PDoT: Private DNS-over-TLS with TEE Support

Yoshimichi Nakatsuka
University of California, Irvine
nakatsuy@uci.edu

Andrew Paverd*
Microsoft Research
andrew.paverd@ieee.org

Gene Tsudik
University of California, Irvine
gene.tsudik@uci.edu

ABSTRACT

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 paper, we propose a novel Private DNS-over-TLS (*PDoT*) architecture. *PDoT* includes a DNS Recursive Resolver (RecRes) that operates within a Trusted Execution Environment (TEE). Using *Remote Attestation*, DNS clients can authenticate, and receive strong assurance of trustworthiness of *PDoT* RecRes. We provide a proof-of-concept implementation of *PDoT* and use it to 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*.

KEYWORDS

Domain Name System, Privacy, Trusted Execution Environment

ACM Reference Format:
Yoshimichi Nakatsuka, Andrew Paverd, and Gene Tsudik. 2019. **PDoT**: Private DNS-over-TLS with TEE Support. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3359789.3359793

1 INTRODUCTION

The Domain Name System (DNS) [26] is a 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. [33] proposed a DNS design that runs over Transport Layer Security (TLS) connections [14]. DNS-over-TLS protects privacy of DNS queries and prevents man-in-the-middle (MiTM) attacks against DNS responses. [33] also demonstrated practicality of DNS-over-TLS in real-life applications. Several open-source recursive resolver (RecRes) implementations, including Unbound [8] and Knot Resolver [5], currently support DNS-over-TLS. In addition, commercial support for DNS-over-TLS has been increasing, e.g., Android P supports it [2] as does Cloudflare's `1.1.1.1` RecRes [1]. However, despite attracting interest in both academia and industry, some problems remain.

One major issue is 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 encrypted using TLS, RecRes must decrypt the DNS query in order to resolve it and contact the relevant NS-s. This allows RecRes to learn unencrypted DNS queries, which poses privacy risks if a malicuous RecRes misuses this 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"* [1]. Although helpful, this still requires users to trust the auditor and can only be used by operators who can afford an auditor.

The second problem is that clients authenticate RecRes using certificates. Certificate-based authentication is natural for websites, since the user (client) already knows the website and the certificate securely binds a public key to that website's URL. However, the same does not hold for DNS, since most users have little or no idea what DNS is, much less which resolver's organization is trustworthy. Therefore, ideally, we need a way to authenticate RecRes without any user involvement. One way to address this issue is by creating 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.

An alternative approach is to use *Remote Attestation* (RA) so that clients can check what software a given RecRes is running. In this context, the identity of RecRes is no longer relevant, since clients can make trust decisions based on how RecRes behaves. RA is one of the main features of modern hardware-based Trusted Execution Environments (TEEs), such as Intel Software Guard Extensions (SGX) [24] and ARM TrustZone [9]. Such TEEs are now widely available, with Intel CPUs after the 7th generation supporting SGX, and ARM Cortex-A CPUs supporting TrustZone. TEEs with RA capability are now also available in cloud services, such as Microsoft Azure [25]. In this paper, we use these features to address the two problems posed above. Specifically, our contributions are:

- We design a Private DNS-over-TLS (*PDoT*) architecture, the main component of which is a privacy-preserving DNS RecRes that operates within a commodity TEE. Running an RecRes inside a TEE prevents even the RecRes operator from learning clients' DNS queries, thus providing query privacy. Our RecRes design also addresses the authentication challenge by

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

ACSAC '19, December 9–13, 2019, San Juan, PR, USA

ACM ISBN 978-1-4503-7628-0/19/12.
https://doi.org/10.1145/3359789.3359793

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 [**?**]. Our results show that *PDoT* handles DNS queries without leaking information while achieving sufficiently low latency and offering acceptable throughput (Sections 5 and 6).
- In order to quantify privacy leakage via traffic analysis, we performed an Internet measurement study. It shows that 94.7% of 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 7).

2 BACKGROUND

2.1 Domain Name System (DNS)

DNS is a distributed system that translates host and domain names into IP addresses. DNS includes three types of entities: *Client Stub* (client), *Recursive Resolver* (RecRes), and *Name Server* (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, the 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 (TEE)

A Trusted Execution Environment (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 remote attestation (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*. 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 Processor Reserved Memory (PRM). It holds 4KB pages of Enclave Page Cache (EPC) that stores code and data that run in that enclave. PRM is protected by CPU to prevent non-enclave access to this memory region. Also, processes can enter and leave an enclave only through special functions: *ECALLs* and *OCALLs*, respectively. These functions are realized by adding special CPU instructions. Any illegal attempt to enter or leave without calling these functions forces an enclave to shut down.

Attestation Service. SGX provides two types of attestation: local and remote. Local attestation enables one enclave to attest another (running on the same 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 saying that the application enclave is indeed running the code that it claims to be running. Once it receives an attestation verification report, the verifier can make an informed trust decision about behavior of the attested enclave.

Side-Channel Attacks. SGX is vulnerable to several side-channel attacks [22, 30], and various mechanisms have been proposed [13, 27**?**] to mitigate them. 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 & REQUIREMENTS

3.1 Adversary Model

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

Our first type of adversary is a malicious RecRes operator who has full control over the physical machine, its OS and all applications, including 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 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 adversary type is a network adversary, which is strictly weaker than the malcious RecRes operator. In the passive case, this adversary can observe any packets that flow into and out of the RecRes. In the active case, this adversary can also modify and/or forge network packets. DNS-over-TLS alone (without *PDoT*) is sufficient to thwart a passive network adversary. However, since an active adversary could attempt to redirect clients to a malicious RecRes, clients need a efficient mechanism for authenticating the RecRes and determining whether it is trustworthy, which is the main contribution of *PDoT*.

For either adversary, we do not consider Denial-of-Service (DoS) attacks against 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 [33].

3.2 System Requirements

In the context of the aforementioned adversary model, we now define system requirements for a privacy-preserving RecRes:

- **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 convey information, such as source and destination IP addresses. In Section 7.1, 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. Only minimal software modifications should be imposed. Also, for the purpose of transition and compatibility, a privacy-preserving RecRes should be able to effectively interact with legacy clients that support 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: the query privacy guarantees provided by *PDoT* rely on the forward-looking assumption that the communication between RecRes and the respective NS-s will also be protected by DNS-over-TLS. The DNS Privacy (DPrive) Working Group is working towards a standard for encryption and authentication of DNS resolver-to-authoritative communication [**?**], using essentially the same mechanism as DNS-over-TLS. We expect an increasing number of NS-s to begin to support this standard in the near future. Once *PDoT* is enabled at the RecRes, it can provide incremental query privacy for any queries served from a DNS-over-TLS NS. As we explain in Section 5, with small design modifications, *PDoT* could be adapted for use in NS-s.

4 SYSTEM MODEL & 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 running inside TEE is genuine and runs code that it trusts. (4) Client proceeds with the rest of the TLS handshake procedure only if verification succeeds. (5) Client sends a DNS query to the RecRes through the secure TLS channel it has just set up. (6) RecRes receives a DNS query from client (in its secure memory) and learns the domain name that the client wants to resolve. (7) RecRes sets up a secure TLS channel to appropriate NS in order to resolve the DNS query. (8) RecRes sends a DNS query to NS over that channel. If NS's reply includes an IP address of the next NS, 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 it 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*.

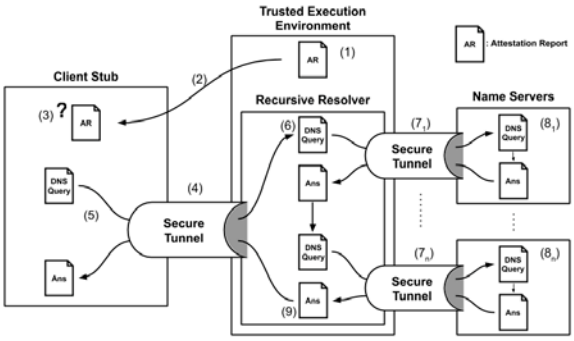


Figure 1: Overview of the proposed system.

4.2 Design Challenges

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

- **C1: TEE Limited Functionality.** In order to satisfy their security requirements, TEE-s often limit the functionality provided to applications that run 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.** Typically, the amount of memory that a TEE can use is small. The size of an SGX enclave is virtually as large as the size of the host machine's memory. However, this is realized through page swapping, which itself requires additional instructions. Moreover, the page to be swapped must be encrypted due to SGX enclave security requirements of SGX, thus adding even more instructions. Therefore, introducing page swapping places heavy burden on the performance of the application in the enclave. To avoid page swapping, enclave size should be the same as EPC of the Intel CPU – typically, 128MB. Since RecRes is a performance-critical application, its size should ideally not exceed 128MB. This limit negatively impacts RecRes throughput, as it bounds the number of threads that can spawn in TEE.
- **C3: TEE Call-in/Call-out Overhead.** Applications that require functionality that is not present within the TEE must switch to the non-TEE side. This introduces additional overhead since switching between TEE and non-TEE "worlds" requires additional instructions. Identifying and limiting the number of times RecRes switches back and forth (while keeping RecRes functionality correct) is a substantial challenge.

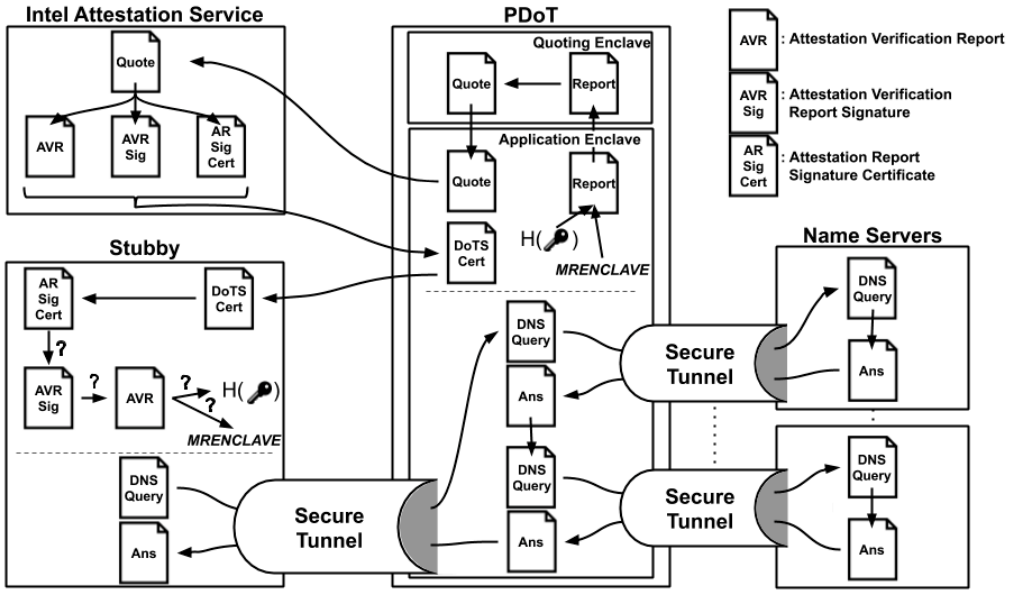


Figure 2: Overview of *PDoT* implementation.

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 to use off-the-shelf Intel SGX as the platform for the proof-of-concept *PDoT* implementation in order to support an accurate performance evaluation on real hardware (Section 6). This means that our implementation is subject to the performance and memory constraints in the current version of Intel SGX, and is thus best suited for small-scale networks (e.g., the public WiFi network provided by a coffee shop). However, as TEE technology advances, we expect that our design will be able to scale to larger networks.

5.1 *PDoT*

PDoT consists of two parts: (1) trusted part residing in TEE enclaves, and (2) untrusted part that operates in the non-TEE region. 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 an 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 an 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 Intel Attestation Service (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 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. This certificate, in addition to the public key, includes: (1) attestation verification report, (2) attestation verification report signature, and (3) attestation report signing certificate (extracted from (1)). 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. 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. (For more detail, see Section 5.2). Only if the authentication and validation succeed, the client resumes the handshake process.

DNS Query Resolving Process. The client sends a DNS query over the TLS channel established earlier. RecRes receives a DNS query from the client, decrypts it within the application enclave and obtains the target name. It starts 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

[1] In implementing this process, we heavily relied on SGX RA TLS [20] 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.

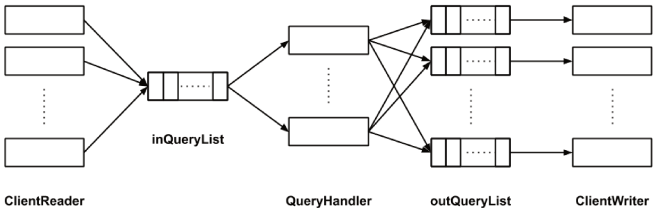


Figure 3: Overview of *PDoT* threading model.

channel. Once it receives the answer from NS, RecRes returns to the client, also over a TLS channel.

Figure 3 illustrates how *PDoT* divides DNS query resolution process into three threads: (1) receiving DNS query – *ClientReader*, (2) resolving it – *QueryHandler*, and (3) returning the answer – *ClientWriter*.

ClientReader and ClientWriter threads are spawned anew upon each query. Dividing receiving and sending processes and giving them a dedicated thread is helpful because many clients send multiple DNS queries within a short timespan without waiting for the answer to the previous query.[3] When ClientReader thread receives a DNS query from the client, it stores the query and a client ID in a FIFO queue, called *inQueryList*.

QueryHandler threads are spawned when *PDoT* starts up. The number of QueryHandler threads is configured by RecRes operator. QueryHandler threads are shared among all current ClientReader and ClientWriter threads. When a QueryHandler thread detects an entry in the inQueryList, it removes this entry and retrieves the query and the client ID. QueryHandler first checks whether this client is still accepting answers from RecRes. If not, QueryHandler simply ignores this query and moves on to the next one. If the client is still accepting answers, QueryHandler resolves the query and puts the answer into a FIFO queue (called *outQueryList*) dedicated to that specific client.

There might be cases when NS response is too slow. In such cases, QueryHandler thread gives up on resolving that particular query and moves on to the next query, because it is very likely that the request was dropped. This also prevents resources (such as mutex) from being locked up by this QueryHandler thread. In our implementation, this timeout was set to be the same as the client's timeout, since there is no point in sending the answer to the client after that.

Once an answer is added to outQueryList dedicated to its client, ClientWriter uses that answer to compose a DNS reply packet and sends it to the client. The reason we have N outQueryLists for N clients is to improve performance. With only one outQueryList, ClientWriter threads must search through the queue to find the answer for the connected client. This takes $O(M \times N)$ time, where N is the number of clients and M is the number of queries each client sends. Instead, with N outQueryLists, we reduce complexity to $O(1)$ because ClientWriter thread merely selects the query at the head of the list.

[3]For example, a client has received a webpage that includes images and advertisements that are served from servers located at different domains. This triggers multiple DNS queries at the same time.

Caching. Some DNS recursive resolvers also provide the ability to cache results on the resolver. Caching can be beneficial from the clients' perspective because if the answer to a query is already cached, the RecRes can send the answer immediately, thus reducing query latency. The RecRes also benefits from not having to establish connections to external NS-s in order to answer the query. However, irrespective of how it is implementated, caching at the resolver introduces potential privacy leakage (e.g., timing measurements can reveal whether or not a certain domain was already in the cache). This is an orthogonal challenge, which we discuss in Section 7.2.

To explore the possibility of caching in a privacy-preserving resolver, we implemented a simple in-enclave cache for *PDoT*. It uses a red-black tree data structure and stores all records associated to the client's query, indexed by the queried domain. This provides $O(log_2(N))$ access times with N entries in the cache. In production, *PDoT* could also use existing techniques to mitigate against side-channel attacks on the cache's memory access patterns (e.g., [**? ? ?**]). During remote attestation, clients can ascertain whether the resolver has enabled caching, and which mitigations it uses.

***PDoT* ANS with TEE support.** With minor design changes, our *PDoT* RecRes design can be modified for use as an ANS. Similarly to the caching mechanism described above, a *PDoT* ANS looks up the answers to queries from an internal database, rather than contacting external NS-s. In the same way that clients authenticate a *PDoT* RecRes, the RecRes can authenitcate the *PDoT* ANS. Clients can thus establish the trustworthiness of both the RecRes and ANS using *transitive attestation* [**?**].

5.2 Client with *PDoT* Support

We took the Stubby client stub from the `getdns` project [7] which offers DNS-over-TLS support and modified it so that it can perform remote attestation during the TLS handshake. In this section, we 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 there is, the client reuses it. If not, it attempts to establish a new TLS 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 we assume that the client trusts 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 a copy from (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, 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 are various possible options for deciding which MRENCLAVE values are trustworthy. For example, the recursive resolver vendors could publish lists of expected MRENCLAVE values for their resolvers. For open source resolvers like *PDoT*, anyone can recalculate 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 the 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, 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 TEE is a challenge for *PDoT* 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, this processes might redirect the packet to a malicious process or simply drop it. We discuss this issue in Section 6.1. 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 RecRes machine. This allows *PDoT* to use all available EPC memory. Second, we fix the number of QueryHandler threads in order to save space. This is possible because of dis-association of QueryHandler and ClientReader/Writer threads.

OCALL and ECALL Overhead. ECALLs and OCALLs require additional instructions 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 ClientReader thread – DNS query from the client, for QueryProcessor thread – query from inQueryList, and for ClientWriter thread – response from outQueryList. *PDoT* was implemented so that these threads wait inside the enclave. If we were to wait outside the enclave, we would have to make an ECALL to enter the enclave each time the thread proceeds.

6 EVALUATION

6.1 Security Analysis

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

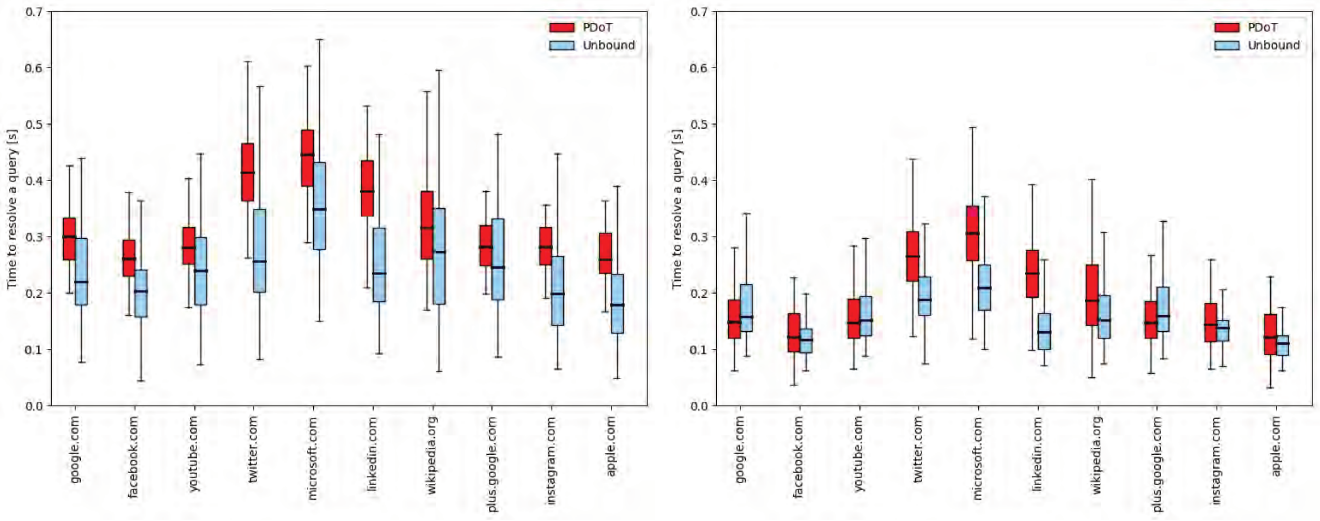
However, a malicious RecRes operator may attempt to connect the socket to a malicious TLS server that resides in either: 1) an untrusted region, or 2) a separate enclave that the operator itself created. If the operator can trick the client into establishing a TLS connection with the malicious TLS server, the adversary can obtain the 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 but 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 the 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 to a packet sent to an NS. The consequent amount of privacy leakage is discussed in Section 7.1

6.2 Deployability

Section 5 explains how *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** 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 stub resolvers to support *PDoT*. In the same way that browser vendors currently include and maintain a list of trusted root CA certificates in their browsers, browser/OS vendors 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, so even allowing for multiple versions for each, the list of trusted MRENCLAVE values will be orders of magnitude smaller than the list of public keys of every trusted resolver, as would be required for standard DNS-over-TLS.



(a) Latency of *PDoT* and Unbound (Cold Start)

(b) Latency of *PDoT* and Unbound (Warm Start)

Figure 4: Latency comparison of *PDoT* and Unbound

6.3 Performance Evaluation

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. We used Ubuntu 16.04 and the Intel SGX SDK version 2.2. We configured our RecRes to support up to 50 concurrent clients and process queries using 30 QueryHandle threads. For comparison, we performed the same benchmarks using Unbound [8], a popular open source RecRes.

6.3.1 Latency Evaluation. The objective of our latency evaluation is 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 with Unbound. To meet requirement R3, *PDoT* should not incur a significant increase in latency compared to Unbound.

Experimental Setup. The client and RecRes ran on the same physical machine to remove network delay. We conducted the experiment using *PDoT* and Unbound as the RecRes, and Stubby as the client. We measured latency under two different 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, the cold start measurements also include the time required to establish the TLS connection. In this experiment, the caching mechanisms of both *PDoT* and Unbound were disabled.

We created a python program to feed DNS queries to the client. The program sends 100 queries sequentially for ten different domains. That is, the program waits for an answer to the previous query before sending the next query. We used the top ten domains of the Majestic Million domain list [6].

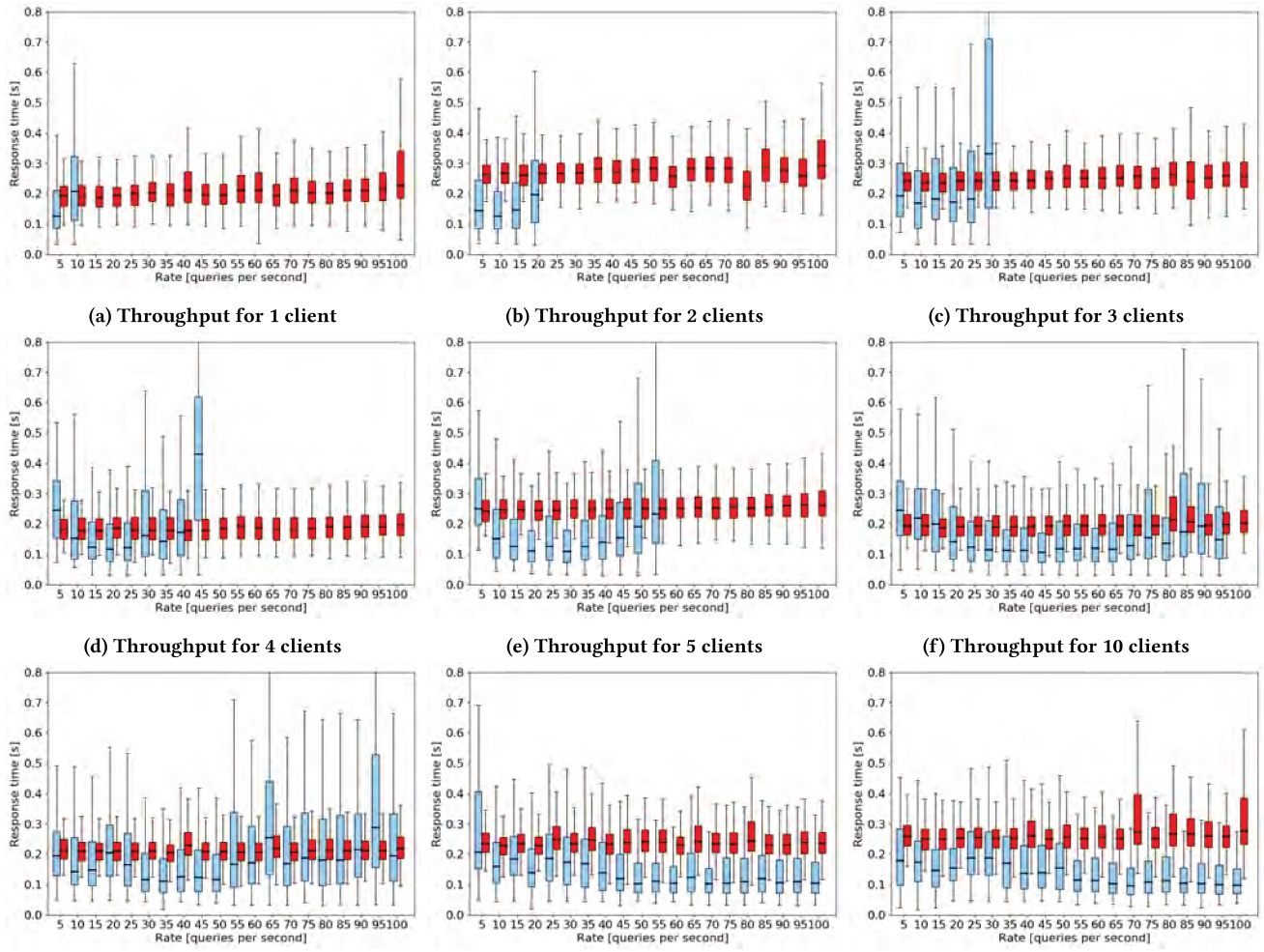
The python program measures the time between sending the query and receiving an answer. For the cold start experiment, we spawned a new Stubby client and established a new TLS connection for each query. In the warm start scenario, we first established the TLS connection by sending a query for another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

Note that the numeric latency values we obtained are specific to our experimental setup because they depend on the network bandwidth of our RecRes, and the latency between our RecRes and the 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 the average latency over a large set of domains. Instead we took multiple measurements for each of a small set of domains (e.g., 100 measurements for each of 10 domains) so that we could analyse the range of response latencies for each domain.

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

For the cold-start case (Figure 4a), although Unbound is typically faster than our proof-of-concept *PDoT* implementation, the range of latencies is similar. For 7 out of 10 domains, the upper whisker of *PDoT* was lower than that of Unbound. Overall, *PDoT* shows an average 22% overhead compared to Unbound in the cold-start setting.

For the warm-start case (Figure 4b), we see that 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 9% overhead compared to Unbound. In practice, once the client has established a connection to RecRes, it will maintain this connection, and thus the vast majority of queries will see only the warm-start latency.



(a) Throughput for 1 client (b) Throughput for 2 clients (c) Throughput for 3 clients

(d) Throughput for 4 clients (e) Throughput for 5 clients (f) Throughput for 10 clients

(g) Throughput for 15 clients (h) Throughput for 20 clients (i) Throughput for 25 clients

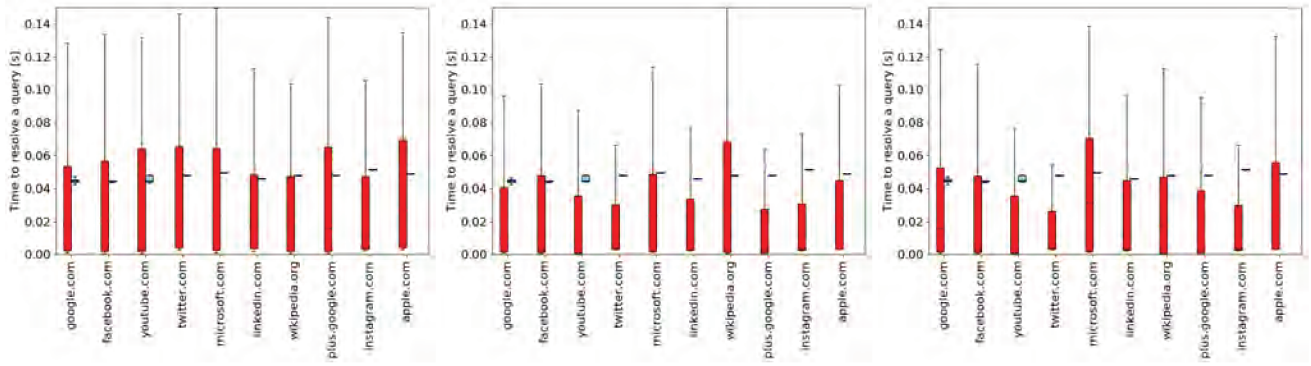
Figure 5: Throughput comparison of *PDoT* (red) and Unbound (blue)

6.3.2 Throughput evaluation. The objective of 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 fulfill requirement R4.

Experiment setup. The client and RecRes were run on different machines, so that the RecRes could use all available resources of a single machine. This is representative of a local RecRes running in a small network (e.g., a coffee shop WiFi network). We conducted this experiment using the same two RecRes-s as in the latency experiment. Stubby was configured to reuse TLS connections. To simulate a small to medium-scale network, we varied the number of concurrent clients between 1 and 25 and adjusted the query arrival rate from 5 to 100 queries per second. Query rates were uniformly distributed among the clients, e.g., for an overall rate of 100 queries per second with 10 clients, each client sends 10 queries per second. To eliminate any variability in resolving the query, all queries were for the domain `google.com`. We maintained each constant query rate for one minute. In this experiment, the caching mechanisms of both *PDoT* and Unbound were disabled.

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 axes shows different query rates and vertical axis shows the range of response latencies for each query rate. Measurement are plotted using the same box plot arrangement as in the latency evaluation. Blue boxes show the result for Unbound and red boxes show the result for *PDoT*.

If queries arrive faster than RecRes can process them, the queries will start to build up in a queue, and the latency of each successive response will increase as the length of the queue grows. In this case, the average latency will continue to increase indefinitely until queries begin to timeout or RecRes runs out of memory. Therefore we say this rate of query arrival is *unsustainable*. On the other hand, if RecRes can sustain the rate of query arrival, the average response latency will remain roughly constant irrespective of how long RecRes runs. For this experiment, we define a *sustainable* rate



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

Figure 6: Latency comparison of *PDoT* (red) and Unbound (blue) with different number of domains in cache

of query arrival as one for which the average response latency is constant over time, and below one second (well below a typical DNS client timeout). Figure 5 only shows cases in which the query arrival rate is sustainable for the respective RecRes. In other words, the presence of a box in Figure 5 shows that the RecRes can achieve that level of throughput.

Surprisingly, we observed that Unbound cannot handle query rates exceeding 10 queries per second per client (i.e., Unbound's maximum sustainable rate was $10n$ queries per second distributed among n clients). This is because Unbound's design only uses one query processing thread per client. In contrast, *PDoT* was able to handle more than 100 queries per second in all cases because our design uses a separate pool of QueryHandle threads.

Overall, Figure 5 confirms that our proof-of-concept implementation achieves at least the same throughput as Unbound across the range of clients and query arrival rates, and can achieve higher throughput when the number of clients is low. Although Unbound again achieves slightly lower latency, this is consistent with our latency measurements in Section 6.3.1 and is likely due to that fact that Unbound is an optimized production-grade RecRes.

6.3.3 Caching evaluation. We evaluated the performance of both resolvers with caching enabled; Unbound with its default caching behavior, and *PDoT* with our simple proof-of-concept cache.

Experiment setup. The experimental setup is similar to that of the latency evaluation described in Section 6.3.1. In this case, we pre-populated the resolvers' caches with varying numbers of domains and measured the response latency for a representative set of 10 popular domains.

Results and observations. As shown in Figure **??**, 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 a higher variability than Unbound. This is most likely due to the combination of our unoptimized caching implementation and the latency of accessing enclave memory. Nevertheless, Figure **??** shows that even with the memory limitations of current hardware enclaves, *PDoT* can still benefit from caching a small number of domains.

7 DISCUSSION

This section discusses some potential privacy issues in *PDoT*.

7.1 Information Revealed by IP Addresses

Even if the connections between the client, RecRes, and NS-s are all 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 in order to correlate packets sent from clients to the RecRes with subsequent packets sent from the RecRes to the 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 the 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 a DNS query for 1,000,000 domains and gathered information about ANS-s that can possibly provide the answer to that query. 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* [12], we limited our querying rate, in order to avoid placing undue load on any servers.

We used the Majestic Million domain list [6] to obtain 1,000,000 popular domain names. As shown in Figure 6, only 5.7% of the domains we queried were served by non-privacy-preserving ANS-s (i.e., ANS-s that hold only one record). Examples of domain names served from non-privacy-preserving ANS-s included: `tinyurl.com`[4], `bing.com`, `nginx.org`, `news.bbc.co.uk`, and `cloudflare.com`. On the other hand, 9 out of 10 queries were served by a privacy-preserving ANS, and 65.7% by ANS-s that hold over 100 records.

[4]Note that since `tinyurl.com` is a URL shortening service, this is actually still privacy-preserving because the adversary can not learn which short URL was queried.

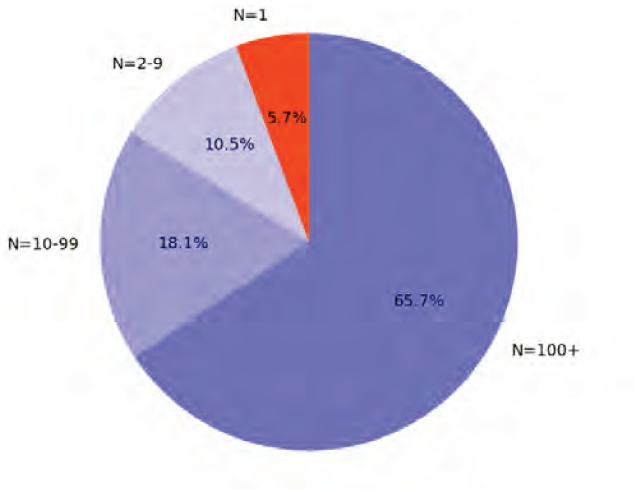


Figure 7: Percentage of Majestic Million domains answered by an ANS with at least N records

It is important to note that the above results and Figure 6 are still approximations. Since we do not have data for domains outside the Majestic Million list, we cannot make claims about whether or not 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. On the other hand, 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 the 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 against the ANS. Therefore, this type of analysis would have to be performed by an organization with privileged access to all ANS-s' zone files.

7.2 Caching & Timing attacks

Introducing a cache into an RecRes would allow the adversary to launch timing attacks and help guess the domain name queried by the end-user. We consider two types of timing attacks:

- **Measuring time between query and response.** This the simplest attack, whereby the adversary monitors the network between client and RecRes, and records the time for the RecRes to respond to the client. If the response time is shorter (compared to other queries), it likely has been served from a cache. This attack can be launched by both adversary types described in Section 3.1. One obvious countermeasure is to artificially delay the response, so that it matches the latency of NS-served responses.
- **Correlating client and RecRes requests.** To counter the above countermeasure, an adversary may attempt to correlate DNS requests sent from the client to the RecRes with those sent from RecRes to NS-s. This can be done by measuring time the two packets were sent. If the adversary manages to succeed, it can distinguish requests that involve contacting an NS from those that do not. Responses to the latter type must be served from the cache. This attack can be also launched by a malicious RecRes or a network sniffer. One way to counter it is to always send a query to the NS, but this negates all benefits of caching. Another way to counter this is by randomly mixing requests to the NS.

Considering the two types of timing attacks discussed above, the information leakage depends on whether the adversary is passive or active. The former can (at most) guess the domain name. If the caching strategy is Most Recently Used (MRU), the domain name must be one of the popular ones. The active adversary that generates its own DNS queries, has better chances of guessing the end-user's query target. This adversary can query a wide range of domain names and keep a list of those that result in cache hits.

8 RELATED WORK

There have been many work to protect the privacy of DNS queries [11, 16, 17, 23, 28, 31, 32]. For instance, Lu et al. [23] proposed a privacy-preserving DNS that uses distributed hash tables, different naming scheme, and computational private information retrieval method. Federrath et al. [17] introduces a dedicated DNS Anonymity Service to protect the DNS query using an architecture that distributes the top domains by broadcast and uses low-latency mixes for requesting the remaining domains. These scehems all assume that the involved parties do not act maliciously.

There have also been activities in the Internet standards community that aim to protect the security and privacy of DNS. DNS Security Extensions (DNSSEC) [21] provides data origin authentication and integrity to the DNS by using public key cryptography, but no privacy. Bortzmeyer [10] proposed a scheme to enhance the privacy of DNS queries by revealing the sections of the domain name that is only necessary to resolve the DNS query. Additionally, though they are not accepted as Internet standards, several protocols have been proposed that encrypt and authenticates DNS packets between the client and the RecRes (DNSCrypt [3]) and RecRes and NS-s (DNSCurve [4]). Moreover, the original DNS-over-TLS paper has been drafted as an Internet standard [19]. All these methods assume that the RecRes operator can be trusted and does not attempt to learn anything from the DNS queries.

Moreover, there has been active research on establishing trust through TEEs to protect confidentiality and integrity of network functions. More specifically, SGX has been used to protect network functions, especially middle boxes. For example, Endbox [18] aims to distribute middle boxes at client edges that the clients connect through VPN to ensure the confidentiality of the client's traffic while remaining maintainable. LightBox [15] is another middle box that runs in an enclave, but they aim to protect the client's traffic from the third-party middle box service provider while maintaining the performance. Lastly, ShildBox [29] aims to protect confidential network traffic that flow through untrusted commodity servers and provide a generic interface for easy deployability. These work focus on protecting the security of confidential data that flows in the network, but do not target DNS queries in particular.

9 CONCLUSION & FUTURE WORK

In this paper, we propose *PDoT*, a novel design of a DNS RecRes 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). In order to quantify the potential for privacy leakage through traffic analysis, we performed an Internet measurement study which 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 Unbound to Intel SGX and conduct a performance comparison with *PDoT*, as well as explore methods to improve *PDoT*'s performance using caching while maintaining client privacy.

ACKNOWLEDGMENTS

The authors thank Geonhee Cho for the initial data collection for the privacy-preserving ANS analysis (Section 7.1). The authors also thank the paper's shepherd, Roberto Perdisci, and the anonymous reviewers for their valuable comments. 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.

REFERENCES

[1] [n. d.]. DNS over TLS - Cloudflare Resolver. https://developers.cloudflare.com/1.1.1.1/dns-over-tls/
[2] [n. d.]. DNS over TLS support in Android P Developer Preview. https://security.googleblog.com/2018/04/dns-over-tls-support-in-android-p.html
[3] [n. d.]. DNSCrypt. https://dnscrypt.info/
[4] [n. d.]. Introduction to DNSCurve. https://dnscurve.org/index.html
[5] [n. d.]. Knot Resolver. https://www.knot-resolver.cz/
[6] [n. d.]. Majestic Million. https://blog.majestic.com/development/majestic-million-csv-daily/
[7] [n. d.]. Stubby. https://dnsprivacy.org/wiki/display/DP/DNS+Privacy+Daemon+-+Stubby
[8] [n. d.]. Unbound. https://nlnetlabs.nl/projects/unbound/about/
[9] ARM. [n. d.]. ARM Security Technology - Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html
[10] S. Bortzmeyer. 2016. *DNS Query Name Minimisation to Improve Privacy.* Technical Report. https://doi.org/10.17487/RFC7816
[11] Sergio Castillo-Perez and Joaquin Garcia-Alfaro. 2008. Anonymous Resolution of DNS Queries. Springer, Berlin, Heidelberg, 987–1000. https://doi.org/10.1007/978-3-540-88873-4_5
[12] V.G. Cerf. 1991. *Guidelines for Internet Measurement Activities.* Technical Report. https://doi.org/10.17487/rfc1262
[13] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. , 857–874 pages. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan
[14] T. Dierks and E. Rescorla. 2008. *The Transport Layer Security (TLS) Protocol Version 1.2.* Technical Report. https://doi.org/10.17487/rfc5246
[15] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. 2017. LightBox: Full-stack Protected Stateful Middlebox at Lightning Speed. (Jun 2017). arXiv:1706.06261 http://arxiv.org/abs/1706.06261
[16] Annie Edmundson, Paul Schmitt, and Nick Feamster. [n. d.]. ODNS: Oblivious DNS. https://odns.cs.princeton.edu/
[17] 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, Heidelberg, 665–683. https://doi.org/10.1007/978-3-642-23822-2_36
[18] 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 *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE, 386–397. https://doi.org/10.1109/DSN.2018.00048
[19] Zi Hu, Liang Zhu, John Heidemann, Allison Mankin, Duane Wessels, and P Hoffman. 2016. Specification for DNS over Transport Layer Security (TLS). https://doi.org/10.17487/RFC7858
[20] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. Integrating Remote Attestation with Transport Layer Security. (Jan 2018). arXiv:1801.05863 http://arxiv.org/abs/1801.05863
[21] Matt Larson, Dan Massey, Scott Rose, Roy Arends, and Rob Austein. [n. d.]. DNS Security Introduction and Requirements. ([n. d.]). https://tools.ietf.org/html/rfc4033
[22] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy.* IEEE, 605–622. https://doi.org/10.1109/SP.2015.43
[23] Y. Lu and G. Tsudik. 2010. Towards Plugging Privacy Leaks in the Domain Name System. In *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P).* IEEE, 1–10. https://doi.org/10.1109/P2P.2010.5569976
[24] 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, USA, 1–1. https://doi.org/10.1145/2487726.2488368
[25] Microsoft. [n. d.]. Introducing Azure confidential computing. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/
[26] P.V. Mockapetris. 1987. *Domain names - implementation and specification.* Technical Report. https://doi.org/10.17487/rfc1035
[27] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS Symposium.* https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/t-sgx-eradicating-controlled-channel-attacks-against-enclave-programs/
[28] 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, New York, USA, 191–200. https://doi.org/10.1145/2665943.2665959
[29] 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, USA, 1–14. https://doi.org/10.1145/3185467.3185469
[30] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy.* IEEE, 640–656. https://doi.org/10.1109/SP.2015.45
[31] Fangming Zhao, Yoshiaki Hori, and Kouichi Sakurai. 2007. Analysis of Privacy Disclosure in DNS Query. In *2007 International Conference on Multimedia and Ubiquitous Engineering (MUE'07).* IEEE, 952–957. https://doi.org/10.1109/MUE.2007.84
[32] Fangming Zhao, Yoshiaki Hori, and Kouichi Sakurai. 2007. Two-Servers PIR Based DNS Query Scheme with Privacy-Preserving. In *The 2007 International Conference on Intelligent Pervasive Computing (IPC 2007).* IEEE, 299–302. https://doi.org/10.1109/IPC.2007.27
[33] Liang Zhu, Zi Hu, John Heidemann, Duane Wessels, Allison Mankin, and Nikita Somaiya. 2015. Connection-Oriented DNS to Improve Privacy and Security. In *2015 IEEE Symposium on Security and Privacy.* IEEE, 171–186. https://doi.org/10.1109/SP.2015.18