1] https://github.com/ipfs/bbloom (2022-09-27)

at the server. This requires some considerations concerning refreshing or updating U. In order to update U, it is possible to use a push or (periodic) pull approach. Push updates could be communicated, similarly to a pub-sub mechanism, using Bitswap's current functionality. The pub-sub mechanism has the advantage that changes of U can be communicated immediately. Since Bitswap is mainly used in combination with IPFS, though, we assume a high churn rate [26]. In case of high churn and constantly changing set of neighbors, a pull mechanism seems to be more efficient. We therefore favor a pull mechanism in our PoC for all protocols. Accordingly, U is requested on a need-to-know basis and stored once received.

C. PSI Integration

The integration of the ECDH-PSI required some additional considerations and changes. ECDH-PSI can work with different cyclic groups. The server and the client, however, are required to use the same cyclic group. Different groups can have different point sizes, making it possible to distinguish between the used group by the size of the CID. Larger groups provide higher security at the cost of longer computation time and an increased number of transferred bytes. For simplicity, we chose to use one fixed group in our implementation. Alternatively, the used group could be negotiated, during the protocol handshake. It should be kept in mind, that supporting multiple groups at once raises the memory overhead as each supported group requires its own set of U and V.

Furthermore, ECDH-PSI requires that the answers arrive in the same order as they were sent. This is not guaranteed in Bitswap as messages can arrive out of order, split between multiple messages, or even indefinitely delayed. As an alternative, it would be sufficient, if the answer can be matched to the request. This is difficult, however, since the protocol requires a modification of the request in form of a multiplication. We therefore add an identifier into the CID of v_i. The identifier is realized as global incremental $16\,b$ unsigned integer and included in the multihash.

We cache the transformed request, v_i. The transformation needs to be calculated only once and can then be reused for all requests. Similarly, U can be reused for all peers.

The PoC implementation is available in a public GitHub repository[2] as well as the used cryptographic transformations[3]. For the group calculations, we use the CIRCL package [27].

VI. Evaluation

For our evaluation, we compare the different protocols with respect to their impact on the Bitswap performance, security, and privacy. The baseline for our comparison is the behavior of Vanilla-Swap, specifically `go-bitswap v10.0.1` [24].

A. Performance

The enhanced privacy mechanisms eventually add overhead to the Bitswap protocol. In the following, we evaluate the communication and memory overhead theoretically, and give an empiric evaluation of the computation overhead.

[2]https://github.com/epikd/go-bitswap/tree/psi-bitswap

[3]https://github.com/epikd/psiMagic

TABLE I: Communication overhead for n_c #client interests, n_s #elements server, and m #server. For PSI-based protocols, we assume an ideal BF and ristretto255.

	Client	**Server**
Bloom-Swap	$U' = \sum_{i=1}^{m} BF_i$	U
PSI-Swap	$U' = \sum_{i=1}^{m} U_i^*$, $W' = \sum_{i=1}^{m} W_i$	U, W
BEPSI-Swap	$U' = \sum_{i=1}^{m} BF_i$, $W' = \sum_{i=1}^{m} W_i$	U, W
BF	$\approx 1.44 \cdot \log_2(\frac{1}{f}) \cdot n_s$	
V	$\leq 3\,B \cdot n_c$	
W	$\leq 3\,B \cdot n_c$ (if `HAVE`), $\leq 45\,B \cdot n_c$ (if `DontHAVE`)	
U^*	$\approx 43\,B \cdot n_s$	

1) Communication and Memory: Our protocols add a small communication overhead, in the form of at most a few bytes per message and no additional round trip times. However, there is a comparatively large one-time communication overhead, i.e., the inventory list of the server U. An overview of the communication overhead can be found in TABLE I. U is also the biggest memory overhead, since every neighbor has its own U. The remaining memory overhead depends on the number of pending request and consists mainly of data needed for mapping c_i, and x_i.

For the content discovery in Vanilla-Swap, it is necessary to send a `WANT-HAVE` per CID per server, and the server needs to answer with `HAVE` or `DontHAVE`. While clients send a `WANT-HAVE` to all neighbors for the first discovery of content, it expects only servers holding the block to answer the request. In Bloom-Swap, it is only necessary to request the BF once. Once the client acquired the BF of a server, it is no longer necessary to send `WANT-HAVE`s, which can save two round trip times. For both PSI-Swap and BEPSI-Swap, each `WANT-HAVE` requires a `DontHAVE` answer. While this is the same as the normal exchange, the overall overhead is larger for the first discovery, since every server is expected to send a `DontHAVE` answer. Additionally, a `DontHAVE` is $2\,B$ longer than a `HAVE`. As a consequence of false positives in Bloom-Swap and BEPSI-Swap, unnecessary `WANT-BLOCK` messages will increase the communication overhead.

While there are no additional messages, the requested CIDs differ. Vanilla-Swap and the PSI-based protocols use different CIDs for the discovery, specifically the multihash of the CID differs. The multihash consists of a code field, length field, and digest field. The default multihash code is `0x12` ($1\,B$) with a length field of $1\,B$ and a digest length of $32\,B$. In PSI-Swap and BEPSI-Swap, we have different multihash codes, due to the varying ID and a digest length, which depends on the cyclic group. The digest itself is a compressed point with varying length, i.e., $32\,B$ (ristretto255), $33\,B$ (P-256), $49\,B$ (P-384), and $67\,B$ (P-521). The ID is an `uint16` which is encoded as a variant integer ($1\,B$–$3\,B$). Since our protocols require multi-codec identifier, we require CIDv1. For P-256,

the communication overhead is around $2\,B$–$4\,B$ per CID.

The remaining overhead is mainly a one time overhead and a result of the request of U. The size of the one time overhead mainly depends on the number of elements stored by the server. On the client side, the overhead is a request, an additional `WANT-HAVE` of a DummyCID. The size of the DummyCID depends on the multi-codec identifier, multihash code, and multihash length. The minimal size of the DummyCID is $4\,B$. The request adds some additional encoding, resulting in an additional overhead of $\approx$ $6\,B$–$8\,B$. On the server side, the overhead is the sending of U as a `HAVE`. For PSI-Swap, the overhead depends on the cyclic group, and can be approximated to $43\,B \cdot n$ if all elements are sent using ristretto255, excluding some additional wire encoding. For Bloom-Swap and BEPSI-Swap, the overhead depends on the BF implementation and false positive rate. The size m of a BF is given by

$$m = \log_2(\mathrm{e}) \cdot n \cdot \log_2(\frac{1}{f}) \approx 1.44 \cdot n \cdot \log_2(\frac{1}{f}) \qquad (5)$$

where n is the number of elements and f the false positive rate of the BF. To reconstruct the BF the number and type of Hashing functions is also required. Due to the usage of `uint64` for the BF bit set, `bbloom` has a step-wise size increase. For $n = 50$ and $f = 0.0001$, the `bbloom` BF of the CID digest has a size of $201\,B$. In our PoC, the BF needs to fit in one Bitswap message. However, Bitswap has a maximum receive size of $4\,MB$. This restricts the maximal server set to $\approx 875'000$ elements. Larger sizes are possible, however this would require additional logic to split and reassemble the BF.

The size of the BF also determines the memory overhead on the server side for Bloom-Swap. For the PSI-based protocols, there is an additional overhead of a plain CID to transformed CID mapping. In BEPSI-Swap, it would be sufficient to only store the BF, however, storing the mapping speeds up recreation of the BF in case the set changes.

The memory overhead of the client consists of U of all neighbors, two maps connecting plain CID and ID, and a mapping of plain CID to transformed CID. The translation of plain CID to ID is necessary for PSI. The mapping of plain CID to transformed CID allows to reuse the transformed CID.

The requesting set size will consist mostly of a single file, which starts with a single root CID. Depending on the file size, more blocks will follow. A normal user probably provides only a few files, which results in a small server set size. We therefore assume low or asymmetrical set sizes. The biggest memory and communication overhead is the creation and transfer of U. Through the leakage of some bytes of information, it should be possible to reduce the size of the requested U, reducing the initial overhead. The leaked information could be the first bits of the digest or the multihash code. However, leaking data reduces the gained privacy.

2) Computation: The computation overhead consists of lookups, de-/encoding, queuing, and the group calculations. For Bloom-Swap, there is a BF lookup for every selected peer, which yields no significant overhead. For the PSI-based protocols, the biggest overhead are the cyclic group calculations of V and X for clients, and U and W for servers.

In order to get an impression of the time span of the delay, we run the Go benchmarks of the (isolated) PSI calculations.[4] We evaluate the point creation of V, W, and X, which can be considered as encryption (V/U), re-encryption (W), and decryption (X). The benchmarks are run via `go test`. For the benchmarks, we use a consumer-grade device (denoted as Laptop) and a resource-restricted device (denoted as Raspberry). The Laptop is a Lenovo ThinkPad T580 with an Intel Core i7-8550U CPU with $1.8\,GHz$, and $16\,GB$ RAM, using Manjaro $64\,b$ (Kernel 5.10, go1.19.3 linux/amd64m) The Raspberry is a Raspberry Pi 4 B Rev 1.2 with $4\,GB$ RAM, using Raspberry Pi OS Desktop $64\,b$ (Kernel 5.15, go1.19.2 linux/arm64). Fig. 4 shows the results for different number of elements. Each benchmark was run 30 times with a different random number, the results are the mean values and show a linear behavior. While calculating W and X consists of multiplications only, V/U also includes a hashing operation.

For the feasibility of the protocol, it is important that especially the overhead for the server is low. While a client might be willing to contribute more resources to protect her privacy, a server's resource should be considered as scarce. Therefore, W is an important aspect. U only needs to be adjusted every time the blocks stored by the server changes, and W needs to be calculated on every request. Larger groups (e.g., P-521) incur a significant latency, which consumes a larger amount of resources on the server side. However, the computation time of smaller groups (e.g., ristretto255 or P-256) is much lower. The overhead of the client is bigger. While V only needs to be calculated once and can then be reused, X needs to be calculated for all servers. In IPFS with the current default settings for the connection manager, this can result in up to 100 more calculations. For clients, it means that requesting 10 CIDs could result in 1'000 calculations of X. Therefore, the computation latency is $\approx 78.7\,ms$ for the Laptop and $\approx 0.422\,s$ for the Raspberry, assuming ristretto255.

One important condition to make the privacy protection feasible is a low impact on other voluntary participants, i.e. servers. Servers have a one-time overhead by generating the BF and therefore U. Next to the one time overhead, Bloom-Swap reduces communication cost by eliminating most of the `WANT-HAVE` messages. PSI-Swap slightly increases message processing by requiring the calculation of W for each request. Even on a resource-restricted device, the overhead of PSI-Swap is for a low number of CIDs only a few milliseconds. Since the overhead on the client side is in general larger, our PSI-based protocols cannot completely replace Vanilla-Swap. For a reasonable number of CIDs, the overhead even on the server side is moderate. We therefore deem our protocols as a feasible privacy supplement.

B. Security

Since the protocols add an overhead, they might be exploitable for Denial-of-Service (DoS) attacks. In this section,

[4] https://github.com/epikd/psiMagic

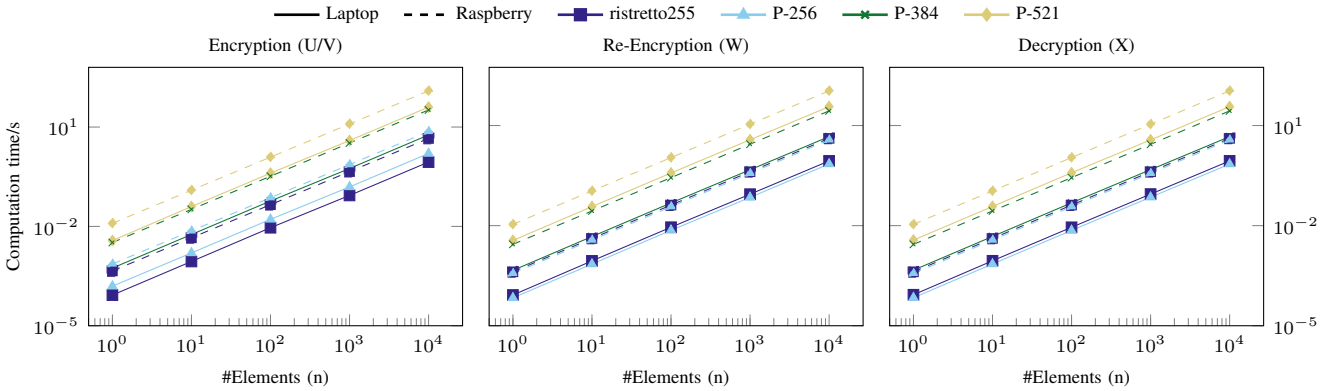


Fig. 4: Cyclic group computation time.

we evaluate amplification vectors, which potentially can be used for reflection and flooding attacks.

Our protocols have an amplification vector by requesting U (which is only a few bytes) and hence generating an answer including U, which could be potentially large. Another more reliable amplification vector already exists in Vanilla-Swap, if a requested block is stored. In this case, the answer to a `WANT-BLOCK` request is the block, which causes an amplified answer. While these amplification vectors exist, connections are secured by default with `Noise` [28], effectively preventing the exploitation of this attack vector, e.g., for reflection attacks.

An attacker, however, can flood a victim with messages, e.g., many `WANT-HAVE` requests, to keep the victim busy, impeding requests of other peers. Potential targets for such attacks are pinning services, which are responsible for keeping blocks available, or gateways, which make IPFS content available to non-IPFS speaking clients. In Vanilla-Swap, a flood of request causes a series of CID look-ups and scheduled tasks for sending data. Furthermore, as long as the attacker stays connected the requested CIDs are stored in a peer specific ledger. Our PSI-based protocols have a similar behavior, although, the requested CIDs are no longer stored in a ledger, the flood of request causes a series of calculations instead of look-ups. Assuming a node is not downwards compatible, Bloom-Swap would improve the resilience to such floods, since it could simply ignores requests, not requesting the BF. Therefore, for Bloom-Swap, it would be necessary to send floods of messages requesting the BF. All versions are similarly vulnerable to a flood of `WANT-BLOCK` requests.

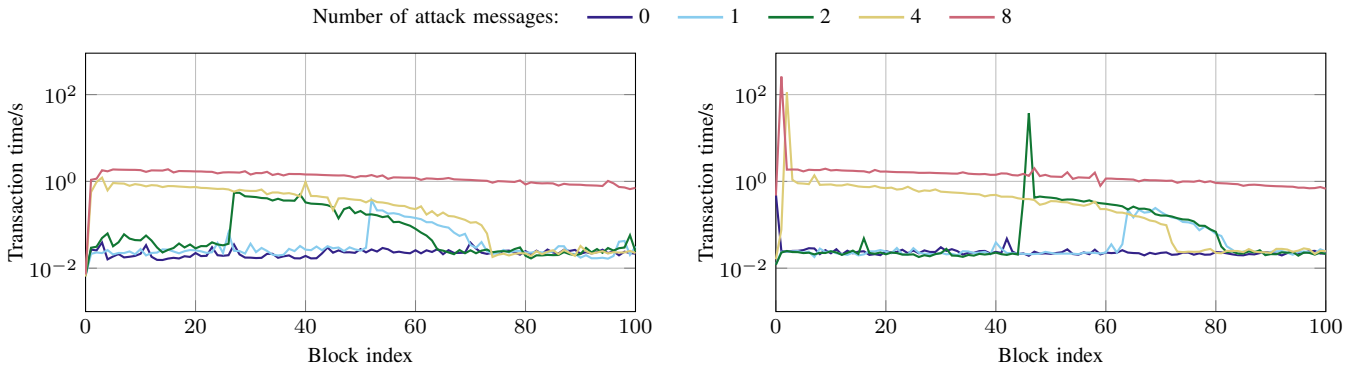
We investigate the risk of a flood DoS attack with the following experiment. As the setup for our experiment, we directly connect the Laptop and Raspberry, using a $1\,m$ CAT6a network cable, which results in an average ICMP ping latency of around $223\,\mu s$. All wireless communication on Raspberry and Laptop are turned off. On the Raspberry, we deploy a Bitswap server using a libp2p [29] host for network management. We configured Bitswap to use QUIC as transport protocol for both attacker and client. To the server's block store, we add 1'000 blocks, 100 randomly created, $256\,kB$ blocks and 900 smaller sample blocks. We chose $256\,kB$, since it is the default aimed block size of the IPFS chunker. In our experiment, we assume that the attacker has more resources (deployed on Laptop) than the victim (deployed on Raspberry).

We simulate an attacker with a libp2p host who generates and sends Bitswap messages containing many `WANT-HAVE` requests, i.e., attack messages. Specifically, a single attack message contains $89'240$ `WANT-HAVE` requests for different CIDs, giving the message a size of $4'194'285\,B$. Please note that this is the maximum message size for our Bitswap-based PoC. For our evaluation, we increase the number of attack messages and observe the impact of the requests.

The impact of flooding is evaluated using transaction success and transaction delay [30]. In our case, a transaction is the retrieval of a block. We define the transaction as successful, when the retrieval of a block is faster than the current default provider search delay of $1\,s$. For our experiment, the client executes 100 transactions retrieving the 100 blocks of size $256\,kB$. Our results assume a hot start, i.e., the required creation and request of U is executed beforehand. In our experiment, the hot start is ensured through request and retrieval of one small block by the client, before sending the attack messages. The experiment is run using Vanilla-Swap and BEPSI-Swap. Due to the hot start, the results of BEPSI-Swap are similar to PSI-Swap. Furthermore, the short latency reduces possible speed advantages of Bloom-Swap, rendering the result to be similar to Vanilla-Swap. The code for our experiment is publicly available[5].

Before we ran the experiment for different attack loads, we checked if it is possible to kill the server entirely. As it turns out, the server running Vanilla-Swap can indeed be killed due to memory exhaustion. We needed between 37–40 message to kill Vanilla-Swap on our Raspberry. Bloom-Swap, PSI-Swap, and BEPSI-Swap were able to receive more messages without problems. The memory exhaustion comes from storing the requested CIDs in the peer's ledger, which is not used by our protocols. To avoid side effects due to an increased memory usage, we removed the ledger utilization for Vanilla-Swap in our experiments.

[5]https://github.com/epikd/go-bitswap/tree/psi-bitswap/experiment



(a) Vanilla-Swap.

(b) BEPSI-Swap.

Fig. 5: Block retrieval time after different request floods.

In Fig. 5, we show the transaction time for each block for varying attack loads, i.e., increasing number of attack messages. In Fig. 5a, we can already see an effect after one attack message, where more than the last 20 blocks experience a transaction delay. Since all transactions require less than $1\,s$, they are still successful. After 4 attack messages, we can see the first transaction failures, and after 8 attack messages almost all transactions fail. The increase seems to appear after some time and seems to stem from the Bitswap task scheduler. The result shows Vanilla-Swap's vulnerability to flooding attacks, where sending $\approx 32\,MB$ of data, can cause multiple transaction failures.

For BEPSI-Swap, we observe that the transaction time remains low until approximately the first attack message is completely processed, which takes $\approx 35\,s$. In order to show the effect of the DoS, we postponed the block retrieval for BEPSI-Swap and started it after $36\,s$. The results for BEPSI-Swap (and PSI-Swap) can be seen in Fig. 5b. While the general behavior is approximately the same as in Vanilla-Swap, there are some anomalies in the form of a spike for a single block. From the results, Bitswap seems to stall the answer of requests until all requests are processed, resulting in a spike for the 1st block after the attack message is processed. The transaction time, after this spike shows a similar trend as Vanilla-Swap.

Using the PSI-based protocols, it is possible to first stall and then delay transactions from a server for a certain time. However, the same attack vector also exists in Vanilla-Swap and enables an adversary to delay transactions from a server. The results show that Bitswap in general and the PSI-based protocols in particular should implement a rate limiting, i.e., limit the number of requests per peer in a specific time interval. Since the PSI-based protocols Swap can be used simultaneously to Vanilla-Swap, there could be different limit for Vanilla-Swap and the PSI-based protocols to account for the different computation overhead.

Please note, that these results do not imply that there is a general threat in the IPFS network. The experiment targeted a device with comparatively low resources, and the attacker was very close to the victim with a latency of only a few hundreds of microseconds. A closer inspection of the reasons of the transaction delay and a more detailed analysis of the threat are necessary, but out of scope of this work. We shared these results with Protocol Labs.

C. Privacy

Compared to Vanilla-Swap, all of our protocols improve clients' content discovery privacy by obfuscating the item of interest. In particular, interests are shared on a need-to-know basis only. Upon closer inspection, the different protocols provide slightly different privacy properties, specifically, the amount of information a server learns.

For the PSI-based protocols, a server learns the time of a request and the number of interested items. While leaking the set sizes is typically considered as non-critical, the leak could affect the client privacy in Bitswap. If a client is interested in a file consisting of multiple blocks, these blocks are requested along a Directed Acyclic Graph (DAG), starting with the root. If a client's requests follow a DAG, a server could infer the interest based on her storage. For file request, it therefore might be beneficial for client privacy to forgo discovery on follow-up requests from the same peer. In Bloom-Swap, the server only learns that a client was interested in some CID.

Due to the uncertainty of the BF-based approaches, a client might leak more information due to false positives. In the case of false positive, a client sends a `WANT-BLOCK`, revealing the interest, despite the server not storing the item. This risk can be adjusted with the false positive rate of the BF.

A malicious (active) server can learn additional information. The malicious server can manipulate U, claiming that it stores more CIDs than actually available. As result, it will receive client requests, eventually learning client interests. Since this problem is a Vanilla-Swap inherent problem, we consider semi honest (passive) adversaries only. Yet, it adds an attack vector to circumvent the privacy protection for our protocols.

While improving the privacy of the server is not our primary goal, our protocols still have an effect on the server privacy. Since the server sends a list of all stored elements, more information about the server is revealed. In case of Bloom-Swap, an approximation of all elements of the server is

revealed. However, the BF provides some plausible deniability for the server. The false positive rate is therefore a trade-off between client and server privacy. In case of PSI-Swap, the number of elements stored by the server are revealed. However, a client needs to know and request a specific CID to gain the storage information, which is similar to Vanilla-Swap. Since BEPSI-Swap is a combination of Bloom-Swap and PSI-Swap, it mainly shares the advantages of both protocols. Due to the BF, the server gains plausible deniability,

VII. Conclusion

We explored three different protocols to improve the privacy of the Bitswap content discovery process. We find that all protocols effectively reduce the privacy leak of a peer's interest. The feasibility trade-offs of the protocols, however, differ. Bloom-Swap can reduce the communication cost of the discovery process and adds client privacy at the cost of a reduced server privacy. PSI-Swap and BEPSI-Swap can improve the privacy (also for servers) at the cost of a computation overhead. While feasible for reasonable request rates, the overhead of the PSI-based protocols can be considerable for large numbers of items. In Summary, all proposed protocols improve the privacy of content discovery. While Bloom-Swap offers the additional potential to speed up data exchange, BEPSI-Swap provides better privacy with a reasonable overhead.

Acknowledgements

We thank Protocol Labs for funding our research.

References

[1] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *CoNext '09: Proceedings of the 5th ACM International Conference on Emerging Networking Experiments and Technologies*, Rome, Italy, Dec. 2009, pp. 1–12.

[2] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman, "A survey of information-centric networking," *IEEE Communications Magazine*, vol. 50, no. 7, pp. 26–36, 2012.

[3] E. Daniel and F. Tschorsch, "Ipfs and friends: A qualitative comparison of next generation peer-to-peer data networks," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 1, pp. 31–52, 2022.

[4] J. Benet, "IPFS - content addressed, versioned, P2P file system," vol. abs/1407.3561, Jul. 2014. [Online]. Available: http://arxiv.org/abs/1407.3561.

[5] B. Pinkas, T. Schneider, and M. Zohner, "Scalable private set intersection based on ot extension," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 2, pp. 1–35, 2018.

[6] 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. [Online]. Available: https://doi.org/10.1145/362686.362692.

[7] N. Angelou, A. Benaissa, B. Cebere, W. Clark, A. J. Hall, M. A. Hoeh, D. Liu, P. Papadopoulos, R. Roehm, R. Sandmann, *et al.*, "Asymmetric private set intersection with applications to contact tracing and private vertical federated machine learning," vol. abs/2011.09350, 2020. [Online]. Available: https://arxiv.org/pdf/2011.09350.pdf.

[8] N. Trieu, K. Shehata, P. Saxena, R. Shokri, and D. Song, "Epione: Lightweight contact tracing with strong privacy," *arXiv preprint arXiv:2004.13293*, 2020.

[9] Á. Kiss, J. Liu, T. Schneider, N. Asokan, and B. Pinkas, "Private set intersection for unequal set sizes with mobile applications.," *Proc. Priv. Enhancing Technol.*, vol. 2017, no. 4, pp. 177–197, 2017.

[10] M. Rosulek and N. Trieu, "Compact and malicious private set intersection for small sets," in *CCS '21: Proceedings of the 2021 ACM SIGSAC Conference on Computer & Communications Security*, Virtual Event, Republic of Korea, Nov. 2021, pp. 1166–1181. [Online]. Available: https://doi.org/10.1145/3460120.3484778.

[11] K. Pentikousis, B. Ohlman, E. B. Davies, S. Spirou, and G. Boggia, *Information-centric networking: Evaluation and security considerations*, RFC 7945 (Informational), RFC, Fremont, CA, USA: RFC Editor, Sep. 2016. DOI: 10.17487/RFC7945. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7945.txt.

[12] R. Tourani, S. Misra, T. Mick, and G. Panwar, "Security, privacy, and access control in information-centric networking: A survey," *IEEE communications Surveys & Tutorials*, vol. 20, no. 1, pp. 566–600, 2017.

[13] E. Mannes and C. Maziero, "Naming content on the network layer: A security analysis of the information-centric network model," *ACM Computing Surveys*, vol. 52, no. 3, pp. 1–28, 2019.

[14] A. Chaabane, E. De Cristofaro, M. A. Kaafar, and E. Uzun, "Privacy in content-oriented networking: Threats and countermeasures," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 3, pp. 25–33, 2013.

[15] C. Bernardini, S. Marchal, M. R. Asghar, and B. Crispo, "Privicn: Privacy-preserving content retrieval in information-centric networking," *Computer Networks*, vol. 149, pp. 13–28, 2019.

[16] N. Fotiou, D. Trossen, G. F. Marias, A. Kostopoulos, and G. C. Polyzos, "Enhancing information lookup privacy through homomorphic encryption," *Security and Communication Networks*, vol. 7, no. 12, pp. 2804–2814, 2014.

[17] M. Ion, J. Zhang, and E. M. Schooler, "Toward content-centric privacy in icn: Attribute-based encryption and routing," in *ICN '13: Proceedings of the 3rd ACM SIGCOMM Workshop on Information-Centric Networking*, Hong Kong, China, Aug. 2013, pp. 39–40.

[18] B. Li, D. Huang, Z. Wang, and Y. Zhu, "Attribute-based access control for icn naming scheme," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 2, pp. 194–206, 2016.

[19] S. Wang, Y. Zhang, and Y. Zhang, "A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems," *IEEE Access*, vol. 6, pp. 38 437–38 450, 2018.

[20] L. Balduf, S. Henningsen, M. Florian, S. Rust, and B. Scheuermann, "Monitoring data requests in decentralized data storage systems: A case study of ipfs," in *ICDCS '22: Proceedings of the 42nd IEEE International Conference on Distributed Computing Systems*, Bologna, Italy, Jul. 2022, pp. 658–668.

[21] B. Pinkas, T. Schneider, and M. Zohner, "Faster private set intersection based on OT extension," in *USENIX Security '14: Proceedings of the 23rd USENIX Security Symposium*, San Diego, CA, USA, Aug. 2014, pp. 797–812. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/pinkas.

[22] D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert, "Mobile private contact discovery at scale," in *USENIX Security '19: Proceedings of the 28th USENIX Security Symposium*, Santa Clara, CA, USA, Aug. 2019, pp. 1447–1464.

[23] OpenMined, *Github – openmined/psi: Private set intersection cardinality protocol based on ecdh and bloom filters*, https://github.com/OpenMined/PSI, Accessed: 2022-06.

[24] Protocol Labs, *Github – ipfs/go-bitswap: The golang implementation of the bitswap protocol*, https://github.com/ipfs/go-bitswap, Accessed: 2022-09.

[25] A. De la Rocha, D. Dias, and Y. Psaras, "Accelerating content routing with bitswap: A multi-path file transfer protocol in ipfs and filecoin," p. 11, 2021.

[26] E. Daniel and F. Tschorsch, "Passively measuring ipfs churn and network size," in *ICDCSW '22: Proceedings of the 42nd IEEE International Conference on Distributed Computing Systems Workshops*, Bologna, Italy, Jul. 2022, pp. 60–65.

[27] A. Faz-Hernández and K. Kwiatkowski, *Introducing circl: An advanced cryptographic library*, Available at https://github.com/cloudflare/circl. v1.2.0 Accessed Sep 2022, Cloudflare, Jun. 2019.

[28] T. Perrin, "The noise protocol framework - revision 34," pp. 1–65, Jul. 11, 2018. [Online]. Available: http://www.noiseprotocol.org/noise.pdf.

[29] Protocol Labs, *Github – libp2p/go-libp2p: Libp2p implementation in go*, https://github.com/libp2p/go-libp2p, Accessed: 2022-11.

[30] J. Mirkovic, A. Hussain, B. Wilson, S. Fahmy, P. Reiher, R. Thomas, W.-M. Yao, and S. Schwab, "Towards user-centric metrics for denial-of-service measurement," in *Proceedings of the 2007 workshop on Experimental computer science*, 2007, p. 14.