Continuous User Behavior Monitoring using DNS Cache Timing Attacks

Hannes Weissteiner*, Roland Czerny*, Simone Franza*, Stefan Gast*, Johanna Ullrich† and Daniel Gruss*
*Graz University of Technology
Graz, Austria
Email: {firstname.lastname}@tugraz.at
†University of Vienna
Vienna, Austria
Email: johanna.ullrich@sba-research.org

***Abstract*—The Domain Name System (DNS) is a core component of the Internet. Clients can query DNS servers to translate domain names to IP addresses. Local DNS caches can alleviate the time it takes to query a DNS server, reducing delays to connection attempts. Prior work showed that DNS caches can be exploited via timing attacks to test whether a user has visited a specific website recently but lacked eviction capabilities, *i.e.*, could not monitor when precisely a user accessed a website, others focused on DNS caches in routers. All prior attacks required some form of code execution (e.g., native code, Java, or JavaScript) on the victim's system, which is also not always possible.**

We introduce DMT, a novel Evict+Reload attack to continuously monitor a victim's Internet accesses through the local DNS cache. The foundation of DMT is reliable DNS cache eviction: We present 4 DNS cache eviction techniques to evict the local DNS cache in unprivileged and sandboxed native attacks, virtualized cross-VM attacks, as well as browser-based attacks, *i.e.*, a website with JavaScript and a scriptless attack exploiting the serial loading of fonts integrated in websites. Our attack works both in default settings and when using DNS-over-TLS, DNSSEC, or non-default DNS forwarders for security. We observe eviction times of 77.267 ms on average across all contexts, using our fastest eviction primitive and reload and measurement times of 685.86 ms on average in the best case (cross-VM attack) for 100 domains and 14.710 s on average in the worst case (JavaScript-based attack). Hence, the blind spot of our attack for a granularity of five minutes is smaller than 0.26 % in the best case, and 4.92 % in the worst case, resulting in a reliable attack. In an end-to-end cross-VM attack, we can detect website visits from a list of 103 websites (in an open-world scenario) reliably with an F_1 score of 92.48 % within less than one second. In our JavaScript-based attack, we achieve F_1 scores of 95.94 % and 84.93 % for detecting accesses to 3 websites, with and without DNSSEC, respectively. We argue that DMT leaks information valuable for extortion and scam campaigns, or to serve exploits tailored to the victim's EDR solution.

I. Introduction

The Domain Name System (DNS) is responsible for translating human-readable Internet domain names into numerical IP addresses before connection establishment [64], [65]. For this purpose, DNS resolution involves a slow multi-hop process of querying a recursive DNS server, which may contact multiple authoritative servers in a hierarchical manner, starting from DNS root servers, then top-level domain (TLD) servers, and finally the authoritative server for the specific domain [64]. DNS caches alleviate both the latency problem and reduce network traffic. These caches store recently resolved domain names, their associated IP addresses, and time-to-live (TTL) temporally limiting the entry's validity. Later requests for the same domain are answered immediately from the cache.

Caches are known to introduce side channels [72]. By probing the cache, an adversary can determine whether specific data is in the cache, which is recently visited domains in the case of the DNS caches [24], and allow to conclude on users' (potentially sensitive) web activity. Since caches may be shared on a local machine, in a router, or any system that is part of the DNS resolution chain, an attacker can exploit the cache to spy on users of the corresponding system.

DNS cache timing attacks [24], [29], [63] exploit these cache state differences through timing, *i.e.*, cache hits are faster than cache misses. Grangeia [29] summarized three methods to measure the cache state of a network DNS cache: (1) preventing recursive resolution, (2) setting of a low TTL, and (3) measuring the time it takes to resolve a domain name. Grangeia [29] points out that reading from the cache is a destructive operation, *i.e.*, the cache state is modified by reading from it and has to be reset before it can be exploited again. Felten and Schneider [24] were the first to discuss leakage of domain accesses from timings of the local DNS cache. Their work, more than 2 decades old, only performs a limited experiment to investigate leakage from the DNS cache. More specifically, they only perform a single, destructive measurement as evidence for the leakage. In contrast, we present a full Evict+Reload-style attack on the local DNS cache and scientifically evaluate success rates and reliability in a continuous monitoring scenario. Based on the publicly available abstract, recent work by Moav et al. [63] proposed a flush-reload attack against DNS forwarders to track users and IoT devices, which will become public at some point in August. However, their work on cache-based

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.242287
www.ndss-symposium.org

timing side channels on DNS is still under embargo, making a comparison with our work impossible as we do not have access to their paper. Current main-stream Linux distributions use `systemd-resolved`, which acts as a local DNS cache, as the default DNS resolver [15]. While flushing the DNS cache in `systemd-resolved` is an unprivileged operation, the corresponding `resolvectl` interface is not available in virtual machines, application sandboxes, and web browsers. However, reliable eviction from restricted environments, which is a prerequisite for an Evict+Reload-style attack [32] on the DNS cache to continuously monitor a victim's activity, has not been demonstrated yet.

In this paper, we introduce DNS Monitoring via Timing (DMT), a novel Evict+Reload attack on the victim's **local** DNS cache state from unprivileged and even sandboxed contexts. DMT continuously monitors a victim's activity by combining known timing primitives to probe the DNS cache with previously unknown **reliable DNS cache eviction** techniques: In total, we present 4 techniques to evict the local DNS cache from 3 restricted environments, including virtual machines, and websites with and without JavaScript. We also introduce three measurement primitives, allowing us to measure DNS resolution timing from native code, JavaScript, and scriptless HTML respectively. The combination of these techniques allows us to probe and evict the DNS cache 3 times per minute, allowing us to track user behavior with a high temporal granularity. We compare our other eviction techniques with the use of the `resolvectl flush-caches` command, which is not available in restricted evironments, and achieve a similar reliability. Only one of our primitives exploits specific behavior in `systemd-resolved` for flushing. The other 3 exploit regular DNS cache behavior, and are hence applicable to other DNS resolvers as well.

DMT can be mounted remotely by an off-path attacker, via malicious JavaScript code, or even with plain HTML. We perform only a single timing measurement to determine whether a distinct domain was accessed. Since we do not rely on transmited content, data, or any other website characteristics, DMT does not qualify as a fingerprinting attack. Changes or region-specific adaptations to the target website do not affect the attack's reliability. In fact, when operating from a browser, DMT benefits from strict *Cross Origin Resource Sharing* (CORS) policies, as they cause the request to fail after the headers are fetched, but before any content is loaded, reducing noise in our measurements. When operating from native code, no connection is made to the target server at all. Alltogether, this makes DMT less susceptible to misclassification compared to classic website fingerprinting attacks [35].

We evaluate our attacks with default settings, DNS-over-TLS, and DNSSEC, in unprivileged and sandboxed native attacks, virtualized cross-VM attacks, as well as browser-based attacks. The latter encompasses a website with JavaScript, and also scriptless attack exploiting the serial loading of fonts integrated in websites. We show that the timing differences between cached and uncached domain names are significant, and are reliably detected in almost all cases when performing the attack from native code using system-provided DNS APIs. In absence of direct access to system-wide APIs, DMT leverages other, widely available APIs (e.g., JavaScript `fetch`) triggering implicit domain resolutions. In these experiements, we achieved a worst-case false-negative rate of $75.8\,\%$ with a single measurement. Execution times vary depending on the scenario: In the best case, the cross-VM attack, our attack takes, on average, $763.122\,\text{ms}$ to monitor 100 domains, and each additional domain takes another $2.931\,\text{ms}$. In the worst case, the JavaScript-based attack, it takes $14.787\,\text{s}$ for 100 domains, and an increase of $147.110\,\text{ms}$ per additional domain.

For our experimental end-to-end attack, we execute unprivileged native code *within a virtual machine*, to detected accessed domains by the host from a list of 103 websites (in an open world scenario). We achieved a F_1 score of $92.48\,\%$ in $685.855\,\text{ms}$ ($n = 3000$, $\sigma_{\bar{x}} = 14.893\,\text{ms}$). If the DNS server returns errors, subsequent eviction takes either $77.267\,\text{ms}$ ($n = 2562$, $\sigma_{\bar{x}} = 0.287\,\text{ms}$) (in case the DNS server reports errors), and $5\,\text{s}$ if it times out. We demonstrate that DMT works even without JavaScript by using CSS to trigger sequential network requests, achieving a reliability of up to $87.5\,\%$. Beyond that, we observe that DNS-over-TLS has no effect and DNSSEC even improves reliability of our attack, reducing the false-negative rate by $11\,\%$ and achieving a worst-case false-negative rate of $13.6\,\%$. Finally, we discuss that DMT works within a VPN setting.

Contributions. In summary, we make the following contributions:

- We introduce DMT, a novel Evict+Reload attack on the victim's **local** DNS cache, based on **reliable DNS cache eviction**: We present 4 DNS cache eviction techniques to evict the local DNS cache from 3 restricted environments, including unprivileged native code (even in virtual machines), and websites with and without JavaScript.
- Our attack works with default settings, DNS-over-TLS, and DNSSEC, in unprivileged and sandboxed native attacks, virtualized cross-VM attacks, as well as browser-based attacks, *i.e.*, a website with JavaScript as well as a scriptless attack exploiting the serial loading of fonts integrated in websites.
- In the best case, the cross-VM attack, monitoring 100 domains takes $763.122\,\text{ms}$ with an increase by $2.931\,\text{ms}$ for any additional domain. In the worst case, the JavaScript-based attack, it takes $14.787\,\text{s}$ and each additional domain adds another $147.110\,\text{ms}$.
- In an end-to-end attack from inside a VM, we reliably detect the host's website visits from a list of 103 websites (in an open-world scenario) with an F_1 score of $92.48\,\%$ in less than one second.

Outline. In Section II, we provide background on DNS and DNS caching, timing side channels, and DNS side channels. In Section III, we present the high-level overview of our attack. In Section IV, we present techniques to read the DNS cache state. In Section V, we present techniques to evict the DNS cache reliably. In Section VI, we evaluate the end-to-end DMT cross-VM attack. In Section VII, we demonstrate website access

tracking from JavaScript in the browser. In Section VIII, we contextualize our work and discuss its impact. We conclude in Section IX.

II. Background

In this section, we provide background on the role of DNS on the Internet, DNS caching, timing side channels, and existing network and DNS side channels.

A. The Role of DNS on the Internet

The Domain Name System (DNS) has been developed to translate human-readable domain names into IP addresses [64], [65], e.g., before accessing a website, and is a distributed infrastructure: The root name servers resolve the top level domains (TLDs), such as `.com`, `.org`, and `.net`, while TLDs operate their own name servers for the resolution of second-level domains (SLDs). Therefore, they point to the authoritative servers for specific domains that know the records and are typically managed by domain registrars.

Practically, a client sends a DNS query to its configured recursive DNS server. If unaware of the appropriate IP address, the recursive server resolves the domain name by querying the root name server, the one responsible for the top-level domain, and eventually the authoritative server. Thereby, individual steps might be omitted if the response is already known. The recursive resolver is typically operated locally (e.g., by the ISP), but there is a recent trend towards public resolvers [22].

Over time, more information has been incorporated into DNS, turning DNS into an even more important component of the Internet. In DNS records, among others, Sender Policy Framework (SPF) [45] specifies a domain's legitimate email servers, DomainKeys Identified Mail (DKIM) [17] stores cryptographic keys to verify email authenticity, and Domain-based Message Authentication, Reporting and Conformance (DMARC) [48] announces policies on email handling. DNS also also plays a role in malware protection [79], parental control systems [53], censorship [36], and DDoS protection [43]. Due to its importance, DDoS attacks against the DNS have serious consequences for the Internet as a whole [88].

Traditional DNS is neither encrypted nor integrity protected, facilitating cache poising [58], interception [77], or user fingerprinting [5], and has lead to numerous improvements in domain resolution. Domain Name System Security Extensions (DNSSEC) [37] provides cryptographic authentication of DNS resource records, but does not encrypt the queries and replies. This changed with DNS over HTTPS (DoH) [38] and DNS over TLS (DoT) [39] tunneling DNS requests over the well-established protocols HTTPS and TLS, respectively, to provide confidentiality.

B. DNS Caching

In the worst case, a DNS query is forwarded from the client to the recursive server, which in turn iteratively queries the root server, the TLD's server, as well as the authoritative servers. DNS queries travel back and forth over the network, in most cases the Internet, altogether causing a non-negligible delay of multiple tens or even hundreds of milliseconds [42]. Beyond that, the repeated resolution of the same domain name is common, e.g., when revisiting a website, motivating caching to gain performance benefits.

By the definition of a time-to-live (TTL), caching has been an integral part of DNS right from its beginning [64], [65]. The TTL defines a resource record's lifetime in seconds and therefore limits the time that a response can be kept in the cache. Its value is defined by the record's originator. At a later point in time, also negative caching has been introduced by DNS [3], [94], *i.e.*, also negative results are stored in the cache. Recommendations for the TTL vary between five minutes and 24 hours, depending on the distinct scenario as a tradeoff between performance and flexibility needs to be found [66].

In practice, a multi-level caching architecture for DNS emerged, frequently reducing DNS requests to a round-trip time of a few milliseconds [14]. Recursive server implementations cache answers from the root name servers, the TLD's server, and authoritative servers by default to avoid further queries. These servers typically serve multiple clients, *i.e.*, a client might even benefit from another client that tried to reach the domain before. This applies even more strongly to public resolvers, serving a larger customer base. Operating systems operate DNS caches that are shared by all their applications. In case of a cached resource, communication over the network for DNS resolution is prevented at all. Even browsers operate their own caches, limiting coordination with the operating system.

Caches also have introduced challenges for both functionality and security. First, (legitimate) resource record modifications by the authoritative servers take longer to propagate to the clients. Second, cache poising attacks [58], [59], [52] lead to the storage of illegitimately modified resources. Upon request, these poisoned records are forwarded to the clients tricking them in a connection with a potentially malicious destination. Third, the differences in timing between a cached and an uncached records forms a timing side channels [24].

C. Timing Side Channels

Side channels are a powerful means to extract information without exploiting any bugs. One of the earliest and most commonly used is the timing side channel [47]. Timing can originate in software [47] or hardware, e.g., due to caching [71], [95]. For cache-timing side channels, there are generic techniques like Flush+Reload [95], Evict+Reload [32], Prime+Probe [71], [55], [60], and Flush+Flush [31]. These techniques follow a pattern of resetting the state of the cache and measuring the state of the cache, typically in a destructive way that necessitates resetting the state again.

Some side channels can be mounted in scenarios with a *remote* attacker. For instance, JavaScript-based attacks are often considered *remote* [24], [70], [30], [92]. Special APIs can be attacked remotely [9], [12], [16], [81], [18]. Some remote attacks even only send packages to a victim and observe timing differences through this [49], [27].

D. Network and DNS Side Channels

Network side channels are inherent to the operation of networking components, exploiting normal, standards-compliant behavior of the network stack. A well-established attack vector is traffic analysis, usually performed by a passive attacker able to intercept network traffic [4], [67], [87], [82]. Traffic analysis extracts privacy-sensitive information from traffic characteristics like packet sizes, directions, and timings. Among the most widely explored are fingerprinting attacks, targeting applications [90], [84], videos [23], [83], and websites [78], [10]. A large number of *website fingerprinting* studies focus on privacy-enhancing tools like Tor or VPNs. Early work by Hintz [35] showed that the size patterns of individual asset downloads from the (now obsolete) SafeWeb proxy already reveal the visited website. Subsequent work expanded this to analyzing individual packet sizes [11], packet direction ratios, and total packet counts [74], [73]. Klein and Pinkas [46] track users by giving them a unique combination of DNS records, which are stored in the DNS cache, identifying them across browsers. Bushart and Rossow [13] demonstrated a passive fingerprinting attack relying only on encrypted DNS traffic via DoT and DoH. Many works have improved feature extraction and classification over time [34], [93], [33], [78], [10], [85], [86], [82], [8], [20], [41]. Additionally, network side channels have been demonstrated remotely, both over the Tor network [68], [61] and on standard network infrastructure [44], [28], [2], [27], with more active techniques requiring interaction between the victim and a remote server.

Most closely related to our work are DNS cache timing attacks [24], [29], [75], [63]. DNS cache timing attacks infer a resource's presence in a cache. Felten and Schneider [24] were the first to demonstrate that DNS cache timing can be used to infer whether a domain name is cached. They used a timing side channel to determine whether a domain name was cached by the DNS resolver. Grangeia [29] extended their work with three methods (1) preventing recursive resolution, (2) setting of a low TTL, and (3) measuring the time it takes to resolve a domain name, like Felten and Schneider [24]. Klein and Pinkas [46] exploited DNS cache timing differences to track users. They point out that reading from the cache is a destructive operation, *i.e.*, the cache state is modified by reading from it and has to be reset before it can be exploited again. Recent work by Moav et al. [63] is not public yet but under embargo. Based on the abstract, their work uses a flush operation to reset the DNS cache state, which is not available in all environments. Furthermore, we suspect their attack targets the router as the abstract describes an attack on IoT devices within the same network. Hence, none of these related works can reliably evict the DNS cache without flush operations that are commonly unavailable or privileged, *i.e.*, they cannot monitor when precisely a user accessed a website. Still, shared and public DNS resolvers can be targeted with these methods to investigate domain access distribution for user access statistics [75], [40], [51], [69], [50], [76], user tracking [54], and malicious domain detection [57], [25], [26], [56]. Beyond that, public resolver caches were exploited for covert communication channels [80]. In practice, however, success depends on the resolver's specific cache structure [62].

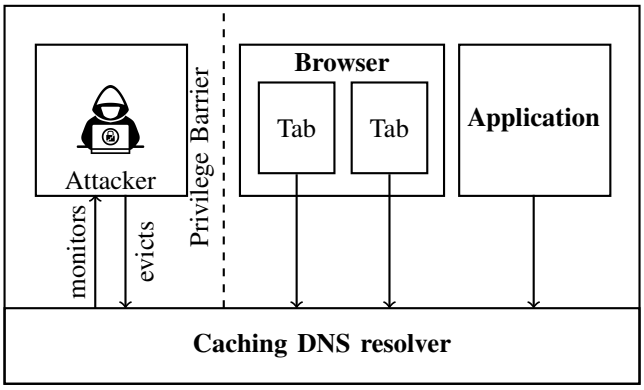


Fig. 1. General overview of the working principle of DMT. Applications (e.g., the browser), interact with the DNS resolver. Resolved domain names are added to the shared system-wide DNS cache. The unprivileged attacker (e.g., native code, JavaScript, or scriptless, possibly in a VM) monitors the state of the DNS cache, which leaks information about user behavior and running applications on the entire system.

A single shared cache for all clients has the highest cache hit rates. Isolated caches per client have prohibitive overheads. Lastly, some recursive resolvers do not cache at all, potentially for security reasons.

III. High-Level Overview and Threat Model

In this section, we describe the DMT attack. We first define the threat model and give a high-level overview on the attack. We then describe the three scenarios in which DMT can be mounted: **a) local attacks, unprivileged, sandboxed, and cross-VM**, **b) remote attacks with JavaScript** and **c) scriptless remote attacks without JavaScript**.

A. Threat Model and Attack Scenarios

The attacker wants to spy on a victim by monitoring the contents of the local DNS cache. In our model, the attacker is unprivileged, yet able to run either native code (possibly in a VM or sandbox), JavaScript in the browser, or have a scriptless HTML page rendered by the browser. By monitoring the DNS cache state, the attacker obtains a list of recently accessed domain names, along with an approximate time of access. This information is a breach of privacy already, and can be used in various ways, including targeted advertisements, deanonymization of a user, and targeted scam campaigns, e.g., phishing or extortion. For instance, in sextortion scams an attacker claims to have compromising material of the victim, e.g., by claiming to have hacked the victim's webcam. The attacker can significantly increase the perceived credibility of the scam, by listing timestamps where the victim accessed potentially embarrassing or controversial websites. Similarly, online shops could use browsing information to adjust prices of products based on recent browsing activity, e.g., by increasing the price of a product that the victim has recently browsed. Finally, DNS cache information can also leak which

Endpoint Detection and Response (EDR) solution is installed on a victim's system (before the HTML page has even finished loading), and thus serve as a building block in a tailored exploit chain to bypass the specific EDR solution.

We assume privacy-concerned victims may use a VPN and non-default DNS server settings to avoid, e.g., surveillance systems depending on country and region. Still, DMT works in such scenarios, as the DNS cache is still a shared resource on the victim's system, *i.e.*, it can leak information about the victim's web activity despite the entire traffic, including the DNS requests, being protected by a VPN.

B. Working Principle

Figure 1 shows an overview of our attack. To obtain fine-grained access data, the attacker continuously monitors the DNS cache by measuring the lookup latency of target domain names. If the domain name is not cached, the DNS resolver needs to perform a network round-trip to the DNS server. Depending on the DNS server, this round-trip introduces significant delay to the resolution process. During the measurement, the target domain name is added to the cache. Thus, to improve the temporal resolution of the attack, the attacker needs to evict the DNS cache after each measurement. To do this, we introduce multiple DNS cache eviction primitives and compare them to the flushing functionality available in native, non-sandboxed code. These primitives allow an attacker to evict the DNS cache from application sandboxes, virtual machines, JavaScript and even scriptless.

C. Attack Scenarios

An attacker can monitor the DNS cache state in multiple scenarios, where the attacker in each scenario has primitives to measure the cache state (as we discuss in Section IV) and to evict the DNS cache (as we discuss in Section V). We start from higher privileged scenarios, such as native code execution, and then move to lower privileged scenarios, such as JavaScript in the browser and scriptless HTML.

Local Native, Sandboxed, and VM-based Attacks. In a local native scenario, the attacker can run unprivileged code on the same machine as the victim albeit under a different user account. Thus, the attacker cannot access, e.g., the target user's browser history files directly, and instead uses the DNS cache as a side channel for the victim's web activity. This scenario is realistic in a multi-user environment, such as a shared machine or a thin-client architecture, in which a malicious unprivileged employee could spy on browsing activities by their coworkers. Both with lightweight application sandboxes, such as Firejail and Docker, but also with virtual machines in a NAT networking setup can use the system-wide DNS resolver, leaving our attacks unaffected. We find this to even be the default on a Debian 12 installation resulting in the same capabilities as regular non-privileged users for DNS resolution, *i.e.*, the local DNS cache of the system is used. However, sandboxed and virtualized attackers by default cannot access DNS management interfaces, such as `resolvectl`. Hence, they cannot flush the DNS cache directly but have to resort to eviction primitives based on DNS resolution (cf. Section V).

Remote JavaScript. Since our measurement and eviction primitives only require DNS resolution (cf. Section V), we can also mount DMT from websites. In this scenario, the victim only needs to open the attacker-controlled website. This can be achieved by sending a link to the victim, or by embedding the attacker's website in a third-party website, e.g., via ad networks or vulnerabilities in the website, such as persistent Cross-Site Scripting (XSS) exploits. In this scenario, the attacker can use JavaScript, which is executed on the local machine within the browser's sandbox, and e.g., using `fetch`, can trigger DNS resolutions. The Cross-Origin Resource Sharing (CORS) mechanism, which is designed to prevent XSS attacks, ensures that the `fetch` request fails without transmitting any content from the requested domain but the DNS resolution is still performed, ironically resulting in a higher reliability of our attack than without CORS.

Remote Scriptless. Even though JavaScript is an important part of modern websites, some security-conscious users may disable JavaScript in their browsers. This thwarts JavaScript-based attacks, as the attacker cannot use JavaScript to trigger DNS resolutions. However, DMT can also be mounted from a plain HTML page, without any JavaScript. By including resources from other domains, the browser will automatically perform a DNS lookup to download the resource. Since without the ability to directly execute code, the attacker cannot use any timing APIs, such as `performance.now()`, to measure the request latency, we develop a fully scriptless attack that relies on the browser's serial loading of resources. We surround the target request with **two additional requests** to an attacker-controlled server. The timing difference between these two requests can be measured on the attacker server to infer the latency of the target request. When measuring more than one domain, only one extra request is needed per additional domain. We demonstrate that even in this very restricted threat model, resolution-based eviction primitives are still practical. This scenario shows that mitigating DMT is difficult, as it only relies on the minimal features required to access domains rather than JavaScript or native code execution.

IV. Measuring the Local DNS Cache State

In this section, we analyze the timing differences between cached and uncached domain names in different scenarios including native code, JavaScript code in the browser, and plain HTML in the browser. In our measurements, we focus on `systemd-resolved`, which is the default resolver for prominent Linux distributions such as Fedora and Ubuntu [15], and the recommended option for a caching DNS resolver on Arch Linux [6]. Distributions not using `systemd-resolved` commonly do not ship with a dedicated DNS resolver, and instead use `libresolv` via `glibc` to resolve DNS queries. Because `libresolv` does not implement a DNS cache, these distributions, in their default configuration, are not vulnerable to DMT.

Measurement Setup. All measurements are performed on a cloud VPS using Google DNS (IP 8.8.8.8, 8.8.4.4) as the configured DNS server. The ping latency to the DNS server is approximately 5 ms on average, which is significantly lower than on typical consumer Internet connections and thus constitutes a worst-case scenario for our measurements. The VPS runs a minimal Linux installation with `systemd-resolved` as the DNS resolver. For browser-based measurements, we use Chromium 136.0.7103.25 in headless mode, instrumented by Playwright 1.52.0 using Python. For measurements that require an external attacker server, we use another VPS, also running a minimal Linux installation. Each measurement runs over the span of multiple days, with long (5 s or more) pauses between requests to avoid flooding any of the servers with requests.

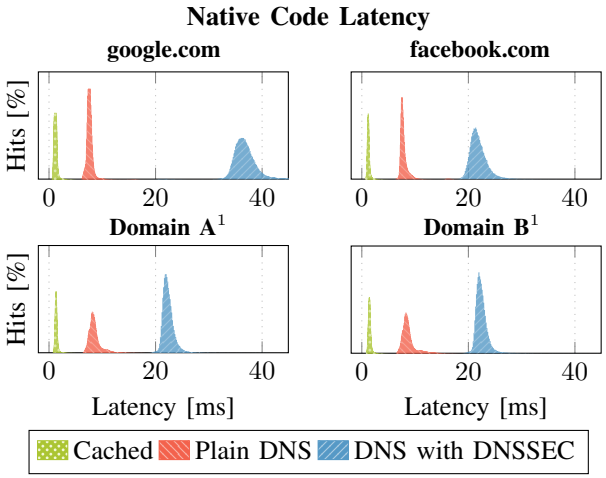
Measurement Strategies. Measuring the domain resolution latency requires different strategies depending on the execution context. If the attacker has access to the `resolvectl` command, they can use `resolvectl query`, which reports the resolution time and the source of the resolution, *i.e.*, `network` or `cache`, which eliminates the need for timing measurements completely. In native code contexts without access to `resolvectl`, the attacker can use native library functions, such as `getaddrinfo` or `gethostbyname`, to resolve a domain name, and measure the latency until the function returns. In contrast, JavaScript does not expose dedicated DNS resolution functions. Instead, the attacker can use the `fetch` API to trigger a DNS resolution. However, as `fetch` performs an entire HTTP request, this likely results in added noise due to unrelated effects like the web server's response time. In a scriptless environment, the attacker needs to rely on requests to attacker-controlled servers before and after a target request, to measure the runtime of the target request, which adds even more noise due to two additional network requests. In the following, we describe and evaluate measurement strategies for *native code*, *JavaScript* and *scriptless browser contexts*. Additionally, we also evaluate the JavaScript measurement strategy on a consumer Internet connection, showing that we achieve an even higher accuracy when attacking end users.

A. Native Code

Native code yields the highest accuracy when measuring the DNS cache state. On Linux, the attacker cannot directly list the DNS cache as `resolvectl show-cache` is a privileged operation.[1] Yet, attackers might be capable to use the unprivileged `resolvectl` command, and use `resolvectl query`, directly reporting whether DNS data was returned from the cache, for resolution.

If `resolvectl` is not available (e.g., in a jailed environment or a virtual machine), the attacker can alternatively measure the latency to distinguish cached from uncached domain names. For example, the command-line tool `dig` resolves domain names, and reports the resolution time in milliseconds. By default, `dig` uses the system-wide DNS resolver, and is thus suitable for our attack. In jailed environments without the capability to execute other programs, the attacker can use `libc`'s `getaddrinfo` function, or similar, programming language specific APIs, such as Python's `socket.gethostbyname` to resolve domain names.

Evaluation. We measure the execution time of `socket.gethostbyname` in Python. For 4 different domains, Figure 2 shows the latency distributions for cached and uncached domain names, both with and without DNSSEC, and reveals clear differences between cached and uncached domains. While cached domain names take on average 1.6 ms to resolve, uncached take 8 ms. With DNSSEC, the latencies become even larger. While cached domains still resolve in 1.6 ms, uncached take 20 ms probably due to validation each response.



[1] Domain names blinded for peer review

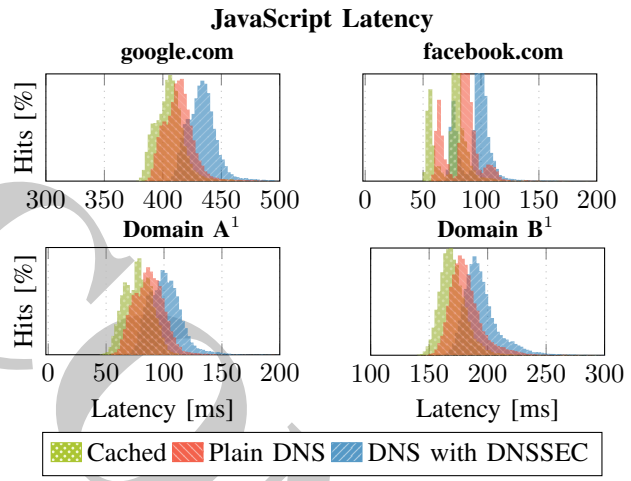
Fig. 2. Histogram of domain resolution latencies for cached and uncached domain names, with and without DNSSEC enabled, recorded in native code. Except for extreme outliers, there is a clear separation between cached and uncached domain names. With DNSSEC enabled, the latency difference is even larger. Interestingly, the shape of resolution timings for Domain A and B is very similar, as they both use CloudFlare as authoritative DNS server.

B. JavaScript

In the browser, the is no direct access to the DNS resolver. Instead, JavaScript's `fetch` API implicitly triggers DNS resolution. Measuring the request latency, an uncached domain causes again a longer runtime. However, as `fetch` performs a full HTTP request, the latency of connecting with web server adds noise to the measurement.

The Cross-Origin Resource Sharing (CORS) mechanism requires the browser to send a preflight request in advance of the actual request. This is an `OPTIONS` request, which does not download any data. As most websites forbid cross-origin requests, the `fetch` request will fail directly after the preflight response, minimizing noise from additional data

[1]In contrast, on Windows 11, the PowerShell commands `Clear-DnsServerCache` and `Get-DnsServerCache` are available to unprivileged users. Thus, on Windows, an attacker capable of running PowerShell commands can directly obtain the entire DNS cache state, without performing any timing measurements. As the DNS cache is shared between users, unprivileged attackers can monitor local DNS cache activity, with a fine-grained resolution.



[1] Domain names blinded for peer review

Fig. 3. Histogram of `fetch` latencies for cached, and uncached domain names, with and without DNSSEC enabled, recorded in JavaScript. Compared to Figure 2, the measurements are significantly noisier, but still allow us to distinguish between cached and uncached domain names. The significant difference between latencies of different domains is caused by the response latencies of the different web servers.

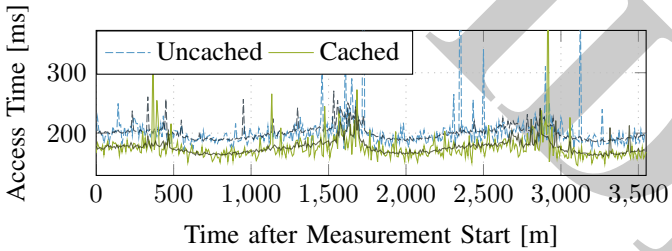
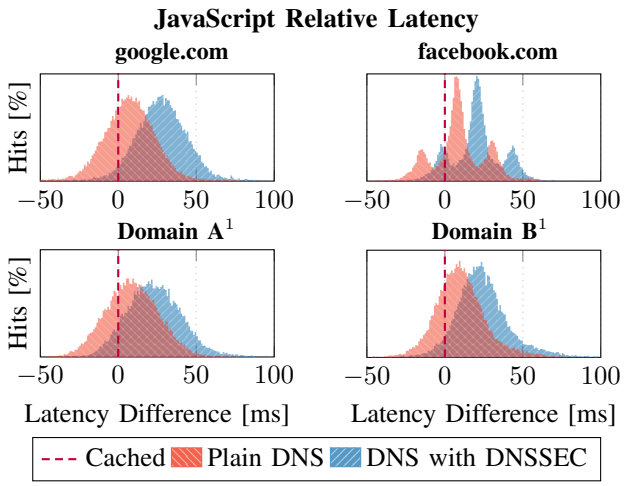


Fig. 4. Trace of `fetch` latencies for cached and uncached accesses of **Domain A**[1] over a long measurement timeframe (approximately 60 h). Bright lines show one raw sample every 500 seconds, while dark lines show the average of 50 samples (one every 10 seconds) during the same timeframe. The offset between cached and uncached latencies stays relatively constant over time, while the absolute latencies vary, adding noise to long-term measurements, while not affecting actual cache state measurements. [1] Domain name blinded for peer review

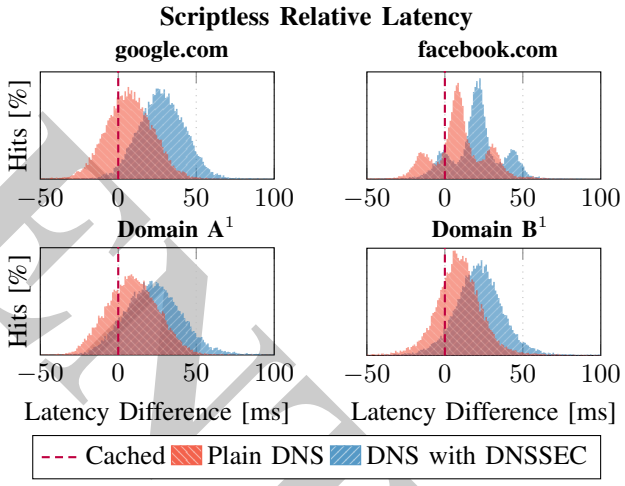
transfer. While the `OPTIONS` request still adds latency, and consequently noise, to the measurement (compared to only resolving the domain), the websites' content do not have an impact as no data is downloaded by the browser.

Evaluation. In Figure 3, we show histograms of domain resolution latencies for the same domains as in the native code evaluation. The latency difference between cached and uncached DNS resolutions is, despite JavaScript's limitations, in the range of 5 ms to 30 ms. Consequently, the temporal resolution of JavaScript's `performance.now()` timestamp is sufficient for our measurements. The noise is caused by network jitter, see Figure 4, for the latency of cached and uncached accesses over a timeframe of about 60 h. With a few exceptions, the latency difference between cached and uncached accesses remains relatively stable over time. Both



[1] Domain names blinded for peer review

Fig. 5. Histogram of the latency differences between an uncached and the following cached domain lookup, with and without DNSSEC, recorded in JavaScript. This representation reduces the noise caused by slow changes in the server response time, as seen in Figure 4. While the latency of uncached domain names overlaps with the cached latency, the difference is still significant even without DNSSEC. DNSSEC separates the lookup latencies even further.



[1] Domain names blinded for peer review

Fig. 6. Histogram of the latency differences with a scriptless measurement. The latency differences are similar to the JavaScript measurements in Figure 5, indicating that our method of using alternative fonts to serialize website requests is effective and does not introduce significant extra noise. Thus, although scriptless attacks require more engineering effort, they are just as reliable as JavaScript-based attacks.

Figure 3 and Figure 5 show a peculiar histogram shape for our measurements on facebook. Because this shape does not appear in Figure 2, we assume that we measure the latency of different servers which we are routed to by a load balancer.

C. Scriptless

In absence of JavaScript (e.g., a deactivation for security reasons), DMT can be mounted from a plain HTML page. In

```
body {
  font-family: "EvictionFont", "DMTFont";
  background-image:
       url("https://201.attacker.com/eviction"),
       url("https://202.attacker.com/eviction"),
       ...
       url("https://1000.attacker.com/eviction");
}

@font-face {
  font-family: "DMTFont";
  src: url("https://attacker.com/measurement-start"),
       url("https://target-domain.com/random-value"),
       url("https://attacker.com/measurement-end");
}

@font-face {
  font-family: "EvictionFont";
  src: url("https://1.attacker.com/eviction"),
       url("https://2.attacker.com/eviction"),
       ...
       url("https://200.attacker.com/eviction");
}
```

Listing 1. DMT using sequential loading of fallback fonts with CSS. To measure the latency of multiple domains, we only require one additional measurement request per domain. Browser cache eviction is achieved by filling the cache with many attacker-controlled entries.

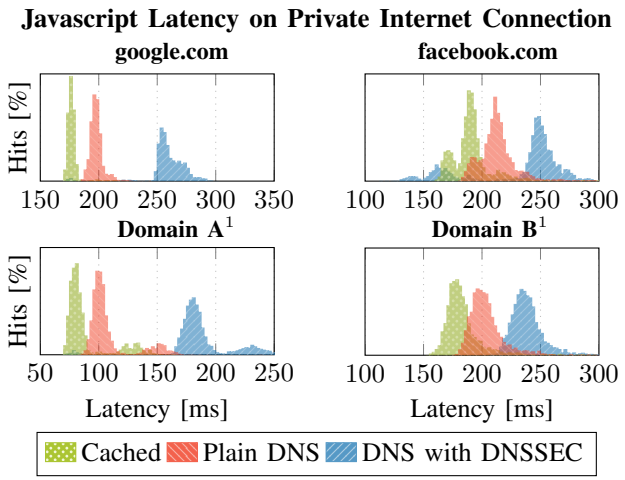
our scriptless attack, the attacker exploits CSS features [91], in particular the browser's font fallback. As fallback fonts are loaded sequentially, and this behavior is measured to infer the latency of the DNS requests.

As shown in Listing 1, we define a custom font, `DMTFont`, including three URLs to measure the target domain's DNS latency. The first URL points to an attacker-controlled server, and triggers the start of the measurement. Since the server does not return a valid font, the browser proceeds with the second URL, pointing to the target domain. This URL includes a random path to ensure that no previous CORS result is cached, and no data is returned. Yet, it triggers DNS resolution before failing. The third URL points again to the attacker-controlled server, signaling the end of the measurement. Finally, we compute the DNS latency as the time between the first and third request. To measure more domains in a row, we insert one request to the attacker-controlled server after each domain. An attacker can also define additional font-families assigned to different CSS selectors to parallelize measurements. We must evict the target domains from the browser's DNS cache first. Otherwise, the browser would not forward the requests to the system-wide DNS resolver. See Section V-E for a detailed explanation of our cache eviction strategy.

Evaluation. For a realistic measurement, we measure latencies accross the Internet to an external server. In Figure 6, we show the latency distributions for the same domains as before. Again, we achieve results allowing a clear differentiation between cached and uncached domain names. As before, the fluctuations in absolute latencies might be caused by jitter or other network-related affects.

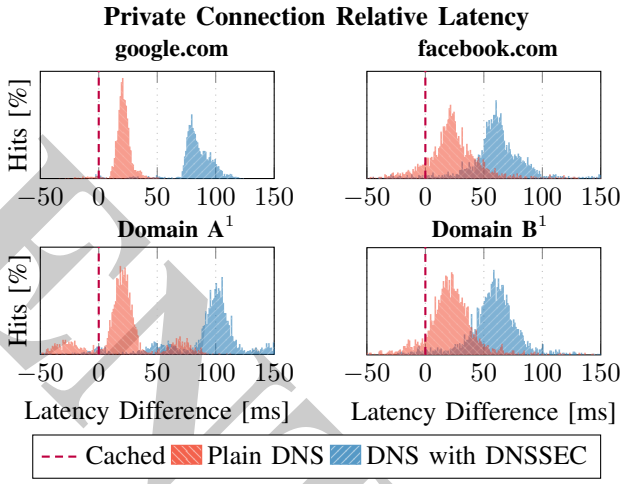
D. Consumer Internet Connection

All experiments were performed on a VPS in a commercial data center with a fast Internet connection. According to `speedtest-cli` [89], download speed was 1 960 Mbit/s, upload 1 518 Mbit/s, and ping latency to Google's DNS server (8.8.8.8) 5 ms. Additionally, we performed the worst-performing experiment, the JavaScript-based attack, on a private Internet connection (download: 48 Mbit/s, upload: 8.4 Mbit/s, DNS server latency: 13.5 ms). Figure 7 shows the results for this scenario: Due to the higher latencies to the DNS server, the timing difference in latencies for cached and uncached domains is more nuanced than before, see in Figure 3. As the higher latencies represent users scenarios more closely, the measurements using the fast Internet connections are the worst-case scenario from a user's perspective.



[1] Domain names blinded for peer review

Fig. 7. Histogram of `fetch` latencies on a private Internet connection. In contrast to Figure 3, the distributions are significantly more separated. The latencies were recorded on a 50 Mbit/s home Internet connection, instead of a commercial data center connection. This demonstrates that our other measurements are the worst-case scenario for our attack, as most users will not have such a fast connection.



[1] Domain names blinded for peer review

Fig. 8. Histogram of the latency differences in JavaScript on a private connection. Compared to Figure 5, which was measured on a cloud machine in a data center, this histogram shows that slower connections result in a larger latency difference, and thus a more reliable measurement.

TABLE I
SUMMARY OF THE LATENCY DIFFERENCES AND ACCURACY OF OUR MEASUREMENT PRIMITIVES FOR EACH CONTEXT, FOR **FACEBOOK.COM**.

	DNSSEC	Average Offset[1]	Standard Deviation[1]	False Negatives
Native	✓	20.663 ms	1.677 ms	0.000 %
	✗	6.701 ms	1.517 ms	0.006 %
JavaScript	✓	20.356 ms	16.871 ms	13.641 %
	✗	8.942 ms	17.487 ms	24.192 %
Private JS[2]	✓	82.363 ms	18.041 ms	1.110 %
	✗	22.902 ms	31.044 ms	13.209 %
Scriptless	✓	20.744 ms	17.102 ms	12.494 %
	✗	9.395 ms	18.039 ms	23.463 %

[1] To account for a small number of large outliers in measurements (*i.e.*, multiple seconds, likely due to connectivity issues), we eliminated the top and bottom 0.1 percentile of our measurements for our average and standard deviation calculations. They are still included in the computations of the false negatives. We still keep over 20 thousand measurements for each scenario on commercial servers.
[2] Measured on a private connection, with approximately 2 500 samples.

E. Limitations

All of our measurements relied on public DNS resolvers but many networks, such as corporate networks, rely on self-operated caching DNS resolvers. These resolvers can provide sub-millisecond latencies for cached domains, compared to the 5 ms latencies of to Google's DNS server. With shorter latencies, differentions between cached and uncached domains becomes harder. Yet, there is a trend towards public resolvers, which often have features like DNS-over-TLS, increased privacy, or reduced filtering.

F. Summary

Table I summarizes the results of our measurement primitives for each of the three execution contexts. Using system APIs, we achieve the most precise measurements in native code. With a single measurement, cache hits and misses are differentiated with almost no false-positives and false-negative. From the browser, we can only resolve domain names by means of a full web request, adding noise to our measurements. For most websites, CORS preflight requests reduce this noise again by terminating the request before any data is downloaded. Still, this workaround leads a false-negative rate of 24.192 % in JavaScript, and 23.929 % in plain HTML. While this error rate still allows us to distinguish cached and uncached resolutions with a high level of confidence in few measurements, it prevents reliable single-measurement attacks. DNSSEC increases the DNS latency and therefore reduces the false-negative rate to 13.641 % and 12.494 %, respectively. The scriptless attack performs slightly better than the JavaScript measurement, indicating that `fetch` introduces more noise than a plain HTTP request generated by the browser. We assume that the browser is optimized to load HTTP resources, such as fonts, as fast as possible, while `fetch` is aimed towards greater flexibility and usability, leading the observed results.

V. EVICTING THE LOCAL DNS CACHE STATE

DNS cache eviction allows us to increase the temporal resolution of our attack beyond the TTL of the targeted DNS entries. In this section, we present four eviction strategies for the system-wide DNS cache, available from different execution contexts, and in different system configurations. Some browsers implement their own, application-private DNS cache, which in principle is also vulnerable to timing attacks. However, that cache cannot be used for cross-application monitoring and is not shared between normal and private browsing modes, making it a weaker attack target. Thus, we also demonstrate how this cache can be bypassed in Google Chromium.

Setup. For some of our eviction strategies, we require fine-grained control over DNS responses. Thus, we implemented a custom DNS server that can return arbitrary DNS responses. However, with default configurations, victims do not access our DNS server directly. We purchased a public domain name, and set up a subdomain `NS` records that point to our DNS server's IP address. This delegates the responsibility of resolving the subdomains to our DNS server. We use random subdomain prefixes to bypass server-side and client-side DNS caches for requests to our DNS server. The DNS server selected by the victim client (e.g. Google DNS or provider default) recursively resolves the subdomain and forwards the result to the victim, giving us fine-control over the DNS responses sent to the victim.

A. Direct Cache Flushing

The **first** primitive is a direct cache flush. On many common caching DNS resolvers, such as `systemd-resolved` on Linux, or Windows, clearing the DNS cache is not a privileged operation. On `systemd-resolved`, the DNS cache can be flushed by executing the `resolvectl flush-caches` command. On Windows, the `Clear-DnsServerCache` PowerShell command can be used to flush the DNS cache. This primitive is the fastest (10.987 ms on average) and most reliable way to flush the DNS cache. However, these commands are only available when the attacker has local code execution on the system. Thus, from a sandboxed environment, such as FireJail or a virtual machine, this primitive is not available.

B. Individual DNS Requests

The **second** primitive is a simple cache eviction by filling the cache with random entries. Even though RFC 1536 recommends that DNS cache sizes should be unbounded [19], common real-world DNS caches (e.g., `systemd-resolved`, Android's `resolv`, Windows 11) have a fixed maximum cache size, restricting memory usage. Additionally, most DNS caches evict the entries with the shortest remaining TTL first. Thus, the attacker can fill the cache with random eviction entries with long TTLs, which will evict all other entries. This primitive relies on intended behavior of the DNS resolver, and thus is available in all execution contexts. However, it is rather slow. For the first eviction, when the cache does not contain

any eviction entries, every entry used for eviction needs to be resolved. With a resolution time of approximately 30 ms per domain, and sequential resolution, the eviction takes over 2 min for `systemd-resolved`, where the default cache size is 4 096 entries. Later evictions are faster, as they only need to resolve entries that have been evicted by legitimate DNS requests. However, the if attacker can parallelize eviction, the eviction time with this primitive is greatly reduced. In our experiments, parallel resolution reduced full eviction time to 5.109 s using 100 threads. As this primitive requires entries with a large TTL to evict all other entries, subsequent legitimate DNS requests will preferably evict other legitimate entries. Thus, this primitive alone can only be used to track DNS resolution of a single target domain.

Cache Hole-Punching. The attacker can eliminate the single-domain limitation by "punching a hole" in the DNS cache for legitimate entries, after filling it with eviction entries. For this, the attacker queries a domain with large DNS responses, containing multiple entries with short TTLs, which can be evicted by legitimate queries. On `systemd-resolved`, all entries in a DNS response are guaranteed to be cached, even if the cache is full with entries that have a longer TTL. The resolver will evict old entries to make space for the new entries, *before* adding them to the cache. This creates space for new, legitimate entries, so multiple domains can be resolved without evicting each other. This primitive is available in all contexts on systems with `systemd-resolved`, and eliminates the single-domain limitation of the previous primitive.

C. Large DNS Responses

The **third** eviction strategy forces the DNS resolver to drop multiple legitimate entries at once by sending a DNS request that returns a large number of entries, again with a small TTL. In contrast to cache-hole-punching, this strategy removes legitimate entries from the cache, instead of eviction entries. This eliminates the need for filling the entire cache with indiviual, single DNS requests. However, even DNS-over-TCP [21], the maximum size of a DNS response is limited to 65 535 B due to the 16-bit length field in the DNS header. Even with a very short domain name, such as `a.b`, and compression enabled, we did not succeed in fitting 4096 entries in a single response. Instead, our maximum number of entries per response was 4 091 , which works to domains with of up to 16 characters, such as `longerexample.com`. Thus, it is not possible to evict the entire cache in a single request, using intended behavior.

Eviction Priming. To work around the limited number of entries in a DNS response, the attacker can combine individual requests and a large response to evict the entire cache. The attacker first primes the cache by resolving a small number (< 10) of requests that return single entries with long TTLs[2]. Afterwards, the attacker requests a DNS response containing a large number of entries with a short TTL. Because the cache is primed with long TTLs, `systemd-resolved` will evict all other entries first. Afterwards, the TTL of the large response expires, leaving only a small number of primed entries in the cache. This primitive also works from all execution contexts and requires significantly fewer DNS requests. However, some firewalls and DNS servers do not allow for such large DNS responses. For example, on the CloudFlare public DNS server, we were only able to send up to 87 entries in a single response. We found that the default-configured DNS server for our cloud evaluation machines allows for responses of arbitrary sizes, enabling this primitive. To speed up the eviction process, we send the 10 priming requests in parallel, and the large response request sequentially. In total, this eviction primitive takes approximately 1.387 s on average. An evaluation of a selection of public DNS servers, their limits, and their behavior when responses are too large, can be found in Table II.

[2]Experiments showed that requesting the domains in a single request caused them to be evicted prematurely, leading to unreliable eviction of other entries.

TABLE II
MAXIMUM NUMBER OF DNS RESOURCE RECORDS RETURNED BY PUBLIC DNS SERVERS.

DNS Server	Max Answer Records	Error on Exceeding Limit
Google	248	Delegation Failure
Cloudflare	87	Delegation Failure
OpenDNS	83	Delegation Failure
Quad9	72	Delegation Failure
Yandex	4089[1]	Timeout
Verisign	250	Truncated/EOF

[1] Because the domain used to measure the limits contains more than 16 characters, we cannot reach 4 091 records in a single query response.

D. Error-based Eviction

Our **fourth** primitive exploits the error handling behavior of `systemd-resolved`. When the DNS response contains an error, which can be caused e.g., by returning more entries than allowed by the intermediate DNS server, `systemd-resolved` will retry resolving the domain name. After three retries, `systemd-resolved` will switch to the configured fallback DNS server, causing `systemd-resolved` to flush the entire cache. The time required between initiating the request and the switch to the fallback DNS server depends on the behavior of the upstream DNS server. Some DNS servers, like Google DNS, will return an error immediately. For these servers, the DNS cache is evicted after approximately 79.8 ms. Other servers do not respond to the query at all, which causes `systemd-resolved` to wait for a timeout of 5 s before switching to the backup DNS server, which delays the eviction. After switching to the backup DNS server, `systemd-resolved` will try to resolve the domain again. In our experiments, if the responses remain invalid, `systemd-resolved` will repeat this retry-and-switch process for multiple minutes, clearing the DNS cache each time. The performance of this eviction technique depends on the behavior of the upstream DNS server.

Eviction Loop Recovery. To avoid the constant eviction caused by the retry-loop, the attacker can employ a custom DNS server, which replies with a valid response after 4 resolu-

TABLE III
AVAILABLE EVICTION STRATEGIES FOR SYSTEMD-RESOLVED.

Primitive	Availability			Eviction Time	Config-Dependent
	RCE	JS	HTML		
Direct Flushing	✓	✗	✗	10.987 ms	no
Many Requests	✓	✓	✓	5.109 s	no
Large Response	✓	✓	✓	1.387 s	yes[1]
Error-Based	✓	✓	✓	79.1 ms to 5 s	yes[2,3]

[1] DNS Server allowing arbitrary response size.
[2] Fallback DNS server configured.
[3] Eviction time depends on upstream DNS behavior.

tion attempts. This allows `systemd-resolved` to correctly resolve the domain name after switching to the fallback DNS server. Thus, no more retries are necessary, and the eviction loop stops. While this primitive relies on very specific behavior of `systemd-resolved`, it is available in all execution contexts. Additionally, it evicts the entire DNS cache in under one second, and requires only a single DNS request instead of multiple coordinated requests. This primitive is dependent on the system configuration, as it requires a fallback DNS server to be configured. However, most public DNS providers, such as Cloudflare, Google and Quad9, provide a fallback DNS server, which indicates that many systems will be vulnerable to this primitive.

E. Browser DNS Cache Bypass

Modern browsers implement their own application-private DNS cache to speed up DNS resolution. However, this cache is not shared across processes, preventing cross-application leakage. This is similar to CPU cache attacks, where the cache levels closer to the CPU core are often private to a single core, making them a less powerful attack target.

Therefore, we must bypass the browser's DNS cache to ensure that the victim's DNS requests are forwarded to the system-wide DNS resolver. We achieve this by filling the cache with attacker-controlled entries. Since Chromium's DNS cache is limited to 1000 entries by default [7], we insert 1000 attacker-controlled domains to occupy it fully. Each entry is assigned a long TTL, ensuring the browser evicts target domains before attacker-controlled ones. As a result, subsequent DNS requests for target domains are forwarded to the system-wide DNS resolver. Because the browser will cache these target domains again after resolution, we repeat the eviction step before each measurement. As we show in Section IV-C, an attacker can fill the browser's DNS cache using only CSS, enabling cache eviction even in a fully scriptless scenario. During a measurement with multiple target domains, the long TTLs of the attacker-controlled entries ensure that genuine entries are evicted first, causing accesses to the system-wide DNS cache each time.

F. Summary

In Table III, we summarize the available eviction strategies for `systemd-resolved`. The first primitive, direct flushing, is only available in native code, when the attacker is able to send signals or execute the `resolvectl` command. The second primitive relies on individual DNS requests combined with Hole-Punching, which is available in all contexts and independent of system configuration. However, this is the slowest of the available strategies. The third primitive employs large DNS responses containing many entries, combined with a small number of priming requests. This primitive is the fastest, but requires a DNS server that allows for large responses, which is not the case for all public DNS servers. The last primitive triggers a DNS cache flush by exploiting a fallback mechanism in `systemd-resolved`. This primitive is available in all contexts, but requires a fallback DNS server to be configured. It only requires a single DNS request, and evicts the entire cache in a single step. However, this strategy relies on implementation-specific behavior of `systemd-resolved`, which might change with future versions.

VI. CROSS-VM END-TO-END ATTACK

In this section, we evaluate the end-to-end attack performance in a cross-VM scenario, *i.e.*, tracking the host's DNS cache from an unprivileged attacker inside a virtual machine. For this attack, we combine the native code execution measurement primitive from Section IV with the error-based eviction primitive from Section V.

A. Setup

The host system, representing the victim, runs Ubuntu 24.04 LTS, with `systemd-resolved` version `225.4-1ubuntu8.8` and has DNSSEC turned off. On this system, we set up a `libvirt` virtual machine using `virt-manager`, representing the attacker, with a default configuration, running Debian 12. In the virtual machine, we execute unprivileged native code to monitor the DNS cache of the host system. The attacker and the victim are independent of each other, and do not directly communicate.

The host uses a consumer-grade Internet connection, and the home router advertises itself as the DNS server with two IPv6 fallbacks. By default, `libvirt` uses `dnsmasq` for internal DNS resolution. Queries that cannot be fulfilled by the `dnsmasq` cache are forwarded to the host's DNS server. Despite this additional layer of caching, we found minimal interferences with our measurements as most entries are evicted anyway between the measurements. Additionally, `dnsmasq` cache hits are significantly faster than hits in the host's DNS cache, allowing us to distinguish the two caches. For the end-to-end attack, we execute unprivileged native code in the attacker's virtual machine, and due to our resolver's configuration use error-based eviction to evict the system's cache. To synchronize the ground truth with the attacker's measurements, we use time-slicing based on the system clock.

Victim. The victim has a list of 103 domains, including the top 100 websites from the Alexa Top 1M list [1], **Domain A**, **Domain B**, and `asdf.com`. At the beginning of the victim's timeslice, the victim randomly chooses 8 domains, resolves

TABLE IV
CROSS-VM END-TO-END ATTACK RESULTS

True Positives	False Negatives
22 999	3 502
False Positives	**True Negatives**
240	314 498

(F_1 Score 92.48%)

them in a random order, and saves the results to a file with a timestamp. The victim then waits for the next timeslice.

Attacker. Inside the virtual machine, the attacker uses native DNS resolution APIs in Python due to local, unpriviledged code execution. At the beginning of the attacker's timeslice, it measures the execution time of `socket.gethostbyname()` for all of the 103 domains. Therefore, we used ten threads in parallel to reduce the measurement's runtime. The attacker then uses thresholds to distinguish between `dnsmasq` hits, host DNS cache hits, and cache misses. We use a threshold of 2 ms to distinguish `dnsmasq` hits from host DNS cache hits, and a threshold of 15 ms to distinguish host DNS cache hits from misses. These thresholds are empirically determined in a few minutes, and might differ from host to host depending on system configuration and performance. Finally, the attacker evicts the host's entire DNS cache using error-based eviction. The detected domain names are later compared to the ground truth.

B. Results

We measured 3 313 timeslices, each lasting 20 s. In each timeslice, the victim resolved 8 domains, resulting in 26 464 total resolved domains. The attacker measures all 103 domains in each timeslice, resulting in 341 239 total measurements.

Measurement Reliability. Table IV summarizes the results of our end-to-end attack. The F_1 score of 92.48 % emphasizes that our attack reliably detects domains in the victim's DNS cache. Minimizing false positive, we classified hits in the `dnsmasq` cache as misses. The remaining 240 false positives might thus be misclassified `dnsmasq` hits. Our measurement did not consider latency fluctuations (cf. Figure 4), but more frequent calibration by the attacker might improve the side channel's reliability. Beyond that, the attacker might combine multiple measurements due to the short measurement interval.

Measurement Speed. To keep our blind spot (*i.e.*, the time between the start of a measurement and eviction when website accesses can be missed) small, we aim to measure all target domains as fast as possible. In our experiment, the attacker measured all 103 domains on average in 539.78 ms using 10 parallel threads. Upon receipt of large responses, the DNS server as configured by our connection's ISP does not respond. After calling `socket.gethostbyname()`, error-based eviction consequently takes 5 s to evict the DNS cache. Our ISP provides two fallback DNS servers. As `gethostbyname` is blocking, the call takes a total of 15 s to return in our test setup and takes the major part of the attacker's timeslice. This would be different with DNS servers responding with an error (e.g., Google DNS, CloudFlare, Quad9). Then, eviction would only take 79.8 ms. Summarizing, the time required for an iteration, consisting of measurement and eviction, is 5.5 s for our specific case, and 0.58 s in a generic case. In an optimized implementation, the attacker could evict the cache asynchronously to increase the granularity of the measurements, as the remaining 10 s of the function call are only waited for the resolver to fail. Consequently, the attacker could send the eviction query already before starting the measurement, timing it such that the eviction query times out after the measurement is complete.

Measurement Interval. The configured DNS server caused slow timeouts in `gethostbyname`. Consequently, we set the timeslices in our experiment to 20 s with a blind spot of 5 s. Thus, we measure the victim's online activity three times per minute, sufficient for most website monitoring attacks. Beyond that, attackers can vary the timeslice, either to increase temporal granularity or decrease the blind spot's relative size.

Our experiment shows that, even in a non-ideal default setup, DMT reliably monitors web activity of the host from a virtual machine. The same attack can also be performed between two virtual machines, jointly using the host DNS cache. DMT leaks sensitive information about the victim's web activity, even when the victim is taking measures to protect their privacy, such as only allowing third-party code in unprivileged mode in a virtual machine.

VII. JAVASCRIPT ACCESS DETECTION

In this section, we evaluate the end-to-end attack performance using JavaScript, thereby simulating an attacker that can monitor the victim's accesses continuously. This experiment combines the `fetch` JavaScript measurement presented in Section IV with the error-based primitive and the browser DNS cache bypass from Section V.

Setup. The setup consists of a host system representing the victim, which is connected via Ethernet cable to a consumer-grade connection, and an attacker using the same infrastructure described above. The victim's device runs Ubuntu 24.04.2 LTS, with `systemd-resolved` version `255.4-1ubuntu8.10`, and Chromium version 139.0.7258.5 instrumented with Playwright. The attacker deploys the custom DNS server presented in Section V and a web server that hosts a malicious website. The JavaScript code of the malicious website periodically measures the loading delay of monitored websites to detect whether the victim has accessed them. At the start of a measurement cycle, we perform an HTTP `HEAD` request for each target website using JavaScript's `fetch` API and measure its duration. We append a random string at the end of each URL to bypass any browser caching. Furthermore, we set the `cache` option of `fetch` to `no-store` to prevent the browser from using cached HTTP resources. Each request is then repeated with a different random string to measure the timing of a cache hit. For every website, we store a list of the 10 most recent cache-hit measurements, replacing the oldest one after each cycle. This approach enables us to detect website accesses despite potential network fluctuations over long measurements

TABLE V
JAVASCRIPT END-TO-END ATTACK RESULTS

Domain	DNSSEC ✗	DNSSEC ✓
amazon.com	87.39 %	98.08 %
pornhub.com	69.44 %	93.20 %
reddit.com	97.96 %	96.55 %
Macro-average	84.93 %	95.94 %

periods. To detect a cache hit, we add a website-specific offset to the median of the 10 measurements, and use the resulting value as a threshold to distinguish cache hits from misses. The adversary derives the thresholds from the hit-miss histograms of each website in an offline phase preceding the attack.

After the measurement phase, the attacker evicts the browser cache by loading 4000 domains—resolved by the attacker's DNS server. Then, they clear the `systemd-resolved` cache using the error-based primitive presented in Section V. After 5 s, the measurement cycle is repeated, enabling the attacker to monitor the victim's activity continuously. The total duration of one cycle is approximately 25 s.

Evaluation. We evaluate the attack in two scenarios: with and without DNSSEC, testing the access-detection rates for three domains: `amazon.com`, `pornhub.com`, and `reddit.com`. We run a script on the victim's device that loads the attacker's website inside Chromium. For each measurement cycle, the script selects a random subset of the domains and loads them inside a new browser tab, simulating user accesses. Then, the attacker starts the measurement cycle. We perform a total of 100 measurement cycles for each scenario and finally compute the F_1 scores for each website.

Results. We summarize our results in Table V. While DNSSEC is supposed to increase the security of traditional DNS, our findings indicate that its presence drastically accentuates the effect of DMT, potentially posing a threat to user privacy. Indeed, for all tested domains, DNSSEC leads to F_1 scores above 93 %, with `amazon.com` reaching a high accuracy of 98.08 %. Traditional DNS requests are, on average, faster than requests with DNSSEC, and hence, lower the gap between cache hits and misses, causing more frequent misclassifications. In particular, the accuracy for detecting accesses to `amazon.com` drops to 87.39 %, when DNSSEC is turned off. Because we cannot resolve a domain name directly and instead have to time the `fetch` request, the accuracy of the end-to-end attack also depends on the performance of the servers where the domains are hosted. This effect is evident when comparing the F_1 scores of `reddit.com` with the ones of `pornhub.com`. For the former, we achieve accuracies above 96 % in both scenarios, showing consistent response timings, while for the latter the F_1 score increases from 69.44 % to 93.20 %, when enabling DNSSEC. Our findings demonstrate that DMT is effective in an end-to-end setting without native code execution and represents a significant threat to user privacy, especially when DNSSEC is active.

VIII. DISCUSSION

DMT's measurement primitives can be used to continuously monitor a victim's Internet activity with a high level of accuracy. In particular our scriptless remote attack without JavaScript, demonstrates that DMT is a practical threat even for security-conscious users. DMT exploits intended behavior: The DNS cache is designed to speed up DNS resolution, and thus intentionally introduces timing differences. Thus, mitigating DMT will always be a tradeoff between performance and privacy where any miss for privacy reasons will introduce an additional latency for the user that is currently avoided. Given the numerous works on the security of caches in other contexts, future work may also investigate which cache security mechanisms also apply to the DNS cache.

Practical Implications. DMT can be used to monitor user behavior from sandboxes, virtual machines, and the browser. An important aspect of DMT is that it exploits a **local cache** to infer information about Internet usage. Network contention, such as large downloads, increase the timing differences between cached and uncached DNS resolutions, making the attack more reliable from a native code context. In the browser, we need to perform a network request to trigger a DNS resolution, which adds the noise caused by the network contention to the measurement. Still, we show in Section IV-D that DMT works slightly better on slower network connections, as the timing differences between cached and uncached resolutions are larger. Features like DNSSEC, DNS-over-TLS, or VPNs do not mitigate DMT, as they are designed to protect the data in transit, not the local cache. Instead, the increased latencies caused by such features also improve the reliability of DMT. Currently, to defend against DMT, users can only disable the DNS cache entirely, which comes at a significant performance penalty for each request and more significant privacy concerns as well: Every DNS resolution will go to a remote system (e.g., router, ISP), that can monitor a user's Internet accesses more precisely then. Furthermore, DNS caches on these remote systems (e.g., router, ISP) are more likely shared across multiple users, or even public, and thus can be used for attacks again. While most modern browsers implement their own DNS cache, attackers can measure the latency of this cache as well, and thus can still monitor the victim's behavior inside a browser instance.

IX. CONCLUSION

In this paper, we presented DMT, an Evict+Reload-style attack that infers user behavior by monitoring the local DNS cache state using timing side channels. We demonstrated that the timing differences between cached and uncached DNS resolutions can be measured from native code, JavaScript, and even scriptless HTML. We also presented four primitives that enable attackers to evict the system DNS cache from different execution contexts, and found primitives to evict the browser DNS cache as well. We demonstrated that DMT can be used in an end-to-end attack to continuously monitor user behavior, even across applications and some virtual machine configurations, with a relatively small blind spot of 0.26 % in

the best case, and $4.92\,\%$ in the worst case, when measuring with a granularity of five minutes. Website access monitoring attacks using DMT are highly reliable, with an F_1 score of $92.48\,\%$ in a native cross-VM setup, and F_1 scores of $95.94\,\%$ and $84.93\,\%$ in an end-to-end JavaScript scenario with and without DNSSEC, respectively. We discussed currently possible mitigations and their tradeoffs. Finally, we discussed the implications of DMT, from scams to extortion campaigns, to serving exploits tailored to the victim's environment, such as their EDR solution or installed applications, before the web page has finished loading.

Ethics Considerations

Except for the experiment on a private Internet connection in Section IV-D, all of our long-running experiments were performed on virtual private servers in a data center, to not impact any end users. All servers were rented specifically for this experiment, and do not contain any user data. Custom DNS servers were shut down after the experiments, and only respond to requests containing special input options, to avoid being used for DNS reflection attacks. Experiments that do not require timing (e.g., for eviction tests) were performed on a private network, with a locally-running DNS server. All measurements with requests to third-party servers were performed with long delays between iterations to avoid overloading the servers. While the measurement websites were hosted on public IP addresses, they do not collect any user data except DNS resolution timings, thus, even if a user accessed our measurement website unintentionally, their DNS measurements cannot be linked to their identity.

References

[1] Alexa Internet, Inc., "The top 1 million sites on the web," 5 2023. [Online]. Available: https://www.alexa.com/topsites

[2] G. Alexander and J. R. Crandall, "Off-path round trip time measurement via TCP/IP side channels," in *INFOCOM*, 2015.

[3] M. Andrews, "RFC 2308: Negative Caching of DNS Queries (DNS NCACHE)," 1998.

[4] N. Apthorpe, D. Reisman, S. Sundaresan, A. Narayanan, and N. Feamster, "Spying on the smart home: Privacy attacks and defenses on encrypted iot traffic," *arXiv:1708.05044*, 2017.

[5] O. Arana, H. Benitez-Perez, J. Gomez, and M. Lopez-Guerrero, "Never Query Alone: A distributed strategy to protect Internet users from DNS fingerprinting attacks," *Computer Networks*, vol. 199, no. 5, 2021.

[6] ArchWiki, "Domain name resolution," 2025. [Online]. Available: https://wiki.archlinux.org/title/Domain_name_resolution

[7] T. C. Authors, "resolve_context.cc," 2025. [Online]. Available: https://github.com/chromium/chromium/blob/main/net/dns/resolve_context.cc

[8] A. Bahramali, A. Bozorgi, and A. Houmansadr, "Realistic Website Fingerprinting By Augmenting Network Traces," in *CCS*, 2023.

[9] D. J. Bernstein, "Cache-Timing Attacks on AES," 2005. [Online]. Available: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[10] S. Bhat, D. Lu, A. Kwon, and S. Devadas, "Var-CNN: A Data-Efficient Website Fingerprinting Attack Based on Deep Learning," *PoPETS*, vol. 4, pp. 292–310, 2019.

[11] G. D. Bissias, M. Liberatore, D. Jensen, and B. N. Levine, "Privacy Vulnerabilities in Encrypted HTTP Streams," in *PET*, 2006.

[12] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.

[13] J. Bushart and C. Rossow, "Padding Ain't Enough: Assessing the Privacy Guarantees of Encrypted DNS," in *USENIX FOCI*, 2020.

[14] T. Callahan, M. Allman, and M. Rabinovich, "On Modern DNS Behvior and Properties," *Computer Communication Review*, vol. 43, no. 3, pp. 8–15, 2013.

[15] M. Catanzaro and Z. Jędrzejewski-Szmek, "Changes/systemd-resolved," 2021. [Online]. Available: https://fedoraproject.org/wiki/Changes/systemd-resolved#Release_Notes

[16] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical study of timing channels on seL4," in *CCS*, 2014.

[17] D. Crocker, T. Hansen, and M. Kucherawy, "RFC 6376: DomainKeys Identified Mail (DKIM) Signatures," 2011.

[18] S. A. Crosby, D. S. Wallach, and R. H. Riedi, "Opportunities and limits of remote timing attacks," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 3, p. 17, 2009.

[19] D. P. B. Danzig, A. Kumar, S. Miller, D. C. Neuman, and D. J. Postel, "RFC 1536: Common DNS Implementation Errors and Suggested Fixes," 1993.

[20] X. Deng, Q. Yin, Z. Liu, X. Zhao, Q. Li, M. Xu, K. Xu, and J. Wu, "Robust Multi-tab Website Fingerprinting Attacks in the Wild," in *S&P*, 2023.

[21] J. Dickinson, S. Dickinson, R. Bellis, A. Mankin, and D. Wessels, "RFC 7766: DNS Transport over TCP - Implementation Requirements," 2016. [Online]. Available: https://www.rfc-editor.org/info/rfc7766

[22] T. V. Doan, J. Fries, and V. Bajpai, "Evaluating Public DNS Services in the Wake of Increasing Centralization of DNS," in *IFIP Networking*, 2021.

[23] R. Dubin, A. Dvir, O. Pele, and O. Hadar, "I Know What You Saw Last Minute—Encrypted HTTP Adaptive Video Streaming Title Classification," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 12, pp. 3039–3049, 2017.

[24] E. W. Felten and M. A. Schneider, "Timing attacks on web privacy," in *CCS*, 2000.

[25] H. Gao, V. Yegneswaran, Y. Chen, P. Porras, S. Ghosh, J. Jiang, and H. Duan, "An empirical reexamination of global DNS behavior," in *SIGCOMM*, 2013.

[26] H. Gao, V. Yegneswaran, J. Jiang, Y. Chen, P. Porras, and S. Gosh, "Reexamining DNS From a Global Recursive Resolver Perspective," *IEEE Transactions on Networking*, vol. 14, no. 1, pp. 43–56, 2014.

[27] S. Gast, R. Czerny, J. Juffinger, F. Rauscher, S. Franza, and D. Gruss, "SnailLoad: Exploiting Remote Network Latency Measurements without JavaScript," in *USENIX Security*, 2024.

[28] X. Gong, N. Kiyavash, and N. Borisov, "Fingerprinting Websites Using Remote Traffic Analysis," in *CCS*, 2010.

[29] L. Grangeia, "DNS Cache Snooping or Snooping the Cache for Fun and profit," 2004. [Online]. Available: https://intranet.csc.liv.ac.uk/~coopes/comp319/2016/papers/dns_cache_snooping.pdf

[30] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA*, 2016.

[31] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.

[32] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security*, 2015.

[33] J. Hayes and G. Danezis, "k-fingerprinting: A Robust Scalable Website Fingerprinting Technique," in *USENIX Security*, 2016.

[34] D. Herrmann, R. Wendolsky, and H. Federrath, "Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naïve-Bayes Classifier," in *CCSW*, 2009.

[35] A. Hintz, "Fingerprinting Websites Using Traffic Analysis," in *PET*, 2003.

[36] N. P. Hoang, A. A. Niaki, J. Dalek, J. Knockel, P. Lin, B. Marczak, M. Crete-Nishihata, P. Gill, and M. Polychronakis, "How Great is the Great Firewall? Measuring China's DNS Censorship," in *USENIX Security*, 2021.

[37] P. Hoffman, "RFC 9364: DNS Security Extensions (DNSSEC)," 2023.

[38] P. Hoffman and P. McManus, "RFC 8484: DNS Queries over HTTPS (DoH)," 2018.

[39] Z. Hu, L. Zhu, J. Heidemann, A. Manking, D. Wessels, and P. Hoffman, "RFC 7858: Specification for DNS over Transport Layer Security (TLS)," 2016.

[40] W. Jiang, T. Luo, T. Koch, Y. Zhang, K.-B. Ethan, and M. Calder, "Towards Identifying Networks with Internet Clients Using Public Data," in *IMC*, 2021.

[41] Z. Jin, T. Lu, S. Luo, and J. Shang, "Transformer-based Model for Multi-tab Website Fingerprinting Attack," in *CCS*, 2023.

[42] L. Jinjin, J. Jiang, H. Duan, K. Li, and J. Wu, "Measuring Query Latency of Top Level DNS Servers," in *PAM*, 2013.

[43] M. Jonker, A. Sperotto, R. van Rijswijk-Deij, R. Sadre, and A. Prais, "Measuring the Adoption of DDoS Protection Services," in *IMC*, 2024.

[44] S. Kadloor, X. Gong, T. Tezcan, and N. Borisov, "Low-Cost Side Channel Remote Traffic Analysis Attack in Packet Networks," in *IEEE ICC*, 2010.

[45] S. Kitterman, "RFC 7208: Sender Policy Framework (SPF) for Authorizing Use of Somains in Email, Version 1," 2014.

[46] A. Klein and B. Pinkas, "DNS Cache-Based User Tracking," in *NDSS*, 2019.

[47] P. Kocher, "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems," in *CRYPTO*, 1996.

[48] M. Kucherawy and E. Zwicky, "RFC 7490: Domain-based Message Authentication, Reporting, and Conformance (DMARC)," 2015.

[49] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, "NetCAT: Practical Cache Attacks from the Network," in *S&P*, 5 2020.

[50] J. Li, Z. Lin, X. Ma, J. Li, J. Qu, X. Luo, and X. Guan, "DNSScope: Fine-Grained DNS Cache Probing for Remote Network Activity Characterization," in *INFOCOM*, 2024.

[51] J. Li, X. Ma, J. Tao, and X. Guan, "Boosting practicality of DNS cache probing: A general estimator based on Bayesian forecasting," in *ICC*, 2013.

[52] X. Li, C. Lu, B. Liu, Q. Zhang, Z. Li, H. Duan, and Q. Li, "The Maginot Line: Attacking the Boundary of DNS Caching Protection," in *Usenix Security*, 2023.

[53] M. Liberato, A. Affinito, B. Meijerink, M. Jonker, A. Botta, and A. Sperotto, "To Block Or Not To Block? Evaluating Parental Controls Across Routers, DNS Services, and Software," in *TMA*, 2025.

[54] C.-H. Lin, L. Shan-Hsin, Huang-Hsiu-Chuan, C.-W. Wang, C.-W. Hsu, and S. Shieh, "DT-Track: Using DNS-Time Side Channel for Mobile User Tracking," in *DSC*, 2019.

[55] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.

[56] X. Ma, J. Zhang, J. Tao, J. Li, J. Tian, and X. Guan, "DNSRadar: Outsourcing Malicious Domain Detection Based on Distributed Cache-Footprints," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 11, pp. 1906–1921, 2014.

[57] X. Ma, J. Li, J. Tao, and X. Guan, "Towards active measurement for DNS query behavior of botnets," in *GLOBECOM*, 2012.

[58] K. Man, Z. Quian, Z. Wang, X. Zheng, Y. Huang, and H. Duan, "DNS Cache Poisoning Attack Reloaded: Revolutions with Side Channels," in *CCS*, 2020.

[59] K. Man, X. Zhou, and Z. Quian, "DNS Cache Poisoning Attack: Resurrections with Side Channels," in *CCS*, 2021.

[60] C. Maurice, N. Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters," in *RAID*, 2015.

[61] P. Mittal, A. Khurshid, J. Juen, M. Caesar, and N. Borisov, "Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting," in *CCS*, 2011.

[62] D. Mo, Y. Zhu, Z. Jie, Y. Sun, Q. Liu, and B. Fang, "Unveiling Flawed Cache Structures in DNS Infrastructure via Record Watermarkings," in *GLOBECOM*, 2023.

[63] G. Moav, Y. Afek, A. Bremler-Barr, and A. Klein, "DNS FLaRE: A Flush-Reload Attack on DNS Forwarders," in *USENIX Security*, 2025.

[64] P. V. Mockapetris, "RFC 1034: Domain names-concepts and facilities," 1987.

[65] ——, "RFC 1035: Domain names-implementation and specification," 1987.

[66] G. Moura, J. Heidemann, R. Schmidt, and W. Hardaker, "Cache Me If You Can: Effects of DNS Time-to-Live," in *IMC*, 2019.

[67] N. Msadek, R. Soua, and T. Engel, "IoT device fingerprinting: Machine learning based encrypted traffic analysis," in *Wireless Communications and Networking Conference (WCNC)*, 2019.

[68] S. J. Murdoch and G. Danezis, "Low-cost traffic analysis of Tor," in *S&P*, 2005.

[69] A. Niaki, W. Marczark, S. Farhoodi, A. McGregor, P. Gill, and N. Weaver, "Cache Me Outside: A New Look at DNS Cache Probing," in *PAM*, 2021.

[70] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications," in *CCS*, 2015.

[71] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *CT-RSA*, 2006.

[72] D. Page, "Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel," *Cryptology ePrint Archive, Report 2002/169*, 2002.

[73] A. Panchenko, F. Lanze, J. Pennekamp, T. Engel, A. Zinnen, M. Henze, and K. Wehrle, "Website Fingerprinting at Internet Scale," in *NDSS*, 2016.

[74] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website Fingerprinting in Onion Routing Based Anonymization Networks," in *WPES*, 2011.

[75] M. A. Rajab, F. Monrose, A. Terzis, and N. Provos, "Peeking through the cloud: DNS-based estimation and its application," in *ACNS*, 2008.

[76] A. Randall, E. Liu, G. Akiwate, R. Padmanabhan, G. M. Voelker, S. Savage, and A. Schulman, "Trufflehunter: Cache Snooping Rare Domains at Large Public DNS Resolvers," in *IMC*, 2020.

[77] A. Randall, E. Liu, R. Padmanabhan, G. Akiwate, G. M. Voelker, S. Savage, and A. Schulman, "Home is Where the Hijacking is: Understanding DNS Interception by Residential Routers," in *IMC*, 2021.

[78] V. Rimmer, D. Preuveneers, M. Juarez, T. Van Goethem, and W. Joosen, "Automated website fingerprinting through deep learning," in *NDSS*, 2017.

[79] E. Rodriguez, R. Anghel, S. Parkin, M. van Eeten, and C. Ganan, "Two Sides of the Shield: Understanding Protective DNS adoption factors," in *USENIX Security*, 2023.

[80] S. Saha, S. Karapoola, C. Rebeiro, and K. V, "YODA: Covert Communication Channel over Public DNS Resolvers," in *DSN*, 2023.

[81] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," in *ESORICS*, 2019.

[82] M. Shen, Z. Gao, L. Zhu, and K. Xu, "Efficient fine-grained website fingerprinting via encrypted traffic analysis with deep learning," in *International Symposium on Quality of Service (IWQOS)*, 2021.

[83] M. Shen, J. Zhang, K. Xu, L. Zhu, J. Liu, and X. Du, "Deepqoe: Real-time measurement of video qoe from encrypted traffic with deep learning," in *International Symposium on Quality of Service (IWQoS)*, 2020.

[84] M. Shen, J. Zhang, L. Zhu, K. Xu, and X. Du, "Accurate decentralized application identification via encrypted traffic analysis using graph neural networks," *TIFS*, vol. 16, pp. 2367–2380, 2021.

[85] P. Sirinam, M. Imani, M. Juarez, and M. Wright, "Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning," in *CCS*, 2018.

[86] P. Sirinam, N. Mathews, M. Rahman, and M. Wright, "Triplet Fingerprinting: More Practical and Portable Website Fingerprinting with N-shot Learning," in *CCS*, 2019.

[87] M. Skowron, A. Janicki, and W. Mazurczyk, "Traffic fingerprinting attacks on internet of things using machine learning," *IEEE Access*, vol. 8, pp. 20 386–20 400, 2020.

[88] R. Sommese, K. Claffy, R. van Rijswijk-Deij, A. Chattopadhyay, A. Dainotti, A. Sperotto, and M. Jonker, "Investigating the impact of DDoS attacks on DNS infrastructure," in *IMC*, 2022.

[89] Speedtest, "Speedtest CLI," 2025. [Online]. Available: https://www.speedtest.net/apps/cli

[90] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Robust smartphone app identification via encrypted network traffic analysis," *TIFS*, 2017.

[91] L. Trampert, D. Weber, L. Gerlach, C. Rossow, and M. Schwarz, "Cascading Spy Sheets: Exploiting the Complexity of Modern CSS for Email and Browser Fingerprinting," in *NDSS*, 2025.

[92] T. Van Goethem, C. Pöpper, W. Joosen, and M. Vanhoef, "Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections," in *USENIX Security*, 2020.

[93] T. Wang and I. Goldberg, "Improved Website Fingerprinting on Tor," in *WPES*, 2013.

[94] D. Wessels, W. Carroll, and M. Thomas, "RFC 9520: Negative Caching of DNS Resolution Failures," 2023.

[95] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security*, 2014.