Bringing Decentralized Search to Decentralized Services

Mingyu Li, Jinhao Zhu, and Tianxu Zhang, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Shanghai AI Laboratory; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Cheng Tan, *Northeastern University;* Yubin Xia, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Shanghai AI Laboratory; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Sebastian Angel, *University of Pennsylvania;* Haibo Chen, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Shanghai AI Laboratory; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

https://www.usenix.org/conference/osdi21/presentation/li

This paper is included in the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation.

July 14–16, 2021

978-1-939133-22-9

Open access to the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Bringing Decentralized Search to Decentralized Services

Mingyu Li, Jinhao Zhu, Tianxu Zhang, Cheng Tan[†], Yubin Xia, Sebastian Angel[⋆], Haibo Chen

Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Shanghai AI Laboratory

Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

[†]*Northeastern University* [⋆]*University of Pennsylvania*

Abstract

This paper addresses a key missing piece in the current ecosystem of decentralized services and blockchain apps: the lack of decentralized, verifiable, and private search. Existing decentralized systems like Steemit, OpenBazaar, and the growing number of blockchain apps provide alternatives to existing services. And yet, they continue to rely on centralized search engines and indexers to help users access the content they seek and navigate the apps. Such centralized engines are in a perfect position to censor content and violate users' privacy, undermining some of the key tenets behind decentralization.

To remedy this, we introduce DESEARCH, the first decentralized search engine that guarantees the integrity and privacy of search results for decentralized services and blockchain apps. DESEARCH uses trusted hardware to build a network of workers that execute a pipeline of small search engine tasks (crawl, index, aggregate, rank, query). DESEARCH then introduces a *witness* mechanism to make sure the completed tasks can be reused across different pipelines, and to make the final search results verifiable by end users. We implement DESEARCH for two existing decentralized services that handle over 80 million records and 240 GBs of data, and show that DESEARCH can scale horizontally with the number of workers and can process 128 million search queries per day.

1 Introduction

Most of today's online services—including search, social networks, and e-commerce—are centralized for reasons such as economies of scale, compatible monetization strategies, network effects, legal requirements, and technical limitations. Yet, since the birth of the Internet, there have been periods of intense interest in decentralization, including the peer-to-peer systems bonanza of the early and mid 2000s [57, 90, 107] and the current blockchain boom [39, 94, 113]. A rich set of decentralized services have appeared and are able to offer most of the functionalities that common centralized online services provide, as listed in Figure 1. Proponents of decentralization argue that centralized services often employ anti-consumer practices owing to their monopolistic positions [18, 27], and the mismatch between users' expectations and operators' incentives [44]. Further, centralized services are particularly susceptible to censorship [4, 41] (either self-imposed or coerced through technical or legal means) and collect vast amounts of user information [13, 14].

Service	Centralized	Decentralized
Currency	U.S. Dollars	Bitcoin [94]
Online Marketplace	eBay	OpenBazaar [28]
Social Media	Twitter	Steemit [40]
Video Sharing	Youtube	DTube [8]
Social Network	Facebook	Mastodon [16]
Public Storage	DropBox	IPFS [59]
Messaging	Slack	Matrix [25]
Video Conference	Zoom	Zipcall [50]
Website Hosting	WiX [47]	ZeroNet [49]
Financial Betting	Etoro [12]	Augur [1]
Supercomputing	Titan [45]	Golem [17]
Document Collaboration	Google Docs	Graphite [21]

FIGURE 1—Centralized services and decentralized alternatives.

While the idea of building fully decentralized services is alluring, developers must currently make a significant compromise: they must defer search functionality to a centralized service. For example, OpenBazaar [28] makes a strong case for a decentralized marketplace, but users must use a centralized search engine such as Bazaar Dog [46] or Duo Search [9] to discover what items are for sale in the first place. A similar compromise is made by other popular services [8, 16, 28, 40, 49]. This state of affairs is problematic because search is not an optional feature but rather a core component of these systems. Without decentralized search, the purported goals of anti-censorship is hard to attain: the search engine could trivially track users' queries, and opaquely censor particular content [3, 4, 7, 32, 41]. For example, Steemit [40] is a decentralized social media service where posts are stored on the public Steem blockchain [39], but Steemit developers have been known to prevent users' posts from appearing on the front end site [41].

Prior proposals. Several search engines [29, 81] propose reaching consensus amongst replicas to ensure the correctness of search indexes. However, these engines rely on a central website hosted at the third party to answer queries. As a result, an end-user who visits this website has no way to validate the integrity of the displayed results, or to determine whether there are missing entries. As an alternative, peer-to-peer-based search engines [24, 48] allow shared indexes between peers and queries can be issued to any peer (essentially implementing a distributed hash table). However, these engines do not support verifiable indexes, and allow peers to monitor clients'

requests, leading to severe privacy concerns. Finally, many blockchains encourage users to run their own indexer node or to use a third-party indexer [30, 35] to access the content in the blockchain. However, most users lack the resources and the expertise needed to deploy their own indexers, and third-party indexers must be trusted to not censor content or violate users' privacy.

Goals and contribution. Building a decentralized search engine that avoids the aforementioned shortcomings is far from trivial. First, the search engine should be able to authenticate the data source to make sure the dataset contains all data items and is free from forgeries. Second, the user's intention, including the query keywords and search results, should be kept private from any party. Third, the search engine should be able to provide a proof of execution to clients that explains how the search results were generated, and why the results are correct. Last, the search engine should have reasonable costs and scale to support many users.

To meet these goals, we introduce DESEARCH, the first decentralized search engine that allows users to verify the correctness of their search results, and preserves the privacy of user queries. DESEARCH outsources fragments of search tasks such as crawling, indexing, aggregation, ranking, and query processing to trusted execution environments (TEEs) running on untrusted *executors* that compose a decentralized network, and introduces new data structures and mechanisms that make executors' operations reusable and externally verifiable.

First, since each executor only has a local view of the computation, DESEARCH uses *witnesses*, which are a type of object that reflects the dataflow and establishes the correctness of results. A witness ensures that executors cannot lie about which sources they crawled, how they aggregated data, computed the index, or responded to a query. Verifying witnesses is not cheap, so DESEARCH amortizes the verification cost by reusing previously checked witnesses across queries, using designated executors—verifiers—to check witnesses on behalf of clients.

Second, DESEARCH uses a public storage service called *Kanban*. Kanban allows executors in the network to exchange intermediate information, agree on a snapshot of data in the system, manage membership, tolerate faults, and verify results. To detect rollback on Kanban data, DESEARCH summarizes an epoch-based snapshot and stores it on an append-only distributed log.

Finally, DESEARCH protects the privacy of queries with two techniques. To prevent leaks from access patterns, DESEARCH adapts an existing oblivious RAM library [102]. To resist volume side channels, DESEARCH returns the same amount of data for all search queries. It does so by equalizing the lengths of result entries. This approach does not reduce the performance or quality of the service because search engines need not display all of the content but rather a small snippet. As an analogy in the Web context, search engines like Google do not display the entirety of a Web site's content in the search result; instead, they typically display the URL of each site and a small text snippet.

We built a prototype of DESEARCH in $2,600$ lines of code, and have adapted it to work with Steemit (a decentralized social media service), and OpenBazaar (a decentralized e-commerce service). Our evaluation of DESEARCH on 1312 virtual machines across the wide-area network with variable network latency and executor failures shows that DESEARCH scales well as executors join the network, and can handle over 128 million requests per day. Checking the correctness of the displayed results is also affordable: users can verify results in under 1.2 seconds by consulting dedicated verifiers.

To summarize, the contributions of this paper are:

- The design of DESEARCH, the first decentralized search engine that allows any executors with a TEE to join and provide search functionality for decentralized services.
- A witness mechanism that organizes verifiable proofs from short-lived executors to form a global dataflow graph. Through these witnesses, DESEARCH offers fast verification for search queries.
- A prototype of DESEARCH built for Steemit [40] and OpenBazaar [28], and an evaluation of DESEARCH's performance and scalability.

While DESEARCH enables, for the first time, scalable, verifiable, and private search for existing decentralized services, it is not a viable replacement for traditional Web search engines (e.g., Google). Besides the obvious issue of scale, DESEARCH's target applications expose a single source of data (their underlying distributed log or proof of storage mechanism), which gives DESEARCH an anchor for its witness data structure. In contrast, the Web has no such single log, which would prevent DESEARCH from proving that all Web pages had been crawled and indexed. Nevertheless, we believe DESEARCH fills a crucial void.

The rest of the paper is organized as follows. Section 2 describes the motivation, the problem, and the threat model of DESEARCH, and discusses potential solutions. Section 3 provides an overview of DESEARCH, highlights DESEARCH's components, and explains how they work cooperatively. Section 4 presents how DESEARCH achieves a decentralized yet verifiable search; Section 5 introduces Kanban, a verifiable storage; Section 6 describes how DESEARCH provides oblivious search. Section 7 gives the implementation details. Section 8 evaluates DESEARCH in a local heterogeneous cluster and a geo-distributed environment. Finally, Section 9 discusses other aspects and Section 10 compares related work.

2 Motivation and problem statement

This section describes our motivation, target setting and threat model, and potential solutions that fall short.

2.1 Motivation

We give a brief overview of two representative decentralized services and the problems related to search that might arise in those systems.

Social Media. Steemit [40] is a social media platform that stores user-generated contents on the public Steem blockchain [39]. Although the raw data from the blockchain is tamper-proof, Steemit's front-end servers can censor or manipulate the search results before delivering them to users [41]. One can think of these servers as centralized curators for decentralized storage. These servers also know who searches for what, which may reveal users' interests and preferences.

Marketplaces. OpenBazaar [28] is an e-commerce marketplace built on top of a peer-to-peer network and storage. To help users search for items, OpenBazaar provides an API [5] for third-party search engines to crawl and index items. But existing search engines [2, 9, 34, 46] are opaque; they can bias results towards item listings that benefit them financially. For example, a listing owner could pay the search engine to promote its listing or hide other listings. Additionally, these engines can learn users' purchasing habits (and other information) from keywords and search histories.

In short, existing decentralized services currently lack trusted search that offers integrity and privacy. A decentralized search engine should have strict requirements on the visibility of the search queries and the correctness of the search results. Below, we formalize these goals.

2.2 Decentralized search

Consider a decentralized system where volunteers called *executors* together operate a search engine. *Users* want to search over some *source data* (e.g., data stored in a blockchain) using some *search algorithm*. Both the source data and the search algorithm are public and accessible to all users. For each search, a user sends *keywords* to the executors running the search algorithm and expects a *search result*—a ranked list of summaries for entries in the source data alongside pointers to the corresponding full entries.

After receiving the search results, users want to verify that the results are actually correct—that they are derived from executing the search algorithm on the latest source data and the provided keywords. Users also want to keep their searches private. In detail, the challenge is to design a decentralized search engine that meets these goals:

- **Integrity:** the search results should correspond to the correct execution of the published search algorithm on the most recent source data. We divide this into two properties: (1) *Execution integrity*, meaning that the search algorithm is faithfully executed on *some* source data and the output search results are ranked in the correct order. (2) *Data integrity*, meaning that the source data used is legitimate, up-to-date, and there are no missing or forged data records. In combination, these two properties prevent (undetectable) biased results and censorship.
- **Privacy:** the search engine (or any third party) should not leak the user's search query or the corresponding search results. There are two aspects to this. (1) *Content*: the user's query (search keywords) and the corresponding search results are never available in the clear to the search engine. (2) *Metadata*: the number and size of the messages exchanged by the user and the search engine is independent of the search results and the provided keywords.
- **Scalability:** executors that join the search engine should contribute meaningfully towards its capacity: more executors should result in higher search throughput.

Search Engine	Integrity		Privacy		Scalability
	Execution	Data	Content	Metadata	
YaCy [48]	○	○	○	○	●
Presearch [29]	●	●	○	○	●
The Graph [20]	●	●	○	○	●
IPSE [24]	●	○	○	○	●
BITE [89]	●	●	●	●	○
Rearguard [108]	●	○	●	◐	○
Oblix [91]	●	○	●	●	○
vChain [114]	●	●	○	○	○
GEM^2-Tree [117]	●	●	○	○	●
X-Search [92]	○	○	◐	○	○
CYCLOSA [98]	○	○	○	○	●
DESEARCH	●	●	●	●	●

FIGURE 2—Comparison between prior work and DESEARCH. X-Search [92] obfuscates query keywords but does not hide them. Rearguard [108] hides the index size but not the result size.

Figure 2 shows existing decentralized or private search engines. None of them meets all of our desired goals.

2.3 Potential approaches

How to provide integrity and/or privacy for an execution (for example, search) has been studied broadly. We list some approaches below (see more in Section 10).

Replication such as PBFT [67] and Ethereum [113] is one approach to build systems that can guarantee execution integrity within a given number of (Byzantine) faults. However, it requires performing the computation multiple times and traditional replication protocols do not provide privacy.

Another line of work [99, 105, 111] uses cryptography—such as fully homomorphic encryption (FHE) [74], secure multi-party computation (MPC) [116], and verifiable computation (VC) [62, 97, 105]—to provide execution integrity and/or privacy. Though promising, it remains an open problem to build a system that can support a complex enough search model and the large-scale datasets used by today's decentralized services.

Trusted execution environments (TEE) provide another approach to build systems that protect a sensitive execution from being tampered with or eavesdropped. However, existing TEEs either (1) have limited private memory (128 MB

per node); (2) lack memory encryption [55] which makes them vulnerable to physical attacks that are realistic in a decentralized setting; or (3) are susceptible to memory tampering [54, 84]. Although newly proposed TEEs [43] have expanded enclave memory to 1 TB, they relax the security guarantees (e.g., they are vulnerable to physical replay attacks) owing to the loss of integrity tree protection. Prior distributed frameworks like VC3 [104] and Ryoan [77] address many of the above shortcomings but are not designed for a decentralized environment. Their predefined computation graph is not a good fit for a dynamic environment where the source data and the set of executors is constantly in flux.

Finally, systems like Dory [71] that implement private search for a file system allow users to get pointers to objects that match certain keywords, but they do not implement features expected from a search engine such as finding non-exact matches, ranking results, or providing summaries.

DESEARCH uses a combination of new and existing techniques—TEEs, an append-only log, an epoch-based storage service, authenticated data structures (hash trees), and oblivious RAM—to address the challenges in Section 2.2.

2.4 Threat model

DESEARCH assumes a decentralized network where untrusted executors are operated by unknown parties, but are equipped with TEEs. We assume reliable TEEs with no microarchitectural or analog side channels [64, 110] or vulnerabilities to voltage changes or physical tampering [93]. We also assume the TEE manufacturers do not inject backdoors into TEEs, or share their private keys. Finally, we assume that the TEE remote attestation mechanism works as intended. While these assumptions are undeniably strong given existing TEE's track record, we expect this technology to mature over the years and for many of the current weaknesses to be addressed.

In DESEARCH, executors join the system and volunteer their TEEs (or they are paid or incentivized to do so, which is orthogonal). Executors can be malicious: they can deny services, modify the inputs that go into the TEE or the outputs that come out of it. They can also corrupt data outside the TEE's protected memory, or replay inputs and outputs. Moreover, executors observe memory accesses, and I/O patterns [65, 115] (either which memory is accessed or how much data/how many times).

DESEARCH requires that the data over which the decentralized service will offer search be stored in a publicly auditable source. This requirement boils down to ensuring that the data source has its own mechanism to check that the data added is not removed or tampered with (e.g., storing the data in a blockchain or IPFS [59]).

3 System overview

In this section we give the design principles of DESEARCH, which is a decentralized search engine for decentralized services and blockchain apps. We start by highlighting some of the challenges present when building a decentralized search engine that meets our requirements (§2.2). Some of these challenges stem from our decentralized environment, while others come from the limitations of today's TEEs and the dynamic nature of search.

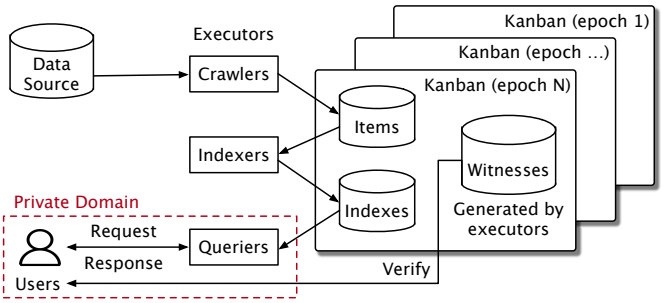


FIGURE 3—DESEARCH's architecture. DESEARCH obtains raw data from public decentralized services (e.g., Steem blockchain) as the data sources, and stores the intermediate data (i.e., items and indexes) on Kanban, a public append-only storage that creates snapshots periodically. DESEARCH executors generate witnesses along with the search pipeline. Privacy (§2.2) is offered for users in the query phase (within the dashed rectangle).

First, decentralization requires that executors be allowed to freely join and depart the system (§2.2), which means that executors can go offline unexpectedly. Thus, a standard search engine design with long-running tasks and stateful components is unfavorable, as one executor's leaving can heavily impact the service.

Second, today's TEE instances (DESEARCH uses Intel SGX) have limited memory (128MB or 256MB); working sets in excess of this limit require expensive paging mechanisms. Indeed, our experiments reveal that the latency incurred by paging far exceeds what is acceptable for a user-facing service like search. For example, when we run a search service for Steemit, which requires a 31 GB index, in a single optimized SGX instance, it takes 16 (resp., 65) seconds to respond to a single-keyword (resp., two-keyword) query. Recent work [95, 96] has explored ways to enlarge the trusted memory through software-based swapping (either via a managed runtime or compiler instrumentation), but those solutions leak the application's memory access pattern. As a result, we find that to achieve acceptable latency, it is necessary for the search engine's functionality to be split into small tasks that are processed by many SGX instances in parallel.

Third, search services are dynamic, and it is hard to track and verify the whole search process from crawling to query servicing. In particular, a search engine is unable to plan a computation graph (like in big-data [73, 104, 111] or machine-learning [79, 87] systems) as the arrival of new source data or user search queries is unpredictable, and the set of available executors is unknown a priori.

Overview. DESEARCH addresses these challenges by decomposing a search into a pipeline of short-lived tasks where

each executor is responsible for a single task at a time and operates only on a small portion of data. Executors are stateless. They fetch inputs and write outputs from and to a storage service named *Kanban*. Kanban is a cloud-based key-value store that provides high-availability and data integrity. We describe Kanban in detail in Section 5.

To track the completion of the dynamically executing tasks within the search pipeline, DESEARCH uses *witnesses* (§4), which are cryptographic summaries that capture the correct transfer of data among executors. A witness is also a proof of an executor's behavior, which allows users to verify their search results *ex post facto*.

Figure 3 shows the architecture of DESEARCH.

Search pipeline and Kanban. In DESEARCH, executors are categorized into four roles: crawlers, indexers, queriers, and masters (for simplicity, masters are omitted in Figure 3). The first three roles comprise a full search pipeline—*crawlers* fetch data from public sources (for example a blockchain [39, 94, 113] or P2P storage [28, 59]); *indexers* construct inverted indexes; *queriers* process queries, rank and build search results.

Instead of point-to-point communication, executors in the pipeline communicate through Kanban. Kanban also stores the data (items, indexes, and witnesses) generated by executors, and provides data integrity (but not confidentiality) by periodically creating snapshots of all current state and committing a digest of the snapshot to a public log (e.g., Ethereum [113] or Steem [39]). We call the time between two consecutive snapshots an *epoch*; we call the data corresponding to the beginning of an epoch, an *epoch snapshot*.

For privacy (§2.2), DESEARCH comprises public and private domains. Executors in the public domain access public data (like public source data) and produce shared information (like indexes). On the other hand, users' interactions with DESEARCH happen in the private domain, and their communication (for example, search requests and responses) are encrypted and kept secret.

Masters. *Masters* are the executors that provide crucial membership services: (1) a service that authenticates a TEE node who wants to join DESEARCH; (2) a job assignment service that coordinates the independent executors to form the search pipeline with minimal repeated work; (3) a key management service (KMS) that allows anyone to identify if an executor is a legitimate DESEARCH member. Regarding managing the KMS and task coordination, masters periodically (in the beginning of an epoch), release a list of public keys from legitimate executors on Kanban so that users can verify their signatures and communicate with them. This list includes the active nodes to ensure the service's availability. Masters hold the *root key* that serves as the identity of the DESEARCH system, allowing the public to recognize DESEARCH. We describe how the system is bootstrapped, how new masters join, and how the root key is generated in Section 9.

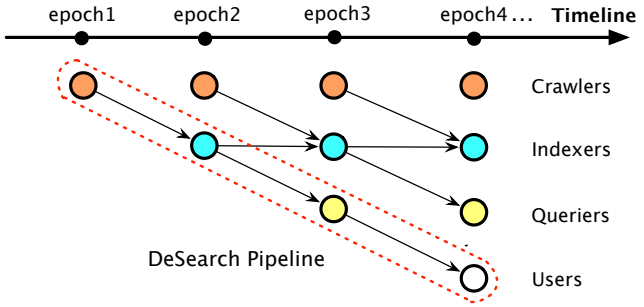


FIGURE 4—In DESEARCH, executors and users use data from the last epoch as inputs, which we call the *offset-by-one* strategy. For example, an indexer uses items of the last epoch to generate new indexes (denoted by oblique arrows), and may merge indexes from the prior epoch (denoted by horizontal arrows).

Workflow. DESEARCH's executors perform an ordinary search pipeline—crawling, indexing, and serving queries. In DESEARCH, data integrity (§2.2) is defined with respect to a particular epoch snapshot; DESEARCH uses an *offset-by-one* strategy where an executor always uses data from the last epoch in order to ensure that the flow of data is verifiable when expressed as a pipeline of tasks. Figure 4 shows an example of this process. Along every step of the pipeline, each executor generates a *witness*, a proof of what the executor has seen, has done, and how the data has been transferred. We discuss witnesses in Section 4.

To conduct a search, a user first retrieves a list of active queriers. This list is maintained by both masters and queriers: queriers update their status on the list with signed proofs specifying that they have seen the most-recent epoch; the legitimacy of the status proofs is verified by masters. Users know this by checking that the list is signed by masters.

With the active querier list, the user randomly selects one as the leader, and sends (encrypted) search keywords to the leader. The leader then seeks more peer queriers to collectively handle the request. That is, different queriers have different portions of indexes and together serve one user search request. The leader finally aggregates results by ranking based on relevance, and returns to the user a list of the k most relevant items. One item comprises a link (to the original content) and a content snippet that summarizes the item and often contains the searched keywords.

Together with the search results, the user also receives witnesses from queriers. The witnesses produced by the search pipeline (all of them) form a witness tree, which users can verify by starting from the witnesses received from the leader querier, traversing the tree, checking every node (witness), and confirming that the search has been correctly executed. We discuss this verification process next.

4 Verifiable search

In DESEARCH, search functions are outsourced to independent executors in a decentralized environment, and data is

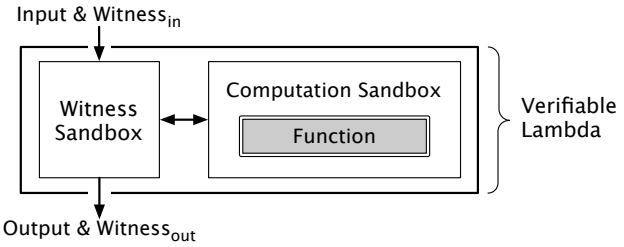


FIGURE 5—A verifiable lambda is composed of a witness sandbox and a computation sandbox. The function is one of: crawling, indexing, and querying. The witness$_{in}$ contains the hash for inputs and is from prior lambdas. The witness$_{out}$ is the witness generated by this lambda. The sandbox design is necessary because it isolates the witness processing from the function execution; the buggy or malicious function cannot tamper with the integrity of witness.

partitioned across executors. As a result, an executor only has a local view of its own computation and cannot protect the entire search pipeline even though it runs on a TEE assumed to be correct. Specifically, guaranteeing integrity (§2.2) has two parts: (a) ensuring that results are generated by the correct algorithm (*execution integrity*) despite intermediate data being transferred among executors through untrusted channels; and (b) ensuring that results are derived from the desired dataset without the absence of data records or the inclusion of bogus data (*data integrity*).

DESEARCH uses *verifiable lambdas* and *witnesses* to address challenge (a), and *Kanban* with epochs for challenge (b). We detail them below.

Verifiable lambda and witnesses. As stated earlier (§3), DESEARCH splits a search pipeline into small tasks. Each task is executed by a basic unit, called *verifiable lambda*. The concept of a verifiable lambda (short as lambda) is borrowed from serverless computing [36] and SGX sandboxing systems [77, 104]. A key difference is that DESEARCH's lambda requires a TEE enclave abstraction that yields a *witness* (we discuss it briefly, but it is a type of certificate of correct execution) after every computation, allowing the intermediate data to be verified and reused.

A lambda is composed of the two sandboxes shown in Figure 5: (1) a witness sandbox that validates the input upon loading, and generates a witness before delivering the result to the next lambda; (2) a computation sandbox which runs the main function in a self-contained execution environment that does not use any external services; the goal is to resist Iago attacks [68] (attacks in which a malicious OS causes the process to harm itself). All I/O activity of the computation sandbox must be routed to the witness sandbox, which logs the I/O data and produces a auditable record stored within the witness. This design isolates the buggy or malicious function execution from witness processing, so that the integrity of the witness is easier to reason about. For multiple inputs, a lambda can batch them to avoid generating many witnesses.

For a chain of lambdas, to ensure correct results, a seemingly straightforward solution is to encode a signed nonce in each execution of the pipeline. However, this approach does not work for search because a search pipeline often requires the data (such as crawling data and indexes) from multiple previous pipeline executions, and a signed nonce fails to capture the relationship between these multiple inputs and the output (critical for verifying data integrity such as data completeness). Hence, we introduce a certificate called *witness* for each lambda.

A lambda's witness is a tuple:

$$\Big\langle \Big[H(in1), H(in2), \cdots\Big],\ H(func),\ H(out) \Big\rangle_{signed}$$

that mirrors how an *output* is generated by performing a *function* over a list of *inputs*. The $H(in)$ and $H(out)$ are hashes of the input and output blobs, and $H(func)$ is the hash of the program binary that runs in this lambda. A witness is signed by the lambda, and anyone can verify the signature using DESEARCH's KMS (§3). A witness has a feature called "multiple input, single output" (MISO), which consumes multiple inputs and produces exactly one output. We find that search is a natural fit for MISO, as mutilple on-chain items yield an index, and multiple indexes serve a query.

Generally speaking, witnesses can be thought of as a *proof-carrying metadata* for a decentralized system, which explains how an output is being generated and with which piece of code, and enables a user to examine the computation process from the effect to the cause for a whole-pipeline verification. The notion of verifiable lambda and its witness mechanism are general enough to support other scenarios such as a decentralized and verifiable recommender system (§9).

Witness-based verification. All witnesses from a search process form a tree, which we call a *witness tree* (see an example in Figure 6). Users can verify their search results by traversing and checking the corresponding trees, the roots of which are the witnesses that users receive from queriers.

To check a single witness, a user first verifies whether the witness is signed by a legitimate executor and then checks if the hash of the executed function is as expected. This approach, combined with the integrity guarantees of the underlying TEE, ensures that the lambda which produced the witness faithfully executed the desired function.

Now consider the data transition between two adjacent lambdas in a search pipeline. The former lambda commits its output and a signed witness to Kanban; the latter lambda fetches one (or multiple) pair of data and witnesses from Kanban, checks their signatures, validates if the hash in the witness matches the data, and feeds the data to the computation function. A user can verify that the inputs of a latter lambda are indeed the outputs from a former lambda by checking whether $H(in)$ of the latter equals $H(out)$ from the former.

Finally, users check if data sources are genuine by checking whether $H(in)$ in the beginning (the crawling phase) is indeed a correct summary of the original data source. Users need

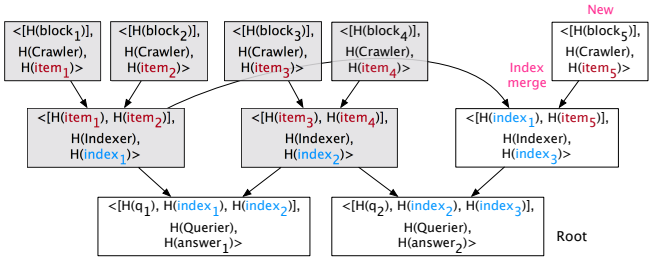


FIGURE 6—Two example witness trees. A rectangle represents a witness, edges represent data dependencies, and "H(...)" indicates hashes. This figure contains two search queries ("q_1" and "q_2"): q_1 happens first, and there is a merge update to the indexes ($index_3$ derives from $index_1$ and $item_5$), then q_2 happens. As a result, q_1 and q_2 share a subtree indicated in gray color.

to download the contents from the data source and calculate their hashes, which is a very expensive procedure (we address this issue shortly). If all the above checks pass, users confirm that their search results are faithfully produced because all steps—crawling from the data source, the intermediate data transferring between tasks, and each task in the search pipeline—are verified to be authentic and faithfully executed.

Providing efficient verification. The aforementioned verification process works in principle, but in practice, a performance challenge arises. To verify a search, a user would have to download all the witnesses, check all the signatures and hashes, and examine the data source. To lighten the burden of verification on the user side, DESEARCH uses *delegated verification*: users offload parts of their verifications to some executors dedicated for verification, which we call *verifiers*.

Beyond simplifying user-side verification, delegated verification also saves work by batching and deduplicating verifications from different users. This is based on an observation that serving different search requests uses a lot of shared indexes, hence the witnesses from the shared portion can be reused (as an example, see the gray subtree in Figure 6). Because of delegated verification, verifiers have the opportunity to batch many common witnesses, which they verify once for all. Finally, users only have to verify the final step—the querier phase's witness, significantly accelerating the verification (see delegated verification's speedup in §8.2). Because verifiers are only delegated to accelerate the verification of the witnesses in the public domain, it does not leak any information about particular users or their queries.

Data integrity. The above verification ensures that functions are executed as expected, but there is no guarantee that these functions see all data even if all functions are protected by TEEs. In fact, there is no definition of "all data" from a user's perspective because newly generated data takes time to be reflected in the search results. It is therefore unspecified what data must appear in any particular search result.

To define *data completeness* (a part of data integrity, §2.2) for searches, DESEARCH divides time into *epochs*, and executors write data to Kanban annotated with the current epoch (we elaborate on epochs in §5). We define a search that uses a complete set of data if each step (represented by witnesses) in a search pipeline (represented by a witness tree) uses *all* inputs in Kanban before the step's epoch, and each input is from the *most recent* epoch available. For example, a querier's task in epoch i satisfies our condition if the querier loads all indexes generated before epoch i, and the loaded portions of the indexes are from their latest version before epoch i.

To check the data integrity of a witness, verifiers first recognize the epoch when the witness was generated, then load the snapshot of Kanban immediately before that epoch (§5), and finally verify if the executor used all the up-to-date inputs. In practice, verifiers need not load the data in the snapshot; they load the metadata including data ids and their hashes.

5 Kanban

As mentioned in Section 4, Kanban is a storage system that provides high availability and data integrity. Kanban is hosted in public clouds for availability, but as a decentralized system, DESEARCH does not trust these cloud providers for correctness. Instead, DESEARCH creates snapshots of Kanban periodically for each epoch, called *epoch snapshots*, and commits their digests (hashes) to a public distributed log. This approach provides *epoch-based data integrity*: DESEARCH guarantees the integrity of all of the data included in a committed epoch. Of course, an alternative—using SGX-based cloud storage [100] for Kanban—works in principle, but current SGX cannot scale to a large trusted memory with integrity support (§3).

The rest of this section will introduce Kanban's usage and guarantees, and then discuss how DESEARCH uses Kanban as a storage and coordination service.

Kanban overview. Kanban serves two main purposes: storing data for the search pipeline (including items, indexes, and their witnesses), and enabling executors to communicate and coordinate their tasks. Kanban exposes APIs for each service.

As a data storage, Kanban exposes key-value-like APIs with `put(key, val)` and `get(key)`. Keys are constructed by the data types, epoch numbers, and chunk numbers. For example, "INDEX-#1000-v3" represents the 3rd chunk of the index for epoch 1000. Witnesses are also stored in the data storage, using the output hash $H(out)$ as the key. Thanks to the MISO feature of witnesses (§4), anyone can download a particular value from Kanban via `get()`, calculate its hash, and use this hash as the key to retrieve the corresponding witness in order to understand how the data was generated.

For communication, Kanban provides a mailbox for every executor with `send(mailbox, msg)` and `recv(mailbox)`, using the executor's public key as the mailbox address. Invoking `send()` allows an executor to submit a message to a specified mailbox, and `recv()` to download messages. All messages are encrypted using the mailbox owner's public key,

which can be obtained from the KMS (§3). Consequently, only owners can read message contents. For example, masters and queriers use the mailbox to negotiate the active querier list; this negotiation is encrypted in case Kanban maliciously attempts to forge or block particular executors.

Note that both storage and communication APIs are wrappers of the canonical key-value APIs, and Kanban can easily adapt to different underlying (cloud) storage systems. Our Kanban implementation uses Redis, a popular key-value store, as the underlying storage (see §7).

Epoch-based data integrity. Kanban requires executors to sign their submitted data (using Ed25519) to prevent data tampering or forgeries. Still, the underlying storage can *equivocate* and show different views of the data to different executors by omitting, rolling back, or forking the data. Detecting such divergences often requires clients (executors in our context) to synchronize out-of-band [83, 86], which is too expensive in a decentralized environment.

DESEARCH uses a loose synchronization approach: masters periodically synchronize Kanban's states with other executors. This loose synchronization works because of two observations. First, search engines are not supposed to instantly reflect newly generated data in search results because crawling and indexing takes time; as a (admittedly apples-to-oranges) comparison, Google crawls a site every 3 days or even longer [6]. Second, most of the tasks in the search pipeline are idempotent, so it is acceptable if two executors end up working on the same task. For example, it is safe for two crawlers to crawl the same data source, or two indexers to generate indexes for the same items, as the results are the same. Duplicate work sacrifices efficiency but not correctness.

To synchronize states with other executors, masters periodically create epoch snapshots of Kanban's data (excluding the data in the mailboxes which is used for coordination and is ephemeral), summarizes the snapshot as a digest, and commits the digest to a public append-only log (DESEARCH's implementation uses an EOS blockchain [11]). After the log accepts the digest, a new epoch is committed and is visible to all executors (assuming the public log is available).

DESEARCH guarantees *epoch-based data integrity*: for a committed epoch, all data included in this epoch is immutable and must be visible to all executors; otherwise, verification will fail. To see how DESEARCH guarantees this, if Kanban hides data from or returns stale data to an executor, the data integrity checks (§4) of this executor's witness will fail. This is because verifiers know the epoch of the witness (say epoch i) and the data this executor should have read (data in epoch $i-1$). If the witness missed any data or read some stale version, the verifier rejects. Before using one epoch for checking the data integrity, verifiers must ensure that the data (represented by their ids and hashes) in one epoch is consistent with the digest on the log. The verifier fetches all the data ids and hashes in one epoch, calculates their digest, and compares it with the digest on the public log.

Task coordination by epochs. DESEARCH's pipeline is coordinated through Kanban, which is based on epochs. An epoch is 15 minutes by default. Executors learn the current epoch number by querying the public log. As mentioned earlier, executors follow an offset-by-one strategy where they read data from the last committed epoch rather than the current epoch (see Figure 4). This guarantees that the DESEARCH pipeline only uses data that is already authenticated by the epoch-based digests on the log.

Our current implementation uses masters to assign jobs for crawlers and indexers. Masters also take charge of data collection in each epoch—they decide what data to include in the current epoch. DESEARCH requires other executors to put the current/correct epoch number in their outputs. If an executor in epoch i fails to do so, for example, it disregards the epoch or fails to submit its outputs on time (before masters commit epoch i), the data is discarded by the masters and the work is wasted. But this waste is acceptable as tasks are small.

Supporting multiple clouds. Though our current implementation only uses one cloud as Kanban's underlying storage, we plan to extend Kanban with multiple clouds for better availability [60], and more importantly, to lower the risk of vendor lockdown. With multiple clouds, executors write to all clouds and read from any one of them. Master executors are obligated to synchronize different clouds.

Data synchronization among clouds is challenging, which often requires running an expensive consensus protocol. However, by Kanban's epoch design, DESEARCH is able to do synchronization infrequently, only when committing an epoch. And if data diverges between clouds (note that data cannot be forged, due to signatures), master executors are in charge of merging the data. The key takeaway is that DESEARCH (or rather a search service) can tolerate infrequent synchronizations, so masters have plenty of leeway to orchestrate an epoch on which all clouds agree.

6 Oblivious search

DESEARCH guarantees integrity (§2.2) by leveraging witnesses and SGX. One might hope that SGX would also provide privacy for searches, as SGX supports confidential computing [26] and we assume that SGX works as designed (§2.4). However, DESEARCH's design in Section 4 leaks information: an adversary can learn users' keywords without breaking any of the guarantees of SGX. Below, we discuss concrete privacy violations, and then show how DESEARCH addresses these violations with ORAM and equalizing message lengths.

Privacy violations. To start a search, a user initiates an encrypted session (via TLS) to a querier selected from the active list of queriers published on Kanban by masters. Although the messages are encrypted/authenticated and computations are confidential (offered by SGX), adversaries can still conduct

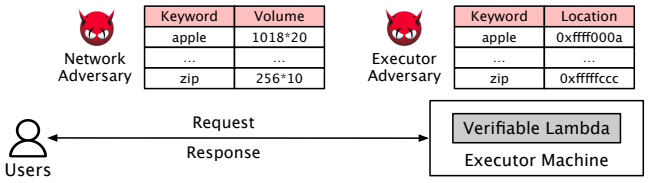


FIGURE 7—A search faces two privacy challenges: a network adversary can learn keyword information by monitoring request/response volumes, and an executor adversary can infer keywords by observing memory accesses.

two types of attacks (see examples in Figure 7).

First, an *executor adversary* that runs queriers can observe memory access patterns (both EPC and DRAM) to infer user keywords. Specifically, the adversary issues search requests with all possible keywords to the querier it hosts; by observing memory accesses, it can construct a dictionary that maps a keyword to a memory location [115]. Consequently, when a real user sends a query, the adversary can infer the user's keywords by observing which memory locations are accessed by looking them up in the dictionary.

Second, a *network adversary* can eavesdrop on the communication between users and queriers, and among queriers (which occurs when collaboratively serving one request). By monitoring the number of packets and their sizes, the adversary can learn information about keywords [66] because candidate lists for different keywords have different lengths, and returned items (e.g., a post on Steemit) also vary in size. Similar to an executor adversary, a network adversary can construct a dictionary that maps keywords to response lengths.

DESEARCH + ORAM. To prevent attacks from the executor adversaries, DESEARCH, like much prior work [89, 102], uses Oblivious RAM (ORAM) to hide memory access patterns. In particular, DESEARCH uses Circuit-ORAM [112] as follows: DESEARCH creates a key-value store where the keys are search keywords and values are lists of item ids. ORAM then guarantees that an executor adversary cannot learn anything about which object in the key-value store is being accessed (that is, which search keywords or item ids) from the memory accesses themselves. For a keyword that does not match any item, DESEARCH performs dummy accesses.

Note that Circuit-ORAM does not support concurrent accesses, so DESEARCH leverages multiple ORAM replicas for higher throughput. Specifically, DESEARCH encodes the underlying data in multiple ORAM instances, and accesses different instances to process queries. This is safe because we use ORAM exclusively for read-only workloads, and each instance is independent and has its own position map. This requires more storage space, but DESEARCH allows executors to make this trade-off.

Equalizing response lengths. Beyond ORAM, DESEARCH needs to avoid leaking information from the number of matched result items (count) and the length of each item (volume). We observe that results from search engines are highly regular: search results are displayed in multiple pages; each page contains a fixed number of items; and each item contains a link (e.g., URL) and a small content snippet highlighting the keywords. Therefore, DESEARCH equalizes result lengths by returning a fixed number of entries for each search request, and each entry has a 256-byte summary of the original contents, which we believe to be sufficient for most cases. For comparison, we find that over 80% of search results from Google are within 256 bytes. DESEARCH hides the counts and volume of keywords by padding search queries to the same length and limiting the number of keywords to 32 (the maximum supported by Google).

DESEARCH Components	Language	Total LoC
Openbazaar Crawler	Golang	178
Steemit Crawler	Python	190
Indexer	C++	701
Querier	C++	925
Master	C++	106
Verifier	C++	502
Search Engine Library (Sphinx [38])	C++	37,271 (+63)
ORAM Library (ZeroTrace [102])	C++	3,851 (+188)
Crypto Library (HACL⋆)	C/C++	4,071

FIGURE 8—Lines of code of each component in DESEARCH.

7 Implementation

System components. Figure 8 lists the components of DESEARCH's implementation. We implement our own crawlers that parse raw data from Steemit and OpenBazaar. The Steemit crawlers aggregate data from the Steem blockchain, and the OpenBazaar crawlers work as OpenBazaar peers to retrieve the online shopping items. Our indexer and querier borrow the tokenizer implementation from Sphinx v2.0 [38]. DESEARCH's indexer is implemented to support oblivious index access with ZeroTrace's ORAM implementation [102].

DESEARCH's verifiable lambda is built on Intel SGX SDK 2.13. We use Redis v6.2 as Kanban, and implement a Kanban protocol using Redis++ v1.2 [31] (a Redis client library). DESEARCH commits epoch snapshot digests to an EOS blockchain [11] testnet, which acts as an append-only log. Further, we deploy a dedicated smart contract that provides APIs to read and write the on-log data.

DESEARCH parameters. DESEARCH has several parameters that heavily influence the search performance. We elaborate on these parameters and our choices below.

We set up Circuit-ORAM with a bucket size of 2 (parameter Z) and set the stash size to 10 because they can achieve the best temporal and spatial efficiency. We use two independent ORAM instances, one for the index and another for the search result summaries, and overlap their operations to minimize the latency.

	Shard Size		
Metrics	**10K**	**100K**	**1M**
SGX Load Time	0.3s	2.9s	20.8s
ORAM Setup Time	2.0s	23.6s	267.7s
Worst Response Time	24ms	52ms	452ms
DRAM Required	121MB	285MB	2.34GB
SGX EPC Used	79MB	80MB	86MB

FIGURE 9—Execution time and memory usage of a DESEARCH querier under different shard sizes. SGX load time includes fetching the index/summary files from Kanban and in-enclave deserialization. ORAM setup time includes initializing two Circuit-ORAM instances for inverted index and search result summaries, respectively.

When the targeted dataset exceeds one executor's capacity (including both SGX EPC's and ORAM's capacity), DESEARCH has to split the dataset into shards. We experiment with several shard sizes (Figure 9), and choose 1M data items per shard because 1M-item-shard does not exceed SGX physical memory capacity and the response time is acceptable (within 1 second).

In DESEARCH, each epoch is set to 15min because most existing blockchains yield at least one block in this time frame [22]. It is also long enough for the master to summarize the epoch snapshot on Kanban. For shorter epochs, one could use an incremental hash function [58, 105] to create the digest incrementally throughout the epoch.

Side-channel defenses. DESEARCH use the Ed25519 implementation from a formally verified cryptographic library HACL⋆ v0.2.1 [120], which is resistant to digital (cache and timing) side channels [69, 82]. For ORAM block encryption, we choose AES-NI-based AES-128-GCM. AES-NI is purportedly side-channel resistant according to Intel [78]. Finally, we apply patch (commit f74c8a4) from Intel for SGX-OpenSSL [23] to mitigate hardware vulnerabilities [110].

Limitations. DESEARCH's current implementation only supports full-text search. The links of images, audio, or video, encoded in the texts may be hosted in other unverified servers. DESEARCH does not guarantee their integrity. In terms of privacy, DESEARCH implementation does not hide the frequency of ORAM accesses. Frequency smoothing techniques [76] can help at the cost of additional storage overhead.

8 Evaluation

Our evaluation answers the following questions:

- What is the overall performance of DESEARCH, in terms of end-to-end latency, throughput, and scalability? (§8.1)
- How long does it take to verify a search result? (§8.2)
- Does DESEARCH tolerate executor failures? (§8.3)

Experimental setup. We deploy DESEARCH on a small set of SGX-enabled desktop machines: three machines with 12-core Intel i7-8700, three with 8-core Intel i7-8559U, three with 8-core Intel i7-9700, and all nine machines have at least 8GB DRAM. These machines are connected by a 1Gbps local network. To simulate a large-scale decentralized environment, we also deploy DESEARCH on 1312 nodes of AWS EC2 VMs. Each node is an AWS t2.medium instance with 2-vCPU of Xeon E5-2676 and 4GB of DRAM. These nodes are spread across four geographic regions: Singapore (Asia), London (Europe), West Virginia (East America), Califonia (West America), and are connected through a wide-area network.

In the following experiments, we run DESEARCH in one of two modes: (1) normal mode, in which we run DESEARCH as-is; and (2) simulation mode, where we run DESEARCH under SGX SDK simulation mode and add ORAM latencies to each query instead of actually interacting with DESEARCH's ORAM implementation. We use the simulation mode for the large-scale scalability experiment ("Scalability" in §8.1) which requires hundreds to thousands of executors.

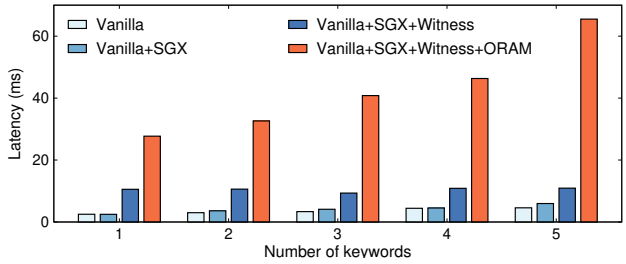
Datasets. We run DESEARCH on two datasets:

- *Steemit.* Steemit [40] is a decentralized blogging and social media service built upon the Steem blockchain [39]. DESEARCH crawlers constantly fetch the latest posts from the Steem blockchain and write them to Kanban. At the time of evaluation, we fetched a total of 81,681,388 posts, distributed across 952 epochs (nearly 10 days) leading to a 234GB dataset. The corresponding indexes contain 296K keywords (we omit 659 stopwords according to Google stopword list [42]). All of this results in 37.68GB crawling-phase witnesses and 6.25GB indexing-phase witnesses.
- *OpenBazaar.* OpenBazaar [28] is a decentralized e-commerce platform where individuals can trade goods without middlemen. The OpenBazaar frontend provides an API [5] for a customized search engine to update the shop contents by crawling IPFS [59], an append-only storage. DESEARCH crawls all OpenBazaar's shopping lists but ignores the ones that are removed by sellers (though they still exist in IPFS). At the time of evaluation, OpenBazaar has (on average) 21K listings per day, and there are 85K keywords in the indexes. Crawling and indexing witnesses are 10.26MB and 1.28MB in size.

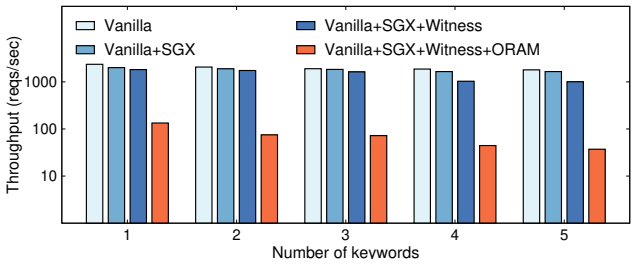
8.1 Serving performance

Querier bootstrap. To offer search for Steemit, DESEARCH splits the Steemit dataset (234GB) into 82 shards, each managing 1 million data items (Steemit posts). One Steemit querier serves one shard. It takes 333.5s for a querier to finish bootstrap, which includes establishing an SGX verifiable lambda, fetching the index and summary files from Kanban, and initializing ORAM instances. An OpenBazaar querier takes 25s to boot because it has a much smaller dataset—a querier serves 21K OpenBazaar data items.

Search throughput and latency. We run a DESEARCH querier for the two datasets on an SGX normal-mode machine and run clients on another machine. To capture a steady-state



(a) The 99th percentile latency of multi-keyword searches.



(b) The average throughput of multi-keyword searches.

FIGURE 10—Differential analysis of latency and throughput for multi-keyword searches. "Vanilla" is a native (unprotected) querier; "SGX" represents the SGX-based isolation; "Witness" represents the lambda confinement; "ORAM" represents the Circuit-ORAM protection. Note that the applied ORAM does not support concurrency while others use 8 threads.

performance, we warm up each querier by issuing 10,000 requests before the experiments. We measure the end-to-end throughput and latency by randomly searching from a list of 10,000 frequently appeared keywords (in each dataset).

For the Steemit dataset, DESEARCH has an average throughput of 133.8 requests/sec, and the 99 percentile end-to-end latency is 21.71ms. Although the measured throughput is modest, we expect that a decentralized network can achieve higher throughput as more executors join (see "Scalability" below). For OpenBazaar, DESEARCH's throughput is 581 requests/sec on average, and the 99 percentile latency is 9.19ms.

Overhead analysis. How do DESEARCH's techniques (including trusted hardware, the witness mechanism, and oblivious protections) affect search performance? To answer this question, we conduct a differential analysis for a Steemit querier, which manages 1M data items. Again, clients issue search queries by randomly picking keywords from a top-10,000 keyword list. But now we experiment with multi-keyword searches with up to 5 keywords. Figure 10 shows the average throughputs and the 99th percentile latency for different multi-keyword searches.

Figure 10 illustrates that putting a querier into SGX with DESEARCH's lambda enforcement decreases throughputs by 15.2% and adds 0.4% overhead to latency. This is because each request triggers *ecalls* (i.e., a context switch between SGX enclave and user-space) in DESEARCH's isolated sandboxes. In the future, we can optimize DESEARCH with the asynchronous call mechanism as studied in SCONE [56].

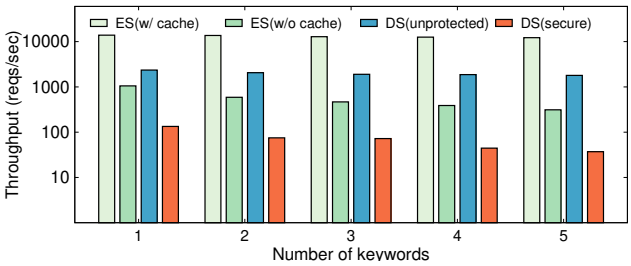


FIGURE 11—Throughput comparison between ElasticSearch and DESEARCH for multi-keyword searches. "ES" represents ElasticSearch and "DS" represents DESEARCH. Both "ES" and "DS (unprotected)" use multi-threading with 8 threads (CPU core number is 8)."DS (secure)" uses ORAM which does not support concurrent accesses, hence it does not benefit from multi-threading.

DESEARCH's witness generation imposes 7.6% overhead in throughput and 8ms in latency. Specifically, the witness generation consists of hashing the lambda's inputs and outputs with standard SHA-256, and signing this witness with Ed25519. Finally, the dominating overhead factor is Circuit-ORAM, which incurs 12.6× throughput degradation and increases latency by 4.5–11.9×. While this overhead is considerable, ORAM provides strong privacy guarantees against the executor adversary that controls the lambda.

Considering different multi-keyword searches, their latency is proportional to the number of keywords as searching multiple keywords requires fetching multiple rounds of index blocks from the Circuit-ORAM server. Historical statistics [19] show that 71.3% search queries do not exceed four keywords. Four-keyword ORAM-based searches in DESEARCH have a 46.3ms (99 percentile) latency, which is acceptable in a human interactive process.

Comparison with ElasticSearch. ElasticSearch [10] (or ES) is a popular search engine system that has been widely deployed. We compare DESEARCH with ES under an 1M-items dataset. We configure ES's Java runtime memory to 2.5GB, which is the maximum memory consumed by a DESEARCH Steemit querier. Figure 11 depicts the results.

By enabling caching, ES can achieve a throughput of 13.9K requests/sec, 4.8× faster than a DESEARCH implementation without integrity and privacy protection. But this is an unfair comparision because DESEARCH does not have caches (caches will break the security guarantee of ORAM). We further experiment with ES by disabling caching. Figure 11 shows that DESEARCH has higher throughputs than ES without caching because DESEARCH's implementation is simpler and has less functionality (e.g., complex item ranking policy). Finally, we compare ES with the full-fledged DESEARCH with SGX verifiable lambda and ORAM. As the ORAM operations limit the concurrency of the service, DESEARCH's throughput drops significantly.

We also observe that ES's throughput decreases mildly as the number of search keywords increases (around 11%

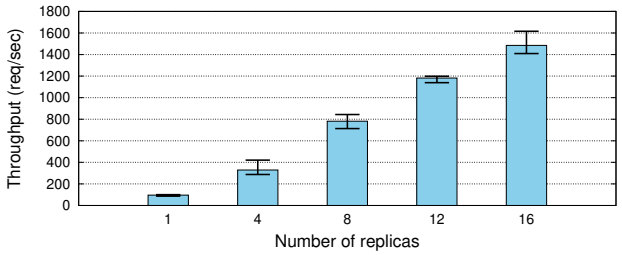


FIGURE 12—DESEARCH's throughputs for different number of replicas. Each replica consists of 82 queriers.

	Native	Delegated	Speedup
Execution Integrity			
Witness Download	517.0s	–	
Signature Verification	4h25min	–	
Data Integrity			
Witness Tree Verification	201.9s	–	
Final-Phase Verify			
Verifier Interaction	0s	1.0s	
Signature Verification	0s	0.2s	
Total Time	4h33min	1.2s	13,681x

FIGURE 13—User-side verification costs for a native (with an 8-core CPU) verification and a delegated verification.

decrease from single-keyword to five-keyword search), while the throughput of the full-fledged DESEARCH drops more rapidly (72.3% decrease). This is because multi-keyword searches require multiple rounds of ORAM item retrievals.

Scalability. To evaluate DESEARCH's scalability, we measure the overall throughput of DESEARCH with different number of *replicas*; each replica is a fully functional DESEARCH instance that consists of 82 Steemit queriers. We run DESEARCH in simulation mode, and use 82×16 2-core AWS EC2 VMs as servers to simulate a decentralized network of executors. Each virtual machine hosts a Steemit querier. From 4 to 16 replicas, we compose a geo-distributed setup where these replicas are equally deployed on four regions (i.e., Singapore, London, West Virginia, Califonia). We deploy clients on 8 other 96-core c5.24xlarge machines.

The results are shown in Figure 12. DESEARCH's throughputs increase horizontally with the number of replicas. With 16 replicas, DESEARCH can support 1484 requests/sec, that is 128 million requests per day.

8.2 Verification cost

A user verifies the final results by holding: (1) the digests of each epoch-based snapshot (retrieved from the public log), (2) executors' public keys, (3) witnesses from queriers (sent to users with search results) and other executors (stored on Kanban). Since the first two can be prefetched, a verification does not include fetching them. Figure 13 shows the *user-side* costs of native verifications and delegated verifications for a single-keyword search on the Steemit dataset.

In a native verification, a user verifies the search results on their own. To verify execution integrity, it takes 517s to download witnesses from Kanban, and 15698s to verify the signatures using Ed25519. The majority of the time is spent on checking crawlers' witnesses, as the number of witnesses is proportional to the number of items crawled. To verity data integrity, the user first ensures the witnesses from Kanban are consistent with the digests from the public log, and then breadth-first traverses the witness tree. This process takes 201.9s to complete.

In a delegated verification, the user sends the witnesses received from queriers to a verifier, and the verifier examines the execution and data integrity on behalf of the user. If the verifier accepts, the user only needs to verify the witnesses in the private domain, namely the witnesses from queriers. It takes 1.0s to interact with a verifier and 0.2s to verify the hashes and signatures of queriers' witnesses.

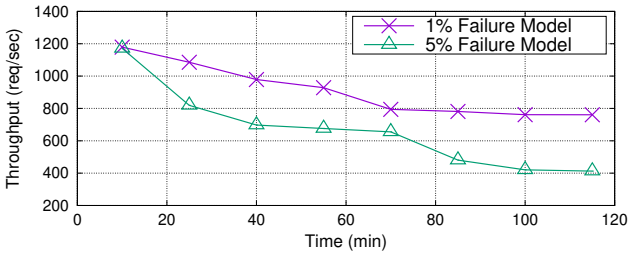


FIGURE 14—DESEARCH throughput changes under 1% and 5% per-epoch executor failure rate. The experiment uses Steemit dataset with 16 replicas and runs for 2 hours (8 epochs).

8.3 Fault tolerance

To understand how executor failures affect DESEARCH's performance, we run DESEARCH for Steemit with 16 replicas and 20 shards (4×20 queriers), and randomly kill a certain number of queriers in each epoch. In particular, we have two failure workloads: one kills 1% of the current available queriers in each epoch (15 min), and the other kills 5% per epoch. If a querier does not respond in 1 second, clients will issue the request to other queriers. Note that our experiment ensures that the remaining online queriers always comprise a complete dataset, otherwise all queries will fail (since they will only be able to provide partial results).

We run the experiments under these two workloads for 2 hours (8 epochs) each. Figure 14 shows the results. When 1% queriers fail per epoch, we observe that DESEARCH's throughput drops from 1,179 requests/sec to 761 requests/sec after 2 hours, a 35.4% decrease. As a comparison, in the last epoch, DESEARCH loses 7.7% ($= 1 - 0.99^8$) of the initial queriers. The throughput degrades significantly because every search has to get responses from all shards to make up a full result, but killed nodes do not perfectly distribute across shards. Hence, shards with fewer available queriers become a performance bottleneck. In the workload with 5% failure rate

per epoch, the throughput drops from 1,171 requests/sec to 412 requests/sec in 8 epochs (a 64.8% throughput degradation), while 33.7% ($= 1 - 0.95^8$) queriers are killed.

9 Discussion

Incentive model. As with existing decentralized systems [94, 113], DESEARCH relies on volunteers to offer service availability. To encourage TEE owners to join in DESEARCH, we discuss a possible incentive model, inspired by Teechain [85], where cryptocurrency can be securely transferred from a blockchain to a TEE enclave. We observe that current Steem Dollars (SBD) flow from readers to mostly popular post authors as rewards. Our incentive model, instead, transfers SDB from readers $\rightarrow$ queriers $\rightarrow$ indexers $\rightarrow$ crawlers $\rightarrow$ authors, where data needs to be paid for the usage. Our witness is a natural fit for proving the usage. A higher contribution of TEE computing power would then translate into a higher cryptocurrency reward.

System bootstrap. To bootstrap DESEARCH, any TEE executor can become a master, as long as it downloads the code of master and runs the code within an attestable enclave. The master's initialization will generate a pair of public/private keys. The former is recorded on the append-only log for public availability, and the latter is kept within the enclave, being DESEARCH's root key. Any party can use the TEE's remote attestation mechanism to verify the root key's genuineness. The first master then registers itself in the active list on Kanban. A set of executors (say, 21, a DESEARCH parameter) are required to join and serve as masters to start the first epoch. DESEARCH masters run independently and form a decentralized network to avoid a single point of failure. Masters can change over time via periodic selections (e.g., BFT sortition protocols [75]). Executors that join at a later time after the masters network is active will simply be assigned other roles.

Further, to resist possible supply-chain attacks [37], DESEARCH can use reproducible builds [33] to ensure verifiability from source code to lambda images.

Beyond search. DESEARCH, as a general framework, is not limited to search. As an example, it is intuitive to extend DESEARCH to a verifiable recommender system for OpenBazaar by replacing querier's ranking function with a recommendation function. A user can verify that the advertisements are chosen based on their interests, without opaque manipulation. Also, DESEARCH can be used as a "watchdog" for other search services. Users can continue to use an existing search engine, and cross-validate the results with DESEARCH.

10 Related Work

Decentralized search engines. There are several prior efforts in building decentralized search services. For example, YaCy [48] is a peer-to-peer distributed search engine since 2004. It enables decentralized index generation and supports shared indexes among peers. YaCy assumes that peers are honest, which might not be true in an open environment. Presearch [29] leverages a blockchain to provide decentralized search, but inherits blockchain's characteristics, including duplicated computations and no privacy guarantees. The Graph [20] and PureStake [30] are indexing services for decentralized storage, but neither provides privacy, and PureStake does not provide integrity either.

Verifiable search for decentralized services. There is an increasing interest in providing verifiable search for decentralized services, due to the diversified usages of decentralized applications (see Figure 1). For verifiable blockchain searches, vChain [114] adopts authenticated data structures (ADS) while GEM^2-Tree [117] explores on-chain indexes. Compared with vChain and GEM^2-Tree which only support range-based searches, DESEARCH provides full-text search and offers verifiability via witness-based dataflow tracking.

IPSE [24] provides search over IPFS [59] (a decentralized storage) and provides hash-based content verifiability. Freenet [15] is an anonymous file-sharing network (similar to BitTorrent) but is not search-oriented. Compared to DESEARCH, IPSE and Freenet lack data integrity (§2.2), which is troublesome because incomplete data sources can make the search vulnerable to censorship [41].

Private search with TEE. TEE is a hot topic for providing private search. To hide users' search intention, X-Search [92] uses a cloud-side TEE proxy while Cyclosa [98] adopts browser-side TEE proxies. X-Search and Cyclosa are metasearch engines (a proxy between users and a search engine) that reveal query keywords and results to search providers, whereas DESEARCH is a complete search engine that provides query and result privacy.

A long line of prior works (Opaque [119], OCQ [72], ZeroTrace [102], Oblix [91], Obliviate [51], etc.) have explored TEE and ORAM combination to protect search. Similar efforts have been made by combining symmetric searchable encryption and TEEs (e.g., Rearguard [108]), or private information retrieval and TEE (e.g., SGX-IR [106]). While DESEARCH uses similar primitives, DESEARCH's architecture results in the first fully built decentralized system to serve searches on real-world datasets with integrity and privacy.

Secure big-data systems with TEE. Many prior systems use TEEs for big-data computation [61, 72, 77, 101, 104, 119]. VC3 [104] secures a map-reduce framework with TEE, and Opaque [119] protects SQL queries for Spark SQL. Unlike the setting of VC3 and Opaque, DESEARCH faces a dynamic computation graph because the set of live executors is constantly changing given our decentralized environment. DESEARCH therefore employs an epoch-based snapshot and witnesses mechanism to ensure data and execution integrity. Ryoan [77] provides distributed sandboxes for private data computation. Compared with Ryoan, DESEARCH offers publicly verifiable witnesses for reusable intermediate data and

effectively reduces the verification cost (see Section 8.2).

TEE and serverless computing. The notion of verifiable lambda is inspired by serverless computing. DESEARCH extends this notion from centralized cloud-based computing to decentralized computing. S-FaaS [52], T-FaaS [63], Clemmys [109] are TEE-based serverless systems that protect serverless workloads with TEE. Instead, DESEARCH utilizes epoch-based Kanban and TEE-generated witnesses to maintain and verify the state of the (stateless) TEE lambdas.

New decentralized systems. Recently, new architectures of decentralized systems have been proposed to address the limitations (low-throughput, resource waste, lack of privacy) of conventional decentralized ledgers or blockchains. Algorand [75] and Blockene [103] propose new consensus protocols to achieve high-throughput. Omniledger [80] and Protean [53] introduce sharding to scale out the blockchain. Similarly, DESEARCH shards executors to different roles, and offloads states to Kanban to achieve high scalability.

Other TEE-based systems for decentralized services. TEEs have been used to build a provable blockchain oracle [118], off-chain smart contracts [70], Bitcoin fast payment channel [85] and lightweight clients [89], and online-service secure sharing [88]. They share the same goal towards shaping a better decentralized world but differ from DESEARCH in their target functionality.

11 Conclusion

DESEARCH is the first decentralized search engine to support existing decentralized services, while guaranteeing verifiability owing to its *witness* mechanism and offering privacy for query keywords and search results. DESEARCH achieves good scalability and minimizes fault disruptions through a novel architecture that decouples the decentralized search process into a pipeline of verifiable lambdas and leverages a global and highly available Kanban to exchange messages between lambdas. We implement DESEARCH on top of Intel SGX machines and evaluate it on two decentralize systems: Steemit and OpenBazaar. Our evaluation shows that DESEARCH can scale horizontally with the number of executors and can achieve the stringent subsecond latency required for a search engine to be widely usable.

DESEARCH 's source code will be released at:
`https://github.com/SJTU-IPADS/DeSearch`

Acknowledgments

We thank the anonymous reviewers of OSDI 2020 and OSDI 2021 for their helpful and constructive comments, and our shepherd Marko Vukolic for his guidance. We also thank Sajin Sasy for helpful discussion about ORAM. This work was supported in part by National Key Research and Development Program of China (No. 2020AAA0108500), China National Natural Science Foundation (No. 61972244, U19A2060, 61925206), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 19511121100). Sebastian Angel was funded by NSF Award CNS-2045861 and DARPA contract HR0011-17-C0047. Cheng Tan was funded by AFOSR FA9550-18-1-0421 and NSF CNS-1514422. Yubin Xia (xiayubin@sjtu.edu.cn) is the corresponding author.

References

[1] Augur. `https://www.augur.net/`.
[2] Blockstamp openbazaar explorer. `https://bazaar.blockstamp.market/`.
[3] Cam4 live-streaming adult site exposed 7tb records publicly. `https://www.2-spyware.com/cam4-live-streaming-adult-site-exposed-7tb-records-publicly`.
[4] Censorship by google. `https://en.wikipedia.org/wiki/Censorship_by_Google`.
[5] Content discovery on OpenBazaar. `https://openbazaar.org/blog/decentralized-search-and-content-discovery-on-openbazaar/`.
[6] Crawl stats report - search console help - google support. `https://support.google.com/webmasters/answer/9679690`.
[7] Data-enriched profiles on 1.2b people exposed in gigantic leak. `https://threatpost.com/data-enriched-profiles-1-2b-leak/150560/`.
[8] Dtube. `https://d.tube`.
[9] Duo search is a search engine for openbazaar. `https://bitcoinist.com/duo-search-is-a-search-engine-for-openbazaar/`.
[10] Elasticsearch. `https://www.elastic.co/`.
[11] Eosio blockchain software & services. `https://eos.io/`.
[12] Etoro. `https://stocks.etoro.com/`.
[13] Facebook is illegally collecting user data, court rules. `https://thenextweb.com/facebook/2018/02/12/facebook-is-illegally-collecting-user-data-court-rules/`.
[14] Facebook suspends cambridge analytica for misuse of user data, which cambridge denies. `https://www.cnbc.com/2018/03/16/facebook-bans-cambridge-analytica.html`.
[15] Freenet. `https://freenetproject.org/`.
[16] Giving social networking back to you - the mastodon project. `https://joinmastodon.org/`.
[17] Golem network. `https://golem.network/`.
[18] Google faces antitrust investigation by 50 us states and territories. `https://www.theguardian.com/technology/2019/sep/09/google-antitrust-investigation-monopoly`.
[19] Google search statistics and facts 2020. `https://firstsiteguide.com/google-search-stats/`.
[20] The graph is an indexing protocol for querying networks like ethereum and ipfs. `https://thegraph.com`.
[21] Graphite docs. `https://www.graphitedocs.com/`.
[22] How many bitcoins are mined everyday? `https://www.buybitcoinworldwide.com/how-many-bitcoins-are-there/`.

[23] Intel® software guard extensions ssl. https://github.com/intel/intel-sgx-ssl.
[24] Ipfs search. https://ipfs-search.com/#/search.
[25] Matrix - an open network for secure, decentralized communication. https://matrix.org/.
[26] Microsoft azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/.
[27] The monopoly-busting case against google, amazon, uber, and facebook. https://www.theverge.com/2018/9/5/17805162/monopoly-antitrust-regulation-google-amazon-uber-facebook.
[28] OpenBazaar. https://openbazaar.org/.
[29] Presearch is a decentralized search engine, powered by the community. https://www.presearch.io/.
[30] Purestake: Blockchain infrastructure for proof of stake networks. https://www.purestake.com/.
[31] Redis client written in c++. https://github.com/sewenew/redis-plus-plus.
[32] Report: 267 million facebook users ids and phone numbers exposed online. https://www.comparitech.com/blog/information-security/267-million-phone-numbers-exposed-online/.
[33] Reproducible Builds. https://reproducible-builds.org/.
[34] Searchbizarre. https://searchbizarre.com/.
[35] Searching the blockchain (indexer v2) - algorand developer docs. https://developer.algorand.org/docs/features/indexer/.
[36] Serverless computing. https://en.wikipedia.org/wiki/Serverless_computing.
[37] The solarwinds orion breach, and what you should know. https://blogs.cisco.com/security/the-solarwinds-orion-breach-and-what-you-should-know.
[38] Sphinx: Open source search server. http://sphinxsearch.com/.
[39] Steem. https://steem.com/.
[40] Steemit. https://steemit.com/.
[41] Steemit censoring users on immutable social media blockchain's front-end. https://cryptoslate.com/steemit-censoring-users-immutable-blockchain-social-media/.
[42] Stop words - words ignored by search engines. https://www.link-assistant.com/seo-stop-words.html.
[43] Supporting intel sgx on multi-socket platforms. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms.html.
[44] A timeline of facebook's privacy issues — and its responses. https://www.nbcnews.com/tech/social-media/timeline-facebook-s-privacy-issues-its-responses-n859651.
[45] Tital-advancing the era of accelerated computing. https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/.
[46] Welcome to bazaar dog, your scrappy open bazaar search provider. https://www.bazaar.dog/.
[47] Wix.com: Free website builder. https://www.wix.com/.
[48] Yacy - decentralized search engine. https://yacy.net/.
[49] Zeronet. https://zeronet.io/.
[50] Zipcall.io. https://meet.questo.ai/.
[51] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.
[52] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. S-faas: Trustworthy and accountable function-as-a-service using intel SGX. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, 2019.
[53] Enis Ceyhun Alp, Eleftherios Kokoris-Kogias, Georgia Fragkouli, and Bryan Ford. Rethinking general-purpose decentralized computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.
[54] AMD. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/.
[55] ARM. Arm TrustZone Technology. https://developer.arm.com/ip-products/security-ip/trustzone.
[56] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, et al. Scone: Secure linux containers with intel sgx. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
[57] Randy Baden, Adam Bender, Neil Spring, Bobby Bhattacharjee, and Daniel Starin. Persona: An online social network with user-defined privacy. In *Proceedings of the ACM SIGCOMM Conference*, 2009.
[58] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 1997.
[59] Juan Benet. Ipfs - content addressed, versioned, p2p file system. *ArXiv*, abs/1407.3561, 2014.
[60] Alysson Neves Bessani, Miguel P. Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2011.
[61] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnés, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.
[62] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.
[63] Stefan Brenner and Rüdiger Kapitza. Trust more, serverless. In *Proceedings of the ACM International Conference on Systems and Storage (SYSTOR)*, 2019.
[64] Jo Van Bulck, M. Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In

USENIX Security Symposium, 2018.

[65] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the USENIX Security Symposium*, 2017.

[66] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[67] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[68] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[69] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[70] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[71] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. Dory: An encrypted search system with distributed trust. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[72] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Oblivious coopetitive analytics using hardware enclaves. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2020.

[73] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *Proceedings of the USENIX Security Symposium*, 2015.

[74] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009.

[75] Y. Gilad, Rotem Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[76] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *Proceedings of the USENIX Security Symposium*, 2020.

[77] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: a distributed sandbox for untrusted computation on secret data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[78] Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. `https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html`.

[79] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[80] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[81] Ziliang Lai, Chris Liu, Eric Lo, Ben Kao, and Siu-Ming Yiu. Decentralized search on decentralized web. *ArXiv*, abs/1809.00939, 2019.

[82] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the USENIX Security Symposium*, 2017.

[83] Jinyuan Li, M. Krohn, David Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[84] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected I/O operations in amd's secure encrypted virtualization. In Nadia Heninger and Patrick Traynor, editors, *Proceedings of the USENIX Security Symposium*, 2019.

[85] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter R. Pietzuch. Teechain: a secure payment network with asynchronous blockchain access. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[86] Prince Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[87] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[88] Sinisa Matetic, Moritz Schneider, Andrew Miller, Ari Juels, and Srdjan Capkun. Delegatee: Brokered delegation using trusted execution environments. In *Proceedings of the USENIX Security Symposium*, 2018.

[89] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. BITE: bitcoin lightweight client privacy using trusted execution. In *Proceedings of the USENIX Security Symposium*, 2019.

[90] P. Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, 2002.

[91] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[92] Sonia Ben Mokhtar, Antoine Boutet, Pascal Felber, Marcelo Pasin, Rafael Pires, and Valerio Schiavoni. X-search: revisiting private web search using intel SGX. In *Proceedings of the*

ACM/IFIP/USENIX International Middleware Conference, 2017.

[93] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[94] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.

[95] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless OS services for SGX enclaves. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2017.

[96] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. Cosmix: A compiler-based system for secure memory instrumentation and execution in enclaves. In *USENIX ATC*, 2019.

[97] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

[98] Rafael Pires, David Goltzsche, Sonia Ben Mokhtar, Sara Bouchenak, Antoine Boutet, Pascal Felber, Rüdiger Kapitza, Marcelo Pasin, and Valerio Schiavoni. CYCLOSA: decentralizing private web search through sgx-based browser extensions. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2018.

[99] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[100] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[101] Do Le Quoc, Franz Gregor, Jatinder Singh, and Christof Fetzer. Sgx-pyspark: Secure distributed data analytics. In *International World Wide Web Conference (WWW)*, 2019.

[102] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. Zerotrace : Oblivious memory primitives from intel SGX. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[103] Sambhav Satija, Apurv Mehra, Sudheesh Singanamalla, Karan Grover, Muthian Sivathanu, Nishanth Chandran, Divya Gupta, and Satya Lokam. Blockene: A high-throughput blockchain over mobile devices. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[104] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[105] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. Proving the correct execution of concurrent services in zero-knowledge. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[106] Fahad Shaon and Murat Kantarcioglu. Sgx-ir: Secure information retrieval with trusted processors. In *DBSec*, 2020.

[107] Ion Stoica, Robert Tappan Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference*, 2001.

[108] Wenhai Sun, Ruide Zhang, Wenjing Lou, and Y. Thomas Hou. REARGUARD: secure keyword search using trusted hardware. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2018.

[109] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. Clemmys: towards secure remote execution in faas. In *Proceedings of the ACM International Conference on Systems and Storage (SYSTOR)*, 2019.

[110] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[111] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2019.

[112] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[113] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151, 2014.

[114] Cheng Xu, Ce Zhang, and Jianliang Xu. vChain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the ACM SIGMOD Conference*, 2019.

[115] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[116] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1982.

[117] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. GEM^2-Tree: A gas-efficient structure for authenticated range queries in blockchain. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2019.

[118] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[119] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca A. Popa, Joseph Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[120] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.