PDoT: Private DNS-over-TLS with TEE Support

YOSHIMICHI NAKATSUKA, University of California, Irvine
ANDREW PAVERD*, Microsoft
GENE TSUDIK, University of California, Irvine

3

Security and privacy of the Internet Domain Name System (DNS) have been longstanding concerns. Recently, there is a trend to protect DNS traffic using Transport Layer Security (TLS). However, at least two major issues remain: (1) How do clients authenticate DNS-over-TLS endpoints in a scalable and extensible manner? and (2) How can clients trust endpoints to behave as expected? In this article, we propose a novel Private DNS-over-TLS (PDoT) architecture. PDoT includes a DNS Recursive Resolver (RecRes) that operates within a Trusted Execution Environment. Using *Remote Attestation*, DNS clients can authenticate and receive strong assurance of trustworthiness of PDoT RecRes. We provide an open source proof-of-concept implementation of PDoT and experimentally demonstrate that its latency and throughput match that of the popular Unbound DNS-over-TLS resolver.

CCS Concepts: • **Security and privacy → Web protocol security**; *Hardware-based security protocols*; *Network security*; *Privacy protections*;

Additional Key Words and Phrases: Domain name system, privacy, trusted execution environment

ACM Reference format:
Yoshimichi Nakatsuka, Andrew Paverd, and Gene Tsudik. 2021. PDoT: Private DNS-over-TLS with TEE Support. *Digit. Threat.: Res. Pract.* 2, 1, Article 3 (February 2021), 22 pages.
https://doi.org/10.1145/3431171

1 INTRODUCTION

The Domain Name System (DNS) [41] is a global distributed system that translates human-readable domain names into IP addresses. It has been deployed since 1983 and, throughout the years, DNS privacy has been a major concern.

In 2015, Zhu et al. [52] proposed a DNS design that runs over Transport Layer Security (TLS) connections [19]. DNS-over-TLS protects privacy of DNS queries and prevents man-in-the-middle attacks against DNS responses. Reference [52] also demonstrated the practicality of DNS-over-TLS in real-life applications. Several open source

*Work done partially while visiting the University of California, Irvine, as a US-UK Fulbright Cyber Security Scholar.

The first and third authors were supported in part by NSF Award Number:1840197, titled: “CICI: SSC: Horizon: Secure Large-Scale Scientific Cloud Computing.” The first author was also supported by The Nakajima Foundation. The second author was supported by a US-UK Fulbright Cyber Security Scholar Award.

Authors’ addresses: Y. Nakatsuka and G. Tsudik, University of California, CA 92697, United States of America; emails: {nakatsuy, gene.tsudik}@uci.edu; A. Paverd, Microsoft Research Ltd, 21 Station Road, Cambridge CB1 2FB, United Kingdom; email: andrew.paverd@microsoft.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

2576-5337/2021/02-ART3 $15.00
https://doi.org/10.1145/3431171

recursive resolver (RecRes) implementations, including Unbound [34] and Knot Resolver [18], currently support DNS-over-TLS. In addition, commercial support for DNS-over-TLS has been increasing, e.g., Android P devices [25] and Cloudflare's 1.1.1.1 RecRes [12]. However, despite attracting interest in both academia and industry, some problems remain.

The first challenge is that clients need a way to authenticate the RecRes. Certificate-based authentication is natural for websites, since the user (client) knows the URL of the desired website and the certificate securely binds this URL to a public key. However, this approach cannot be used for a DNS RecRes, because the RecRes does not have a URL or any other unique long-term user-recognizable identity that can be included in the certificate. One way to address this issue is to provide clients with a white-list of trusted RecRes-s' public keys. However, this is neither scalable nor maintainable, because the white-list would have to include all possible RecRes operators, ranging from large public services (e.g., 1.1.1.1) to small-scale providers, e.g., a local RecRes provided by a coffee-shop.

Even if the RecRes can be authenticated, the second major issue is the lack of means to determine whether a given RecRes is trustworthy. For example, even if communication between client stub (client) and RecRes, and between RecRes and the name server (NS) is authenticated and encrypted using TLS, the RecRes must decrypt the DNS query to resolve it and contact the relevant NS-s. This allows the RecRes to learn unencrypted DNS queries, which poses privacy risks of a malicious RecRes misusing the data, e.g., profiling users or selling their DNS data. Some RecRes operators go to great lengths to assure users that their data is private. For example, Cloudflare promises, "We will never sell your data or use it to target ads" and goes on to say, "We've retained KPMG to audit our systems annually to ensure that we're doing what we say" [12]. On March 31, 2020, Cloudflare released the results of the privacy examination of its 1.1.1.1 DNS resolver [15]. The report is publicly available [13] and it assures that during the time of inspection (conducted from February 1, 2019, to October 31, 2019), 1.1.1.1 was configured in such a way that it supports the commitment given by Cloudflare. While this report gives some guarantee that the 1.1.1.1 resolver is honoring the client's privacy, there are three drawbacks of this method. First, it only provides a guarantee to a particular point in time. We cannot be certain whether the privacy promise was kept before and after the inspection. Second, it takes a long time to conduct the inspection and release the report. 1.1.1.1 was announced on April 1, 2018 [14], *two* years before the privacy report was released. Third, this method requires users to trust the auditor and can only be used by operators who can afford an auditor. Although 1.1.1.1 may be one of the famous public resolvers that support DNS-over-TLS and DNS-over-HTTPS, a recent study shows that there are many smaller organizations that also provide such resolvers [36]. Since these organizations cannot affort to be inspected, it is more difficult for them to convince their customers that they are protecting their privacy.

In this article, we use Trusted Execution Environments (TEEs) and Remote Attestation (RA) to address these two problems. By using RA, the identity of the RecRes is no longer relevant, since clients can check what software a given RecRes is running and make trust decisions based on how the RecRes behaves. RA is one of the main features of modern hardware-based TEEs, such as Intel Software Guard Extensions (SGX) [39] and ARM TrustZone [6]. Such TEEs are now widely available, with Intel CPUs after the seventh generation supporting SGX, and ARM Cortex-A CPUs supporting TrustZone. TEEs with RA capability are also available in cloud services, such as Microsoft Azure [40]. Specifically, our contributions are as follows:

- We design a Private DNS-over-TLS (PDoT) architecture, the main component of which is a privacy-preserving RecRes that operates within a commodity TEE. Running the RecRes inside a TEE prevents even the RecRes operator from learning clients' DNS queries, thus providing query privacy. Our RecRes design addresses the authentication challenge by enabling clients to trust the RecRes based on how it behaves, and not on who it claims to be (see Section 4).
- We implement a proof-of-concept PDoT RecRes using Intel SGX and evaluate its security, deployability, and performance. All source code and evaluation scripts are publicly available [32]. Our results show

that PDoT handles DNS queries without leaking information while achieving sufficiently low latency and offering acceptable throughput (see Sections 5 and 7).
- To quantify privacy leakage via traffic analysis, we performed an Internet measurement study. It shows that 94.7% of the top 1,000,000 domain names can be served from a *privacy-preserving* NS that serves at least two distinct domain names, and 65.7% from a NS that serves 100+ domain names (see Section 8).

2 BACKGROUND

2.1 Domain Name System

DNS is a distributed system that translates host and domain names into IP addresses. DNS includes three types of entities: *Client Stub* (client), RecRes, and NS. Client runs on end-hosts. It receives DNS queries from applications, creates DNS request packets, and sends them to the configured RecRes. Upon receiving a request, RecRes sends DNS queries to NS-s to resolve the query on client's behalf. When NS receives a DNS query, it responds to RecRes with either the DNS *record* that answers client's query, or the IP address of the next NS to contact. RecRes thus recursively queries NS-s until the record is found or a threshold is reached. The NS that holds the queried record is called: Authoritative Name Server (ANS). After receiving the record from ANS, RecRes forwards it to client. It is common for RecRes to cache records so that repeated queries can be handled more efficiently.

2.2 Trusted Execution Environment

A TEE is a security primitive that isolates code and data from privileged software such as the OS, hypervisor, and BIOS. All software running outside TEE is considered untrusted. Only code running within TEE can access data within TEE, thus protecting confidentiality and integrity of this data against untrusted software. Another typical TEE feature is RA, which allows remote clients to check precisely what software is running inside TEE.

One recent TEE example is Intel SGX, which enables applications to create isolated execution environments called *enclaves*. The CPU enforces that only code running within an enclave can access that enclave's data. SGX also provides RA functionality.

Memory Security. SGX reserves a portion of memory called the Enclave Page Cache (EPC). It holds 4-KB pages of code and data associated with specific enclaves. EPC is protected by the CPU to prevent non-enclave access to this memory region. Execution threads enter and exit enclaves using SGX CPU instructions, thus ensuring that in-enclave code execution can only begin from well-defined call gates. From a software perspective, untrusted code can make *ECALLs* to invoke enclave functions, and enclave code can make *OCALLs* to invoke untrusted functions outside the enclave.

Attestation Service. SGX provides two types of attestation: local and remote. Local attestation enables one enclave to attest another (running on the same SGX machine) to verify that the latter is a genuine enclave actually running on the same CPU. Remote attestation involves more entities. First, an application enclave to be attested creates a *report* that summarizes information about itself, e.g., code it is running. This report is sent to a special enclave, called *quoting enclave*, which is provided by Intel and available on all SGX machines. Quoting enclave confirms that requesting application enclave is running on the same machine and returns a *quote*, which is a report with the quoting enclave's signature. The application enclave sends this quote to the Intel Attestation Service (IAS) and obtains an *attestation verification report*. This is signed by the IAS confirming that the application enclave is indeed a genuine SGX enclave running the code it claims. Upon receiving an attestation verification report, the verifier can make an informed trust decision about the behavior of the attested enclave.

Side-Channel Attacks. SGX is vulnerable to side-channel attacks [35, 49], and various mechanisms have been proposed [17, 44, 47] to mitigate them. There have also been various studies investigating network side channel attacks against encrypted DNS [28, 36, 46] (including both DNS-over-TLS [29, 52] and DNS-over-HTTPS [27]).

Since defending against side-channel attacks is orthogonal to our work, we expect that a production implementation would include relevant mitigation mechanisms.

3 ADVERSARY MODEL AND REQUIREMENTS

3.1 Adversary Model

The adversary's goal is to learn, or infer, information about DNS queries sent by clients. We consider two types of adversaries, based on their capabilities:

The first type is a malicious RecRes operator who has full control over the physical machine, its OS and all applications, including the RecRes. We assume that the adversary cannot break any cryptographic primitives, assuming that they are correctly implemented. We also assume that it cannot physically attack hardware components, e.g., probe CPU physically to learn TEE secrets. This adversary also controls all of the RecRes's communication interfaces, allowing it to drop/delay packets, measure the time required for query processing, and observe all cleartext packet headers.

The second type is a network adversary, which is strictly weaker than the malicious RecRes operator. In the passive case, this adversary can observe any packets that flow into and out of RecRes. In the active case, this adversary can modify and forge network packets. Note that this represents the strongest form of network adversary who can observe and potentially manipulate both the *downstream* (i.e., client – RecRes) and *upstream* (i.e., RecRes– NS) communication. In the common case, the network adversary would only be able to observe the downstream communication (e.g., the adversary is another user connected to the same wireless network as the victim). DNS-over-TLS alone (without PDoT) is sufficient to thwart a passive network adversary. However, since an active adversary could redirect clients to a malicious RecRes, clients need an efficient mechanism to authenticate the RecRes and determine whether it is trustworthy, which is one of the main contributions of PDoT.

We do not consider Denial-of-Service (DoS) attacks on RecRes, since these do not help to achieve either adversary's goal of learning clients' DNS queries. Connection-oriented RecRes-s can defend against DoS attacks using cookie-based mechanisms to prevent SYN flooding [52].

3.2 System Requirements

We define the following requirements for the overall system:

- **R1: Query Privacy.** Contents of client's query (specifically, domain name to be resolved) should not be learned by the adversary. Ideally, payload of the DNS packets should be encrypted. However, even if packets are encrypted, their headers leak information, such as source and destination IP addresses. In Section 8.1, we quantify the amount of information that can be learned via traffic analysis.
- **R2: Deployability.** Clients using a privacy-preserving RecRes should require no special hardware. Minimal software modifications should be imposed. Also, for the purpose of transition and compatibility, a privacy-preserving RecRes should be able to interact with legacy clients that only support unmodified DNS-over-TLS.
- **R3: Response Latency.** A privacy-preserving RecRes should achieve similar response latency to that of a regular RecRes.
- **R4: Scalability.** A privacy-preserving RecRes should process a realistic volume of queries generated by a realistic number of clients.

Note: Query privacy guarantees provided by PDoT rely on the forward-looking assumption that communication between RecRes and respective NS-s is also protected by DNS-over-TLS. The DNS Privacy Working Group is working toward a standard for encryption and authentication of DNS resolver-to-ANS communication [8], using essentially the same mechanism as DNS-over-TLS. We expect an increasing number of NS-s to begin supporting

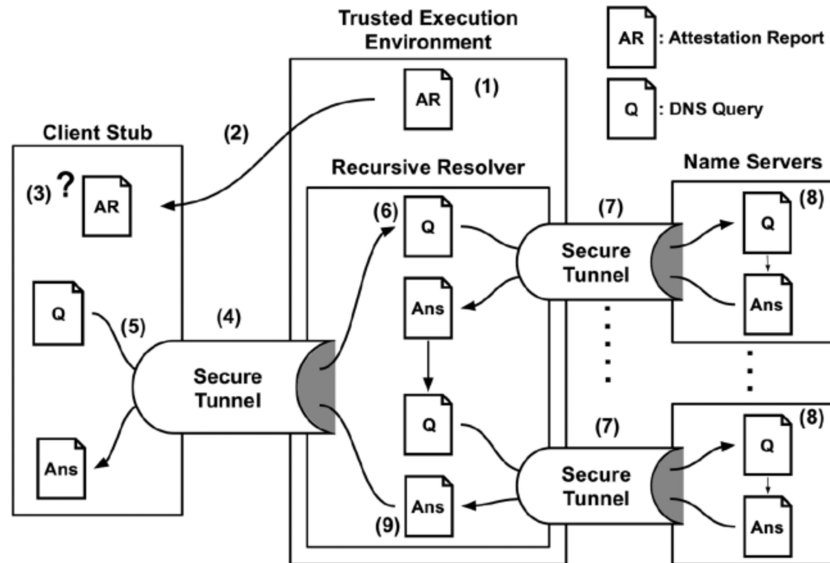


Fig. 1. Overview of the proposed system.

this standard in the near future. Once PDoT is enabled at the RecRes, it can provide incremental query privacy for queries served from a DNS-over-TLS NS. As discussed in Section 5, with small design modifications, PDoT can be adapted for use in NS-s. Additionally, as a temporary solution until DNS-over-TLS becomes the norm for protecting communications between the RecRes and NS, PDoT can be modified to support DNS Security Extensions (DNSSEC) [5]. Although DNSSEC does not provide DNS query privacy, it provides a means for PDoT to authenticate the NS.

4 SYSTEM MODEL AND DESIGN CHALLENGES

4.1 PDoT System Model

Figure 1 shows an overview of PDoT. It includes four types of entities: client, RecRes, TEE, NS-s. We now summarize PDoT operation, reflected in the figure: (1) After initial start-up, TEE creates an attestation report. (2) When client initiates a secure TLS connection, the attestation report is sent from RecRes to the client alongside all other information required to setup a secure connection. (3) Client authenticates and attests RecRes by verifying the attestation report. It checks whether RecRes is running inside a genuine TEE and running trusted code. (4) Client proceeds with the rest of the TLS handshake procedure only if verification succeeds. (5) Client sends a DNS query to RecRes through the secure TLS channel it has just set up. (6) RecRes receives a DNS query from client, decrypts it into TEE memory, and learns the domain name that the client wants to resolve. (7) RecRes sets up a secure TLS channel to the appropriate NS to resolve the query. (8) RecRes sends a DNS query to NS over that channel. If NS's reply includes an IP address of the next NS, then RecRes sets up another TLS channel to that NS. This is done repeatedly, until RecRes successfully resolves the name to an IP address. (9) Once RecRes obtains the final answer, it sends this to client over the secure channel. Client can reuse the TLS channel for future queries.

Note that we assume RecRes is not under the control of the user. In some cases, users could run their own RecRes-s, which would side-step the concerns about query privacy. For example, modern home routers are sufficiently powerful to run an in-house RecRes. However, this approach cannot be used in public networks (e.g., airports or coffee shop WiFi networks), which are the target scenarios for PDoT.

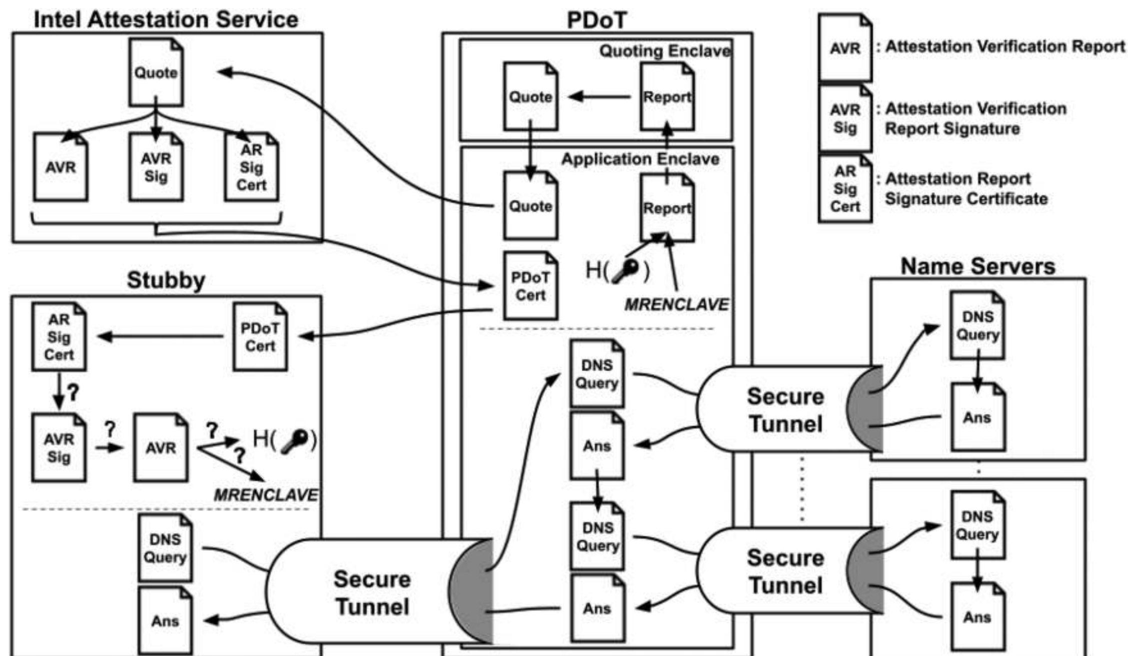


Fig. 2. Overview of PDoT implementation.

4.2 Design Challenges

The following key challenges were encountered in the process of PDoT's design:

- **C1: TEE Functionality Limitations.** To satisfy their security requirements, current TEEs (e.g., SGX and TrustZone) often limit the functionality available to code that runs within them. One example is the inability to fork within the TEE. Forking a process running inside the TEE forces the child process to run outside the TEE, breaking RecRes security guarantees. Another example is that system calls, such as socket communication, cannot be made from within the TEE.
- **C2: TEE Memory Limitations.** A typical TEE has a relatively small amount of memory. Although an SGX enclave can theoretically have a large amount of in-enclave memory, this will require page swapping of EPC pages. The pages to be swapped must be encrypted and integrity protected to meet the security requirements of SGX. Therefore, page swapping places a heavy burden on performance. To avoid page swapping, enclave size should be less than the size of the EPC—typically 128 MB. Since RecRes is a performance-critical application, its size should ideally not exceed 128 MB. This limit negatively impacts RecRes throughput, as it bounds the number of threads that can be spawned in a TEE.
- **C3: TEE Call-in/Call-out Overhead.** Applications requiring functionality that is not available within the TEE must switch to the non-TEE side. This introduces additional overhead, both from the switching itself, and from the need to flush and reload CPU caches. Identifying and minimizing the number of times RecRes switches back and forth (whilst keeping RecRes functionality correct) is a substantial challenge.

5 IMPLEMENTATION

Figure 2 shows an overview of the PDoT design. Since our design is architecture independent, it can be implemented on any TEE architecture that provides the features outlined in Section 2.2. We chose the off-the-shelf Intel SGX as the platform for the proof-of-concept PDoT implementation to conduct an accurate performance

evaluation on real hardware (see Section 7). Therefore, our implementation is subject to performance and memory constraints in the current version of Intel SGX. It is thus best suited for small-scale networks, e.g., a WiFi hotspot provided by a typical coffee shop. However, as TEE technology advances, we expect that our design will scale to larger networks.

5.1 PDoT

PDoT consists of two parts: (1) a trusted part residing in TEE enclaves and (2) untrusted part that operates elsewhere. The former is responsible for resolving DNS queries, and the latter – for accepting incoming connections, assigning file descriptors to sockets, and sending/receiving data received from the trusted part.

Enclave Startup Process. When the application enclave starts, it generates a new public-private key-pair within the enclave. It then creates a *report* that summarizes enclave and platform state. The report includes a SHA256 hash of the entire code that is supposed to run in the enclave (called *MRENCLAVE* value) and other attributes of the target enclave. PDoT also includes a SHA256 hash of the previously generated public key in the report. The report is then passed on to the SGX quoting enclave to receive a *quote*. The quoting enclave signs the report and thus generates a quote, which cryptographically binds the public key to the application enclave. The quoting enclave sends the quote to the application enclave, which forwards it to the IAS to obtain an *attestation verification report*. It can be used in the future by clients to verify the link between the public key and the MRENCLAVE value. After receiving the attestation verification report from IAS, the application enclave prepares a self-signed X.509 certificate required for the TLS handshake. In addition to the public key, the certificate includes: (1) attestation verification report, (2) attestation verification report signature, and (3) attestation report signing certificate, extracted from (1). The MRENCLAVE value and hash of public key are enclosed in the attestation verification report.

TLS Handshake Process.[1] Once the application enclave is created, PDoT can create TLS connections and accept DNS queries from clients. The client initiates a TLS handshake process by sending a message to PDoT. This message is captured by untrusted part of PDoT and triggers the following events.[2] First, untrusted part of PDoT tells the application enclave to create a new TLS object within the enclave for this incoming connection. This forces the TLS endpoint to reside inside the enclave. The TLS object is then connected to the socket where the client is waiting to be served. The RecRes then exchanges several messages with the client, including the self-signed certificate that was created in the previous section. Having received the certificate from RecRes, the client authenticates RecRes and validates the certificate (see Section 5.2). Only if the authentication and validation succeed does the client resume the handshake process.

DNS Query Resolving Process. The client sends a DNS query over the TLS channel established above. Upon receiving the query, the RecRes decrypts it within the application enclave and obtains the target domain name. The RecRes begins to resolve the name starting from root NS, by doing the following repeatedly: (1) set up a TLS channel with NS, (2) send DNS queries and receive replies via that channel. Once the RecRes receives the answer from the NS, the RecRes returns it to the client over the original TLS channel.

Figure 3 illustrates the threading model of PDoT. For each client, we spawn two threads, a *ClientReader* and one or more *QueryHandler* threads.

The *ClientReader* thread is responsible for an accepting incoming connection from a client and performing the TLS handshake. Once a TLS connection has been established, this thread reads the incoming DNS queries and stores them in client-specific FIFO queues—the *QueryLists* in Figure 3.

The *QueryHandler* threads are responsible for performing the recursive resolution of queries, and sending the responses to the clients. Each *QueryList* has one or more *QueryHandler* threads associated with it. Using the

[1]This design is derived from the SGX RA-TLS [31] whitepaper.
[2]Since we consider a malicious RecRes operator, it has an option not to trigger these events. However, clients will notice that their queries are not being answered and can switch to a different RecRes.

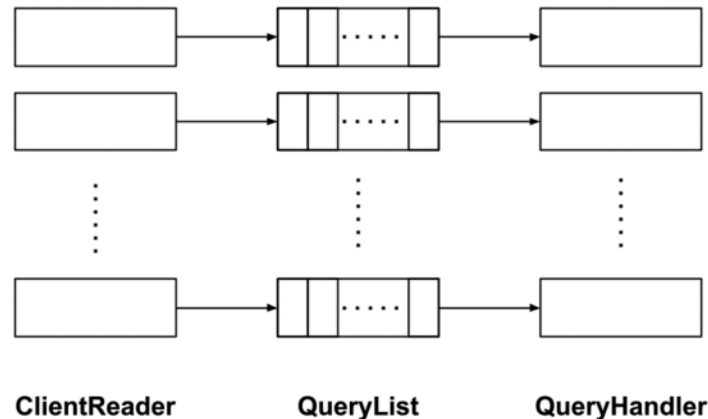


Fig. 3. Overview of PDoT threading model.

thread synchronization primitives in the SGX SDK, the *QueryHandler* threads wait on a condition until they are signalled by the *ClientReader* thread to indicate that there is a pending query in the *QueryList*. Once signalled, a *QueryHandler* thread first checks whether the client is still accepting responses from RecRes, and if so, resolves the query and sends the response via the established TLS connection.

In our implementation we only use one *QueryHandler* thread per client to maximize the number of concurrent client connections we can support. Due to the limitations on enclave memory, only a limited number of threads can execute within the enclave concurrently. This is controlled by the number of Thread Control Structure data structures allocated during enclave compilation. Additionally, by using only a single *QueryHandler* thread per client, this thread can be given exclusive write access to the client's TLS connection, thus avoiding the need for costly thread synchronization.

Having a dedicated *ClientReader* thread to read and buffer the incoming queries from a single client enables PDoT to *receive* concurrent requests. This allows a client to send multiple DNS queries within a short timespan without waiting for the answers to previous queries. For example, when a client loads a webpage that includes images and advertisements from different domains, multiple DNS queries are triggered at the same time. However, because we chose to use one *QueryHandler* thread per client, PDoT does not *resolve* the concurrent requests it received from a single client. In comparison to alternative designs (described below), we observed that this approach provides higher throughput when multiple clients are connected (see Section 7.4).

We previously implemented an alternative design using a dedicated *ClientWriter* thread per client to buffer responses and write them out the the client. However, we found that this increased the number of concurrent threads in the enclave without providing a performance benefit. Moreover, using a dedicated *ClientWriter* thread introduced additional queues and therefore required additional thread synchronization variables.

We also previously implemented another alternative design using a single *QueryList* shared between multiple *ClientReader* threads and multiple *QueryHandler* threads. This allowed *ClientReader* threads to share multiple *QueryHandler* threads, thus allowing PDoT to resolve multiple DNS queries from a single client concurrently. However, we observed that this causes contention between the *ClientReader* threads when accessing the single *QueryList*, leading to a large variance in query response times and causing some queries to time-out.

Caching. To measure the influence of caching, we implemented a simple in-enclave cache for PDoT. It uses a red-black tree data structure and stores all records associated with the clients' queries, indexed by the queried domain. This results in $O(log_2(N))$ access times with N entries in the cache. We discuss the potential privacy risks of enabling caching and propose possible mitigation strategies in Section 6.

PDoT ANS with TEE support. With minor design changes, PDoT RecRes design can be modified for use as an ANS. Similarly to the caching mechanism described above, an PDoT ANS can look up the answers to queries in an internal database, rather than contact external NS-s. The same way that clients authenticate PDoT RecRes, the RecRes can authenticate the PDoT ANS. Clients can thus establish trust in both RecRes and ANS using *transitive attestation* [3].

5.2 Client with PDoT Support

We picked the Stubby client stub from the `getdns` project [33], which offers DNS-over-TLS support and modified it to perform remote attestation during the TLS handshake. We chose to use Stubby, as it is the most well developed, open sourced DNS-over-TLS client. We now describe how the client verifies its RecRes, decides whether the RecRes is trusted, and emits the DNS request packet.

RecRes Verification. After receiving a DNS request from an application, the client first checks whether there is an existing TLS connection to its RecRes. If so, then the client reuses it. If not, then it attempts to establish a new connection. During the handshake, the client receives a certificate from RecRes, from which it extracts: (1) attestation verification report, (2) attestation verification report signature, and (3) attestation report signing certificate. This certificate is self-signed by IAS and the client is assumed to trust it. From (3), the client first retrieves the IAS public key and, using it, verifies (2). Then, the client extracts the SHA256 hash of RecRes's public key from (1) and verifies it against (3). This way, the client is assured that RecRes is indeed running in a genuine SGX enclave and uses this public key for the TLS connection.

Trust Decision. The client also extracts the MRENCLAVE value from (1), which it compares against the list of acceptable MRENCLAVE values. If the MRENCLAVE value is not listed or one of the verification steps fail, then the client stub aborts the handshake, moves on to the next RecRes, and re-starts the process. Note that the trust decision process is different from the normal TLS trust decision process. Normally, a TLS server-side certificate binds the public key to one or more URLs and organization names. However, by binding the MRENCLAVE value with the public key, the clients can trust RecRes based on its behavior, and not its organization (recall that the MRENCLAVE value is a hash of RecRes code). There several options for deciding which MRENCLAVE values are trustworthy. For example, vendors could publish lists of expected MRENCLAVE values for their resolvers. For open source resolvers like PDoT, anyone can re-compute the expected MRENCLAVE value by recompiling the software, assuming a reproducible build process. This would allow trusted third parties (e.g., auditors) to inspect the source code, ascertain that it upholds required privacy guarantees, and publish their own lists of trusted MRENCLAVE values.

Sending DNS request. Once the TLS connection is established, the client sends the DNS query to RecRes over the TLS tunnel. If it does not receive a response from RecRes within the specified timeout, then it assumes that there is a problem with RecRes and sends a DNS reply message to the application with an error code SERVFAIL.

5.3 Overcoming Technical Challenges

As discussed in Section 4.2, PDoT faced three main challenges, which we addressed as follows.

Limited TEE Functionality. The inability to use sockets within the TEE is a challenge, because the RecRes cannot communicate with the outside world. We address this issue by having a process running outside the TEE, as described in Section 5.1. This process forwards packets from the client to TEE through ECALLs and sends packets received from TEE via OCALLs. However, because it is outside the TEE, this process might redirect the packet to a malicious process or simply drop it. We discuss this issue in Section 7.1. Another function unavailable within TEE is forking a process. PDoT uses `pthreads` instead of forking to run multiple tasks concurrently in a TEE.

Limited TEE Memory. We use several techniques to address this challenge. First, we ensure no other enclaves (other than the quoting enclave) run on the RecRes SGX machine. This allows PDoT to use all available EPC memory. Second, we minimize the number of threads running inside the enclave to save space.

OCALL and ECALL Overhead. ECALLs and OCALLs introduce overhead and therefore should be avoided as much as possible. For example, all threads mentioned in the previous section must wait until they receive the following information: for the *ClientReader* thread, the DNS query from the client, and for the *QueryHandler* thread, the query from the QueryList. PDoT was implemented so that these threads wait inside the enclave whenever possible to minimize the number of expensive enclave entries and exits.

6 PRIVACY-PRESERVING DNS CACHING

Some DNS recursive resolvers cache query results and can use these to answer a query directly in the case of a cache hit. Caching is beneficial from the client's perspective, because it reduces query latency. The RecRes also benefits from not having to establish connections to external NS-s. However, if implemented naively, then caching at the RecRes may allow adversaries to learn something about the victim's query. In this section, we discuss the potential privacy risks, and show how these can be overcome in PDoT.

6.1 Attacks Exploiting Caching

As explained in Section 3.1, we consider attacks by either a malicious RecRes operator or a network adversary. Both aim to learn information from a naive caching implementation.

Network adversary: As discussed in the previous section, the network adversary cannot see the contents of the victim's query thanks to the TLS connection. However, we assume this adversary can still observe all (encrypted) packets sent and received by the RecRes on both the upstream and downstream communication links. We assume the network adversary can also submit queries to the RecRes and measure the time between queries and responses.

The network adversary's goal is to infer whether a specific query, submitted by either the victim or himself, was answered from the cache. This could be achieved by measuring the time between query and response, since a cache hit can be answered relatively quickly. Alternatively, the adversary could observe any (encrypted) recursive queries on the upstream interface, since a cache hit can be answered without consulting upstream NS-s. Knowing whether a query was answered from the cache can leak information to the adversary in two ways:

- The adversary could have primed the cache with specific domains *before* the victim arrives; thus whenever the victim's query is answered from the cache, the adversary can infer that the victim queried one of the primed domains.
- The adversary can probe the cache by submitting queries *after* the victim; thus whenever one of his own queries is answered from the cache, the adversary can infer that the victim had previously queried this domain.

In most cases, these attacks would be complicated by the presence of other users and the inability of the network adversary to clear the RecRes's cache. However, we consider the worst-case scenario, since the network adversary can block packets from other users to isolate the victim's traffic.

Malicious RecRes operator: A malicious RecRes operator can perform the same attacks as the network adversary and can also observe the memory access pattern of the RecRes. In addition to learning whether or not a specific query was answered from the cache, the malicious operator also aims to learn *which* cached response was used. Depending on the type of TEE used to implement PDoT, the malicious operator might be able to monitor PDoT's (encrypted) memory accesses. For example, it has been shown that in Intel SGX, an adversary

in control of the OS can monitor memory accesses deterministically at page granularity (typically 4 kB) [49], or probabilistically at cache-line granularity [35]. Similarly to the network adversary, this information can be leveraged in two ways:

- The malicious operator could prime the cache with specific domains before the victim arrives, and record the address of each primed entry in the cache. When there is a cache hit, the address of the response reveals exactly which domain the victim queried.
- If there are multiple users, then the malicious operator could wait for the users to fill the cache, whilst recording which users created which cache entries. Afterwards, the malicious operator could submit its own queries, and if there are any cache hits, the address reveals which user queried for that domain.

6.2 Mitigating Cache Attacks

We propose the following techniques to mitigate the above attacks. In some cases, multiple techniques must be combined. During remote attestation, clients can ascertain which mitigations the RecRes is using.

A nocache bit: Short of disabling caching completely, the impact of the above attacks could be minimized by giving users the choice of whether or not to use the cache on a per-query basis. For *non-sensitive* queries, the cache could be used as normal, whilst for *sensitive* queries, the users could specify that the query should not be answered from the cache (even if an answer is available) and that the response should not be cached. This could be done by assigning the currently unused Z bit in the DNS query header as a nocache (NC) bit. Historically, the Z bit was used to indicate that only a response from the primary server for a zone was acceptable [21], which is already very similar to the nocache bit. The only addition is that the RecRes would not cache responses to queries with this bit set.

The distinction between sensitive and non-sensitive queries should be made by each user or their client software (e.g., when browsing in "incognito mode," all DNS queries could be marked as sensitive). The adversary would still be able to perform all the above attacks, but the impact to the user would be minimal since these would only reveal information about non-sensitive queries. Sensitive queries would be indistinguishable from cache misses. Although less efficient than normal non-privacy-preserving caching, this approach is more efficient than disabling caching completely, whilst providing similar privacy guarantees.

Delayed responses: In most cases, a realistic network adversary's abilities are weaker than the assumptions we make in Section 3.1. Specifically, this adversary would not usually be able to see upstream communication between the RecRes and the NS-s. For example, in the coffee shop scenario, a malicious user on the wireless network may be able to observe the communication between other wireless clients and the RecRes, but not the RecRes's upstream communication, as this would take place via a wired interface. This type of network adversary can thus only infer whether there was a DNS cache hit (either his own or a victim's query) by measuring the time between the query and response.

One possible mitigation is therefore to introduce artificial delay when answering queries from the cache [2]. For example, when there is a cache miss, the RecRes itself can measure the time required to resolve a particular query and store this time measurement along with the answer in the cache. When this answer is subsequently served from the cache, the RecRes can simply wait for the same amount of time before sending the response to the client. Although this results in higher-latency responses for the clients, it is still beneficial for the RecRes, as it reduces the load on the upstream connection. This addresses both types of attacks by the weaker network adversary, although it does not help against the malicious RecRes operator, who can observe whether or not the RecRes makes an upstream query.

Pre-populated cache: On startup and periodically while it is running, the RecRes itself could query for example the top 1,000 most popular domains and store the results in the cache. This mitigates the first of the two possible attacks the network adversary can perform, because when there is a cache hit by the victim, the adversary cannot

distinguish whether it is for a primed or a pre-populated domain. In other words, this negates the adversary's ability to prime the cache. Note that the network adversary can still probe the cache after the victim's query. The malicious RecRes operator can also bypass this mitigation by observing the RecRes's memory access pattern to distinguish between primed and pre-populated domains.

Oblivious memory accesses: To prevent the malicious RecRes operator for learning which memory address in the cache contained the desired result, we need to make all cache memory accesses oblivious to the operator. This could be used in conjunction with the pre-populated cache mitigation above. Since this is a more general challenge for applications using SGX enclaves, multiple techniques are already available, including general-purpose oblivious RAM schemes (e.g., References [16, 43]), and purpose-built oblivious access schemes (e.g., Reference [47]). These techniques could be applied to the caching mechanism described in the previous section.

6.3 DNS Cache Poisoning

Although not an attack on privacy, introduction of a cache also opens PDoT to the possibility of DNS cache poisoning attacks, whereby a network adversary causes incorrect answers to be stored in the cache. This could be used as the precursor to a man-in-the-middle attack where the adversary redirects the victim to a malicious IP address, even though the domain appears to be correct.

A network adversary could attempt to poison the cache by pretending to be an upstream NS and responding to upstream queries from the RecRes before the real NS response arrives. Alternatively, the adversary could force the RecRes to query a malicious upstream NS with which the adversary is colluding (e.g., simply by querying the relevant domain). The DNS response from the malicious NS could also include falsified *additional information* records for other domains, which would also poison the cache. These attacks can be mitigated by the RecRes using DNSSEC [5] wherever possible.

7 EVALUATION

7.1 Security Analysis

This section describes how query privacy (Requirement R1) is achieved, with respect to the two types of adversaries, per Section 3.1.

Malicious RecRes operator. Recall that a malicious RecRes operator controls the machine that runs PDoT RecRes. It cannot obtain the query from intercepted packets since they flow over the encrypted TLS channel. Also, because the local TLS endpoint resides inside the RecRes enclave, the malicious operator cannot retrieve the query from the enclave, as it does not have access to the protected memory region.

However, a malicious RecRes operator may attempt to connect the socket to a malicious TLS server that resides in either (1) an untrusted region or (2) a separate enclave that the operator itself created. If the operator can trick the client into establishing a TLS connection with the malicious TLS server, then the adversary can obtain plaintext DNS queries. For case (1), the verification step at the client side fails, because the TLS server certificate does not include any attestation information. For case (2), the malicious enclave might receive a legitimate attestation verification report, attestation verification report signature, and attestation report signing certificate from IAS. However, that report would contain a different MRENCLAVE value, which would be rejected by the client. To convince the client to establish a connection with PDoT RecRes, the adversary has no choice except to run the code of PDoT RecRes. Therefore, in both cases, the adversary cannot trick the client into establishing a TLS connection with a TLS server other than the one running a PDoT RecRes.

Network Adversary. Recall that this adversary captures all packets to/from PDoT. It cannot obtain plaintext queries since they flow over the TLS tunnel. The only information it can obtain from packets includes cleartext header fields, such as source and destination IP addresses. This information, coupled with a timing attack, might

let the adversary correlate a packet sent from the client with a packet sent to an NS. The resulting privacy leakage is discussed in Section 8.1

7.2 Deployability

Section 5 argues that PDoT clients do not need special hardware and require only minor software modifications (Requirement R2). To aid deployability, PDoT also provides several configurable parameters, including the number of QueryHandle threads (to adjust throughput), the amount of memory dedicated to each thread (to serve clients that send a lot of queries at a given time), and the timeout of QueryHandle threads (to adjust the time for a QueryHandle thread to acquire a resource). Another consideration is incremental deployment, where some clients may request DNS-over-TLS without supporting PDoT. PDoT can handle this situation by having its TLS certificate **also** be signed by a trusted root CA, since legacy clients will ignore PDoT-specific attestation information.

On the client side, an ideal deployment scenario would be for browser or OS vendors to update their client stubs to support PDoT . The same way that browser vendors currently include and maintain a list of trusted root CA certificates in their browsers, they could include and periodically update a list of trustworthy MRENCLAVE values for PDoT resolvers. This could all be done transparently to end-users. As with root CA certificates, expert users can manually add/remove trusted MRENCLAVE values for their own systems. In practice, there are only a handful of recursive resolver software implementations. Thus, even allowing for multiple versions of each, the list of trusted MRENCLAVE values would be orders of magnitude smaller than the list of public keys of every trusted resolver, as would be required for standard DNS-over-TLS.

7.3 Latency Evaluation

We aim to assess overhead introduced by running RecRes inside an enclave. To do so, we measure the time to resolve a DNS query using PDoT and compare this with the corresponding latency incurred by Unbound [34], a popular open source RecRes. To meet requirement R3, PDoT should not incur a significant increase in latency compared to Unbound.

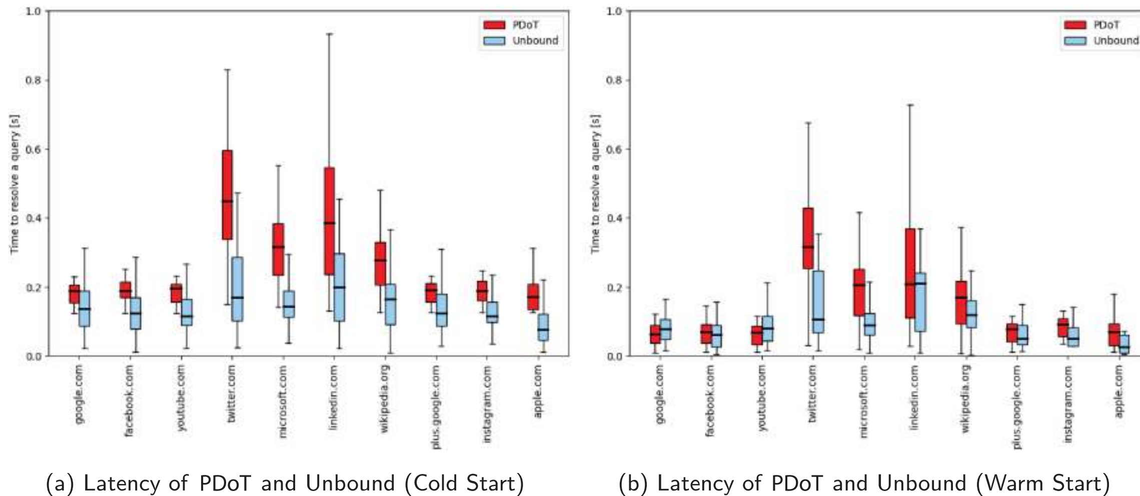
Experimental Setup. We ran PDoT on a DC4s_v2 series virtual machine (VM) in Microsoft Azure. This VM has four virtual CPUs and 16 GB of memory and supports Intel SGX with up to 112 MB of EPC. We used Ubuntu 18.04 for the OS and Intel SGX SDK version 2.9. We configured our PDoT to support up to 100 concurrent clients and used Stubby as the DNS-over-TLS client.

We measured latency under two scenarios: cold start and warm start. In the former, the client sets up a new TLS connection every time it sends a query to the RecRes. In the warm start scenario, the client sets up one TLS connection with the RecRes at the beginning, and reuses it throughout the experiment. In other words, cold start measurements also include the time to establish the TLS connection. In this experiment, caching mechanisms of both PDoT and Unbound were disabled.

We created a Python program to feed DNS queries to the client. It sends 100 queries sequentially (i.e., waits for an answer to the previous query before sending the next query) for each of the top ten domains in the Majestic Million domain list [38].

The Python program measures the time between sending the query and receiving an answer. For the cold start experiment, we spawned a new Stubby client and established a new TLS connection for each query. In the warm start scenario, we first established the TLS connection by sending a query for another domain (not in the top 10) but did not include this in the timing measurement.

Note that the numeric latency values are specific to our experimental setup, because they depend on network bandwidth of the RecRes, and latency between the latter and relevant NS-s. The important aspect of this experiment is the ratio between the latencies of PDoT and Unbound. Therefore, it is not meaningful to compute average latency over a large set of domains. Instead, we took multiple measurements for each of a small set of



(a) Latency of PDoT and Unbound (Cold Start) (b) Latency of PDoT and Unbound (Warm Start)

Fig. 4. Latency comparison of PDoT and Unbound.

domains (e.g., 100 measurements for each of 10 domains) so as to analyse the range of response latencies for each domain.

Results and observations. The results are shown in Figure 4. Red boxes show latency of PDoT and the blue boxes of Unbound. In these plots, boxes span from the lower to upper quartile values of collected data. Whiskers span from the lowest datum within the 1.5 interquartile range (IQR) of the lower quartile to the highest datum within the 1.5 IQR of the upper quartile. Median values are shown as black horizontal lines inside the boxes.

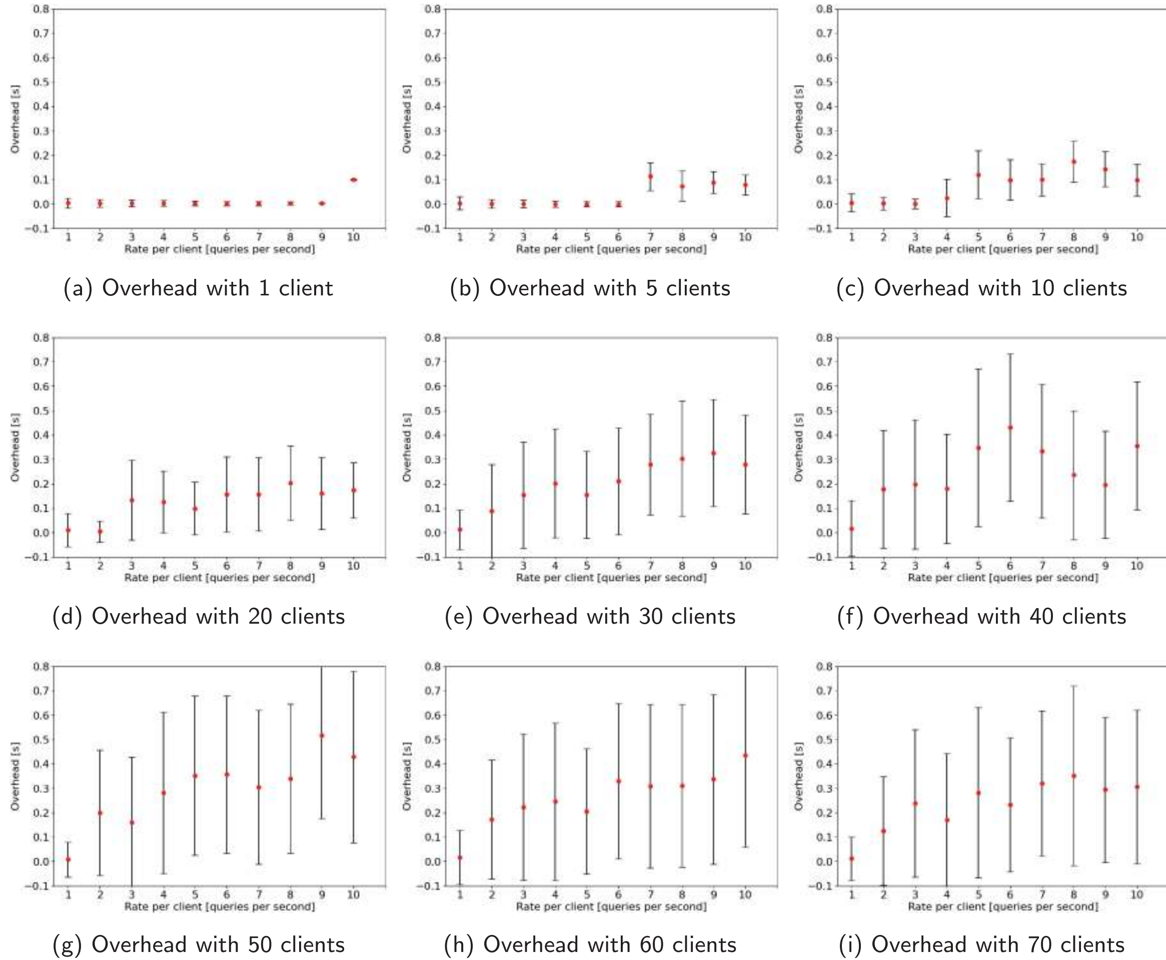
For the cold-start case in Figure 4(a), although Unbound is typically faster than our proof-of-concept PDoT implementation, the range of latencies is similar. For 5 of 10 domains, the upper whisker of PDoT was lower than that of Unbound. Moreover, we observed that the range of latencies of PDoT overlap with that of Unbound. Across the tested domains, PDoT shows an average of 73% overhead compared to Unbound in this setting.

For the warm-start case in Figure 4(b), the median latency is lower across the board compared to the cold-start setting, because the TLS tunnel has already been established. In this setting, PDoT shows an average of 44% overhead compared to Unbound. Compared to the cold start setting, the difference of the range of latencies is smaller. In fact, for half of the domains, the range of PDoT latencies is smaller than that of Unbound. In practice, once the client has established a connection to RecRes, it will maintain this connection; thus, the vast majority of queries will see only the warm-start latency.

7.4 Throughput Evaluation

The objective of our throughput evaluation is to measure the rate at which the RecRes can sustainably respond to queries. PDoT's throughput should be close to that of Unbound to satisfy requirement R4.

Experiment setup. We used the same experimental setup as for the latency evaluation (Section 7.3). The client and RecRes were run on different VMs, located in the same virtual network, so that the RecRes could use all available resources of a single VM. This is representative of a local RecRes running in a small network (e.g., a coffee shop WiFi network). We conducted this experiment using Stubby and the same two RecRes-s as in the latency experiment. Stubby was configured to reuse TLS connections where possible and the caching mechanisms of both PDoT and Unbound were disabled. To eliminate further variance due to external network latency, we also ran our own authoritative NS in the same virtual network as the RecRes. Running our own authoritative NS also has the benefit of not applying strain on the public NS during our evaluation. All queries sent to the RecRes



(a) Overhead with 1 client
(b) Overhead with 5 clients
(c) Overhead with 10 clients
(d) Overhead with 20 clients
(e) Overhead with 30 clients
(f) Overhead with 40 clients
(g) Overhead with 50 clients
(h) Overhead with 60 clients
(i) Overhead with 70 clients

Fig. 5. Throughput overhead comparison of PDoT.

could be resolved with a single upstream query to our local NS. To simulate a small to medium-scale network, we varied the number of concurrent client connections between 1 and 70 and adjusted the query rate from 1 to 10 queries per second per client. We maintained each constant query rate for half a minute.

Results and observations. The results of our throughput experiments are shown in Figure 5. Each graph corresponds to a different number of clients. The horizontal axis shows different per-client query rates and the vertical axis shows the overhead for response latencies compared to the average response latency of Unbound for the same setting. During our preliminary evaluation, we found that the average latency of Unbound is consistently about 0.05 s across the evaluated scenarios. The red dots show the average overhead of PDoT, and the black error bars represent the standard deviation.

Figure 6 is a combination of the individual plots in Figure 5, showing the total rate at which PDoT is processing queries for different numbers of concurrent client connections in the x-axis and the average latency overhead in the y-axis. From this figure, we can see that the average latency overhead increases linearly as the total number of queries sent by the clients increases up to 200 queries per second. However, we can observe that the increase in the overhead is smaller after 200 queries per second. In fact, we can almost see that the overhead is converging

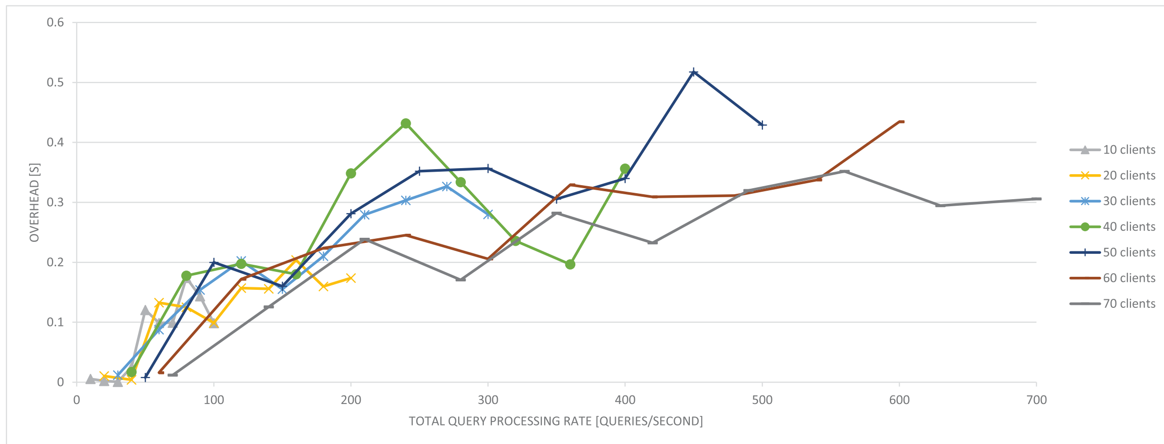


Fig. 6. Average overhead of PDoT for different numbers of clients.

to a particular value. For instance, after 500 queries per second, the latency overhead that 70 clients experience is around 0.3 s on average.

7.5 Caching Evaluation

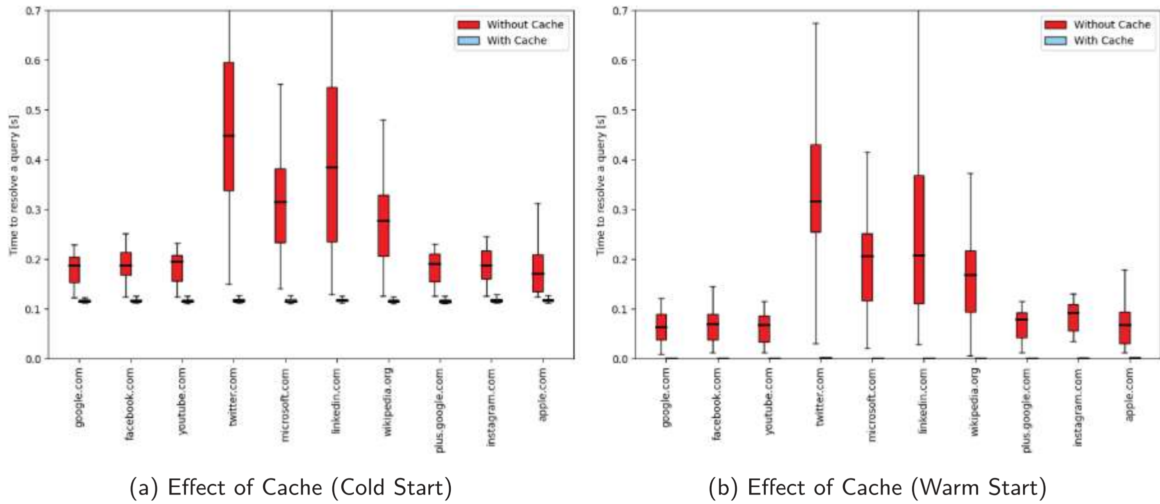
7.5.1 Caching Domains. First, we counted how many domains can be handled by the in-enclave cache, as described in Section 5. A single client was set up to query domains from the Majestic Million domain list [38]. The client queried domains until PDoT experienced a significant increase in latency, indicating that the cache starts to exceed the size of the EPC. Although the cache could grow beyond this point, this would lead to increased latency, because memory pages would have to be swapped in and out of the EPC. The memory setting for each thread was the same as the throughput evaluation. Even though different queries have answers of different sizes, this experiment showed that PDoT can cache at least 10,000 domains.

7.5.2 Effect on Latency. Next, we quantified the effect of caching on query latency by repeating the latency evaluation with and without caching. The results are shown in Figure 7. The average latency of PDoT without caching in the cold start setting is 249 ms and with caching is 117 ms. In the warm start setting, the average latency without caching is 123 ms and with caching is 1 ms. In addition to reducing average latency, caching also reduces the variance of the latency, thus preventing time-outs and improving the stability of the system.

7.5.3 Benefiting from Caching. Section 7.5.2 showed how both users and PDoT can potentially benefit from caching query answers and Section 7.5.1 showed that our simple proof-of-concept in-enclave cache can accommodate at least 10,000 domains. The overall performance impact of this type of cache depends heavily on the pattern of DNS queries sent by each user. However, we expect that a significant number of queries would fall within the most popular 10,000 domains, and could thus be answered from the cache. Additionally, PDoT operators can also pre-populate the cache with domains that are frequently queried by their users.

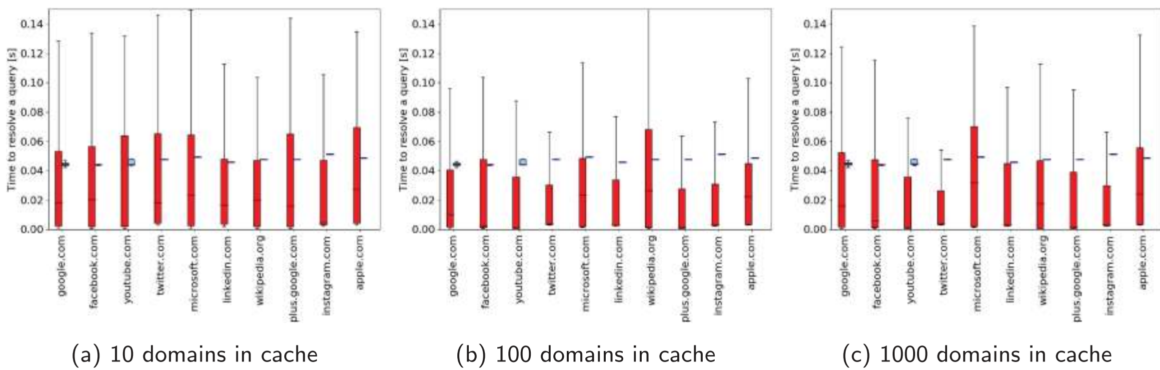
7.5.4 Effect of Different Numbers of Domains in Cache. Last, we evaluated performance of both resolvers with caching enabled; Unbound with its default caching behavior, and PDoT with the proof-of-concept caching mechanism.

Experiment setup. To simulate a realistic small-scale network, we ran PDoT on a low-cost Intel NUC consisting of an Intel Pentium Silver J5005 CPU with 128 MB of EPC memory and 4 GB of RAM, using Ubuntu 16.04 and the



(a) Effect of Cache (Cold Start) (b) Effect of Cache (Warm Start)

Fig. 7. Latency comparison of PDoT without caching (red) and with caching (blue).



(a) 10 domains in cache (b) 100 domains in cache (c) 1000 domains in cache

Fig. 8. Latency comparison of PDoT (red) and Unbound (blue) with different number of domains in cache.

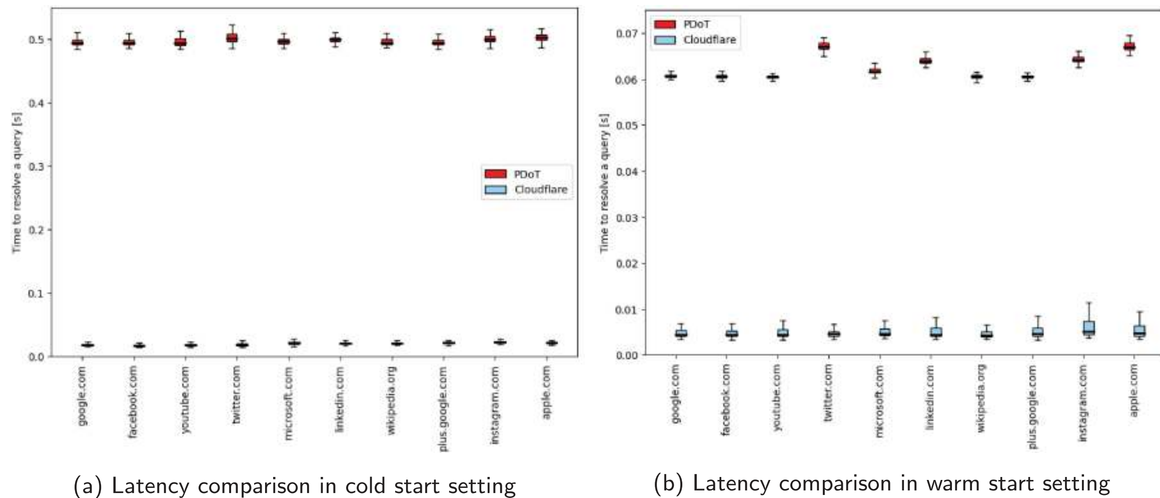
Intel SGX SDK version 2.2. We pre-populated resolvers' caches with varying numbers of domains and measured response latency for a representative set of 10 popular domains.

Results and observations. Figure 8 shows the response latency with caching enabled. The box and whisker plots have the same meaning as in Figure 4. Unbound serves responses from cache with a consistent latency irrespective of the number of entries in the cache. Although PDoT achieves lower average latencies when the cache is relatively empty, it has higher variability than Unbound. This is possibly due to the combination of our unoptimized caching implementation and latency of accessing enclave memory. Nevertheless, Figure 8 shows that even with the memory limitations of current hardware enclaves, PDoT can still benefit from caching a small number of domains.

7.6 Real-world Evaluation

In this experiment, we evaluated the difference in latency between Cloudflare's `1.1.1.1` and PDoT.

Experiment setup. PDoT was set up in an Microsoft Azure VM with caching enabled. The VM was set up on the US East coast region. Meanwhile, we confirmed that `1.1.1.1` resides on the US West coast region. A Stubby



(a) Latency comparison in cold start setting

(b) Latency comparison in warm start setting

Fig. 9. Latency comparison of PDoT (red) and 1.1.1.1 (blue).

client on a machine residing on the US West coast region. We ran the same latency evaluation script we used in Section 7.3. Since it is highly likely that 1.1.1.1 has the ten domains cached in its system, we also pre-populated the domains in PDoT's cache before conducting the experiment.

Results and observations. Figure 9 shows the result of the latency measurement. The box and whisker plots have the same meaning as in Figure 4. We can see that in both settings there is a difference in latency between the two resolvers. In the cold start setting, the average latency of PDoT was 0.5 s while 1.1.1.1 was 0.02 s. In the warm start setting, the average latency of PDoT was 0.06 s while 1.1.1.1 was 0.005 s. The difference in latency in the warm start setting can be due to several reasons. This includes the network latency due to the location of the resolver, 1.1.1.1 using optimized caching mechanisms, more powerful machines, and load balancing methods. As the cold start setting includes the TLS handshake, we assume the difference in latency in the cold start setting is largely due to the fact that 1.1.1.1 uses optimized cryptographic operations and optimized TLS handshake methods. Overall, we can conclude that the query response latency of our proof-of-concept PDoT is well below standard timeout time, which is 5 s for DNS queries. A production level PDoT can adopt the methods that 1.1.1.1 uses to further decrease its latency in resolving queries.

8 DISCUSSION

8.1 Information Revealed by IP Addresses

Even if the connections between the client, RecRes, and NS-s are encrypted using TLS, some information is still leaked. The most prominent and obvious is source/destination IP addresses. The network adversary described in Section 3.1 can combine these cleartext IP addresses with packet timing information to correlate packets sent from client to RecRes with subsequent packets sent from RecRes to NS.

Armed with this information, the adversary can narrow down the client's domain name query to one of the records that could be served by that specific ANS. Assuming the ANS can serve R domain names, the adversary has a $1/R$ probability of guessing which domain name the user queried. When $R > 1$, we call this a *privacy-preserving ANS*. This prompts two questions: (1) What percentage of domains can be answered by a privacy-preserving ANS and (2) What is the typical size of anonymity set (R) provided by a privacy-preserving ANS?

To answer these questions, we designed a scheme to collect records stored in various ANS-s. We sent DNS queries for 1,000,000 domains from the Majestic Million domain list [38] and gathered information about ANS-s

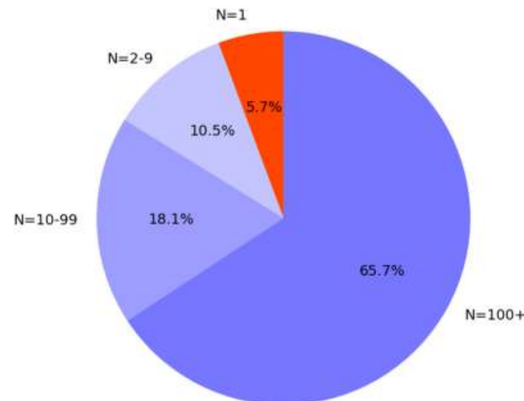


Fig. 10. Percentage of Majestic Million domains answered by an ANS with at least N records.

that can possibly provide the answer for each. By collecting data on possible ANS-s, we can map domain names to each ANS and thus estimate the number of records held by each ANS. Following the *Guidelines for Internet Measurement Activities* [11], we limited our querying rate to avoid placing undue load on any servers.

As shown in Figure 10, only 5.7% of domains we queried were served by non-privacy-preserving ANS-s, i.e., those that hold only one record). Examples of domain names served from non-privacy-preserving ANS-s included: `tinyurl.com`[3], `bing.com`, `nginx.org`, `news.bbc.co.uk`, and `cloudflare.com`. However, 9 of 10 queries were served by a privacy-preserving ANS and 65.7% by ANS-s that hold over 100 records.

These results are still approximations. Since we do not have data for domains outside the Majestic Million list, we cannot make claims about whether these would be served by a privacy-preserving ANS. We hypothesize that the vast majority of ANS-s would be privacy-preserving for the simple reason that it is more economical to amortize the ANS's running costs over multiple domains. However, we can be certain that our results for the Majestic Million are a strict lower bound on the level of privacy, because the ANS-s from which these are served could also be serving other domains outside of our list. It would be possible to arrive at a more accurate estimate by analyzing zone files of all (or at least most) ANS-s. However, virtually all ANS-s disable the interface to download zone files, because this could be used to mount DoS attacks. Therefore, this type of analysis would have to be performed by an organization with privileged access to all ANS-s' zone files.

8.2 Supporting DNS-over-HTTPS

Concurrently with the development of DNS-over-TLS, there have also been significant development of protocols and standards for DNS resolution over HTTPS (DoH) [26].

Similarly to DoT, the aim of DoH is to protect the confidentiality and integrity of DNS queries and responses in transit between the client and RecRes. DoH still uses TLS as the underlying secure channel but layers the DNS queries and responses on top of HTTPS. Whereas DoT uses its own dedicated port (853), DoH uses the same port as other HTTPS traffic (443). The last main difference is that DoH adds the ability for a server to *push* DNS responses to a client without the client having to first send a request.

DoH is already supported by a similar set of public DNS resolvers as DoT (e.g., Cloudflare's `1.1.1.1`) and is gaining support in web browsers (e.g., Firefox [9]) and operating systems (e.g., Windows 10 [30]). In February 2020, it was estimated that on average there are still seven times more DoT requests than DoH requests, but this may change in future [4].

[3]Since `tinyurl.com` is a URL shortening service, this is actually still privacy preserving, because the adversary cannot learn which short URL was queried.

DoH faces similar challenges to DoT and thus would also benefit from the security guarantees provided by PDoT. PDoT could be modified to support DoH by adding an HTTP parsing layer. Although this slightly increases the amount of software running within the TEE (i.e., the software trusted computing base), only minimal HTTP parsing functionality would be required.

9 RELATED WORK

There has been much prior work aiming to protect the privacy of DNS queries [10, 22, 23, 37, 45, 50, 51]. For example, Lu et al. [37] proposed a privacy-preserving DNS that uses distributed hash tables, different naming schemes, and methods from computational private information retrieval. Federrath et al. [23] introduced a dedicated DNS Anonymity Service to protect the DNS queries using an architecture that distributes the top domains by broadcast and uses low-latency mixes for requesting the remaining domains. These schemes all assume that all parties involved do not act maliciously.

There have also been some activities in the Internet standards community that focused on DNS security and privacy. DNSSEC [5] provides data origin authentication and integrity via public key cryptography. However, it does not offer privacy. Bortzmeyer [7] proposed a scheme Also, though not Internet standards, several protocols have been proposed to encrypt and authenticates DNS packets between the client and the RecRes (DNSCrypt [42]) and RecRes and NS-s (DNSCurve [1]). Moreover, the original DNS-over-TLS paper has been converted into a draft Internet standard [29]. All these methods assume that the RecRes operator is trusted and does not attempt to learn anything from the DNS queries.

Furthermore, there has been some research on establishing trust through TEEs to protect confidentiality and integrity of network functions. Specifically, SGX has been used to protect network functions, especially middle-boxes. For example, Endbox [24] aims to distribute middle-boxes to client edges: Clients connect through VPN to ensure confidentiality of their traffic while remaining maintainable. LightBox [20] is another middle-box that runs in an enclave; its goal is to protect the client's traffic from the third-party middle-box service provider while maintaining adequate performance. Finally, ShieldBox [48] aims to protect confidential network traffic that flows through untrusted commodity servers and provides a generic interface for easy deployability. These efforts focus on protecting confidential data that flows in the network, and do not target DNS queries.

10 CONCLUSION AND FUTURE WORK

This article proposed PDoT, a novel DNS RecRes design that operates within a TEE to protect privacy of DNS queries, even from a malicious RecRes operator. In terms of query throughput, our unoptimized proof-of-concept implementation matches the throughput of Unbound, a state-of-the-art DNS-over-TLS recursive resolver, while incurring an acceptable increase in latency (due to the use of a TEE). To quantify the potential for privacy leakage through traffic analysis, we performed an Internet measurement study that showed that 94.7% of the top 1,000,000 domain names can be served from a *privacy-preserving* ANS that serves at least two distinct domain names, and 65.7% from an ANS that serves 100+ domain names. As future work, we plan to port the Unbound RecRes to Intel SGX and conduct a performance comparison with PDoT, as well as to explore methods for improving PDoT's performance using caching while maintaining client privacy. We also plan to investigate supporting DNS-over-HTTPS.

ACKNOWLEDGMENTS

We thank Geonhee Cho for the initial data collection for the privacy-preserving ANS analysis in Section 8.1. We are also grateful to the article's shepherd, Roberto Perdisci, and the anonymous ACSAC'19 and DTRAP reviewers for their valuable comments.

REFERENCES

[1] 2009. Introduction to DNSCurve. Retrieved May 29, 2019 from https://dnscurve.org/index.html.

[2] G. Acs, M. Conti, P. Gasti, C. Ghali, and G. Tsudik. 2013. Cache privacy in named-data networking. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*. 41–51.

[3] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. 2019. S-FaaS: Trustworthy and accountable function-as-a-service using Intel SGX. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW'19)*.

[4] Darren Anstee. 2020. Disappearing DNS: DoT and DoH, Where One Letter Makes a Great Difference. Retrieved May 15, 2020 from https://www.securitymagazine.com/articles/91674-disappearing-dns-dot-and-doh-where-one-letter-makes-a-great-difference.

[5] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. 2005. *DNS Security Introduction and Requirements*. Technical Report. DOI: https://doi.org/10.17487/rfc4033

[6] ARM. 2009. ARM Security Technology—Building a Secure System using TrustZone Technology. Retrieved May 29, 2019 from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html.

[7] S. Bortzmeyer. 2016. *DNS Query Name Minimisation to Improve Privacy*. Technical Report. DOI: https://doi.org/10.17487/RFC7816

[8] S Bortzmeyer. 2018. Encryption and authentication of the DNS resolver-to-authoritative communication. Retrieved from https://tools.ietf.org/html/draft-bortzmeyer-dprive-resolver-to-auth-01.

[9] Jon Brodkin. 2020. Firefox turns encrypted DNS on by default to thwart snooping ISPs. Retrieved May 15, 2020 from https://arstechnica.com/information-technology/2020/02/firefox-turns-encrypted-dns-on-by-default-to-thwart-snooping-isps/.

[10] Sergio Castillo-Perez and Joaquin Garcia-Alfaro. 2008. *Anonymous Resolution of DNS Queries*. Springer, Berlin, 987–1000. DOI: https://doi.org/10.1007/978-3-540-88873-4_5

[11] V. G. Cerf. 1991. *Guidelines for Internet Measurement Activities*. Technical Report. DOI: https://doi.org/10.17487/rfc1262

[12] Cloudflare. DNS over TLS—Cloudflare Resolver. Retrieved May 29, 2019 from https://1.1.1.1/dns/.

[13] Cloudflare. [n.d.]. 1.1.1.1 Resolver Examination Report. Retrieved from https://www.cloudflare.com/compliance/.

[14] Cloudflare. [n.d.]. Announcing 1.1.1.1: The Fastest, Privacy-first Consumer DNS Service. Retrieved from https://blog.cloudflare.com/announcing-1111/.

[15] Cloudflare. [n.d.]. Announcing the Results of the 1.1.1.1 Public DNS Resolver Privacy Examination. Retrieved from https://blog.cloudflare.com/announcing-the-results-of-the-1-1-1-1-public-dns-resolver-privacy-examination/.

[16] Manuel Costa, Lawrence Esswood, Olga Ohrimenko, Felix Schuster, and Sameer Wagh. 2017. The pyramid scheme: Oblivious RAM for trusted processors. *arXiv:1712.07882*. Retrieved from https://arxiv.org/abs/1712.07882.

[17] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. *Sanctum: Minimal Hardware Extensions for Strong Software Isolation*. 857–874. https://www.usenix.org/conference/usenixsecurity16/technicalsessions/presentation/costan.

[18] cs.nic. 2019. Knot Resolver. Retrieved May 29, 2019 from https://www.knot-resolver.cz/.

[19] T. Dierks and E. Rescorla. 2008. *The Transport Layer Security (TLS) Protocol Version 1.2*. Technical Report. DOI: https://doi.org/10.17487/rfc5246

[20] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. 2017. LightBox: Full-stack protected stateful middlebox at lightning speed. arxiv:1706.06261. Retrieved from http://arxiv.org/abs/1706.06261.

[21] D. Eastlake. 2013. *Domain Name System (DNS) IANA Considerations*. Technical Report. DOI: https://doi.org/10.17487/rfc6895

[22] Annie Edmundson, Paul Schmitt, and Nick Feamster. 2018. ODNS: Oblivious DNS. Retrieved May 29, 2019 from https://odns.cs.princeton.edu/.

[23] Hannes Federrath, Karl-Peter Fuchs, Dominik Herrmann, and Christopher Piosecny. 2011. *Privacy-preserving DNS: Analysis of Broadcast, Range Queries and Mix-based Protection Methods*. Springer, Berlin, 665–683. DOI: https://doi.org/10.1007/978-3-642-23822-2_36

[24] David Goltzsche, Signe Rusch, Manuel Nieke, Sebastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Cosa, Christof Fetzer, Pascal Felber, Peter Pietzuch, and Rudiger Kapitza. 2018. EndBox: Scalable middlebox functions using client-side trusted execution. In *Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'18)*. IEEE, 386–397. DOI: https://doi.org/10.1109/DSN.2018.00048

[25] Google. 2018. DNS over TLS Support in Android P Developer Preview. Retrieved May 29, 2019 from https://security.googleblog.com/2018/04/dns-over-tls-support-in-android-p.html.

[26] P. Hoffman. 2018. *DNS Queries Over HTTPS (DoH)*. Technical Report. DOI: https://doi.org/10.17487/rfc8484

[27] P. Hoffman and P. McManus. 2018. DNS Queries over HTTPS (DoH). DOI: https://doi.org/10.17487/RFC8484

[28] Rebekah Houser, Zhou Li, Chase Cotton, and Haining Wang. 2019. An investigation on information leakage of DNS over TLS. In *Proceedings of the 15th International Conference on Emerging Networking Experiments and Technologies (CoNEXT'19)*. Association for Computing Machinery, Inc., New York, NY, 123–137. DOI: https://doi.org/10.1145/3359989.3365429

[29] Zi Hu, Liang Zhu, John Heidemann, Allison Mankin, Duane Wessels, and P. Hoffman. 2016. *Specification for DNS over Transport Layer Security (TLS)*. Technical Report. DOI: https://doi.org/10.17487/RFC7858

[30] Tommy Jensen, Ivan Pashov, and Gabriel Montenegro. 2019. Windows Will Improve User Privacy with DNS over HTTPS. Retrieved May 15, 2020 from https://techcommunity.microsoft.com/t5/networking-blog/windows-will-improve-user-privacy-with-dns-over-https/ba-p/1014229.

[31] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. Integrating remote attestation with transport layer security. arxiv:1801.05863. Retrieved from http://arxiv.org/abs/1801.05863.

[32] SPROUT Lab. 2019. PDoT Source Code. Retrieved from https://github.com/sprout-uci/PDoT.

[33] NLnet Labs. Stubby. Retrieved May 29, 2019 from https://dnsprivacy.org/wiki/display/DP/DNS+Privacy+Daemon+-+Stubby.

[34] NLnet Labs. Unbound. Retrieved May 29, 2019 from https://nlnetlabs.nl/projects/unbound/about/.

[35] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE, 605–622. DOI: https://doi.org/10.1109/SP.2015.43

[36] Chaoyi Lu, Baojun Liu, Zhou Li, Shuang Hao, Haixin Duan, Mingming Zhang, Chunying Leng, Ying Liu, Zaifeng Zhang, and Jianping Wu. 2019. An end-to-end, large-scale measurement of DNS-over-encryption: How far have we come? In *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*. Association for Computing Machinery, New York, NY, 22–35. DOI: https://doi.org/10.1145/3355369.3355580

[37] Y. Lu and G. Tsudik. 2010. Towards plugging privacy leaks in the domain name system. In *Proceedings of the 2010 IEEE 10th International Conference on Peer-to-Peer Computing (P2P'10)*. IEEE, 1–10. DOI: https://doi.org/10.1109/P2P.2010.5569976

[38] Majestic. 2012. Majestic Million. Retrieved from https://blog.majestic.com/development/majestic-million-csv-daily/.

[39] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*. ACM Press, New York, New York, 1 page. DOI: https://doi.org/10.1145/2487726.2488368

[40] Microsoft. 2017. Introducing Azure Confidential Computing. Retrieved May 29, 2019 https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/.

[41] P. V. Mockapetris. 1987. *Domain Names—Implementation and Specification*. Technical Report. DOI: https://doi.org/10.17487/rfc1035

[42] DNSCrypt Project. 2019. DNSCrypt. Retrieved May 29, 2019 from https://dnscrypt.info/.

[43] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2017. ZeroTrace: Oblivious memory primitives from Intel SGX. *IACR Cryptology ePrint Archive* 2017 (2017), 549.

[44] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*.

[45] Haya Shulman and Haya. 2014. Pretty bad privacy: Pitfalls of DNS encryption. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society (WPES'14)*. ACM Press, New York, NY, 191–200. DOI: https://doi.org/10.1145/2665943.2665959

[46] Sandra Siby, Marc Juarez, Claudia Diaz, Narseo Vallina-Rodriguez, and Carmela Troncoso. 2020. Encrypted DNS –> Privacy? A traffic analysis perspective. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'20)*. DOI: https://doi.org/10.14722/ndss.2020.24301 arxiv:1906.09682

[47] Sandeep Tamrakar, Jian Liu, Andrew Paverd, Jan-Erik Ekberg, Benny Pinkas, and N. Asokan. 2017. The circle game: Scalable private membership test using trusted hardware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS'17)*. DOI: https://doi.org/10.1145/3052973.3053006

[48] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2018. ShieldBox: Secure middleboxes using shielded execution. In *Proceedings of the Symposium on SDN Research (SOSR'18)*. ACM Press, New York, New York, 1–14. DOI: https://doi.org/10.1145/3185467.3185469

[49] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE, 640–656. DOI: https://doi.org/10.1109/SP.2015.45

[50] Fangming Zhao, Yoshiaki Hori, and Kouichi Sakurai. 2007. Analysis of privacy disclosure in DNS query. In *Proceedings of the 2007 International Conference on Multimedia and Ubiquitous Engineering (MUE'07)*. IEEE, 952–957. DOI: https://doi.org/10.1109/MUE.2007.84

[51] Fangming Zhao, Yoshiaki Hori, and Kouichi Sakurai. 2007. Two-servers PIR based DNS query scheme with privacy-preserving. In *Proceedings of the 2007 International Conference on Intelligent Pervasive Computing (IPC'07)*. IEEE, 299–302. DOI: https://doi.org/10.1109/IPC.2007.27

[52] Liang Zhu, Zi Hu, John Heidemann, Duane Wessels, Allison Mankin, and Nikita Somaiya. 2015. Connection-oriented DNS to improve privacy and security. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE, 171–186. DOI: https://doi.org/10.1109/SP.2015.18

Received May 2020; revised September 2020; accepted October 2020